UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MACHINE LEARNING FOR IMPACT-BASED FLASH FLOOD WARNINGS:

HAZARD REPORT OPERATIONALIZATION FOR IMPACT PREDICTIONS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By
Jorge A. Duarte García
Norman, Oklahoma, USA

2023

MACHINE LEARNING FOR IMPACT-BASED FLASH FLOOD WARNINGS:

HAZARD REPORT OPERATIONALIZATION FOR IMPACT PREDICTIONS

A DISSERTATION APPROVED FOR THE

SCHOOL OF INDUSTRIAL AND SYSTEMS ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Charles D. Nicholson, Chair

Dr. Jonathan J. Gourley, Co-Chair

Dr. Pierre E. Kirstetter

Dr. Andrés D. González

Dr. Randa L. Shehab

Acknowledgements

The author would like to thank and recognize the following people:

- Dr. Justyna Adamiak, thank you for your advice, persistence, strength, caring, patience, and unwavering support through this journey; I am absolutely sure that I wouldn't have made it this far without you by my side. Kocham cię z całego serca.
- Dr. Maciej Adamiak, the most interesting and relevant ideas explored in this work can be traced directly back to conversations we had. Your help during the most critical phases of the development of this dissertation can not easily be repaid. Dziękuję bardzo.
- My Family, it has been only through the continued efforts you have made and support you have provided, that the path I've walked has brought me here. Thank you for so much.
- Dr. J.J. Gourley, Dr. H. Vergara, and Dr. P. Kirstetter: words are not enough to express my gratitude towards the kindness, trust, and support you have deposited in me throughout the last seven years of my life. Thank you for providing me with life-changing opportunities, and enabling me to grow.
- Dr. Charles Nicholson, thank you for working with me along this whole endeavor, and venturing beyond the realms of convention. Your patience, support, and commitment are unparalleled.
- L. B. von Mustaszo, thank you for always giving so much, while expecting so little in return.
- My Friends, this journey would not have been the same, nor as meaningful, without you along the way.

Abstract

Floods account for approximately one third of all global geophysical hazards, and flash floods allow for extremely short lead times for warnings to be emitted. Flash flood warnings are weather-related alerts which serve to inform of potential hazardous conditions which threaten life or property. The National Weather Service has transitioned to an impact-based format for flash flood warnings, which aims to provide additional valuable information about hazards, that facilitate improved public response and decision making. This work responds to the need for new decision support tools, which enable forecasters to anticipate distinct levels of impacts associated with flash flood forecasts, and provide support for issuing impact-based flash flood warnings.

This dissertation proposes a foundation over which said decision support systems can be built. First and foremost, by constituting an unprecedented data set of historical flash flood reports with associated impact categories, achieved by the systematic application of a language-based impact framework (Flash Flood Severity Index) as a natural language processing task piped through a large language model (OpenAI's GPT-3.5-turbo). Secondly, through a proof-of-concept machine learning model trained to predict the severity of flash flood forecasts, based on operational flash flood forecasts, geomorphological data, and vulnerability layers.

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Preface

As part of the University of Oklahoma's (OU) Cooperative Institute for Severe and High-Impact Weather Research and Operations (CIWRO), I have worked with the National Oceanic and Atmospheric Administration's (NOAA) National Severe Storms Laboratory (NSSL), where I conduct collaborative research on various topics related to hydrology and hydrometeorology as part of the Warning Research and Development Division's (WRDD) Stormscale Hydrology Group. One of WRDD's main functions is to design, test and transition new warning and decision-making tools and technologies to the National Weather Service (NWS). Within this collaborative context, surrounded by expert hydrologists, meteorologists and engineers, I've had the opportunity of working on flash flood and debris flow characterization, modeling, monitoring and alerting projects. The present study is an example of this kind of research, which is ultimately aimed towards innovating and/or improving existing tools and technologies, which enable relevant actors and organizations to better understand complex processes, make better decisions and issue warnings in the face of natural hazards.

This research was funded by the NOAA/OAR - Joint Technology Transfer Initiative (JTTI) program under Federal Award No. NA20OAR4590354, U.S. Department of Commerce, National Oceanic and Atmospheric Administration.

Chapter 1

Introduction

Floods are one of the most ubiquitous and devastating natural hazards, and they account for approximately one third of all global geophysical hazards. Flash floods are types of floods that allow for extremely short lead times for warnings to be emitted [1]. In the United States, during 2014 alone, over $2.8 billion dollars of direct flood damages occurred, and 55 flood-related deaths were recorded (39 were flash flood-related) [2]. The National Weather Service (NWS) is the organization in charge of the monitoring and emission of flash flood warning related alerts (watches and warnings) in the United States, and these play a crucial role in the protection of life and property linked to the occurrence of sever weather events.

As part of the National Oceanic and Atmospheric Administration (NOAA), the National Severe Storms Laboratory's (NSSL) Warning Research and Development Division (WRDD) aims to *develop new weather and water related applications and water resource management tools help NWS forecasters produce more accurate and timely warnings of flood events* [3]. Contributing to NSSL's mission, the Flooded Locations and Simulated Hydrographs (FLASH) project [4] provides a collection of flash flood related products, which were developed at WRDD, thoroughly tested by forecasters and scientists (Hydrometeorological Testbed Hydro Experiments of 2018 and 2019), and successfully transitioned to operations as part of the Multi-Radar Multi-Sensor (MRMS) [5] suite of products. Both MRMS and FLASH produce 1km gridded outputs over the entire United States (US), including the conterminous US (CONUS), and other outside territories including Alaska, Hawaii, Guam and Puerto Rico.

Since 2016, NOAA's Weather Program Office's (WPO) aims to ensure the continuous development and transition of the latest scientific and technological advances into the NWS, through the Joint Technology Transfer Initiative (JTTI) program [6]. Projects funded by the JTTI program are supported by NOAA to develop, test and evaluate matured weather research that can potentially transition to operations. The present line of work stems from the objectives and needs established by a JTTI project proposal titled *Products to Guide Impact-Based Flash Flood Warnings in the National Weather Service*. This project's overarching goal is to *develop, evaluate and transition products to the NWS that will guide forecasters in the selection of flash flood damage threat tags*, to be included within flash flood warnings.

Having established a brief overview of the work being addressed, the remainder of this chapter introduces concepts and terms pertaining to relevant areas in the following way. Section 1.1 lays out the overarching significance of the present study, in light of clear research gaps and problems to be addressed; section 1.2 presents a comprehensive taxonomy of machine learning systems, as well as a few relevant deep learning applications, architectures and models that enabled the development of the present work; section 1.3 introduces concepts in hydrology, floods, and flash flood forecasting; lastly, section 1.4 defines the core problems and research questions addressed by this work, as well as the main objectives and milestones to be fulfilled by this dissertation work.

1.1 Significance of the Study

The present work is significant and novel in several ways. There exists a research gap which stems from the context out of which this research is performed (NOAA National Severe Storms Laboratory's Warning Research and Development Division, Stormscale Hydrology research group), for providing forecasters with decision support tools that enable them to anticipate distinct impact levels associated with flash flood forecasts. In order to construct these tools, data is required. While historical reports of flash flood events are collected and maintained by the National Weather Service, these are not operational in nature. This means that, a comprehensive dataset of historical flash flood observations or reports, which contain discrete and concise levels or classes of impacts associated with these events does not exist.

The present dissertation sheds light on the systematic *operationalization* of historical flash flood reports, through novel machine learning methods and applications. This is done with hopes of tracing a feasible path towards the development and implementation of experimental (and one day operational) models for predicting impacts associated to operational flash flood forecasts. In the process of developing this body of work, the first systematically classified flash flood impact dataset has been created, and the performance of this classification is assessed with respect to that of experts which contributed a small subset of human-classified reports. Additionally, a reduced proof-of-concept model which makes use of this novel dataset is proposed, which shows preliminary yet favorable results, which can guide future development of flash flood impact annotation models at a national scale over the United States.

1.2 Machine Learning Systems

Machine learning (ML), a branch of artificial intelligence (AI), is the field of study that revolves around enabling computers to learn without being explicitly programmed with new knowledge [7]. While the focus of AI is to make machines intelligent (*i.e.* to think rationally as humans do), ML is concerned strictly with a computer's ability to learn from past experience [8]. Just as humans learn from experience, learning can occur in a computer program when the program is able to improve its own performance on a specific task, after being exposed to a specific set of data. Then, we could say ML is the science of programming computers in a way that allows them to learn from data, as a means to improve their performance on a given task.

A system can be defined as a functional construct or collection of different elements that together produce results not obtainable by the elements alone, due to the constructive nature of the relationships and connections that exist among its parts [9]. By extending this notion into the field of ML, a ML system can be defined as a construct composed of hardware and software (both programs and data), which is able to learn how to perform specific tasks, by being somehow exposed to relevant experience pertaining to its functional purpose. The mechanism through which ML systems are exposed to experience is referred to as "training", and it is generally understood as part of the pre-operational phase of a ML system, before it is deployed to perform the task is has been trained to do. However, an exception to this occurs in the case of *unsupervised* learning systems, where algorithms are applied to and perform tasks directly on the

data.

1.2.1 A Taxonomy of Machine Learning Systems

In order to provide a clear overview of terminology, definitions, and context concerning various ML technologies, techniques and concepts, this section will introduce various perspectives through which ML systems can be defined, classified and understood. The following four perspectives will be presented in incremental order of specificity: problem types, types of learning, learning mechanisms, and areas of application.

Classification by ML Problem Types

In the most prevalent sense, ML systems are generally classified depending on the type of problem they solve. These problem types are linked to the conceptual description of what is to be achieved by said system (*i.e.* its practical purpose). Most problems targeted by ML systems usually fall under one of the following categories [10]:

- Classification
- Clustering
- Association
- Numeric prediction

Classification ML systems, once trained, aim to provide ways of classifying unseen examples into categories defined by the training data. **Association** ML systems intend to extract association rules from a training set's features, not just those which could predict a particular class value. These association rules describe the relationships between the data set's variables by using mechanisms and measures such as correlation and similarity, rather than constructing predictors for a specific variable (or subset of variables). In a practical sense, association rules allow for the enunciation of relationships that hold true across the whole dataset. For example, contemplating a hypothetical and simple weather dataset on `weather outlook`, `temperature`, and `humidity`, one such association rule could be that whenever:

$$\texttt{temp} = \texttt{hot} \rightarrow [\texttt{outlook} = \texttt{sunny} \vee (\texttt{outlook} = \texttt{overcast} \wedge \texttt{humidity} = \texttt{high})]$$

which means that based on the data provided, whenever `temperature` was recorded

to be `hot`, either: a) the `outlook` was `sunny`, or b) the `outlook` was `overcast` **and** the `humidity` was `high`.

Clustering ML systems provide a means for grouping a collection of examples that belong together, by finding similarities within the feature space of the training data. Lastly, **numeric prediction** ML systems allow to predict a specific numeric quantities from new observations, instead of discrete classes [10].

Classification by ML Learning Types

ML systems can also be classified into different non-exclusive categories, depending on the type of learning they must accomplish in order to perform on specific tasks. These categories can be defined in terms of the following criteria [7]:

1. whether or not these systems are trained with some kind of supervision
2. whether or not they can learn incrementally on the fly
3. whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model.

The first criterion yields the different notions of **supervised**, **unsupervised**, **semi-supervised** and **reinforcement learning**. These four subcategories refer to varying degrees of supervision necessary throughout a ML system's training. The term supervision alludes to whether the data used to train the system includes specific labels –naturally available, reasonably introduced, or engineered in preparation for training a model– which tell ML algorithms if two different data points are related somehow, or not (*e.g.* class labels). Supervised ML systems are trained using labeled data. Unsupervised ML systems are applied on unlabeled data, and often times their purpose is to define or find appropriate labels for the given data set. Semi-supervised learning is a mixed approach which generally attempts to label a large unlabeled dataset, by building an intermediate model from a reduced subset of labeled data. Reinforcement learning differs greatly from the other three types of learning described herein, as it relies on the concepts of an agent within a context it can observe and learn from, depending on how its actions are rewarded (which closely resembles how humans

learn from their own experiences) [7]. An example of this would be an automaton, which learns to follow a safe path out of a tiled room, by receiving positive or negative feedback from its environment. In this case, moving over safe tiles provide a desirable reward which reinforces the automaton's movement behavior, whereas doing so over unsafe tiles provide it with negative reinforcement, incentivizing the automaton to avoid them.

The second criterion highlights the difference between **batch learning** and **online learning**. A batch learning ML system learns from all the available data during its training which is typically extensive, and it is performed as a separate phase before the system is used operationally. When launched into production, this trained ML system operates without learning from its operational data anymore (offline learning). Thus, if this model needed to know about new data, it would have to undergo training (from scratch) once more, using the full dataset (previously known + new data). Conversely for the case of online learning, a ML system is trained incrementally by feeding it individual instances or small groups (mini-batches) of data. Opposed to offline learning, each training step takes little time, and the system can learn about new data on the fly as it is consumed [7].

Lastly, the third criterion differentiates **instance-based learning** from **model-based learning**. The difference here stems from the specific way in which ML systems generalize what they learn (*i.e.* the specific mechanism that leads to adequate performance on unseen data). This categorization mostly concerns the ML task of making predictions (perhaps the most common task), as the true challenge is not only to achieve good predictions during the system's training phase, but to provide comparably good predictions on unseen and new data (operationally). When this is achieved, it can be said that the trained model generalizes well to unseen data. In instance-based learning, a ML system learns from specific data examples –or instances– (during training), which later allows it to compare unseen instances to what has been learned, using a measure of similarity (*i.e.* a measure of distance). Subsequently, through this measure of similarity, the system can make predictions on the new data (*i.e.* predict its class or value). Conversely, during model-based learning, a ML system aims to represent the underlying relationships that exist among the data is is trained on, using a mathematical/statistical model (*i.e.* linear regression, generalized linear model, non-linear regression, *etc.*). Consequently, the system relies on minimizing a cost function (*i.e.* measure of error) which yields the best parameters for the specific model (linear/non-linear, parametric/non-parametric, *etc.*) in use (training phase), which then can be

used as a mechanism to make predictions on unseen data [7].

Classification by ML Learning Mechanisms

Additional to the previous categorizations of ML systems, they can be classified by the types of algorithms, models, and methods they employ to learn from data. These characteristics are inherent to the mathematical principles, design, and implementation of each system, and are closely tied to the second and third criteria discussed above. This characterization of *learners* can also be presented as a list of areas of study within the general realm of ML [8]:

- Regression methods
- Recursive partitioning and trees
- Artificial neural networks
- Support vector machines
- Cluster analysis
- Ensemble learning
- Manifold Learning

Regression methods allude to the process of predicting a numerical quantities, based on correlations between the inputs and outputs passed to a function, which expresses the relationship between the predicted values (output or class) in terms of the input attributes. These input-output relationships can be modeled both parametrically (based on assumptions about said relationship), and non-parametrically (not based on assumptions), linearly (*i.e.* using linear relationships), or non-linearly (*i.e.* nonlinear regression). Additionally, these correlations can be contemplated between one input to one output (simple regression), multiple inputs to a single output (multiple regression), and multiple inputs to multiple outputs (multivariate regression) [8, 10].

Recursive partitioning is a non-parametric statistical method which is the basis for building decision trees [8], and this kind of tree-based method stems from the "divide-and-conquer" approach to learning from a set of independent instances. **Trees** are a logic data structure composed of nodes, which each involves the testing (comparison) of a particular attribute with respect to a specific value. Once the tree has been built (by training a model), chains of nodes can be traversed from the root of the tree to a leaf node. Said leaf nodes hold a specific output (class label, class probability or numeric value) product of following the decision rule composed of that particular node sequence [10].

Artificial neural networks (ANNs) are made up of interconnected sets of neurons, modeled after an abstraction of how human brain cells work and are interconnected. These artificial neurons consist of inputs (dendrites), a signal which travel through the neuron (activation function), and outputs (axon). The idea behind them is that, depending on the input values, the activation function produces a specific response (output) which can in turn be fed as inputs to other neuron (or neurons), and through mechanisms like iterative optimization and backpropagation, learning parameters can be adjusted for each neuron in the network. By stacking and connecting multiple layers of artificial neurons in various ways, distinct deep learning architectures are composed. These intricate and diverse architectures can be beneficial for solving specific, complex problems, and performing particular learning tasks [7, 8].

Support vector machines (SVMs) are a type of ML learning mechanism capable of performing linear and nonlinear classification, regression and outlier detection [7]. SVMs involve the optimization of constrained convex loss functions, which are not affected by problems of local minima (*i.e.* there's generally a clear gradient towards an absolute minimum). By making use of kernel methods, SVMs are particularly well-suited for building large nonlinear classifiers. The "kernel trick" allows SVMs to find optimal separating hyperplanes for high-dimensional feature spaces, by transforming the input space into a lower-dimension space by using a nonlinear kernel function, which both alleviates the curse of dimensionality and allows SVMs to compute linear classifications on this transformed kernel space [8].

The act of arranging large quantities of multivariate data into natural groups (dictated by the behavior exhibited by the data) is referred to as *clustering*. **Cluster analysis** (CA), also known as data segmentation or class discovery, consists of various algorithms and methods which seek to organize or partition a data set into homogeneous subgroups (*i.e.* clusters). CA is perhaps the most well-known example of unsupervised learning, and it allows for the analysis of unstructured multivariate data [8]. The notion of cluster is central to CA methods, but how a specific method considers a cluster varies. Clusters may be described with respect to: 1) locations of a particular points (centroids) or regions of densely packed points (*i.e.* spatially), or 2) with respect to how groups are organized (hierarchically) [7]. Thus, CA relies on measures of similarity (*e.g.* distance) to characterize the behavior of data as relatively similar or different to other data points in the set.

A group of predictors is called an ensemble, thus **ensemble learning** revolves around

the idea of leveraging the "wisdom of the crowd", by generating multiple models which are then combined to produce better results than those achieved by using a single model [7, 8]. Generally speaking, the types of models for ensemble learning tend to be weak or unstable, which in turn is the source of this method's strength. By leveraging the variance of the ensemble, more generalizable and better performing models are achieved [10].

Technically put, a *d*-dimensional manifold is part of an *n*-dimensional space ($d < n$) that locally resembles a *d*-dimensional hyperplane. In simple terms, a manifold can be defined as an lower-dimensional shape, which can be bent and twisted in a higher dimensional space (*i.e.* imagine a spiraled flat ribbon suspended in 3D space) [7]. **Manifold learning** involves several techniques and methods mainly applied towards dimensionality reduction: identifying the manifold (linear or nonlinear) which actually and most accurately represents our data within its original high-dimensional space [8]. In other words, a lower-dimensional representation of our data is sought after in order to overcome the curse of dimensionality, and enabling the construction of less-complex representations and more computationally efficient models of our data. One of the most prevalent dimensionality reduction methods is principal component analysis (PCA).

Classification by ML Applications

Lastly, there are also specific applications of ML, which have given rise to particular areas of study among the field of ML. Among these, some of the most common and relevant applications are listed below (this is by no means a comprehensive list):

- Data analytics
- Predictive analysis
- Natural language processing
- Computer vision

Data analytics is defined by the Institute for Operations Research and the Management Sciences (INFORMS) as "*the scientific process of transforming data into insight for making better decisions*" [11]. In the area and applications of data analytics, ML systems are commonly employed to aid in the decision-making process, particularly by proposing data-driven alternatives to traditional empirical approaches. **Predictive analysis** revolves around the art of making accurate guesses about new output values

that are independent of the training data [8]. This way, predictive models are built by ML systems in order to provide future predictions. Common use cases for predictive modeling include stock market and weather forecasting. **Natural language processing** (NLP) is an area of ML centered on allowing machines to read and write in natural language(*i.e.* human languages) [7]. Part of its purpose revolves around identify patterns from (unstructured) text, in order to extract information that is useful (*i.e.* actionable) for particular purposes [10]. Some of the most prominent applications for NLP include document clustering (where clusters of documents identified as similar are created between collections of unstructured text), sentiment analysis (where text documents are analyzed in order to characterize what the person who wrote it was feeling, and speech recognition (where spoken words are sought to be transcribed into text in an automated way) [7]. Over the last ten years, one of the most important tools to emerge from both of these applications is real-time speech-to-speech language translation applications. However, since the advent of Large Language Models (LLMs) a few years ago, generative and predictive applications like OpenAI's ChatGPT have taken the world by storm [12]. These LLMs dramatically expand on a model's ability not only to understand language, but to offer certain degree of reasoning capabilities, enabling them to perform complex tasks and interactions [13].

Lastly, **computer vision** (CV) is an area of applications which centers around providing machines with the ability to identify patterns in visual representations of data (*i.e.* images) [7]. Some of the most renowned applications of CV include automated object/subject identification, semantic segmentation (image labeling/tagging), and motion tracking.

* * *

Having introduced various concepts by exploring four different perspectives of ML systems, this work's main scope can now be introduced: to discuss how the used of specific ML applications like natural language processing using LLMs, and particular transformer-based deep neural network architectures, could benefit the development of a probabilistic, impact-based, flash flood warning guidance system.

It must be noted that this conceptual introduction was necessary in order to properly characterize these concepts, which are not easily comparable between one another without understanding their specific purposes and scopes. Figure 1.1 summarizes the four aspects of this conceptual taxonomy of machine learning systems.

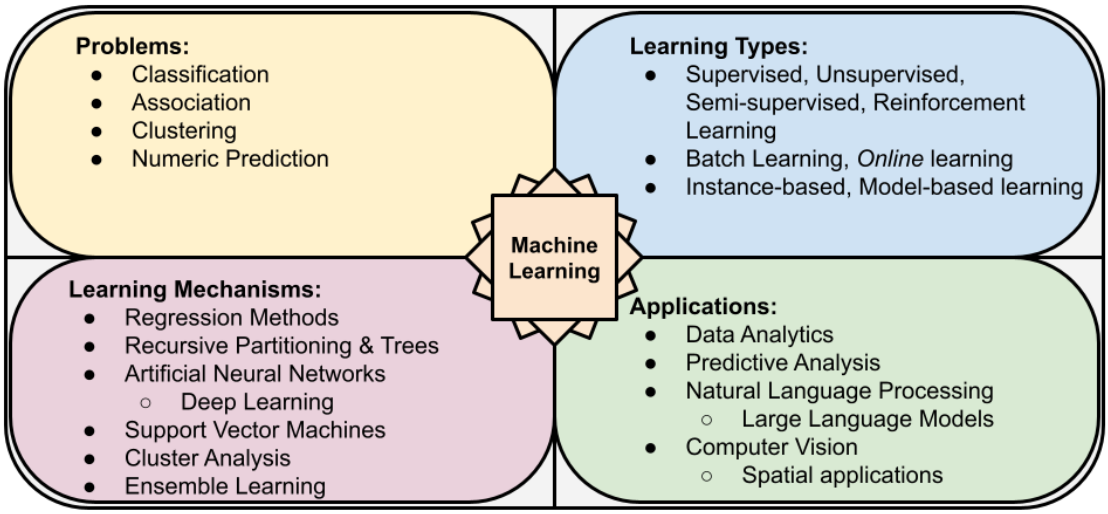
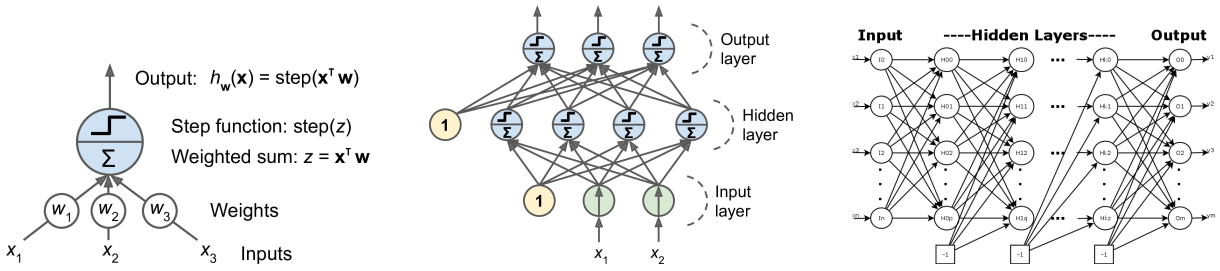


Figure 1.1: Diagram depicting the taxonomy of Machine Learning Systems, broken down by 1) ML Problems, 2) Learning Types, 3) Learning Mechanisms, and 4) ML Applications

1.2.2 Deep Learning and Artificial Neural Networks

As described in the previous section, Artificial Neural Networks (ANNs) are a specific type of learning mechanism, through which ML systems learn. Deep learning (DL) is a subset of architectures for building ANNs, which aim to resolve specific problems and provide scalable solutions to particular applications (*e.g.* prediction, regression, classification, unsupervised learning, dimensionality reduction, *etc.*).

By using a single threshold logic unit (TLU) –a specific type of artificial neuron– it is possible to perform simple binary classification. TLUs compute a weighted sum of their inputs, and then apply a step function to that weighted sum to determine what the output value should be (continuous or discrete). It must be clarified that both the inputs and outputs of an ANN are treated as layers as well. A single layer of stacked TLUs is one of the simplest ANN architectures –known as the perceptron–, where the input layer nodes are connected to every TLU in the perceptron's layer, which is then connected to the output nodes. Visual representations of TLUs and perceptrons are shown in Figure 1.2. As might be expected, more than one layer of TLUs can be used in succession between the inputs and output layers to build more complex networks like the multi-layer perceptron (MLP). For a MLP, the input layer is succeeded by a number TLU layers, known as hidden layers, before reaching the final output layer[7, 8].

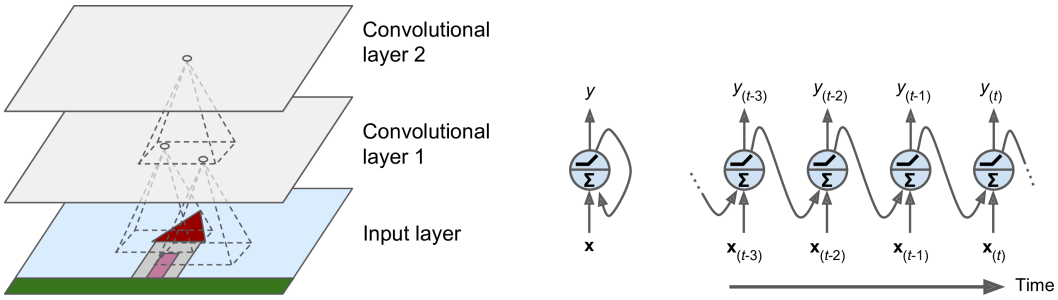


(a) TLU Neuron [7] (b) Perceptron [7] (c) Multi-layer perceptron

Figure 1.2: Examples of (a) a single threshold logic unit and (b) a perceptron comprised of a single hidden layer of TLUs. Notice that the network flow is structured from the bottom (inputs) to the top (outputs). (c) Example of a fully-connected deep multi-layer perceptron network. Note that in this case, the network flow is organized from left (inputs) to right (outputs).

As opposed to shallow neural networks (*i.e.* simple, few number of hidden layers – typically one), deep neural networks (DNNs) are constructed when large quantities of neurons are stacked into increasing numbers of hidden layers (typically two or more). Given their size (or depth), these type of networks generally require large amounts of computational power (and time) to train, and thus DNNs are said to perform perform deep learning [7]. A visual representation of a deep multi-layer perceptron can be be seen in Figure 1.2c

Two of the most prominent deep learning architectures are convolutional neural networks (CNNs) and recurrent neural networks (RNNs). CNNs are characterized by a specific type of hidden layers known as convolution layers, which allow the CNN to model and extract features from the input data. This is really useful when working with image data. RNNs rely on the concept of a recurrent neuron which operates on the basis of time steps, at which the previously generated outputs allow the neuron to produce new ones at each time. Thus, RNNs easily accommodate the notion of time-dependence in data. Figure 1.3 presents visual representation for both convolutional layers and recurrent neurons. These two ANN architectures will be discussed in more depth below.



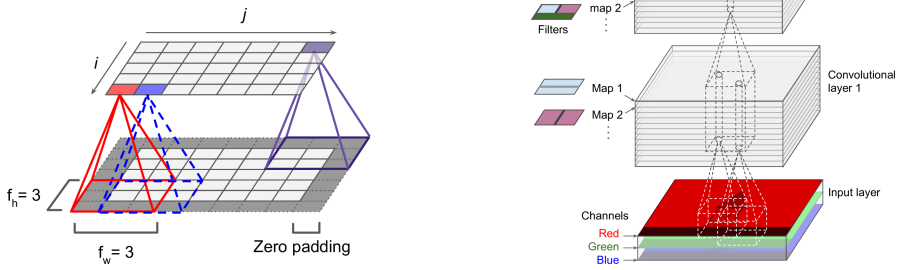
(a) Convolutional layers of a CNN [7]

(b) Recurrent neurons and time steps [7]

Figure 1.3: Examples of (a) convolutional layers in a CNN, and (b) a recurrent neuron, which feeds itself outputs form previous time steps. Notice that for (a) each pixel in an image is represented by a single neuron in the input layer.

Convolutional Neural Networks

Differently from other ANNs, where layers are represented as a 1-dimensional array, CNNs are composed by convolutional layers, which are represented as n-dimensional arrays of neurons. This makes it easier to match neurons at each layer with their corresponding inputs. The basic idea behind convolution is to "summarize" a portion of a layer's neurons, by a single neuron representation in the next layer [7]. Generally these portions overlap with one another, and even though this implies redundancy, it leads to consistent and correlated feature extractions for each layer into the next. This also implies that inputs are reduced dimensionally with each layer, but as each deeper neuron diverges more from the original values of input pixels, it holds potentially meaningful representations of the original data. These representations are known as a feature map. A graphical representation of this can be seen in Figure 1.4a. These feature maps that are generated at each convolutional layer can be stacked using several layers to conform deep CNNs, as shown in Figure 1.5. This can be particularly advantageous for working with multi-channel images, and multi-layered spatial data sets (see inputs on Figure 1.5).



(a) Convolution layers [7]

(b) Deep convolutional layers with stacked feature maps [7]

Figure 1.4: (a) Example of convolution, showing the connections between convolutional layers [7]. Note how the top layer is product of a 3x3 sliding window which condenses features form the lower layer. (b) stacked feature maps from multiple deep convolution layers.

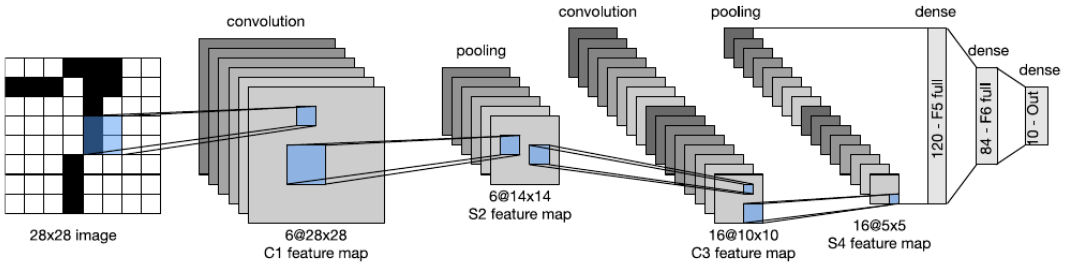
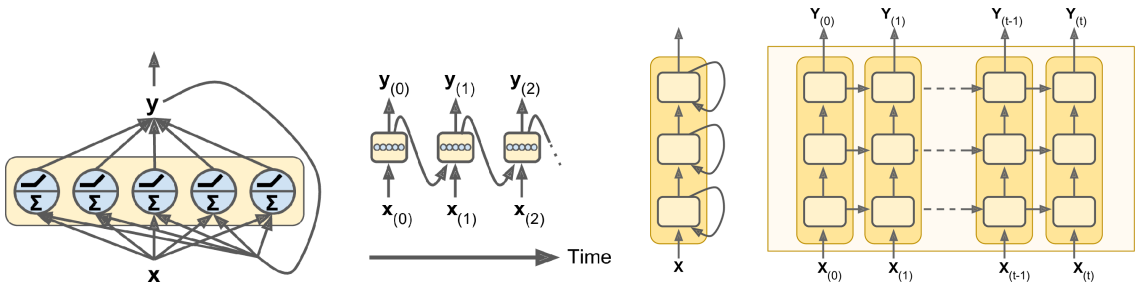


Figure 1.5: Example of a deep convolutional neural network [14]. Notice how feature maps are reduced in dimension at each dense convolutional layer, and also the depths of each dense layer increases before the final dense and output layers.

Recurrent Neural Networks

RNNs are commonly used for handling sequential data, even though they are not the only type of ANN capable of doing so (CNNs can do this too) [7]. A RNN is very similar to other ANNs, except for the fact that that its outputs also point backwards unto itself. The Recurrent neuron shown in Figure 1.3b exemplifies this, and contextualizes the state of the neuron through time. At every time step this recurrent neuron receives the inputs as well as its own outputs from the previous time step. These recurrent neurons can easily be layered, so that at each time step every neuron in the layer receives both the corresponding inputs, as well as the output vector from the previous

time step. Similarly to the case for a single recurrent neuron, this can be seen in Figure 1.6a. As is the case for CNNs, these recurrent layers can be stacked as hidden layers in a network to conform a deer RNN, as see in Figure 1.6b.



(a) A recurrent layer [7]

(b) A deep recurrent network[7]

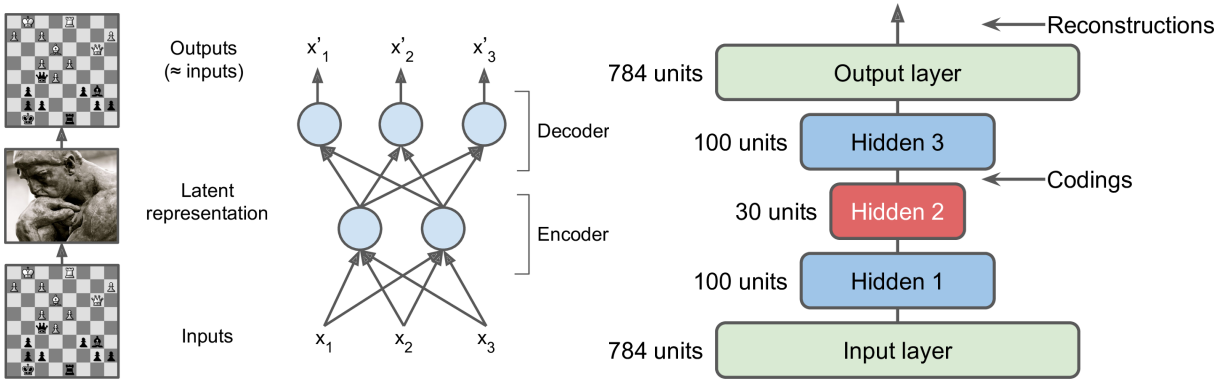
Figure 1.6: Examples of shallow and deep recurrent neural networks [7].

RNNs are one of the most common approaches in DL for NLP-related tasks, as language is naturally expressed as sequences: words are sequences of characters, sentences are sequences of words, *etc.*. Since the output of a recurrent neuron at any time step is a function of all the inputs from previous time steps, RNNs are considered to have a memory of sorts. This allows this kind of neural network to preserve some sort of state across time steps, admittedly for short periods of time given that information is lost as data traverses an RNN. Long Short-Term Memory cells are a kind of recurrent neuron which allows to detect long-term dependencies in the data, and thus allows to overcome this limitation by allowing RNNs to persist relevant patterns through time [7]. LSTM models are widespread for NLP applications, however these type of RNN also plays an important role in building time series prediction models, as they are best suited for extracting meaningful patterns from long-running, non-linear trends.

RNNs usually take a sequence of inputs and produce a sequence of outputs (sequence to sequence network), which is useful for problems like time series prediction. CNNs, on the other hand, generally receive vectors (*i.e.* 2D images) as inputs and output vectors as well (vector to vector networks). However, DNNs can be arranged as sequence-to-vector networks –which are useful for problems like sentiment analysis–, as well as vector-to-sequence networks which enable to tackle problems like image labeling. These sequence-vector and vector-sequence transformations, together with techniques known as *attention mechanisms*, give way to transformer architectures [7].

Transformers and Autoencoders

Autoencoders are a particular kind of ANN, which are capable of learning dense representations of the input data (called latent representations) in an unsupervised way. These networks are composed of two opposing transformer-based architectures: an encoder and a decoder. The idea behind this is that the encoder extracts (or codes) the features that make up an input, and the decoder learns how to construct an output from the latent representation, which is equivalent to the input layer. Figure 1.7 exemplifies these principles. These architectures have given rise to Generative Adversarial Networks (GANs), which leverage adversarial training using two neural networks (a generator and a discriminator). This way, the generator's job is to produce "real enough" outputs, such that the discriminator can't tell them apart from known, real samples. Transformer-based methods have also enabled the use of pre-trained models as a means of transferring expertise or skill from one model to another. This ability also has given rise to very powerful language models approaches based on bidirectional encoder representations (*i.e.* BERT, GPT-3, and GPT-4) [7].



(a) Encoder-decoder architecture [7]

(b) Deep autoencoder architecture [7]

Figure 1.7: Autoencoders. (a) basic concepts behind the encoder-decoder architecture, while (b) presents the architecture for a deep autoencoder; notice the symmetry between the encoder and decoder parts of the deep network.

A particular aspect of transformer-based architectures is that it dispenses with the needs for recurrence when dealing with sequences, and for convolution when dealing with spatial features. Instead, it relies on a mechanism called *self-attention* [7]. Self-attention allows transformers to compute representations of its input and output, without having to rely on RNNs or CNNs [15].

1.2.3 Transformers for Natural Language Processing

Natural language processing (NLP) is a specific machine learning (ML) application (as well as an entire area of study in the realm of ML), which aims to model, and analyze unstructured text documents written in human languages (see see Sec. 1.2.1). Within this discipline, three types of problems stand out: named entity recognition (NER), text classification (TC) and sentiment analysis (SA).

Firstly, named entity recognition (NER) is an NLP problem that revolves around recognizing entities (*e.g.* names, places, objects on which actions are exerted upon, *etc.*) present in a text. This problem is generally hard, due to the fact that different kinds of entities often share names, and names have different meanings depending on context. Second, text classification involves assigning a specific label to a body of text, usually related to the topic around which the text revolves around. This problem is also difficult, as even though there tends to be a high correlation between certain keywords and topics, dealing with unstructured text almost guarantees that these keywords will not be the only indicator needed to express a given class, and words may not be presented in standardized forms (*i.e.* idioms, abbreviations, *etc.*). Semantics play an important role in this field. Lastly, sentiment analysis is a subset of text classification, where the end goal is to classify a body of text with respect to a human emotion or scoring system, generally as a way of extracting significant information from other people's opinions. This problem is strongly relevant nowadays, given the very large amounts of data available from social media at a global scale.

All of these problems have been explored extensively through various approaches involving diverse methods and techniques, and it must be noted that CNNs, RNNs and transformers have been widely employed to tackle these NLP tasks [7]. Some of the most prevalent methods include using RNNs and statistical methods (word2vec) for generating word embeddings (vector representations of words, recall sequence-to-vector networks), as well as other encodings such as "one-hot" and "bag of words" representations [7]. Another interesting problem addressed by NLP is the machine translation of human languages. An example of a simple RNN-based machine translation model for English-to-French translation is shown in Figure 1.8, which showcases the use of word embeddings.

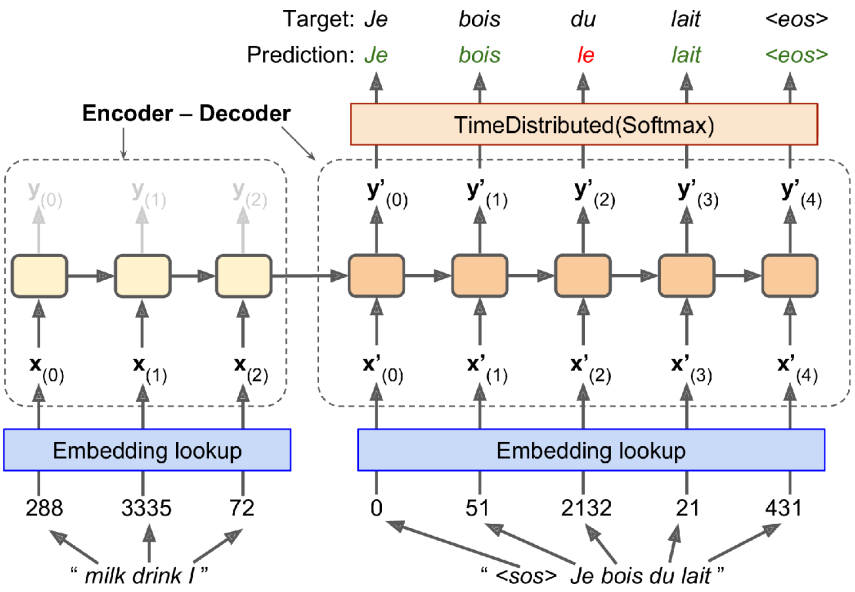


Figure 1.8: Simple RNN-based machine translation model, which exemplifies the use of word embeddings [7]

BERT: Bidirectional Encoder Representations from Transformers

Bidirectional Encoder Representations from Transformers (BERT) is a DL architecture designed by researchers at Google, used for text and language prediction tasks [16]. Pre-trained BERT models are made available to the general public, which have been trained on very large bodies of unlabeled text: the whole of Wikipedia in English (around 2,500 million words), and a comprehensive collection of literary works known as The Toronto Book Corpus (around 800 million words), to name a few. Models of various sizes are available, with an increasing number of parameter ranging from 110 million to 340 million parameters, depending on the amount of self-attention heads.

Pre-trained language models such as BERT enable the abstraction of both contextual and semantic representations from natural language. This is achieved by the use of contextualized word embeddings (which characterize what role a word plays in a sentence) and sentence embeddings (which abstract the meaning a sentence holds), which ultimately implicate that BERT *can be* context aware.

1.2.4 Large Language Models

Recent developments in the field of Natural Language Processing (NLP), have yielded increasingly larger and larger (in terms or trainable parameters) language models. Particular distinction seems to occur when jumping from millions of parameters (BERT, 340 million [16]), to billions of parameters (GPT-3.5, 117 billion), and the fact that LLMs are not necessarily fine-tuned to specific tasks. LLMs have shown noteworthy performance compared to fine-tuned pre-trained models, which has shifted the landscape from task-specific language models, to general purpose language applications. The very rapid adoption of LLMs like OpenAI's ChatGPT [12] by the general public over the last years, have transformed how people interact with these models, and demonstrated the capabilities of applications like natural language generation, code generation, and using LLMs as a general tool for retrieving information and performing search [17].

Generative Pre-trained Transformers

OpenAI's ChatGPT is a *chatbot* application based on an architecture called Generative Pre-Trained Transformers. As its name indicates, and by using the taxonomy previously defined, this ML system corresponds to a Natural Language Processing application (LLM), which relies on Deep Learning architectures based on transformers. At a very basic level, models like `ChatGPT-3.5-turbo` generate responses to user input, by being trained to *predict* the most likely answer that the user's input should have; and these models were trained with unfathomable amounts of text data, which contain in itself knowledge and interactions powered by the Internet. While this simplified explanation can be conveyed in a straightforward way, it is often difficult to reconcile this basic notion with the type of coherent and meaningful interactions one can have with a tool like ChatGPT.

However, it has been demonstrated that while general interactions and responses from GPT-based models can be satisfactory, GPT-3.5 struggles when dealing with prompts related to mathematics, and certain commonsense tasks [13]. Subsequent improvements came to ChatGPT with the release of the much larger GPT-4 model, since this newer version expanded the model's *context memory*, and increased its parameters from 175 billion in GPT-3.5 to over a trillion for GPT-4.

1.2.5 Transformers for Spatial and Temporal Applications

Transformers have traditionally been the dominant architecture for Natural Language Processing (NLP) applications. However, multiple recent developments have attempted and succeeded in transforming the landscape in fields such as computer vision, by relying exclusively on transformer-based architectures to replace traditional applications which rely on Convolution and Recurrence. The transformer's self-attention seems to provide viable alternatives to more traditional RNN and CNN approaches, when certain circumstances can be procured and concessions made. The present section will briefly expose three relevant transformer-based architectures explored throughout the present work.

Vision Transformers

Vision Transformers (ViT) are a type of model which aims to provide a reliable alternative to CNN-based image classification [18]. While transformers are commonplace to NLP tasks, the authors question how CNN-like architectures for computer vision had started incorporating self-attention as a means to replace convolution entirely. Thus, motivated by how the Transformer has had wide success in NLP tasks due to its scalability, the authors proposed applying standard transformers directly to images, with the fewest possible modifications. This approach relies on splitting images into patches, and providing a sequence of linear embeddings of these patches as inputs to a Transformer. Image patches are treated analogously to tokens in NLP applications, where a sequence of tokens can be used to predict a class, or the next expected token in a sequence. This way (and in conjunction with a Multi-Layer Perceptron (MLP) classification head), a ViT model can be trained on a sequence of images for image classification, using labeled image data. Figure 1.9 details the architecture of the ViT model. The ViT architecture not only succeeded in providing a transformer-based alternative to CNN image classification, but it also attained very good results compared to typical convolutional architectures [18].

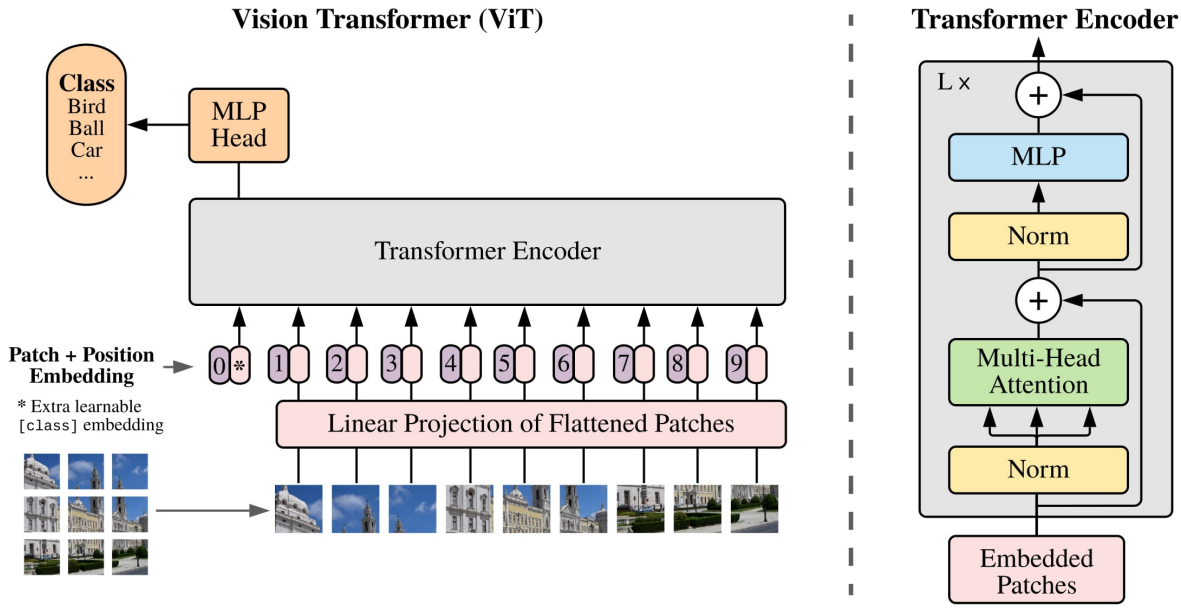


Figure 1.9: Vision transformer architecture [18]

Segmentation Transformers

Segmentation transformers (Segformers) are a type of model which aims to provide a simple and efficient alternative to perform semantic segmentation (labeling or tagging the contents of an image) using transformers, instead of relying on Convolutional Neural Networks (CNNs) [19]. This powerful segmentation framework combines Transformers with lightweight MLP decoders. This enables Segformers to keep model performance scalable by avoiding the use of complex decoder architectures, and also to incorporate both global and local attention to render powerful input representations. Segformers consist of two main modules: a hierarchical transformer which extracts both coarse and fine features, and a lightweight All-MLP decoder, which fuses multi-level features, and predicts the semantic segmentation task. The Segformer architecture is shown in Figure 1.11. In addition to semantic segmentation, hierarchical transformers can also be used for image classification.

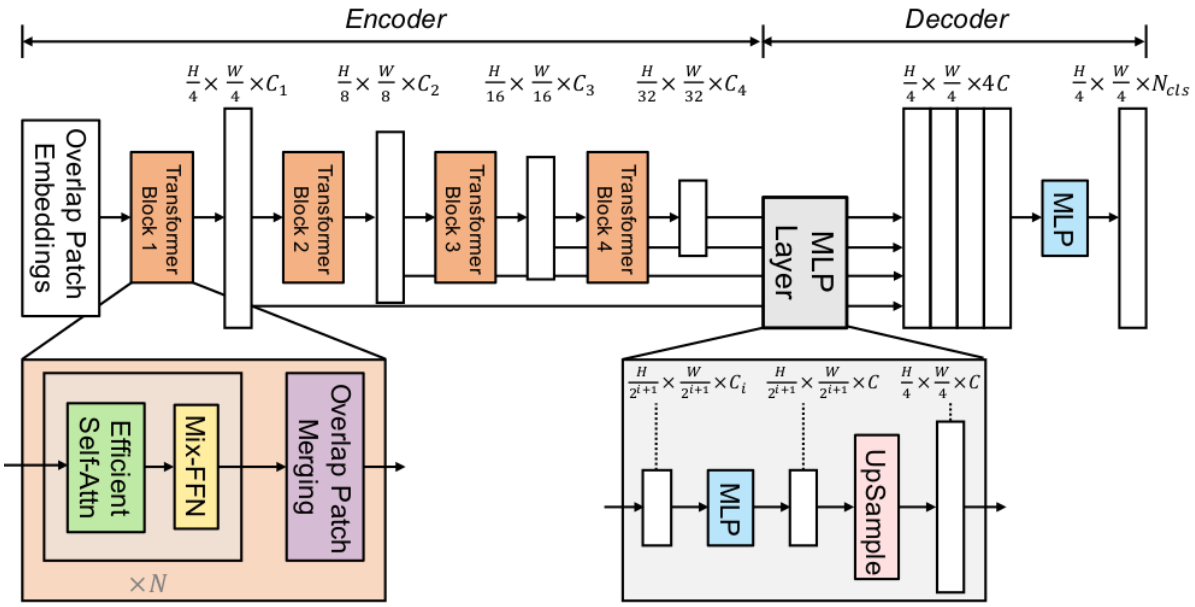


Figure 1.10: Segmentation transformer architecture [19]

Video Masked Autoencoders

Masked image modeling is a self-supervised pre-training method for computer vision tasks, based on the principle of masking out a portion of an image, and then reconstructing the masked-out portion. Masked Autoencoders introduced an asymmetric encoder-decoder architecture for masked image modeling, and Video Masked Autoencoders are an extension from Masked Autoencoders applied to video applications. By masking portions of the successive image frames that make up a video (video tube masking), VideoMAE models are able to deconstruct and reconstruct meaningful representations during the pre-training process [20]. Paired with a classification head at the end of the pipeline, VideoMAE models can be used for image sequence (video) classification.

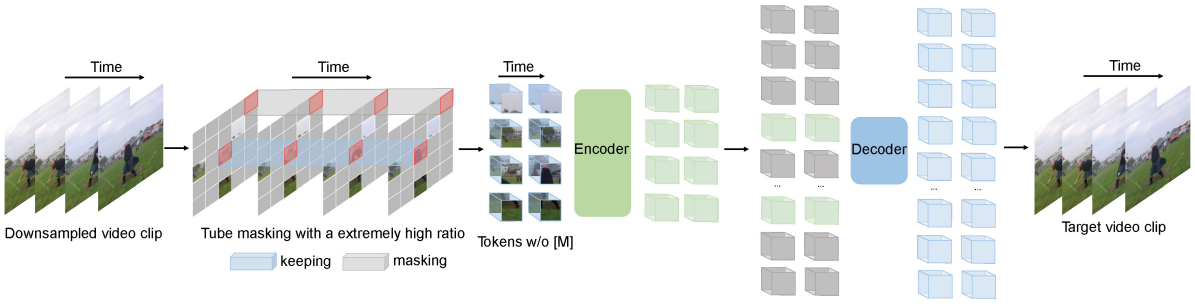


Figure 1.11: Video Masked Autoencoder architecture [20]

1.3 Hydrology, Floods and Flash Flood Forecasting

A basin (watershed, or catchment) is the basic unit used in hydrology, to denote a finite, contiguous area, such that the net rainfall or runoff over that area will contribute water to its outlet. Bounds for a given basin can be defined by the topography, where runoff will travel from higher to lower elevation, and rainfall that falls outside of these bounds will not contribute runoff at the basin's outlet [21]. A graphical representation is shown in Figure 1.12. Gauge stations are usually placed at these outlets to register a stream's behavior, as it responds to the hydrologic processes affecting its watershed. By employing these gauges, as well as weather RADAR and other remote sensing techniques (*i.e.* satellite), hydrologists are able to measure and estimate the intensity and amount of precipitation over a basin (quantitative precipitation estimation, or QPE), as well as the water's stage (level above the stream bed), and flow in the stream (discharge). These are some of the essential components needed for building hydrologic models.

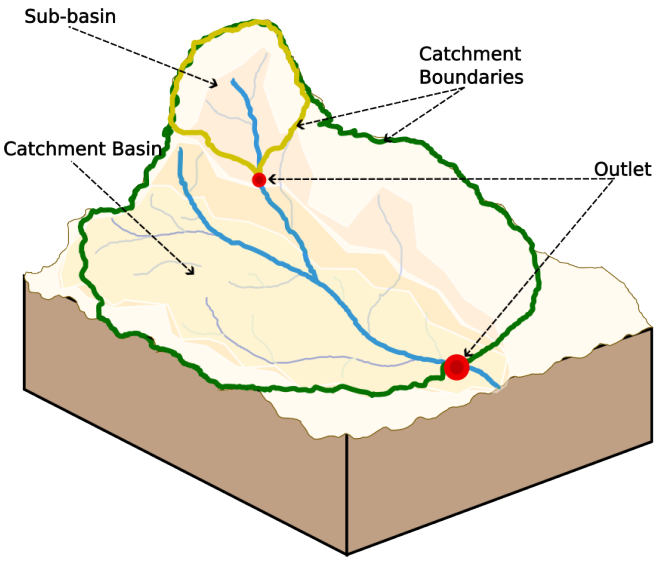


Figure 1.12: A general diagram of a watershed or basin, based on the original design shown in [22].

Flooding conditions are determined by using water levels (or stage) in the stream's channel, with respect to its maximum bankfull conditions. When the stream's stage exceeds the maximum height at which a river's banks are, water in the stream moves outwards over the floodplain, and flooding is observed [21]. Thus, floods are a direct consequence of the rise in water stage in a stream (or body of water), and generally relate to rainfall events that impact their respective basins. Flash floods are floods that follow the causative storm event in a short period of time, with water levels in the drainage network reaching a crest within minutes, to a few hours, after the onset of precipitation. Consequently, flash floods are deemed extremely dangerous events, which generally allow short lead times for the emission of warnings, and for making life-saving decisions. This grave implication has led to direct significant efforts in hydrology throughout the years, for providing reliable and actionable flash flood modeling, monitoring and forecasting methods, frameworks and products.

The Multi-Radar Multi-Sensor (MRMS) system provides precipitation estimates by mosaicking (spatially aggregating and temporally synchronizing) data from over 180 weather RADARs on a grid with a spacing of around $1km^2$ ($0.01°$ latitude and longitude grid spacing), which is updated every 2 minutes across the Conterminous United Stated (CONUS) [5]. The Flooded Locations and Simulated Hydrographs (FLASH) project provides forecasters with a suite of products for flash flood forecasting over

the US and outer territories, based on MRMS rainfall estimates, which are updated at 2-minute and 10-minute intervals (depending on the product) [4]. While FLASH's 2-minute products correspond to RADAR-only QPE products which are based solely on MRMS precipitation data, 10-minute products are based on distributed hydrologic model outputs, which ingest MRMS precipitation alongside additional states and static parameters at every time step. At the core of FLASH's hydrological capabilities lies the Ensemble Framework for Flash Flood Forecasting (EF5), an open-source, distributed hydrologic modeling framework [23].

One set of QPE-only products provided by FLASH are focused around average recurrence intervals (ARI), calculated using 1, 3, 6, 12, and 24-hour precipitation accumulations. An ARI is a measure of event magnitude that indicates the time interval in which given event will occur once, on average, measured in years [21] (*e.g.* 50-year flood, a flood expected to occur on average once in a 50-year period). FLASH ARI products range between zero to 200-year recurrence intervals. Events that occur at larger ARIs (*i.e.* 200-year events) are expected to be catastrophic, compared to those with lower ARIs (*i.e.* 1-year events).

FLASH also provides products based on QPE-to-flash-flood-guidance ratios (QPE/FFG), which are based on comparisons between MRMS QPE accumulations and Flash Flood Guidance (FFG). FFG is a rudimentary but still widely used method for flash flood forecasting, which is defined as "the threshold rainfall, over 1, 3 and 6-hour accumulations, required to initiate flooding on small streams that respond to rainfall within a few hours" [24]. These ratios compare the MRMS QPE value at each grid point on the CONUS domain, with the FFG threshold for the same point at a given accumulation. FLASH QPE2FFG products range from zero up to 5.0, which would indicate up to a 500% exceedance of MRMS QPEs over a predefined FFG threshold for any given grid cell over the CONUS, at a given accumulation.

Aside from these ARI-based and FFG-based products, FLASH produces other products based on three distributed hydrologic models: the Coupled Routing and Excess Storage (CREST) model, the Sacramento Soil Moisture Accounting model (SAC-SMA), and a Hydrophobic model (HP) [4][23]. These products are discharge (Q[$m^3 \cdot s^-1$], also referred to as streamflow), unit discharge (q[$m^3 \cdot s^-1 \cdot \text{km}^2$], also referred to as "UnitQ" or unit streamflow), and soil saturation (SS[%], not produced for the HP model). Q estimates the overland and channel flows in cubic meters per second (cms), while q provides a *basin area-normalized* stream flow, which helps highlight specific locations

that are most likely experiencing anomalous flows (*i.e.* possibly flash floods) [4]. Ultimately, UnitQ describes overland and channel responses over the CONUS in an easy-to-interpret scale, which helps forecasters identify and anticipate increases in flows as responses to the causative precipitation events; particularly for those events which are likely to lead to flash floods and widespread areal flooding.

1.3.1 Flash Flood Warnings and Reports

The National Weather Service (NWS) is the organization in charge of the monitoring and issuance of weather-related alerts (watches and warnings) in the United States. By design, watches and warnings are used whenever weather hazards threaten life or property. While watches are issued whenever the possibility of a significant event is likely, warnings are issued for significant weather events, for which impactful conditions have been met and immediate action is necessary from those within the impacted area.

Reports are a crucial element in informing both the NWS forecasters, as well as the general public, about hazardous weather events and conditions. Local storm reports (LSRs) are issued by the NWS as it receives significant information about hazardous weather conditions in their County Warning Area (CWA). These LSRs serve to both inform the public about the hazardous conditions, as well as the basis for both issuing and verifying weather-related alerts (particularly warnings). These local storm reports are also the basis for the creation and publication of Storm Data reports, which are a post-facto survey and analysis of specific hazardous events and their impacts. Lastly, the general public can also submit their own observations of general and hazardous weather conditions and impacts through the Meteorological Phenomena Identification Near the Ground (mPing) initiative. These three types of reports play a crucial role in both research and operations, with respect to collecting and distributing data about hazardous weather, including flash flooding events. Each type of report and its relationship to flash flooding is detailed below.

Local Storm Reports

Local Storm Reports (LSR) are a type of manual observation, which are collected and used and archived by the National Weather Service (NWS). These reports are filed fol-

lowing the issuance of warning products by NWS forecasters, and sources for their contents usually come from emergency managers, local authorities, 911 calls, cooperative observers, social media, and the general public. These reports are constituted by 11 different attribute fields, including the NWS office which issued the report, time and date, type and magnitude of the event being reported, and a remark about the event. Specifically for flash flood events, LSRs provide a location and a description of the flooding event, pertaining to the observed conditions for the event at the given location (*i.e.* "Road flooded, culvert full/overflowing, etc."). This makes LSRs to be considered near-real-time reports of flood events.

The University of Iowa's Iowa Environmental Mesonet (IEM) maintains a standardized archive of Local Storm Reports [25]. Each report is composed of 14 different fields, which are presented as columns when the data is downloaded as a text file. Relevantly among these fields, for each report there is a specified date, time and location (lat/lon), as well as a text-based remark wich described the incident. It must me noted that even though there is a *magnitude* field, this is left empty in the case of flash flood LSRS. These fields are described in detail in Table 1.1, in accordance with the IEM's official documentation.

Column ID	**Description**
VALID	Timestamp of LSR in GMT/UTC time
MAG	Magnitude value of the LSR
WFO	Weather Forecast Office originating the LSR
TYPECODE	1 character identifier of the report type, IEM specific
TYPETEXT	Textual value of LSR type used in report
CITY	Location used for the LSR
COUNTY	County/Parish of the LSR city
STATE	Two character state abbreviation
SOURCE	Who reported the LSR
REMARK	Text summary with the LSR
LAT	Latitude
LON	Longitude
UGC	IEM computed NWS UGC code (6 character), sometimes null
UGCNAME	IEM computed NWS UGC name (128 character), sometimes null

Table 1.1: Structure of Local Storm Reports

StormDat Reports

Additional post-facto reports, known as Storm Data reports (StormDat), are also constructed and maintained by the NWS based on existing LSRs, on-site surveys of impacted areas, and other sources of information (such as emergency managers, law enforcement agencies, news media *etc.*). NOAA's National Center for Environmental Information (NCEI) maintains the Storm Events Database which is accessible online. StormDat reports are available ranging as far back as January of 1950 [26]. The database includes tornado events since 1950, severe thunderstorms since 1955, and all other weather events (e.g. Flash Floods) since 1996 [27]. StormDat are generally available between 90 to 120 days after an event has occurred, which means that compared to LSRs, can not be considered near-real-time observations.

StormDat reports provide a detailed description as well as updated information regarding an event, including its impact on infrastructure, property, and goods (usually measured in dollar amounts), as well as counts of any injuries or fatalities. Within StormDat reports there is also room to report up to eight different geographic locations (description, latitude, and longitude), which can form anything from a single point, to an octahedral polygon which is meant to denote and describe the area that was impacted by the event. Additionally, StormDat reports introduce the notion of an event versus an episode: a single event is describe in its entirety by a single StormDat events (in turn composed of potentially multiple LSRs), while a single episode is potentially composed of multiple StormDat events (e.g. a long-lived and/or spatially distributed event which affects multiple geographic areas at different times of its lifecycle). Within each StormDat report both a unique identifier and description are given for the report's event, as well as for the episode to which the report corresponds. Table 1.2 lists the fields that make up the structure of each StormDat report.

Columnd ID	Description
Begin Date/Time	Event start date and time
Event Type	Type of event recorded
WFO	Weather forecast office ID which issued the report
End Date/Time	Event start date and time
Timezone	Timezone on which the dates and times are being recorded
UGC Location ID	NWS UGC code
State	Abbreviation of the state in which the event occurred (2 letters)
County FIPS	County FIPS code (5 digits)
County Name	County name in which the event occurred
Region	Abbreviation for the NWS region to which the issuing WFO belongs to
Direct Injuries	Number of direct injuries associated to the reported event
Indirect Injuries	Number of indirect injuries associated to the reported event
Direct Fatalities	Number of direct fatalities associated to the reported event
Indirect Fatalities	Number of indirect fatalities associated to the reported event
Property Damage	Dollar amount derived from property damages associated to the reported event
Crop Damage	Dollar amount derived from crop damages associated to the reported event
Flood Cause	Description of what caused the reported flooding event
Event Source	
Event Narrative	Narrative describing the reported event
Episode Narrative	Narrative describing the overarching conditions which led and relate to the reported event
Event ID	A unique identified for the reported event
Episode ID	A unique identifier which ties multiple StormDat reports to singular episodes
Location #1 (City/Range/Azimuth)	Location description of the event's polygon point #1
Location #1 (Lat)	Latitude of the event's polygon point #1
Location #1 (Lon)	Longitude of the event's polygon point #1
...	...
Location #8 (City/Range/Azimuth)	Location description of the event's polygon point #8
Location #8 (Lat)	Latitude of the event's polygon point #8
Location #8 (Lon)	Longitude of the event's polygon point #8

Table 1.2: Structure or StormDat reports

mPING Reports

NOAA's National Severe Storms Laboratory (NSSL) collects weather reports from the general public through a free mobile application available to iOS and Android smartphones. The Meteorological Phenomena Identification Near the Ground (mPing) allow for the collection of weather observations at ground level, which weather radars cannot *see* (since they measure at elevations of 0.5°-19.5°) [28]. By anonymously crowdsourcing weather reports a database has been maintained since 2012 through a partnership between NOAA/NSSL and The University of Oklahoma's Cooperative Institute for Severe High-Impact Weather Research and Operations (CIWRO, formerly the Cooperative Institute for Mesoscale Meteorological Studies – CIMMS).

Data acquired through mPing which includes type and size (hail) of precipitation (liquid, mixed, solid), which helps forecasters verify dual-polarization RADAR-based object size, shape, and type. This data is also used by researchers who develop computer algorithms and weather models, to automatically sort and classify precipitation types, and distinguish non-precipitation signals in RADAR data [29]. Additional to precipitation types (rain, snow, hail), other observation reports can be submitted through the mPing app: winter weather impacts (downed trees, frozen/burst water pipes, school/business closures, power/internet outages, *etc.*), wind damage (size of limbs broken, uprooted trees, home/building destruction), tornadoes (on ground, water spout), reduced visibility (fog, dust/sand, snow, snow squall, smoke), and floods (four levels of increase impacts). The four impact levels mPing allows for observed floods are:

1. River/Creek overflowing; Cropland/Yard/Basement flooding
2. Street/Road flooding; Street/Road closed; Vehicles stranded
3. Homes/Buildings filled with water
4. Vehicles/Homes/Buildings swept away

It must be taken into account that, since these mPing reports address observed impacts specifically, they are widely applicable to widely varied flooding scenarios and types (fluvial, pluvial, groundwater, and coastal flooding), and not just to flash flooding cause by the rapid onset of precipitation.

1.3.2 Flash Flood Impact Frameworks

Throughout the development of the present work, two approaches to systematically characterize, quantify, and prescribe impacts to flash flood events have been encountered. The Impact-Based Warning (IBW) framework has been designed and enforced by the NWS across the US, while the Flash Flood Severity Index (FFSI) was developed by researchers from the NWS, the NWS Warning Decision Training Division, NOAA's NSSL, The University of Oklahoma, the Virginia Polytechnic Institute and State University, Columbia University, Kansas State University, the University of Illinois at Urbana-Champaign, Colorado State University, and Princeton University. The following section will describe these approaches in detail, and discuss their particularities.

Impact-Based Flash Flood Warnings

Since late 2019, the NWS has followed a national initiative to transition all warnings into an impact-based format, including flash flood warnings. The main goals of impact-based warnings (IBW) are to provide additional valuable information about hazards to media an Emergency Managers, which facilitate improved public response and decision making; and to better meet societal needs in the most life-threatening weather events [30]. Specifically for flash flood warnings, IBWs aim to provide enhanced information about the hazard through an impact narrative, and a flash flood damage threat tag describing the magnitude of the event. To describe these magnitudes, a three-class framework is used to describe the level of impact associated with each flash flood warning: base-level flooding, considerable flooding, and catastrophic flooding. These additional fields (impact narrative and damage threat tag) intend to help determine which calls to action –such as triggering Wireless Emergency Alerts (WEAs)– should be enacted by the emergency management community and by the general public [31]. It must me noted that guidance on how to determine specific impact classes from operational flash flood forecast products, such as FLASH's unit streamflow (UnitQ), is loosely provided, and carries large amounts of uncertainty. This is evident when observing that the most likely ranges for UnitQ at which `considerable` and `catastrophic` flooding can occur, overlap completely with one another. Table 1.3 details the UnitQ value ranges associated to each IBW class. It is clear that discerning between these two classes based exclusively on a single

FLASH product's output is a difficult tasks, and forecaster experience and knowledge play a crucial roles in the decision-making process.

Category	Impact Label	UnitQ Most Likely Range
1	Base-level flood	200 - 400 cfs/mi^2
2	Considerable flood	750 - 1075 cfs/mi^2
3	Catastrophic flood	750 - 1075 cfs/mi^2

Table 1.3: Impact Based Warning Labels [31]; UnitQ values excerpted from slides courtesy of the National Weather Service Warning Decision Training Division.

Flash Flood Severity Index

The flash flood severity index (FFSI) is a damage-based, post-event assessment tool and impact framework, rated in a scale of five impact-based categories ranging from very minor flooding, to catastrophic flooding [32]. These categories were designed to be similar to those division of impact observations associated with mPing reports (see Section 1.3.1). In contrast to the IBW framework which aims to prescribe the impact level a given flash flood warning is expected to have as it is being issued, the FFSI framework was constructed by analyzing historical case studies, local storm reports, flash flood warnings, and performing interviews with forecasters. By pairing specific impact descriptions related to locations and infrastructure (which also revolve around human activity) with event impact levels, FFSI associates increasingly impactful event observations with higher impact categories. Table 1.4 details which impacts are associated to each impact class. As a result of this, the FFSI framework managed to establish a concise but yet flexible impact-based approach to categorize and characterize levels of flash flood impacts.

Category	Impact Description
1 - Minor flood	River/creek overflowing; cropland/yard/basement flooding
2 - Moderate flood	Street/road flooding; road closures
3 - Serious flood	Vehicles, homes and/or buildings inundated with water; road/bridge damage
4 - Severe flood	Vehicles and/or mobile homes swept away
5 - Catastrophic flood	Buildings/large infrastructures submerged; permanent homes swept away

Table 1.4: Flash Flood Severity Index [32]

* * *

From the above descriptions on two different flash flood impact frameworks, it can be observed that they fall into different locations in a subjective-objective guidance spectrum, while also being differentiated in scope, by being forecast-oriented (prognostic) vs observation-oriented (retrospective). IBW's guidance is more subjective and streamflow-oriented, since there are guiding principles on how to use observed streamflow forecasts to approximate which IBW classes to attach to a given warning (see Figure 4.2), but this guidance also allows large uncertainty for deciding between the two most severe levels of impacts. mPing's guidance is more objective and observation-oriented, as users can report specific observed impacts grouped into four categories. Lastly and similarly to mPing, FFSI's guidance is more objective and impact-oriented, since it provides forecasters clear impact-based guidance, which would enable forecasters to determine one of five appropriate FFSI category based on the reported and/or observed impacts. It should be noted that FFSI's guidance is deemed relatively more objective than mPing's, since it encompasses one more class, and it pertains specifically to flash floods, whereas mPing's is more generalized towards any flooding. A representation of this guidance-scope comparison can be seen in Figure 1.13.

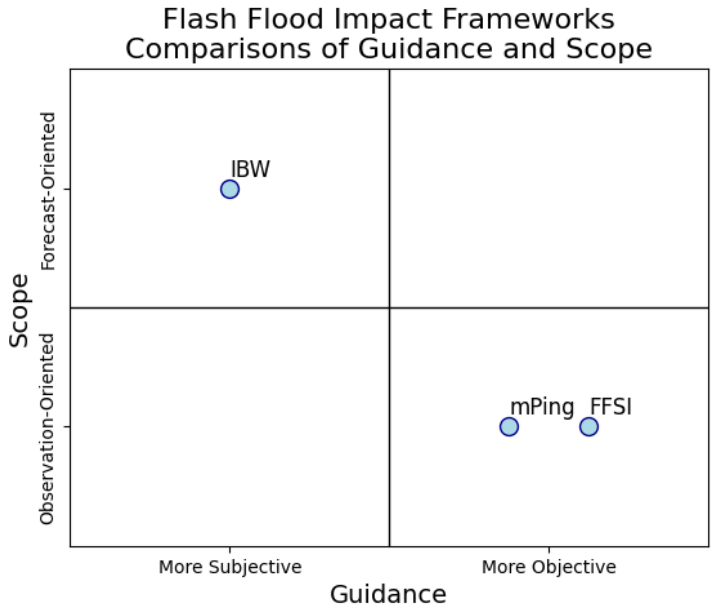


Figure 1.13: Quad chart comparing the guidance and scope of the three flash flood impact frameworks: Impact-based warnings vs mPing vs Flash Flood Severity Index.

From the above, it is clear that IBW and FFSI exist in opposite ends of the spectrum, where:

- It is possible to estimate IBW impact labels from streamflow forecast quantities, but this guidance does not exist for FFSI.
- It is possible to estimate FFSI impact classes from observed impacts, but this guidance does not exists for IBW.

This dichotomy provides an opportunity to perform foundational research at the intersection of these two impact frameworks, and to hopefully pave the way towards an actionable syncretism between the two. This would ultimately enable a consistent estimation of impacts levels, whether based on impact observations or streamflow forecasts.

1.4 Operationalization of Flash Flood Reports

As part of the National Oceanic and Atmospheric Administration (NOAA), the National Severe Storms Laboratory's (NSSL) Warning Research and Development Division (WRDD) aims to "*develop new weather and water related applications and water resource management tools help NWS forecasters produce more accurate and timely warnings of flood events*" [3]. Contributing to NSSL's mission, and as previously described in Section 1.3, the FLASH project provides a collection of flash flood related products to forecasters [4]. These products were developed at WRDD, thoroughly tested by forecasters and scientists (Hydrometeorological Testbed Hydro Experiments of 2018 and 2019), and successfully transitioned to operations as part of the MRMS [5] suite of products.

Since 2016, NOAA's Weather Program Office's (WPO) aims to ensure the continuous development and transition of the latest scientific and technological advances into the NWS, through the Joint Technology Transfer Initiative (JTTI) program [6]. Projects funded by the JTTI program are supported by NOAA to develop, test and evaluate matured weather research that can potentially transition to operations. This dissertation stems from a funded JTTI project titled *Products to Guide Impact-Based Flash Flood Warnings in the National Weather Service*. This project's overarching goal was to "*develop, evaluate and transition products to the NWS that will guide forecasters in the selection of flash flood damage threat tags*", meant to be included within impact-based flash flood warnings. As stated in Section 1.3.2, the NWS has measures in place to communicate possible impacts associated with flash flood predictions, when forecasters issue warnings.

Building on products like LSRs and FLASH's unit streamflow products, and funded by the aforementioned JTTI initiative, the present dissertation stems from the need to design and prototype novel experimental forecasting tools that can distill information from MRMS-based FLASH products, to aid forecasters in the issuance of impact-based flash flood warnings. Because of the NWS' initiative to enforce IBWs, there is an opportunity to provide forecasters with a product that informs their decision-making process of assigning specific damage threat tags to IBWs. This is deemed a critical decision support tool for the NWS, since assessing the probable specific damages associated with each impact category can be a difficult task, as these relationships vary by location, proximity to population, infrastructure, bodies of water, *etc.*. Additionally, there are multiple frameworks for describing flash flood impacts (IBW vs FFSI,

see Section 1.3.2), which are prone to be objectively compared and contrasted, using operational flash flood forecasting products as a common baseline.

The above gives rise to the following research question which guides the present working **hypothesis**: *Given an operational flash flood forecast, it is possible to annotate specific threat tags that inform users (forecasters) of possible associated impacts expected for the forecasted event.*

In order to address this hypothesis, the following two main challenges must be addressed:

1. a data set which can relate historical flash flood events with specific measures of impact must be built, and
2. a proof-of-concept model capable of producing impact probabilities based on real-time measures of precipitation and flow response (among other variables) must be designed and implemented.

To reach these objectives, this project will be undertaken in two different, successive phases. Initially, a machine learning (ML) model to classify preexisting (and future) flash flood reports into three impact categories (base, considerable and catastrophic) must be built, using a sample of reports manually classified by experts. Once this classification model has been trained and tested to produce sound results, it will be used to construct a surrogate, historical (retrospective) observation data set of impact-classified flash flood reports. These observations will then enable the second phase of the project's overarching objective. Subsequently, for the project's second phase, this observation dataset can be used in conjunction with archived FLASH product outputs, to construct a ML model which predicts flash flood impact probabilities, given QPEs and predicted flow responses as inputs. Once this probabilistic predictive model has been trained, tested and proved to work satisfactorily on historical events and data, it shall enable the final implementation of a real-time capable experimental ML system to forecast flash flood impact probabilities.

This chapter has provided a succinct but rich introduction to the work guiding the main theme of this dissertation. With the aforementioned objectives and milestones in mind, the Chapter 2 presents a literature review of the relevant state of the art for present work. Chapter 3 presents an overarching methodology for this dissertation

work, which is broken down into three sections which detail specific efforts made towards the fulfillment of the present work: Section 3.1 will revolve around attempts to use pre-trained language models and expertly-classified reports, to create an LSR classification model for the IBW impact framework. Section 3.2 will showcase the use of an LLM (ChatGPT) for classifying historical LSRs into probabilistic FFSI impact categories, a performance evaluation for said classification, and the creation of an operationalized report dataset. Lastly, Section 3.3 will demonstrate the potential usefulness of the aforementioned dataset, by training machine learning models which can predict impact classes for historical flash flood events. Figure 1.14 details how these efforts tie into one another, and how they correspond to specific milestones for this dissertation. Results from these three sections will be presented in corresponding sections within Chapter 4. Finally, outcomes, lessons learned, and future work will discussed in Chapter 5.

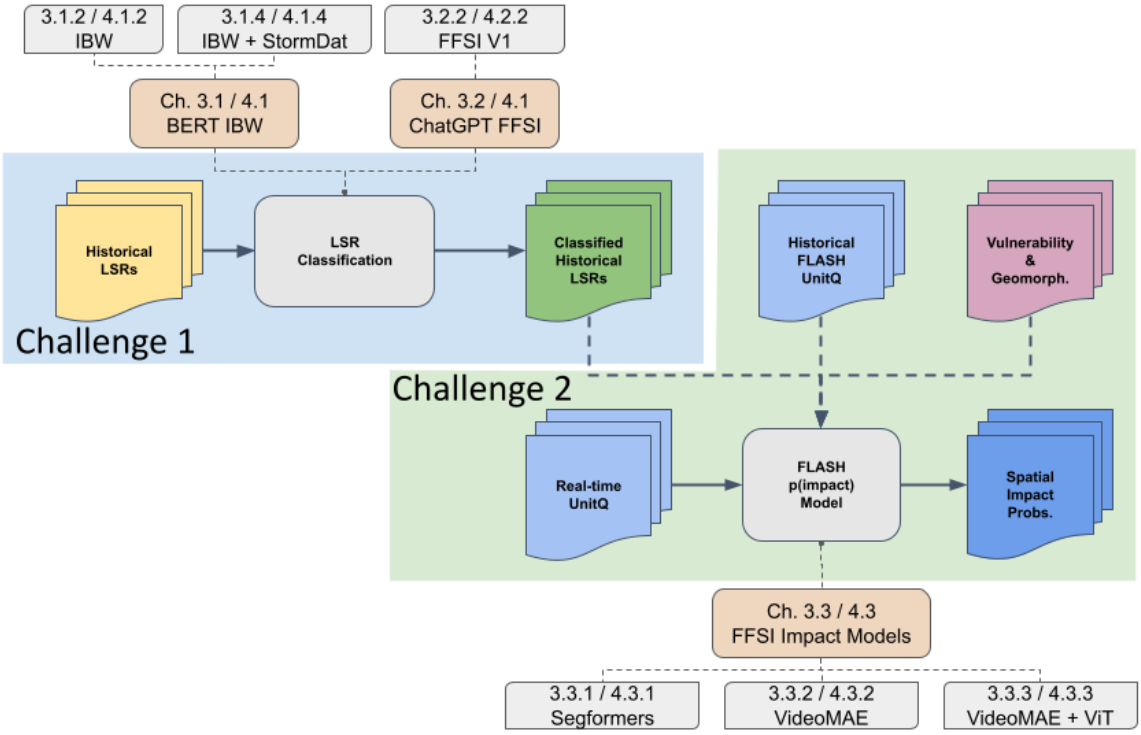


Figure 1.14: Methodology structure with respect to this dissertation's chapters and milestones.

Chapter 2

Literature Review

This literature review is aligned with the two main objectives Section 1.4: 1) the classification of text-based flash flood reports into actionable threat categories, and 2) the development of a ML-based product which can produce spatial, probabilistic, flash flood impact predictions.

In order to establish a broad overview of the available work, and to help define a state of the art concerning relevant uses of Machine Learning (ML), SSL, NLP and DL applications, search queries were constructed by creating (sensible) combinations of the following search terms: *machine learning*, *semi-supervised learning*, *natural language processing*, *deep learning*, *weather*, *floods*, *text*, *classification*, *prediction*. These keyword-based search queries (i.e. "machine learning, weather, text, classification", or "floods, semi-supervised learning") were then used as input in the application *PublishOrPerish* [33], which allowed to retrieve and analyze academic citations from a wide variety of data sources such as *Google Scholar*, *Microsoft Academic*, *Crossref*, *Scopus* and *Web of Science* (among others). After exploring multiple PublishOrPerish search results from around a dozen search queries in various of the aforementioned data sources, 50 different pertinent publications were identified between the years 2007 and 2021. However, after deciding only to focus on (roughly) the last 5 years of developments (2017-2021), this number was reduced to 38.

Reference	Publication Type			SM	xAI	SSL		ML	Deep Learning			NLP			Weather			Floods					Landslides	General Hazards
	R	D	S			Cla	Clu		DNNs	CNNs	RNNs	TC	SA	NER	WF	NWP	EI	FF	MFP	UF	FM	I&R		
[34]		✓		✓				✓				✓												✓
[35]	✓			✓			✓					✓												✓
[36]	✓					✓															✓	✓		
[37]	✓					✓				✓							✓							
[38]		✓				✓				✓							✓							
[39]	✓			✓					✓			✓							✓					
[40]	✓			✓						✓				✓						✓	✓	✓		
[41]	✓			✓		✓				✓		✓												✓
[42]	✓			✓				✓				✓							✓			✓		
[43]	✓					✓														✓	✓			
[44]			✓								✓								✓					
[45]			✓			✓	✓																	
[46]	✓			✓						✓			✓				✓							
[47]			✓						✓	✓	✓	✓	✓	✓										
[48]	✓					✓														✓	✓	✓		
[49]	✓					✓				✓											✓			
[50]	✓				✓					✓					✓		✓							
[51]	✓									✓										✓	✓	✓		
[52]	✓				✓						✓							✓	✓					
[53]	✓									✓					✓	✓								
[54]			✓						✓	✓	✓			✓										
[55]	✓					✓			✓			✓												
[56]	✓								✓		✓								✓					
[57]	✓					✓			✓	✓									✓		✓			
[58]			✓						✓	✓	✓	✓	✓	✓										
[59]	✓								✓										✓					
[60]	✓							✓											✓	✓	✓	✓		
[61]	✓					✓			✓														✓	
[62]	✓							✓		✓											✓	✓		
[63]	✓							✓						✓			✓		✓			✓		
[64]	✓			✓					✓			✓							✓					
[65]	✓					✓				✓									✓		✓			
[66]	✓				✓			✓										✓	✓					
[67]	✓							✓			✓								✓		✓			
[68]	✓			✓				✓				✓			✓									
[69]	✓			✓				✓				✓			✓									
[70]	✓			✓				✓				✓												
[71]	✓								✓										✓		✓	✓		

Table 2.1: Literature review summary. Column abbreviations: **R**=Research, **D**=Dataset, **S**=Survey or Review, **SM**=Social Media Data, **xAI**=Variable Importance/Model Explainability, **SSL**=Semi-Supervised Learning, **Cla**=Classification, **Clu**=Clustering, **ML**=Machine Learning, **DNNs**=Deep Neural Networks, **CNNs**=Convolutional Neural Networks, **RNNs**=Recurrent Neural Networks, **NLP**=Natural Language Processing, **TC**=Text Classification, **SA**=Sentiment Analysis, **NER**=Named Entity Recognition, **WF**=Weather Forecasting, **NWP**=Numerical Weather Prediction, **EI**=Event Identification, **FF**=Flash Floods, **MFP**=Modeling, Forecasting and Prediction,**UF**=Urban Flooding, **FM**=Flood Mapping, **I&R**=Impacts and Risk.

Of the 38 most relevant publication: two were dataset-related publications [34, 38], five were review or survey papers [44, 45, 47, 54, 58], and 31 were research papers [35–37, 39–43, 46, 48–53, 55–57, 59–71]. Thirteen of these 38 publications involved SSL, 16 of them involved NLP, 25 involved DL (most of them involved more than one of these aspects). Ten of the 38 articles involved other ML methods not related to Artificial Neural Networks (ANNs) or DL, and only two of these 10 involved non-NLP tasks. Of those articles related to SSL, 11 focused on semi-supervised classification, one paper concerned semi-supervised clustering, and one article presented a brief literature review. Of those papers concerning DL, 12 dealt exclusively with convolutional neural networks (CNNs), six involved deep multi-layer perceptrons (MLPs), four involved the use of recurrent neural networks (RNNs), and three of them presented comprehensive surveys or reviews involving multiple DL applications using MLPs, RNNs, and CNNs. Most of the articles concerning NLP revolved around text classification (12), while five of them focused on named entity recognition (NER), and only three of them dealt with sentiment analysis (SA). Only three of all 38 publications included or discussed measures of variable importance, model explainability or *explainable AI*. Eleven out of the 38 reviewed publications included Social Media data sources for NLP-related tasks, and only one of these 11 made use of SSL.

Twenty one of the publications revolved around floods, eight of them concerned weather, three of them dealt with general disasters and hazard events (i.e. earthquakes, floods, hurricanes, *etc.*), and one of them targeted landslides specifically. Of those articles related to Weather, most revolved around weather event identification, and weather forecasting; only one of them focused on numerical weather prediction. Most of the flood-related papers focused on flood modeling, forecasting and prediction, and flood mapping. Flood risk and impacts (i.e. flood extent), and urban flooding were also relevantly present with 12 and 9 occurrences respectively. Only two articles revolved around flash floods specifically. Detailed aspects of this literature review's composition can be seen in table 2.1, and the remainder of this chapter will further explore these 38 publications, with respect to the main topics pertaining this dissertation.

2.1 Traditional Machine Learning

During the literature review process, only two articles were found that exclusively employed non ANN-based methods and were not related to NLP tasks [60, 66]. These

studies targeted flood stage prediction for urban floods, and flash flood event peak prediction respectively. While [60] focused on urban flood risk mapping, [66] explored predictor importance aside from flash flood prediction.

In Wu *et al.* [60], data warehousing efforts were made to centralize and categorize surface and terrain data, as well as historical flood data. Then, a Gradient Boosted Decision Tree (GBDT) models were trained on the warehoused data, to predict the depth of urban flooding, given specific return periods of rainfall. Outputs produced by these GBDT models were used to construct flood condition maps for Zhengzhou City in China, in order to assess the city's flood risks associated to different precipitation intensities.

The work by Potdar *et al.* [66] made use of an Extreme Gradient Boosting (XGBoost) model (another technique for predictor selection and multivariate regression analysis, based on decision trees), to predict the peak discharge of flash flood events. A flash flood event dataset spanning ten years was used, which included morphological data for each basin over which a flooding event occurred, as well as information on the causative precipitation event, including the spatial organization of rainfall. The use of XGBoost allowed the authors to perform significant variable importance analysis, confirming from a data-driven approach, that even though spatial organization of rainfall influences a basin's response, peak discharge is more primarily driven by a basin's physiography. Moreover, these results were constructed with unprecedented spatial and temporal representativeness with respect to the characterization of floods across multiple basins (as opposed to the traditional single-basin approaches).

Interestingly, these two papers characterized by using non ANN-based ML approaches made use of boosting methods for decision trees. Boosting refers to any ensemble method that can combine several weak learners into a strong learner [7]. In both cases, the strengths of building ensembles of weak multivariate decision trees were leveraged for multivariate regression, and in the case of [66], predictor selection capabilities were also leveraged for in-depth variable importance analyses.

2.2 Semi-Supervised Learning

Across the literature review, 12 distinct applications (and one literature review [45]) of SSL were identified, which targeted diverse tasks such as NLP or flood mapping, by making use of both non ANN-based methods, as well as deep learning: [35–38, 41, 43, 48, 49, 55, 57, 61, 65].

In Zhang & Vucetic [35], the authors took a semi-supervised approach towards training different classifiers using twitter data (up to 400 million tweets), corresponding to six different disasters between Oct. 2012 and Jul. 2013. Among these disasters, hurricane Sandy, a tornado in Oklahoma, and a flood in Alberta, Canada were included. In total seven different semi-supervised models were built by using relatively small starting sets of labeled data (100 labeled instances or less). Three different clustering methodologies were used separately: (bag of words, word2vec, and Brown Clustering) to build pseudo-labeled datasets, which later trained a logistic regression model to classify tweets as disaster-related, or not. Three differently sized datasets were constructed for testing these models (for each clustering methodology), and served to compare all model results with respect to available training data. Results showed that models built using word2vec clustering provided overall the best performance, and regarding dataset size, results reinforced the notion of "the more data the better".

In the work by Gnecco *et al.* [36], supervised learning and semi-supervised approaches were compared for training SVM-based classifiers, to detect flood-exposed areas as well as areas of high flood risk on the Tanaro River basin in Italy. In this case, models were built using morphological basin features derived from digital elevation models (DEMs), and trained to model the relationship between morphology and the flood hazard. Results showed that semi-supervised approaches outperformed their supervised counterparts, while the number of labeled instances for both was kept the same. Even though if the supervised model were to be trained with a larger amount of labeled instances, the semi-supervised model showed a larger capability for generalization.

The two consecutive works by Racah *et al.* [37, 38] explored the identification of extreme weather events in large scale climate simulation datasets. To achieve this, multi-channel, convolutional autoencoder architectures were used to implement 3D, spatiotemporally-informed models, which were used for semi-supervised bounding-

box prediction. The CNN's data representations were used to explore variable importance and model explainability, leading to exploratory analysis of weather events within the dataset. The author's work presented in [37], yielded a dataset of classified simulated extreme weather events, which was published in [38].

The work of Alam *et al.* [41] presented an inductive, semi-supervised technique to make use of unlabeled tweets, alongside with a subset of relevant tagged tweets, for classifying twitter data as relevant of not to specific hazards such as floods and earthquakes. Making use of Graph-based SSL, a CNN classification model was built using pre-trained word embeddings. These embeddings were extracted from a large crisis-related dataset, using a word2vec model (composed of about 2 million words). Several models were built based on flood-related data and earthquake-related data. Supervised and Semi-supervised models were built and compared, using varying amounts of labeled data for each hazard. Semi-supervised models were found to provide significant improvements with respect to the supervised-only baselines.

The paper by Liang *et al.* [43] introduced a novel cognitive framework for expert-guided labeling of training data (based on SSL), to address the flood mapping problem using remote sensing data. Their flood mapping methodology was based on using SVMs for classification, a thresholding method called NORM-THR for water delineation, and a graph-based clustering computer vision method (based on DL) called Planetoid. Results showed that a semi-supervised approach improved upon the state-of-the-art supervised techniques, and by crowdsourcing the labeling of training cases, the performance of a domain expert used to label training sets could be matched, or even exceeded.

In Zhao *et al.* [48], the author's work revolved around the identification of flood-prone areas, using a weakly labeled SVM (WELLSVM) trained using a semi-supervised approach. Maps produced by the WELLSVM were compared and contrasted to supervised alternatives using logistic regression, ANN and SVM models. Results showed that WELLSVM could better use the spatial information of unlabeled data and outperform the other methods. The authors highlighted these results, as they could potentially lead to an improvement of urban flood management.

In their work, Luu *et al.* [49] address the problem of detecting and mapping flooded areas using both optical (satellite) and hyperspectral (PolSAR) remote sensing images. By using a CNN, these two types of images allowed to identify flooded areas, particu-

larly those affected by cloud cover or irregular reflections of radar waves. Through a semi-supervised approach, expertly-interpreted radar scattering artifacts were labeled, in order to better inform the CNN model, allowing it to perform better classifications when using polarimetric data as input, aside from optical images.

The work by Croce *et al.* [55] proposes a *Generative Adversarial Network*-based (GAN), Bidirectional Encoder Representations from Transformers-like (BERT) architecture, for the semi-supervised classification of unlabeled text examples using only a very limited amount of annotated examples. This work extended the limits of transformer-based architectures (like BERT) for NLP, particularly for poor training conditions. The authors adopted adversarial training in order to overcome the limitations of transformer-based architectures, when few labeled examples are available. Results showed that by using between 50 and 100 labeled instances, good performance was obtained during several sentence classification tasks, which is a truly remarkable feat. This work proved that the proposed GAN-BERT architecture systematically improves the robustness of transformer-based architectures, while not introducing computational costs at inference time (however training costs certainly increased).

The paper by Islam *et al.* [57] compared several traditional supervised ML approaches for flood detection (MLP, SVM, CNN) to recent semi-supervised domain adaptation-based (SSDA) approaches, based on multi-modal, multi-temporal image datasets. SPOT-5 (satellite) and weather RADAR images corresponding to a flood event in the UK were used as training data. Results presented for a SSDA method using only 20 labeled data samples achieved comparable results to those achieved by the MLP, SVM, and CNN models, which made use of over 40,000 labeled data samples for training.

In the work by Yao *et al.* [61], the authors presented a semi-supervised DNN, for predicting landslide susceptibility. For this, a large spatial data set was used, where only a limited number of labeled instances was used. The DNN's performance was compared to supervised-only DNN, SVM and logistic regression models. Landslide susceptibility maps were produced for each of these models, and evaluated in terms of sensitivity, specificity and accuracy. Results showed that the semi-supervised DNN approach outperformed all other supervised models, and the authors concluded that incorporating pseudo-labeled instances for DNN applications allows to fully explore the potential of deep learning for spatial landslide modeling.

The publication by Paul & Ganju [65] revolves around predicting flooded pixels for the NASA Impact Flood Detection competition. The authors drastically improved upon previous supervised techniques, by proposing a cyclical, semi-supervised scheme with pseudo-labeling of images using U-Net (a kind of CNN) ensembles. Results obtained by this work established a new state-of-the-art for this competition using Sentinel-1 (satellite) data, achieving considerable improvements with respect to the preexisting baseline.

2.3 Deep Learning

Throughout the review process, 9 specific applications (and one literature review [44]) of supervised (or unsupervised) deep learning architectures and methods were identified, spanning applications like weather forecasting, weather event identification, flash flood prediction, flood mapping, as well as flood impact and risk identification: [50–53, 56, 59, 62, 67, 71]. Two of these DL-related publications also involved model explainability or explainable AI [50, 52].

In the article by McGovern *et al.* [50], the authors presented and compared various model interpretation and visualization (MIV) methods for machine learning, specifically targeted towards deep learning applications in meteorological domains. Image space-based methods like saliency maps, gradient-weighted class-activation maps (GRAD-CAM), and backward optimization were some of the methods explored in this work. Additionally, model agnostic methods like partial-dependence plots were highlighted as ways to provide concrete explanations on how a given predictor is important for a model (i.e. how it affects the model's output). Even though this work revolves around CNNs, the authors ascertain that the methods presented in their work can directly transferred for RNN applications.

The work by Potnis *et al.* [51] revolved around using encoder-decoder networks – also known as autoencoders–, to classify remote sensing images for urban flooding and assessing the impacted areas. This work made use of a High Performance Computing platform for training a CNN-based model known as ERFNet, using WorldView-2 satellite imagery of floods in Srinagar, India. The resulting model, was able to correctly identify and segment (segregate) flooded areas from buildings, vegetation and roads. In order to create a working dataset for the ERFNet model, a software tool markGT

was designed and implemented for efficiently annotating satellite imagery, and the creation of an urban flooding dataset.

In Saint Fleur *et al.* [52], the application of a knowledge extraction method (KnoX) over a RNN trained for predicting flash floods was presented. It must be noted that flash flood prediction was restricted to a single basin in France. By making use of the KnoX method, the most relevant prediction variables were successfully identified from the model, in order to inform and lead the future implementation of better forecasting models for the Gardon de Mialet basin. Cumulative rainfall was found to be an important predictor for models where past discharge is not used as an input, and base flow was consistently found to play a role in the prediction of flash floods as well.

The article by Wayn *et al.* [53] was the only work in the review pool which addressed numerical weather prediction (NWP). The authors exposed the development of an elementary weather prediction data-driven models, using CNNs to predict 500 hPa geopotential height values over a specific location. The 500 hPa geopotential height shows approximately how far one has to go up in the atmosphere before the pressure drops to 500 hPa (on average 5.5 km above sea level), and weather systems near to the Earth's surface, generally move in the same direction as the winds at the 500 hPa level [72]. The CNN model outperformed other types of NWP models based on persistence, climatology and atmospheric dynamics. However, the CNN model was not able to outperform a full-physics operational model. Nevertheless, the CNN approach was found to be capable of forecasting significant changes in the intensity of weather systems, and by incorporating 700hPa to 300hPa barometric data, results were shown to improve dramatically. Some of the best CNN models were able to capture climatology and annual variability of 500hPa heights, yielding realistic atmospheric forecast states at lead times of up to 14 days.

The work by Faruq *et al.* [56] explored the implementation of long short-term memory (LSTM) networks –a type of RNN– to forecast time-series of river water levels, in order to emit flood warnings for the Klang River in Malaysia. In this work, a fully-connected network with 100 LSTM hidden units was trained with 10000 data points for 100 epochs, and compared to the results produced by a Radial Basis Function Neural Network (RBFNN). Validation was performed on 2000 data points, and both the LSTM model and the RBFNN model were found to predict water levels with an extremely high coefficients of determination ($R^2 = 0.98$ and $R^2 = 0.98$, respectively). The authors reflected on how the performance offered by RBFNNs was comparable to a

much more complex LSTM model, however, they failed to discuss any strategies to mitigate overfitting.

The publication by Sankaranarayanan *et al.* [59] presented efforts for the prediction of monsoon-induced floods in India, using ambient temperature and precipitation intensity as inputs. To achieve this, the authors implemented a deep MLP, which they compared to multiple other ML algorithms like SVMs, Naïve Bayes classifier (NB), and K-Nearest Neighbors (KNN). Models were trained with data spanning over two decades, from 1990 to 2002, however the authors recognize that the exclusion of newer data is a great limitation to their approach. The DNN model was able to produce probabilities of flood occurrence with a 91% accuracy, and its precision, recall and F1-scores outperformed all other ML models (SVMs, NB and KNN).

In Hashemi-Beni & Gebrehiwot [62], the problem of mapping floods through regions covered by vegetation and below the canopy was addressed, by using CNNs and a region growing (RG) method. Relying on Uncrewed Aerial Vehicle (UAV) optical (LiDAR) images, the RG method permitted the estimation of flood extents under the covered areas, whereas the CNN model allowed to detect bodies of water and floods in the image data. Results showed that this data augmentation technique allowed to improve the flood mapping performance, and produced actionable results.

The work by Nevo *et al.* [67] involved the Google Operational Flood Forecasting System (GOFFS), which has been in operations since 2018. This paper revolved around two of the main components of the GOFFS system: 1) an LSTM network which allows the system to model and forecast river stage, and 2) a thresholding model which allows the system to compute inundation extents, which works in conjunction with a manifold model which computes the depth of the inundation extents. Particularly, the manifold model is presented in this paper for the first time, and according to the authors, presents an alternative to hydraulic modeling for flood inundation mapping. At the time of writing, this article was still in a pre-print discussion stage.

In Zhang *et al.* [71], the authors employed DL methods and weather RADAR-based rainfall data to predict inundated areas in Baltimore, MD, USA. By building a deep MLP model capable of identifying areas affected by heavy rainfall and at risk of flooding (by calculating the average height above nearest drainage), the property and demographic data of these areas was identified and analyzed, yielding an estimation of socioeconomic impact. The results shown in this work aim to provide further insight

into environmental justice, which could aid households and government agencies to improve decision-making with respect to flood risk.

2.4 Natural Language Processing

From the review pool, 10 different papers concerning NLP applications were found (aside from three literature surveys [47, 54, 58]): [34, 39, 40, 42, 46, 63, 64, 68–70]. These supervised (or unsupervised) applications included both ANN-based and non ANN-based methods for different NLP tasks such as Sentiment Analysis (SA), Named Entity Recognition (NER) and Text Classification (TC).

In Imran *et al.* [34], a dataset of word2vec word embeddings was built from 52 million crisis-related tweets, corresponding to 19 different disasters (including earthquakes, cyclones, typhoons, and floods). This human-annotated dataset provided annotations on each tweet's topic, as well as out-of-vocabulary word tagging and normalization (i.e. colloquialism, contractions and abbreviations). This work highlighted the construction process and quality control of the aforementioned dataset, and authors mention that it had been previously used to construct classifier models such as Naïve Bayes, SVMs and Random Forests.

The study presented by Tkachenko *et al.* [39], employed a Deconstructed Cascade Correlation Matrix (DCCM) –a type of neural network architecture– built using polysemous (having multiple semantics meanings) image tags. The main objective was to predict the dependency of risk-signaling tags (i.e. flood, flooding, floodplain, *etc.*) from a set of specific input tags present in the data set's images (i.e. nature, landscape, river, water, *etc.*). These relationships were explored from a time-dependence perspective, and the appearance of tagged images in the Flickr social network was compared to flood reports emitted by official and secondary weather report sources between 2004 and 2014. The results presented demonstrated that flood-related tags tended to correlate with hydrologically themed tags, present in the metadata of pictures on social media.

In the publication by Wang *et al.* [40], CNN-based NLP was applied to Twitter data to extract specific locations mentioned in tweets related to urban flooding (named entity recognition), which were later geocoded (the act of transforming the name of a

place or location, into actual georeferenced coordinates) using a search engine API. Additionally, mentions of water depth were extracted from tweets. Subsequently, CNNs were used to classify crowdsource-labeled pictures of floods, rivers, and other bodies of water into "flooding" or "not flooding" categories. Using both the twitter-derived and the image datasets, the authors were able to construct a map of road closures related to the classified crowdsourced images. Also, two case studies were presented where NLP applications through CNNs helped improve urban flood monitoring.

In their article, Barker *et al.* [42] developed a national-scale twitter data mining pipeline for improving situational awareness during flood events across Great Britain, mixing grounded environmental data sources and retrieving geodata from social media. First tweets were retrieved from at-risk of flood areas (based on the national flood warning and risk levels), and then these were classified into "relevant" (flood-related) or "irrelevant" tweets. This was done using a word embedding model (word2vec) and a logistic regression model. The authors demonstrated the national-scale social geodata pipeline using over 400,000 georeferenced tweets obtained between Jun. 20th and Jun. 29th of 2016.

The work by Qian *et al.* [46] explores a CNN approach to perform sentiment analysis on weather-related social media microblog posts (i.e. tweets), in order to classify them into 14 distinct weather event categories. Models were trained using manually labeled data, which was first scraped from the internet using a web spider module (a program which "crawls" the Web, in search for specific information). By using a word2vec word embedding representations product of a CNN, which were fed to an LSTM network, the extracted tweets were classified. The article presents a brief analysis, and overall favorable performance was achieved for predicting all 14 classes. In the conclusions, the authors acknowledge the difficulty of labeling real-world data, and suggest that future work makes use of unsupervised methods for labeling data (i.e. a semi-supervised approach).

The paper by Lai *et al.* [63] exposes a CNN-based NLP model for named entity recognition, which extracts information from online newspaper websites, to identify historical weather and flooding events. The authors report this model is apparently the first of its kind, and is able to successfully extract detailed flooding information, risk and impact data for all the conterminous Unites States (CONUS). The data extracted from news articles includes names and locations of street closures due to flooded roads, cost data on risk reduction project aimed at mitigating risk of flooding, as well as im-

pacts and impacted locations associated to flooding events. In total, over 27,000 street closures, over 54,000 risk reduction projects, and over 430,000 storm events were identified by this work, including details of project costs and flood cause.

In Nair *et al.* & Palkar [64], the authors looked to classify tweets related tot he Kerala floods in India, making use of several transformer-based NLP models like BERT, XLNet and Ernie2.0. which were presented and constrasted. Around 4,500 tweets were collected from the social network using well-known hashtags related to the flood events. These were subjected to cleaning and manually labeled to conform the working data set. Four different labels were used to categorize the topic or intention of each tweet: appreciation/donation, help, destruction/loss, and political news. Deep models were built for each of the transformer-based methods (768 hidden layers), and all three were found to produce highly favorable results. However, Ernie2.0 was found to produce the highest Mathew's correlation coefficient, indicating the better outcome of the three.

The successive works by Purwandari *et al.* [68, 69] revolved around the use of SVMs to classify weather-related tweets, and produce maps with the associated weather features described within them. In [68], the authors exemplified and compared the results of classifying tweets using SVMs, Multinomial Naïve Bayes and Logistic regression, yielding that SVMs was the best performer. This work was extended then in [69], where tweets were used as a "social sensor" for weather in Indonesia, using text classification. Reported accuracy for this SVM-based system was 93%, and led to the implementation of a weather visualization tool based on weather-related tweets.

Lastly, the publication by Sattaru *et al.* [70] focused in the use of geocoding, Naïve Bayes classifiers and sentiment analysis (VADER) to find flood-related tweets. The information extracted from these tweets were used to provide an additional layer of information, as well as new data elements for a web monitoring portal, centered around flooding hotspots for the Chennai region in India.

* * *

While an extensive literature review was performed on applications related to machine learning and flooding, no bodies of work were identified, where authors attempted to formally produce a historical flash flood impacts dataset, specifically with an operational outlook. Notable mentions of work related to the present work the following

works: Terti *et al.*, which focuses on the integration of physical and social dynamics leading to model forecasts of circumstance-specific human losses during a flash flood, where a random forest model was trained to assess the likelihood of fatality occurrence for a given circumstance as a function of representative indicators [73]; Diakakis *et al.* [74], where a flash flood impact severity scale was proposed, based on flash flood events over Greece, which categorized impacts into four different impact types, and each type consists of ten distinct levels of impact; Lastly, Bucherie *et al.* present a comprehensive dataset of flash flood events over Ecuador between 2007 and 2020, which incorporated multiple metrics including a Flash Flood Susceptibility Index (derived from geomorphological characteristics) and a Flash Flood Confidence Index (based on how reliable an event's description is [75])[76].

Most of the flood reports other works used came from social media platforms, mostly X (the platform formerly known as Twitter) [35] [41] [34] [40], and were generally employed to perform validation on other types of products or models. And while multiple works built models predicting specific natural hazards and their impacts [63] [71] [51] [65], none were oriented towards real-time applications for operational flash flood forecasts. Additionally, no works relying on the classification of hazardous weather reports using large language models were found. Most of the works found which deal with flooding and flood impacts, revolve around studying flood inundation areas, building flood risk models from Radar and Satellite images, or classifying past satellite data into binary flood/no-flood classes [51] [71] [65]. However, none of the works surveyed directly aim to characterize and predict flash flood impacts, with aims of providing decision support systems for operational use.

In the present work, pre-trained language models, and large language models will be explored as a viable alternatives to manual expert classification for building a comprehensive historical flash flood impact dataset, which then can be used to develop experimental machine learning models capable of annotating flash flood forecast with anticipated impacts.

Chapter 3

Methodology

As previously stated in Section 1.4, two main challenges must be overcome to address this dissertation's main research question. First, historical flash flood reports must be classified into impact categories, with only a reduced set of labeled examples available. This task is clearly a NLP-task of text classification. Making use of a pre-trained language model could be beneficial, as shown by [58], in order to establish a feasible LSR classification model using the reduced set of labeled instances. Once this text classification model has been trained and tested to produce sound results, it can be the used to construct a surrogate, historical (retrospective) observation data set of impact-classified flash flood reports. Conversely, the challenge could also be addressed by establishing a natural language-based methodology (prompt instructions) to classify unlabeled historical reports, through the use of a pre-trained Large Language Model. Said LLM could be prompted to follow said methodology to classify each historical LSR (both labeled and unlabeled) into specific impact classes. This classification could then be assessed by contrasting the expert classifications with the ones provided by the LLM.

Secondly, this surrogate observations dataset can be used in conjunction with historical archived FLASH product outputs, geomorphology, and vulnerability layers to construct a ML model which predicts flash flood impact probabilities, given predicted streamflow responses as inputs. Given the spatiotemporal dependence of rainfall and flow responses, approaches that can deal with spatial features, as well as temporal features should be explored. While more common deep learning architectures like convolutional neural networks and recurrent neural networks could help address this spa-

tiotemporal dependency, novel transformer-based approaches are available to tackle complex spatial and temporal characterization of feature inputs. Once this impact prediction model has been trained, tested and proved to operate satisfactorily on historical events and data, it could enable the future implementation of a real-time experimental ML system capable of forecasting flash flood impacts.

Regarding the first challenge of construction of a historical flash flood impacts dataset, the present chapter will address the development of both aforementioned strategies: 1) the use of pre-trained language models to train an LSR classification model, and 2) the use of a language-based classification framework through a pre-trained LLMs to classify all LSRs. These will be detailed in Sections 3.1 and 3.2 respectively, and their associated results will be shown in Chapter 4, in Sections 4.1 and 4.2 respectively. With respect to the second challenge of training a ML model to forecast flash flood impacts, this chapter will address the implementation and testing of three distinct approaches to gradually explore how to model the spatiotemporal dependency of the input data. This work will be detailed in Section 3.3, and its respective results will be shown in Chapter 4, in Sections 4.3.

3.1 BERT for IBW LSR Classification

As defined in Section 1.3.1, LSRs are a type of manual observation collected and archived in near-real-time by the NWS, and are filed following the issuance of warning products. These reports include a wide variety of information about an event, which includes the hazard type (*e.g.* flash flood, hail, wind, winter weather, *etc.*), the event's location (latitude and longitude), a source for the report (*e.g.* emergency services, emergency manager, social media, news, *etc.*), and an event's description (referred to as 'remark'), among others. Some reports include a 'magnitude' field (*e.g.* hail size, wind speed, snowfall), but for the specific case of flash flood LSRs, this field is left blank. LSRs are normally used for event validation, which help NWS forecasters evaluate the effectiveness and accuracy of their issued flash flood warnings. LSRs are also the basis for StormDat reports (see Section 1.3.1), which are in-depth follow-up reports on a given event, that include thorough surveys and dollar-amount assessments of damages to property and goods (e.g. crops) of the impacted areas.

The NWS transitions to an impact-based format for flash flood warnings in late

2019, aiming to provide detailed information about the hazard, including an *impact narrative* and a *damage threat tag*. Specifically for the latter, the initiative proposes the inclusion of one of three impact severity classes for each recorded event: `base`, `considerable`, and `catastrophic`. We will refer to this three class system as the *Impact-Based Warning* (IBW) framework. This new warning format aims to determine and inform which calls to action to enact as appropriate responses to each event's anticipated impacts (*e.g.* trigger Wireless Emergency Alerts (WEAs) or not).

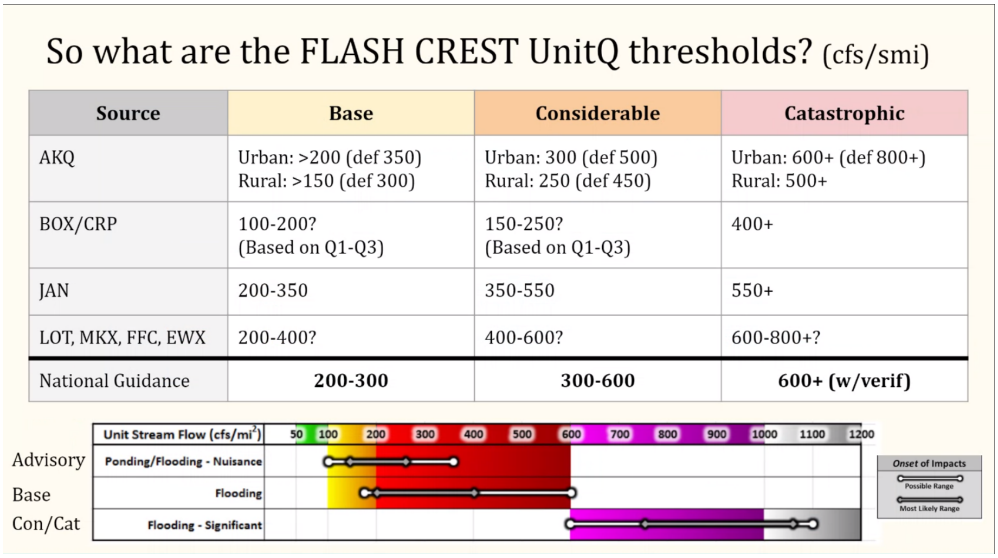


Figure 3.1: Impact-Based Warning guidance on how to evaluate FLASH CREST UnitQ values with respect to three impact categories. Excerpted from slides courtesy of the National Weather Service Warning Decision Training Division.

Figure 3.1 shows an example of guidance to NWS forecasters, on how to apply and interpret thresholds on FLASH unit streamflow, in order to determine which IBW category to use for a given report. Even though 'National Guidance' is provided to assess IBW categories at the CONUS scale, note how for different weather forecasts offices (WFOs) listed on the 'Source' column, there are different ranges for each of the IBW categories. This is due to the fact that each WFO is in charge of different geographic locations distributed around the CONUS, which will respond differently to similar values of FLASH products (not just UnitQ). Uncertainty is also expressed in the lower portion of the diagram, as ranges of UnitQ values overlap when deciding between `considerable` and `catastrophic` labels. Similar guidance has been issued for other FLASH products such as Average Return Intervals (ARIs) and QPE-to-FFG Ratios (see Section 1.3).

As expressed in Section 1.4, the motivation behind the present work stems from a NOAA-funded project which aims to *build a novel experimental tool, that can distill information from MRMS-based FLASH products, to aid forecasters in the issuance of probabilistic impact-based flash flood warnings*. In order to fulfill this goal, two main milestones must be accomplished. Firstly, a dataset of flash flood events with their respective impact classes must be constructed, since a dataset of this kind is not only unavailable for this purpose, but currently is non-existent. This dataset would enable us to use LSRs as surrogate impact observations for flash flood events. Secondly, having access to a comprehensive impact-based flash flood event dataset, a prototype model can be trained to predict the impact class of flash flood forecasts over the CONUS. The present chapter will specifically focus on documenting initial efforts made in attaining the first milestone. The present chapter will address the use of a pre-trained deep learning language model (BERT, see Section 1.2.3) to train an LSR classification model based on a subset of expertly-classified LSRs. Having access to expertly-labeled data, a classification model would enable the retrospective classification of additional unlabeled historical LSRs. A flowchart detailing the scope of this section can be seen in Figure 3.2.

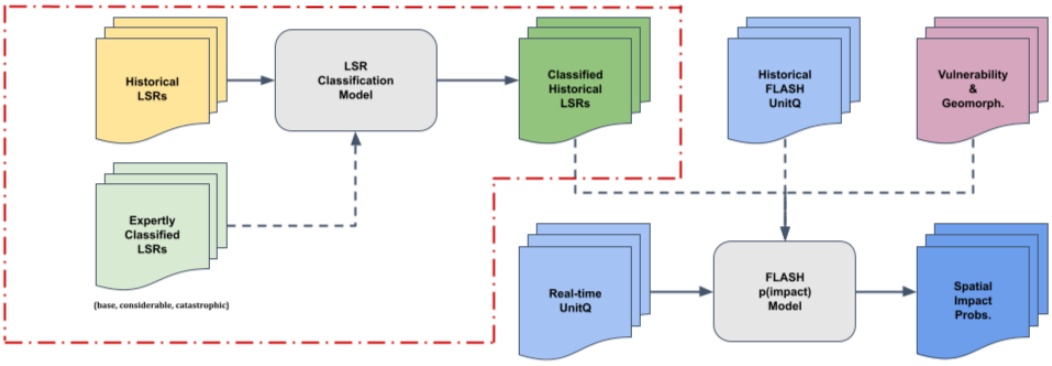


Figure 3.2: IBW-based workflow for addressing the research motivation of the current work. Notice the delineated region of the initial stage, which highlights the scope of the present section.

A dataset comprised of 663 LSRs was obtained, where each report was classified by experts (NWS forecasters as part of an IBW/FLASH focus group) into one of the three IBW categories: `base`, `considerable`, and `catastrophic`. The reports included in this expertly-classified dataset span a time period between May of 2018 to June of 2020, and are located along the coast of the Gulf of Mexico in the South Eastern US. Figure 3.3 shows their spatial distribution.

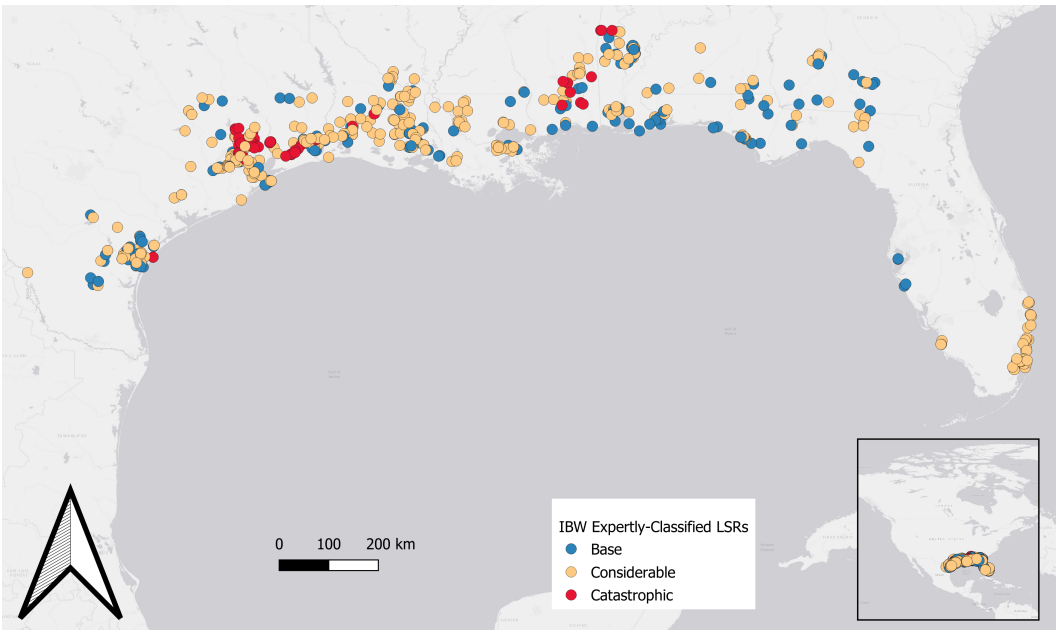


Figure 3.3: Visualization of the geographical distribution of the 663 expertly-classified IBW LSRs, colored by impact class.

Among these reports, 183 were classified as `base` events, 423 were classified as `considerable` events, and 57 were classified as `catastrophic` events. A bar plot with these proportions is shown in Figure 3.4. This reflects a severe class imbalance among the three classes, particularly towards the `catastrophic` class (which is expected, given the infrequent nature of these events). However, it must be said that even though from an operational perspective it is expected that `base` events are to be the most frequent, it was surprising to see that the most frequent class among this small dataset corresponded to `considerable` events.

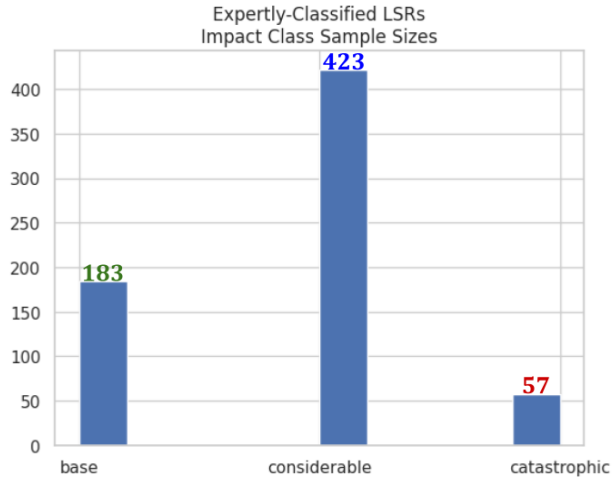


Figure 3.4: Bar plot of IBW class counts, reflecting the severe class imbalance towards the `considerable` class.

A second dataset of historical unlabeled LSRs has been retrieved from the experimental product archives of NSSL's Stormscale Hydrology Group. A total 94,847 LSRs is available between January 2005 and June 2022. However, for the scope of this work, a subset of 22,329 historical unlabeled LSRs between April 2018 and June 2022 was contemplated. These reports are deemed *unlabeled* since they don't have an associated IBW class, or any other kind of label which can help systematically classify them into varying levels of impacts or severity. The locations of most of these 22,329 LSRs is shown in Figure 3.5, however it must be noted that within these historical reports, there are a number of them which correspond to other non-CONUS territories such as Hawaii, Puerto Rico, and Guam.

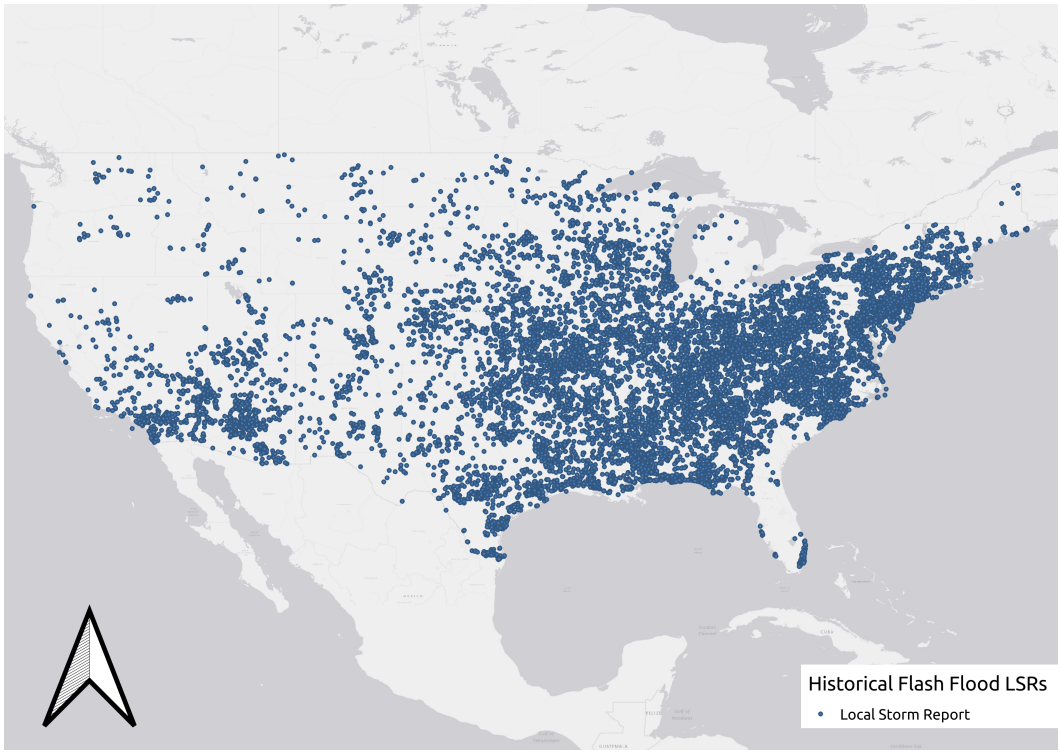


Figure 3.5: Visualization of the geographical distribution of the 22,329 historical unlabeled LSRs over the CONUS.

During this exploratory phase of the present work, expertly-classified reports and pre-trained language models were relied upon to create an LSRs classification model. Initially, models that relied exclusively on the LSR remarks were explored yielding insufficient performance due to the small training sample size. In hopes of enriching this small dataset, further impact data from StormDat reports (see Section 1.3.1) was incorporated into the training dataset with hopes of improving model performance.

3.1.1 LSR Event Matching to FLASH Max UnitQ

In order to assess how good the expert's classification is, as well as the LSR classification model's, a baseline derived from some certain notion of 'ground truth' must be established. For this purpose, FLASH's maximum unit streamflow (MaxUnitQ) was selected as an objective measure with which to compare and contrast the severity of the reported events, as reflected by their assigned impact classes.

To enable direct comparison between an event's class label and a specific measure of

MaxUnitQ, flash flood predictions across time and space corresponding to each event must be reduced to a single quantity. This means that systematic definitions for both an event's duration and it's area of influence must be defined, as well as a reduction mechanism to produce a single value. These criteria will allow to objectively *match* LSR reports to specific FLASH product outputs in a consistent way. For this case, since LSRs are issued while an event is typically underway (as impacts are being observed), an event's duration is assumed to be captured by FLASH within six hours of any given report (a sequence of 60 MaxUnitQ output). Similarly, an event's location is assumed to be captured by the pixel that corresponds to a report's location, or by any pixel adjacent to it (effectively, a $\sim$1km^2 radius around the report's location). Finally, the maximum MaxUnitQ value present across the spatiotemporal search window [$\delta_t = -6h$, $\delta_r = 1km$] will be considered to be representative of a reported event's magnitude.

With the help of Dr. Humberto Vergara (a Research Scientist in NSSL's Stormscale Hydrology group at the time) a Matlab program has been implemented to ingest and process a Comma-Separated Values (CSV) file containing Flash Flood LSRs. For each of the LSRs the program queries an archive of FLASH products (GeoTIFF rasters), finds the 36 MaxUnitQ files corresponding to the time frame starting six hours prior to the LSR's report (6 files per hour), and extracts for each file the MaxUnitQ values present within a 1km radius of the grid cell corresponding to the report's location. From this 3x3x36 matrix of MaxUnitQ values, the maximum value is identified, extracted, and associated to each of the reports present in the CSV file. This extraction was performed for both the 663 expertly-classified LSRs, as well as for the 22,329 historical unlabeled LSRs. The general source code for this implementation can be found in the Code Appendix Listing C1.

3.1.2 Remark-only BERT Models

To train an LSR classification model, a compound deep learning architecture was explored. The main idea of this design was to rely on a pre-trained BERT model to ingest and extract meaning from the LSR remarks, and then pair BERT's 'interpretation' of a report's narrative with the expert's assigned class label to train a fully connected neural network for classifying these processed remarks. The pre-trained BERT model used for building the LSR classification architecture was of the "Large" variety, trained on the English language for uncased text (all text lower-case) with the following architecture:

24 layers of transformer blocks, followed by 1024 hidden layers, and 16 self-attention heads (340 million parameters) [16]. First, a preprocessor converts an LSR's remarks into token sequences in the format accepted by pre-trained BERT models. Next, the pre-trained model encoder takes these token sequences and produces a latent representation (embeddings) based on its interpretation of the text remarks (a feature vector of size 768). This embedding representation is then fed to a dense network classification head (single layer of 128 neurons), along with the LSR's IBW class assigned by the experts. Ultimately this dense network classifier is trained to predict the IBW label associated to each report, based solely on BERT's latent representation of an LSR's remark. Figure 3.6 details the model's architecture.

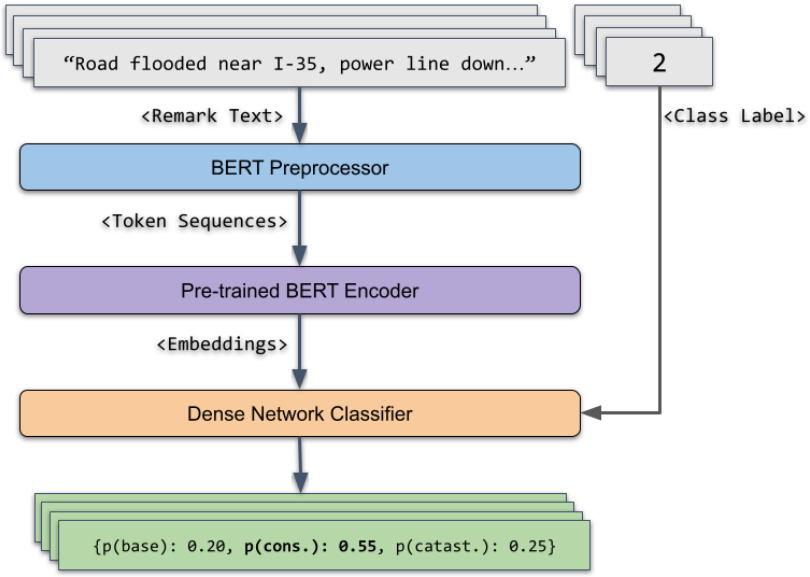


Figure 3.6: Architecture of the implemented pre-trained BERT-based deep learning LSR classifier.

The machine learning architecture described in Figure 3.6 was implemented in Python [77] using TensorFlow [78], and both the BERT preprocessor and the pre-trained model were obtained from TensorFlow Hub [79] [80] (presently acquired by Kaggle, and as of Nov. 15, 2023 called Kaggle Models [81]).

This architecture was trained in two ways: 1) using all 663 expertly-classified reports, and 2) using a balanced data subset where each class was represented by 57 instances for a total of 171 expertly-classified LSRs. Results for these models are presented in section 4.1.2 and 4.1.4.

3.1.3 LSR Event Matching to StormDat Data

A subsequent approach for an LSR classification model is explored following the baseline established in the previous subsection. A data scarcity issue has been identified for training a DL model using the 663 expertly-classified reports as provided, as well as a severe class imbalance which is dominated by the `considerable` class. A data enrichment approach is explored with hopes of improving the data's usefulness, since increasing the number of expertly-classified samples is not an option.

StormDat reports, as described in Section 1.3.1, are post-facto reports based on detailed impact surveys, which are based on LSR reports. StormDats contain much more information regarding the impacted area and the event's severity, but are typically generated days to weeks after an event has occurred.Additionally, each StormDat report includes an event narrative, and an episode narrative. Nevertheless, this insightful information can be paired back to the expertly-classified reports in order to better inform our model on how to classify them into IBW impacts. Specifically, event impact measures like the number of direct and indirect injuries, direct and indirect fatalities, property damage costs, and crop damage costs. The assumptions here are: 1) damage costs should correlate proportionally to event impact classes, and 2) fatalities and injuries should correlate to most `catastrophic` and some `considerable` events. This idea was derived from previous efforts led by researchers at NSSL's Stormscale Hydrology group, led by Dr. J.J. Gourley, where it was shown that flash flood severity simulations are proportionally related to economic impacts observed over the US. In this work, FLASH Average Recurrent Intervals were shown to be correlated to increasing dollar amount ranges of StormDat Property Damage reports. Figure 3.7 shows a box plot exploring this relationship. These new features could help improve our model's skill, as we could provide additional quantitative information associated to each LSR remark for our model to draw upon.

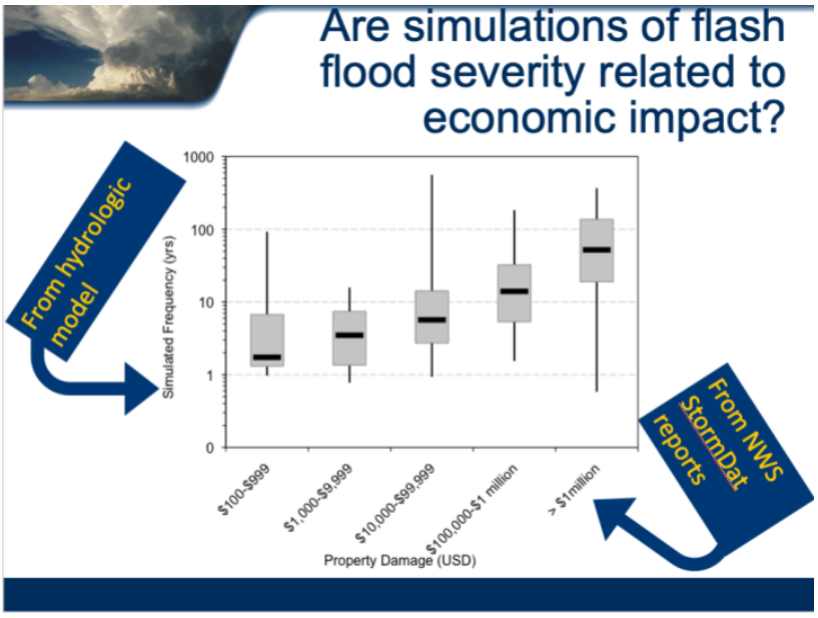


Figure 3.7: Box plot analysis showing the relationship between increasing ranges of property damage, associated with higher values of FLASH ARIs. Slide courtesy of Dr. J.J. Gourley, NSSL's Stormscale Hydrology group.

In order to match StormDat impact features to our collection of both expertly-classified and historical unlabeled reports, spatial and temporal assumptions had to be made like in the case of Section 3.1.1 for matching UnitQ values to LSRs. For this case, a three hour buffer before and after the StormDat report's start and end timestamps were contemplated. Also, increasingly relaxed (and uncertain) spatial matching criteria were used to match all LSRs to StormDat reports. The first level of spatial matching –and the most certain– tries to match an LSR's location (latitude and longitude), which lies within the boundaries of the reported polygon of a StormDat report. The second level of spatial matching looks for LSRs that occur in the same state and county as reported in the StormDat report. Thirdly, the next criterion looks for LSRs which are within a 25km radius from the StormDat polygon's centroid. Lastly, the most uncertain spatial matching criterion looks for any LSRs matched in time, whose county name matches the county of the StormDat report. This heuristic approach was applied over an archive of StormDat reports spanning the period between 2006 - 2022, and is described in Algorithm 3.1 below.

The distance of 25km from the polygon's centroid is meant to represent an spatial measure of generalized county extents. It was determined by performing a sensitivity analysis based on taking increasing distance thresholds which contemplated the

Algorithm 3.1 StormDat Matching Heuristic

```
for StormDat in STORMDATS do
    for LSR in LSRS that match the StormDat's time window ±3h do
        if LSR location is within StormDat polygon then
            LSR and StormDat matched by: point-in-polygon
        else if LSR location is the same state and county as the StormDat report then
            LSR and StormDat matched by: state-county
        else if LSR location is within a 25km radius from StormDat centroid then
            LSR and StormDat matched by: distance-25km
        else if LSR location is the same county as the StormDat report then
            LSR and StormDat matched by: county
        else
            LSR and StormDat are not matched
        end if
    end for
end for
```

squared root of the area of the largest county in the CONUS (San Bernardino County, CA, USA 51,936km^2) as the largest feasible cutoff (228km).

3.1.4 LSR + StormDat BERT Models

After matching and extracting StormDat impact data onto our expertly-classified LSR dataset, a new LSR classification model is trained using a modified architecture of the previous model (See Figure 3.6). The data obtained from the StormDat reports is represented in numerical quantities, which in the case of injuries and fatalities signify counts of people, while for crop and property damages these numbers represent dollar amounts. Since we are relying on a pre-trained BERT model, which exclusively understands language, the model's data loader needs to be modified. In this case, we need it to read in the StormDat's event narrative, episode narrative, and numerical variables for each LSR (in addition to its remark), format the text as lower-case, express the numerical quantities in a more descriptive way, and then append them to the original text of the LSR remark. This means that an LSR with 3 indirect injuries, and $1,000 in property damage would have the text appended to its respective remark, as shown in Listing 3.1.

```
<ORIGINAL LSR REMARK> \n
<STORMDAT EVENT NARRATIVE> \n
<STORMDAT EPISODE NARRATIVE> \n
direct injuries: 0. indirect injuries: 3. \
direct fatalities: 0. indirect fatalities: 0. \
property damage: $1,000. crop damage: $0.
```

Listing 3.1: Structure of enhanced LSR-StormDat remarks, consisting of appended LSR remarks, StormDat narratives, and *textualized* event impact quantities.

This way BERT can now potentially ingest an enhanced LSR-StormDat remark which contains additional descriptions from the StormDat report, as well as quantitative information that describe the impacts the flash flood event resulted in, all expressed in natural language. The resulting modified data ingestion architecture is detailed in Figure 3.8.

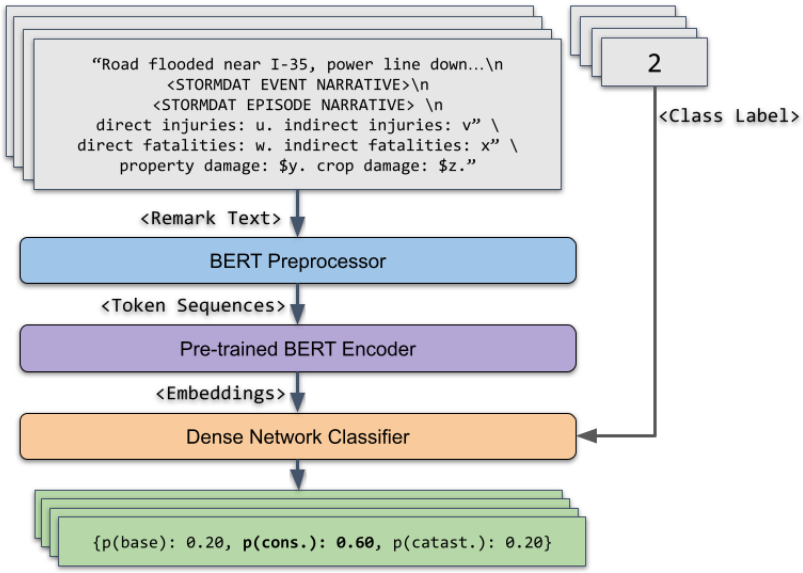


Figure 3.8: Architecture of the implemented pre-trained BERT-based deep learning LSR classifier, which ingests enhanced LSR-StormDat remarks.

3.2 ChatGPT-based Flash Flood Severity Index Dataset

Attempting to overcome the shortcomings of relying on the severely limited collection of expertly-classified IBW LSRs, a new approach was taken towards fulfilling the first objective of this dissertation work. To create an impact-annotated historical flash flood report dataset, based on the 22,329 historical unlabeled LSRs, a systematic way of classifying these reports into impact categories is needed. As detailed in Section 1.3.2, even though the IBW framework allows for the classification of flash flood forecasts into three impact classes based on their streamflow values, this guidance is uncertain when differentiating between the two top classes (these value ranges also vary widely by location across the CONUS, see Figure 3.1). Thus, relying solely on a UnitQ based characterization of impacts is not feasible using IBW guidance. Therefore, this task must be approached using a framework which can translate specific impacts detailed in an LSR's remark (natural language) into impact labels: opposed to IBW's streamflow-based guidance, FFSI framework fits this need precisely. FFSI's class descriptions are expressed in natural language, objectively, concisely, and are consistently stratified in increasing levels of impact. And even though FFSI was originally conceived as a post-facto assessment tool, given the nature of its construction, is now a perfect candidate to be used to retrospectively classify LSRs.

Being able to rely on a language-based framework for classifying LSRs based on their remarks (FFSI), the problem now revolves around how to systematically assess and evaluate these report. Given the very large volume of flash flood LSRs we have at our disposal (22,329 for this study, but over 95,000 in existence between 2005-present), relying on human experts to perform this work would require great amounts of time and resources. However, with the recent advent of widely available pre-trained Large Language Models (like OpenAI's ChatGPT), the opportunity to leverage this type technology to address repetitive language-based tasks has been identified. This is particularly desirable, since these publicly-available models offer access through Application Programming Interfaces (APIs), which make it convenient to write programs for issuing large quantities of requests (or prompts) to, and receiving responses from these models.

Once the LSRs have been classified into FFSI classes, the performance of this classification will be examined. A natural path for this will be to re-classify our IBW expertly-classified dataset into FFSI classes. Then, by comparing the distributions of each re-

ports corresponding UnitQ with the class distribution each impact framework, objective comparisons between the two frameworks will be drawn and analyzed. With this in mind, Section 3.2.1 will provide an overview on how the previous UnitQ matching heuristic presented in Section 3.1.1 was revisited and improved, in order to provide a more comprehensive look at each event's UnitQ characteristics across a systematically-defined spatiotemporal event scale. Results of this new UnitQ extraction method will be shown in Chapter 4.2 Section 4.2.1.

Section 3.2.2 will detail how both a software framework and *textualized* FFSI definitions have been implemented in order to prompt ChatGPT to provide probabilistic impact classifications for both the expertly-classified and the historical unlabeled LSRs. Results on this ChatGPT-based FFSI classifications will be presented in Chapter 4.2, Section 4.2.2.

3.2.1 FLASH Product Moment Extraction

As described in section 3.1.1, singular MaxUnitQ values for both the expertly-classified and the historical unlabeled LSRs were extracted. This was done by contemplating a spatial search window of one grid cell ($\sim$1km), and a temporal search period over six hours before each report ($\delta_t = -6h$, $\delta_r = 1km$), and extracting the maximum MaxUnitQ value found within those search parameters. While this method seemed to provide the previous BERT-based efforts with a sufficient measure of 'ground truth' with which to compare model results, a more meaningful representation of each event can be achieved by 1) extracting values for both MaxUnitQ and MaxARI products, and 2) extracting statistical distribution moments for each product, and their intersections, instead of singular product measures. Additionally, the spatial search window has been increased from one to a four kilometer radius, recognizing that the location where one may observe the highest MaxUnitQ activity many not exactly correspond to where the impacts were reported , ($\delta_r = 4km$). Furthermore, while a total temporal search period of six hours is maintained, this time around the search procedure will contemplate the past three hours before, as well as the following three hours after an event was reported ($\delta_t = \pm 3h$). This should help account for time uncertainties in both FLASH's response to an event, as well as in LSR issuance and report times (e.g. a reported event of a given magnitude, may grow to be more intense or weaker over time). Figure 3.9 shows a general overview of this new method for extracting and matching FLASH

product data to flash flood event.

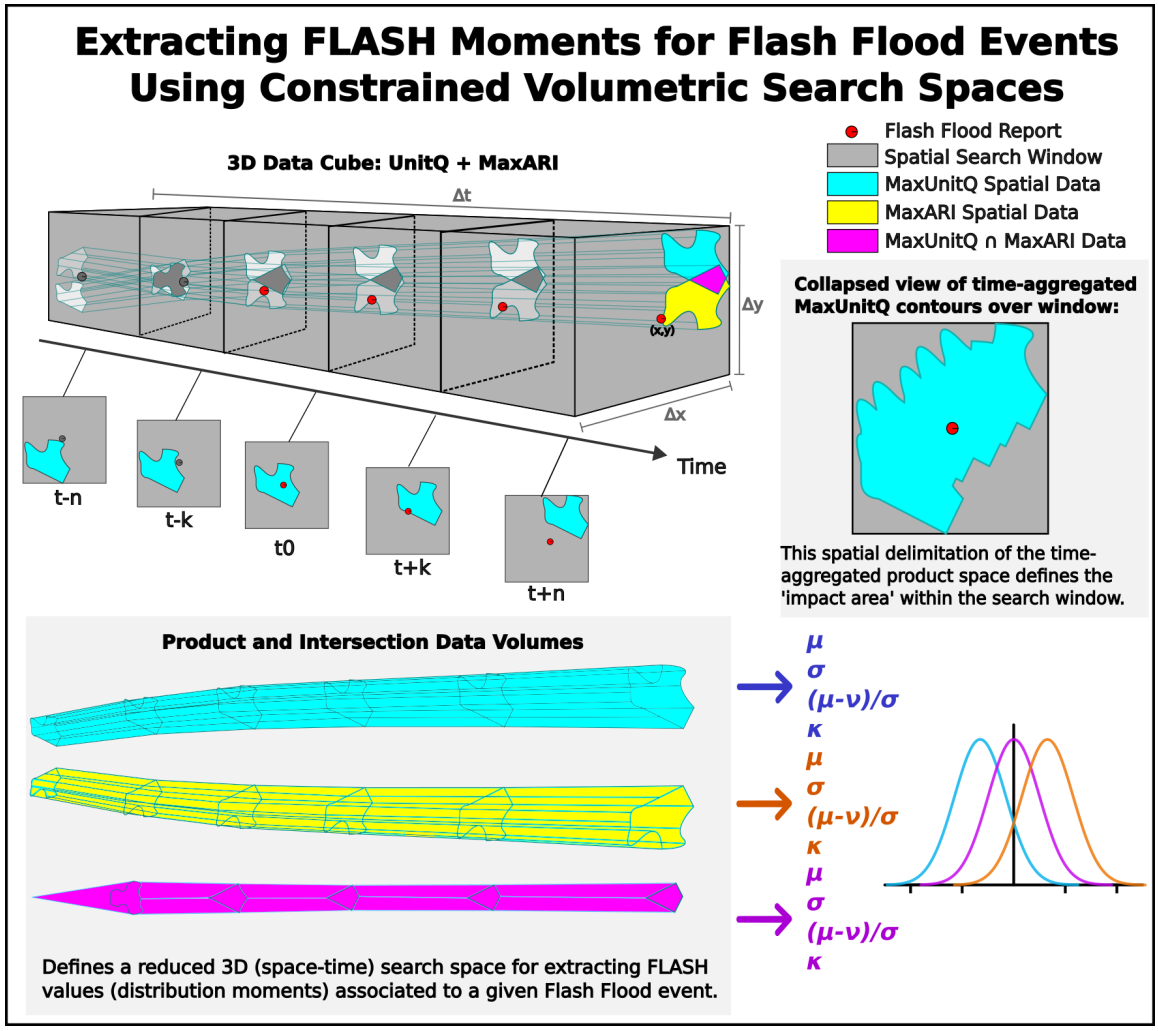


Figure 3.9: Depiction of the FLASH product statistical moment extraction process. For the present work $\delta_t = \pm 3h$ and $\delta_r = 4km$ were used, which means that $\Delta X = \Delta Y = 9$, and $\delta_t = |\pm 3h| = 6h$. Note that only one intersection is portrayed above, while both MaxUnitQ∩MaxARI and MaxARI∩MaxUnitQ were contemplated. Additional to the first four statistical moments, quantiles Q90, Q95 and Q99 were extracted, as well as the MAX for each search volume.

While the search window systematically constrains our possible search space for each event, the behavior of the FLASH products and their intersections over time, constitute a reduced 3D search space (compared to the 3D search window domain) in which only the event-related behavior is captured. The reasoning behind including MaxARI data in addition to MaxUnitQ, is that while MaxUnitQ represents the flood response component of each reported event, MaxARI provides a representation of the

precipitation which has caused the flash flood event being reported. While these event representations rely on a systematic approach to defining their spatial and temporal extents, this allows us to treat all flash flood event scales as comparable units in exchange for some uncertainty. Not only are statistical moments being extracted from these individual products, but also for the volumetric (spatial + temporal dimensions) intersections of MaxUnitQ∩MaxARI and MaxARI∩MaxUnitQ. MaxUnitQ∩MaxARI represents MaxUnitQ values which are cropped by an overlapping MaxARI spatial field, while MaxARI∩MaxUnitQ represents MaxARI values which are cropped by an overlapping MaxUnitQ field. If the data fields don't overlap at any given time step, the values for MaxUnitQ∩MaxARI will be exactly the same as MaxUnitQ, and those of MaxARI∩MaxUnitQ will be the same as MaxARI. Note that these intersections were also performed in the all possible combinations, since it is feasible that at any given time step only data is found for one of the products, leading to one of these cases to result in null values if the first operand in the intersection is `null`.

For each flash flood event's report, once the MaxARI, MaxUnitQ, MaxUnitQ ∩ MaxARI, and MaxARI ∩ MaxUnitQ spatiotemporal search volumes are established within the δ_t and δ_r boundaries, the first four statistical moments are calculated from each reduced 3D search space: mean, variance, skewness, and kurtosis. These four moments provide us with a comprehensive overview of the distribution of values found in each of the search volumes, and serve as a proxy representation for every event. Additional to these extracted moments, the maximum value, as well as quantiles Q90, Q95, and Q99 were also calculated, which provide us with a sensible look at the behavior of the most extreme values as they approach the MAX. By including these moments and quantiles, we can now interrogate the overall behavior of each event's data, as well as the variability at the higher-end of the distribution, in order to better assess the categorization of each event into impact classes.

The new modular python program, based on the one previously implemented for matching UnitQ FLASH outputs to LSRs (Section 3.1.1, was extended to include perform product aggregations and calculations described above.

For each of the LSRs the program queries an archive of FLASH products (GeoTIFF rasters), and for each FLASH product requested (MaxUnitQ and MaxARI) finds the 36 MaxUnitQ files corresponding to the time frame spanning three hours prior to the LSR's report (6 files per hour) and extending to three hours after the report. Having found the files for each product, it crops the product data (which is originally at the

CONUS scale) to the extents defined by the LSR location and the spatial search window radius. Thus, the program constructs the four aforementioned 3D search volumes by taking the non-empty values found for all time steps in the search domain: MaxARI, MaxUnitQ, MaxUnitQ∩MaxARI, and MaxARI∩MaxUnitQ. For each of these, it then calculates and extracts the first four statistical moments, the Q90, Q95, Q99 quantiles, and the maximum value. Lastly, it writes these newly extracted values with their associated original LSR data into a copy of the original LSR CSV file. This extraction was performed for both the 663 expertly-classified LSRs, as well as for the 22,329 historical unlabeled LSRs. The general source code for this implementation can be found in the Code Appendix Listings C2, C3, C4, and C5.

3.2.2 FFSI V1

Textualization literally means *to render something as text*, but in our case specifically we use the term to imply the re-expression of simple ideas (i.e. a list of impacts and impact classes) as complete, self-contained conditional statements that relate these ideas to a specific context (i.e. a list of conditional statements that relate impacts to specific impact classes). Generative pre-trained LLMs like ChatGPT can not only deal with the context of language, but also the specificity of what is being said. Thus, in order to make use of the ChatGPT API to classify LSR remarks, the FFSI impact definitions [32] (see Table 1.4) need to be transformed into an actionable prompt that the GPT model can interpret as a task, and that provide it with clear instructions to follow. This can be achieved by transforming each FFSI class' impact descriptions (which is a list of specific impacts) into a *textualized* conditional statement. Plainly said, we can *ask* ChatGPT to assign each class to an LSR only if the remark mentions any of the associated impacts. Figure 3.10 provides an overview of the workflow of using this textualized FFSI definitions and the ChatGPT API.

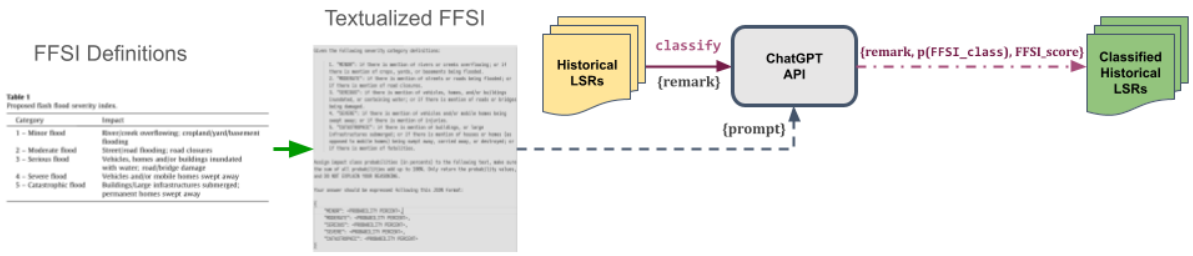


Figure 3.10: Flowchart that describes the process of using textualized FFSI definitions and the ChatGPT API to classify LSR remarks into impact classes.

Furthermore, we can also request that ChatGPT not only assigns LSRs a given class, but instead assigns a vector of joint probabilities for each of the classes. Lastly, we can also request that ChatGPT provides these outputs in a computer-friendly format like JSON, structuring labels for each class with the probability values associated for each class. Listing 3.2 describes the prompt used to interact with the ChatGPT API, which incorporates a) the textualized FFSI definitions illustrating how to interpret each LSR remark, b) a task to assign joint impact probabilities to the classification, and c) a task to format the responses we will receive from the API. Notice that explicit requests are to be made in order for ChatGPT not to provide an explanation of its reasoning; even though this is usually a desirable feature, at the time of implementation, it only makes processing the model's output much more irregular and difficult to parse.

```
Given the following severity category definitions:

  1. "MINOR": if there is mention of rivers or creeks
   ↪  overflowing; or if there is mention of crops, yards, or
   ↪  basements being flooded.
  2. "MODERATE": if there is mention of streets or roads being
   ↪  flooded; or if there is mention of road closures.
  3. "SERIOUS": if there is mention of vehicles, homes, and/or
   ↪  buildings inundated, or containing water; or if there is
   ↪  mention of roads or bridges being damaged.
  4. "SEVERE": if there is mention of vehicles and/or mobile
   ↪  homes being swept away; or if there is mention of injuries.
  5. "CATASTROPHIC": if there is mention of buildings, or large
   ↪  infrastructures submerged; or if there is mention of houses
   ↪  or homes (as opposed to mobile homes) being swept away,
   ↪  carried away, or destroyed; or if there is mention of
   ↪  fatalities.

Assign impact class probabilities (in percents) to the following
 ↪  text, make sure the sum of all probabilities add up to 100%.
 ↪  Only return the probability values, and DO NOT EXPLAIN YOUR
 ↪  REASONING.

Your answer should be expressed following this JSON format:

{
    "MINOR": <PROBABILITY PERCENT>,
    "MODERATE": <PROBABILITY PERCENT>,
    "SERIOUS": <PROBABILITY PERCENT>,
    "SEVERE": <PROBABILITY PERCENT>,
    "CATASTROPHIC": <PROBABILITY PERCENT>
}
```

Listing 3.2: ChatGPT prompt composed of *textualized* FFSI definitions, specific instructions for probability calculations, and output formatting.

For classifying LSRs, the ChatGPT API needs to first be provided with an *system initiation task*, which defines the type of action we are expecting of the LLM (i.e. instructions on how to address out prompts). In this case, the initialization task is the textualized FFSI definitions. Once the task is initialized, the API can be sequentially prompted with *user tasks* to produce classification for each LSR remark. A python tool has been implemented to query the ChatGPT API, allowing it to process large amounts of LSRs in batches. This is very important since the API sometimes can time out due to

server load, or network communication issues. More relevantly, this implementation also is able to resume work after interruptions, without having to re-process more than one batch each time it resumes work (as it writes partial outputs for each batch). For processing our LSR datasets, a batch size of 20 has been chosen as a good compromise between number of partial files produced, file size output, and minimizing the amount of re-processed LSRs at any given failure.

In addition to the joint probabilities of FFSI impact class, a singular numeric representation of each LSR's joint probability was calculated, and named as *FFSI Score*. This FFSI score is calculated by taking the percentage fraction of each class' probability, and multiplying it by the integer which increasingly represents each impact class. This results in a floating point value between [1, 5] for each possible combination of joint probabilities assigned to a given LSR. It should be noted that for LSRs that ChatGPT can not classify using the textualized FFSI definitions, an FFSI score of zero was assigned. The formula for the FFSI score is described in Equation 3.1. Some examples of FFSI scores an their corresponding joint probabilities are presented in Table 3.1.

$$\begin{aligned}\text{FFSI Score} =&(p(MINOR) * 1) + (p(MODERATE) * 2) + (p(SERIOUS) * 3) + \\ &(p(SEVERE) * 4) + (p(CATASTROPHIC) * 5) = [1, 5]\end{aligned} \tag{3.1}$$

Joint Class Probabilities	**FFSI Score**
{p(MIN), p(MOD), p(SER), p(SEV), p(CAT)}	[1,5]
{0.7, 0.3, 0.0, 0.0, 0.0}	1.29
{0.0, 0.7, 0.3, 0.0, 0.0}	2.30
{0.1, 0.3, 0.5, 0.1, 0.0}	2.60
{0.0, 0.1, 0.7, 0.2, 0.0}	3.09
{0.0, 0.0, 0.5, 0.5, 0.0}	3.50
{0.0, 0.0, 0.0, 0.0, 1.0}	5.00

Table 3.1: Example of joint FFSI class probabilities, and their corresponding FFSI scores.

Once all batches have been processed, the tool combines all intermediate results into a single CSV file which contains the original LSR information and the FFSI clas-

sification. The general source code for this implementation can be found in the Code Appendix Listings C6, C7, and C8.

While working on the FFSI-based LSR classification, an extended set of impact definitions for FFSI were provided by the original developers of the FFSI framework (from now on referred to as FFSI V2). They have made recent strides to extend the original FFSI definitions (FFSI V1), aimed at operational use within their respective NWS Weather Forecast Offices. It must be noted that FFSI V2 definitions are much more extensive and detailed than FFSI V1. An example of textualized FFSI V2 definitions can be found in the Appendix Listing A1.

3.3 FFSI-Based Flash Flood Impacts Model

Section 3.2 shows the methods performed to construct a flash flood impacts dataset comprised of 22,329 LSR reports between 2018 and 2022. This historical dataset has been classified into probabilistic FFSI impact categories (joint FFSI probabilities + FFSI score) through a ChatGPT+FFSI approach, and each report has been matched to corresponding FLASH MaxUnitQ and MaxARI products, for which statistical moments have been extracted (see Section 3.2.1). This addressed the present work's first main objective.

In order to address the second challenge of this dissertation, a proof-of-concept model capable of producing impact probabilities based on real-time measures of precipitation and flow response (among other variables) will be designed and implemented. However, before delving into model building and training, important constraints must be chosen to propose this proof-of-concept application; particularly concerning the extension of the CONUS domain. The MRMS CONUS domain is represented by 7000x3500 pixels, each with a horizontal and vertical resolution of $\sim$1km (0.01^{o}), for a total of 24,500,000 pixels of which around 10,000,000 represent grid cells over land. Working at this is computationally expensive, specially when thinking of training deep learning models. For this reason, a subdomain within the CONUS was chosen.

Following the flash flood events that occurred in the state of Kentucky July 26th-30th of 2022, a case study was conformed by NSSL's Stormscale Hydrology group

to hold a workshop at the 14th International Precipitation Conference. Having prepared multiple datasets for this workshop, a decision was made to restrict our proof-of-concept model to the state of Kentucky (KY). Figure 3.11 details the location of the minimum bounding box around the state of Kentucky, with respect to the CONUS domain.

Figure 3.11: CONUS and Kentucky Subdomain. The MRMS CONUS domain spans 7000x3500px, with a total area of composed by 24,500,000px^2 of which $\sim 10,000,000$px^2 are overland. The Kentucky state subdomain spans 779x280px, covering an area of 218,120px^2

To consistently be able to match LSRs to the state of KY and having some additional spatial context for those events near the border, first a minimum bounding box was obtained for the KY state line. Then, this bounding box was inflates by 10km to each side, in order to provide a sensible buffer surrounding the state line. This subdomain consists of 779x280px, and a total of 218,120 grid cells with MRMS spacing ($\sim$1km, 0.01^o). Figure 3.12 shows the KY state line with a dark shaded minimum bounding box, and a green overlay which illustrates the 10km padding around the bounding box.

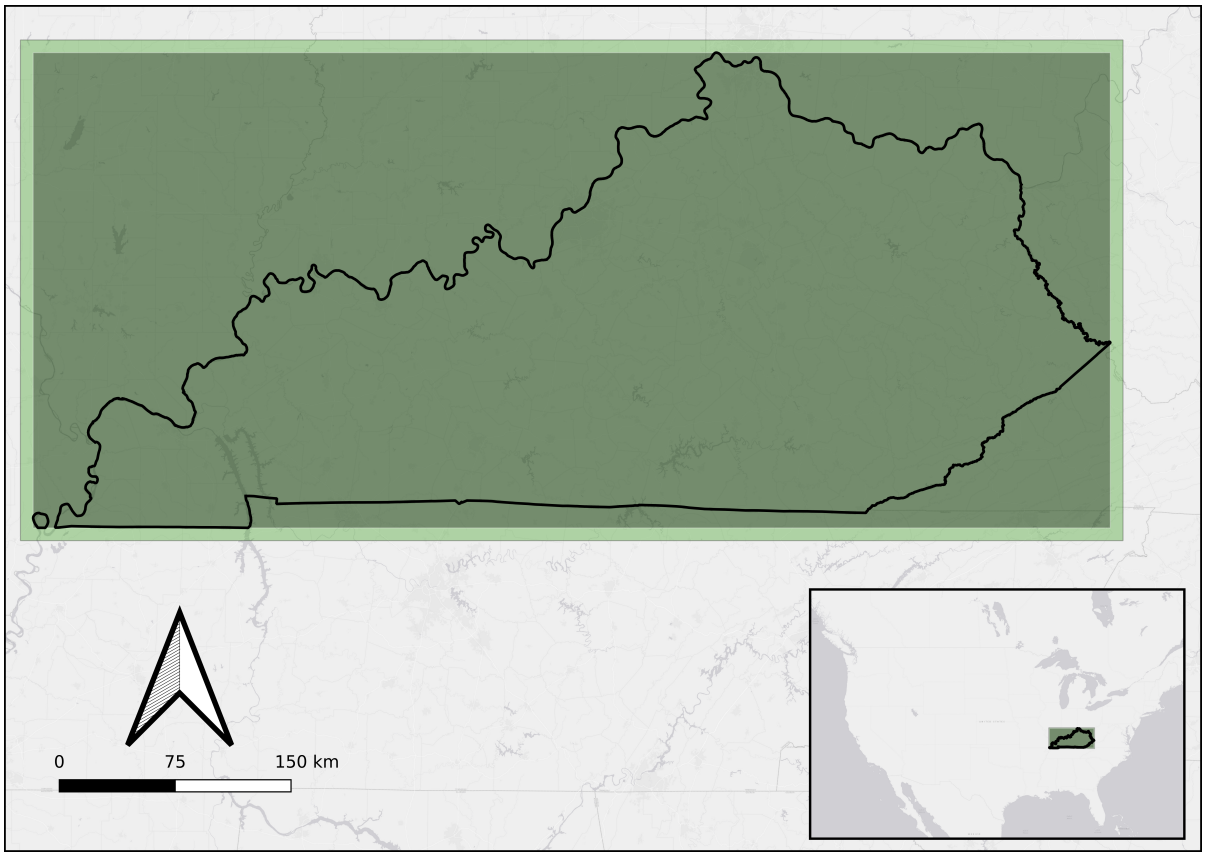


Figure 3.12: KY state line with a dark shaded minimum bounding box, and a green overlay which illustrates the 10km padding around the bounding box.

Having defined our working subdomain, the next step is to collect all LSRs that correspond to the state of KY. Fortunately, `state` is an attribute for all LSRs, so a simple selection by state was enough to retrieve all the KY LSRs from both the historical and the expertly-classified LSR datasets. A grand total of 979 flash flood LSR reports were found to correspond to the state of Kentucky between 2018 and 2022. A total of 779 non-null, non-edge LSRs were retrieved for the state of KY. Null events are those with an FFSI score $<= 1$, while non-edge events are those that allow for a 128x128 pixel buffer around the LSR location, without outcropping the bounds of the subdomain. Of these 779 non-null, non-edge LSRs, only 153 complete cases were found, for which no missing UnitQ or ARI time steps were found in the dataset (i.e. no missing FLASH product data spanning $\pm 3h$ surrounding the LSR report time). Figure 3.13 shows the locations of the 979 LSRs found within the KY subdomain.

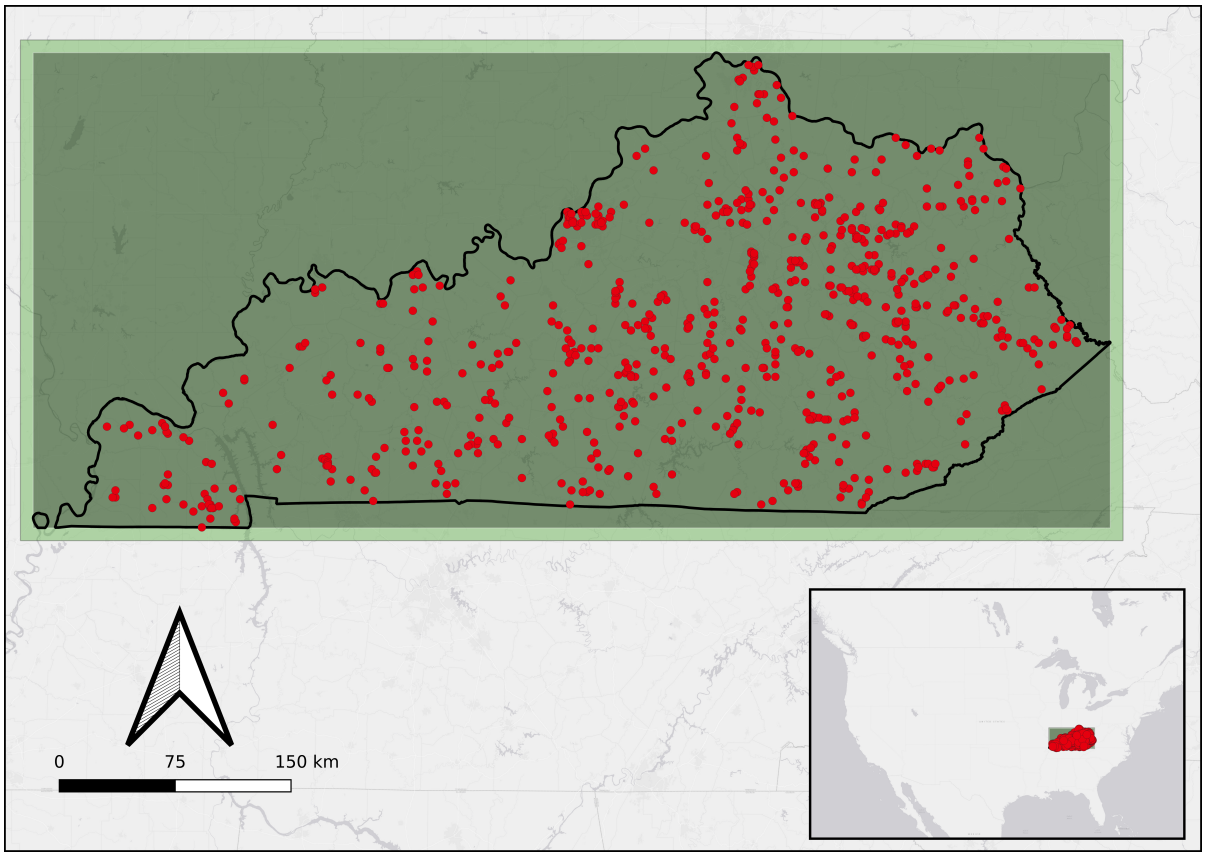


Figure 3.13: Location of the 979 total LSRs found within the Kentucky subdomain.

Having identified the LSR reports with their associated GPT-based FFSI classification, we now must prepare our training dataset. Since we want to annotate FLASH MaxUnitQ flash flood forecasts, MaxUnitQ will be our most obvious input attribute, alongside FLASH's MaxARI. While MaxARI will inform the model regarding the causative rainfall associated to a specific flash flooding event, MaxUnitQ will inform the model about the estimated response to the causative rainfall. These will also be our only *dynamic* data (i.e. data that changes through time), and they are shown in Figure 3.14. Conversely, additional *static* data layers will inform the model about the underlying morphology and vulnerability, associated with the location over which the reported flash flood event occurred. Among geomorphological variables, imperviousness (percentage of imperviousness associated with a given grid cell), digital elevation, flow accumulation (the amount of pixels upstream that drain to a given pixel), and flashiness (how prone a given stream pixel is to a rapid response and to cause flash flooding) [2] were selected. Regarding vulnerability layers, the following attributes were selected: population density, primary road density, mobile home parks, low wa-

ter crossings, camp grounds,and building centroids. All static layers are represented at a grid cell level, and a sample of them are shown in Figure 3.15.

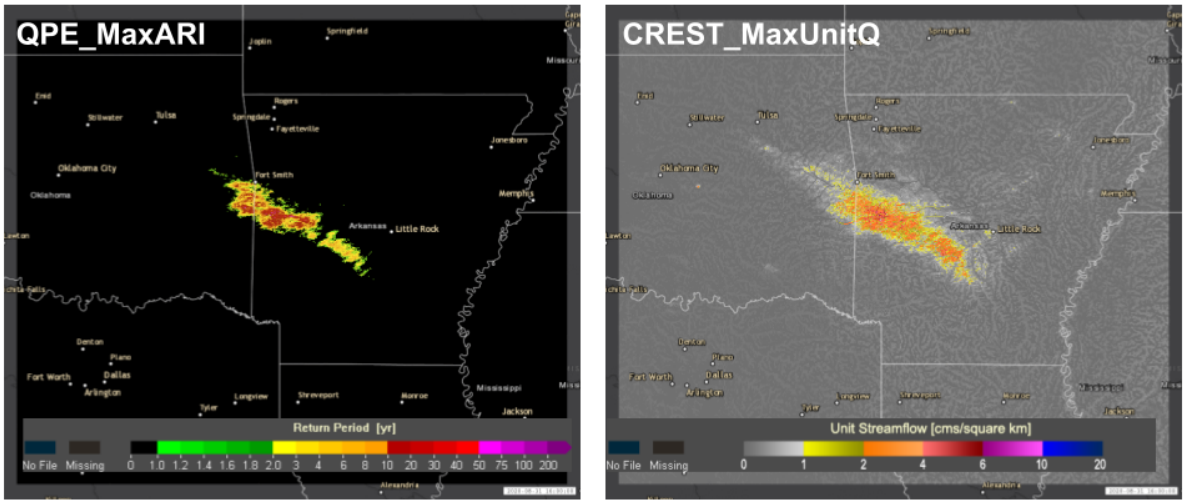
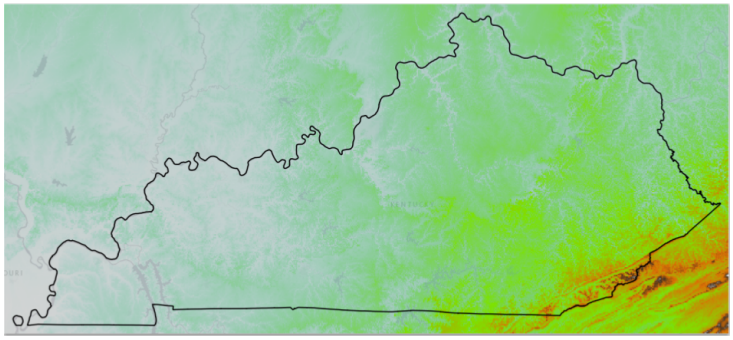


Figure 3.14: FLASH products - (left) Flash Maximum Average Return Interval (MaxARI); (right) FLASH Maximum Unit Streamflow (MaxUnitQ)



(a) KY DEM

(b) KY Flashiness

Figure 3.15: (top) Kentucky digital elevation, notice the Cumberland Mountains and Black Mountain towards the southeastern portion of the state, by the border with the state of Virginia.; (bottom) Kentucky flashiness.

Static vulnerability layers were readily available to NSSL's Stormscale Hydrology group, since they were generated as part of the JTTI-funded project mentioned in Section 1.4. Geomorphological layers were also readily available as part of the FLASH project's operational nature, since these layers are generally used as parameters for the CREST hydrologic model in EF5 [23]. To prepare these static layers, a cropping mask for the KY subdomain is employed to process and prepare these rasters. Conversely, while we previously accessed the dynamic MaxARI and MaxUnitQ layers to extract the statistical moments described in Section 3.2.1, the FLASH product files associated to each LSR report's spatial were only read but not retrieved. Fortunately, by making use of the same codebase used for the moment extraction, with slight modification, a product extraction python tool was engineered to find, crop, and save the CONUS-scale FLASH products corresponding to each LSR. For each LSR event, FLASH product outputs were extracted at 10 minute intervals, for the previous three hours to the reported time of each event; this means that for each LSR report, a total of 18 outputs for both MaxUnitQ and MaxARI were extracted (3h 10min = 18 time steps). The product extractor has been implemented with the capability to impute missing intermediate FLASH output files for any of the products, by assuming a steady state between the start and the end of the missing period. However, there were cases for which imputation was not possible, and only a few of the 18 time steps had any data available.

Now that we can extract our input data for each LSR (ten static layers and two dynamic layers: MaxUnitQ and MaxARI – each a sequence of 18 frames), we must also convert our FFSI classification into a spatialized representation which is compatible with our 2D/3D input data over the KY domain. For each LSR event, a buffer area is calculated around its reported location, within the Kentucky subdomain. This buffer will be the same spatial search window with which we extracted the statistical moments in Section 3.2.1: a search radius of 4 pixels ($\delta_r = 4$), which results in a 9x9 pixel window centered on the LSR location. Once this buffer is determined, an output raster is built for each LSR, where the extents are those of the Kentucky subdomain, but all values are *empty* except for the 9x9 pixel buffer around the LSR location, for which all pixels hold de value corresponding to each LSR's FFSI score. This is described in Figure 3.16 below.

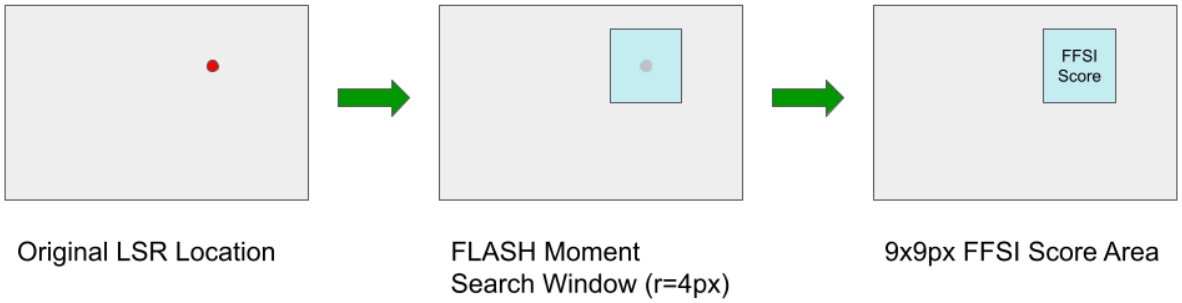


Figure 3.16: Spatialized FFSI score outputs over the Kentucky subdomain, calculated for a single LSR.

Having extracted, cropped and composed all of the necessary attributes that make up our training dataset of 779 LSRs over the Kentucky subdomain, we can now articulate what constitutes a training sample. A training sample is a tensor of dimensions (779, 228, 47) which represents an LSR event, composed by 10 geomorphological and vulnerability static layers, a sequence of 18 MaxUnitQ layers that span the previous 3 hours before the event, a sequence of 18 MaxARI layers that span the previous 3 hours before the event, and a single output layer with a spatialized FFSI score. Note that the tensor's dimensions correspond to the KY subdomain's shape (779x228 pixels) and the total number of layers that constitute a training sample (47). Figure 3.17 shows the structure of a training sample tensor representing a single LSR event.

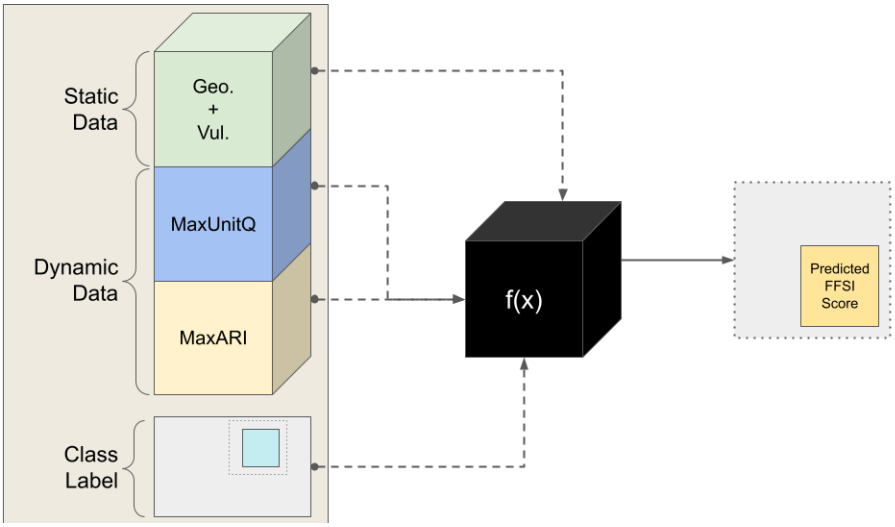


Figure 3.17: Structure of a training sample, composed of static layers, dynamic layers, and a class label output layer.

All three experimental models listed below were readily available as PyTorch im-

plementations [82], which could then be reused through a Python API. Models and documentations are also available through the Hugging Face community website [83].

3.3.1 Segmentation Transformers

The first model explored with the 779 non-null, non-edge training dataset was a Segmentation Transformer [84] (Segformer, see Section 1.2.5). For this model, the 47 layers of each training sample will be used as a whole. The idea behind this exploration is to attempt to disregard the temporal features in the sequence of FLASH product data, and test whether the Segformer model can pick up on any spatial features from the static and dynamic data, when consumed as a whole (i.e. a 46-channel image), and attempts to perform semantic segmentation based on the provided class labels in the output layer. The segformer architecture explored in this phase corresponds to the largest model, denominated `MiT-b5`, which has 82 million parameters, and a Multi-Layer Perception decoder, which acts as a classification head [19]. A diagram of this model's implementation is shown in Figure 3.18. Results for this model's training can be seen in Section 4.3.1.

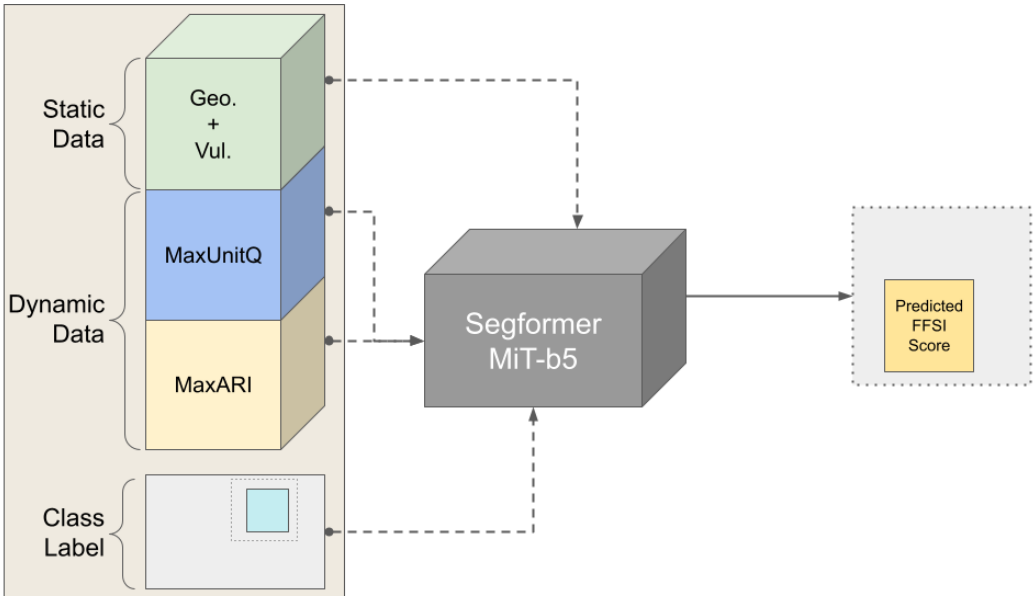


Figure 3.18: Segformer model implementation.

3.3.2 Video Masked Autoencoders

Proceeding with further experimentation using only the dynamic data from the 779 non-null, non-edge training samples, a Video Masked Autoencoder [85] model (VideoMAE, see Section 1.2.5) with a classification head was explored to test modeling the dynamic inputs and their sequential nature. Particularly, for this approach, the MaxUnitQ and MaxARI sequential data are treated by the model as 18 frames of a two-channel video. During training, the features extracted by the VideoMAE model are piped through a classification head comprised of 12 hidden layers of size 768 (the default for Hugging Face's implementation of their `VideoMAEForVideoClassification` class), alongside spatialized FFSI class label. A depiction of this architecture is shown in Figure 3.19. Results for this model's training can be seen in Section 4.3.2.

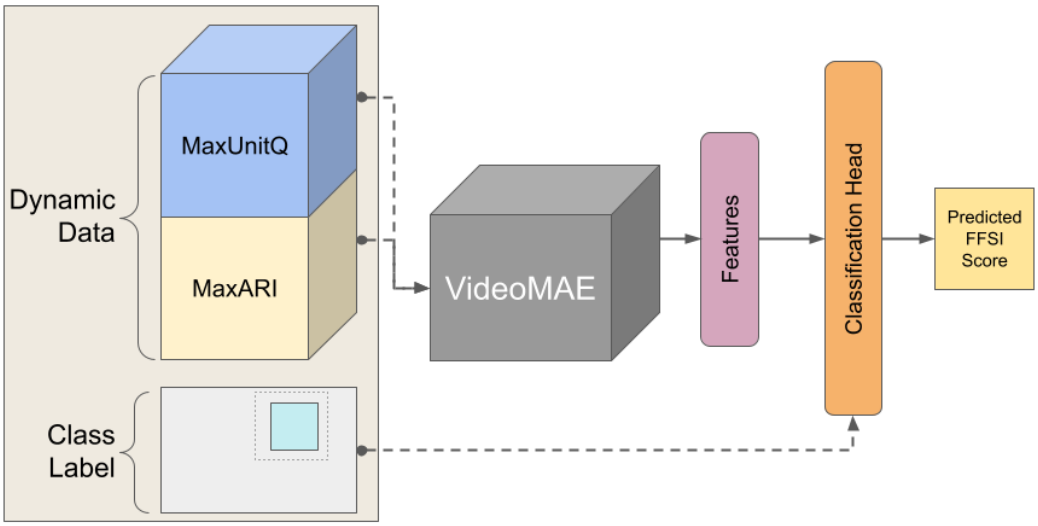


Figure 3.19: VideoMAE model implementation.

3.3.3 VideoMAE + Vision Transformers

As the last portion of the experimentation process, a composite approach was devised to use different models for dealing with the static and dynamic attributes of our dataset. A VideoMAE component will ingest the dynamic FLASH product data, and extract features from it using the two-channel video approach discussed in Section 3.3.2. For the static layers, a Vision Transformer [86] (ViT, see Section 1.2.5) model will be employed to ingest the static layers as a multi-channel (10 channels), to perform feature extraction. The latent representations from both the VideoMAE model

and the ViT model will then be concatenated, and piped as input to a simple classification head (two-layer fully connected feed-forward neural network with 768 neurons each), alongside the spatialized FFSI class label for each instance. Figure 3.20 details this compound architecture with distinct models for static and dynamic layers in our training samples. Results for this model's training can be seen in Section 4.3.3.

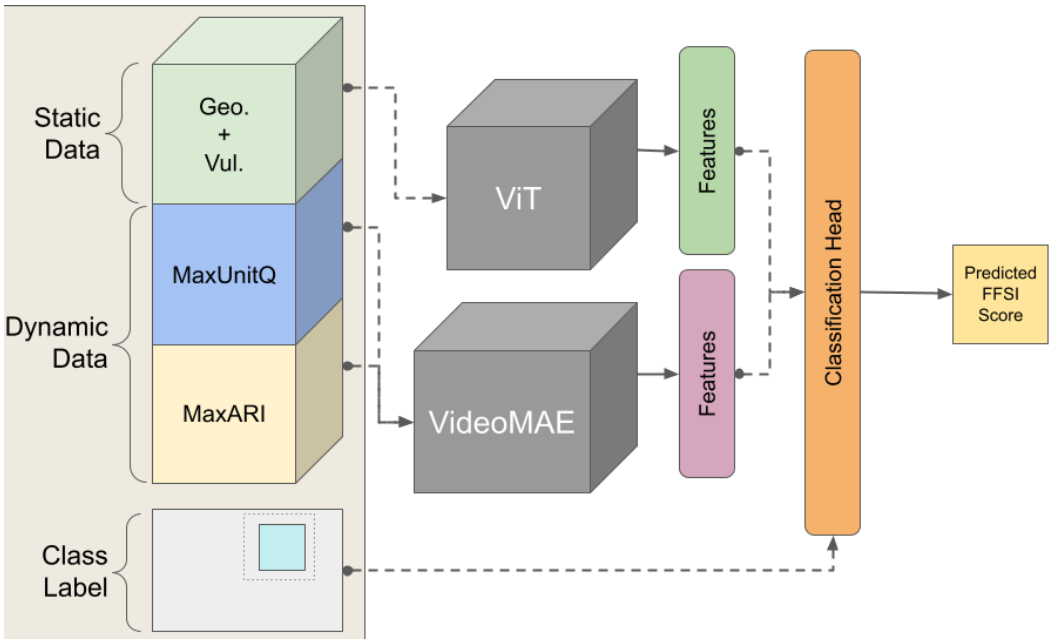


Figure 3.20: VideoMAE+ViT model implementation.

Chapter 4

Results

This chapter contains the results achieved by the various methods detailed in Chapter 3, which address the two main challenges faced by this dissertation. Concerning the firs challenge of constructing a historical flash flood impacts dataset: Section 4.1 will present the results associated with using pre-trained transformer-based language models (BERT) to train an LSR classifier; Section 4.2 will present the results of using generative, predictive Large Language Models (ChatGPT) and a language-based flash flood impact framework (FFSI) to classify LSRs. Regarding the second challenge of training a proof-of-concept ML model to forecast flash flood impact, this chapter will present the results of attempting three distinct approaches which gradually explore its implementation. Section 4.3 presents each of the three approaches in a different section: Segmentation transformer (Segformer) results will be shown in Section 4.3.1, Video Masked Autoencoder (VideoMAE) results will be shown in Section 4.3.2, and lastly a combined approach using VideoMAE and Video Transformer (ViT) results will be shown in Section 4.3.3.

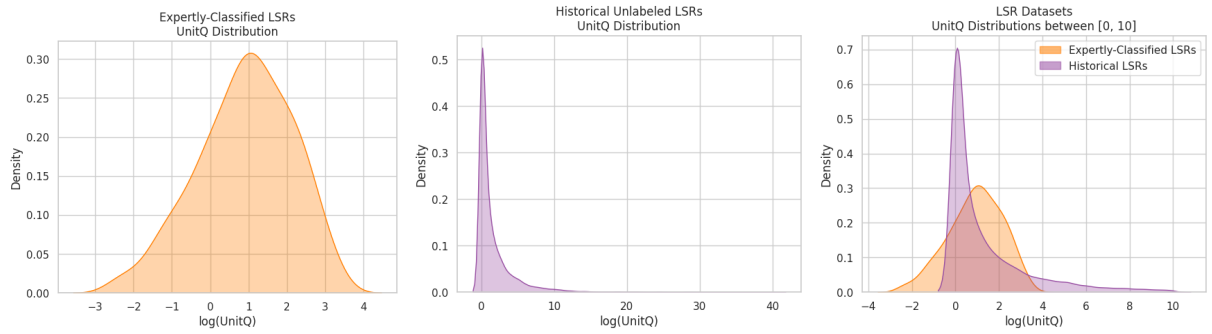
4.1 BERT for IBW LSR Classification

This section will detail the results of training two distinct BERT-based models, as well as their respective data preparation processes. First expertly-classified and historical LSR reports must be matched to FLASH unit streamflow (UnitQ) values which will enable to establish a way of assessing model performance, based on an objective measure

(this is described in Section 4.1.1). Using the expertly-classified dataset, a pre-trained BERT model is used in conjunction with a dense network to train an LSR classification model, which is then validated on the historical dataset (these results are shown in Section 4.1.2). Attempts to improve model performance are incurred in, by incorporating additional event impact data from StormDat events. This entailed matching our LSR datasets with StormDat reports, in order to enrich our training data (this process is shown in Section 4.1.3). Lastly, with this newly enriched dataset, the same BERT architecture used previously was retrained and validated using the historical LSRs (these results are shown in Section 4.1.4).

4.1.1 LSR Event Matching to FLASH Max UnitQ

Following the method described in Section 3.1.1, a temporal search period of $\delta_t = -6h$, and a spatial search radius $\delta_r = 1km$ were used. While UnitQ values were extracted reliably for the 663 expertly-classified reports, this was only possible for 21,852 of the total 22, 829 historical unlabeled reports. Figure 4.1 shows the UnitQ value distributions for both the expertly-classified, and the historical unlabeled LSRs.



(a) (b) (c)

Figure 4.1: (a) Distribution of matched MaxUnitQ values for the expertly-classified dataset; (b) Distribution of matched MaxUnitQ values for the historical unlabeled dataset; (c) Distributions of matched MaxUnitQ values for both datasets, cropped up to MaxUnitQ values of 10.

As can be seen from figure 4.1c, the UnitQ distribution values between the two datasets appear to be 1) biased in opposite directions, and 2) have central tendencies which diverge from one another. Expertly-classified events tend to have a more 'cen-

tral' distribution of log(UnitQ) values, while historical events show a skewed distribution, towards much lower log(UnitQ) vales. This is somewhat expected, since being a much large collection of historical reports, lower-end UnitQ values and events tend to be more prevalent (i.e. extreme events are rare, thus less frequent).

Since we do have access to impact-based warning categories in our expertly-classified dataset, we can also examine the UnitQ values with respect to each IBW class. These distributions are shown in Figure 4.2 below.

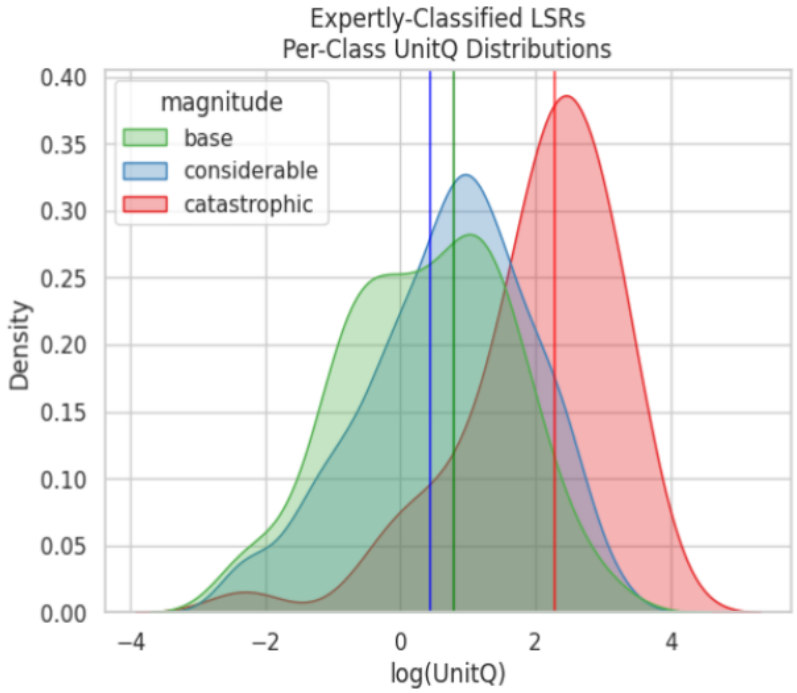


Figure 4.2: Per-class density distribution plot of expertly-classified LSR UnitQ values. Medians for each class are color-coded, and shown by continuous vertical lines.

As can be seen from this per-class distribution of UnitQ values, there is a drastic overlap between class distributions. And while the distribution of `catastrophic` reports seems to be consistently skewed towards high UnitQ values, the distributions for `base` and `considerable` classes seem to overlap almost completely. Furthermore, the medians of the expert classification distributions exhibit an unexpected relative location for the `base` and `considerable` classes, since they appear to be inverted, instead of presenting themselves in an increasing order of magnitude class. This clearly portrays the fact that the experts (forecasters) struggle to discern between the non-

`catastrophic` classes. While this may seem surprising at first, it is logical that this struggle stems directly from the general IBW guidance for UnitQ values presented in Table 1.3, as well as in Figure 3.1. While distinctly low UnitQ value ranges are provided for `base` events, the same ranges are provided for both the `considerable` and `catastrophic` events. This means that the middle class `considerable` will always end up being decided from and confused with both the upper range of `base` UnitQ values, and the lower range of `catastrophic` UnitQ values; and since less extreme events are more frequent, it is more likely that the `considerable` class will be consistently decided upon with higher `base` UnitQ range values.

4.1.2 Remark-only BERT Models

The best training results for the first model trained with the full 663 expertly-classified instances yielded the following results:

Metric	**Value**
Categorical Accuracy	0.638
F1 Score	0.498
Precision	0.640
Recall	0.638

Table 4.1: Training metrics for BERT-based IBW classifier using 663 instances.

As can be seen from the above results, even though our training categorical accuracy reached modest levels, our F1 Score indicates that roughly half of the model's predictions were correct. This point is echoed by out Recall score, which indicates that our true positive (and false negative) rate also correspond to roughly half of our attempts. As a rough validation of our model's (poor) skill, inference was tested on 21,852 unseen historical LSRs (not part of the expertly-classified reports). Validation results yielded that 100% of the unlabeled, unseen reports were classified as `considerable` events. Not only is the training performance insufficient, but the validation results are a clear reflection of the severe class imbalance which favors the `considerable` class in our expertly-classified dataset.

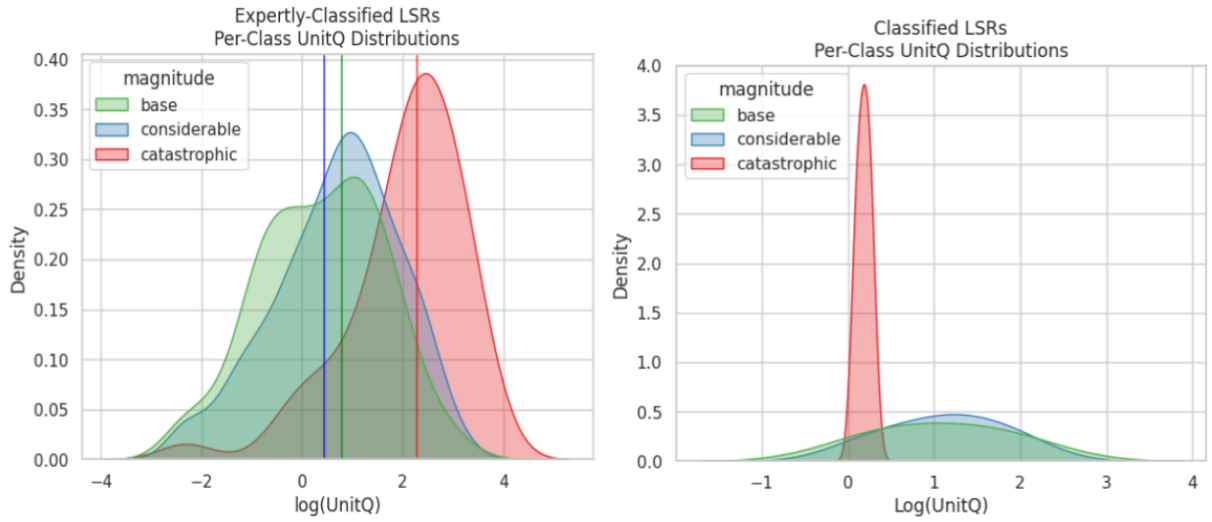
Knowing that performance was not feasibly going to improve by removing data

samples from our training data, the model was trained on the balanced subset of 171 expertly-classified reports to assert whether the class imbalance was responsible for our first model's bias. The best training results for the second model trained yielded the following results:

Metric	**Value**
Categorical Accuracy	0.415
F1 Score	0.411
Precision	0.667
Recall	0.012

Table 4.2: Training metrics for BERT-based IBW classifier using a balanced subset of 171 instances.

From the above performance metrics we can see that as expected, overall performance diminished since the sample size was dramatically reduced (even though precision remained similar). The model's F1 score did not change dramatically but did decrease, implying that roughly 40% of the model's predictions were correct. However, in this case, our Recall metric tells a different story: this second model sees a very high proportion of false negatives in its predictions. As in the previous model, validation was performed on the unseen and unlabeled 21,852 historical LSRs. Interestingly, this model classified 10,351 of the validation instances as `considerable` events (47.37%), 6,524 instances as `catastrophic` events (29.86%), and 4,977 instances as `base` events (22.78%). Therefore, even though by balancing the proportion of instances in the training data, the bias towards the `considerable` class was improved, the performance achieved is unsatisfactory: the model will certainly produce large quantities of false negatives, and the overall skill was not improved at the cost of drastically reducing our training sample size.



(a) (b)

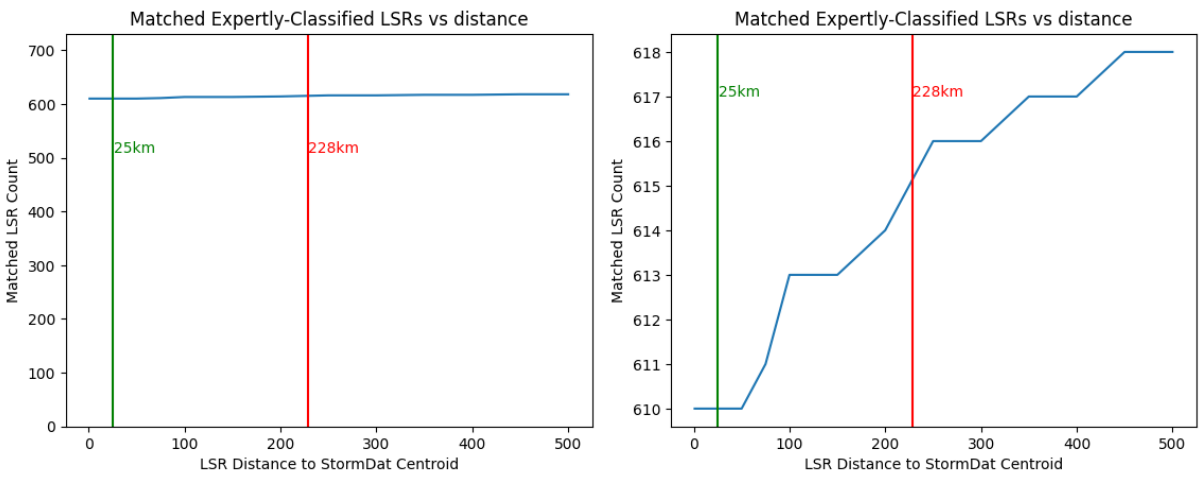
Figure 4.3: (a) ; Measures of central tendency are color-coded, medians are presented in solid vertical lines. (b); ;

Even though performance for this approach is not satisfactory, this clearly establishes a baseline with which to gauge the following approaches. These results are certainly not related to pre-trained BERT models, as numerous other research works and applications have proven that they can handle the heavy lifting for NLP tasks. Rather, what these results show seems to point to our data scarcity problem, which is insufficient to train a classifier based on natural language, due to having such a small set of samples at our disposal.

4.1.3 LSR Event Matching to StormDat Data

As mentioned in Section 3.1.3, a sensitivity analysis was performed to define the distance measure the use in our LSR-StormDat matching heuristic. Increasing values of distance between one and 500km were used to match the expertly-classified LSRs, having as a reference a measure of 228km, which corresponds to the square root of the are of the larges county in the COUNS (San Berdardino, CA, USA). The idea behind this measure was to find an idealized representation for generalized county extents, to define a reasonable report matching radius. As shown in Figure 4.4, the difference between using a 25km distance threshold and a 228km distance threshold is minimal with respect to the sample size, and maintains a reasonable measure of uncertainty

with respect to an LSR's location with respect to a given StormDat report.



(a) (b)

Figure 4.4: Matching Distance Sensitivity Analysis. (a) Overall results for the sensitivity analysis; (b) Detailed results for the sensitivity analysis; Notice that the difference between using a measure of 25km (610 reports) and 228km (615) is only 5 non-matched reports.

StormDat reports (51,623) were matched to both the expertly-classified LSRs (22,329 reports) and historical LSRs (663 reports); a total combined of 22,992 LSRs. Keep in mind that multiple LSRs (events) can be matched to a given StormDat report, and each StormDat report can describe a multiple *events* associated with a single *episode*.

Out of the 22,329 historical unlabeled LSRs, 11,677 reports were matched to 4,741 existing StormDat events, which were associated to 2,372 episodes. This means that not all of the historical LSRs were found to have a matching StormDat report using the heuristic matching approach described in Algorithm 3.1. Conversely, for the 663 expertly-classified LSRs, they were all matched to existing StormDat reports, encompassing 249 events associated to 129 episodes. Figure 4.5 illustrate these proportions. For the 663 expertly classified reports, 313 were matched by state and county, 236 were matched by point-in-polygon, 53 were matched by a distance radius of 25km to the StormDat polygon's centroid, and 9 were matched by only county name. Figure 4.6 illustrates these proportions.

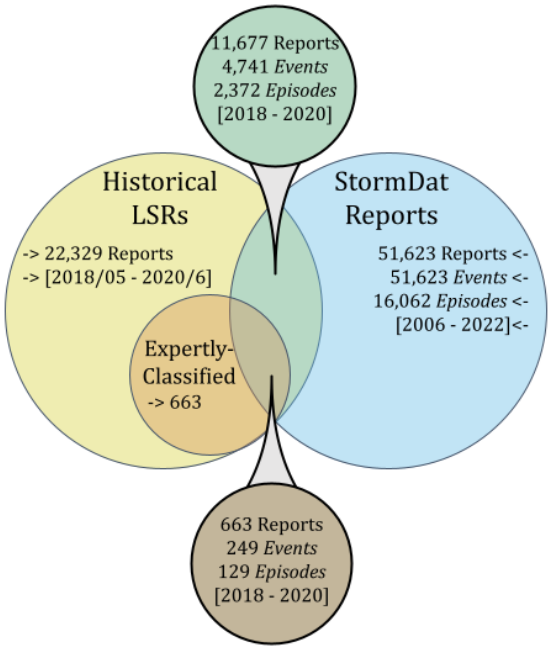


Figure 4.5: Venn diagrams detailing the outcome of the matching between LSR and StormDat reports.

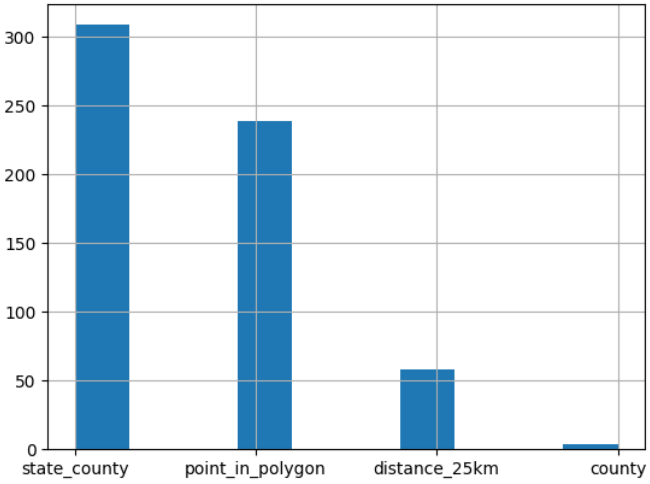
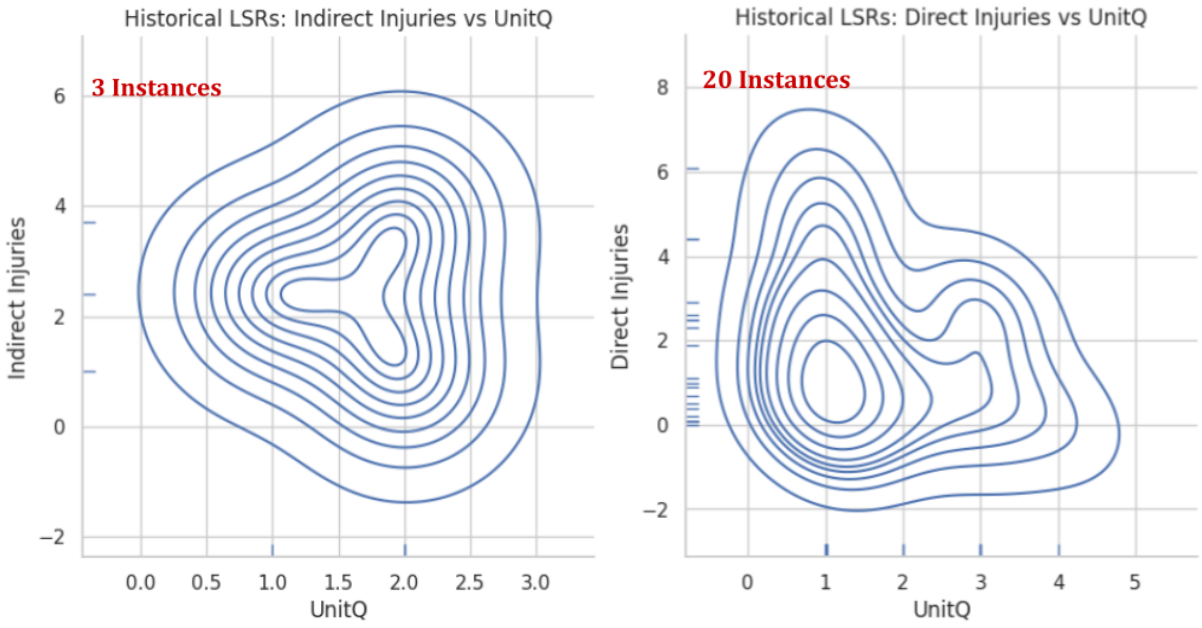


Figure 4.6: Histogram of matching criteria frequency between LSR and StormDat reports, for the 663 expertly-classified reports: 313 reports were matched by state and county, 236 were matched by point-in-polygon, 53 were matched by a distance of 25km from the StormDat polygon's centroid, and 9 were match only by county name.

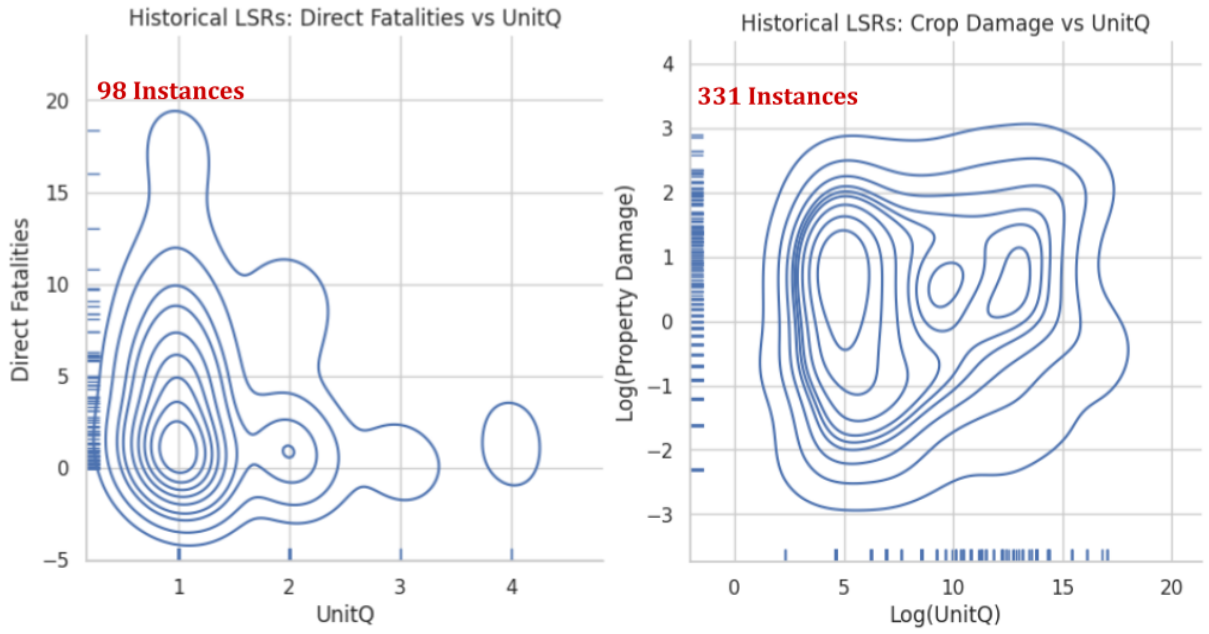
Having been able to associate StormDat-specific measures of impact (indirect injuries, direct injuries, direct fatalities, crop damage, and property damage) to our 11,677

matched LSR datasets, density plots were generated to explore the relationships of UnitQ values associated to LSR reports, with each of these StormDat measures of impact.



(a) (b)

Figure 4.7: Historical LSR StormData variable value distributions vs UnitQ (a) Indirect injuries, 3 matched instances; (b) Direct injuries, 20 matched instances



(a) (b)

Figure 4.8: Historical LSR StormData variable value distributions vs UnitQ (a) Direct fatalities, 98 matched instances; (b) Crop damage, 331 matched instances.

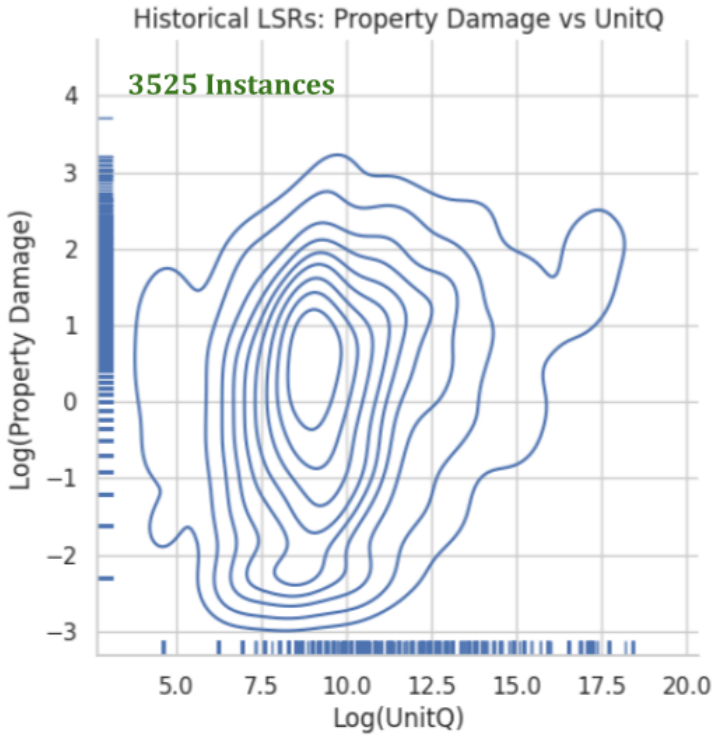
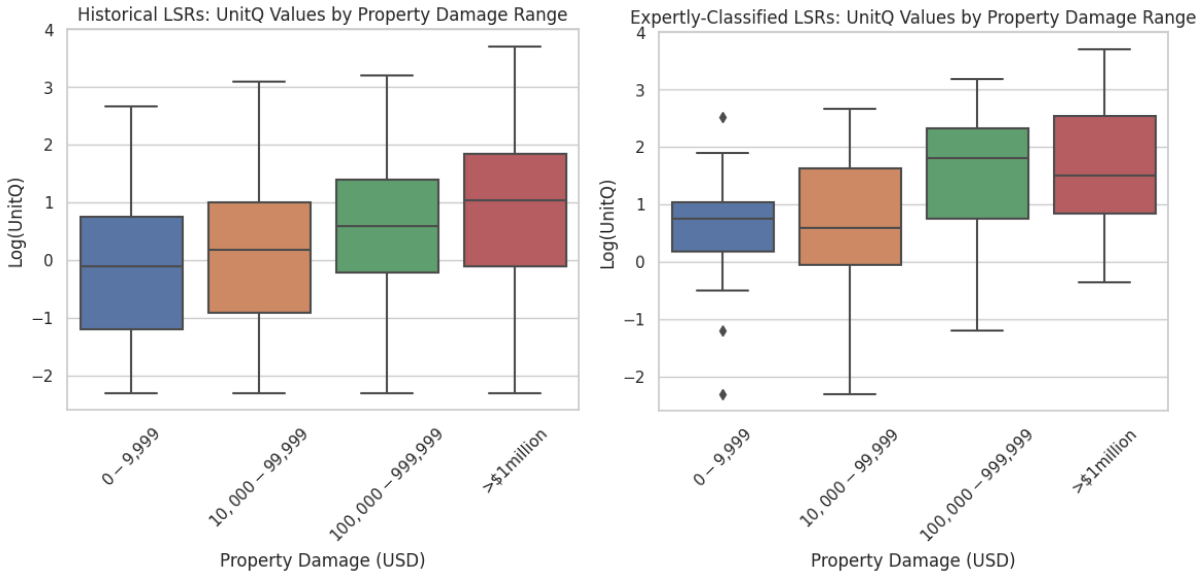


Figure 4.9: Historical LSR StormData Property Damage value distributions vs UnitQ, 3,525 matched instances

Out of the 11,677 matched LSR reports for which 4,741 StormDat reports were matched, 3 matched instances had associated indirect injury values; 20 matched instances had associated direct injury values; 98 matched instances had associated direct fatality values; 331 matched instances had associated crop damage values; and 3,525 matched instances had associated property damage values. This showed that property damage seemed to be the most represented attribute within the 11,677 matched LSR-StormDat reports.

In order to assess the relationship between ranges of property damages with values of UnitQ, a box-plot analysis was performed. It was found that, even though there is a very large overall dispersion of UnitQ values for all damage ranges, higher damage cost ranges are associated with higher UnitQ values. This general trend holds true for both the expertly-classified LSRs, as well as the historical unlabeled LSRS, which can be seen in Figure 4.10.



(a) (b)

Figure 4.10: Box plots of Log(UnitQ) vs StormDat property damage ranges. (a) Log(UnitQ) vs StormDat property damage ranges of historical unlabeled LSRs; (b) Log(UnitQ) vs StormDat property damage ranges expertly-classified LSRs; Note that the general trend shown in the historical LSRs is preserved in the expertly-classified dataset: higher damage cost ranges are associated with higher UnitQ values.

4.1.4 LSR + StormDat BERT Models

The best training results for the model trained on the 663 expertly-classified, enhanced LSR-StormDat remarks yielded the following results:

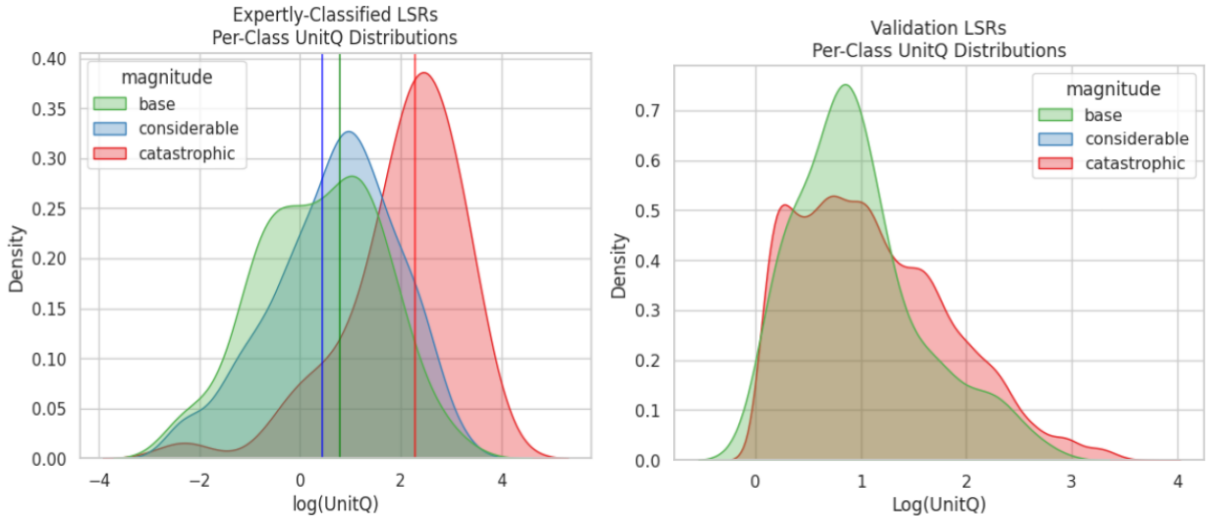
Metric	Value
Categorical Accuracy	0.638
F1 Score	0.498
Precision	0.640
Recall	0.638

Table 4.3: Training metrics for BERT-based IBW classifier using 663 instances which were enriched with StormDat event impact data.

As we can see from the above performance metrics, results are no different from

those obtained by the first model shown in Section 4.1.2. Even though we enriched our remarks with much more information for BERT to pick up on, our training sample size is still too small for us to train a deep learning classifier based on transformer representations derived from these enhanced remarks.

A similar validation was performed using the trained model on the historical LSRs, and results were not favorable. The model not only failed to demonstrate any expected bias towards the `considerable` class (which is the most prevalent in the training dataset), but in fact classified none of the instances as `considerable`. It classified 98.43% of the validations instances as `catastrophic`, and only 1.57% as `base`. This validation reflects a clear failure to improve the classification model's training by introducing these new StormDat features.



(a) (b)

Figure 4.11: (a) Expertly-classified LSRs, training data class density distributions ; (b) Historical LSRs, validation results class density distributions; Measures of central tendency are color-coded, medians are presented in solid vertical lines.

In order to try to understand why these StormDat-derived quantities do not seem to have noticeable effect on our model's training, a closer look into these quantitative variables is warranted. Upon closer inspection, it was found that out of the 663 expertly-classified instances, only 189 had associated StormDat features. Concerning those instances corresponding to the class `considerable`, only two had non-damage StormDat features associated with them, and they were fatalities. Furthermore, by

inspecting the property damage distribution values across the IBW classes for the expertly-classified dataset, all classes show a completely overlapping distribution of property damage amounts with respect to UnitQ values.

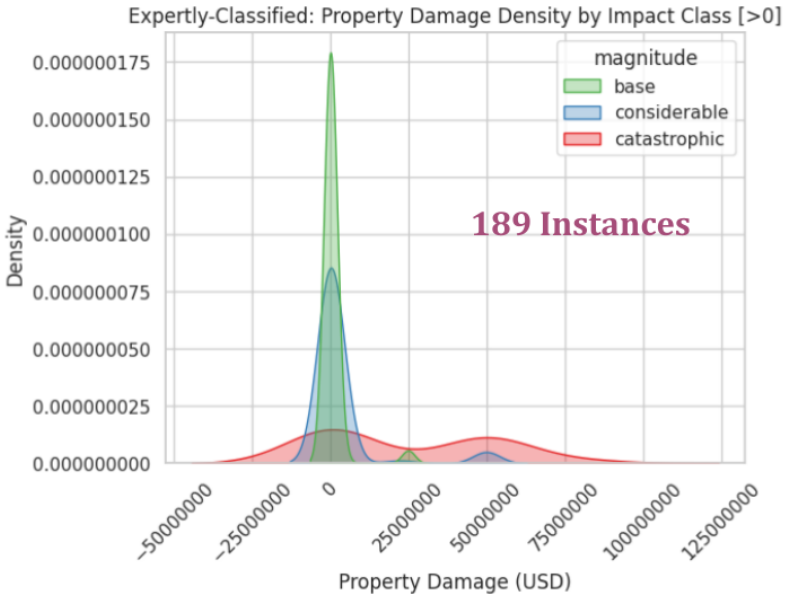
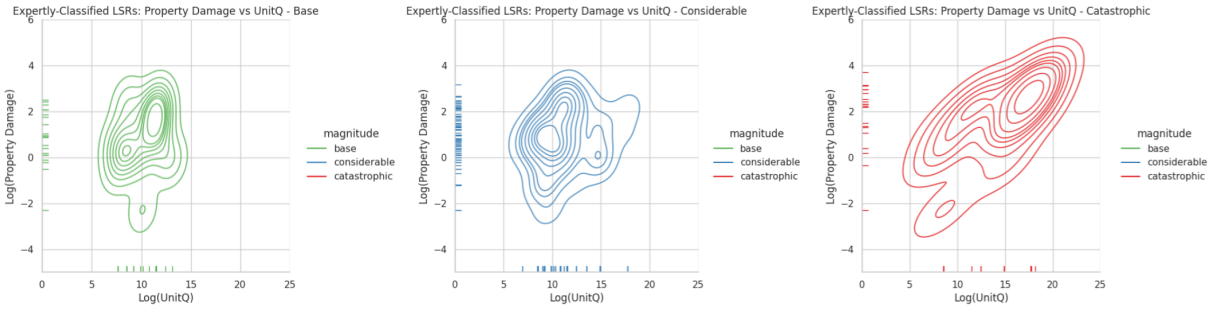


Figure 4.12: Density distribution by impact class of StormData property damage.

More detailed inspection of UnitQ vs property damage values for the StormDat-enriched expertly-classified dataset, showed that for the `base` and `considerable` classes there is no clear trend which correlates increasing values of UnitQ with higher values of property damage. Conversely, there seems to be a more evident trend when looking at the `catastrophic` class.



(a) Base (b) Considerable (c) Catastrophic

Figure 4.13: Expertly-Classified LSR+StormDat Property Damage vs UnitQ - Class Distributions

More relevantly, when we visualizing all distributions together, it can be seen that these class relationships with UnitQ and property damage are completely overlapping, which instead of aiding our classification model to segregate between the classes, is likely making it much harder.

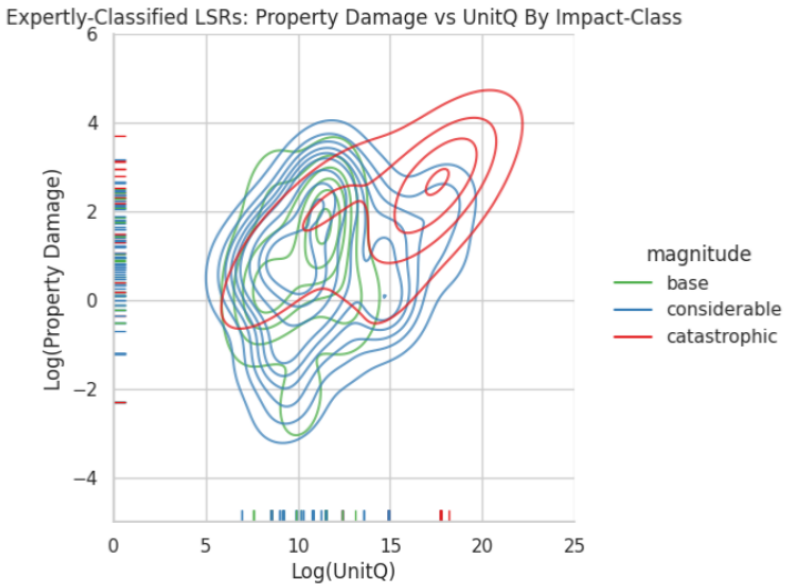


Figure 4.14: Expertly-Classified LSR+StormDat Property Damage vs UnitQ Distributions

As shown in the results presented in this section, none of the multiple BERT-based approaches seemed to yield any favorable results. What's more, throughout out attempts it was impossible for any of the models to memorize the training data (overfitting). This, together with a complete lack of skill in reliably zeroing-in on clear signals from our dataset, indicates a severe lack of training data to address the LSR classification problem in its present formulation. Thus, subsequent sections will explore a different language-based approach to working with the much larger historical unlabeled dataset, leaving the expertly-classified reports as a measure of validation of LSR classification performance.

4.2 ChatGPT-based Flash Flood Severity Index Dataset

As described in Section 3.2, a new approach was taken towards producing a flash flood impacts dataset, relying on the ChatGPT API, and textualized FFSI impact definitions.

First, a new method for matching our expertly-classified and LSR datasets to FLASH products was explored. In large part, this was motivated by the disparity of UnitQ distributions observed after following the previous UnitQ matching method (see Figure 4.1), which we consider to be a direct result of a very tight spatial search window, and the fact that these events were reduced to the extraction of a single UnitQ value for a 6 hour period. As explained in Section 3.2.1, statistical moments are extracted for each LSR, from two distinct FLASH product, and their intersections. These results will be detailed in Section 4.2.1. Subsequently, both the expertly-classified and the historical LSR datasets are classified into FFSI impact classes by the ChatGPT API using the textualized FFSI definitions portrayed in Listing 3.2.

4.2.1 FLASH Product Moment Extraction

As described in the methods Section 3.2.1, MaxUnitQ moments were reliably extracted for both the 663 expertly-classified reports and the 22, 829 historical unlabeled reports. As can be seen from Figure 4.15, compared to results from our previous UnitQ matching procedure shown in Figure 4.1, the extracted distributions for the maximum MaxUnitQ values are much more similar between our historical and expertly-classified LSRs. For the sake of expediency and convenience, we will focus on maximum, Q90, and mean values for MaxUnitQ and MaxARI products for the rest of the analysis, but all other distributions are available in the Appendix.

Distributions of MaxUnitQ maximum values show fairly consistent coincidence between the expertly-classified and the historical datasets (see Figure 4.15. Still, as seen in Section 4.1, the historical LSRs tend to contain more lower MaxUnitQ values, while the expertly-classified dataset tends to favor higher UnitQ values. This slight bias is also expected by both the geographical bias of the expertly-classified dataset, as well as the selected events themselves, which are made up of mostly `considerable` events. Conversely, the historical dataset is composed of events from all over the CONUS, and contains much more samples, which statistically tend to be associated with more frequent, less extreme events. The same can be said for the distributions of MaxUnitQ Q90 in Figure 4.16, and MaxUnitQ mean values in Figure 4.17.

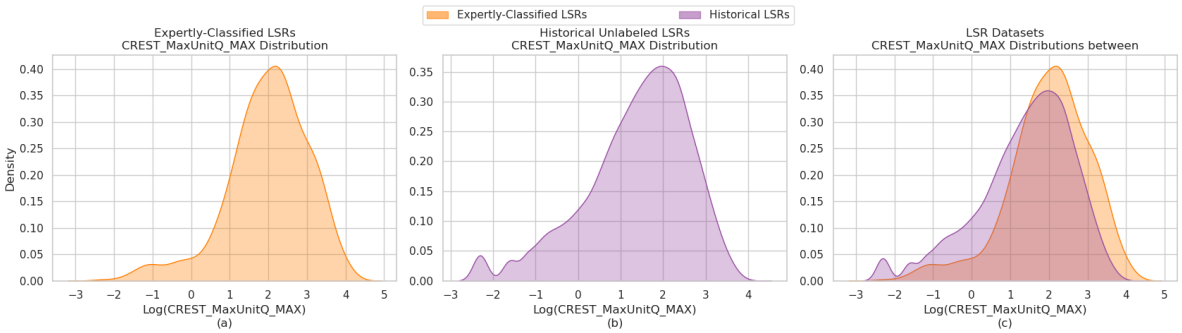


Figure 4.15: Distribution of maximum MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

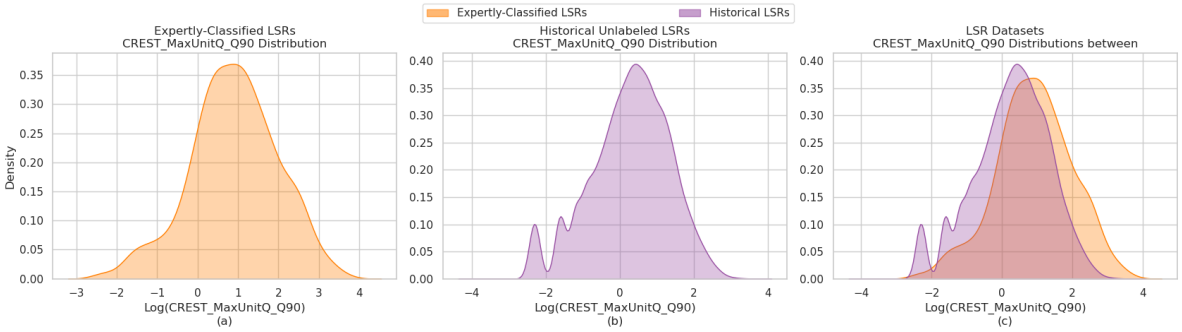


Figure 4.16: Distribution of Q90 MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

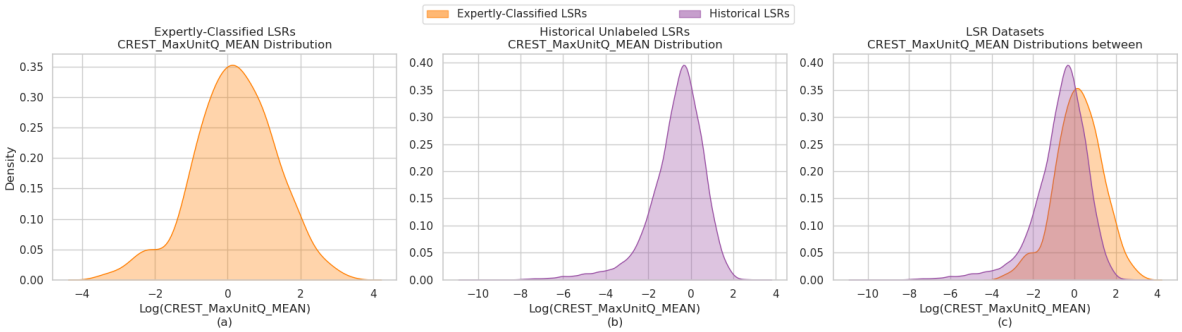


Figure 4.17: Distribution of mean MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

As was the case for MaxUnitQ values, their intersection with MaxARI values (MaxUnitQ ∩ MaxARI, where only MaxUnitQ values withing a non-zero MaxARI field are

contemplated) seem to present the same type of behavior between the distributions. These can be see in Figures 4.18, 4.19, and 4.20.

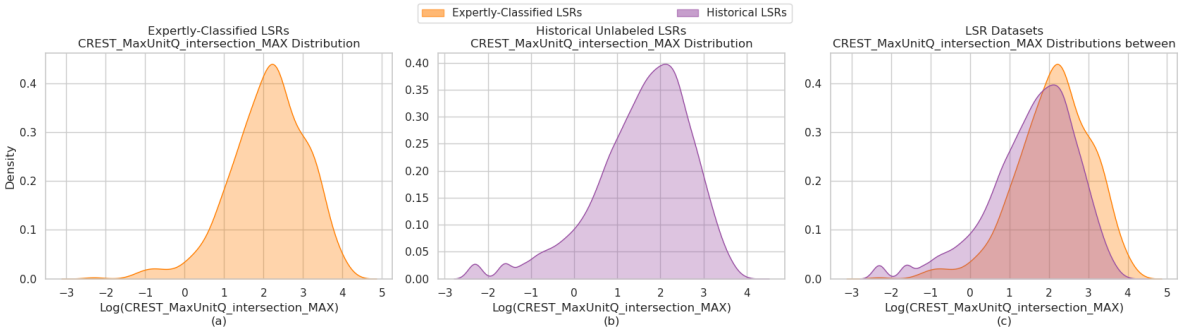


Figure 4.18: Distribution of maximum MaxUnitQ∩MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

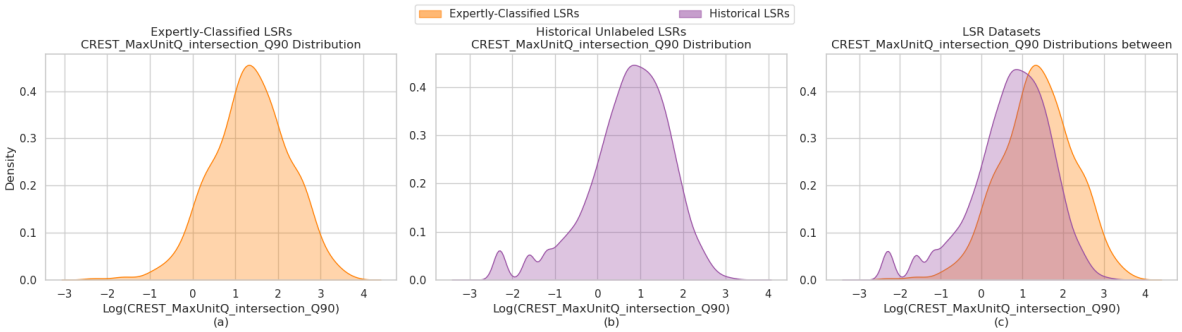


Figure 4.19: Distribution of Q90 MaxUnitQ∩MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

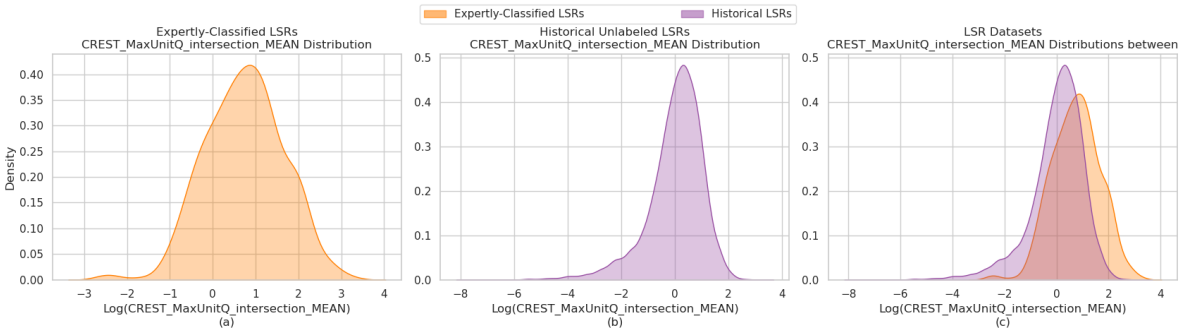


Figure 4.20: Distribution of mean MaxUnitQ∩MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

When comparing the distributions of MaxARI values for maximum, Q90, and mean values, concordance between the distributions is observed. In the case of MaxARI values, we can observed that instead of presenting a pronounced bias towards upper or lower skewness, here it is the mostly observed range of the MaxARI values that seems to be different between the data sets. Overall, the expertly-classified LSRs seem to have a wider spread of maximum and Q90 MaxARI values, as shown in Figures 4.21, and 4.22. In the case of maximum MaxARI values, the historical dataset seems to present a higher frequency of extreme high values of MaxARI, while in the case of Q90 MaxARI values, the same is true for lower MaxARI values. Lastly, when looking at the distribution of mean MaxARI values, while the distribution exhibits the same noticeable bias as the MaxUnitQ distributions, here the difference between the distributions is much less pronounced. This conveys that the distributions of mean MaxARI values across the two datasets exhibit the most similar behavior.

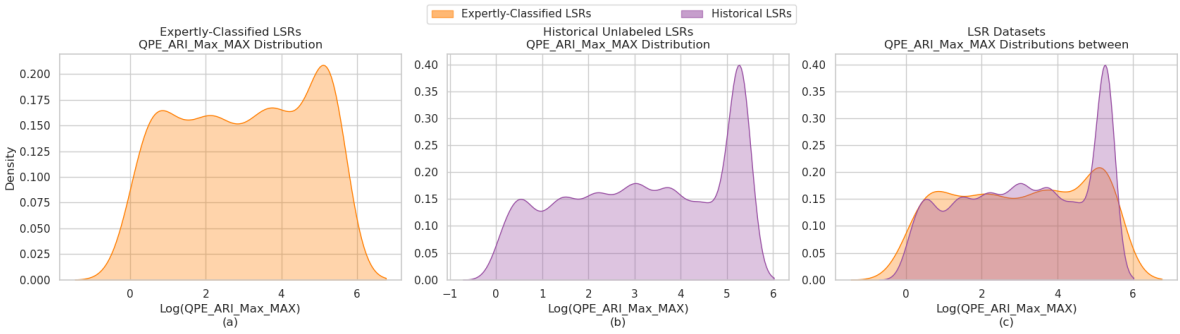


Figure 4.21: Distribution of maximum MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

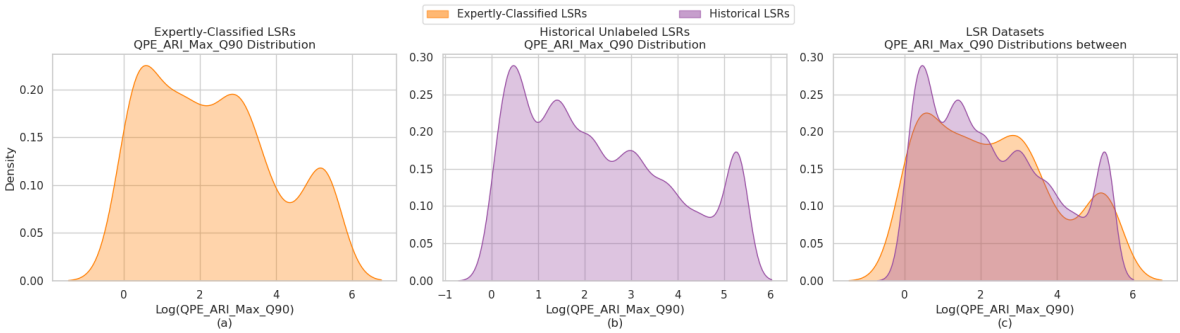


Figure 4.22: Distribution of Q90 MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

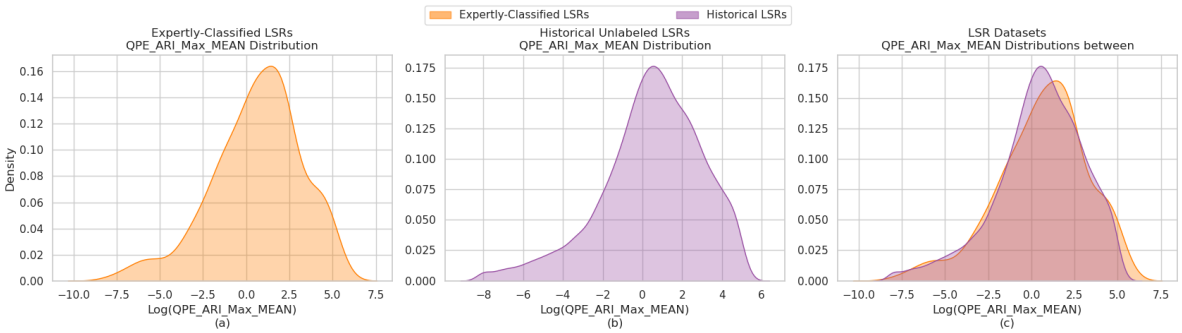


Figure 4.23: Distribution of mean MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

The observations made for the MaxARI distributions seem to carry over to their intersection with MaxUnitQ values (MaxARI ∩ MaxUnitQ, where only MaxARI values withing a non-zero MaxUnitQ field are contemplated). These behaviors can be see in Figures 4.24, 4.25, and 4.26.

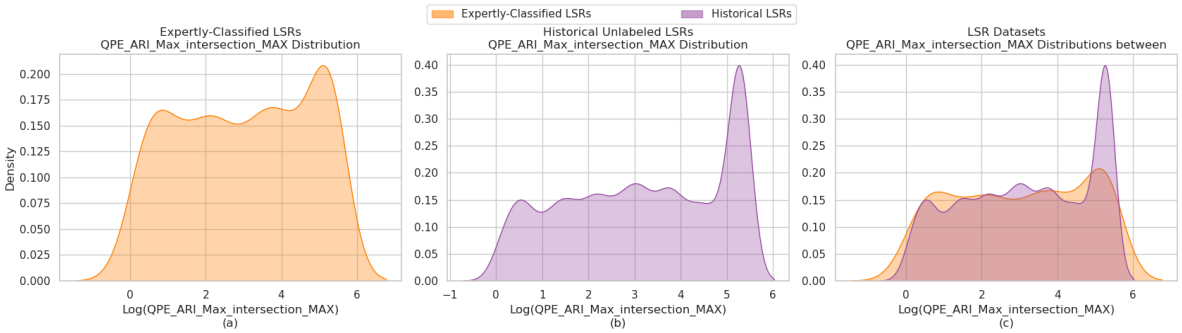


Figure 4.24: Distribution of maximum MaxARI∩MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

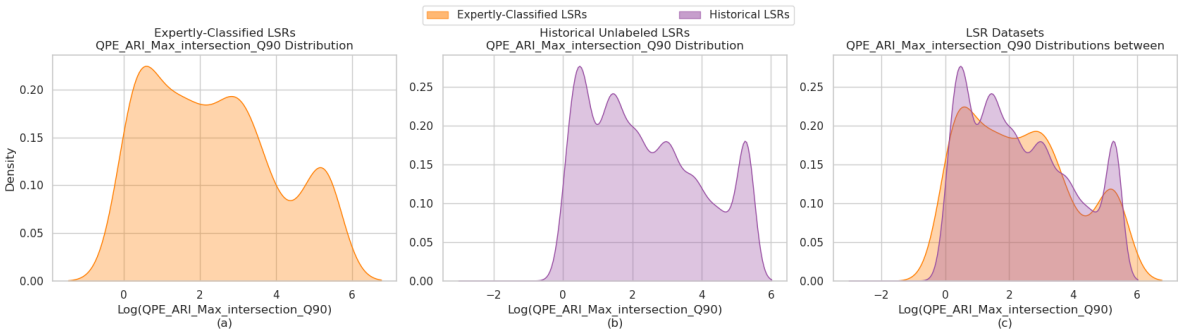


Figure 4.25: Distribution of Q90 MaxARI∩MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

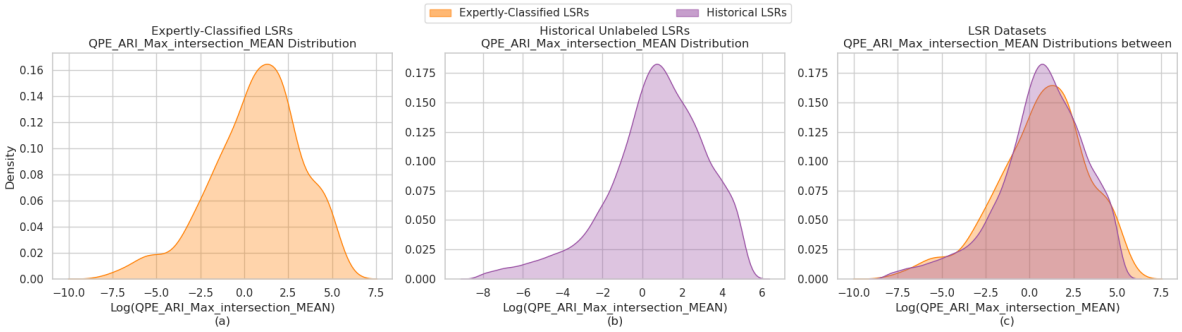


Figure 4.26: Distribution of mean MaxARI∩MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

As in Section 4.1.1, we do have access to impact-based warning categories in our expertly-classified dataset, we can also examine the extracted MaxUnitQ and MaxARI values with respect to each IBW class. These distributions are shown in Figures 4.27, 4.28, 4.29, and 4.30 below.

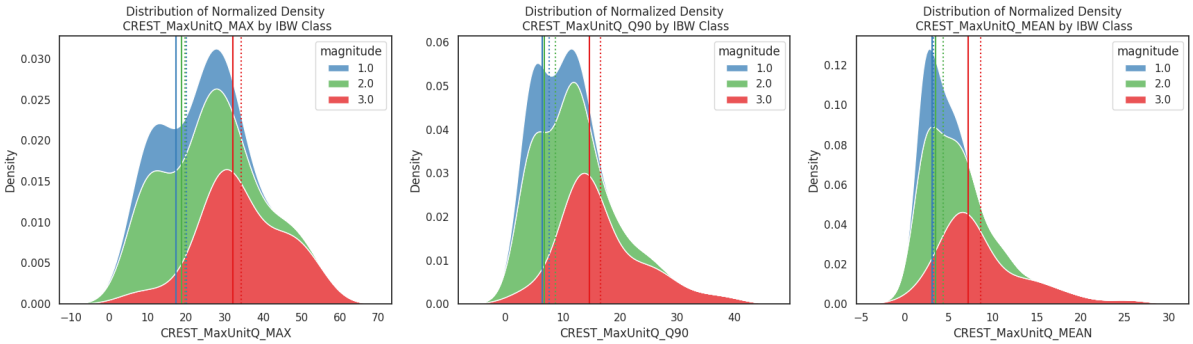


Figure 4.27: Distribution of MaxUnitQ values per IBW class. (a) Maximum, (b) Q90, and (c) Mean. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

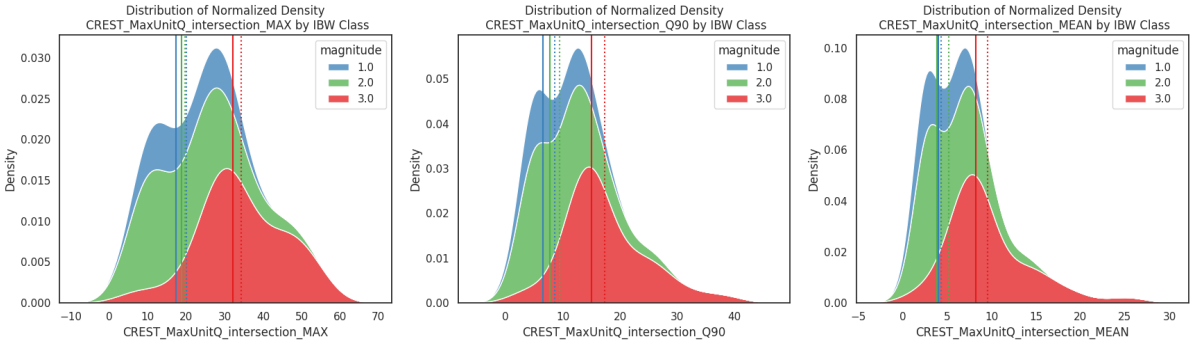


Figure 4.28: Distribution of MaxUnitQ∩MaxARI per IBW class. (a) Maximum, (b) Q90, and (c) Mean. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

By looking at the measures of central tendency, we can see that the medians for the distributions of maximum and Q90 MaxUnitQ values, are presented in increasing order of impact class, which is an improvement over our previous UnitQ matching method. However, a complete overlap of MaxUnitQ value ranges is observed between all the IBW classes. As they did in Section 3.1.1 , these MaxUnitQ distributions reflect the difficulty of forecasters to discern between classes, particularly between the `base` and the `considerable` classes.

From Figures 4.27 and 4.28, we can see that the MaxUnitQ distributions look very similar, with only slight changes in the location and magnitudes of the peaks for each class. The same can be said by examining Figures 4.29 and 4.30, where MaxARI dis-

tributions seem to behave very similar across the cropped and non-cropped extracted product values.

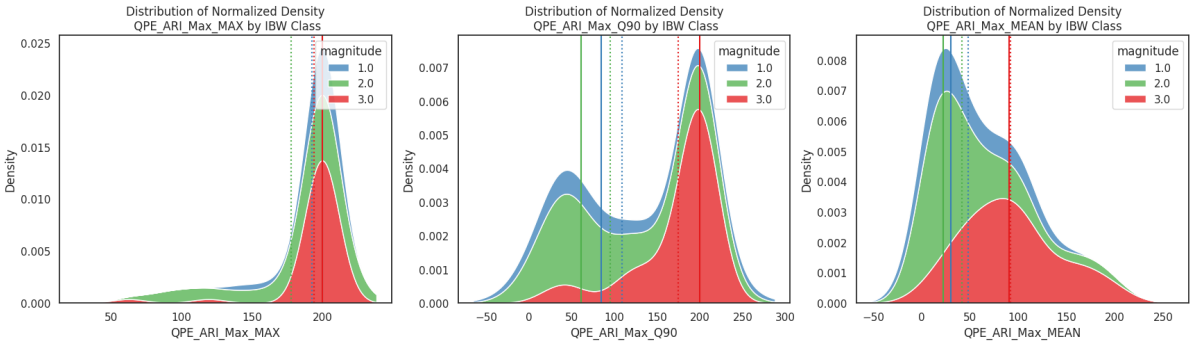


Figure 4.29: Distribution of MaxARI values per IBW class. (a) Maximum, (b) Q90, and (c) Mean. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

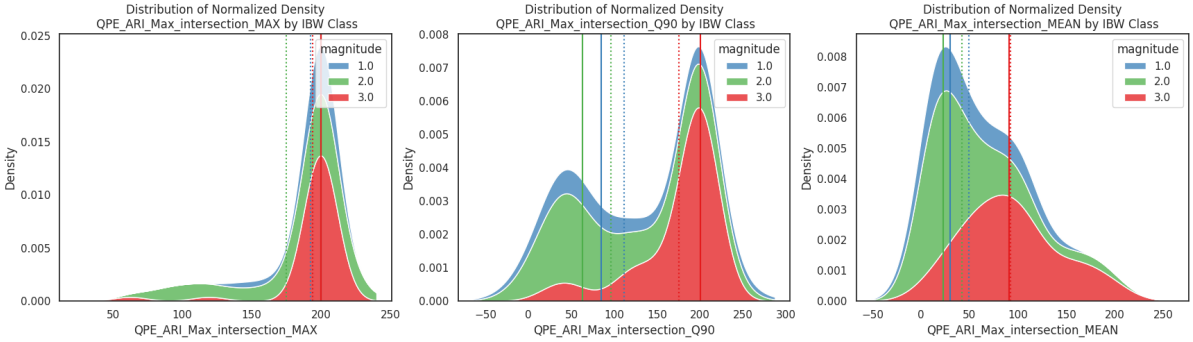


Figure 4.30: Distribution of MaxARI∩UnitQ per IBW class. (a) Maximum, (b) Q90, and (c) Mean. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

These MaxARI distributions also echo the difficulties seen from the MaxUnitQ data, where while higher average MaxARI values are indeed associated with the `catastrophic` class, the variability in MaxARI value ranges translates to a severe overlap across all the IBW classes.

4.2.2 FFSI V1

It took our python tool around 60 hours in total to process all 22,992 LSRs (22,329 expertly-classified + 663 historical unlabeled), querying the `ChatGPT-3.5-turbo` API endpoint (4K token context) with the FFSI V1 textualized definitions. Even though the paid API access allows for a 1 request-per-second rate for queries, all LSRs were processed sequentially, and each LSR was processed in about 5 seconds. Actual processing time should be realistically around half of the time reported ($\sim$30h), but additional delays were introduced when the process crashed or timed out while left running unattended, and was not restarted immediately after it had stopped.

To classify the 663 expertly-classified LSR dataset, our python implementation made 664 requests to the API ($\sim$0.15% additional requests overhead), over which the API processed 241,624 input tokens, generated 31,917 response tokens, and incurred in $0.43 USD of API access costs. To classify the 22,329 historical unlabeled LSR dataset, the python tool made a total of 23,332 requests ($\sim$4.4% additional requests overhead), over which the API processed 8,448,819 input tokens, generated 1,124,318 response tokens, and incurred in $14.92 USD of API access costs. Judging by the additional request overheads, our decision to implement batch processing for the API requests paid off. While processing the 663 expertly-classified reports, only one failure at the beginning of a batch occurred, since only one additional request was performed to process this data set. While processing the 22,329 historical reports, the tool processed 1,003 additional requests, which means the process suffered around 50 interruptions (since it processed around 50 additional overhead batches).

At this point in the development of the present dissertation, and for the first time to the author's knowledge, it has been possible to systematically assign impact classes and class probabilities to historical LSRs, based solely on their remarks. Figure 4.31 shows a similar plot to Figure 3.5, but now all these historical LSRs locations have been color-coded according to their FFSI score, as a direct representation for their most likely FFSI impact class.

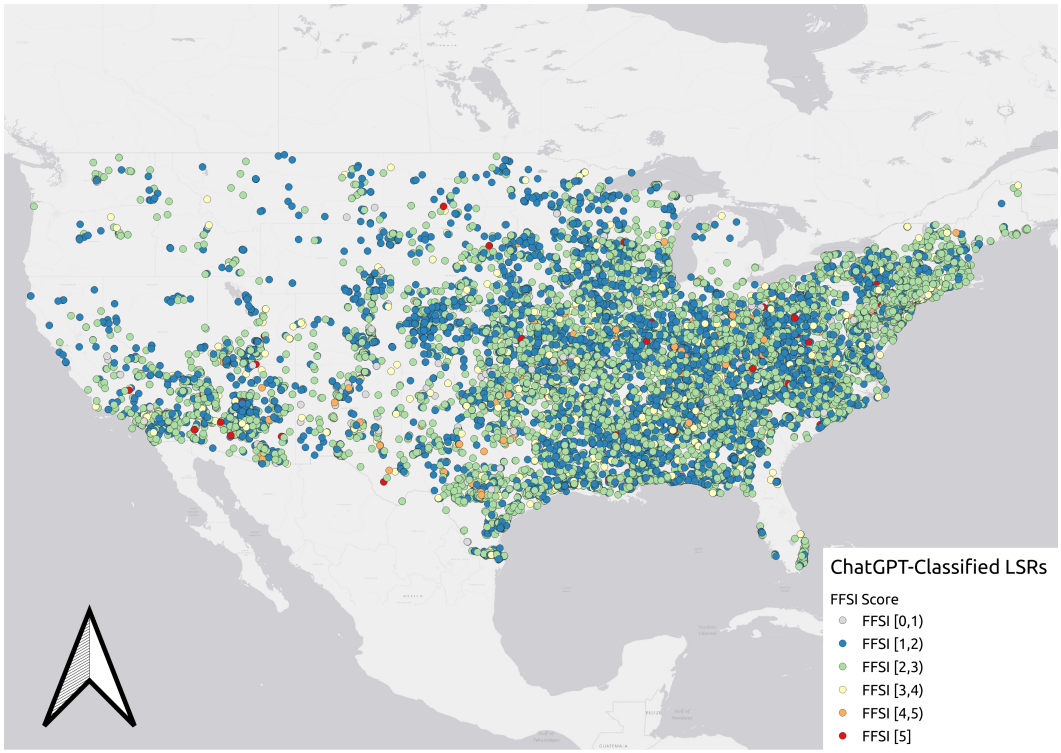


Figure 4.31: Visualization of the geographical distribution of the 22,329 ChatGPT-classified historical LSRs over the CONUS, colored by FFSI Score.

By having created this historical dataset of FFSI-classified LSRs, we come closer to overcoming the first of the two main challenges which are needed to address the present work's hypothesis: *building a reliable dataset of historical flash flood events with specific measures of impact*. To completely fulfill this task, we must evaluate ChatGPT's FFSI classification of these LSRs, as to establish a measure of reliability for this approach. For this purpose, we will rely on the expertly-classified LSRs to establish the expert's IBW classification as a performance baseline, and to compare our GPT-based FFSI classification to it. By analyzing both the IBW and FFSI class distributions over the previously extracted FLASH moments, we can assess whether our systematic FFSI classification matches the expert's performance. Firstly, by performing a correlation analysis between the classes for the two impact frameworks (IBW and FFSI) and the moments extracted, the correlations presented in Table 4.4 were observed.

	IBW	FFSI
IBW	1.000	0.333
FFSI	0.333	1.000
CREST_MaxUnitQ_MEAN	0.106	0.018
CREST_MaxUnitQ_VAR	-0.141	-0.023
CREST_MaxUnitQ_SKEW	0.034	-0.013
CREST_MaxUnitQ_KURT	-0.024	-0.033
CREST_MaxUnitQ_Q90	-0.008	0.052
CREST_MaxUnitQ_Q95	**0.386**	**0.285**
CREST_MaxUnitQ_Q99	0.374	0.270
CREST_MaxUnitQ_MAX	0.342	0.252
CREST_MaxUnitQ_intersection_MEAN	0.175	0.044
CREST_MaxUnitQ_intersection_VAR	-0.145	-0.033
CREST_MaxUnitQ_intersection_SKEW	0.018	-0.017
CREST_MaxUnitQ_intersection_KURT	0.007	-0.005
CREST_MaxUnitQ_intersection_Q90	0.140	0.077
CREST_MaxUnitQ_intersection_Q95	**0.125**	**0.062**
CREST_MaxUnitQ_intersection_Q99	0.147	0.052
CREST_MaxUnitQ_intersection_MAX	0.154	0.046
QPE_ARI_Max_MEAN	0.221	0.107
QPE_ARI_Max_VAR	-0.105	-0.009
QPE_ARI_Max_SKEW	-0.015	-0.012
QPE_ARI_Max_KURT	-0.058	-0.024
QPE_ARI_Max_Q90	**0.359**	**0.247**
QPE_ARI_Max_Q95	0.354	0.244
QPE_ARI_Max_Q99	0.319	0.243
QPE_ARI_Max_MAX	0.266	0.228
QPE_ARI_Max_intersection_MEAN	0.232	0.112
QPE_ARI_Max_intersection_VAR	-0.114	-0.011
QPE_ARI_Max_intersection_SKEW	-0.013	-0.013
QPE_ARI_Max_intersection_KURT	-0.060	-0.026
QPE_ARI_Max_intersection_Q90	**0.379**	**0.258**
QPE_ARI_Max_intersection_Q95	0.374	0.256
QPE_ARI_Max_intersection_Q99	0.339	0.254
QPE_ARI_Max_intersection_MAX	0.288	0.241

Table 4.4: IBW-FFSI correlations for the expertly-classified LSR dataset

From the correlation analysis shown in Table 4.4, we can see that for the MaxUnitQ values, Q95 seems to hold the highest correlation for both IBW and FFSI. Because of this, we will focus on the Q95 distributions for MaxUnitQ as the main point of comparison between the two impact frameworks on our expertly-classified data. All other expertly-classified data IBW vs FFSI distribution plots are included in the Appendix. By comparing how the distributions of MaxUnitQ Q95 values behave when looking at each individual impact class, we can compare how the ChatGPT+FFSI approach performs with respect to the experts' IBW classification. Figure 4.33 below portrays how the same 663 expertly-classified LSRs look through both the FFSI and the IBW impact perspectives.

Expertly-classified LSR Dataset

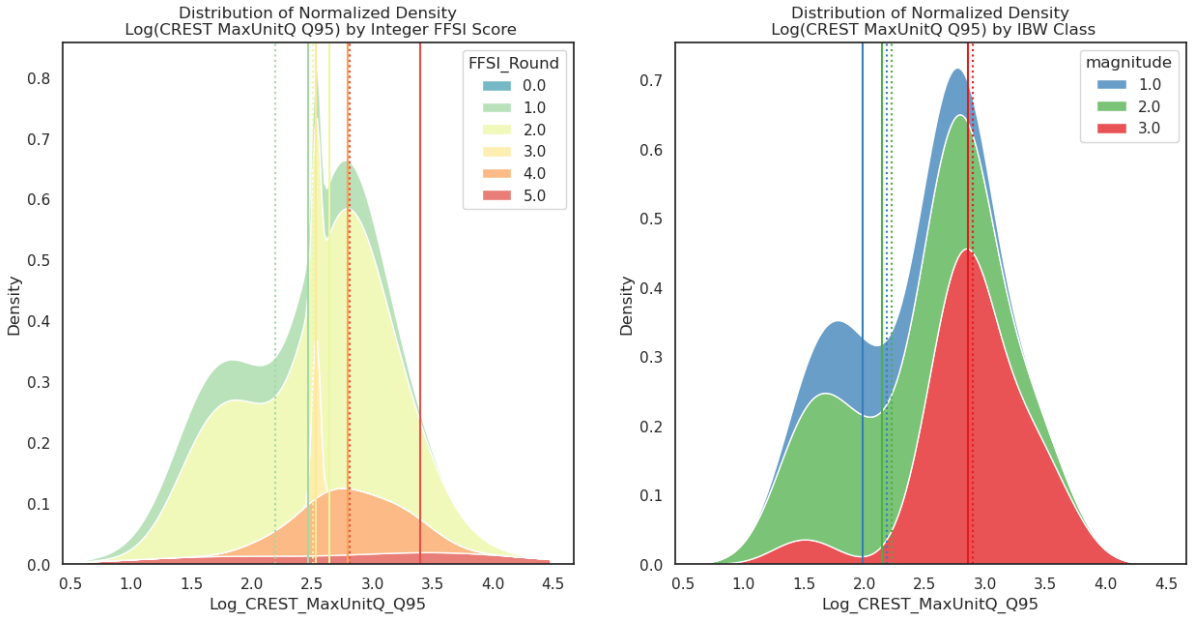


Figure 4.32: FFSI vs IBW MaxUnitQ Density Plots for the expertly-classified LSRs. (left) MaxUnitQ Q95 distributions by FFSI impact class (ChatGPT); (right) MaxUnitQ Q95 distributions by IBW impact class (experts). Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

While we had previously observed the class overlap in the IBW distributions, we can now see that this is also the case for the classification our ChatGPT+FFSI approach has produced. Notably, the distribution of medians for the new FFSI classification seems

to remain consistent with an increasing order of impacts as MaxUnitQ values increase, and similarly to the IBW case, the distance between medians for the medium and low impact classes are much closer together, than to the highest impact class. While visual inspection appears to show that our systematic, ChatGPT+FFSI approach achieved comparable performance to that of the experts using the IBW framework, a more quantitative set of tests will provide a more objective assessment of this performance.

In order to determine whether parametric or non-parametric test should be used to analyze these results, normality tests can be employed. In this case, python's *scipy* package offers a normality test based on D'Agostino and Pearson's test, which combines skew and kurtosis to produce an omnibus test of normality, with the following hypotheses:

- **H0:** Sample comes from a normal distribution
- **H1:** Sample does not come from a normal distribution
- If p-value < α: reject H0
- If p-value > α: can not reject H0

This normality test was applied over the whole expertly-classified dataset, as well as its subsets, which are defined by selecting those instances which correspond to specific FFSI and IBW classes. Results for this normality test are shown in Table 4.5.

NORMALITY TEST - Log(MaxUnitQ Q95)		
Subset	**Test Statistic**	**p-Value**
ALL DATA	6.6900	0.0353
FFSI 1	4.5650	0.1020
FFSI 2	4.8278	0.0895
FFSI 3	–	–
FFSI 4	0.0386	0.9809
FFSI 5	–	–
IBW 1	2.0955	0.3507
IBW 2	4.6177	0.0994
IBW 3	15.0092	0.0006

Table 4.5: Expertly-classified LSRs - FFSI vs IBW Normality Test. Note that for FFSI classes 3 and 5 the sample size was insufficient to calculate the statistic.

When contemplating the whole MaxUnitQ Q95 values from the 663 instances as a whole, the normality test indicates that the H0 is rejected with a 95% confidence (α=0.05), indicating that the data does not come from a normal distribution. Therefore, non-parametric tests will be used for subsequent explorations and assessments. However, when contemplating individual FFSI classes, MaxUnitQ Q95 values for classes `base`, `moderate`, and `severe`, H0 can not be rejected, as these seem to come from normal distributions. The same is true for IBW classes `base` and `considerable`, when contemplated individually. It should also be noted that the sample size of FFSI classes `serious` and `catastrophic` was insufficient to calculate the statistic.

Continuing with our non-parametric tests, in order to determine if multiple medians belonging to different groups in our dataset are statistically different, the Kruskal-Wallis test offers the following hypotheses:

- **H0:** The median is the same for all the data groups
- **H1:** The Median is not equal for all the data groups
- If p-value < α: reject H0
- If p-value > α: can not reject H0

We perform a Kruskal-Wallis test for both the GPT-based FFSI classification, and the expert-based IBW classification. Results for this test are shown in Table 4.6.

KRUSKAL-WALLIS TEST FOR Log(MaxUnitQ Q95)		
Subset	**Test Statistic**	**p-Value**
FFSI	7.6956	0.1034
IBW	31.7481	1.2764e-07

Table 4.6: Kruskal-Wallis Test results for Log(MaxUnitQ Q95) - Expertly-classified LSRs

The Kruskal-Wallis test results show that for the expertly-classified IBW classification, we can reject H0 with 95% confidence (α=0.05), while for the GPT-based FFSI classification, we can only reject H0 with a confidence interval smaller than 89% (α=0.11). These results show that, while the expert's classification shows to have different means with a 95% confidence interval, out FFSI classification only shows to have different means at a lower confidence interval of 89%.

The last of our non-parametric tests is Dunn's test (a posthoc test for the Kruskal-Wallis test), which relies on pairwise comparisons between each independent group's medians, and tells which groups are statistically significantly different from another at some level of alpha. For Dunn's test multiple adjustment types can be used to control the family-wise error rate of p-values, and can dramatically reduce the probability of committing a type I error among the set of multiple comparisons:

- *Bonferroni Adjustment*: `new p-value =` $p * m$, where `p` is the original p-value, and `m` is the total number of comparisons being made.
- *Sidak Adjustment*: `new p-value =` $1-(1-p)^m$, where `p` is the original p-value, and `m` is the total number of comparisons being made.

DUNN'S TEST FOR FFSI Log(MaxUnitQ Q95)					
No Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.071187	0.816603	0.012979	0.054026
FFSI 2	0.071187	1.000000	0.598893	0.159553	0.324748
FFSI 3	0.816603	0.598893	1.000000	0.273582	0.316707
FFSI 4	0.012979	0.159553	0.273582	1.000000	0.994209
FFSI 5	0.054026	0.324748	0.316707	0.994209	1.000000
Bonferroni Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.711867	1.0	0.129788	0.540263
FFSI 2	0.711867	1.000000	1.0	1.000000	1.000000
FFSI 3	1.000000	1.000000	1.0	1.000000	1.000000
FFSI 4	0.129788	1.000000	1.0	1.000000	1.000000
FFSI 5	0.540263	1.000000	1.0	1.000000	1.000000
Sidak Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.522158	1.000000	0.122464	0.426160
FFSI 2	0.522158	1.000000	0.999892	0.824165	0.980291
FFSI 3	1.000000	0.999892	1.000000	0.959086	0.977814
FFSI 4	0.122464	0.824165	0.959086	1.000000	1.000000
FFSI 5	0.426160	0.980291	0.977814	1.000000	1.000000

Table 4.7: Expertly-classified FFSI Dunn's test results

From the above Dunn's test results in Table 4.7, particularly from the Bonferroni-adjusted p-values (which is the most aggressive adjustment), we can see that the smallest p-value corresponds to the difference in medians between FFSI classes `base` and `severe`, and a statistically significant difference is only evidenced at a confidence interval of 87% ($\alpha = 0.13$). Additional testing has been performed on MaxUnitQ Q90 instead of Q95, and while the only statistically significant difference is still between FFSI classes `base` and `severe`, the confidence interval increases to around 93% ($\alpha = 0.063$).

Dunn's test was also performed for the expert IBW classification, which yielded the

results shown in Table 4.8.

DUNN'S TEST FOR IBW Log(MaxUnitQ Q95)			
No Adjustment			
	IBW 1	**IBW 2**	**IBW 3**
IBW 1	1.000000	7.082520e-01	3.042382e-05
IBW 2	0.708252	1.000000	1.483386e-07
IBW 3	0.000030	1.483386e-07	1.000000
Bonferroni Adjustment			
	IBW 1	**IBW 2**	**IBW 3**
IBW 1	1.000000	1.000000	9.127146e-05
IBW 2	1.000000	1.000000	4.450158e-07
IBW 3	0.000091	4.450158e-07	1.000000
Sidak Adjustment			
	IBW 1	**IBW 2**	**IBW 3**
IBW 1	1.000000	9.751673e-01	1.000000
IBW 2	0.975167	1.000000	0.999892
IBW 3	0.000091	4.450158e-07	1.000000

Table 4.8: Expertly-classified IBW Dunn's test results

Taking the IBW Dunn's test results from Table 4.8 in contrast to the FFSI results from table 4.7, the Bonferroni-adjusted p-values indicate that for the expert's IBW classification, classes `base` and `catastrophic` seem to have statistically significant differences in their medians, as well as classes `considerable` and `catastrophic` (both with confidence intervals of 95% and higher). However, medians between classes `base` and `considerable` don't seem to be different with any statistical significance. This once again brings back the issues previously discussed with IBW guidance, and the strong difficulties encountered by experts (forecasters) in distinguishing between the `base` and `considerable` classes.

Historical Flash Flood Impacts Dataset

A similar analysis has been performed for the historical LSR dataset, which now has been constructed into a historical dataset of flash flood impacts, in order to establish the statistical significance of each FFSI class' measures of central tendency.

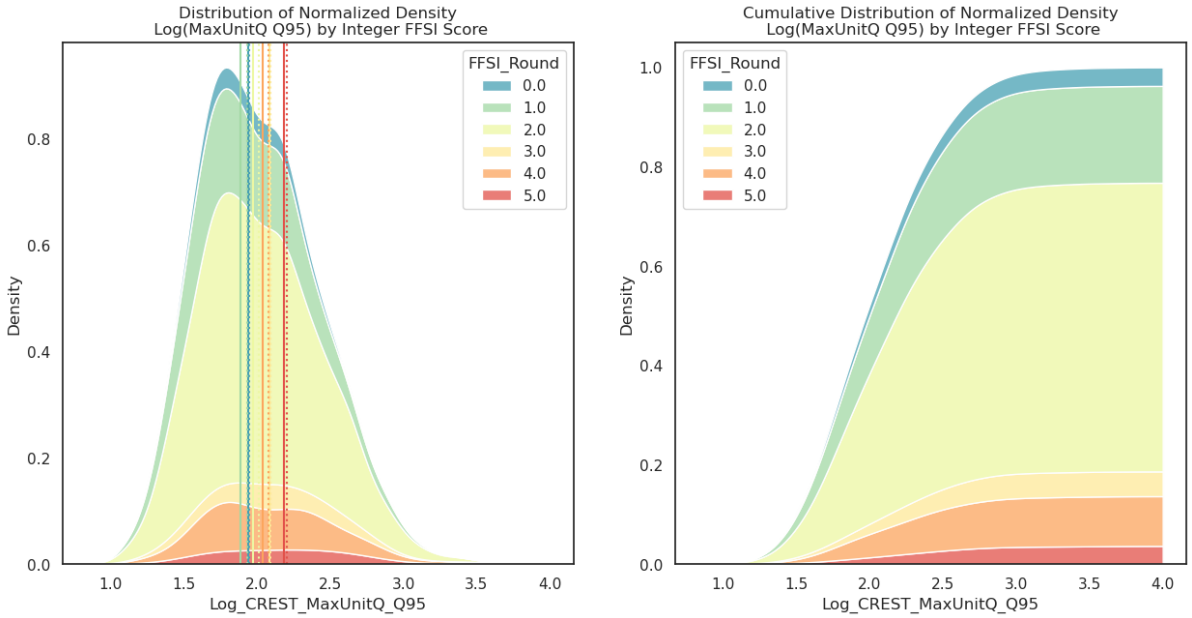


Figure 4.33: (left) MaxUnitQ Q95 distributions by FFSI impact class (ChatGPT); (right) MaxUnitQ Q95 cumulative distributions by FFSI impact class (ChatGPT). Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

From a correlation analysis between the FFSI classification an the extracted moments of FLASH products shown in Table 4.9, it can be observed that Q95 once again holds the highest correlation values for MaxUnitQ, but only the non-intersection extraction).

	FFSI
FFSI	1.000
CREST_MaxUnitQ_MEAN	-0.018
CREST_MaxUnitQ_VAR	0.026
CREST_MaxUnitQ_SKEW	-0.019
CREST_MaxUnitQ_KURT	-0.008
CREST_MaxUnitQ_Q90	-0.002
CREST_MaxUnitQ_Q95	**0.143**
CREST_MaxUnitQ_Q99	0.142
CREST_MaxUnitQ_MAX	0.133
CREST_MaxUnitQ_intersection_MEAN	-0.019
CREST_MaxUnitQ_intersection_VAR	0.035
CREST_MaxUnitQ_intersection_SKEW	-0.019
CREST_MaxUnitQ_intersection_KURT	0.006
CREST_MaxUnitQ_intersection_Q90	**0.004**
CREST_MaxUnitQ_intersection_Q95	0.000
CREST_MaxUnitQ_intersection_Q99	0.001
CREST_MaxUnitQ_intersection_MAX	0.000
QPE_ARI_Max_MEAN	0.077
QPE_ARI_Max_VAR	0.013
QPE_ARI_Max_SKEW	-0.043
QPE_ARI_Max_KURT	-0.033
QPE_ARI_Max_Q90	**0.103**
QPE_ARI_Max_Q95	0.100
QPE_ARI_Max_Q99	0.094
QPE_ARI_Max_MAX	0.087
QPE_ARI_Max_intersection_MEAN	0.075
QPE_ARI_Max_intersection_VAR	0.013
QPE_ARI_Max_intersection_SKEW	-0.042
QPE_ARI_Max_intersection_KURT	-0.034
QPE_ARI_Max_intersection_Q90	**0.104**
QPE_ARI_Max_intersection_Q95	0.102
QPE_ARI_Max_intersection_Q99	0.097
QPE_ARI_Max_intersection_MAX	0.092

Table 4.9: FFSI correlations for the historical LSR dataset

From the normality test results shown in Table 4.10, we can conclude that since the p-value for the whole dataset is much smaller than an $\alpha = 0.05$, within a 99% confidence interval, the data does not seem come from a normal distribution. Analyzing each independent FFSI class' results, we can also see that the above is also true for all classes, except for FFSI class `serious`, for which we can only reject H0 with a $\sim 90\%$ confidence interval.

NORMALITY TEST - Log(MaxUnitQ Q95)		
Subset	**Test Statistic**	**p-Value**
ALL DATA	75.8163	3.4411e-17
FFSI 1	15.2944	0.0005
FFSI 2	39.9363	2.1278e-09
FFSI 3	4.4630	0.1074
FFSI 4	7.1513	0.0280
FFSI 5	7.6523	0.0218

Table 4.10: Historical LSRs - FFSI Normality Test.

Therefore, as we did for the expertly-classified dataset,in order to assess whether the medians for each class are different within statistical significance, a Kruskal-Wallis non-parametric test can be performed. These results are shown in Table 4.11.

KRUSKAL-WALLIS TEST FOR Log(MaxUnitQ Q95)		
Subset	**Test Statistic**	**p-Value**
FFSI	37.5064	1.4166e-07

Table 4.11: Kruskal-Wallis Test results for Log(MaxUnitQ Q95) - Historical LSRs

The Kruskal-Wallis test results indicate that since the p-value is much smaller than an $\alpha = 0.05$, we can reject H0, therefore accepting the alternative that the medians are not equal for all the data groups.

Exploring further the pairwise differences in medians using Dunn's test, we can discern which measures of central tendency are different from one another within statistical significance. Table 4.12.

DUNN'S TEST FOR FFSI Log(MaxUnitQ Q95)					
No Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.000737	0.000171	0.000047	0.000002
FFSI 2	0.000737	1.000000	0.031653	0.040963	0.000634
FFSI 3	0.000171	0.031653	1.000000	0.579613	0.221652
FFSI 4	0.000047	0.040963	0.579613	1.000000	0.061050
FFSI 5	0.000002	0.000634	0.221652	0.061050	1.000000
Bonferroni Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.007372	0.001713	0.000468	0.000018
FFSI 2	0.007372	1.000000	0.316532	0.409626	0.006345
FFSI 3	0.001713	0.316532	1.000000	1.000000	1.000000
FFSI 4	0.000468	0.409626	1.000000	1.000000	0.610500
FFSI 5	0.000018	0.006345	1.000000	0.610500	1.000000
Sidak Adjustment					
	1	**2**	**3**	**4**	**5**
FFSI 1	1.000000	0.007347	0.001712	0.000468	0.000018
FFSI 2	0.007347	1.000000	0.275048	0.341804	0.006327
FFSI 3	0.001712	0.275048	1.000000	0.999828	0.918391
FFSI 4	0.000468	0.341804	0.999828	1.000000	0.467371
FFSI 5	0.000018	0.006327	0.918391	0.467371	1.000000

Table 4.12: Historical FFSI Dunn's test results

Based on the Bonferroni-adjusted p-values of the Dunn's Test, and contemplating a 95% confidence interval, the medians between FFSI class `minor` is statistically significantly different from all other classes; class `moderate` is different from classes `minor` and `catastrophic`; class `serious` is only different from class `minor`; class `severe` is only different from class `minor`; and class `catastrophic` is different from classes `minor` and `moderate`.

* * *

As can be seen from the above statistical results comparing both the expertly-classified dataset's IBW and FFSI classifications, our GPT-based FFSI classification seems to preserve a non-normal distribution, albeit with a less certain confidence interval. However, from the Dunn's test results, we can see that while the expert's classification presents a significant difference for the `catastrophic` class, our GPT-based FFSI classification shows only statistically significant differences between `base` and `severe` classes. Overall, while our model's performance does not exceed the performance shown by the experts in classifying LSRs, given some leniency on the fact that these reports were classified systematically without human intervention, and using an FFSI template, we can state that the ChatGPT+FFSI method provides a comparable performance to that of the experts.

4.3 FFSI-Based Flash Flood Impact Model

As described in Section 3.2, a proof-of-concept framework has been defined, a reduced spatial subdomain over the state of Kentucky (KY) has been designated , and a spatialized flash flood impacts training dataset has been assembled from static and dynamic attributes which describe each LSR event. This has led to the experimentation and training of multiple distinct transformer-based machine learning models, which address specific characteristics of our training data, with respect to the main goals this dissertation looks to fulfill.

This section will present the results obtained from training the machine learning models described in Sections 3.3.1, 3.3.2, and 3.3.3 respectively. When examining these results, bear in mind that, while the Kentucky subdomain holds out 779 non-null, non-edge LSRs, whenever one of the model ingests these LSR training samples, each layer is always cropped to a 128x128 pixel region, centered around the LSR's location (this is the reason why only non-edge cases were preserved), as this 128x128 buffer size, is the effective domain over which these models operate).

4.3.1 Segmentation Transformers

As mentioned in Section 3.3.1, the Segformer model was trained on the 46 input layers of our 779 training samples, treating all data as static (i.e. not taking into account the

Metric	Score
Loss	1.9261
Accuracy	0.0046
F1 Score	0.0046
Precision	0.0046
Recall	0.0046

Table 4.13: Segformer Model - Training Metrics

FLASH product time dependence). Training was performed for several epochs, and the best results are summarized in Table 4.13 below:

As can be seen from these training metrics, this model is not able to extract any meaningful representation by aggregating all static and dynamic layers, to classify our inputs onto a spatialized FFSI score region within the domain. This is further evidenced, when we look at some of the training samples, paired with the predictions made by the model. These are shown in Figure 4.34.

(a) FFSI 1 (b) FFSI 2 (c) FFSI 3 (d) FFSI 4 (e) FFSI 5

Figure 4.34: Various Segformer training samples - Each subfigure above shows a different sample from a different FFSI class (left), paired with its corresponding generated training output by the Segformer (right).

As can be seen from Figure 4.34, the Segformer model lacks any skill in reliably predicting the appropriate class shown on the left panels (a single region of FFSI score), and while multiple classes are predicted upon the 128x128 subdomain region within the KY domain, and they seem to show some spatial features and behaviors, the labels where our input class is expected to be located don't match the inputs. Additionally, training the same model using only the reduced subset of 153 complete cases (no missing dynamic data layers), leads to exactly the same performance metrics.

After considering the above results, it has become clear that while Segformers can be used for image classification, just by virtue alone of the spatial features in our data, a model for impact annotations is not feasible with this architecture.

4.3.2 Video Masked Autoencoders

As mentioned in Section 3.3.2, the VideoMAE model was trained on only the dynamic input layers of our 779 training samples, treating all data as 18 frames of two-channel video. The representations of the dynamic input data are then passed onto a deep classification head, which predicts a given FFSI class for the input data. Training was performed for several epochs, and the best results are summarized in Table 4.14 below:

Metric	Score
Loss	1.6913
Accuracy	0.1946
F1 Score	0.1946
Precision	0.1946
Recall	0.1946

Table 4.14: VideoMAE Model - Training Metrics

While these results look somewhat better than those obtained by the Segformer model, they still reflect rather poor performance. However, these very low values may hint at the fact that the model may be picking up some underlying signal from our data, but issues in our data like missing values may be playing a role in hindering its skill.

Additional training was performed with the reduced set of 153 complete cases (no missing dynamic data layers), and training metrics for this model are shown in Table 4.15.

Metric	Score
Loss	0.7234
Accuracy	0.3223
F1 Score	0.3223
Precision	0.3223
Recall	0.3223

Table 4.15: VideoMAE Model - Complete Cases Training Metrics

In this case, performance shows a considerable improvement. Particularly, looking at accuracy and F1 score presents a completely different panorama from results seen with previous attempt and the Segformer model. These results show that the model is able to pick up *some* of the underlying signal and features that stem from considering the time dimension in the succession of 10-minute FLASH data used to train the model. This means that by taking into account the temporal dimension of our dynamic data, models like this VideoMAE may be able to extract useful patterns to predict and annotate flash flood impacts. Also, this brings to light the fact that relying on sequential data which has missing values will dramatically impact our ability to train a skillful model.

4.3.3 VideoMAE + Visual Transformers

As mentioned in Section 3.3.3, a hybrid VideoMAE + ViT model was trained, where the VideoMAE component deals with only the dynamic input layers of our 779 training samples as 18 frames of two-channel video, and the ViT component deals with the static input layers as a 10-channel image. The representations from both the static and dynamic components are then concatenated and piped trough a simple classifier, which predicts a given FFSI class for the input data. Training was performed for several epochs, and the best results are summarized in Table 4.16 below:

Metric	**Score**
Loss	0.8779
Accuracy	0.4559
F1 Score	0.4559
Precision	0.4559
Recall	0.4559

Table 4.16: VideoMAE+ViT Model - Training Metrics

These results show a marginal increase in values with respect to the VideoMAE model which only ingested the dynamic data, which indicates that the representations of static layers presented by the ViT model are positively contributing to our model's classification skill. However, the model still struggles to *zero-in* on the signal in our

training data. To the point where it is unable to memorize the training dataset (overfitting).

Since improvement was observed in the previous models, additional training was performed with the reduced set of 153 complete cases (no missing dynamic data layers), and training metrics for this model are shown in Table 4.17.

Metric	**Score**
Loss	0.1316
Accuracy	0.8512
F1 Score	0.8512
Precision	0.8512
Recall	0.8512

Table 4.17: VideoMAE+ViT Model - Complete Cases Training Metrics

These results are a dramatic departure from previous experiments. By performing a simple experiment with a hybrid architecture, making sure our dynamic training data has no missing frames, and by leveraging the latent representations in our static data layers, we are now able to train a model that overfits. This conveys the notion that, not only there is *usable signal* in our training data, but our model can actually start to memorize said training data.

* * *

Recollecting some of the results presented at the end of this chapter, we can list the following highlights:

- Adequately capturing the temporal nature of our dynamic data layers is of paramount importance to constructing models for predicting flash flood impacts.
- Static data layers play a crucial role in enriching our dynamic data with additional features, however, not by virtue alone of spatial features can we reliably model flash flood impacts.
- By combining the VideoMAE and ViT architectures for extracting features from our dynamic and static data respectively, it is possible to train a simple machine

learning classifier which demonstrates the potential usefulness of our novel historical flash flood impacts data.

Chapter 5

Discussion

Having presented comprehensive results for our methods in Chapter 4, we can now discuss some of the implications of these results. But first, let's recall the current dissertation's hypothesis and main objectives.

The proposed hypothesis / research statement offered at the beginning of this document is "*Given an operational flash flood forecast, it is possible to annotate specific threat tags that inform users (forecasters) of possible associated impact expected for the forecasted event*". And the two main objectives / challenges defined are the following:

1. a data set which can relate historical flash flood events with specific measures of impact must be built, and
2. a proof-of-concept model capable of producing impact probabilities based on real-time measures of precipitation and flow response (among other variables) must be designed and implemented.

Let us first address objective number one: the creation of a historical flash flood impacts dataset. Starting with two distinct sets of LSR data, a historical unlabeled one, and an expertly-classified one, multiple approaches were attempted to consolidate said dataset.

At first, and without much success, a pre-trained language model methodology was tried, relying on a small amount of expertly-classified IBW LSRs. It was determined that, even though the pre-trained language models seemed to be able to effec-

tively convert our LSR remarks into actionable latent representations, the amount of labeled instances was insufficient to train a deep learning LSR classification model. Further attempts of enriching this dataset with post-survey data from StormDat reports yielded diminishing returns, as the relationships between these new attributes, our initial FLASH UnitQ extraction and matching, and our IBW classification did not offer a clear separable behavior that our classification model could tie into to discern between our impact classes.

By reversing our problem, and instead of relying on a few hundred impact-labeled instances, when making use of a language-based flash flood severity index (FFSI) classification in conjunction with large language models, we were able to capitalize on using the unlabeled historical LSR dataset, which is two orders of magnitude larger than the expertly-classified dataset. By employing FFSI as our framework for classifying both historical and expertly-classified IBW LSRs using solely their remarks, we were able to leverage the ChatGPT API to prompt a GPT-3.5-turbo instance into classifying our reports into probabilistic FFSI classes, for which we also calculated an *FFSI score* which collapses joint class probabilities into a single numerical value. Through this method, it was possible to systematically classify 22,329 historical LSRs between 2018-2022 into FFSI impact classes, therefore fulfilling objective number one of this dissertation. Further statistical testing was performed on this novel dataset, and we were able to show that, while the confidence with which our GPT-based FFSI classification method is lower than the level of confidence offered by the expert IBW classification, overall performance between the datasets is comparable, particularly when taking into account that the 22,329 historical FFSI LSRs were coherently classified without human intervention.

Now let us address objective number two: the development of a proof-of-concept model which is capable of producing flash flood impact probabilities based on real-time measures of precipitation and flow response.

For this proof-of-concept, a limited subdomain within the MRMS CONUS domain was first established as a 10km buffer around the minimum bounding box encasing the state of Kentucky (KY). This reduced our effective modeling area from around 10 million pixels, to just over 218,000. Then, all historical and expertly-classified events that were reported for the state of KY were identified, and specific constraints were enforced to filter out some of these events: all events with an FFSI score less than 1 were not contemplated, as well as all of those LSRs for which a 128x128 pixel buffer

centered around the LSR's location could not be constructed. This was done because several novel transformer architectures were used, and these required a reduced overall dimensionality of our spatial subdomain to be run effectively. The practical implication of this decision is that under the present form, if this methodology were to be transferred to the CONUS scale, our models would only be able to predict or annotate flash flood impacts at a 128x128 pixel resolution over the CONUS. This implies that while FLASH and MRMS operate at a $1km^2$ resolution, our impact predictions would only operate at a $128km^2$ resolution. However, considering that this type of applications and models, to the our knowledge, have never been implemented before, a coarse starting resolution for a flash flood impacts annotation model could be an acceptable compromise.

A spatialized training dataset was build with the flash flood impacts LSR dataset at its core, made up of static geomorphological and vulnerability layers, as well as sequences of dynamic FLASH product outputs: MaxUnitQ and MaxARI. These products represent the flow response and the causative rainfall respectively, for a given flash flooding event reported as an LSR. Spatialized class labels were also derived from the FFSI classifications obtained when constructing the flash flood impacts dataset.

Three different transformer-based architectures were trained, which increasingly tested for specific aspects of our data. Our results for these three models yielded increasingly improving training metrics as 1) the spatial dependence between dynamic FLASH data as incorporated into the model, 2) features derived from static data were used in conjunction with features from dynamic data, and 3) exclusively dynamic training samples without missing frames were used to train our models. Ultimately a hybrid architecture which mixed a VideoMAE model and a ViT model trained on only complete cases for the dynamic data, yielded surprising results showing that our model was being overfitted. This is a relevant aspect of this experiment, since being able to overfit a model shows that, not only there is signal in our data (the novel flash flood impacts dataset is *useful*), but also our model can pick up on that signal an memorize it. This shows room for improvement in terms of architecture tuning and parametrization, which can lead to highly performant models, to be tested in future experimental settings. Now, while this model's spatialized output is in fact a numerical quantity (i.e. the FFSI score as a class label), the FFSI score is a direct representation of the distributions of joint FFSI probability. Thus, it was possible to train a proof-of-concept machine learning model which was capable of producing flash flood impact probabilities from MaxUnitQ and MaxARI inputs.

Having addressed objectives one and two, let us discuss the main research question: *Given an operational flash flood forecast, it is possible to annotate specific threat tags that inform users (forecasters) of possible associated impact expected for the forecasted event*. Since MaxUnitQ is a flash flood forecast product, and specific FFSI class labels are being predicted by our VideoMAE+ViT machine learning model, these flash flood forecasts are being annotated with threat tags. And more relevantly, these threat tags are derived from a flash flood impact framework, which would readily allow a forecaster to translate that FFSI class label into the associated impacts of that specific class' severity. Therefore, this dissertation's hypothesis can not be rejected, and there is strong evidence to support its claim.

5.1 Future work

As briefly mentioned in the closing of Section 1.3.2, the dichotomy between the existence of two distinct flash flood impact frameworks (IBW and FFSI) provides an opportunity to perform foundational research at the intersection of the two. As shown in Figure 5.1 by exploring in detail the relationships that exists between FLASH UnitQ and each of these impact frameworks, a translation framework between the two would ultimately enable a consistent estimation of impacts levels, whether based on impact observations or streamflow forecasts.

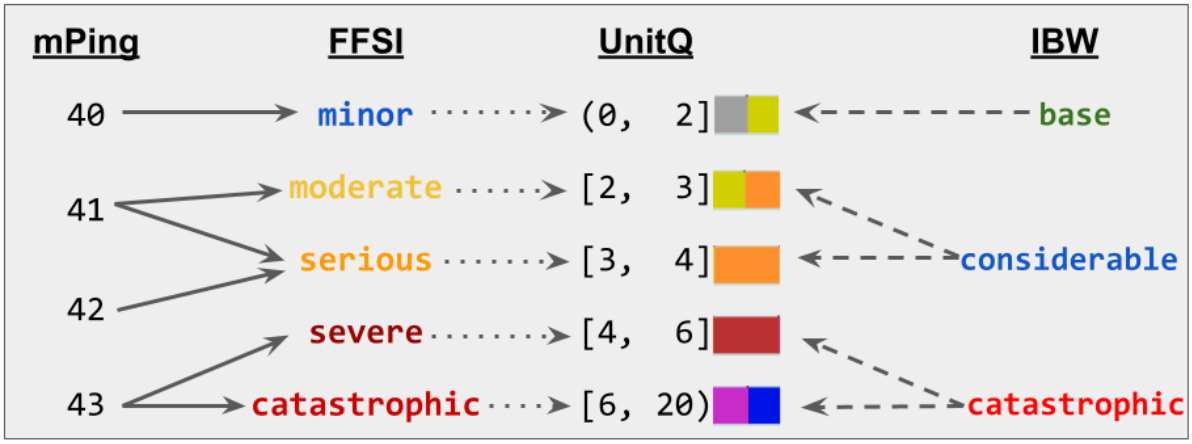


Figure 5.1: Example of a proposed UnitQ-based equivalency between distinct flash flood impact frameworks, like FFSI and IBW.

Lastly, now that a proof-of-concept model has shown feasibility in predicting flash

flood impacts over a reduced subdomain, a CONUS-wide implementation of 128x128 pixel impact annotations must be pursued. This will require training a similar architecture to the VideoMAE+ViT model demonstrated here, but with a much larger set of both training instances, as well as a much larger event subdomain. Particularly for working at the CONUS scale, this implies that much more computation power will be required to work in a domain made up of 10 million pixels. Also, once this model is trained, and ready for testing in experimental mode but in real time, robust data preprocessing pipelines must be engineered to feed FLASH MaxUnitQ and MaxARI in a timely fashion to the model at inference time.

Bibliography

[1] Guy Delrieu, John Nicol, Eddy Yates, Pierre-Emmanuel Kirstetter, Jean-Dominique Creutin, Sandrine Anquetin, Charles Obled, Georges-Marie Saulnier, Véronique Ducrocq, Eric Gaume, Olivier Payrastre, Hervé Andrieu, Pierre-Alain Ayral, Christophe Bouvier, Luc Neppel, Marc Livet, Michel Lang, Jacques Parent du-Châtelet, Andrea Walpersdorf, and Wolfram Wobrock. The catastrophic flash-flood event of 8–9 september 2002 in the gard region, france: A first case study for the Cévennes–Vivarais mediterranean hydrometeorological observatory. *Journal of Hydrometeorology*, 6(1):34–52, February 2005. doi: 10.1175/jhm-400.1.

[2] Manabendra Saharia, Pierre-Emmanuel Kirstetter, Humberto Vergara, Jonathan J. Gourley, Yang Hong, and Marine Giroud. Mapping flash flood severity in the united states. *Journal of Hydrometeorology*, 18(2):397–411, February 2017. doi: 10.1175/jhm-d-16-0082.1.

[3] NOAA National Severe Storms Laboratory. About nssl, 2021. URL `https://www.nssl.noaa.gov/about/`.

[4] Jonathan J. Gourley, Zachary L. Flamig, Humberto Vergara, Pierre-Emmanuel Kirstetter, Robert A. Clark, Elizabeth Argyle, Ami Arthur, Steven Martinaitis, Galateia Terti, Jessica M. Erlingis, Yang Hong, and Kenneth W. Howard. The FLASH Project: Improving the Tools for Flash Flood Monitoring and Prediction across the United States. *Bulletin of the American Meteorological Society*, 98(2):361–372, February 2017. ISSN 0003-0007, 1520-0477. doi: 10.1175/BAMS-D-15-00247.1.

[5] Jian Zhang, Kenneth Howard, Carrie Langston, Brian Kaney, Youcun Qi, Lin Tang, Heather Grams, Yadong Wang, Stephen Cocks, Steven Martinaitis, Ami Arthur, Karen Cooper, Jeff Brogden, and David Kitzmiller. Multi-Radar Multi-Sensor (MRMS) Quantitative Precipitation Estimation: Initial Operating Capabil-

ities. *Bulletin of the American Meteorological Society*, 97(4):621–638, April 2016. ISSN 0003-0007, 1520-0477. doi: 10.1175/BAMS-D-14-00174.1.

[6] NOAA Weather Program Office. Joint Technology Transfer Initiative Program. https://wpo.noaa.gov/Programs/JTTI, November 2021.

[7] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc, Beijing [China] ; Sebastopol, CA, second edition edition, 2019. ISBN 978-1-4920-3264-9.

[8] Alan J. Izenman. *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning (Springer Texts in Statistics)*. Springer, 2013. ISBN 0-387-78188-9.

[9] Benjamin S. Blanchard. *System Engineering Management*. Wiley Series in Systems Engineering and Management. Wiley, Hoboken, N.J, 4th ed edition, 2008. ISBN 978-0-470-16735-9.

[10] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques (the Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2011. ISBN 0-12-374856-9.

[11] Insitute for Operations Research and the Management Sciences. Operations Research & Analytics, 2021. URL `https://www.informs.org/Explore/Operations-Research-Analytics`.

[12] OpenAI. ChatGPT, 2023. URL `https://chat.openai.com`.

[13] Jessica López Espejel, El Hassane Ettifouri, Mahaman Sanoussi Yahaya Alassan, El Mehdi Chouham, and Walid Dahhane. GPT-3.5, GPT-4, or BARD? Evaluating LLMs Reasoning Ability in Zero-Shot Setting and Performance Boosting Through Prompts, September 2023. URL `http://arxiv.org/abs/2305.12477`. arXiv:2305.12477 [cs].

[14] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need,

August 2023. URL http://arxiv.org/abs/1706.03762. arXiv:1706.03762 [cs].

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (arXiv:1810.04805), May 2019.

[17] Zishan Guo, Renren Jin, Chuang Liu, Yufei Huang, Dan Shi, Supryadi, Linhao Yu, Yan Liu, Jiaxuan Li, Bojian Xiong, and Deyi Xiong. Evaluating Large Language Models: A Comprehensive Survey, October 2023. URL http://arxiv.org/abs/2310.19736. arXiv:2310.19736 [cs].

[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, June 2021. URL http://arxiv.org/abs/2010.11929. arXiv:2010.11929 [cs].

[19] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers, October 2021. URL http://arxiv.org/abs/2105.15203. arXiv:2105.15203 [cs].

[20] Zhan Tong, Yibing Song, Jue Wang, and Limin Wang. VideoMAE: Masked Autoencoders are Data-Efficient Learners for Self-Supervised Video Pre-Training, October 2022. URL http://arxiv.org/abs/2203.12602. arXiv:2203.12602 [cs].

[21] Philip B. Bedient, Wayne C. Huber, and Baxter E. Vieux. *Hydrology and Floodplain Analysis*. Pearson, third edition, 2007. ISBN 0-13-174589-1.

[22] Catchment basins. http://basins.ghkates.com/catchment-basins/, Jun 2003.

[23] Z. L. Flamig, H. Vergara, and J. J. Gourley. The ensemble framework for flash flood forecasting (ef5) v1.2: description and case study. *Geoscientific Model Development*, 13(10):4943–4958, 2020. doi: 10.5194/gmd-13-4943-2020. URL https://gmd.copernicus.org/articles/13/4943/2020/.

[24] Audrey Douinot, Hélène Roux, Pierre-André Garambois, Kévin Larnier, David Labat, and Denis Dartus. Accounting for rainfall systematic spatial variability

in flash flood forecasting. *Journal of Hydrology*, 541:359–370, October 2016. doi: 10.1016/j.jhydrol.2015.08.024.

[25] Iowa Environmental Mesonet. IEM :: Archived Local Storm Reports. https://mesonet.agron.iastate.edu/request/gis/lsrs.phtml.

[26] National Weather Service. Storm Events Database | National Centers for Environmental Information, . URL https://www.ncdc.noaa.gov/stormevents/.

[27] NOAA US Department of Commerce. Storm Report Records. URL https://www.weather.gov/unr/storm_reports#:~:text=%22Storm%20Data%22%20is%20a%20monthly,and%20other%20unusual%20weather%20phenomena. Publisher: NOAA's National Weather Service.

[28] NOAA Radar Operations Center. NEXRAD Radar Operations Center, WSR-88D. URL https://www.roc.noaa.gov/WSR88D/WindFarm/DoYourOwn.aspx?wid=dev#:~:text=NEXRAD%20Radar%20Operations%20Center%2C%20WSR%2D88D&text=The%20WSR%2D88D%20radar%20performs,0.5%C2%BA%20and%2019.5%C2%BA.

[29] NOAA National Severe Storms Laboratory. NSSL Projects: mPING. URL https://mping.nssl.noaa.gov/.

[30] NOAA US Department of Commerce. Impact Based Warnings, 2019. URL https://www.weather.gov/impacts/. Publisher: NOAA's National Weather Service.

[31] National Weather Service. Impact-Based Flash Flood Warnings: Factsheet, . URL https://www.weather.gov/media/wrn/FFW-IBW-factsheet.pdf.

[32] Amanda J. Schroeder, Jonathan J. Gourley, Jill Hardy, Jen J. Henderson, Pradipta Parhi, Vahid Rahmani, Kimberly A. Reed, Russ S. Schumacher, Brianne K. Smith, and Matthew J. Taraldsen. The development of a flash flood severity index. *Journal of Hydrology*, 541:523–532, October 2016. ISSN 0022-1694. doi: 10.1016/j.jhydrol.2016.04.005. URL https://www.sciencedirect.com/science/article/pii/S002216941630186X.

[33] Anne-Wil Harzing -. Publish or Perish. https://harzing.com/resources/publish-or-perish, 2007.

[34] Muhammad Imran, Prasenjit Mitra, and Carlos Castillo. Twitter as a Lifeline: Human-annotated Twitter Corpora for NLP of Crisis-related Messages. *arXiv:1605.05894 [cs]*, May 2016. Comment: Accepted at the 10th Language Resources and Evaluation Conference (LREC), 6 pages.

[35] Shanshan Zhang and Slobodan Vucetic. Semi-supervised Discovery of Informative Tweets During the Emerging Disasters. *arXiv:1610.03750 [cs]*, October 2016.

[36] Giorgio Gnecco, Rita Morisi, Giorgio Roth, Marcello Sanguineti, and Angela Celeste Taramasso. Supervised and semi-supervised classifiers for the detection of flood-prone areas. *Soft Computing*, 21(13):3673–3685, July 2017. ISSN 1432-7643, 1433-7479. doi: 10.1007/s00500-015-1983-z.

[37] Evan Racah, Christopher Beckham, Tegan Maharaj, Prabhat, and Christopher Pal. Semi-supervised detection of extreme weather events in large climate datasets. In *WITHDRAWN? - 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, April 2017.

[38] Evan Racah, Christopher Beckham, Tegan Maharaj, Samira Ebrahimi Kahou, Prabhat, and Christopher Pal. ExtremeWeather: A large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events. *arXiv:1612.02095 [cs, stat]*, November 2017.

[39] Nataliya Tkachenko, Stephen Jarvis, and Rob Procter. Predicting floods with Flickr tags. *PLOS ONE*, 12(2):e0172870, February 2017. ISSN 1932-6203. doi: 10.1371/journal.pone.0172870.

[40] Ruo-Qian Wang, Huina Mao, Yuan Wang, Chris Rae, and Wesley Shaw. Hyper-resolution monitoring of urban flooding with social media and crowdsourcing data. *Computers & Geosciences*, 111:139–147, February 2017. ISSN 00983004. doi: 10.1016/j.cageo.2017.11.008.

[41] Firoj Alam, Shafiq Joty, and Muhammad Imran. Graph Based Semi-supervised Learning with Convolution Neural Networks to Classify Crisis Related Tweets. *arXiv:1805.06289 [cs]*, May 2018. Comment: 5 pages. arXiv admin note: substantial text overlap with arXiv:1805.05151.

[42] J.L.P. Barker and C.J.A. Macleod. Development of a national-scale real-time Twitter data mining pipeline for social geodata on the potential impacts of flooding on

communities. *Environmental Modelling & Software*, 115:213–227, May 2018. ISSN 13648152. doi: 10.1016/j.envsoft.2018.11.013.

[43] Jiongqian Liang, Peter Jacobs, and Srinivasan Parthasarathy. Human-Guided Flood Mapping: From Experts to the Crowd. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*, pages 291–298, Lyon, France, 2018. ACM Press. ISBN 978-1-4503-5640-4. doi: 10.1145/3184558.3186339.

[44] Amir Mosavi, Pinar Ozturk, and Kwok-wing Chau. Flood Prediction Using Machine Learning Models: Literature Review. *Water*, 10(11):1536, October 2018. ISSN 2073-4441. doi: 10.3390/w10111536.

[45] Y C A Padmanabha Reddy, P Viswanath, and B Eswara Reddy. Semi-supervised learning: A brief review. *International Journal of Engineering & Technology*, 7(1.8): 81, February 2018. ISSN 2227-524X. doi: 10.14419/ijet.v7i1.8.9977.

[46] Jun Qian, Zhendong Niu, and Chongyang Shi. Sentiment Analysis Model on Weather Related Tweets with Deep Neural Network. In *Proceedings of the 2018 10th International Conference on Machine Learning and Computing*, pages 31–35, Macau China, February 2018. ACM. ISBN 978-1-4503-6353-2. doi: 10.1145/3195106.3195111.

[47] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent Trends in Deep Learning Based Natural Language Processing [Review Article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, August 2018. ISSN 1556-603X, 1556-6048. doi: 10.1109/MCI.2018.2840738.

[48] Gang Zhao, Bo Pang, Zongxue Xu, Dingzhi Peng, and Liyang Xu. Assessment of urban flood susceptibility using semi-supervised machine learning model. *Science of The Total Environment*, 659:940–949, April 2018. ISSN 00489697. doi: 10.1016/j.scitotenv.2018.12.217.

[49] Viet-Hung Luu, Minh-Son Dao, Thi Nhat-Thanh Nguyen, Stuart Perry, and Koji Zettsu. Semi-supervised Convolutional Neural Networks for Flood Mapping using Multi-modal Remote Sensing Data. In *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, pages 342–347, Hanoi, Vietnam, December 2019. IEEE. ISBN 978-1-72815-163-2. doi: 10.1109/NICS48868.2019.9023891.

[50] Amy McGovern, Ryan Lagerquist, David John Gagne, G. Eli Jergensen, Kimberly L. Elmore, Cameron R. Homeyer, and Travis Smith. Making the Black Box

More Transparent: Understanding the Physical Implications of Machine Learning. *Bulletin of the American Meteorological Society*, 100(11):2175–2199, November 2019. ISSN 0003-0007, 1520-0477. doi: 10.1175/BAMS-D-18-0195.1.

[51] Abhishek V. Potnis, Rajat C. Shinde, Surya S. Durbha, and Kuldeep R. Kurte. Multi-Class Segmentation of Urban Floods from Multispectral Imagery Using Deep Learning. In *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, pages 9741–9744, Yokohama, Japan, July 2019. IEEE. ISBN 978-1-5386-9154-0. doi: 10.1109/IGARSS.2019.8900250.

[52] Bob E Saint Fleur, Guillaume Artigue, Anne Johannet, and Severin Pistre. Knowledge Extraction (KnoX) in Deep Learning: Application to the Gardon de Mialet Flash Floods Modelling. *Proceedings ITISE-2019*, page 13, 2019.

[53] Jonathan A. Weyn, Dale R. Durran, and Rich Caruana. Can Machines Learn to Predict Weather? using Deep Learning to Predict Gridded 500-hPa Geopotential Height From Historical Weather Data. *Journal of Advances in Modeling Earth Systems*, 11(8):2680–2693, August 2019. ISSN 1942-2466, 1942-2466. doi: 10.1029/2019MS001705.

[54] Vikas Yadav and Steven Bethard. A Survey on Recent Advances in Named Entity Recognition from Deep Learning models. *arXiv:1910.11470 [cs]*, October 2019. Comment: Published at COLING 2018.

[55] Danilo Croce, Giuseppe Castellucci, and Roberto Basili. GAN-BERT: Generative Adversarial Learning for Robust Text Classification with a Bunch of Labeled Examples. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2114–2119, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.191.

[56] Amrul Faruq, Hudan Pandu Arsa, Shamsul Faisal Mohd Hussein, Che Munira Che Razali, Aminaton Marto, and Shahrum Shah Abdullah. Deep Learning-Based Forecast and Warning of Floods in Klang River, Malaysia. *Ingénierie des systèmes d information*, 25(3):365–370, June 2020. ISSN 16331311, 21167125. doi: 10.18280/isi.250311.

[57] Kazi Aminul Islam, Mohammad Shahab Uddin, Chiman Kwan, and Jiang Li. Flood Detection Using Multi-Modal and Multi-Temporal Images: A Comparative Study. *Remote Sensing*, 12(15):2455, July 2020. ISSN 2072-4292. doi: 10.3390/rs12152455.

[58] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, October 2020. ISSN 1674-7321, 1869-1900. doi: 10.1007/s11431-020-1647-3.

[59] Suresh Sankaranarayanan, Malavika Prabhakar, Sreesta Satish, Prerna Jain, Anjali Ramprasad, and Aiswarya Krishnan. Flood prediction based on weather parameters using deep learning. *Journal of Water and Climate Change*, 11(4):1766–1783, December 2020. ISSN 2040-2244, 2408-9354. doi: 10.2166/wcc.2019.321.

[60] Zening Wu, Yihong Zhou, Huiliang Wang, and Zihao Jiang. Depth prediction of urban flood under different rainfall return periods based on deep learning and data warehouse. *Science of The Total Environment*, 716:137077, May 2020. ISSN 00489697. doi: 10.1016/j.scitotenv.2020.137077.

[61] Jingyu Yao, Shengwu Qin, Shuangshuang Qiao, Wenchao Che, Yang Chen, Gang Su, and Qiang Miao. Assessment of Landslide Susceptibility Combining Deep Learning with Semi-Supervised Learning in Jiaohe County, Jilin Province, China. *Applied Sciences*, 10(16):5640, August 2020. ISSN 2076-3417. doi: 10.3390/app10165640.

[62] Leila Hashemi-Beni and Asmamaw A. Gebrehiwot. Flood Extent Mapping: An Integrated Method Using Deep Learning and Region Growing Using UAV Optical Data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 14:2127–2135, 2021. ISSN 1939-1404, 2151-1535. doi: 10.1109/JSTARS.2021.3051873.

[63] Kelvin Lai, Jeremy R. Porter, Mike Amodeo, David Miller, Michael Marston, and Saman Armal. A Natural Language Processing Approach to Understanding Context in the Extraction and GeoCoding of Historical Floods, Storms, and Adaptation Measures. *Information Processing & Management*, 59(1):102735, October 2021. ISSN 03064573. doi: 10.1016/j.ipm.2021.102735.

[64] Chetana Nair and Bhakti Palkar. Kerala Floods: Twitter Analysis Using Deep Learning Techniques. In Harish Sharma, Mukesh Saraswat, Anupam Yadav, Joong Hoon Kim, and Jagdish Chand Bansal, editors, *Congress on Intelligent Systems*, volume 1334, pages 317–325. Springer Singapore, Singapore, 2021. ISBN 978-981-336-980-1 978-981-336-981-8. doi: 10.1007/978-981-33-6981-8_26.

[65] Sayak Paul and Siddha Ganju. Flood Segmentation on Sentinel-1 SAR Imagery with Semi-Supervised Learning. *arXiv:2107.08369 [cs]*, October 2021. Comment: Equal authorship. Accepted to the Tackling Climate Change with Machine Learning workshop at NeurIPS 2021. Code and models are available at https://git.io/JW3P8.

[66] Akhil Sanjay Potdar, Pierre-Emmanuel Kirstetter, Devon Woods, and Manabendra Saharia. Towards Predicting Flood Event Peak Discharge in Ungauged Basins by Learning Universal Hydrological Behaviors with Machine Learning. *Journal of Hydrometeorology*, August 2021. ISSN 1525-755X, 1525-7541. doi: 10.1175/JHM-D-20-0302.1.

[67] Sella Nevo, Efrat Morin, Adi Gerzi Rosenthal, Asher Metzger, Chen Barshai, Dana Weitzner, Dafi Voloshin, Frederik Kratzert, Gal Elidan, Gideon Dror, Gregory Begelman, Grey Nearing, Guy Shalev, Hila Noga, Ira Shavitt, Liora Yuklea, Moriah Royz, Niv Giladi, Nofar Peled Levi, Ofir Reich, Oren Gilon, Ronnie Maor, Shahar Timnat, Tal Shechter, Vladimir Anisimov, Yotam Gigi, Yuval Levin, Zach Moshe, Zvika Ben-Haim, Avinatan Hassidim, and Yossi Matias. Flood forecasting with machine learning models in an operational framework. Preprint, Catchment hydrology/Modelling approaches, November 2021.

[68] Kartika Purwandari, Join WC Sigalingging, Tjeng Wawan Cenggoro, and Bens Pardamean. Multi-class weather forecasting from twitter using machine learning aprroaches. *Procedia Computer Science*, 179:47–54, 2021.

[69] Kartika Purwandari, Reza Rahutomo, Join W. C. Sigalingging, Mahisa Ajy Kusuma, Aji Prasetyo, and Bens Pardamean. Twitter-Based Text Classification Using SVM for Weather Information System. In *2021 International Conference on Information Management and Technology (ICIMTech)*, pages 27–32, Jakarta, Indonesia, August 2021. IEEE. ISBN 978-1-66544-937-3. doi: 10.1109/ICIMTech53080.2021.9534945.

[70] Jaya Surya Sattaru, C. M. Bhatt, and Sameer Saran. Utilizing Geo-Social Media as a Proxy Data for Enhanced Flood Monitoring. *Journal of the Indian Society of Remote Sensing*, 49(9):2173–2186, September 2021. ISSN 0255-660X, 0974-3006. doi: 10.1007/s12524-021-01376-9.

[71] Ruoyu Zhang, Hyunglok Kim, Emily Lien, Diyu Zheng, Lawrence Band, and Venkataraman Lakshmi. Deep Learning Approach to Predict Peak Floods and

Evaluate Socioeconomic Vulnerability to Flood Events: A Case Study in Baltimore, MD, U.S.A. In *2021 Systems and Information Engineering Design Symposium (SIEDS)*, pages 1–6, Charlottesville, VA, USA, April 2021. IEEE. ISBN 978-1-66541-250-6. doi: 10.1109/SIEDS52267.2021.9483782.

[72] European Centre for Medium-Range Weather Forecasts. ECMWF | Geopotential 500 hPa. https://www.ecmwf.int/en/forecasts/charts/catalogue/medium-z500-t850?facets=undefined&time=2021112812,0,2021112812&projection=classical_europe, 2021.

[73] Galateia Terti, Isabelle Ruin, Jonathan J. Gourley, Pierre Kirstetter, Zachary Flamig, Juliette Blanchet, Ami Arthur, and Sandrine Anquetin. Toward Probabilistic Prediction of Flash Flood Human Impacts. *Risk Analysis*, 39(1): 140–161, 2019. ISSN 1539-6924. doi: 10.1111/risa.12921. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/risa.12921`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/risa.12921.

[74] M. Diakakis, G. Deligiannakis, Z. Antoniadis, M. Melaki, N. K. Katsetsiadou, E. Andreadakis, N. I. Spyrou, and M. Gogou. Proposal of a flash flood impact severity scale for the classification and mapping of flash flood impacts. *Journal of Hydrology*, 590:125452, November 2020. ISSN 0022-1694. doi: 10.1016/j.jhydrol.2020.125452. URL `https://www.sciencedirect.com/science/article/pii/S0022169420309124`.

[75] Andrew Kruczkiewicz, Agathe Bucherie, Fernanda Ayala, Carolynne Hultquist, Humberto Vergara, Simon Mason, Juan Bazo, and Alex de Sherbinin. Development of a Flash Flood Confidence Index from Disaster Reports and Geophysical Susceptibility. *Remote Sensing*, 13(14):2764, January 2021. ISSN 2072-4292. doi: 10.3390/rs13142764. URL `https://www.mdpi.com/2072-4292/13/14/2764`. Number: 14 Publisher: Multidisciplinary Digital Publishing Institute.

[76] Agathe Bucherie, Fernanda Ayala, and Andrew Kruczkiewicz. Ecuador historical flood occurrences and impacts dataset with Flash Flood Confidence Index (2007-2020). 2021. doi: 10.5281/zenodo.4662886. URL `https://zenodo.org/records/4662886`.

[77] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[78] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[79] Tensorflow Hub. TensorFlow Hub, . URL `https://www.tensorflow.org/hub`.

[80] Tensorflow Hub. BERT- TensorFlow Hub, . URL `https://tfhub.dev/google/collections/bert/1`.

[81] Kaggle. Kaggle | Find Pre-trained Models. URL `https://www.kaggle.com/models`.

[82] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-\learning-library.pdf`.

[83] HuggingFace. Hugging Face - Documentation, . URL `https://huggingface.co/docs`.

[84] HuggingFace. SegFormer - Model Documentation, . URL `https://huggingface.co/docs/transformers/model_doc/segformer`.

[85] HuggingFace. VideoMAE - Model Documentation, . URL `https://huggingface.co/docs/transformers/model_doc/videomae`.

[86] HuggingFace. Vision Transformer (ViT) - Model Documentation, . URL https://huggingface.co/docs/transformers/model_doc/vit.

Appendix

FLASH Moment Extraction - MAXUnitQ Distributions

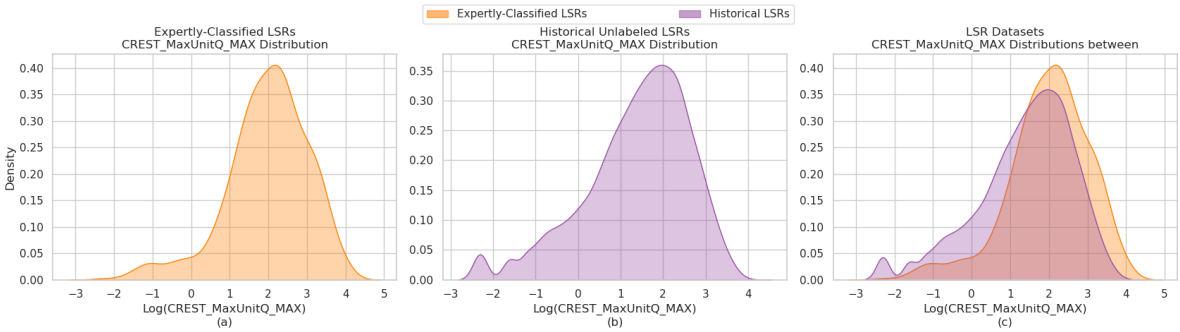


Figure A1: Distribution of maximum MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

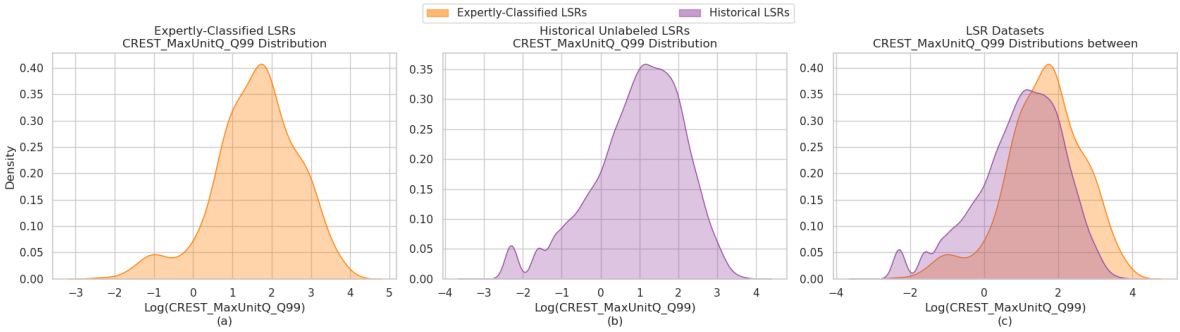


Figure A2: Distribution of Q99 MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

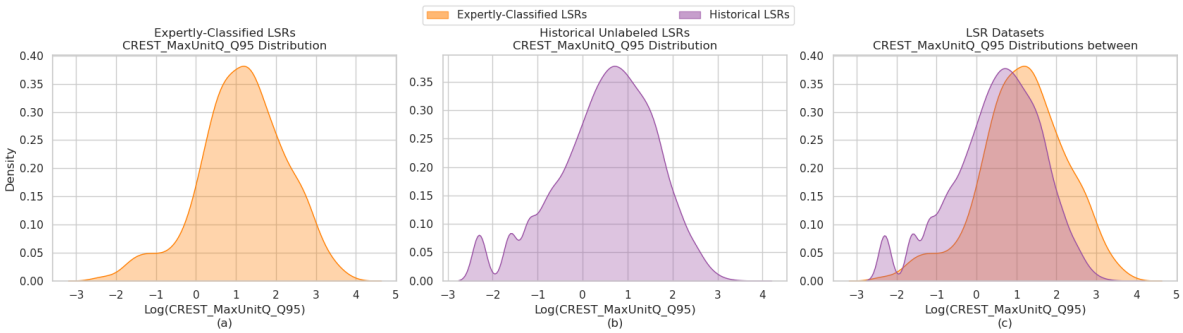


Figure A3: Distribution of Q95 MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

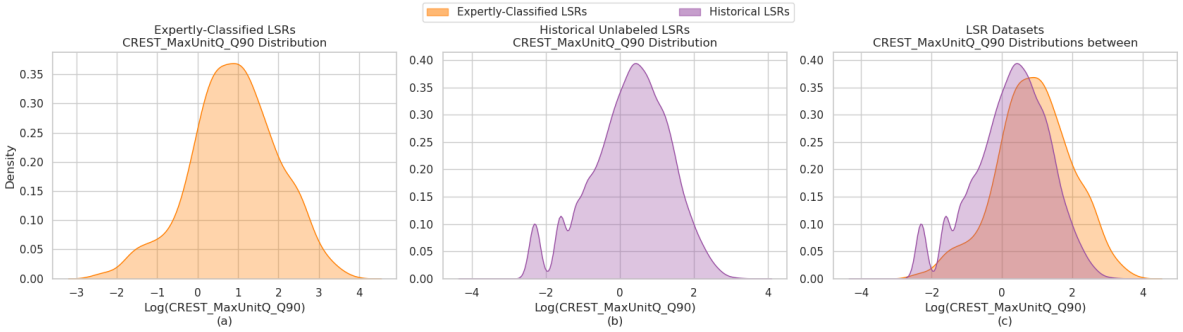


Figure A4: Distribution of Q90 MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

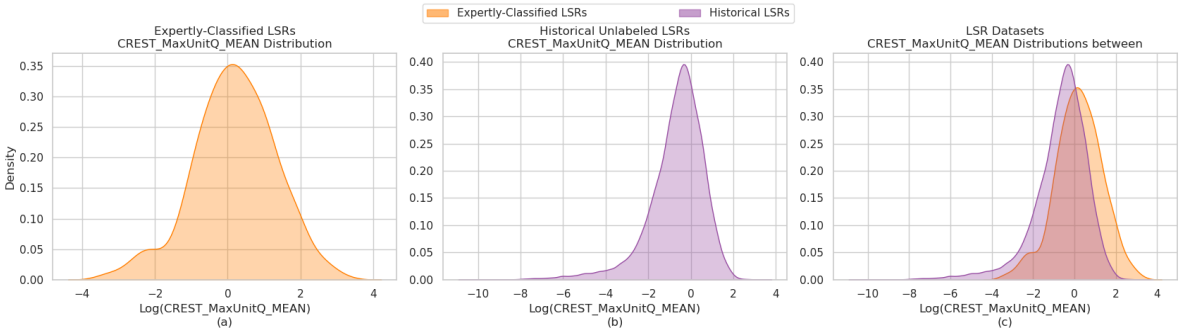


Figure A5: Distribution of mean MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

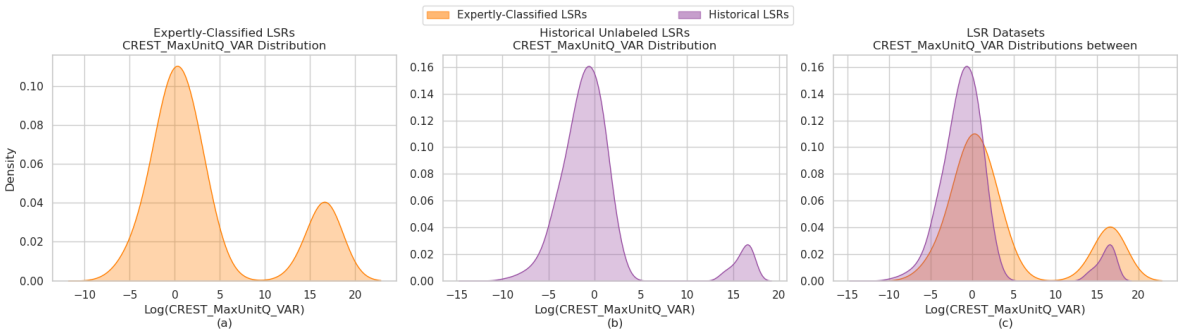


Figure A6: Distribution of MaxUnitQ variance values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

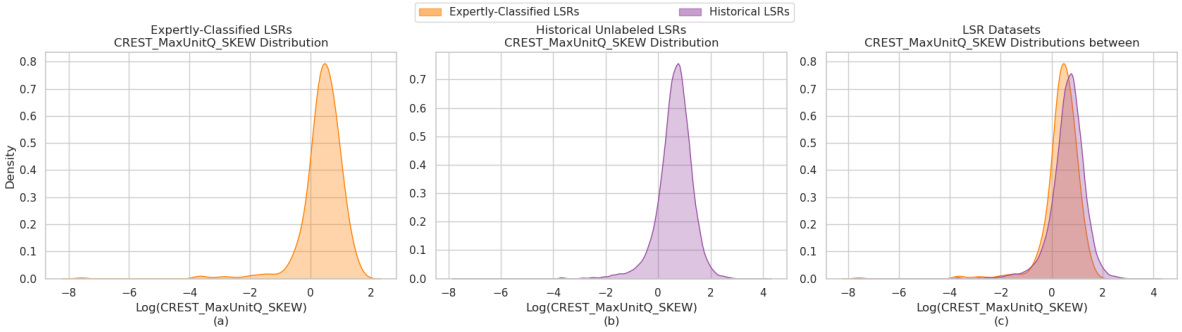


Figure A7: Distribution of MaxUnitQ skewness values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

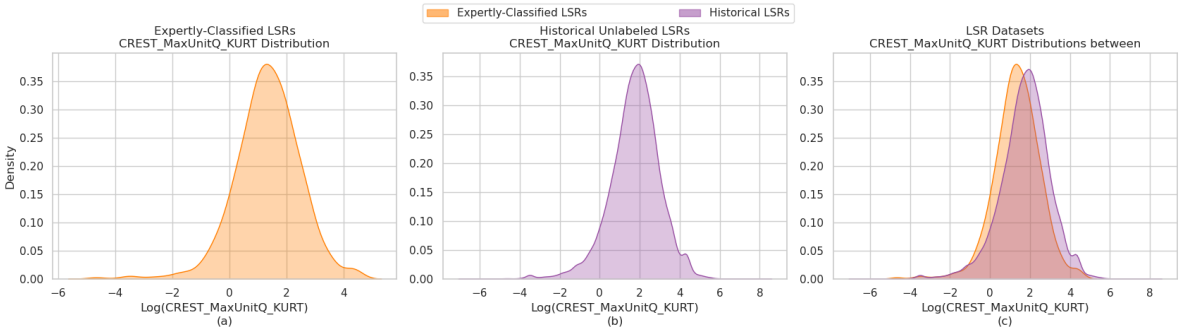


Figure A8: Distribution of MaxUnitQ kurtosis values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

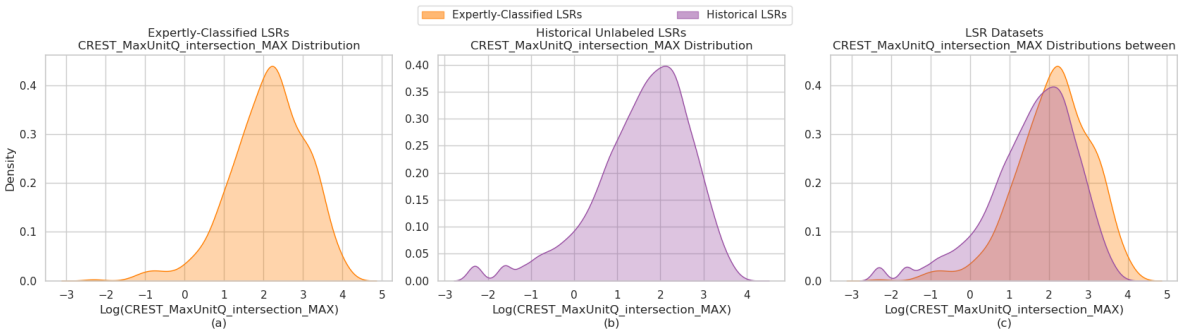


Figure A9: Distribution of maximum MaxUnitQ cropped by MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

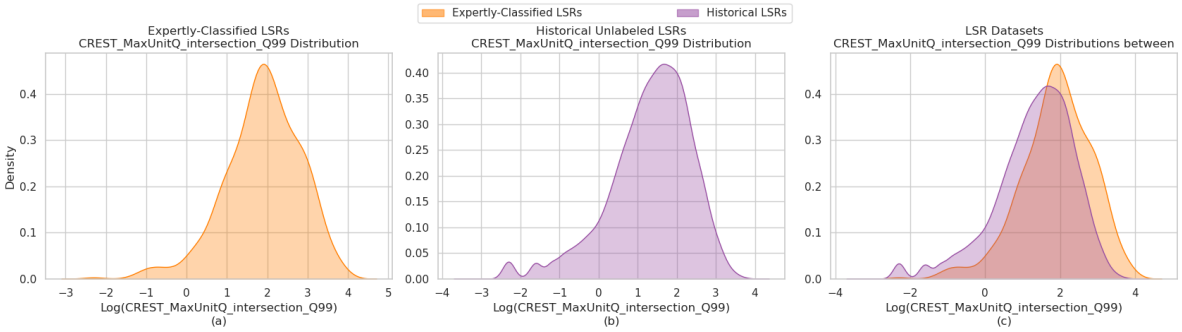


Figure A10: Distribution of Q99 MaxUnitQ cropped by MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

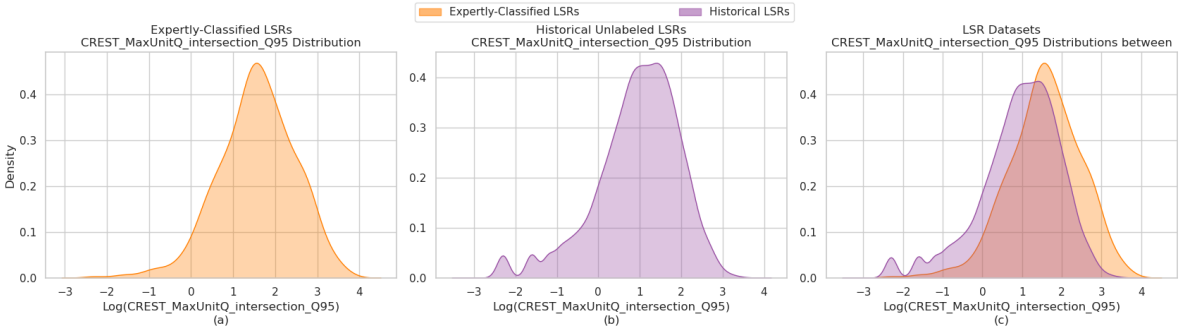


Figure A11: Distribution of Q95 MaxUnitQ cropped by MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

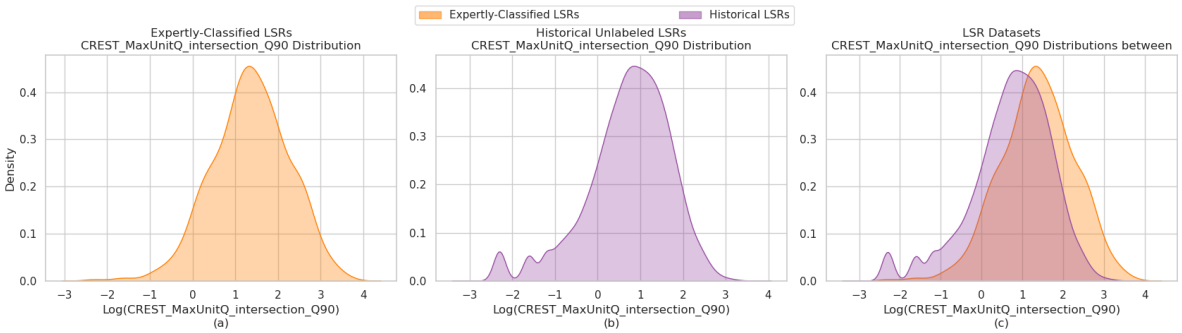


Figure A12: Distribution of Q90 MaxUnitQ cropped by MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

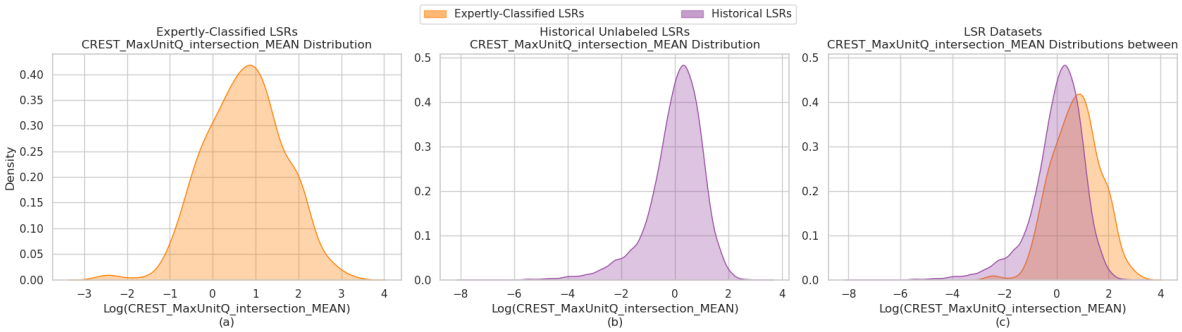


Figure A13: Distribution of mean MaxUnitQ cropped by MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

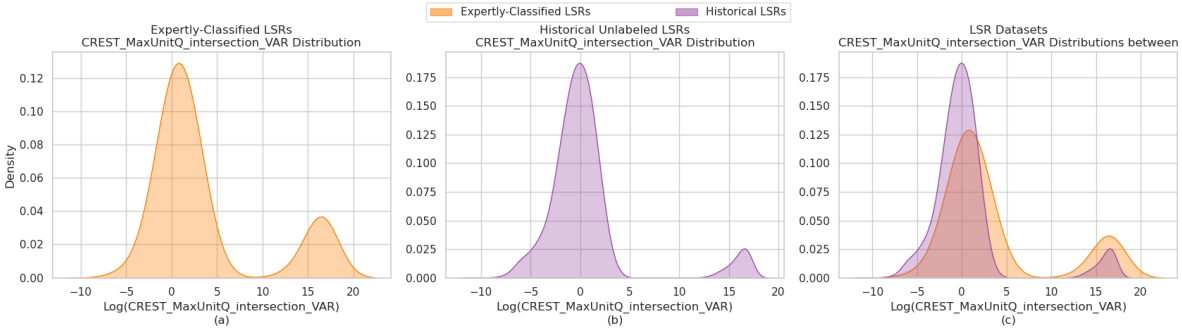


Figure A14: Distribution of MaxUnitQ cropped by MaxARI variance values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

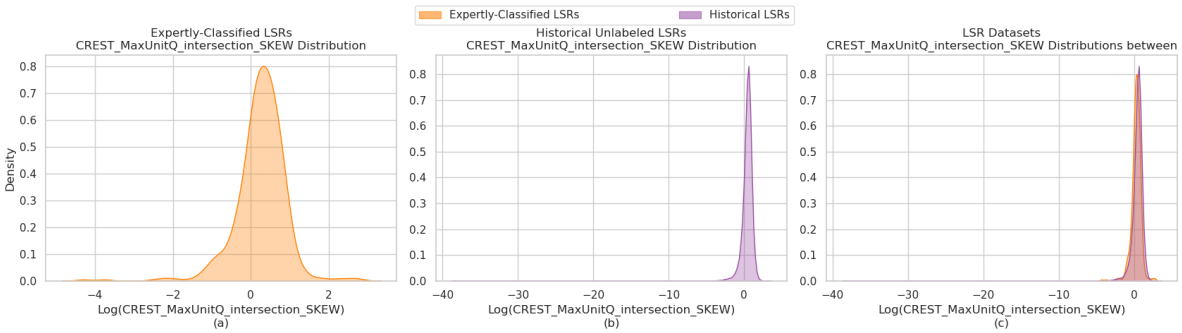


Figure A15: Distribution of MaxUnitQ cropped by MaxARI skewness values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

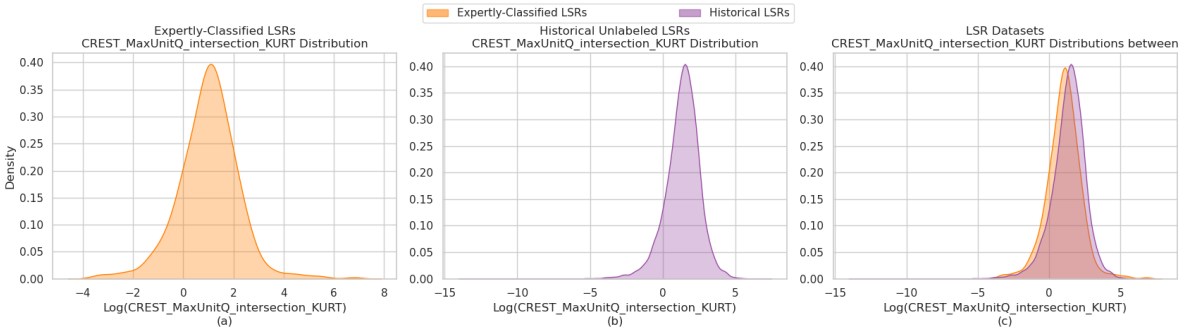


Figure A16: Distribution of MaxUnitQ cropped by MaxARI kurtosis values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

FLASH Moment Extraction - MaxARI Distributions

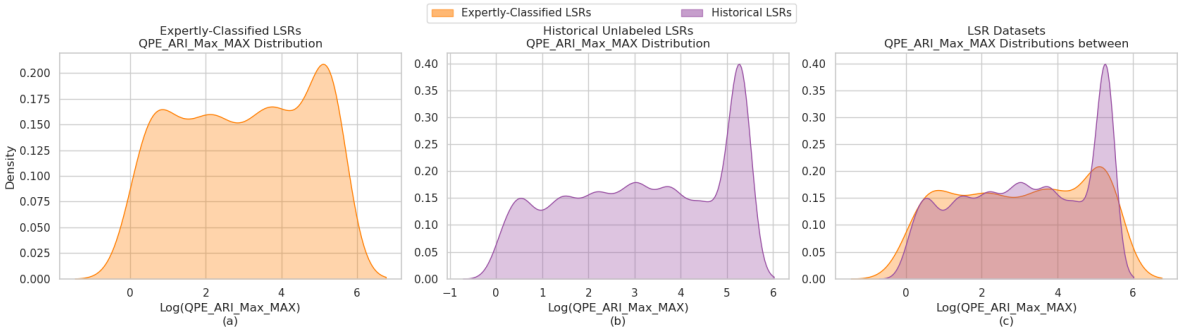


Figure A17: Distribution of maximum MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

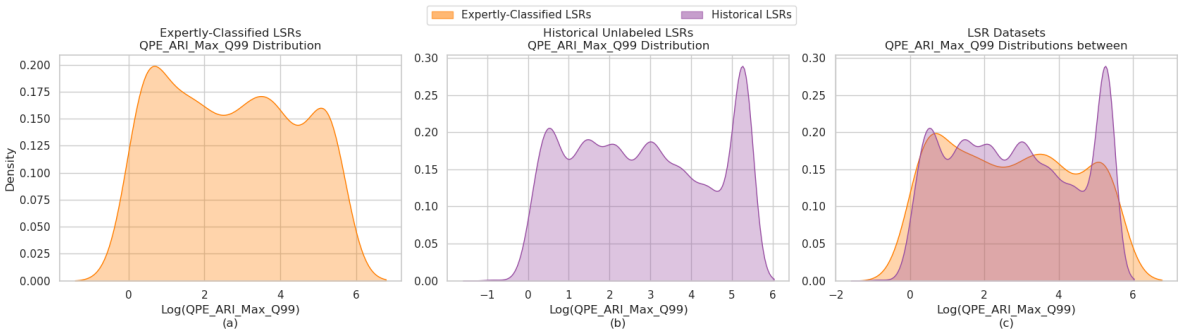


Figure A18: Distribution of Q99 MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

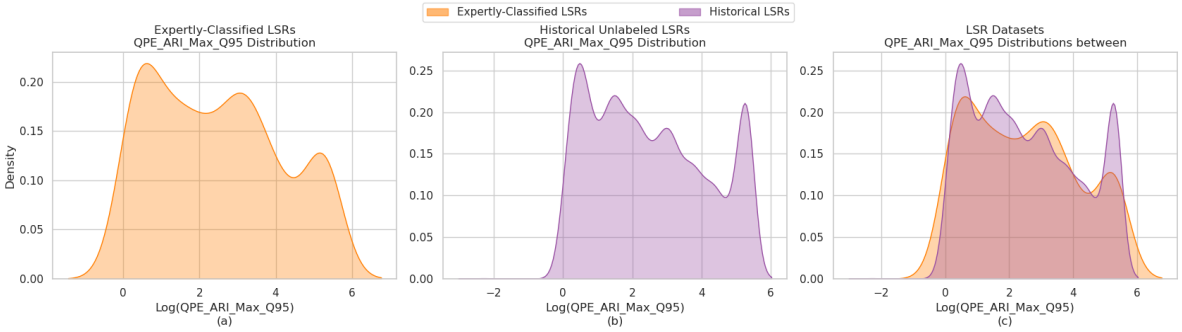


Figure A19: Distribution of Q95 MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

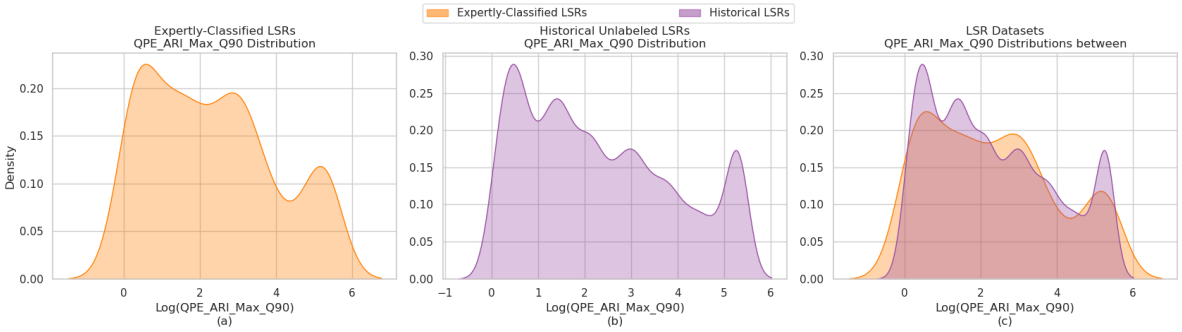


Figure A20: Distribution of Q90 MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

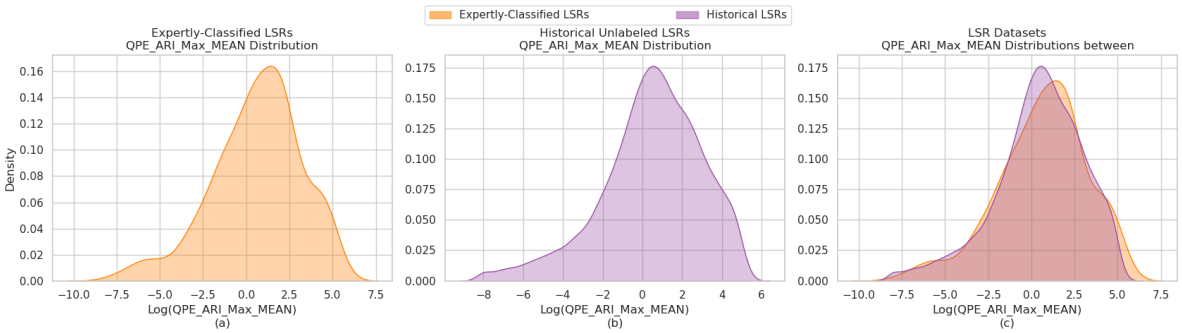


Figure A21: Distribution of mean MaxARI values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

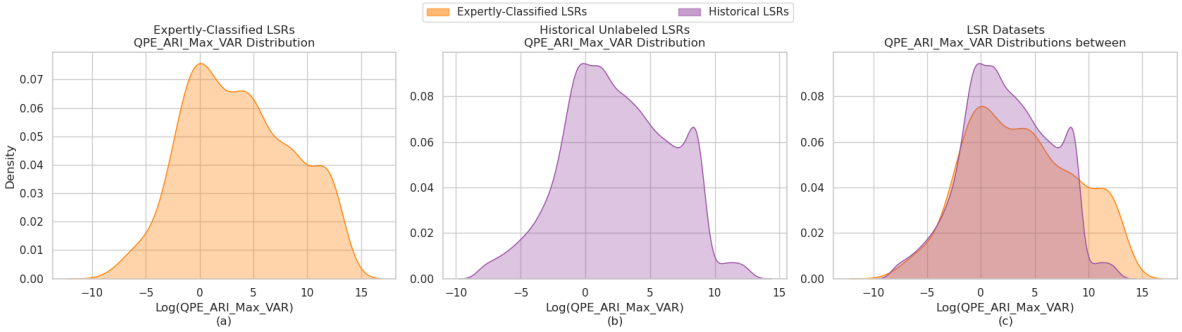


Figure A22: Distribution of MaxARI variance values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

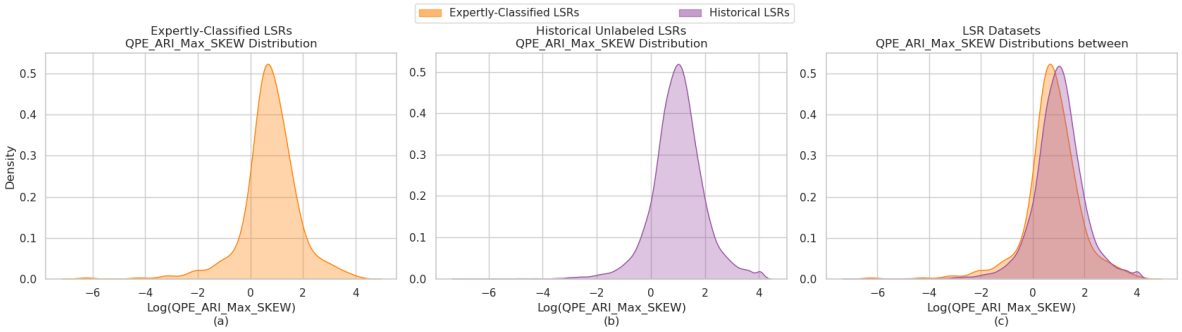


Figure A23: Distribution of MaxARI skewness values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

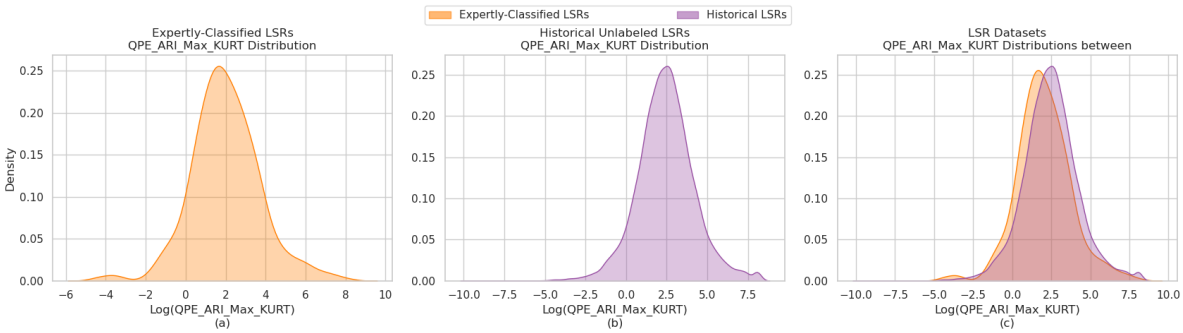


Figure A24: Distribution of MaxARI kurtosis values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

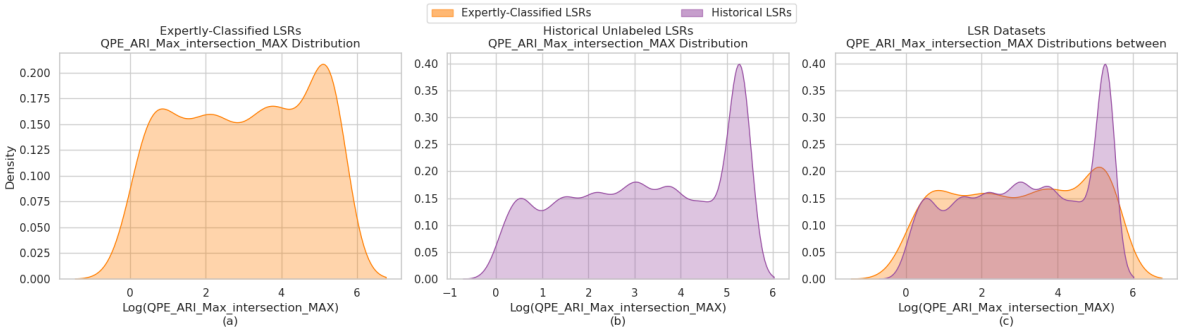


Figure A25: Distribution of maximum MaxARI cropped by MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

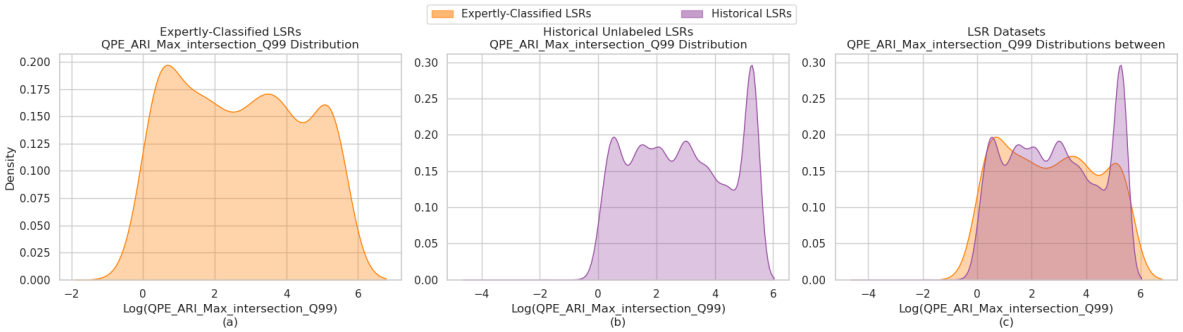


Figure A26: Distribution of Q99 MaxARI cropped by MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

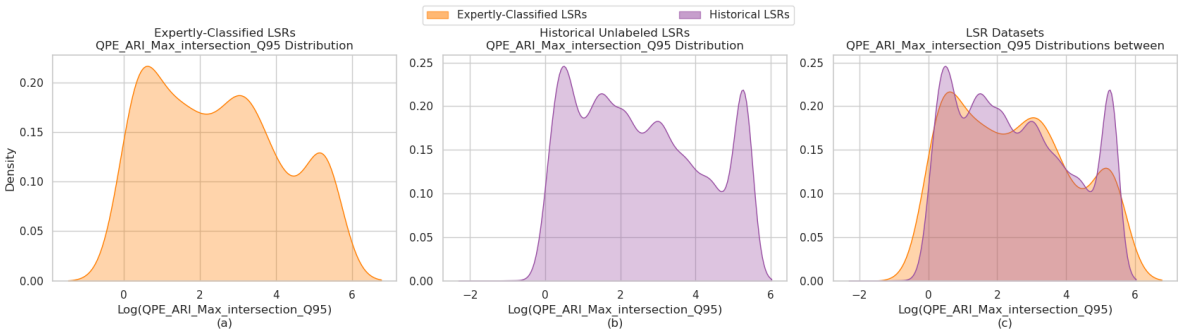


Figure A27: Distribution of Q95 MaxARI cropped by MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

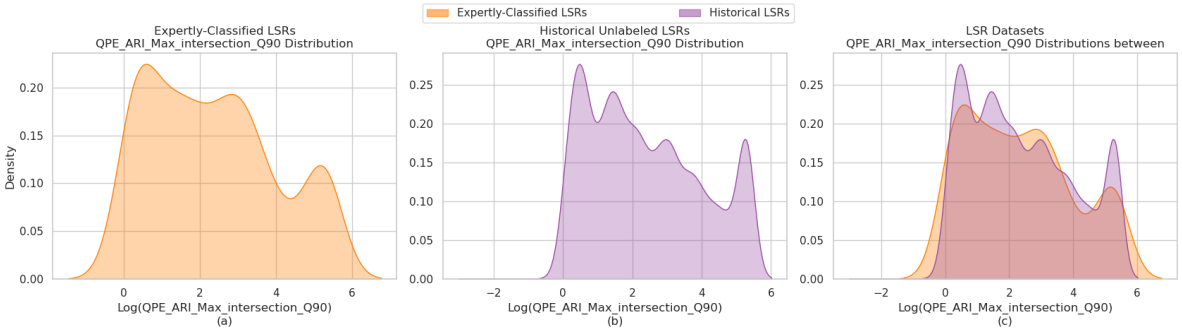


Figure A28: Distribution of Q90 MaxARI cropped by MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

* * *

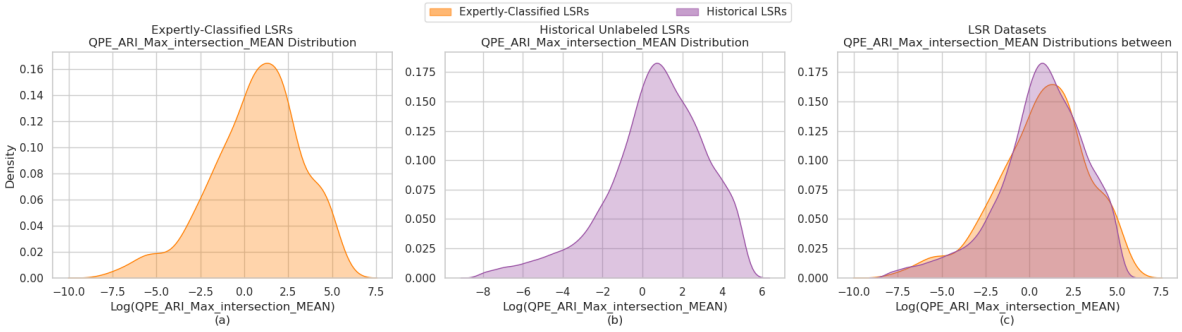


Figure A29: Distribution of mean MaxARI cropped by MaxUnitQ values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

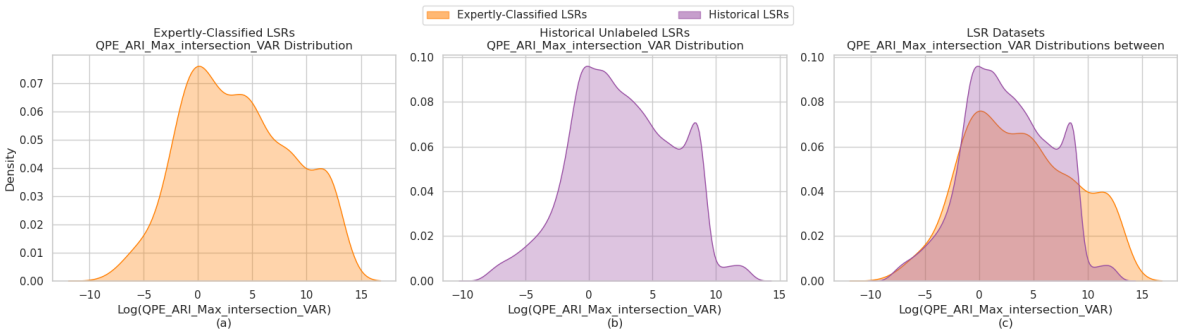


Figure A30: Distribution of MaxARI cropped by MaxUnitQ variance values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

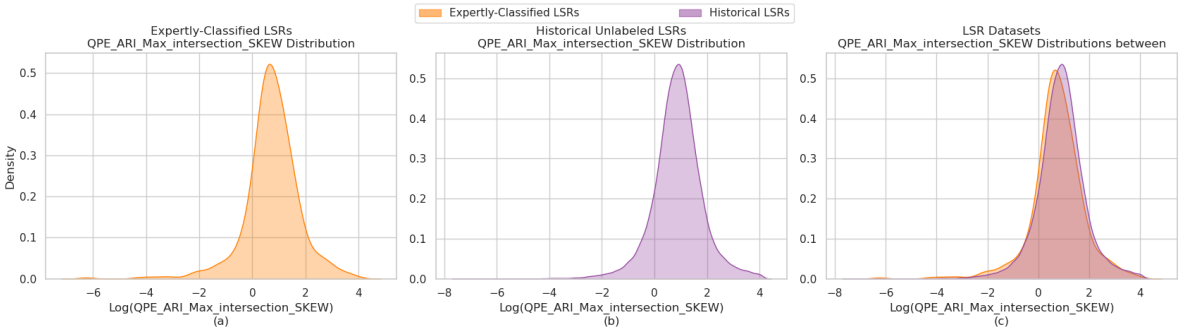


Figure A31: Distribution of MaxARI cropped by MaxUnitQ skewness values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

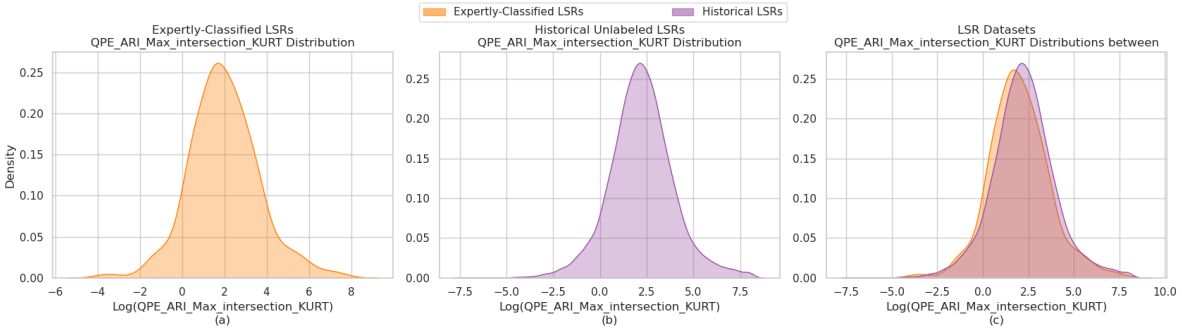


Figure A32: Distribution of MaxARI cropped by MaxUnitQ kurtosis values for (a) the expertly-classified dataset, (b) historical unlabeled dataset, and (c) both datasets.

FLASH Moment Extraction - IBW Class Distributions

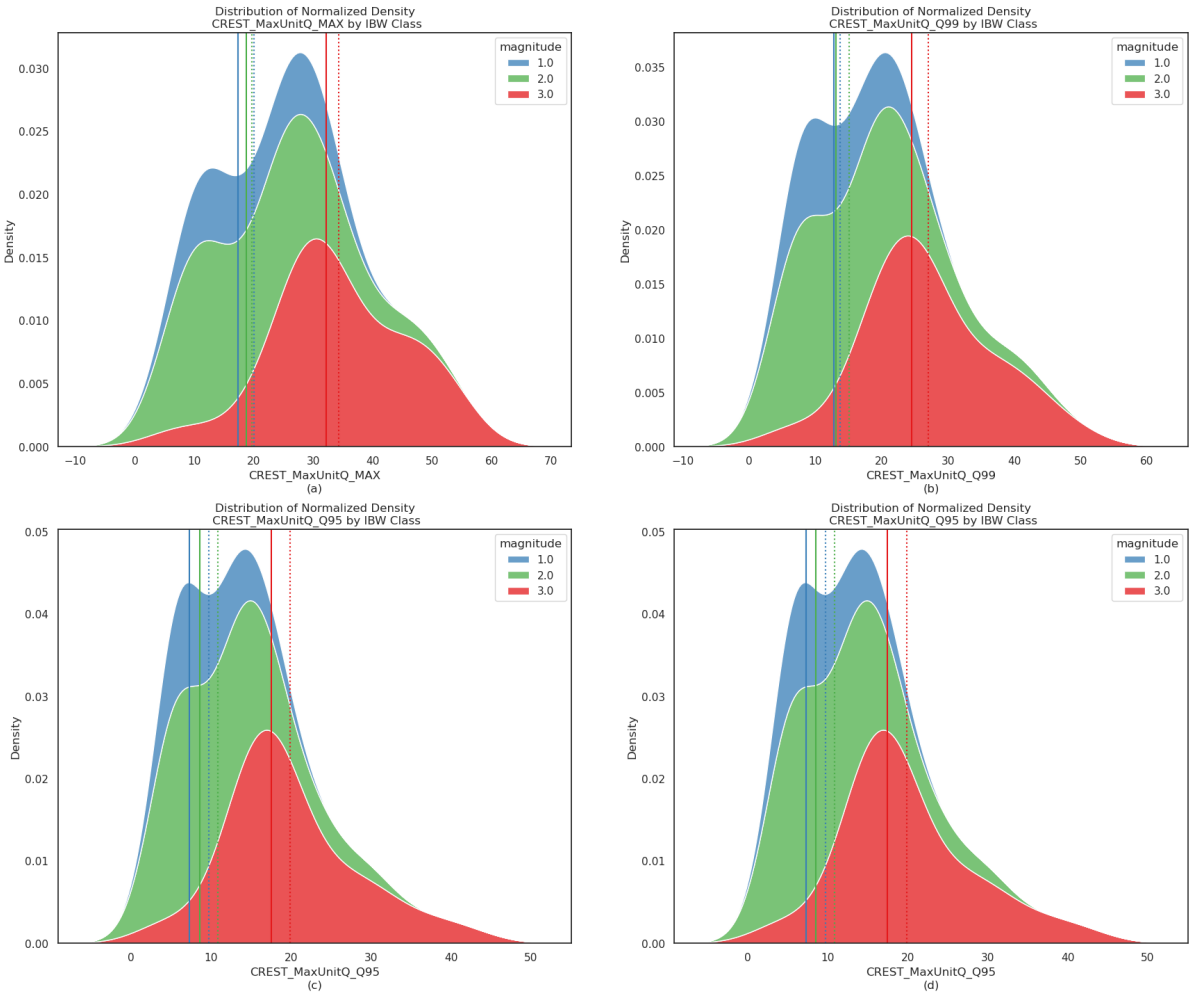


Figure A33: Distribution of MaxUnitQ maximum and quantile per IBW class. (a) Maximum, (b) Q99, (c) Q95, and (d) Q99. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

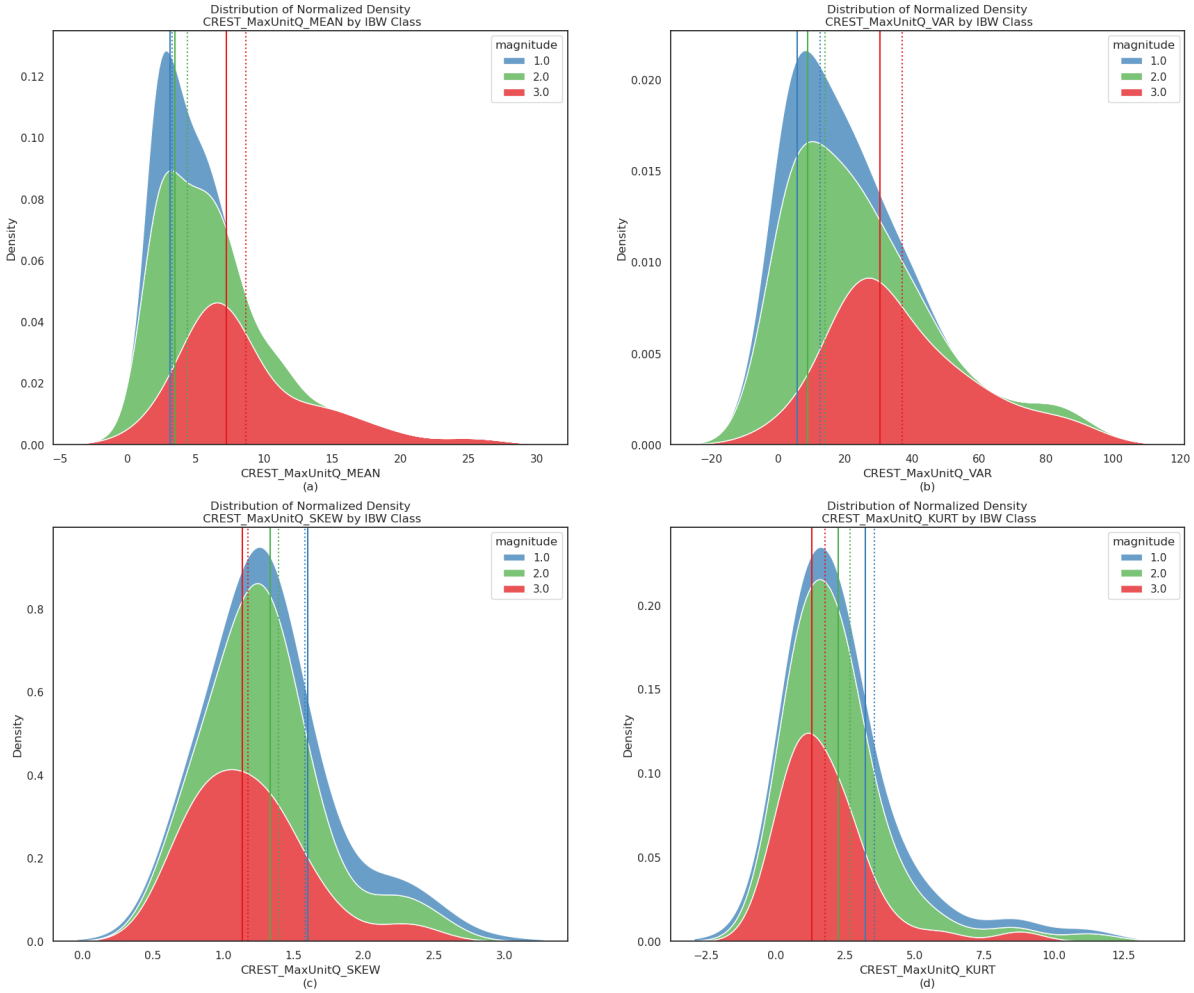


Figure A34: Distribution of MaxUnitQ moments per IBW class. (a) Mean, (b) Variance, (c) Skewness, and (d) Kurtosis. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

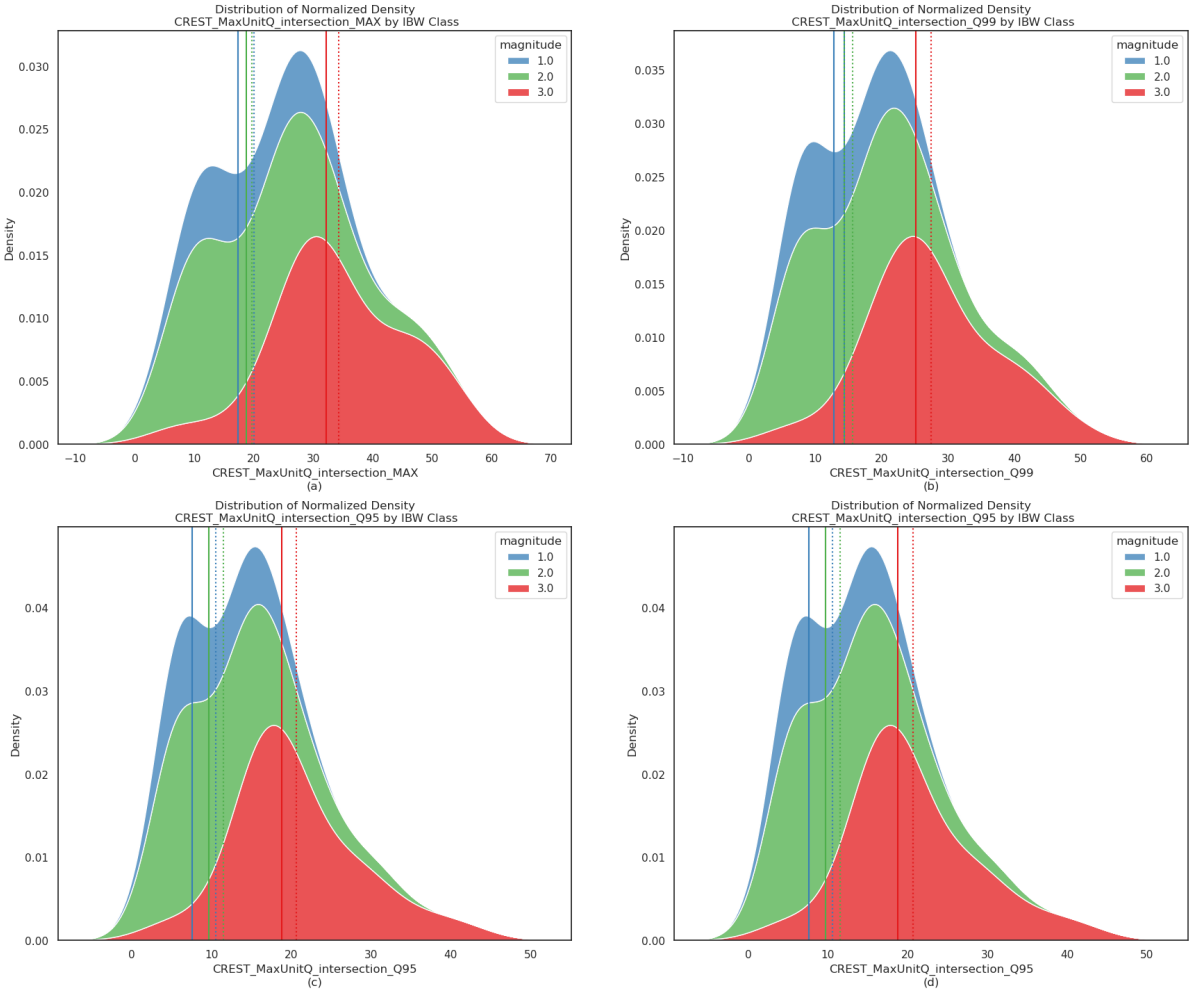


Figure A35: Distribution of MaxUnitQ∩MaxARI maximum and quantile per IBW class. (a) Maximum, (b) Q99, (c) Q95, and (d) Q99. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

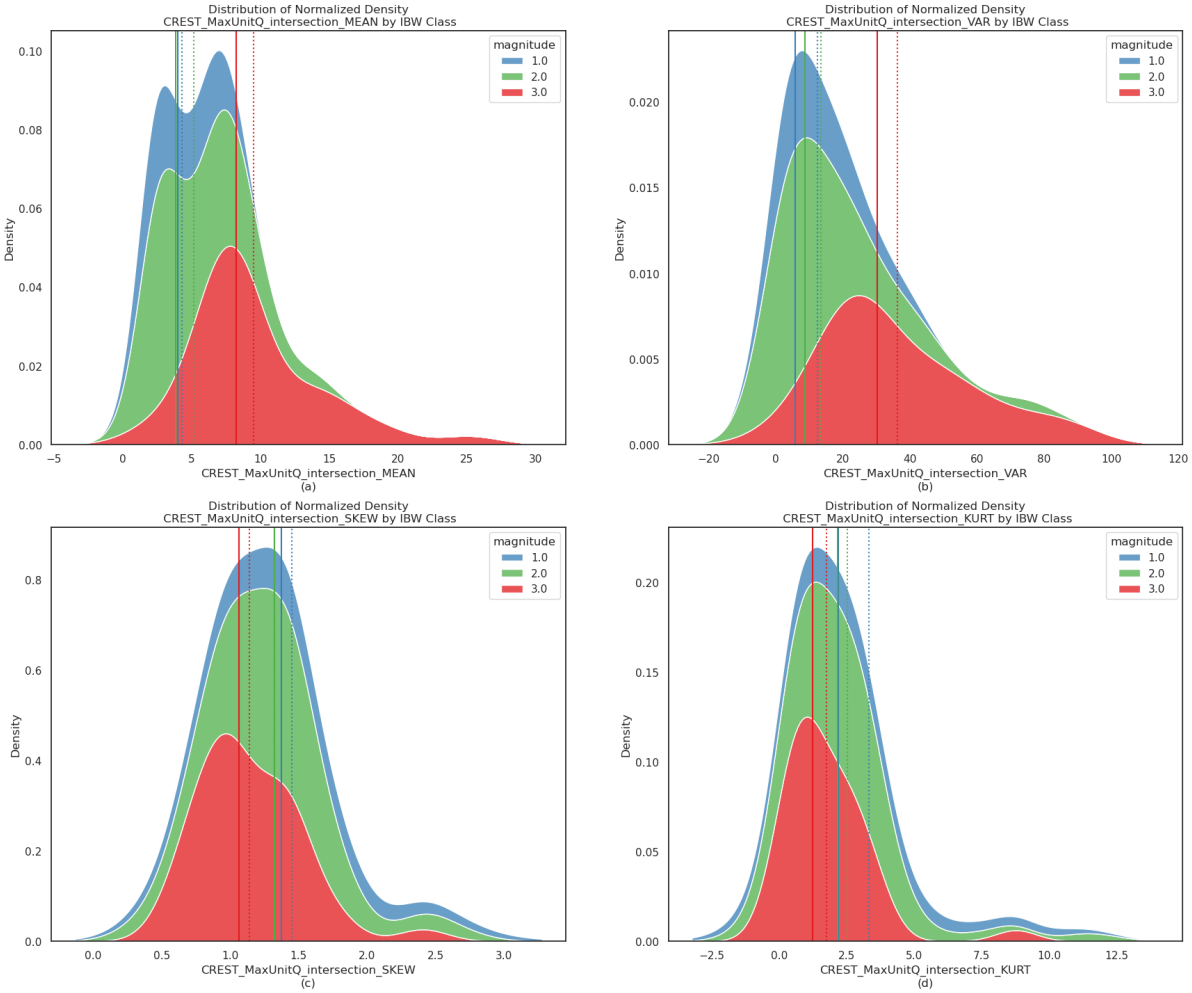


Figure A36: Distribution of MaxUnitQ∩MaxARI moments per IBW class. (a) Mean, (b) Variance, (c) Skewness, and (d) Kurtosis. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

* * *

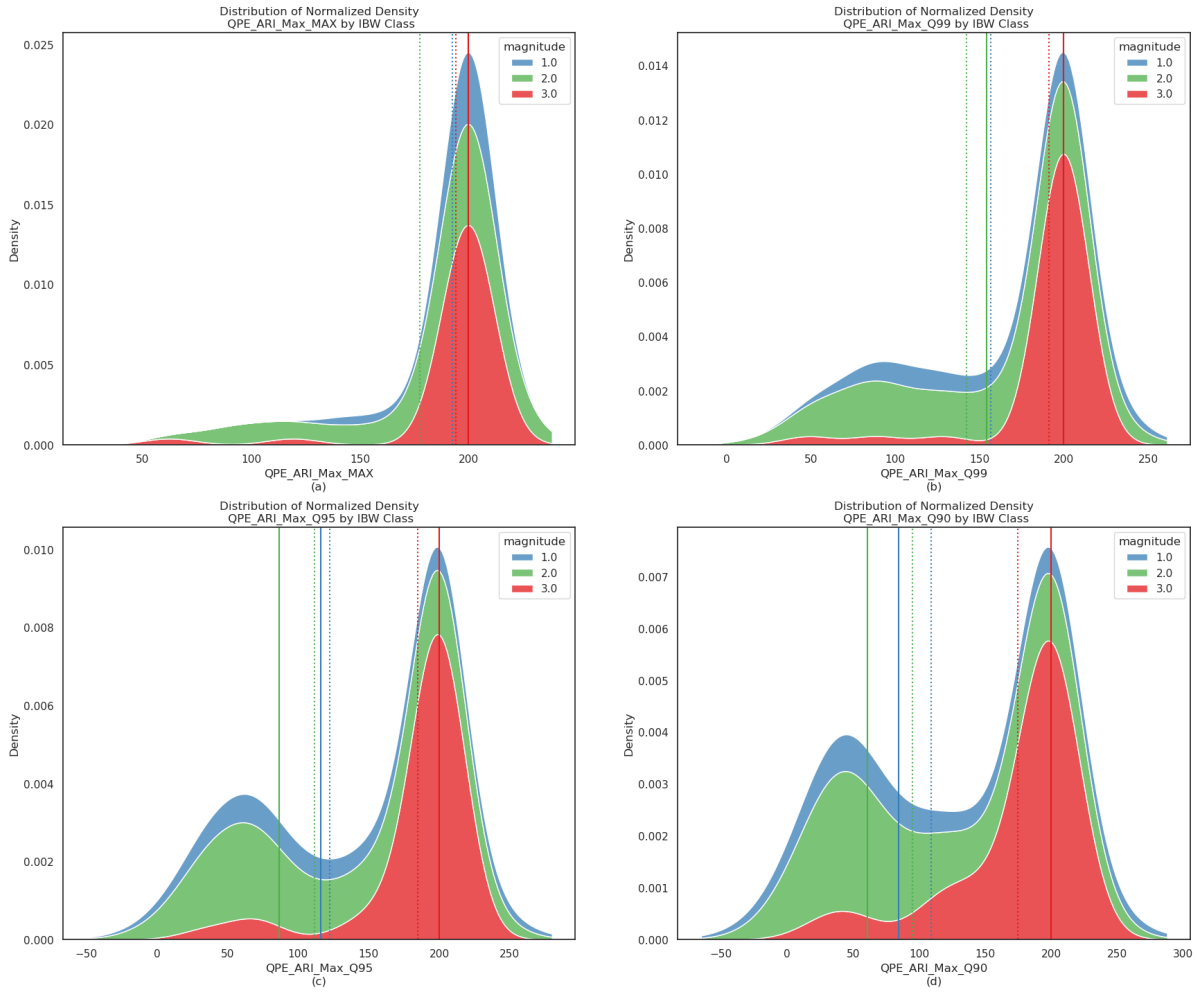


Figure A37: Distribution of MaxARI maximum and quantile per IBW class. (a) Maximum, (b) Q99, (c) Q95, and (d) Q99. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

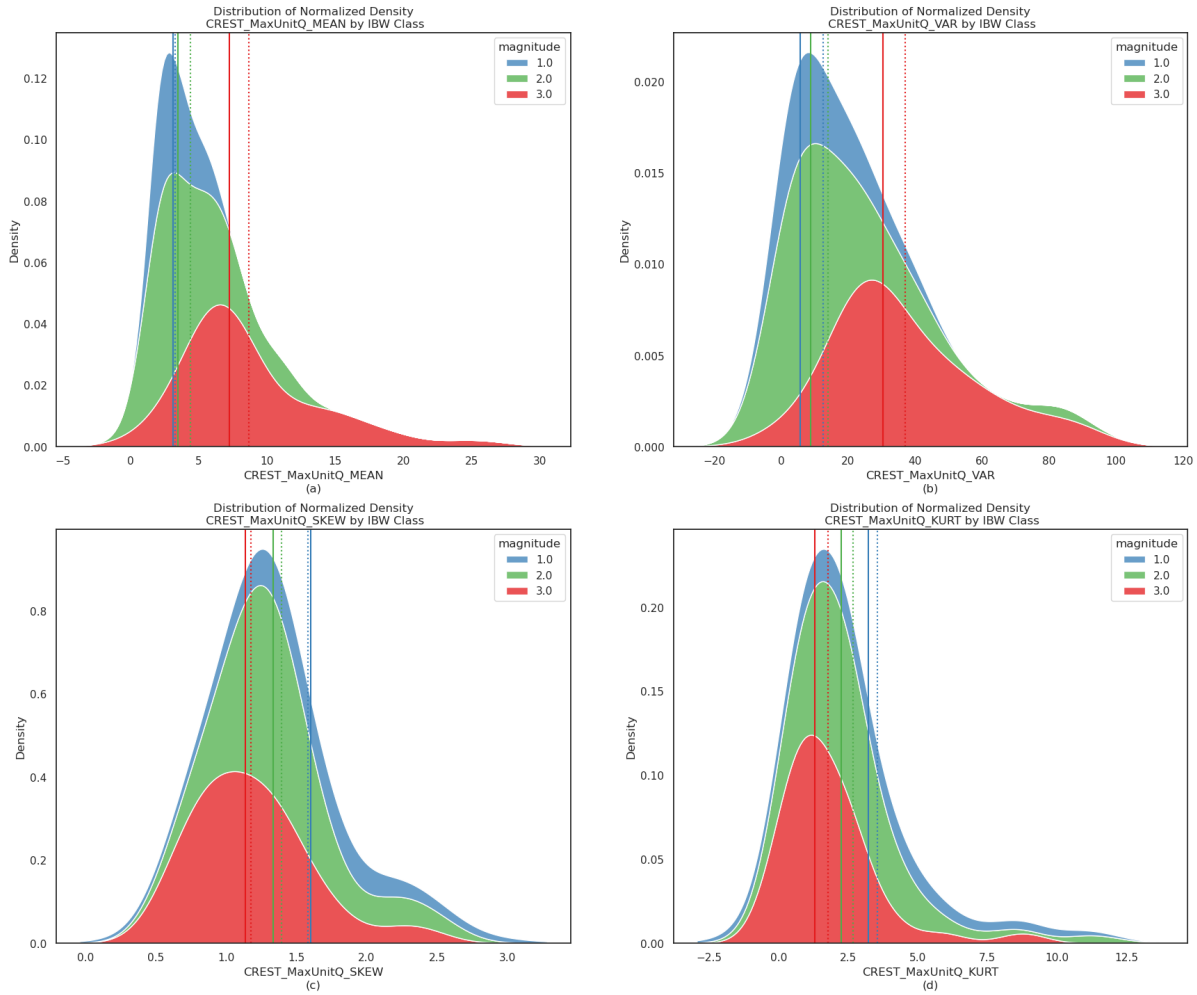


Figure A38: Distribution of MaxARI moments per IBW class. (a) Mean, (b) Variance, (c) Skewness, and (d) Kurtosis. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

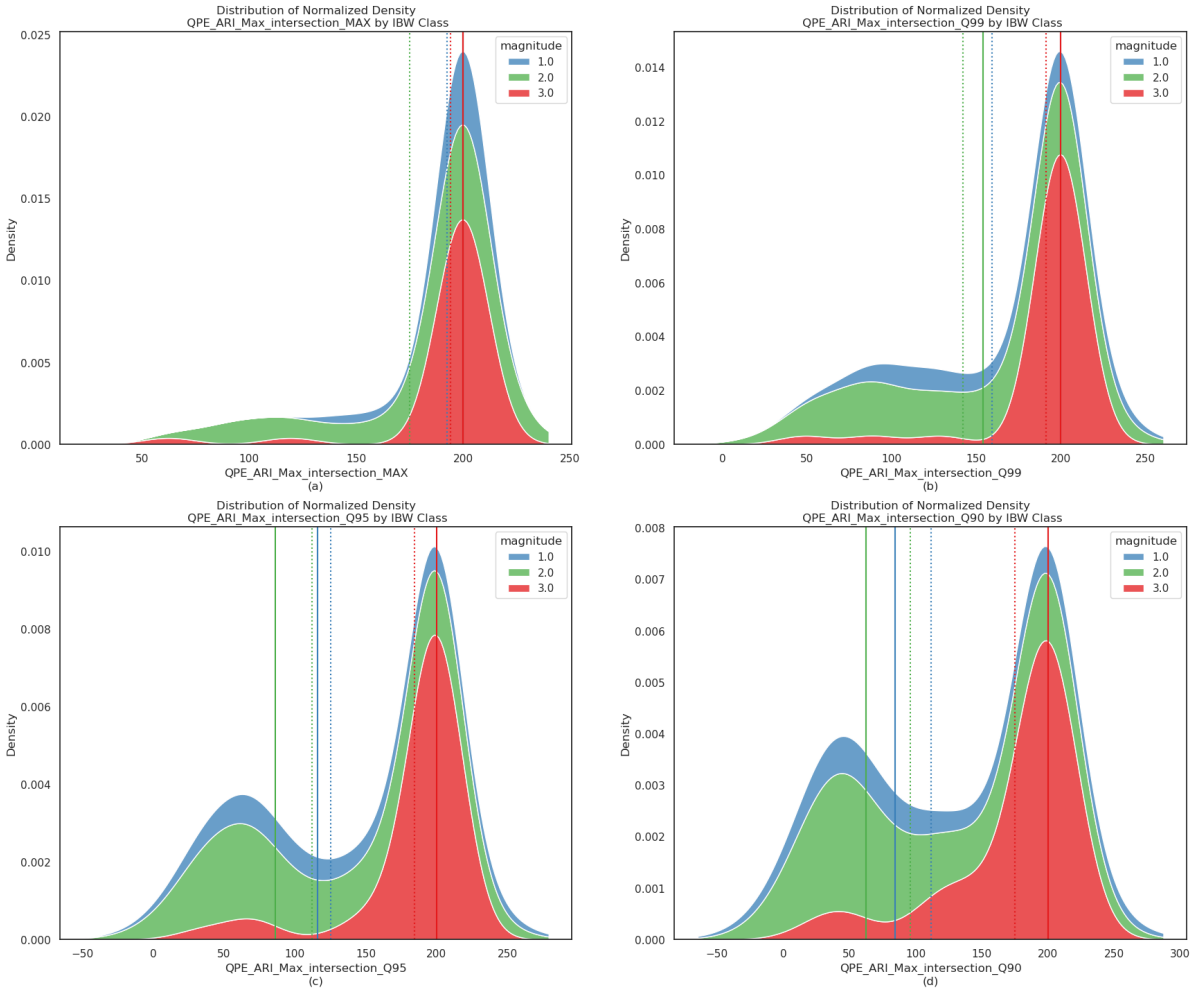


Figure A39: Distribution of MaxARI∩MaxUnitQ maximum and quantile per IBW class. (a) Maximum, (b) Q99, (c) Q95, and (d) Q99. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

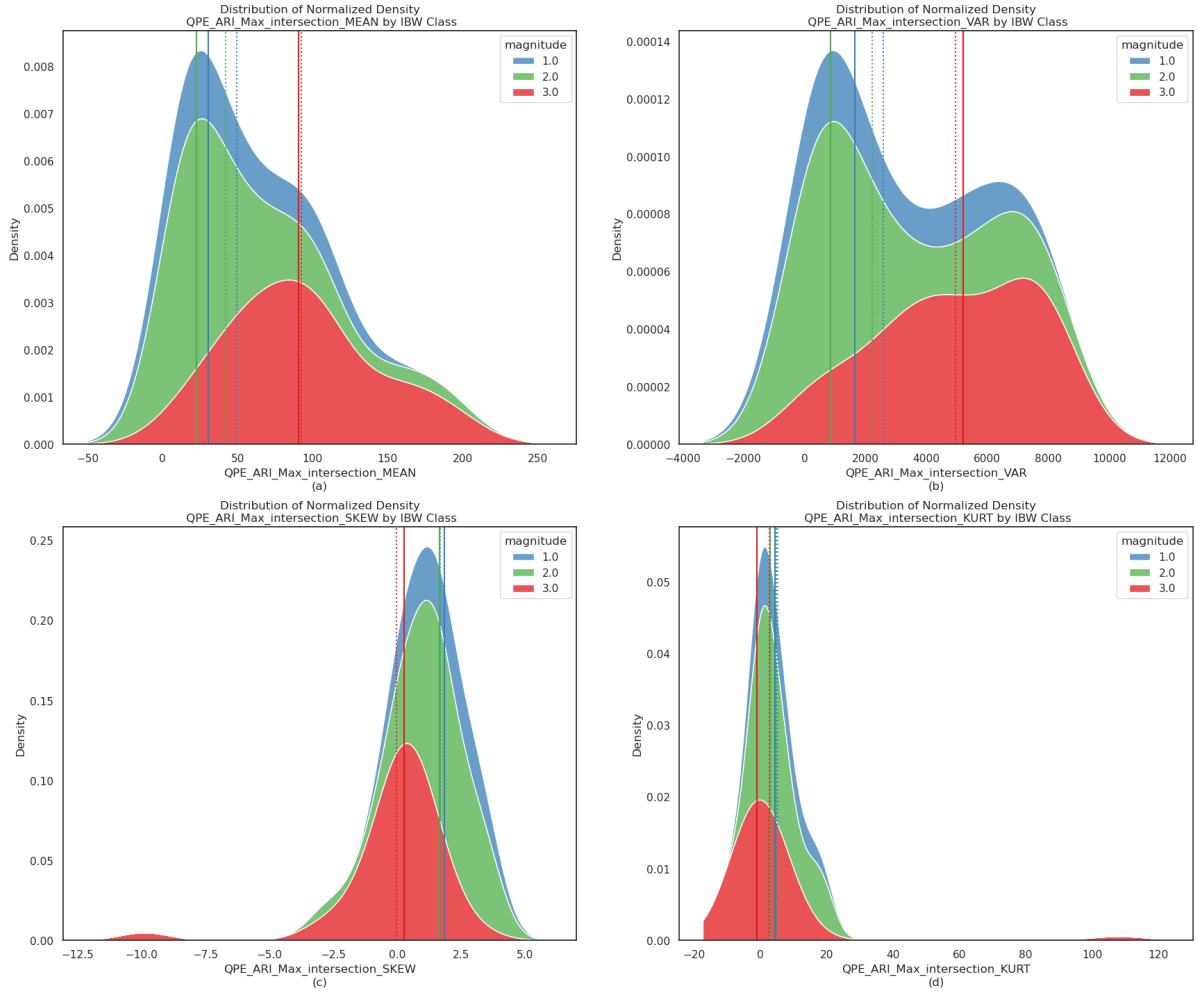


Figure A40: Distribution of MaxARI∩MaxUnitQ moments per IBW class. (a) Mean, (b) Variance, (c) Skewness, and (d) Kurtosis. Measures of central tendency are color-coded, medians are presented in solid vertical lines, and means are shown using dotted lines.

FFSI V2 Prompt Definition

Listing A1: ChatGPT prompt composed of extended *textualized* FFSI definitions, specific instructions for probability calculations, and output formatting.

```
Given the following severity category definitions:
```

1. "MINOR": if there is mention widespread or significant flooding of a bike trail or a walking trail; or if there is mention of a bike trail or a walking trail being impassable or closed; or if there is mention of a dirt road being impassable or closed; or if there is mention of a bike trail or a walking trail being flooded; or if there is mention of a dirt road being flooded; or if there is mention of a shoulder or culvert being flooded; or if there is mention of railroad tracks being approached by water, or ponding; or if there is mention of secondary roads being approached by water, or ponding; or if there is mention of primary roads being approached by water, or ponding; or if there is mention of a basement, crawlspace or garage flooded on a non-permanent structure; or if there is mention of a non-permanent structure being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of a mobile home or trailer being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of a house or apartment building being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of a non-permanent structure being approached by water; or if there is mention of a mobile home or trailer being approached by water; or if there is mention of a house or apartment building being approached by water; or if there is mention of a commercial building being approached by water; or if there is mention of a low-lying area or a field being flooded; or if there is mention of a campground or park being flooded; or if there is mention of a yard being approached by water; or if there is mention of a parking lot being approached by water.

2. "MODERATE": if there is mention of a bike trail or a walking trail being washed out or damaged; or if there is mention of a dirt road being washed out or damaged; or if there is mention widespread or significant flooding of a dirt road; or if there is mention of a shoulder or culvert being impassable or closed; or if there is mention of railroad tracks being impassable or closed; or if there is mention of secondary roads being impassable or closed; or if there is mention of railroad tracks being flooded; or if there is mention of secondary roads being flooded; or if there is mention of primary roads being approached by water, or ponding; or if there is mention of highways or interstate highways being approached by water, or ponding; or if there is mention of a person or vehicle stranded, stalled, stuck, or trapped; or if there is mention of first floor (1st floor) or ground floor being flooded on a non-permanent structure; or if there is mention of water entering the first floor (1st floor) on a non-permanent structure; or if there is mention of a basement, crawlspace or garage flooded on a mobile home or trailer; or if there is mention of a mobile home or trailer being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of a house or apartment building being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of a commercial building being surrounded by water, having flooded foundation, or having water entering the basement or crawlspace; or if there is mention of widespread or significant flooding in low-lying areas or a field; or if there is mention of widespread or significant flooding in a campground or park; or if there is mention of a yard being flooded; or if there is mention of a parking lot being flooded.

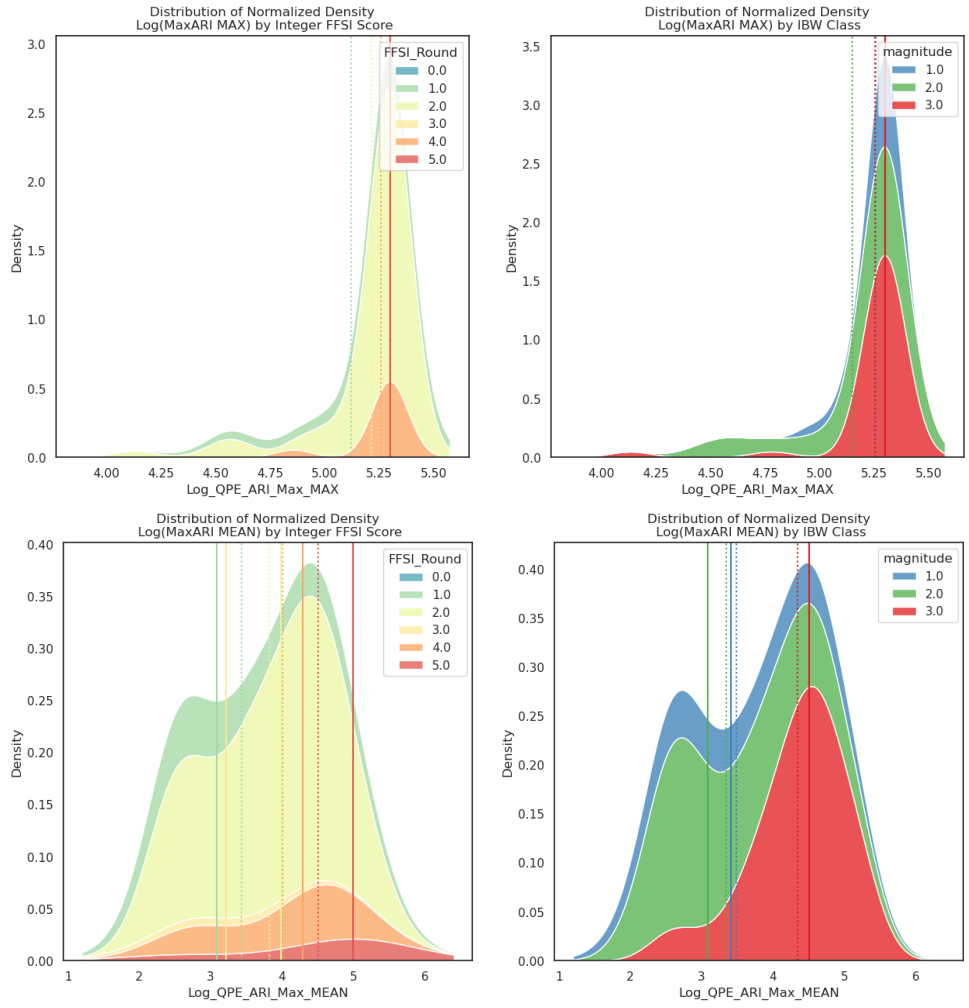
```
3. "SERIOUS": if there is mention of a dirt road being
 ↪ washed out or damaged; or if there is mention of a
 ↪ shoulder or culvert being washed out or damaged; or if
 ↪ there is mention widespread or significant flooding of
 ↪ a shoulder or culvert; or if there is mention
 ↪ widespread or significant flooding of railroad tracks;
 ↪ or if there is mention widespread or significant
 ↪ flooding of secondary roads; or if there is mention of
 ↪ railroad tracks being impassable or closed; or if there
 ↪ is mention of secondary roads being impassable or
 ↪ closed; or if there is mention of secondary roads being
 ↪ flooded; or if there is mention of primary roads being
 ↪ flooded; or if there is mention of highways or
 ↪ interstate highways being flooded; or if there is
 ↪ mention of a person or vehicle being flooded, or
 ↪ floated; or if there is mention of a commercial
 ↪ vehicle, tractor, or trailer being flooded, or floated;
 ↪ or if there is mention of a person or vehicle stranded,
 ↪ stalled, stuck, or trapped; or if there is mention of a
 ↪ commercial vehicle, tractor, or trailer, being
 ↪ stranded, stalled, stuck, or trapped; or if there is
 ↪ mention of a non-permanent structure being swept away,
 ↪ submerged, destroyed, or floated; or if there is
 ↪ mention of water rescues on a non-permanent structure;
 ↪ or if there is mention of first floor (1st floor) or
 ↪ ground floor being flooded on mobile homes or trailers;
 ↪ or if there is mention of water entering the first
 ↪ floor (1st floor) on a mobile home or trailer; or if
 ↪ there is mention of a basement, crawlspace or garage
 ↪ flooded on a house or apartment building; or if there
 ↪ is mention of a basement, crawlspace or garage flooded
 ↪ on a commercial building; or if there is mention of a
 ↪ commercial building being surrounded by water, having
 ↪ flooded foundation, or having water entering the
 ↪ basement or crawlspace; or if there is mention of
 ↪ widespread or significant flooding in a campground or
 ↪ park; or if there is mention of widespread or
 ↪ significant flooding in a yard; or if there is mention
 ↪ of widespread or significant flooding in a parking lot.
```

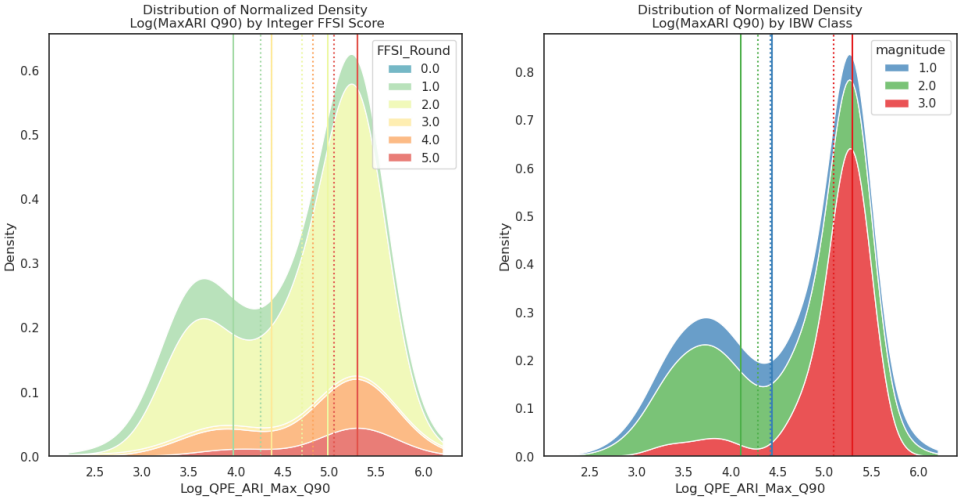
4. "SEVERE": if there is mention of railroad tracks being
↪ washed away or damaged; or if there is mention of
↪ secondary roads being washed away or damaged; or if
↪ there is mention of primary roads being washed away or
↪ damaged; or if there is mention widespread or
↪ significant flooding of primary roads; or if there is
↪ mention widespread or significant flooding of highways
↪ or interstate highways; or if there is mention of
↪ primary roads being impassable or closed; or if there
↪ is mention of highways or interstate highways being
↪ impassable or closed; or if there is mention of a
↪ person or vehicle being swept away; or if there is
↪ mention of a person or vehicle being submerged; or if
↪ there is mention of water rescues on a person or
↪ vehicle; or if there is mention of water rescues on a
↪ commercial vehicle, tractor or trailer; or if there is
↪ mention of water rescues on a mobile home or trailer;
↪ or if there is mention of water rescues on a house or
↪ apartment building; or if there is mention of water
↪ rescues on a commercial building; or if there is
↪ mention of first floor (1st floor) or ground floor
↪ being flooded on houses or apartment buildings; or if
↪ there is mention of first floor (1st floor) or ground
↪ floor being flooded on commercial buildings; or if
↪ there is mention of water entering the first floor (1st
↪ floor) on a house or apartment building; or if there is
↪ mention of water entering the first floor (1st floor)
↪ on a commercial building.

```
5. "CATASTROPHIC": if there is mention of primary roads
 ↪ being washed away or damaged; or if there is mention of
 ↪ highways or interstate highways being washed out or
 ↪ damaged; or if there is mention widespread or
 ↪ significant flooding of highways or interstate
 ↪ highways; or if there is mention of a person or vehicle
 ↪ being swept away; or if there is mention of a
 ↪ commercial vehicle, tractor, or trailer being swept
 ↪ away; or if there is mention of a person or vehicle
 ↪ being submerged; or if there is mention of a commercial
 ↪ vehicle, tractor, or trailer being submerged; or if
 ↪ there is mention of water rescues on a person or
 ↪ vehicle, not associated with a body of water, or near a
 ↪ body of water; or if there is mention of water rescues
 ↪ on a commercial vehicle, tractor or trailer, not
 ↪ associated with a body of water, or near a body of
 ↪ water; or if there is mention of a mobile home or
 ↪ trailer, being swept away, submerged, destroyed, or
 ↪ floated; or if there is mention of a house or apartment
 ↪ building being swept away, submerged, destroyed, or
 ↪ floated; or if there is mention of a commercial
 ↪ building being swept away, submerged, destroyed, or
 ↪ floated; or if there is mention of upper floors being
 ↪ flooded on a house or apartment building; or if there
 ↪ is mention of upper floors being flooded on a
 ↪ commercial building; or if there is mention of water
 ↪ rescues on a mobile home or trailer, not associated
 ↪ with a body of water, near a body of water, or on a
 ↪ body of water; or if there is mention of water rescues
 ↪ on a house or apartment building, not associated with a
 ↪ body of water, near a body of water, or on a body of
 ↪ water; or if there is mention of water rescues on a
 ↪ commercial building, not associated with a body of
 ↪ water, near a body of water, or on a body of water; or
 ↪ if there is mention of first floor (1st floor) or
 ↪ ground floor being flooded on house or apartment
 ↪ building; or if there is mention of first floor (1st
 ↪ floor) or ground floor being flooded on commercial
 ↪ building.

Assign impact class probabilities (in percents) to the following
 ↪ text, make sure the sum of all probabilities add up to 100%.
 ↪ Only return the probability values, and DO NOT EXPLAIN YOUR
 ↪ REASONING.

Your answer should be expressed following this JSON format:
```

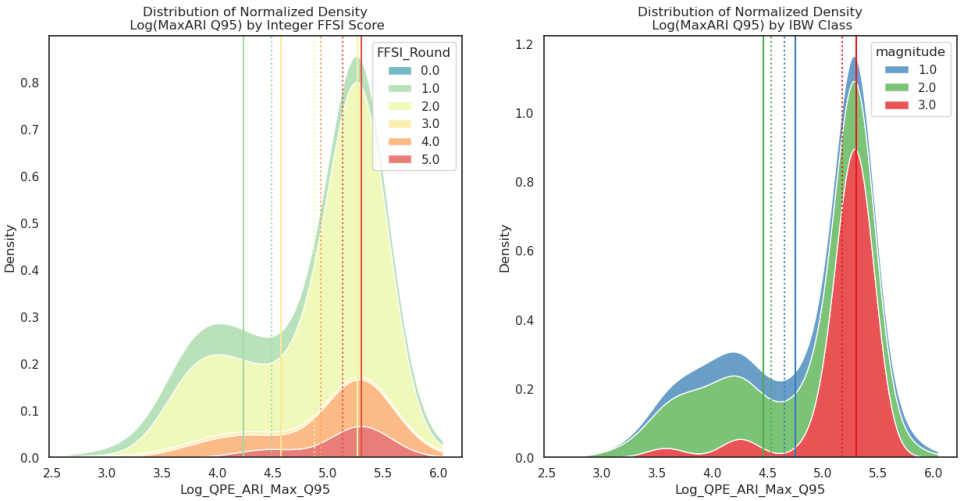
```
{
    "MINOR": <PROBABILITY PERCENT>,
    "MODERATE": <PROBABILITY PERCENT>,
    "SERIOUS": <PROBABILITY PERCENT>,
    "SEVERE": <PROBABILITY PERCENT>,
    "CATASTROPHIC": <PROBABILITY PERCENT>
}
```

FFSI vs IBW Per-Class Densities

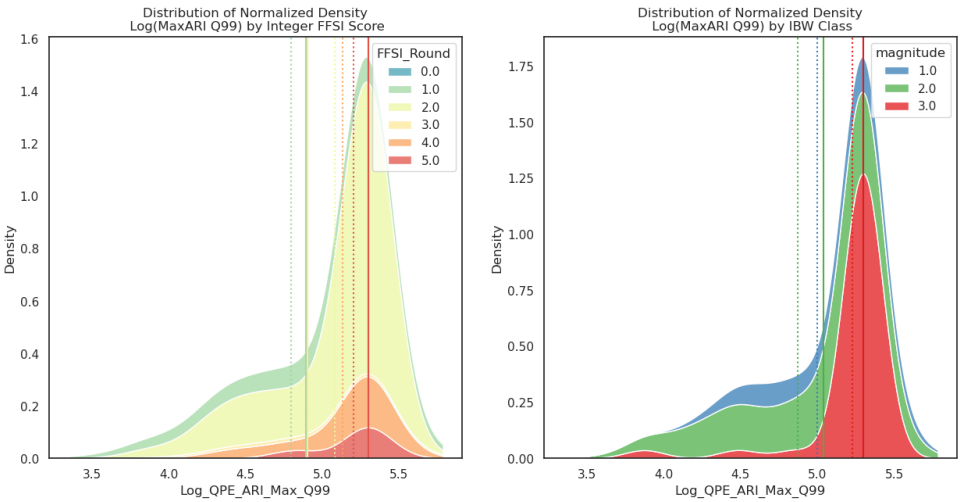



Distribution of Normalized Density
Log(MaxARI Q90) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q90

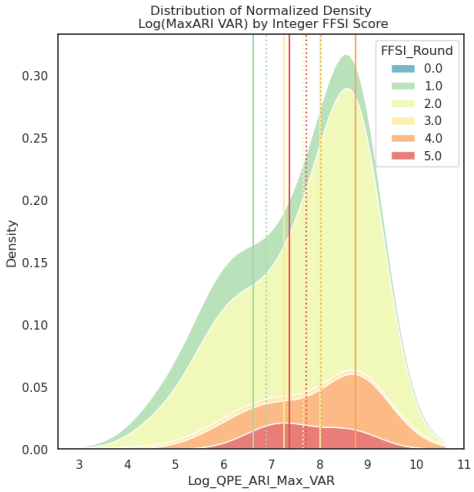
Distribution of Normalized Density
Log(MaxARI Q90) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_QPE_ARI_Max_Q90

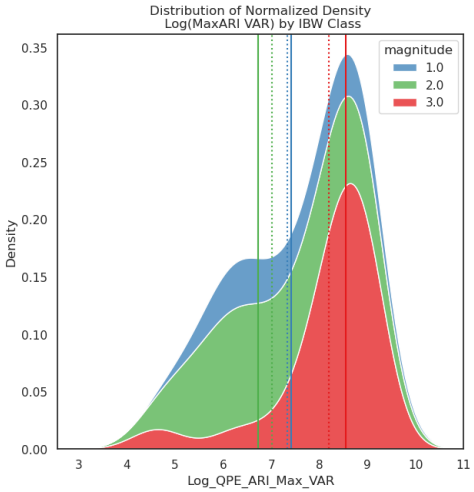

Distribution of Normalized Density
Log(MaxARI Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q95

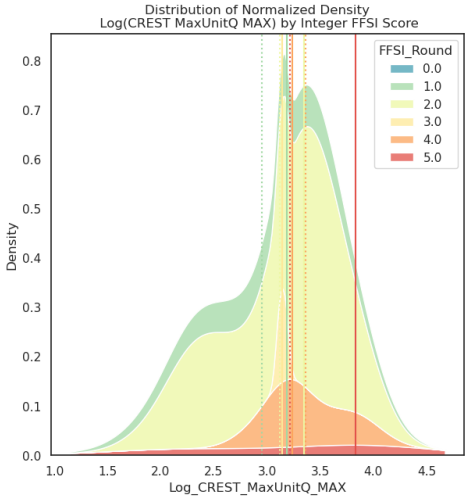
Distribution of Normalized Density
Log(MaxARI Q95) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_QPE_ARI_Max_Q95

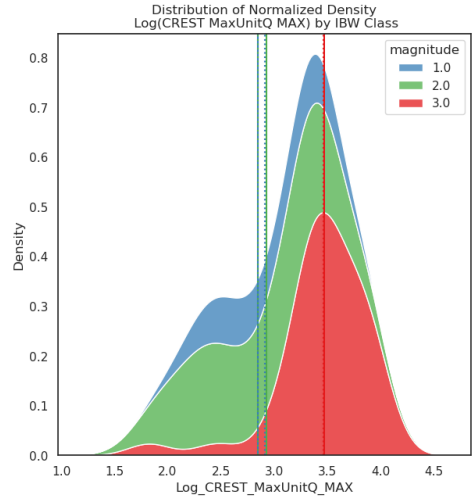

Distribution of Normalized Density
Log(MaxARI Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q99

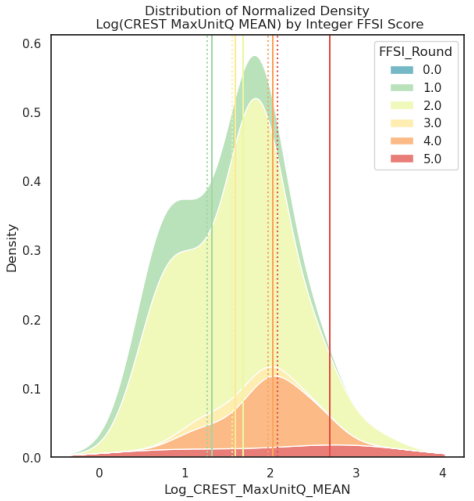
Distribution of Normalized Density
Log(MaxARI Q99) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_QPE_ARI_Max_Q99

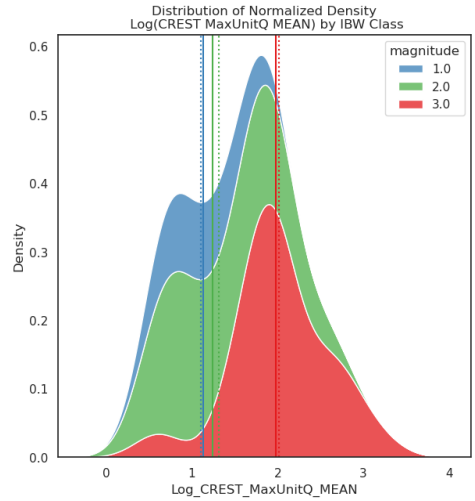

Distribution of Normalized Density
Log(MaxARI VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_VAR

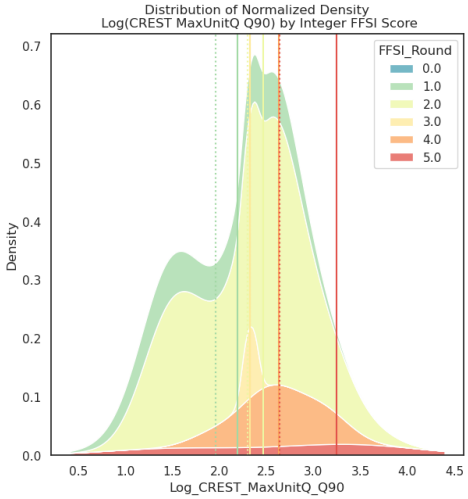

Distribution of Normalized Density
Log(MaxARI VAR) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_QPE_ARI_Max_VAR

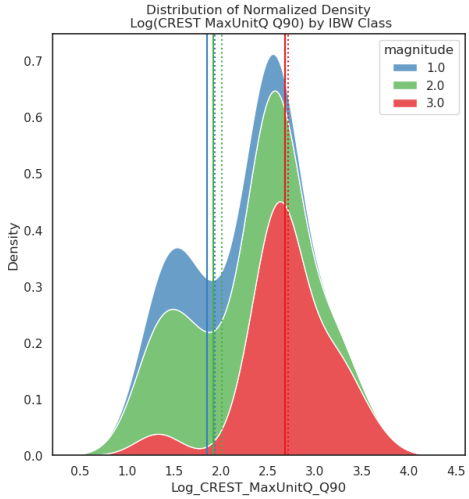

Distribution of Normalized Density
Log(CREST MaxUnitQ MAX) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_MAX

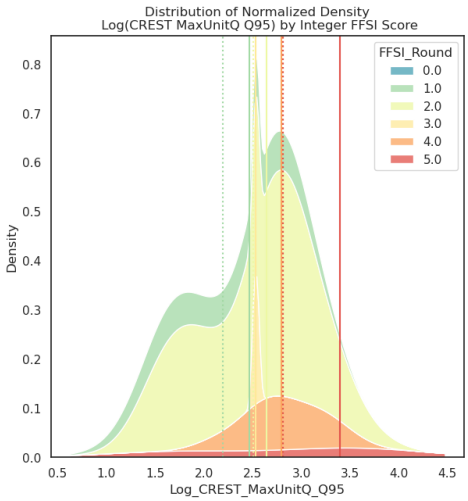

Distribution of Normalized Density
Log(CREST MaxUnitQ MAX) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_MAX

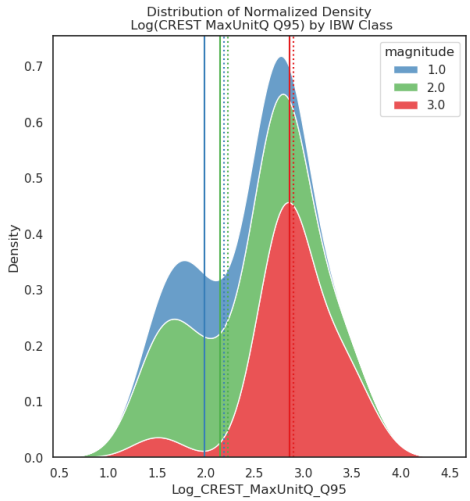

Distribution of Normalized Density
Log(CREST MaxUnitQ MEAN) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_MEAN

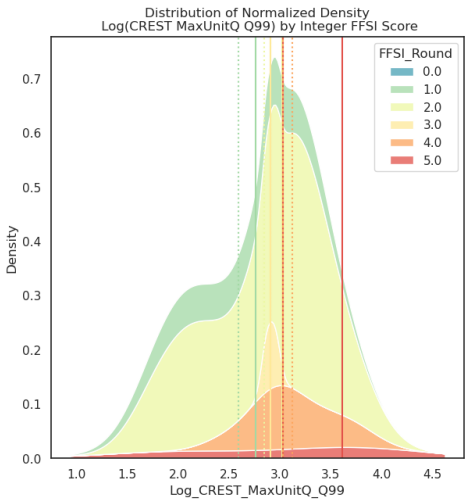

Distribution of Normalized Density
Log(CREST MaxUnitQ MEAN) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_MEAN

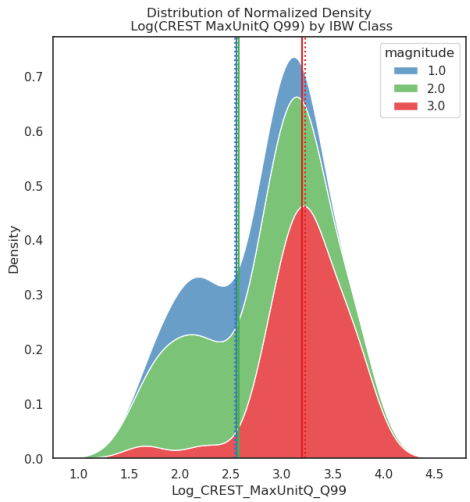

Distribution of Normalized Density
Log(CREST MaxUnitQ Q90) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q90

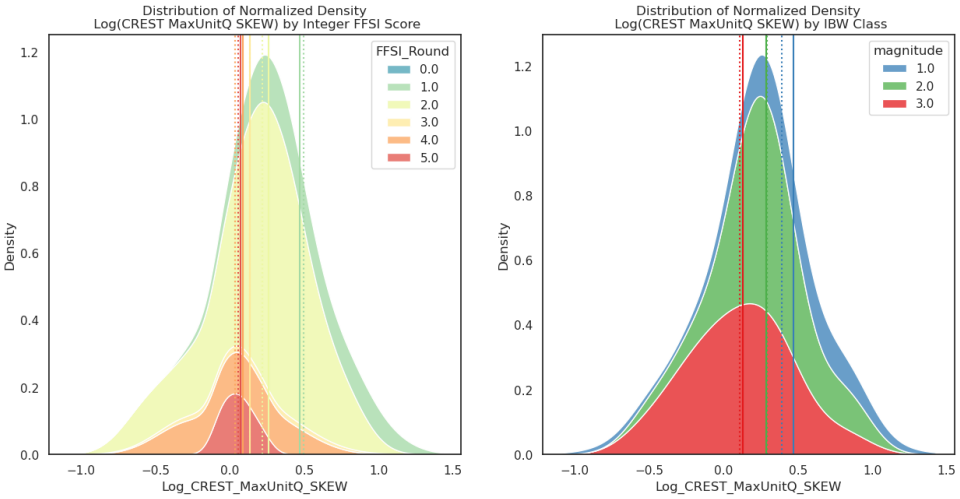

Distribution of Normalized Density
Log(CREST MaxUnitQ Q90) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_Q90


Distribution of Normalized Density
Log(CREST MaxUnitQ Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q95

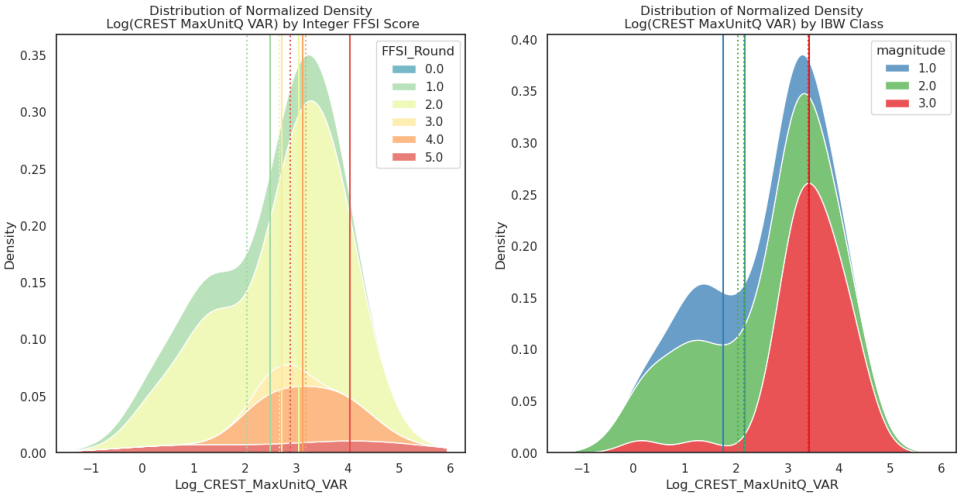

Distribution of Normalized Density
Log(CREST MaxUnitQ Q95) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_Q95


Distribution of Normalized Density
Log(CREST MaxUnitQ Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q99

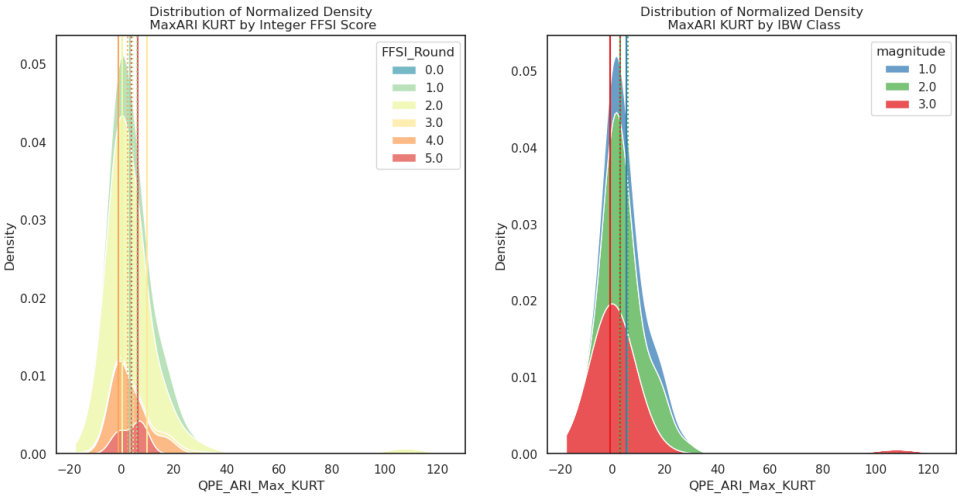

Distribution of Normalized Density
Log(CREST MaxUnitQ Q99) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_Q99


Distribution of Normalized Density
Log(CREST MaxUnitQ SKEW) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_SKEW

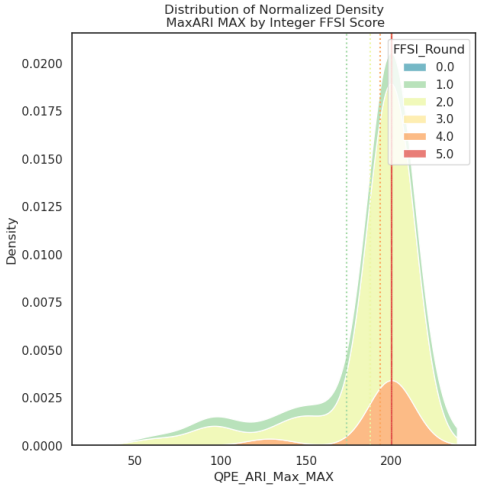
Distribution of Normalized Density
Log(CREST MaxUnitQ SKEW) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_SKEW

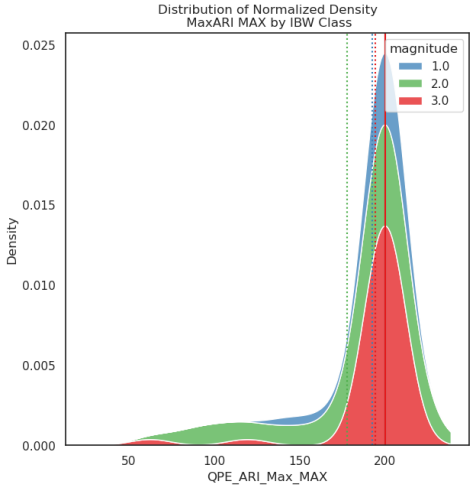

Distribution of Normalized Density
Log(CREST MaxUnitQ VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_VAR

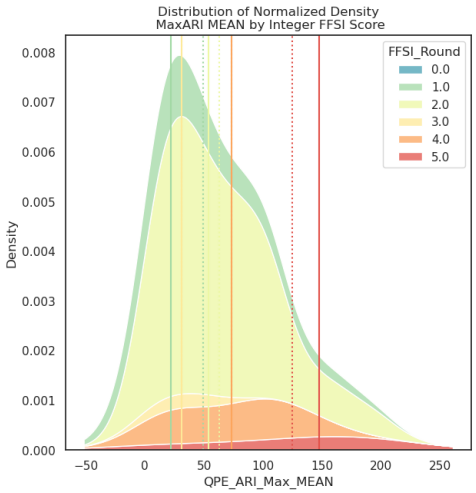
Distribution of Normalized Density
Log(CREST MaxUnitQ VAR) by IBW Class
magnitude
1.0
2.0
3.0
Density
Log_CREST_MaxUnitQ_VAR

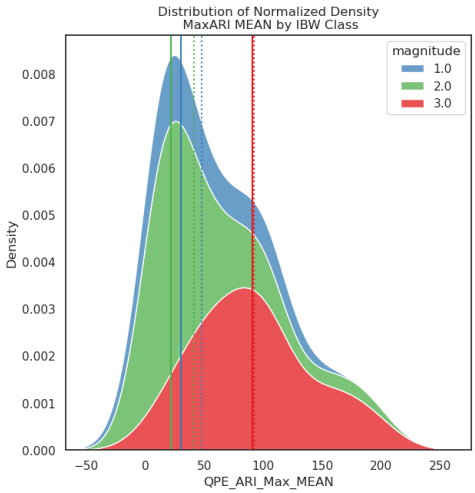

Distribution of Normalized Density
MaxARI KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_KURT

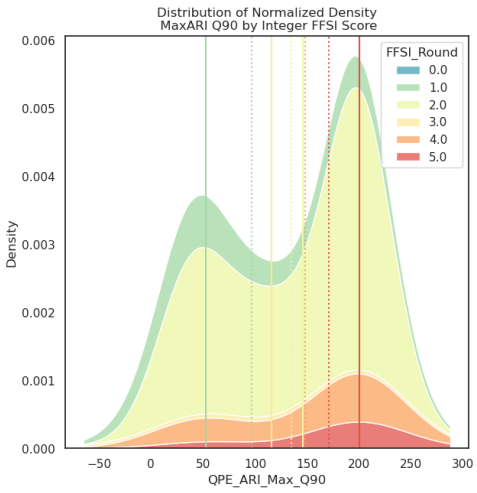
Distribution of Normalized Density
MaxARI KURT by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_KURT

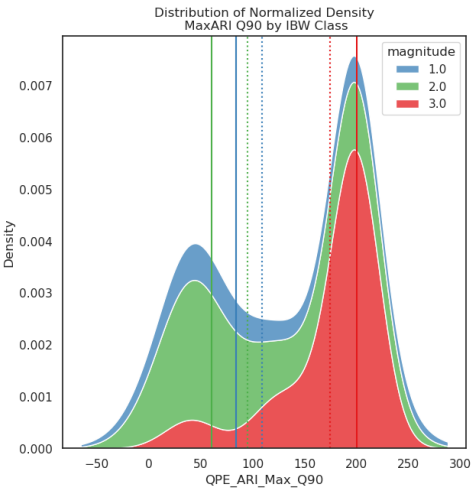

Distribution of Normalized Density
MaxARI MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MAX

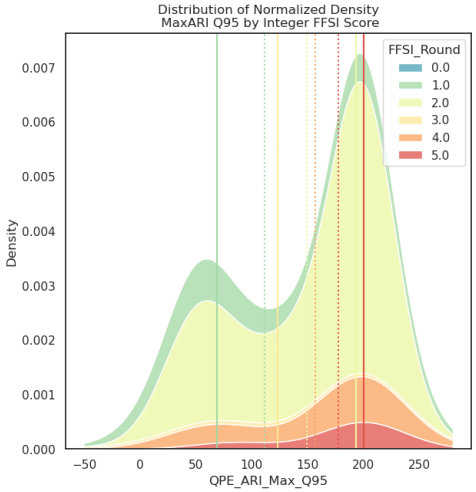

Distribution of Normalized Density
MaxARI MAX by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_MAX

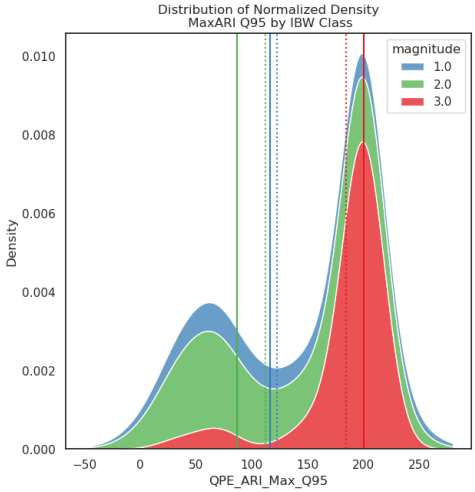

Distribution of Normalized Density
MaxARI MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MEAN

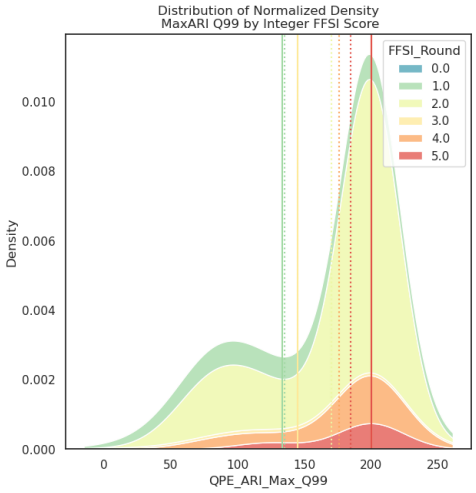

Distribution of Normalized Density
MaxARI MEAN by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_MEAN

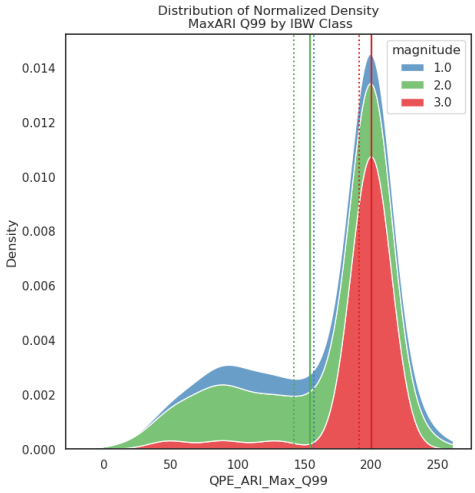

Distribution of Normalized Density
MaxARI Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q90

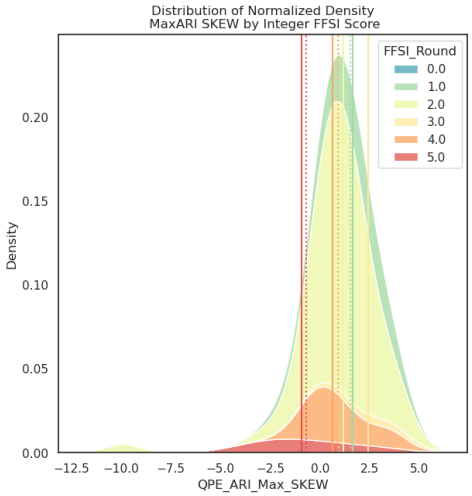

Distribution of Normalized Density
MaxARI Q90 by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_Q90

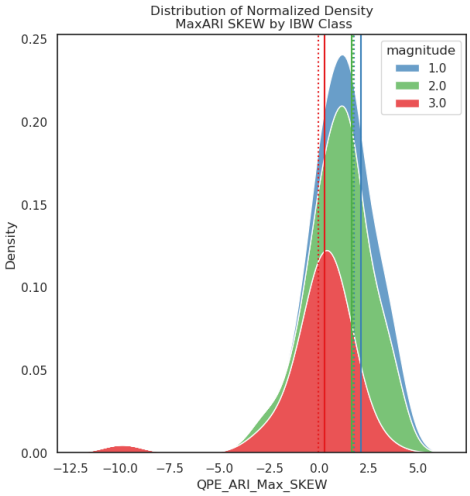

Distribution of Normalized Density
MaxARI Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q95

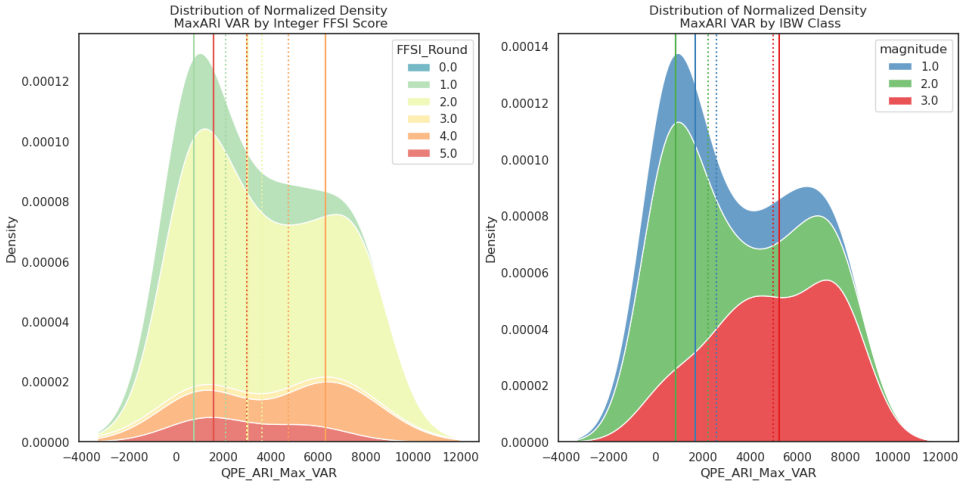

Distribution of Normalized Density
MaxARI Q95 by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_Q95


Distribution of Normalized Density
MaxARI Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q99

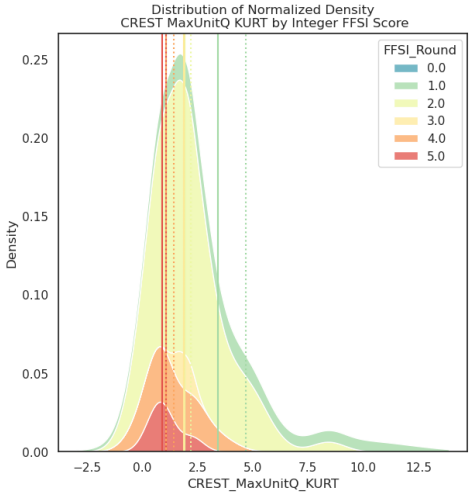

Distribution of Normalized Density
MaxARI Q99 by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_Q99

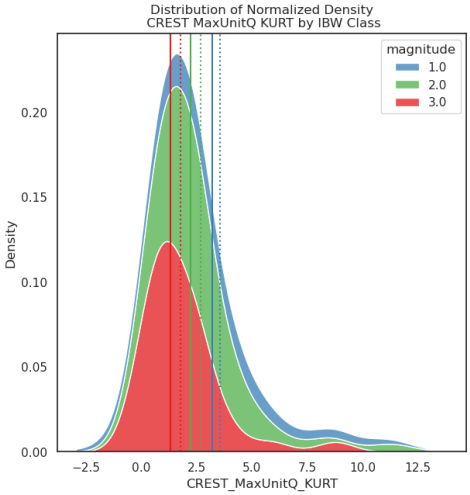

Distribution of Normalized Density
MaxARI SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_SKEW

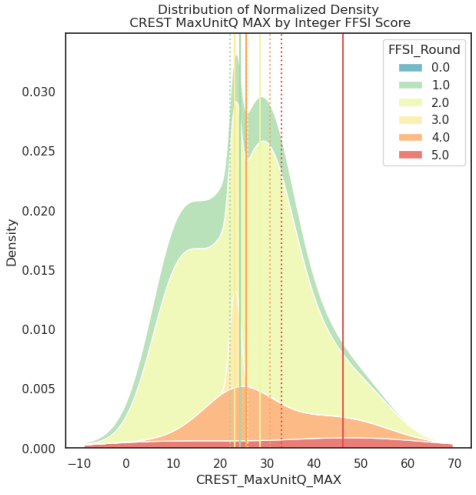

Distribution of Normalized Density
MaxARI SKEW by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_SKEW

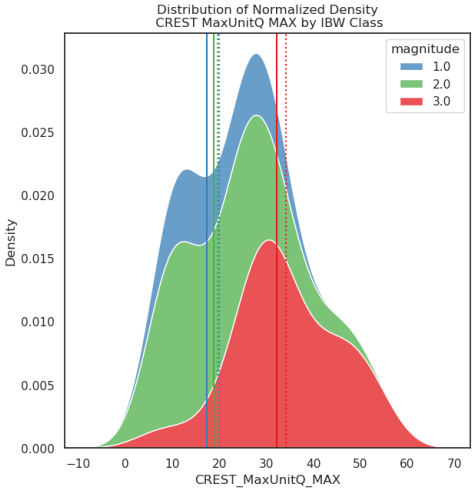

Distribution of Normalized Density
MaxARI VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_VAR

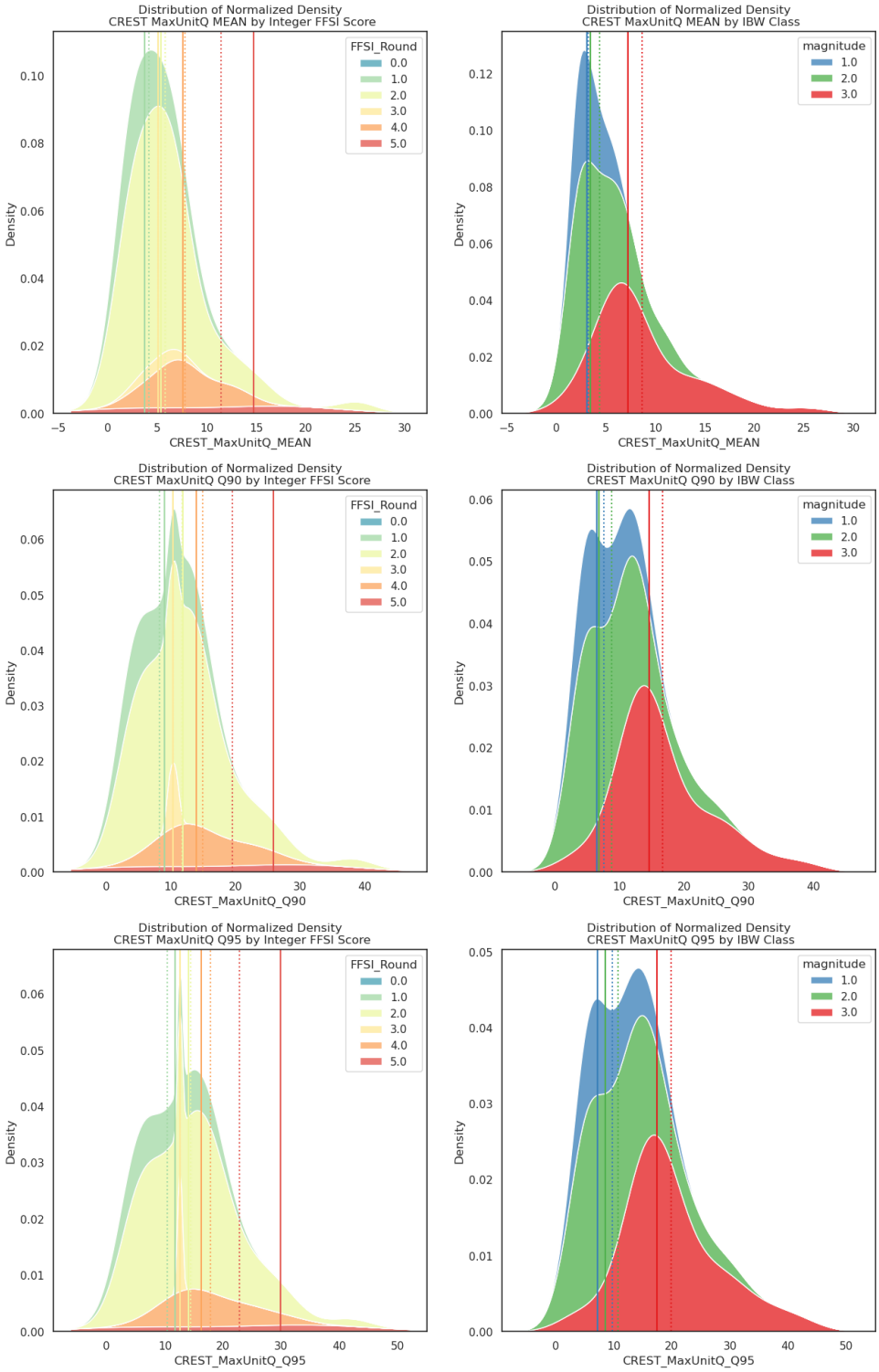
Distribution of Normalized Density
MaxARI VAR by IBW Class
magnitude
1.0
2.0
3.0
Density
QPE_ARI_Max_VAR

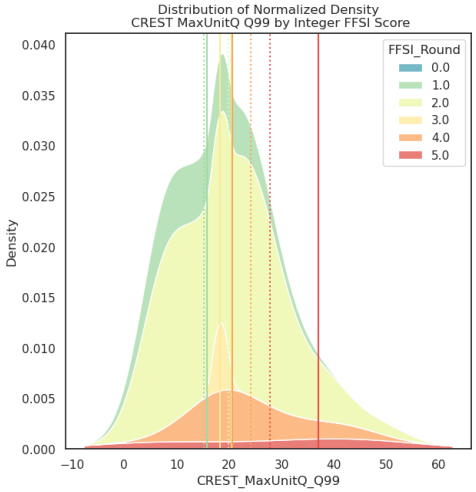

Distribution of Normalized Density
CREST MaxUnitQ KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_KURT

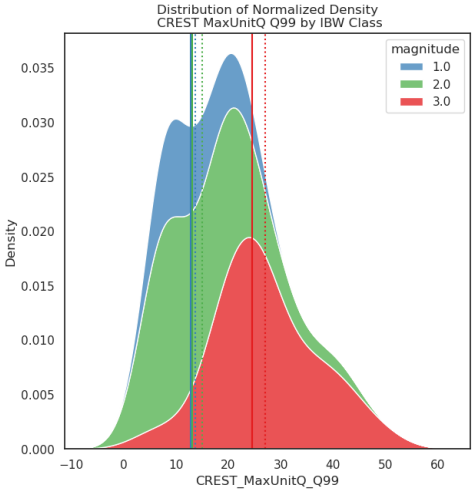

Distribution of Normalized Density
CREST MaxUnitQ KURT by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_KURT

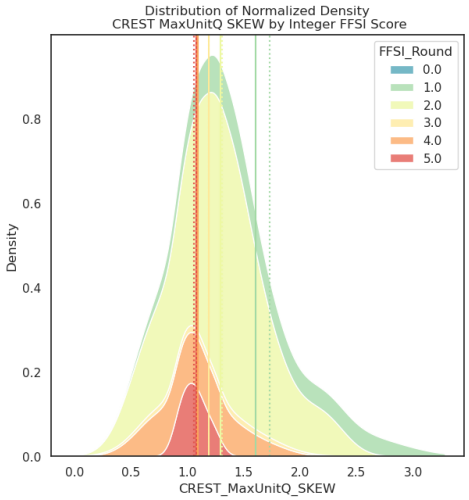

Distribution of Normalized Density
CREST MaxUnitQ MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MAX

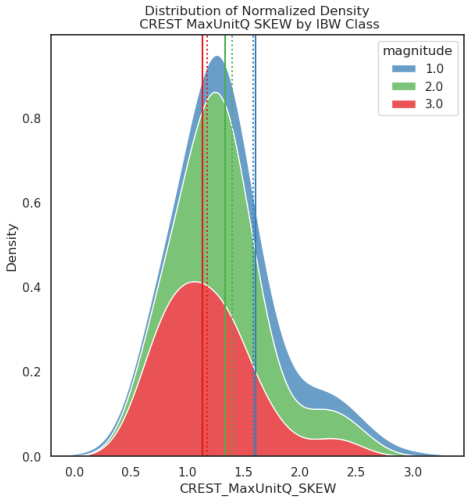

Distribution of Normalized Density
CREST MaxUnitQ MAX by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_MAX

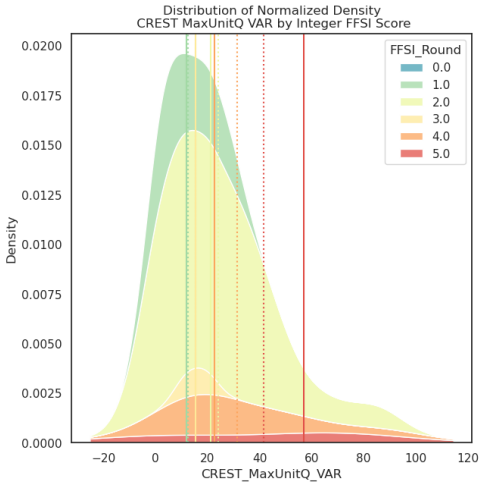

Distribution of Normalized Density
CREST MaxUnitQ MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MEAN
Distribution of Normalized Density
CREST MaxUnitQ MEAN by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_MEAN
Distribution of Normalized Density
CREST MaxUnitQ Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q90
Distribution of Normalized Density
CREST MaxUnitQ Q90 by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_Q90
Distribution of Normalized Density
CREST MaxUnitQ Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q95
Distribution of Normalized Density
CREST MaxUnitQ Q95 by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_Q95

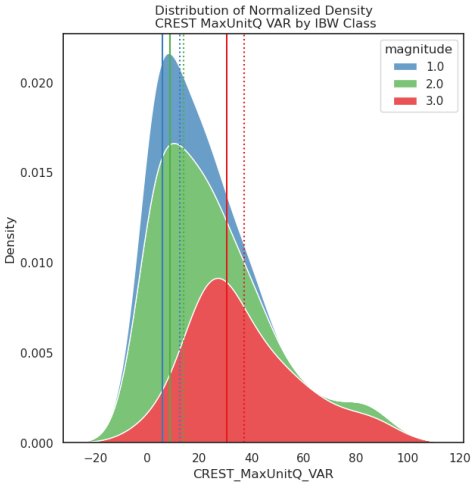

Distribution of Normalized Density
CREST MaxUnitQ Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q99


Distribution of Normalized Density
CREST MaxUnitQ Q99 by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_Q99

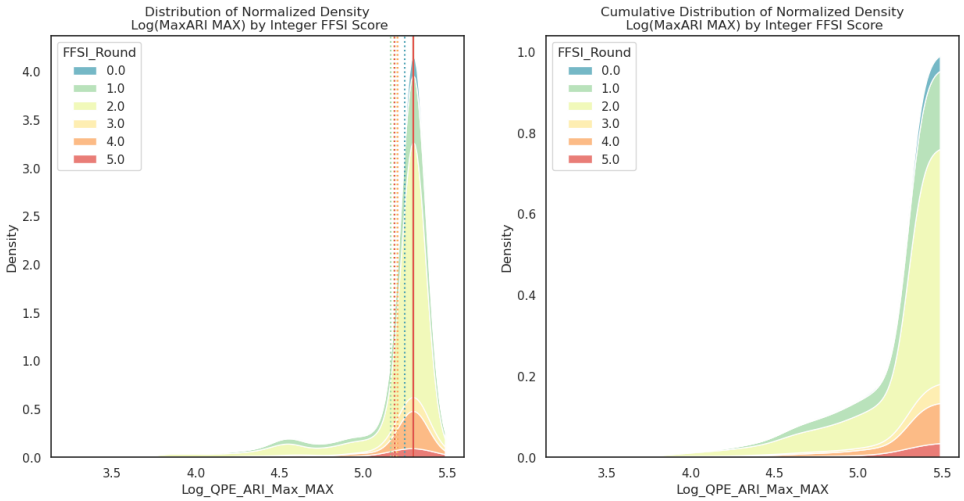

Distribution of Normalized Density
CREST MaxUnitQ SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_SKEW

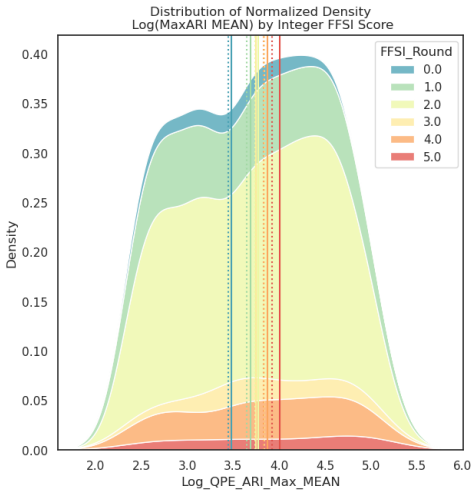

Distribution of Normalized Density
CREST MaxUnitQ SKEW by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_SKEW

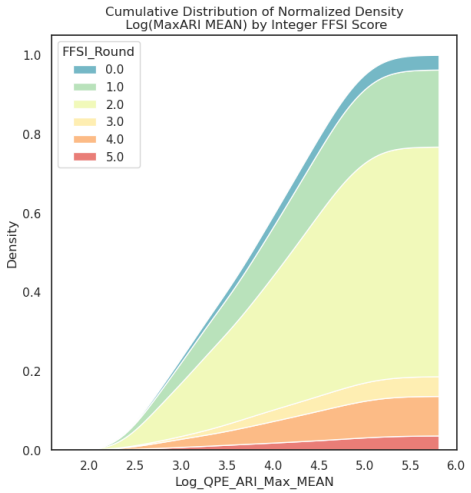

Distribution of Normalized Density
CREST MaxUnitQ VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_VAR

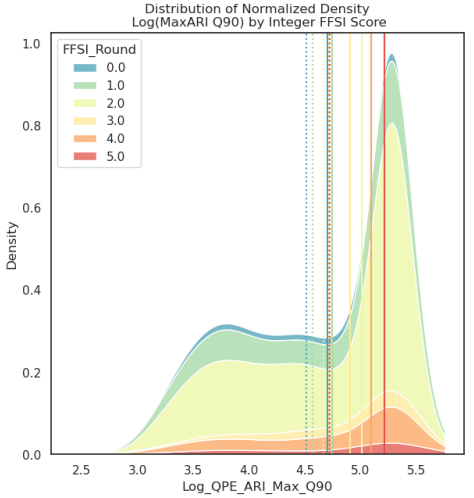

Distribution of Normalized Density
CREST MaxUnitQ VAR by IBW Class
magnitude
1.0
2.0
3.0
Density
CREST_MaxUnitQ_VAR

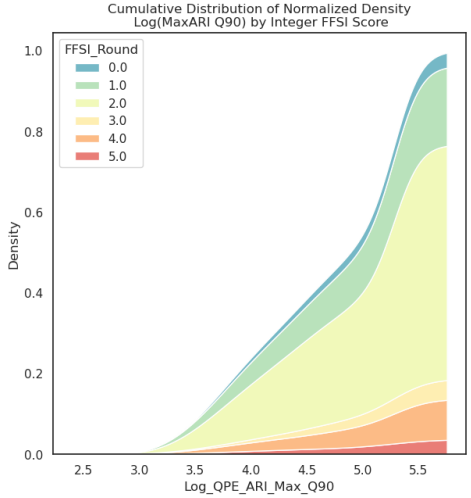
FFSI Per-Class Densities

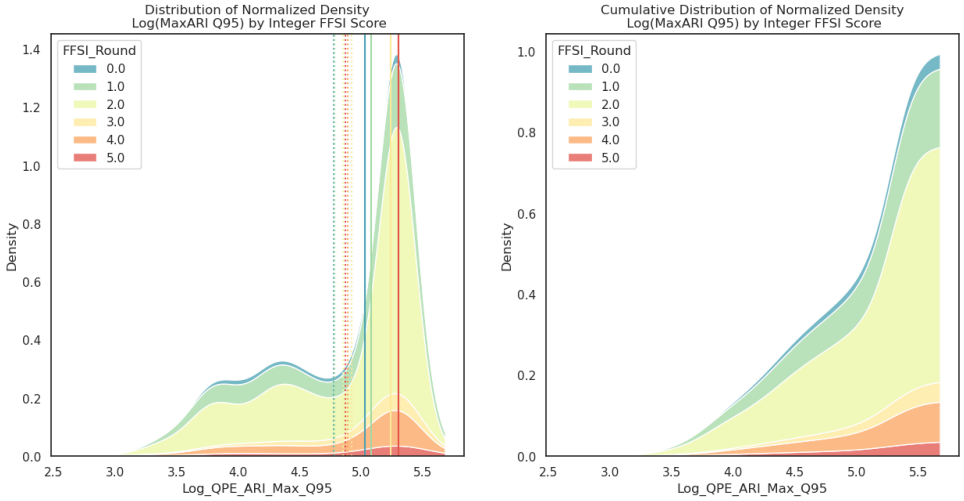




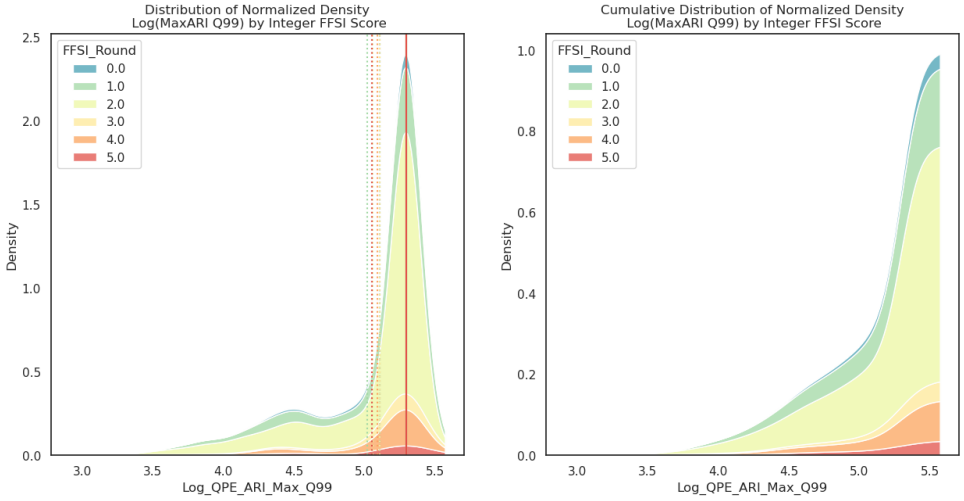




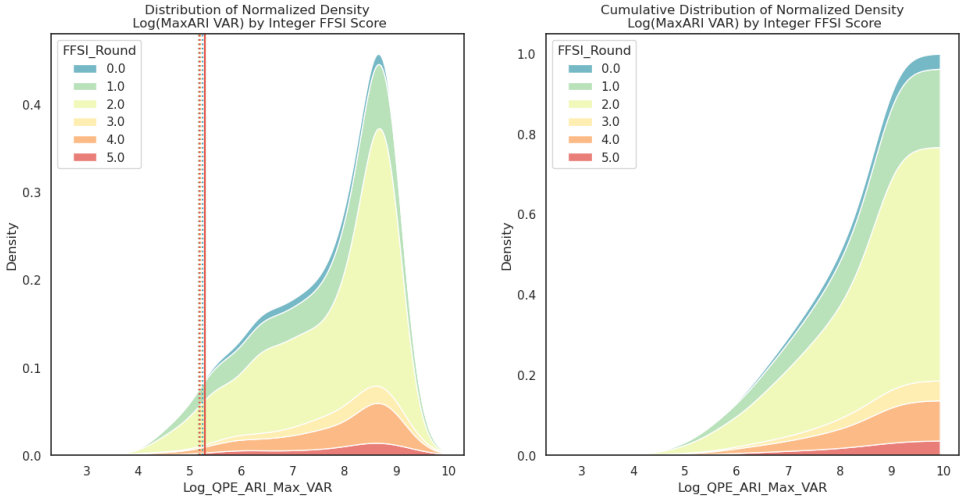



Distribution of Normalized Density
Log(MaxARI Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q95

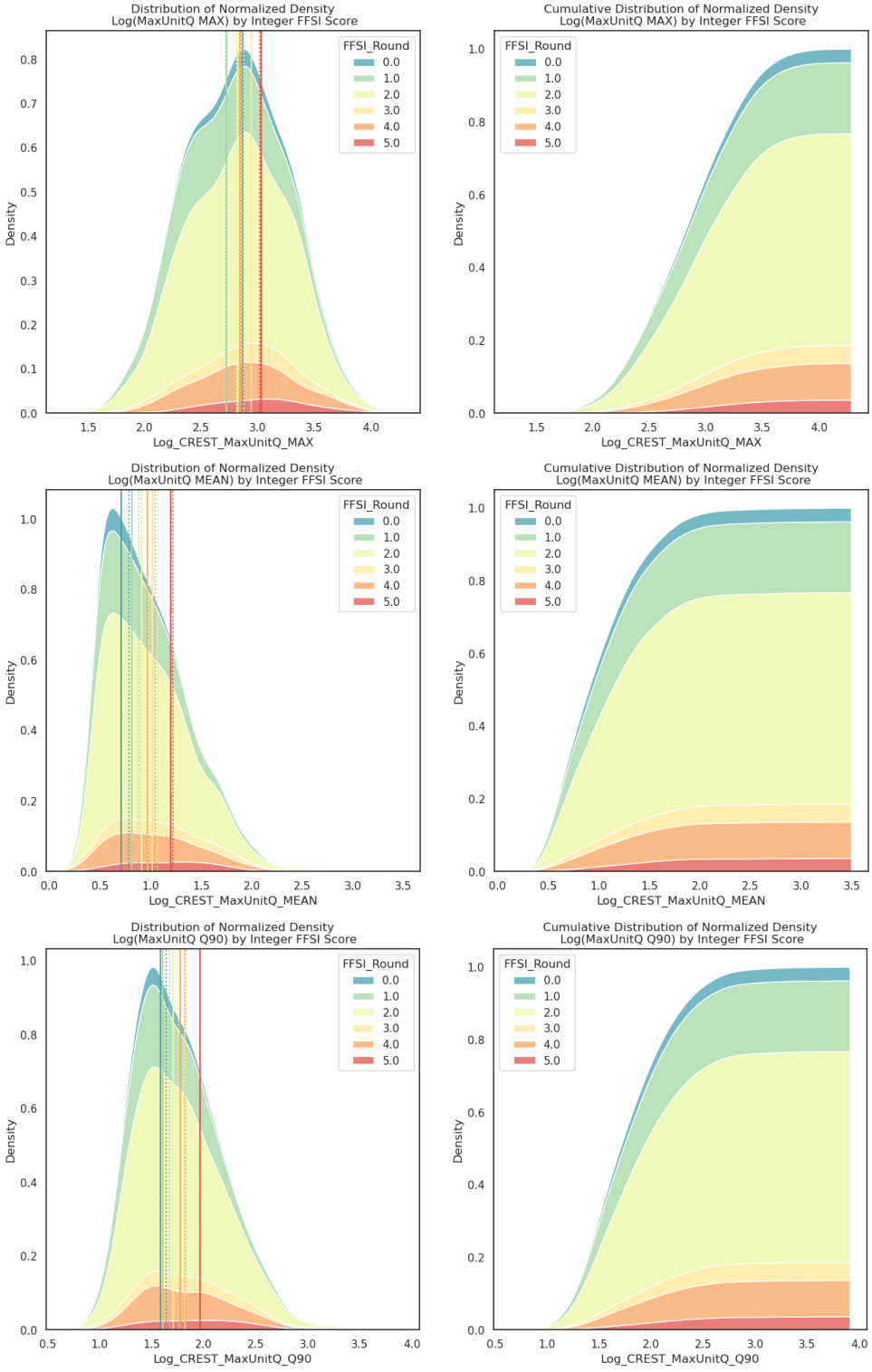
Cumulative Distribution of Normalized Density
Log(MaxARI Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q95

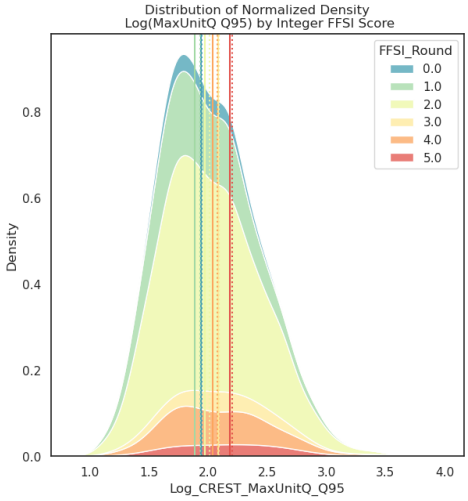

Distribution of Normalized Density
Log(MaxARI Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q99

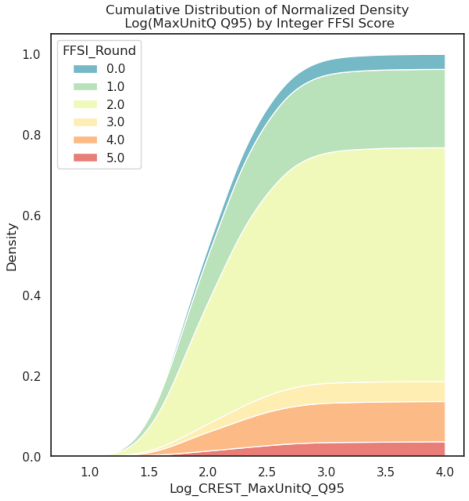
Cumulative Distribution of Normalized Density
Log(MaxARI Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_Q99

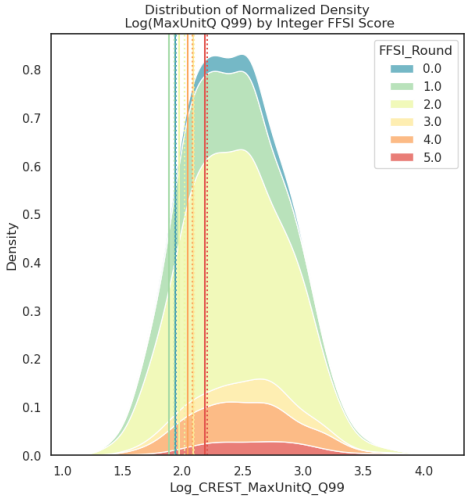

Distribution of Normalized Density
Log(MaxARI VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_VAR

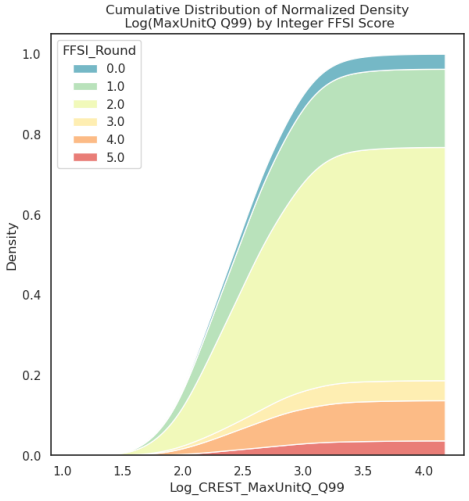
Cumulative Distribution of Normalized Density
Log(MaxARI VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_QPE_ARI_Max_VAR

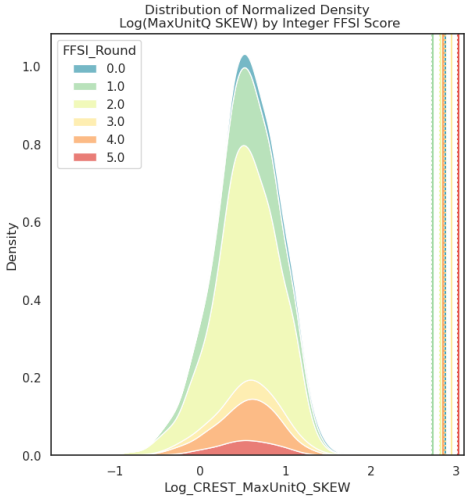

Distribution of Normalized Density
Log(MaxUnitQ MAX) by Integer FFSI Score
Cumulative Distribution of Normalized Density
Log(MaxUnitQ MAX) by Integer FFSI Score
Distribution of Normalized Density
Log(MaxUnitQ MEAN) by Integer FFSI Score
Cumulative Distribution of Normalized Density
Log(MaxUnitQ MEAN) by Integer FFSI Score
Distribution of Normalized Density
Log(MaxUnitQ Q90) by Integer FFSI Score
Cumulative Distribution of Normalized Density
Log(MaxUnitQ Q90) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_MAX
Log_CREST_MaxUnitQ_MEAN
Log_CREST_MaxUnitQ_Q90

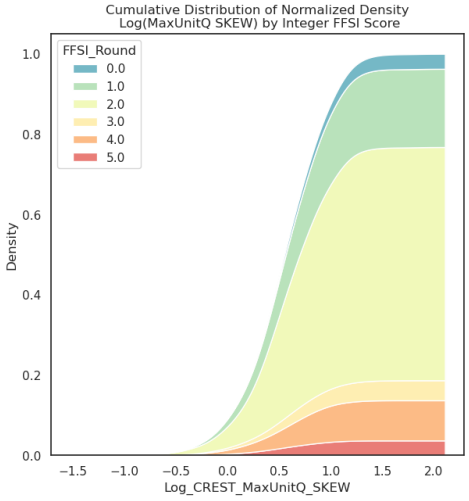

Distribution of Normalized Density
Log(MaxUnitQ Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q95

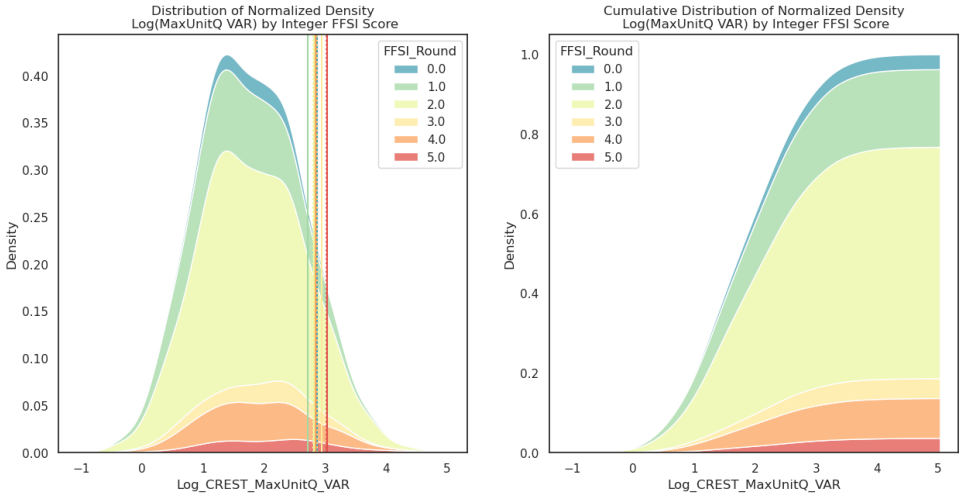

Cumulative Distribution of Normalized Density
Log(MaxUnitQ Q95) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q95


Distribution of Normalized Density
Log(MaxUnitQ Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q99

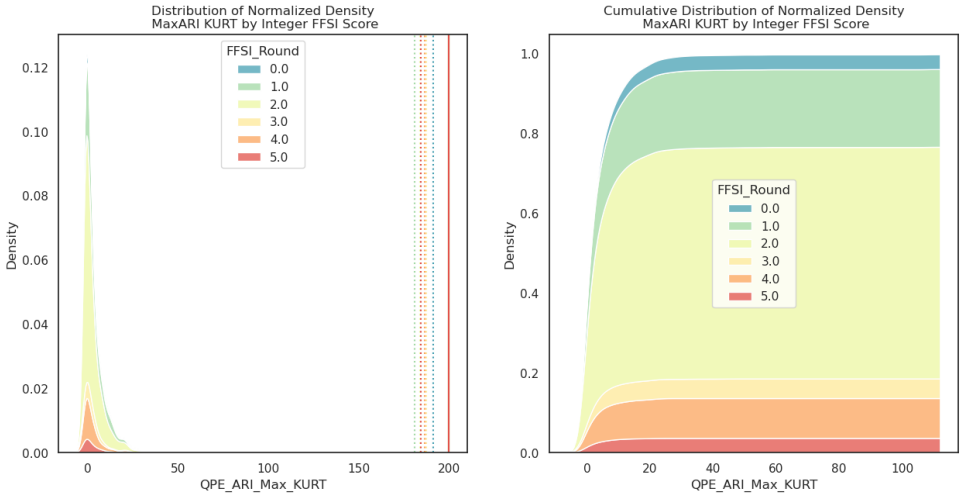

Cumulative Distribution of Normalized Density
Log(MaxUnitQ Q99) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_Q99


Distribution of Normalized Density
Log(MaxUnitQ SKEW) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_SKEW

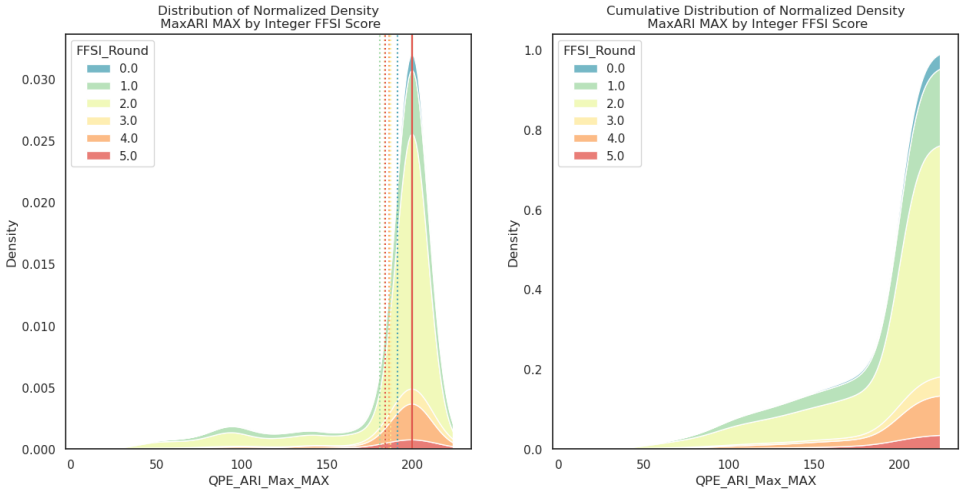

Cumulative Distribution of Normalized Density
Log(MaxUnitQ SKEW) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_SKEW


Distribution of Normalized Density
Log(MaxUnitQ VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_VAR

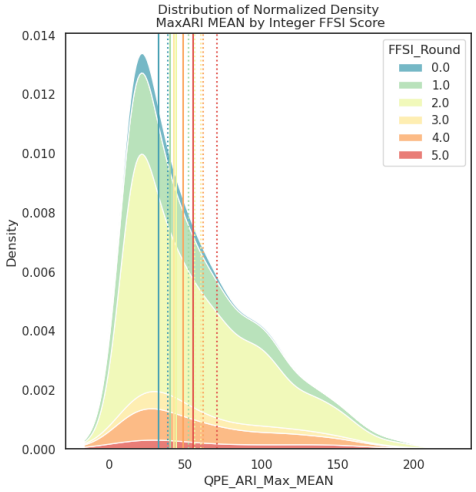
Cumulative Distribution of Normalized Density
Log(MaxUnitQ VAR) by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
Log_CREST_MaxUnitQ_VAR

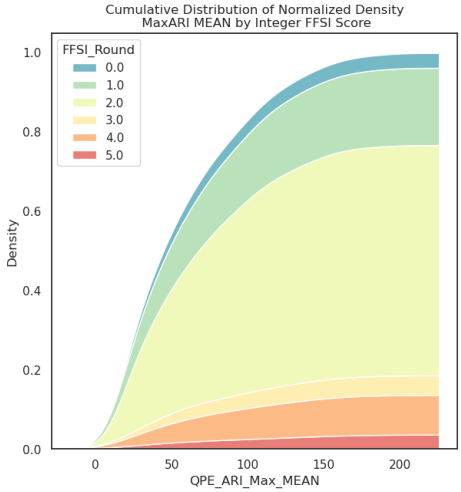

Distribution of Normalized Density
MaxARI KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_KURT

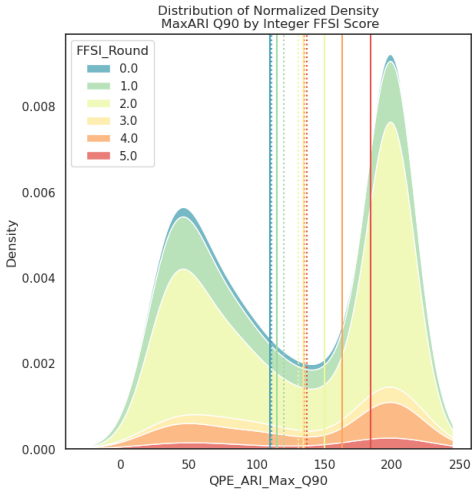
Cumulative Distribution of Normalized Density
MaxARI KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_KURT

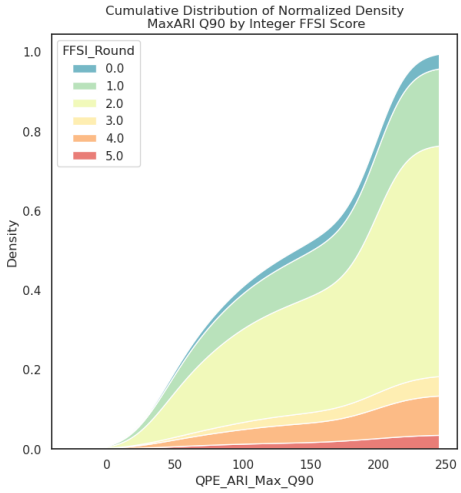

Distribution of Normalized Density
MaxARI MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MAX

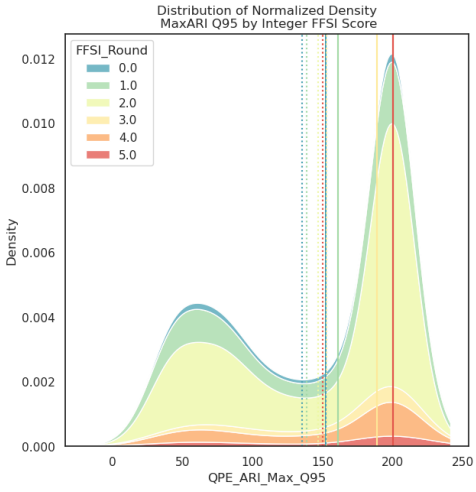
Cumulative Distribution of Normalized Density
MaxARI MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MAX

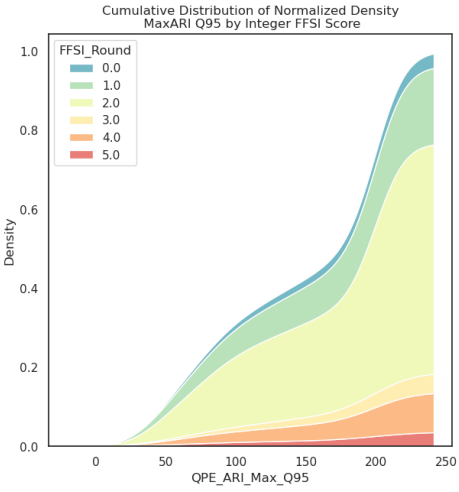

Distribution of Normalized Density
MaxARI MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MEAN

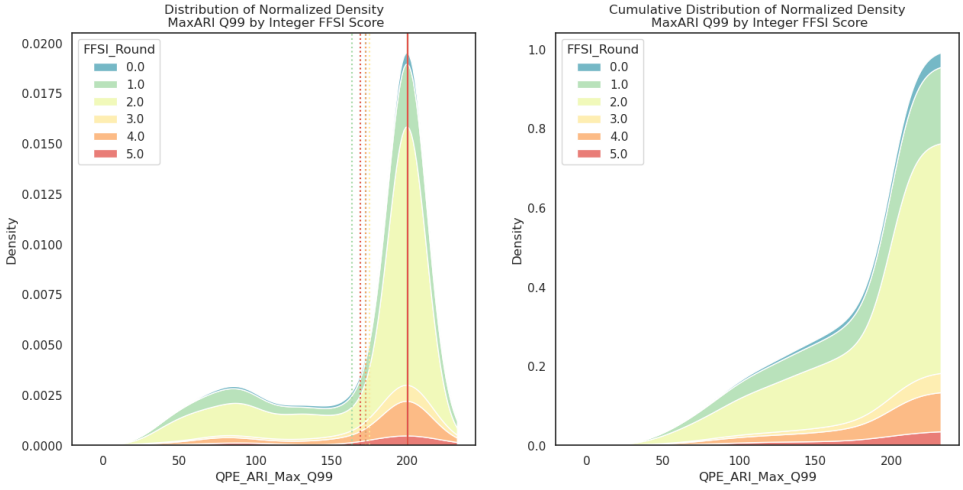

Cumulative Distribution of Normalized Density
MaxARI MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_MEAN


Distribution of Normalized Density
MaxARI Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q90

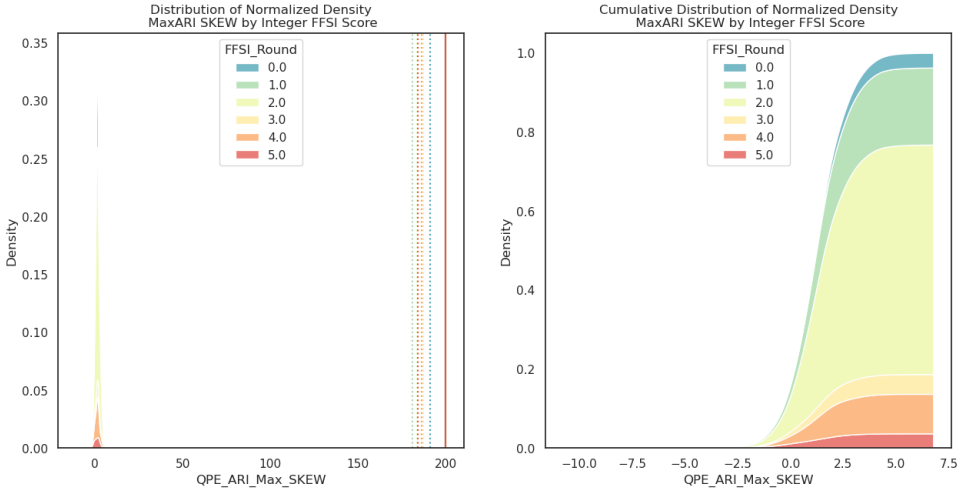

Cumulative Distribution of Normalized Density
MaxARI Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q90


Distribution of Normalized Density
MaxARI Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q95

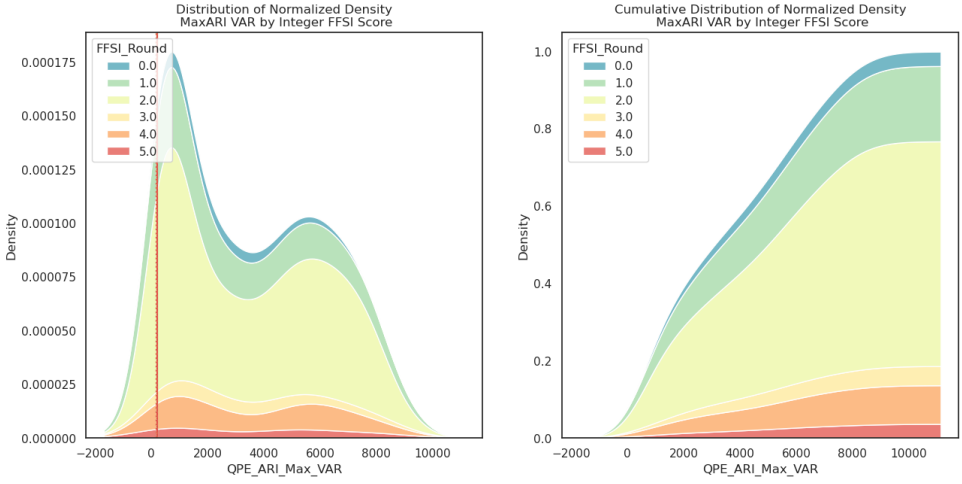

Cumulative Distribution of Normalized Density
MaxARI Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q95


Distribution of Normalized Density
MaxARI Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q99

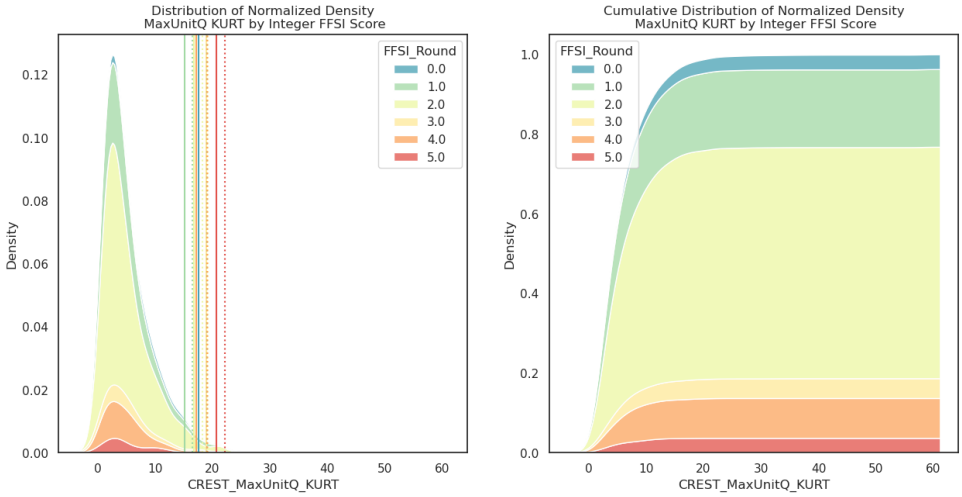
Cumulative Distribution of Normalized Density
MaxARI Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_Q99


Distribution of Normalized Density
MaxARI SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_SKEW

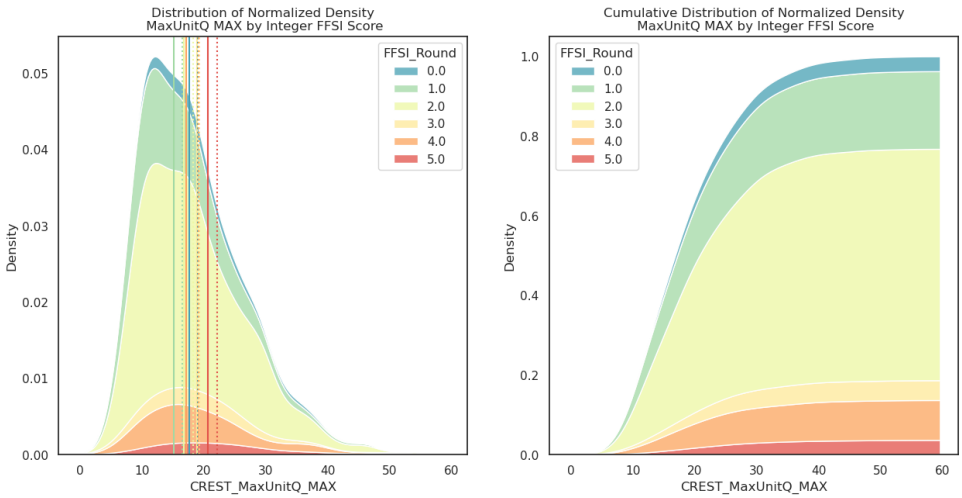
Cumulative Distribution of Normalized Density
MaxARI SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_SKEW


Distribution of Normalized Density
MaxARI VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_VAR

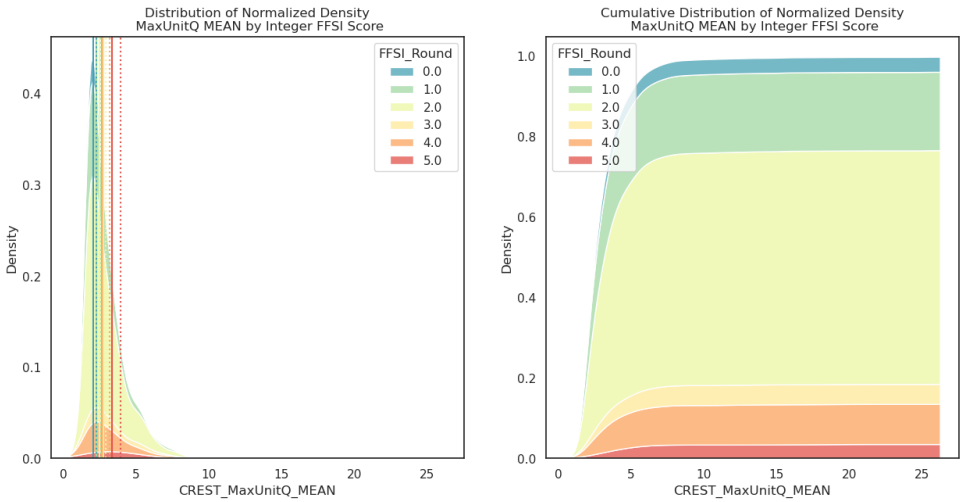
Cumulative Distribution of Normalized Density
MaxARI VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
QPE_ARI_Max_VAR


Distribution of Normalized Density
MaxUnitQ KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_KURT

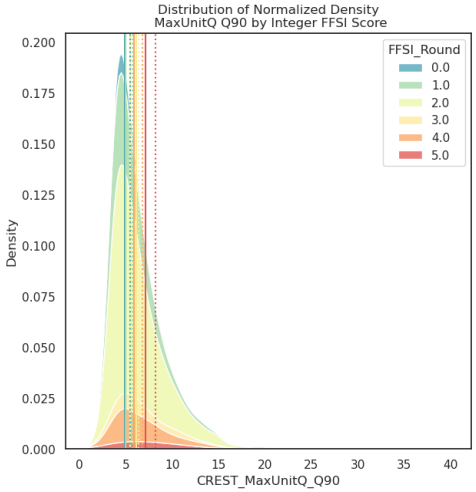
Cumulative Distribution of Normalized Density
MaxUnitQ KURT by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_KURT

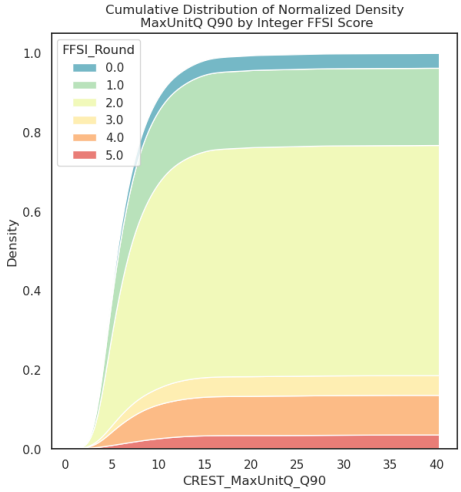

Distribution of Normalized Density
MaxUnitQ MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MAX

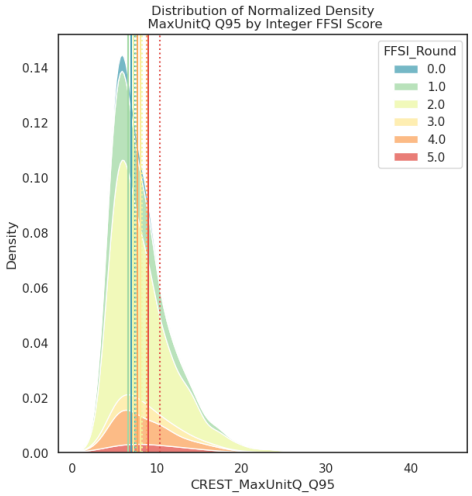
Cumulative Distribution of Normalized Density
MaxUnitQ MAX by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MAX

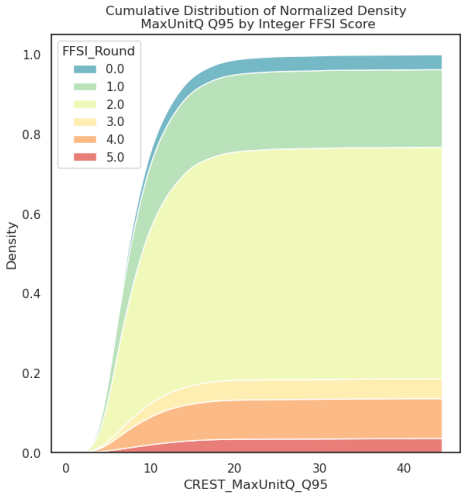

Distribution of Normalized Density
MaxUnitQ MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MEAN

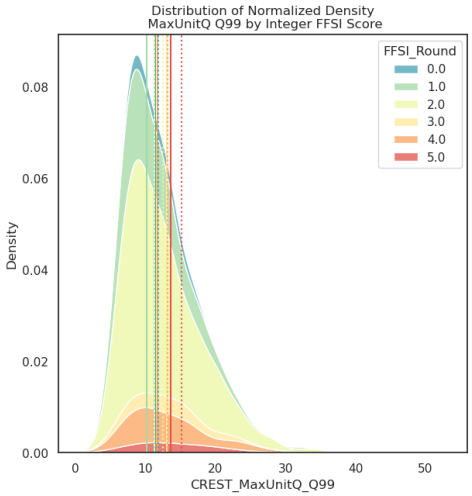
Cumulative Distribution of Normalized Density
MaxUnitQ MEAN by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_MEAN

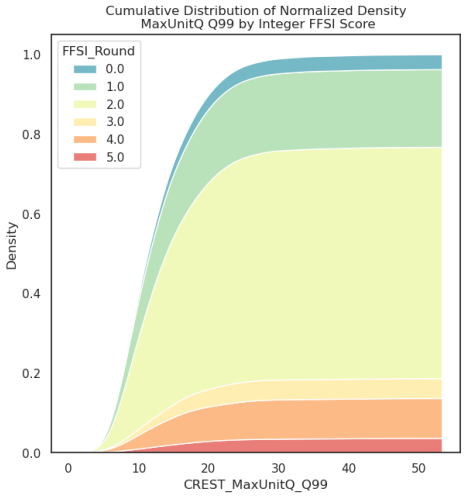

Distribution of Normalized Density
MaxUnitQ Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q90

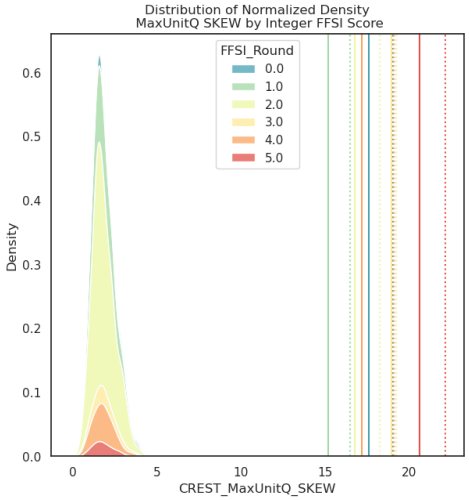

Cumulative Distribution of Normalized Density
MaxUnitQ Q90 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q90

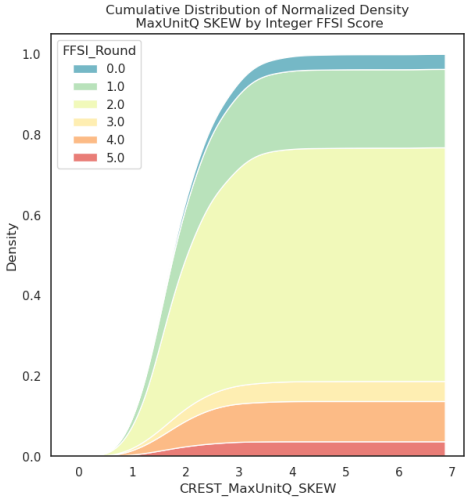

Distribution of Normalized Density
MaxUnitQ Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q95

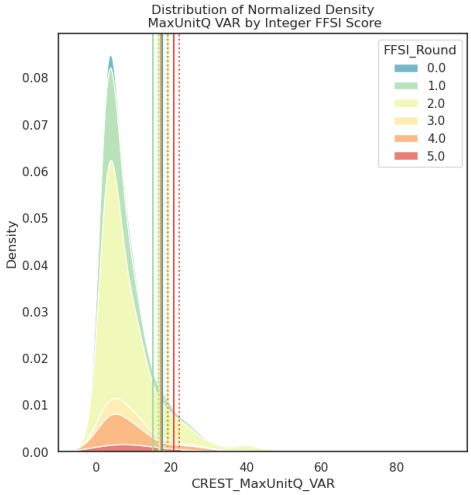

Cumulative Distribution of Normalized Density
MaxUnitQ Q95 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q95

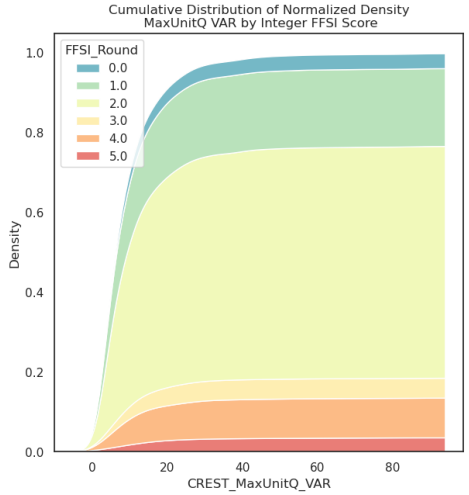

Distribution of Normalized Density
MaxUnitQ Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q99


Cumulative Distribution of Normalized Density
MaxUnitQ Q99 by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_Q99


Distribution of Normalized Density
MaxUnitQ SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_SKEW


Cumulative Distribution of Normalized Density
MaxUnitQ SKEW by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_SKEW


Distribution of Normalized Density
MaxUnitQ VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_VAR


Cumulative Distribution of Normalized Density
MaxUnitQ VAR by Integer FFSI Score
FFSI_Round
0.0
1.0
2.0
3.0
4.0
5.0
Density
CREST_MaxUnitQ_VAR

Code Appendix

UnitQ Extractor

Listing C1: UnitQ Extractor: Matlab script that performs the extraction of UnitQ values for a given set of LSR reports.

```
product_name = [{'maxunitq.'}, {'30U.ARI.'}, {'01H.ARI.'}, {'03H.ARI.'}, {'06H.ARI.'}, {'12H.ARI.'}, {'24H.ARI.'},
↪  {'MAX.ARI.'}, {'01H.RAT.'}, {'03H.RAT.'}, {'06H.RAT.'}, {'MAX.RAT.'}, {'maxunitq.'}, {'maxunitq.'}];
all_product_folder = [{'maxunitq/'}, {'preciprp_30m/'}, {'preciprp_1h/'}, {'preciprp_3h/'}, {'preciprp_6h/'},
↪  {'preciprp_12h/'}, {'preciprp_24h/'}, {'preciprp_max/'}, {'ratio_1h/'}, {'ratio_3h/'}, {'ratio_6h/'}, {'ratio_max/'},
↪  {'maxunitq_sac/'},{'maxunitq_hp/'}];
%product_res = [1/(24*6), 1/(24*30), 1/(24*30), 1/(24*30), 1/(24*30), 1/(24*30), 1/(24*30), 1/(24*30),
↪  1/(24*30), 1/(24*30), 1/(24*6), 1/(24*6)];
% Use 4-min time step for QPE-based products due to memory limitations
%product_res = [1/(24*6), 1/(24*15), 1/(24*15), 1/(24*15), 1/(24*15), 1/(24*15), 1/(24*15), 1/(24*15),
↪  1/(24*15), 1/(24*15), 1/(24*6), 1/(24*6)];
% Force all products to use 10-min frequency due to memory/computational limitations
product_res = zeros(1,numel(product_name)) + 1/(24*6);

prod_i = 1;

root_product_folder = './test_data/';

product = product_name{prod_i};
outFile_product = product;
if (prod_i == 13)
 outFile_product = ['sac_', product];
end

if (prod_i == 14)
        outFile_product = ['hp_', product];
end

product_folder = all_product_folder{prod_i};

mapinfo = geotiffinfo('./test_data/flash_conus_mask1km.tif');

mask = imread('./test_data/flash_conus_mask1km.tif');
conus_pixels = find(mask==1);
total_pixels = numel(conus_pixels);

%Period configuration
tstep_mins = product_res(prod_i)*24*60;
tstep_hrs = tstep_mins/60;
tsteps_per_hrs = 60/tstep_mins;
buffer_prev_in_hrs = 6;

%lsr_data = readtable('./test_data/observations/LSR/geoloc_only_flashfloods_lsr_201804010000_202207010000.csv');
lsr_data = readtable('./test_data/observations/mPing/geoloc_only_flashfloods_mping_201804010000_202207010000.csv');
% VALID,VALID2,LAT,LON
[lsr.year,lsr.month,lsr.day,lsr.hour,lsr.minute,~] = datevec(lsr_data.VALID2);

% Check if necessary with all input tables
lsr_dates = datenum(lsr.year,lsr.month,lsr.day,lsr.hour,lsr.minute,0);

% Only consider LSRs within CONUS
%goodIDX = find(lsr_data.LON > mapinfo.BoundingBox(1) & lsr_data.LON < mapinfo.BoundingBox(2) & lsr_data.LAT >
↪  mapinfo.BoundingBox(3) & lsr_data.LAT < mapinfo.BoundingBox(4));
%goodIDX2 = find(lsr_data.LON > mapinfo.BoundingBox(1) & lsr_data.LON < mapinfo.BoundingBox(2) & lsr_data.LAT >
↪  mapinfo.BoundingBox(3) & lsr_data.LAT < mapinfo.BoundingBox(4) & lsr_dates < datenum('01-Jun-2021'));
%goodIDX3 = find(lsr_data.LON > mapinfo.BoundingBox(1) & lsr_data.LON < mapinfo.BoundingBox(2) & lsr_data.LAT >
↪  mapinfo.BoundingBox(3) & lsr_data.LAT < mapinfo.BoundingBox(4) & lsr_dates < datenum('01-Jun-2020'));
```

```
goodIDX = find(lsr_data.LON > mapinfo.BoundingBox(1) & lsr_data.LON < mapinfo.BoundingBox(2) & lsr_data.LAT >
↪  mapinfo.BoundingBox(3) & lsr_data.LAT < mapinfo.BoundingBox(4) & lsr_dates < datenum('15-Mar-2022 12:00:00'));

fprintf('%f reports of FF\n', numel(goodIDX));
%fprintf('%f reports of FF until June 2021\n', numel(goodIDX2));
%fprintf('%f reports of FF until June 2020\n', numel(goodIDX3));

% Subset of LSR data/info
lsr_LON = lsr_data.LON(goodIDX);
lsr_LAT = lsr_data.LAT(goodIDX);
[sorted_lsr_dates,sorted_IX] = sort(lsr_dates(goodIDX));
lsr_diff_in_hrs = diff(sorted_lsr_dates).*24;
lsr_LON = lsr_LON(sorted_IX); lsr_LAT = lsr_LAT(sorted_IX);

% Round LSRs dates to the closest product resolution
[all_year,all_month,all_day,all_hour,all_minute,~] = datevec(sorted_lsr_dates);

rounded_sorted_lsr_dates = datenum(all_year,all_month,all_day,all_hour,round(all_minute/tstep_mins)*tstep_mins,0);

n_good_lsrs = numel(goodIDX);

% FLASH values at pixel with no LSR
nolsr_flash.val = zeros(n_good_lsrs,1);
nolsr_flash.val_mean_rising_slope = zeros(n_good_lsrs,1);
nolsr_flash.val_t = zeros(n_good_lsrs,1);
nolsr_flash.pix_mat_idx = zeros(n_good_lsrs,1);
% FLASH values at exact pixel with LSR
lsr_flash.val = zeros(n_good_lsrs,1);
lsr_flash.val_mean_rising_slope = zeros(n_good_lsrs,1);
lsr_flash.val_t = zeros(n_good_lsrs,1);
[lsr_rows,lsr_cols] = latlon2pix(mapinfo.RefMatrix, lsr_LAT,lsr_LON);
lsr_rows = round(lsr_rows); lsr_cols = round(lsr_cols);
lsr_flash.pix_mat_idx = sub2ind([mapinfo.Height mapinfo.Width], lsr_rows, lsr_cols);
% FLASH values at pixel in the 1-pixel neighborhood of LSR
lsr_flash.max_val_neigh_t = zeros(n_good_lsrs,1);
lsr_flash.max_val_neigh_mean_rising_slope = zeros(n_good_lsrs,1);
lsr_flash.max_val_neigh = zeros(n_good_lsrs,1);
lsr_flash.neigh_pix_mat_idx = zeros(n_good_lsrs,1);
% LSR row entry in input file
lsr_file.row = goodIDX; % Depends on the file
lsr_file.name = 'geoloc_only_flashfloods_mping_201804010000_202207010000.csv';

% Create array to check for LSRs processing progress
processed_lsrs_counters = 0;
target_number_of_prod_steps = buffer_prev_in_hrs*tsteps_per_hrs;

% Pre-allocate grids with output variables
flash_conus_pixels_timewin = zeros(total_pixels,tsteps_per_hrs*buffer_prev_in_hrs);
max_flash_grid = zeros([mapinfo.Height mapinfo.Width]);
t_of_max_flash_grid = zeros([mapinfo.Height mapinfo.Width]);
positive_rate_of_change = zeros([mapinfo.Height mapinfo.Width]);

% Loop through main period of data
cont_t = 0; accum_gaps = 0;
break_points = find(lsr_diff_in_hrs > buffer_prev_in_hrs);
% Target dates list
all_target_dates = [rounded_sorted_lsr_dates(1) rounded_sorted_lsr_dates(break_points+1)' numel(rounded_sorted_lsr_dates)];
next_dates = [rounded_sorted_lsr_dates(break_points+1)' numel(rounded_sorted_lsr_dates) NaN];

try

for product_date = all_target_dates(1:end-1)
    %Counter
    cont_t = cont_t + 1;

    % Build time series going back "buffer_prev_in_hrs" hours from target date (LSR date)
    fprintf('Current LOOP is: %s - %s. Next date is %s\n',
    ↪  datestr(product_date-buffer_prev_in_hrs/24),datestr(sorted_lsr_dates(break_points(cont_t))),
    ↪  datestr(next_dates(cont_t)));
    % Compute the first window
    window_step = 0;
    this_period = product_date-buffer_prev_in_hrs/24:product_res(prod_i):product_date-product_res(prod_i);
    fprintf('Current analysis window is: %s - %s\n', datestr(this_period(1)),datestr(this_period(end)));
    for start_t = this_period
        window_step = window_step + 1;
        %Try reading in new file
        try
            fprintf('Reading %s\n', [product_folder, product, datestr(start_t, 'yyyymmdd.HHMM'), '00.tif']);
            c_file = imread([root_product_folder, product_folder, product, datestr(start_t, 'yyyymmdd.HHMM'), '00.tif']);
            c_file(c_file<0) = 0;
            flash_conus_pixels_timewin(:,window_step) = c_file(conus_pixels);
        catch ME %If reading file fails, then...
            %Go to next product time step
            ME.message
            flash_conus_pixels_timewin(:,window_step) = nan(total_pixels,1);
            accum_gaps = accum_gaps + 1;
            continue;
        end
        %END Try reading in new file
    end
```

```
    fprintf('Initial array has been built.\n');

    % Continue rolling the window
    this_period = product_date:product_res(prod_i):rounded_sorted_lsr_dates(break_points(cont_t));
    fprintf('Current analysis window is: %s - %s\n', datestr(this_period(1)),datestr(this_period(end)));
    for start_t = this_period
        % Make space for next input file
        %tic;
        %flash_conus_pixels_timewin(:,1:end-1) = flash_conus_pixels_timewin(:,2:end);
        flash_conus_pixels_timewin(:,1) = [];

        %Try reading in new file
        try
            fprintf('Reading %s\n', [product_folder, product, datestr(start_t, 'yyyymmdd.HHMM'), '00.tif']);
     c_file = imread([root_product_folder, product_folder, product, datestr(start_t, 'yyyymmdd.HHMM'), '00.tif']);
     c_file(c_file<0) = 0;
            %flash_conus_pixels_timewin(:,end) = c_file(conus_pixels);
            flash_conus_pixels_timewin = [flash_conus_pixels_timewin, c_file(conus_pixels)];
        catch ME %If reading file fails, then...
     %Go to next product time step
            ME.message
            %flash_conus_pixels_timewin(:,end) = nan(total_pixels,1);
            flash_conus_pixels_timewin = [flash_conus_pixels_timewin, nan(total_pixels,1)];
            accum_gaps = accum_gaps + 1;
        continue;
        end
        %END Try reading in new file

        % Find all LSRs that are active within the analysis window
        active_lsr_idx = find(rounded_sorted_lsr_dates == start_t);
 n_active_lsr_idx = numel(active_lsr_idx);

        fprintf('Working on %f active LSRs after adding %s\n', n_active_lsr_idx, [product_folder, product, datestr(start_t,
        ↪  'yyyymmdd.HHMM'), '00.tif']);

        if (n_active_lsr_idx == 0)
            %toc
            continue;
        end

        % Find the maximum value from all stored grids
        [max_flash_vector, max_IDX] = max(flash_conus_pixels_timewin,[],2);
        flash_val_delta = diff(flash_conus_pixels_timewin,[],2);

        % Create grids
        this_subset_period = start_t-(buffer_prev_in_hrs/24)+product_res(prod_i):product_res(prod_i):start_t;
        t_of_max_flash_grid(conus_pixels) = this_subset_period(max_IDX);
        max_flash_grid(conus_pixels) = max_flash_vector;
        sumPositiveChange = nansum(flash_val_delta.*(flash_val_delta>0),2);
        TotalPositiveCols = sum(flash_val_delta>0,2);
        positive_rate_of_change(conus_pixels) = sumPositiveChange./(TotalPositiveCols.*tstep_hrs);

 % FLASH values at pixel with no LSR
        % Find all non-zero FLASH pixels
        all_nonlsr_pixels = find(max_flash_grid > 0);
        all_nonlsr_pixels(ismember(all_nonlsr_pixels,lsr_flash.pix_mat_idx(active_lsr_idx))) = [];
        % Randomly select non LSR values as the same number of active LSRs
        random_selection_idx = randperm(numel(all_nonlsr_pixels));

        nolsr_flash.val(active_lsr_idx) = max_flash_grid(all_nonlsr_pixels(random_selection_idx(1:n_active_lsr_idx)));
        nolsr_flash.val_t(active_lsr_idx) = t_of_max_flash_grid(all_nonlsr_pixels(random_selection_idx(1:n_active_lsr_idx)));
        nolsr_flash.pix_mat_idx(active_lsr_idx) = all_nonlsr_pixels(random_selection_idx(1:n_active_lsr_idx));
        nolsr_flash.val_mean_rising_slope(active_lsr_idx) = positive_rate_of_change(nolsr_flash.pix_mat_idx(active_lsr_idx));

        % FLASH values at exact pixel with LSR
        lsr_flash.val(active_lsr_idx) = max_flash_grid(lsr_flash.pix_mat_idx(active_lsr_idx));
        lsr_flash.val_mean_rising_slope(active_lsr_idx) = positive_rate_of_change(lsr_flash.pix_mat_idx(active_lsr_idx));
        lsr_flash.val_t(active_lsr_idx) = t_of_max_flash_grid(lsr_flash.pix_mat_idx(active_lsr_idx));

        % FLASH values at pixel in the 1-pixel neighborhood of LSR
        lsr_flash.max_val_neigh_t(active_lsr_idx) = lsr_flash.val_t(active_lsr_idx);
        lsr_flash.max_val_neigh(active_lsr_idx) = lsr_flash.val(active_lsr_idx);
        lsr_flash.neigh_pix_mat_idx(active_lsr_idx) = lsr_flash.pix_mat_idx(active_lsr_idx);
        lsr_flash.max_val_neigh_mean_rising_slope(active_lsr_idx) = lsr_flash.val_mean_rising_slope(active_lsr_idx);

        for i = -1:1
            for j = -1:1
                neighbor_idx = sub2ind([mapinfo.Height mapinfo.Width],lsr_rows(active_lsr_idx)+i,lsr_cols(active_lsr_idx)+j);
                entries_tobe_changed = find(max_flash_grid(neighbor_idx) - lsr_flash.max_val_neigh_t(active_lsr_idx) > 0);

                lsr_flash.max_val_neigh_t(active_lsr_idx(entries_tobe_changed)) =
                ↪  t_of_max_flash_grid(neighbor_idx(entries_tobe_changed));
                lsr_flash.max_val_neigh_mean_rising_slope(active_lsr_idx(entries_tobe_changed)) =
                ↪  positive_rate_of_change(neighbor_idx(entries_tobe_changed));
                lsr_flash.max_val_neigh(active_lsr_idx(entries_tobe_changed)) =
                ↪  max_flash_grid(neighbor_idx(entries_tobe_changed));
                lsr_flash.neigh_pix_mat_idx(active_lsr_idx(entries_tobe_changed)) = neighbor_idx(entries_tobe_changed);
            end
        end
        %toc
    end
end
```

```
% Save dataset
save(['LSR_extraction_outputs_for_', outFile_product, 'mat'], 'lsr_flash', 'nolsr_flash', 'lsr_file', 'all_target_dates',
↪  'rounded_sorted_lsr_dates', 'sorted_lsr_dates', 'sorted_IX');

catch MEOUT
    % Save whatever progress to this point
    save(['LSR_extraction_outputs_for_', outFile_product, 'mat'], 'lsr_flash', 'nolsr_flash', 'lsr_file', 'all_target_dates',
    ↪  'rounded_sorted_lsr_dates', 'sorted_lsr_dates', 'sorted_IX');
end

exit;
```

FLASH Moment Extractor

Listing C2: Moment Extractor: Main file which wraps all functionality and handles the extraction of FLASH product moments for a given set of LSRs.

```
#!/usr/bin/env python3
""" Moment Extractor
"""
from FLASH_info import PRODUCTS
import fileio_common as fileio
from sys import exit
from datetime import datetime
from spacetime_cube import *
from scipy.stats import skew, kurtosis  # , moment
from numpy import mean, var, std, nonzero, quantile
from numpy import max as maximum
import os

# Constant which holds the path to the FLASH data folder
# Flash data folder on Hydros Server yhone (10.197.10.250)
#FLASH_DATA_FOLDER = "/var/www/html/new/flash_web"
#Server2017 FLASH data folder
FLASH_DATA_FOLDER = "/flash_web"
# Local test data folder
#FLASH_DATA_FOLDER = "./data/test/flash_web"

# Maximum order of statistical moments to be extracted from the data:
#   1: Mean
#   2: Variance
#   3: Skewness
#   4: Kurtosis
MAX_MOMENT = 4

# Time delta IN HOURS, defines the temporal search window for extracting data
# from the product files, based on an observation report's (e.g. LSR, mPing)
# time:
#   time_window = (report_time - DELTA_t, report_time + DELTA_t)
DELTA_t = 3

# Spatial delta IN GRID CELLS, defines the spatial search window radius for
# extracting data from the product files, based on an observation report's
# (eg. LSR, mPing) location given by latitude and longitude coordinates:
#   space_window = (report_lat, report_lon) * DELTA_r ^ 2
DELTA_r = 4

# Target products IDs for extraction, intersection, and moment calculations.
# These IDs are defined in the PRODUCTS dictionary, in the file FLASH_info.py.
# Each one of the IDs has associated information on where the product is stored
# within the folder structure of the FLASH_DATA_FOLDER (based on the Hydros
# server subdirectory structure for yhone:/var/www/html/new/flash_web), its
# common filename prefix, and product update frequencies.
TARGET_PRODUCTS = ["CREST_MaxUnitQ", "QPE_ARI_Max"]

# Path to a CSV file with flash flood reports, for which product moments will be
# extracted.
#REPORTS_CSV = "./data/observations/LSR/flashfloods_lsr_201804010000_202207010000.csv"
# Local test reports file
#REPORTS_CSV = "./data/test/observations/LSR/lsr_202207271200_202207301200_impacts.csv"
REPORTS_CSV = "./data/observations/LSR/ibw_classified_lsr.csv"

# Batch size for batch-processing the provided reports. Every time a batch is
# processed, an intermediate results file will be written out, which will enable
# the Moment Extractor to resume its operation without having to repeat all
# progress if it is interrupted.
BATCH_SIZE = 25

# Maximum number of batches to process (use to limit the number of processed
```

```
# reports during testing). MAX_BATCHES = 0 means NO LIMIT!
MAX_BATCHES = 0

# Path to the folder where results are to be written out to
RESULTS_OUTPUT = "./results/"

# Path to a GeoTIFF of the overall DOMAIN to be used as reference
DOMAIN = "./data/mask/flash_conus_mask1km.tif"

# Assume a steady-state when missing input product files?
ASSUME_STEADY_STATE = True

# System frequency for all cubes
SYSTEM_FREQ = 10

# Calculate intersection moments?
INTERSECTION = True


def main():
    """Main function and point of entry for the Moments Extractor.

    Main function, which will be the entry point that will be executed, when
    this program is run as a script from the command line.
    """
    # Extract the domain's shape from the reference GeoTIFF
    domain_metadata = read_geotiff(DOMAIN)[1]
    domain_shape = (domain_metadata['height'], domain_metadata['width'])

    # Read the LSR reports from a standard CSV file
    #lsr_reports = fileio.read_standard_lsrs(REPORTS_CSV, no_category=True)
    lsr_reports = fileio.read_ibw_lsrs(REPORTS_CSV)

    # Keep track of total number of reports
    total_lsr_reports = len(lsr_reports)

    # Define a Unique Identifier for the LSR file that is being processed
    lsr_uuid = fileio.hash_filename(REPORTS_CSV)

    # Define batches for batch processing the LSRs, so that it is easier to
    # restart the process, in case of interruptions or failure
    batches = fileio.define_batches(lsr_reports.shape[0], BATCH_SIZE)

    # Check for pre-existing batch result JSON file, and if found, update the
    # batches dictionary with "processed": True, for the matching batches
    batches = fileio.match_batch_results(file_uuid=lsr_uuid,
                                         batches=batches,
                                         results_path=RESULTS_OUTPUT)

    # Process the LSRs, extracting their locations and timestamps to match
    # existing product files.

    # Hold the total number of batches for future reference
    num_batches = len(batches)

    # Print out which file, how many reports, and how many batches will be
    # processed
    # if verbose:
    print(f"Processing LSR file {lsr_uuid}: {lsr_reports.shape[0]} reports / "
          f"{num_batches} batches")

    # Variables to keep track of the total number of processed reports, as well
    # as the number of processed and skipped batches
    total_processed = 0
    batches_skipped = 0
    batches_processed = 0


    # For each bathch in batches
    for batch_id in batches:
        # if verbose:
        print(f"Batch ID: {batch_id}")

        # IF the current batch has been processed
        if batches[batch_id]["processed"]:
            # Do nothing, and move to the next batch
            # Keep track of how many batches were skipped
            batches_skipped += 1
            # if verbose:
            print(f"WARNING: Skipping batch {batch_id + 1}/{num_batches}"
                  f" - already processed")
            # Since processed nothing, set processed_lsrs to None
            processed_lsrs = None

        else:
            # Get the current batch start and end indices for the LSR DataFrame
            start_idx, end_idx = batches[batch_id]["indices"]

            # if verbose:
            print(f"Indices: ({start_idx}, {end_idx})")

            # Subset the LSR reports to only select the reports for this batch.
```

```
# Notice that the end index is incremented by 1, since the top
# index is always excluded by definition.
current_lsrs = lsr_reports.iloc[start_idx : end_idx + 1]

processed_lsrs = {}

# For each LST report in the current batch
# Process the current batch of LSRs
for index, report in current_lsrs.iterrows():
    print(f"Index: {index}")
    report_cubes = {}
    # For each of the target products we want to use
    for product in TARGET_PRODUCTS:
        try:
            # Get the current LSR's latitude and longitude
            lat = float(report['lat'])
            lon = float(report['lon'])

            # Obtain the grid cell that corresponds to the event's
            # latiitude and longitude location
            event_grid = get_event_grid(DOMAIN, lat, lon)
            #print(f"{lat}, {lon} => {event_grid}")


            # Obtain the event window indices for cropping the
            # product data, and make smaller data cubes
            window_inds = get_event_window_indices(event_grid,
                                                   DELTA_r,
                                                   flatten=False)
            #print(f"Window: {window_inds}")
            #print(f"WindowSize: {len(window_inds)}")


            # Obtain the event's window mask, to crop the domain
            # only to the area of interest defined by DELTA_r
            window_mask = get_event_window_mask(domain_shape,
                                                window_inds,
                                                flatten=False)
            #print(f"Mask: {window_mask}")
            #print(f"MaskShape: {window_mask.shape}")
            #print(f"MaskNon0: {len(window_mask[window_mask>0])}")


            # Extract the current product's metadata
            product_data = PRODUCTS[product]
            # Extract the current product's subdirectory
            data_subdir = product_data['subdir']
            # Extract the current product's file naming prefix
            prefix = product_data['prefix']
            # Extract the current product's update frequency
            frequency = int(product_data['freq'])

            # If a SYSTEM-wide frequency is provided, force all
            # products to be sampled to this frequency
            if SYSTEM_FREQ > 0:
                frequency = SYSTEM_FREQ

            # Convert the current LSR timestamp to FLASH format
            #current_timestamp = parse_timestamp(report['valid2'],
            #                                    frequency,
            #                                    input_format='%Y/%m/%d %H:%M')
            current_timestamp = parse_timestamp(report['time'],
                                                frequency,
                                                input_format='%Y-%m-%d %H:%M:00')
            print(f"{product} Timestamp: {current_timestamp}")


            # Generate the file list needed to create a cube for the
            # current LSR report, using the requested DELTA_t.
            # NOTE: Since we are making sure that the file list
            # contains ONLY paths to files that DO EXIST, inexistent
            # files will have a corresponding None, instead of a
            # path in the list.
            cube_file_list = \
                fileio.generate_cube_file_list(reference_timestamp=current_timestamp,
                                               data_dir=FLASH_DATA_FOLDER,
                                               data_subdir=data_subdir,
                                               product_prefix=prefix,
                                               delta_hours=DELTA_t,
                                               product_freq=frequency,
                                               check_existing=True)
            #print(f"FileList:{cube_file_list}")

            # Handle the cases where any of the requested files does
            # not exist. Assume a steady state for missing product
            # files, instead of zeros (default behavior of
            # build_cube())
            if ASSUME_STEADY_STATE:
                cube_file_list = steady_state(cube_file_list)


            #print(f"FileList:{cube_file_list}")
```

```
                    window_shape = ((2 * DELTA_r) + 1, (2 * DELTA_r) + 1)
                    # Build the cube for the current event's product
                    prod_cube = build_cube(cube_file_list,
                                           domain_shape,
                                           window_mask,
                                           verbose=False)

                    #print(f"CUBE:\n{prod_cube}")
                    #print(f"CUBENon0:{len(prod_cube[prod_cube>0])}")
                    #print(f"SHAPE:\n{prod_cube.shape}")

                    #sys_shape = system_cube_shape(DELTA_t, DELTA_r, SYSTEM_FREQ)
                    #print(f"SYSTEM CUBE SHAPE: {sys_shape}")

                    #if prod_cube.shape != sys_shape:
                    #    prod_cube = shrink_cube(prod_cube, sys_shape)

                    report_cubes[product] = prod_cube

                except Exception as e:
                    print(f"ERROR: Could not build data cube for product {product}!"
                          f"Exception Caught: {e}")
                    exit(1)

            #print(f"CUBES: {report_cubes}")

            report_moments = {}

            for product_key in report_cubes:
                cube_moments = calculate_moments(report_cubes[product_key],
                                                 MAX_MOMENT)
                #print(f"{product_key} moments: {cube_moments}")
                report_moments[product_key] = cube_moments

            if INTERSECTION:
                mask1 = nonzero(report_cubes[TARGET_PRODUCTS[1]])
                intersection1 = report_cubes[TARGET_PRODUCTS[0]][mask1]
                intersection_moments1 = calculate_moments(intersection1,
                                                          MAX_MOMENT)
                #print(f"{TARGET_PRODUCTS[0]}_Intersection moments: {intersection_moments1}")
                report_moments[f"{TARGET_PRODUCTS[0]}_intersection"] = intersection_moments1

                mask2 = nonzero(report_cubes[TARGET_PRODUCTS[0]])
                intersection2 = report_cubes[TARGET_PRODUCTS[1]][mask2]
                intersection_moments2 = calculate_moments(intersection2,
                                                          MAX_MOMENT)
                #print(f"{TARGET_PRODUCTS[1]}_Intersection moments: {intersection_moments2}")
                report_moments[f"{TARGET_PRODUCTS[1]}_intersection"] = intersection_moments2

            print(f"Moments: {report_moments}")

            processed_lsrs[index] = report_moments

        # Write the current batch's result as a JSON file, identified by
        # the lsr_uuid string and the current batch's ID, in the desired
        # output results folder
        batch_filename = f"{lsr_uuid}_{batch_id}.json"
        batch_path = os.path.join(RESULTS_OUTPUT, batch_filename)
        fileio.write_json_results(processed_lsrs, batch_path)

        # if verbose:
        print(f"Wrote partial results file: {batch_path}")

        # Mark batch as processed:
        batches[batch_id]["processed"] = True

        # Keep track of how many batches were processed
        batches_processed += 1

        # if verbose:
        print(f"Processed {len(processed_lsrs)} reports for "
              f"batch {batch_id + 1}/{num_batches}")

    # Break after MAX_BATCHES, if MAX_BATCHES > 0
    if MAX_BATCHES and batch_id == MAX_BATCHES:
        # if verbose:
        print(f"WARNING: MAX_BATCHES of {MAX_BATCHES} reached! HALTING!\n")
        break

# Keep track of how many total LSRs were processed
if processed_lsrs:
    total_processed += len(processed_lsrs)
else:
    total_processed += 0

# Notify the user processing is done, and provide some counts on results
print(f"DONE!\n\t"
      f"Reports file: {REPORTS_CSV}\n\t"
      f"UUID: {lsr_uuid}\n\t"
      f"processed {batches_processed}/{num_batches} batches\n\t"
      f"skipped {batches_skipped}/{num_batches} batches \n\t"
```

```
      f"processed {total_processed} LSRs")

# Consolidate processed JSON results into a single CSV file

# Check if there are any JSON files in the current output path
result_files = fileio.get_files_in_dir(RESULTS_OUTPUT, extension=".json")
# Make sure that the JSON files found match the current UUID
result_files = sorted(list(filter(lambda x: lsr_uuid in x, result_files)))

print(result_files)

# If the list of result files in the output folder, with the requested uuid
# is empty, notify that no batch results were found
if not result_files:
    print("ERROR: No batch result files were found for current uuid!")

# Else if there are result files in the output folder
else:
    # Create a copy of the original LSR DataFrame
    output_lsrs = lsr_reports.copy()
    #output_lsrs.index = output_lsrs.remark
    # Create the new columns for the new results data in the DataFrame
    for product in TARGET_PRODUCTS:
        output_lsrs[f"{product}_MEAN"] = None
        output_lsrs[f"{product}_VAR"] = None
        output_lsrs[f"{product}_SKEW"] = None
        output_lsrs[f"{product}_KURT"] = None
        output_lsrs[f"{product}_Q90"] = None
        output_lsrs[f"{product}_Q95"] = None
        output_lsrs[f"{product}_Q99"] = None
        output_lsrs[f"{product}_MAX"] = None

        if INTERSECTION:
            output_lsrs[f"{product}_intersection_MEAN"] = None
            output_lsrs[f"{product}_intersection_VAR"] = None
            output_lsrs[f"{product}_intersection_SKEW"] = None
            output_lsrs[f"{product}_intersection_KURT"] = None
            output_lsrs[f"{product}_intersection_Q90"] = None
            output_lsrs[f"{product}_intersection_Q95"] = None
            output_lsrs[f"{product}_intersection_Q99"] = None
            output_lsrs[f"{product}_intersection_MAX"] = None


    # For each of these files
    for json_file in result_files:
        # Read the results JSON file
        batch_results = fileio.read_json_results(json_file)
        # Get the current batch ID from the batch results file
        batch_id = json_file.split("/")[-1].split('_')[-1].split('.')[0]
        # Get the LSR dataframe ID corresponding to the beginning of the
        # current batch
        start_index = int(batch_id) * BATCH_SIZE
        # Iterate over each result, and add the data to the corresponding
        # columns in the output LSR dataframe, for the corresponding LSR
        for batch_index in batch_results:
            #idx = start_index + int(batch_index)
            idx = int(batch_index)
            if idx >= total_lsr_reports:
                break
            #print(f"Write index: {idx}")
            for moment_index in batch_results[batch_index]:
                cur_mean = batch_results[batch_index][moment_index]["mean"]
                cur_var = batch_results[batch_index][moment_index]["var"]
                cur_skew = batch_results[batch_index][moment_index]["skew"]
                cur_kurt = batch_results[batch_index][moment_index]["kurt"]
                cur_q90 = batch_results[batch_index][moment_index]["q90"]
                cur_q95 = batch_results[batch_index][moment_index]["q95"]
                cur_q99 = batch_results[batch_index][moment_index]["q99"]
                cur_max = batch_results[batch_index][moment_index]["max"]

                try:
                    output_lsrs.loc[idx][f"{moment_index}_MEAN"] = cur_mean
                    output_lsrs.loc[idx][f"{moment_index}_VAR"] = cur_var
                    output_lsrs.loc[idx][f"{moment_index}_SKEW"] = cur_skew
                    output_lsrs.loc[idx][f"{moment_index}_KURT"] = cur_kurt
                    output_lsrs.loc[idx][f"{moment_index}_Q90"] = cur_q90
                    output_lsrs.loc[idx][f"{moment_index}_Q95"] = cur_q95
                    output_lsrs.loc[idx][f"{moment_index}_Q99"] = cur_q99
                    output_lsrs.loc[idx][f"{moment_index}_MAX"] = cur_max
                except KeyError as error:
                    print(f"WARNING: Index not found: {error}, "
                          f"for {moment_index}")

    output_lsrs.reset_index(drop=True)
    output_lsrs.to_csv(f"./results/{lsr_uuid}_dR{DELTA_r}_dT{DELTA_t}_moments.csv", index=False)

print("DONE!")
```

```
def parse_timestamp(timestamp, frequency=None,
                    #input_format='%m/%d/%y %H:%M',
                    input_format='%Y-%m-%d %H:%M:00',
                    output_format='%Y%m%d.%H%M%S'):
    # Read the input timestamp as a datetime object, with the appropriate
    # string format provided
    datetime_object = datetime.strptime(timestamp, input_format)

    # Set the seconds to 00, since all FLASH products are generated with
    # timestamps consistent with their frequency, and with 00 seconds
    datetime_object = datetime_object.replace(second=00)

    # If a frequency is provided to be matched
    if frequency is not None:
        # Determine whether to round up, or round down to the nearest frequency
        # time step
        minutes = datetime_object.minute
        excess = minutes % frequency

        # Adjust the original timestamp to match the nearest frequency time step
        new_minute = minutes - excess
        datetime_object = datetime_object.replace(minute=new_minute)

    # Return the input timestamp, with the desired output format
    return datetime_object.strftime(output_format)


def next_valid(item_list, default_val=None):
    valid = next((item for item in item_list if item is not None), default_val)
    return valid


def previous_valid(item_list, default_val=None):
    valid = next_valid(reversed(item_list), default_val)
    return valid


def steady_state(item_list, default_val=None):
    # Output list to hold return values
    ans = []
    # For each of the items in the list
    for i, item in enumerate(item_list):
        # If the item is None
        if (i == 0) and (not item):
            # Make it equal to the next existing item
            steady = next_valid(item_list[i:], default_val)
            # Append the 'steady assumption' to the output list
            ans.append(steady)
        elif (i > 0) and (not item):
            # Make it equal to the previous existing item
            steady = previous_valid(ans[:i+1], default_val)
            # Append the 'steady assumption' to the output list
            ans.append(steady)
        # If the item is not None
        else:
            # Append the existing item to the output list
            ans.append(item)

    # Return the output list of non-None items
    return ans


def calculate_moments(cube, n_moments=MAX_MOMENT):
    # Define the results dictionary for the statistical moments
    moments = {
        "mean": None,
        "var": None,
        "skew": None,
        "kurt": None,
        "q90": None,
        "q95": None,
        "q99": None,
        "max": None
    }

    # Handle the various number of moments to be calculated
    # Calculate only mean
    if n_moments == 1:
        try:
            moments["mean"] = mean(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate moments!\n{err}")
        try:
            moments["q90"] = quantile(cube, .90, axis=None)
            moments["q95"] = quantile(cube, .95, axis=None)
            moments["q99"] = quantile(cube, .99, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate quantiles!\n{err}")
        try:
            moments["max"] = maximum(cube, axis=None)
        except Exception as err:
```

```
            print(f"ERROR: Could not calculate maximum!\n{err}")

    # Calculate only mean and variance
    elif n_moments == 2:
        try:
            moments["mean"] = mean(cube, axis=None)
            moments["var"] = var(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate moments!\n{err}")

        try:
            moments["q90"] = quantile(cube, .90, axis=None)
            moments["q95"] = quantile(cube, .95, axis=None)
            moments["q99"] = quantile(cube, .99, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate quantiles!\n{err}")
        try:
            moments["max"] = maximum(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate maximum!\n{err}")

    # Calculate only mean, variance, and skewness
    elif n_moments == 3:
        try:
            moments["mean"] = mean(cube, axis=None)
            moments["var"] = var(cube, axis=None)
            moments["skew"] = skew(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate moments!\n{err}")
        try:
            moments["q90"] = quantile(cube, .90, axis=None)
            moments["q95"] = quantile(cube, .95, axis=None)
            moments["q99"] = quantile(cube, .99, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate quantiles!\n{err}")
        try:
            moments["max"] = maximum(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate maximum!\n{err}")

    # Calculate mean, variance, skewness, and kurtosis
    else:
        try:
            moments["mean"] = mean(cube, axis=None)
            moments["var"] = var(cube, axis=None)
            moments["skew"] = skew(cube, axis=None)
            moments["kurt"] = kurtosis(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate moments!\n{err}")

        try:
            moments["q90"] = quantile(cube, .90, axis=None)
            moments["q95"] = quantile(cube, .95, axis=None)
            moments["q99"] = quantile(cube, .99, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate quantiles!\n{err}")
        try:
            moments["max"] = maximum(cube, axis=None)
        except Exception as err:
            print(f"ERROR: Could not calculate maximum!\n{err}")

    # Return the calculated moments
    return moments


def system_cube_shape(delta_t, delta_r, freq):
    x_dim = (2 * delta_r) + 1
    y_dim = (2 * delta_r) + 1
    n_pixels = x_dim * y_dim
    n_slices = len([x for x in range(-int(delta_t * 60), int(delta_t * 60 + freq), freq)])

    return (n_slices, n_pixels)

def shrink_cube(cube, desired_shape):
    cube_shape = cube.shape
    shrunken_cube = empty(shape=desired_shape)
    stride = int((cube_shape[0] - 1) / (desired_shape[0] - 1))
    print(f"Stride: {stride}")
    if cube_shape != desired_shape:
       for index, row in enumerate(shrunken_cube):
           print(f"Index: {index}")
           if index == 0:
               low = (stride * index) - 1
           else:
               low = (stride * index)
           if low < 0:
               low = 0
           high = (stride * (index + 1)) - 1
           print(f"Low: {low}, High: {high}")
           rows_to_average = cube[low : high]
           print(f"Rows to average: {rows_to_average}")
           average_row = mean(rows_to_average, axis=0)
```

```
            print(f"Averaged row: {average_row}")
            shrunken_cube[index] = average_row
    else:
        shrunken_cube = cube

    return shrunken_cube

# Block of code which will be executed when this file is executed as a script
if __name__ == '__main__':
    # Run the main() function
    main()
    # Terminate with exit code 0!
    exit(0)
```

Listing C3: Moment Extractor: File which holds the relevant information for FLASH product outputs.

```
"""FLASH Product Dictionary
# Template URL for downloading products
http://flash.ou.edu/flash_web/<subdir>/<prefix>.<yyyymmdd>.<HHMM00>.tif

# Dictionary structure
<DICT_ID> : {
  "subdir": "<SUBDIRECTORY_NAME>",
  "prefix": "<FILENAME_PREFIX>",
    "freq": "<PRODUCT_FREQUENCY_MINS>"
}
"""

PRODUCTS = {
    # Hydrologic Models - 10min freq
    "CREST_MaxQ": {"subdir": "maxq_new", "prefix": "maxq", "freq": 10},
    "CREST_MaxUnitQ": {"subdir": "maxunitq", "prefix": "maxunitq", "freq": 10},
    "CREST_MaxSoilSat": {"subdir": "sm", "prefix": "sm", "freq": 10},
    "SAC_MaxUnitQ": {"subdir": "maxunitq_sac", "prefix": "maxunitq", "freq": 10},
    "SAC_MaxSoilSat": {"subdir": "sm_sac", "prefix": "sm", "freq": 10},
    "HP_MaxQ": {"subdir": "maxq_hp", "prefix": "maxq", "freq": 10},
    "HP_MaxUnitQ": {"subdir": "maxunitq_hp", "prefix": "maxunitq", "freq": 10},
    "QPE_10m": {"subdir": "qpeaccum", "prefix": "qpeaccum", "freq": 10},
    # QPE Products - 2min freq
    "QPE_ARI_30m": {"subdir": "preciprp_30m", "prefix": "30U.ARI", "freq": 2},
    "QPE_ARI_1h": {"subdir": "preciprp_1h", "prefix": "01H.ARI", "freq": 2},
    "QPE_ARI_3h": {"subdir": "preciprp_3h", "prefix": "03H.ARI", "freq": 2},
    "QPE_ARI_6h": {"subdir": "preciprp_6h", "prefix": "06H.ARI", "freq": 2},
    "QPE_ARI_12h": {"subdir": "preciprp_12h", "prefix": "12H.ARI", "freq": 2},
    "QPE_ARI_24h": {"subdir": "preciprp_24h", "prefix": "24H.ARI", "freq": 2},
    "QPE_ARI_Max": {"subdir": "preciprp_max", "prefix": "MAX.ARI", "freq": 2},
    "QPE_FFG_1h": {"subdir": "ratio_1h", "prefix": "01H.RAT", "freq": 2},
    "QPE_FFG_3h": {"subdir": "ratio_3h", "prefix": "03H.RAT", "freq": 2},
    "QPE_FFG_6h": {"subdir": "ratio_6h", "prefix": "06H.RAT", "freq": 2},
    "QPE_FFG_Max": {"subdir": "ratio_max", "prefix": "MAX.RAT", "freq": 2},
    # Post Wildfire Products - 2min freq
    "WFR": {"subdir": "wildfirerain", "prefix": "wildfirerain", "freq": 2},
    # IBW Impacts - 10min freq
    "IBW_Base": {"subdir": "impacts/ibw_base", "prefix": "ibw_base.maxunitq", "freq": 10},
    "IBW_Cons": {"subdir": "impacts/ibw_cons", "prefix": "ibw_cons.maxunitq", "freq": 10},
    "IBW_Cata": {"subdir": "impacts/ibw_cata", "prefix": "ibw_cata.maxunitq", "freq": 10},
}
```

Listing C4: Moment Extractor: Common file I/O functionalities, defined for reusing in the main file.

```
""" Common File Input/Output Functions
"""

from datetime import datetime, timedelta
from json import dump, loads
from os import listdir, path

from numpy import array, nan
from osgeo import gdal
from pandas import read_csv, to_datetime
from shortuuid import uuid


def read_geotiff(filename, mask=False, flatten=False):
    """Read a Geotiff file, or a Geotiff mask file.
```

```
    This function uses gdal to open a raster mask file, and then extracts its
    first raster band. It returns the file's data, dimensions and pertinent
    geolocation metadata.

    Args:
        filename (str): path to the mask file (must be a geotiff).

    Returns:
        tuple: tuple containing the maks' data and metadata:
            mask_array (numpy.Array): flat array containing the mask data.
            cols (int): number of columns in the mask file.
            rows (int): number of rows in the mask file.
            geotransform (gdal.GeoTransform): mask file geotransformation data.
            projection (gdal.Projection): mask file projection.
    """

    # Empty GeoTIFF information dictionary
    geotiffinfo = {}

    # Opent the file
    ds = gdal.Open(filename)

    # Extract and save the file's geotransformation and projection
    geotransform = ds.GetGeoTransform()
    projection = ds.GetProjection()

    # Extract the bounding box coordinates
    xres = geotransform[1]
    yres = geotransform[5]
    maxy = geotransform[3]
    miny = maxy + yres * ds.RasterYSize
    minx = geotransform[0]
    maxx = minx + xres * ds.RasterXSize
    bounding_box = [minx, miny, maxx, maxy]

    ref_matrix = [[0, yres], [xres, 0], [minx, maxy]]

    # Extract the first raster band
    band = ds.GetRasterBand(1)

    # Save the first band as an array
    band_array = band.ReadAsArray()

    # Extract the raster's dimensions
    height, width = band_array.shape

    # Add the geotiff's metadata to the GeoTIFF information dictionary
    geotiffinfo["bounding_box"] = bounding_box
    geotiffinfo["ref_matrix"] = ref_matrix
    geotiffinfo["height"] = height
    geotiffinfo["width"] = width
    geotiffinfo["geotransform"] = geotransform
    geotiffinfo["projection"] = projection

    # Flaten the file's data into a 1D numpy array
    if flatten:
        band_array = array(band_array).flatten()

    # If reading a mask file, make sure all non-zero values are returned with
    # with a value of 1
    if mask:
        band_array = band_array[band_array == 1]

    # Return the GeoTIFF's data array, its dimensions, its geotransformation
    # and projection
    return band_array, geotiffinfo


def write_geotiff(output_filename, data_array, geotiffinfo, raster_band=1):
    """Function that writes a NumPy array into a GeoTIFF file.

    This function uses gdal to open a raster mask file, and then extracts its
    first raster band. It returns the file's data, dimensions and pertinent
    geolocation metadata.

    Arguments:
        output_filename {str} -- name for the GeoTIFF file
        data_array {numpy.ndarray} -- NumPy array containing the values to be
            written onto the GeoTIFF file
        geotiffinfo {dict} --  dictionary containing the metadata for the
            Geotiff file: bounding_box, ref_matrix, height, width,
            geotransform, and projection. See read_geotiff() for more details,
            in principle, these metadata should match the Geotiff that was
            read previously.

    Keyword Arguments:
        raster_band {int} -- Geotiff raster's band number to write the data
            into (default: {1})
    """

    # Read the metadata that is needed from the geotiffinfo dictionary
```

```
    # Leave out the bounding_box and ref_matrix properties
    # Raster height
    height = geotiffinfo["height"]
    # Raster width
    width = geotiffinfo["width"]
    # Target Geotransform
    geotransform = geotiffinfo["geotransform"]
    # Target projection
    projection = geotiffinfo["projection"]

    # Initialize the GDAL Geotiff driver
    driver = gdal.GetDriverByName('GTiff')
    # Create an output grid, with the appropriate data type (FLOAT32) and
    # compression enabled
    out_grid_data = driver.Create(output_filename, height, width, raster_band,
                                  gdal.GDT_Float32, ['COMPRESS=DEFLATE'])
    # Set the target geotransform
    out_grid_data.setGeoTransform(geotransform)
    # Set the target projection
    out_grid_data.SetProjection(projection)
    # Set the data's shape to (-1, height) TODO: Verify this works / is needed?
    data_array.shape = (-1, height)
    # Write the data to the defined raster band
    out_grid_data.GetRasterBand(raster_band).WriteArray(data_array, 0, 0)
    # Set the no-data values to -9999.0
    out_grid_data.GetRasterBand(raster_band).SetNoDataValue(-9999.0)
    # Flush out the data, and close the file
    out_grid_data = None


def generate_cube_file_list(reference_timestamp,
                            data_dir, data_subdir,
                            product_prefix,
                            delta_hours=6,
                            product_freq=10,
                            datetime_format='%Y%m%d.%H%M%S',
                            product_extension=".tif",
                            check_existing=False):
    """ Function that generate a list of file paths for a space-time data cube.
    """
    # Construct a datetime object from the reference timestamp, using the
    # provided datetime string format
    datetime_object = datetime.strptime(reference_timestamp, datetime_format)

    # Construct a datetime string list based on the reference timestamp and the
    # provided time delta in hours. The constructed list will always contain an
    # odd number of strings, since the middle one will always be the reference
    # datetime string, and this center will be "sandwiched" by equal amounts of
    # datetime strings spanning the requested time delta
    datetime_list = [datetime_object + timedelta(minutes=x)
                     for x in range(-int(delta_hours*60),
                                    int(delta_hours*(60+product_freq)),
                                    product_freq)]

    # Construct a filename list based on the datetime string list, data
    # subdirectory, product prefix, and product extension
    filename_list = []
    # For each datetime string in the datetime_list
    for item in datetime_list:
        # Extract the individual datetime components
        year = item.year
        month = item.month
        day = item.day
        hour = item.hour
        minute = item.minute

        # Make sure that the days, months, hours, and minutes are always
        # represented by strings composed of two digits
        if day < 10:
            day = '0' + str(day)
        if month < 10:
            month = '0' + str(month)
        if hour < 10:
            hour = '0' + str(hour)
        if minute < 10:
            minute = '0' + str(minute)

        # Construct the complete file path string
        file_path = f"{data_dir}/{data_subdir}/{product_prefix}.{str(year)}{str(month)}{str(day)}.\
        {str(hour)}{str(minute)}00{product_extension}"
        alternate_path = ""
        # For ARIs, check for a possible alternate product prefix!
        if product_prefix == "MAX.ARI":
            alternate_prefix = "MAX.RP"
            alternate_path = f"{data_dir}/{data_subdir}/{alternate_prefix}.{str(year)}{str(month)}{str(day)}.\
             {str(hour)}{str(minute)}00{product_extension}"
        #print(f"Path: {file_path}\nAlternate: {alternate_path}")
        # If we're verifying that files on the list actually exist
        if check_existing:
            # If the current file path exists
            if path.exists(file_path):
                # Add the current file path to the filename list
```

```
                filename_list.append(file_path)
            # If the alternate file path exists
            elif alternate_path and path.exists(alternate_path):
                # Add the alternate file path to the filename list
                filename_list.append(alternate_path)
            # If the current file path does not exist
            else:
                # Add None to the filename list
                filename_list.append(None)
        # If we're not verifying that files on the list exist
        else:
            # Add the current file path to the filename list
            filename_list.append(file_path)

    # Return the constructed filename list
    return filename_list


def read_standard_lsrs(lsr_file_path, no_index=True, no_category=True):
    """ Read a standard CSV file containing a collection of LSRs

    This function reads a "standard" CSV format containing LSR reports, as they
    are provided by the Iowa State Univeristy Local Storm Report Archive:
    https://mesonet.agron.iastate.edu/request/gis/lsrs.phtml
    """
    # Read the file, and define standard names for each columns, as well as
    # approrpiate data types
    if no_category:
        standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                                 names=["valid", "valid2", "lat", "lon",
                                        "mag", "wfo", "typecode", "typetext",
                                        "city", "county", "state", "source",
                                        "remark", "ugc", "ugcname"],
                                 index_col=False,  # "valid2",
                                 dtype={"valid": str, "valid2": str,
                                        "lat": str, "lon": str,
                                        "mag": str, "wfo": str,
                                        "typecode": str, "typetext": str,
                                        "city": str, "county": str,
                                        "state": str, "source": str,
                                        "remark": str, "ugc": str,
                                        "ugcname": str})
        standard_lsrs["category"] = None
    else:
        standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                                 names=["valid", "valid2", "lat", "lon",
                                        "mag", "wfo", "typecode", "typetext",
                                        "city", "county", "state", "source",
                                        "remark", "category", "ugc",
                                        "ugcname"],
                                 index_col=False,  # "valid2",
                                 dtype={"valid": str, "valid2": str,
                                        "lat": str, "lon": str,
                                        "mag": str, "wfo": str,
                                        "typecode": str, "typetext": str,
                                        "city": str, "county": str,
                                        "state": str, "source": str,
                                        "remark": str, "category": str,
                                        "ugc": str, "ugcname": str})

    if not no_index:
        standard_lsrs.set_index('valid2', inplace=True)

        # Make sure that dates and times are represented appropriately
        standard_lsrs.index = to_datetime(standard_lsrs.index)

        # Make sure that report sources, categories, and WFOs are represented
        # appropriately as categorical data
        standard_lsrs.source = standard_lsrs.source.astype('category')
        standard_lsrs.category = standard_lsrs.category.astype('category')
        standard_lsrs.wfo = standard_lsrs.wfo.astype('category')

    # Remove all NaNs from the remarks column (empty remarks) and replace them
    # with blank strings " "
    standard_lsrs.remark.replace(nan, " ", regex=True, inplace=True)

    # Return the loaded data in a Pandas DataFrame
    return standard_lsrs


def read_ibw_lsrs(lsr_file_path, no_index=True):
    """ Read a CSV file containing a collection of Expertly-Classified LSRs

    The 'magnitude' field corresponds to IBW categories for each LSR
    """
    # Read the file, and define standard names for each columns, as well as
    # approrpiate data types
    standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                             names=["time", "office", "local_time",
                                    "county", "location", "state",
                                    "event_type", "magnitude", "source",
                                    "lat", "lon", "remark"],
```

```
                                index_col=False,  # "valid2",
                                dtype={"time": str,
                                       "office": str,
                                       "local_time": str,
                                       "county": str,
                                       "location": str,
                                       "state": str,
                                       "event_type": str,
                                       "magnitude": str,
                                       "source": str,
                                       "lat": str,
                                       "lon": str,
                                       "remark": str})
    standard_lsrs["category"] = None

    if not no_index:
        standard_lsrs.set_index('time', inplace=True)

        # Make sure that dates and times are represented appropriately
        standard_lsrs.index = to_datetime(standard_lsrs.index)
        standard_lsrs.local_time = to_datetime(standard_lsrs.local_time)

        # Make sure that report sources, categories, and WFOs are represented
        # appropriately as categorical data
        standard_lsrs.source = standard_lsrs.source.astype('category')
        standard_lsrs.magnitude = standard_lsrs.source.astype('magnitude')
        standard_lsrs.category = standard_lsrs.category.astype('category')
        standard_lsrs.office = standard_lsrs.wfo.astype('office')

    # Remove all NaNs from the remarks column (empty remarks) and replace them
    # with blank strings " "
    standard_lsrs.remark.replace(nan, " ", regex=True, inplace=True)

    # Return the loaded data in a Pandas DataFrame
    return standard_lsrs


def read_mpings(mping_file_path, no_index=True, no_category=True):
    """ Read a CSV file containing a collection of mPing Flood Reports.
    """
    pass


def get_files_in_dir(dir_path="./", extension=""):
    ''' Return a list of files with the same extension in a given directory.
    '''
    # Output list holding the complete file paths
    found_files = []

    # Iterate over the list of files in the directory
    for file in listdir(dir_path):
        # If the file's extension matches what we are searching for
        if file.endswith(extension):
            # Add the file to the output file list
            found_files.append(path.join(dir_path, file))

    # Return the list of found files
    return found_files


def define_batches(num_reports, batch_size=100):
    '''Define a dictionary of batches to batch process a dataframe of LSRs

    This function takes the total number of reports, and calculates a number of
    batches using a specific batch size. These batches are assigned an ID, and
    index ranges are associates to them, so that the number of records in each
    batch of the input DataFrame's total size corresponds to at most the
    defined batch size.

    Batches are defined as dictionary entries with the following structure:
        {
            <batch_id> : {
                            "indices": (<start_index>, <end_index>),
                            "processed": False
                         }
        }
    where <batch_id> is an integer (starting at 0), <start_index> is the
    first index of this batch, and <end_index> is the last index of the batch.
    Note that the last index in the last batch will be equal to num_reports-1
    since the indices start at 0, and not at 1! The 'processed' key will be
    used to keep track of whether this specific batch has been processed
    successfully or not.
    '''
    # Output dictionary of batches based on the input DataFrame
    batches = {}

    # If batch size is == 0, assume no batching is desired, and output a single
    # batch holding the entirety of the num_reports
    if batch_size == 0:
        batches[batch_size] = {"indices": (0, num_reports - 1),
                               "processed": False}
    # Else if the batch size is not zero, do the appropriate calculations, and
```

```
    # define the number of full and partial batches corresponding to the
    # requested parameters
    else:
        # Number of "full" batches
        full_batches = num_reports // batch_size

        # Remainder for "partial" batch
        partial_batch = num_reports % batch_size

        # Add the "full" batches to the dictionary
        for batch in range(full_batches):
            batches[batch] = {"indices": (batch_size * batch,
                                          batch_size * (batch + 1) - 1),
                              "processed": False}

        # If available, add the partial batch to the dictionary
        if partial_batch > 0:
            batches[full_batches] = {"indices": (batch_size * full_batches,
                                                 (batch_size * full_batches)
                                                 + partial_batch - 1),
                                     "processed": False}

    # Return the batches dictionary
    return batches


def match_batch_results(file_uuid, batches, results_path="./results/"):
    '''

    The expected format for the batch result files is:
        <results_path>/<file_uuid>_<batch_id>.json
    '''

    # Check if there are any JSON files in the specified path
    result_files = get_files_in_dir(results_path, extension=".json")

    # Make sure that the JSON files found match the specified UUID
    result_files = list(filter(lambda x: file_uuid in x, result_files))

    # If the list of files in the requested folder, with the requested uuid is
    # empty, notify that no previous batch results were found, and return a
    # None value
    if not result_files:
        print("WARNING: No previous batch results found for current uuid!")

    # Else if there are valid files that match the uuid
    else:
        # For each of these files
        for json_file in result_files:
            # Determine its batch ID
            batch_id = json_file.split("/")[-1].split("_")[-1].split(".")[0]
            # Set this batch's processed variable as True
            batches[int(batch_id)]["processed"] = True

    # Return the batches dictionary, with updated "processed" values for all
    # the JSON batch results files found
    return batches


def hash_filename(file_path, verbose=False):
    ''' Strip the filename from a path, and generate a 22 digit UUID from it.

    This function receives a complete file path, remove any route paths out of
    it, and then generates a unique 'short' identifier (using shortuui) based
    on the stripped filename (including the file's extension).
    '''
    # Remove any paths from the file name
    stripped_filename = str(file_path).split('/')[-1]

    # If verbose, print the original path and the stripped file name
    if verbose:
        print(f"FILE_PATH: {file_path}\nFILE_NAME: {stripped_filename}")

    # Hash the file name and return a unique identifier
    return uuid(stripped_filename)


def read_json_results(json_file_path):
    ''' Read a JSON file containing LSR remarks, FFSI classification and score.

    This function reads in a JSON file of Classified LSRs into a dictionary,
    including the remarks, the probability classes, and their FFSI scores.
    '''
    # Open the file to be read
    with open(json_file_path, 'r') as j:
        # Load the dictionary as JSON
        contents = loads(j.read())

    # Return the contents of the file
    return contents
```

```
def write_json_results(results, fname="./results/test.json", indent=2):
    ''' Write a JSON file containing LSR remarks, FFSI class probs. and score.

    This function writes out a dictionary of Classified LSRs into a JSON file,
    including the remarks, the probability classes, and their FFSI scores.
    '''
    # Open a new file to be written
    with open(fname, "w") as outfile:
        # Dump the dictionary as JSON
        dump(results, outfile, indent=indent)
```

Listing C5: Moment Extractor: Common spatiotemporal aggregation functionalities, defined for reusing in the main file.

```
"""Common Space-Time Data Cube Functions
"""

from numpy import append, asarray, empty, multiply, zeros, array, count_nonzero
import rasterio

from fileio_common import read_geotiff


def get_event_grid(domain_geotiff, lat, lon, verbose=False):
    """ Obtain the grid cell's row/column indices for a given lat/lon pair.

    This function uses a Geotiff file (which defines the domain), and then
    calculates the row and column indices for the grid's cell which contains
    the provided lat/lon point.
    """
    # Use rasterio to open the domain geotiff
    with rasterio.open(domain_geotiff) as src:
        # Extract the domain's metadata
        metadata = src.meta

        # If verbose, print metadata
        if verbose:
            print(f"Domain metatada:\n\t{metadata}")

        # Use the transform metadata and the coordinates, to obtain the row and
        # column ids where we will find our target coordinates
        rowcol = rasterio.transform.rowcol(metadata['transform'],
                                           xs=float(lon),
                                           ys=float(lat))

        # Return a tuple of row and column indices where the corresponding
        # gridcell for the given lat/lon coords will be located
        return rowcol


def get_event_window_indices(center_rowcol, window_radius, height=3500,
                             width=7000, flatten=False, verbose=False):
    """ Get an event's spatial window's indices (rowcol or flattened).

    This function calculates the corresponding indices for a given event's
    search window, as defined by the event's Lat/Lon, a window radius, and the
    size of the original domain. Indices can be returned as a list of tuples
    representing (row, column) indices, or as a list of integers, representing
    'flattened' indices on a 1D vector.

    TODO: Maybe these calculations can be vectorized? Nested for loops are a
    working solution, but a lazy one.
    """
    # If verbose, print out the center grid's row/col indices
    if verbose:
        print(f"Center grid: {center_rowcol}")

    # Extract the center grid's row and column indices
    center_row, center_col = center_rowcol

    # Define the return variable's empty list, which will hold our indices
    window_rowcols = []

    # Calculate the number of rows and number of columns our search window will
    # have, from the input's window radius (a rectangular subdomain)
    num_rows = (2 * window_radius) + 1
    num_cols = (2 * window_radius) + 1

    # Variable to keep track of the rows we have traversed
    progress_rows = -window_radius
    # For each row in the search window (ignore the row's index)
    for _ in range(num_rows):
        # Variable to keep track of the columns we have traversed
        progress_cols = -window_radius
        # For each column in the search window (ignore the column's index)
```

```
        for _ in range(num_cols):
            # Calculate each of the search window's grid's row and column
            # indices by iterating over the rows and columns that make up the
            # search window
            window_cell_row = center_row + progress_rows
            window_cell_col = center_col + progress_cols
            # Make sure that the resulting indices are WITHIN the original
            # domain! (e.g. ignore those that fall outside of the domain's
            # original bounds)
            if ((window_cell_col >= 0 and window_cell_row >= 0) and
                    (window_cell_col < width and window_cell_row < height)):
                # Append the row/col indices for the current grid cell within
                # the search window to the output list
                window_rowcols.append((window_cell_row, window_cell_col))
            # Increment the number of columns we have traversed in our counter
            progress_cols += 1
        # Increment the number of rows we have traversed in our counter
        progress_rows += 1

    # If the indices are to be returned as 'flattened', in order to locate the
    # search window pixels on a 1D array domain
    if flatten:
        # Instantiate an empy result list for flat indices
        window_rowcols_flat = []
        # For each of the row/col indices we calculated previously
        for rowcol in window_rowcols:
            # Extract the row and column indices separately
            row = rowcol[0]
            col = rowcol[1]
            # Calculate the single-integer index for a given row/col index
            flat_index = (row * width) + (width - (width - col))
            # Append the current flat index to the result list
            window_rowcols_flat.append(flat_index)
        # Overwrite the row/col indices with the flattened indices
        window_rowcols = window_rowcols_flat

    # Return the calculated search window indices
    return window_rowcols


def get_event_window_mask(domain_shape, window_indices, flatten=False):
    """ Return the event window's mask, with the same dimensions as the domain.

    This function takes the domain shape, and creates an Array of equal
    dimensions, where only the event window pixels have non-zero values (1),
    and the rest of the domain is zeros.
    """
    # Greate a blank mask grid of the same shape as the domain, but full of 0s
    mask_grid = zeros(shape=domain_shape)
    # Make sure that only the indices of the event's window are equal to 1
    for index in window_indices:
        mask_grid[index] = 1

    # If the array is to be flattened
    if flatten:
        # Convert he output grid mask into a 1D array
        mask_grid = array(mask_grid).flatten()

    # Return the mask grid
    return mask_grid


def build_cube(file_list, domain_shape, mask_array=None, scale=1,
               verbose=False):
    """ Build a spacetime cube from a list of GeoTIFF files.
    """
    # If a mask file is provided
    if mask_array is not None:
        # Only count the pixels within the mask
        n_pixels = len(mask_array[mask_array>0])
    # If no mask is provided, count all pixels
    else:
        n_pixels = domain_shape[0] * domain_shape[1]

    # Initialize and empty spacetime cube
    cube = empty(shape=(0, n_pixels))

    # Iterate over the list of files that will populate the spacetime cube
    for filename in file_list:
        # If verbose, report the file names
        if verbose:
            print(f"Filename:{filename}")
        # If the current file is not None
        if filename is not None:
            # Read the current file's data
            try:
                data = asarray(read_geotiff(filename)[0])
            except Exception as e:
                print(f"Error reading file {filename}\n{e}")
                data = asarray(zeros(shape=domain_shape))
        # If the current file is None
        else:
```

```
            # Create a zero array with the expected dimensions
            data = asarray(zeros(shape=domain_shape))

        # If a mask was passed into the function
        if mask_array is not None:
            # Scale and crop the data using the mask, append it to the cube
            try:
                cube = append(cube, [multiply(data[mask_array > 0], scale)], axis=0)
            except Exception as e:
                print(f"Error reading file {filename}\n{e}")
                cube = append(cube, [multiply(data, scale)], axis=0)
        # If no mask was provided
        else:
            # Scale the data and append it to the cube
            cube = append(cube, [multiply(data, scale)], axis=0)

    # Return the built spacetime cube
    return cube
```

ChatGPT-Based LSR Classifier

Listing C6: ChatGPT Classifier: Main file which wraps all functionality and handles the classification of provided LSRs using a given prompt definition.

```
#!/usr/bin/env python3

import json
import os
import sys

import openai
from openai.error import RateLimitError, ServiceUnavailableError, APIError

import gpt_common as gpt
import impacts_common as impacts

# Constant which will hold the path to the JSON file containing the OpenAI
# secret key, which enables the use of the ChatGPT API
#KEY_FILE = './secrets/personal_key.json'
KEY_FILE = './secrets/OU-ISE_key.json'

# Timeout wait time
# PERSONAL API KEY = 20
# OU-ISE API KEY = 1
#WAIT_TIME = 20
WAIT_TIME = 1


# Constant which will hold the path to a text file contatining a narrative
↪  for
# flash flood impact category definitions, in a GPT-compatible prompt form
FFSI_DEFINITIONS = './docs/FFSI/FFSI_v1-extended.txt'
#FFSI_DEFINITIONS = './docs/FFSI/FFSI_v2-bullets.txt'

# Constant which will hold the path to a CSV text file containing Local
↪  Storm
# Reports
```

```
#LSR_FILE = './data/test/lsr_202207271200_202207301200_impacts.csv'
#LSR_FILE =
↪  "./data/historical/flashfloods_lsr_201804010000_202207010000.csv"
LSR_FILE = "./data/expertly-classified/ibw_classified_lsr.csv"

# Constant which will hold the batch size we want to use for splitting our
↪  LSR
# dataset, so we can process it one batch at a time
BATCH_SIZE = 20

# Constant which will hold the total number of batches that should be
↪  processed
# MAX_BATCHES = 0 means NO LIMIT!!
MAX_BATCHES = 0

# Constant which will hold the path to the desires results output location
RESULTS_OUTPUT = './results/'


# Main function, which will be the entry point that will be executed, when
↪  this
# program is run as a script from the command line
def main():
    '''Main function and point of entry for the execution of this script.
    '''
    # Import the secret OpenAI API key from the JSON file
    openai.api_key = gpt.read_api_key(KEY_FILE)

    # Read FFSI definition
    impact_defs = impacts.read_textual_definition(FFSI_DEFINITIONS)

    # Read whole LSRs from a standard CSV file
    #lsr_reports = impacts.read_standard_lsrs(LSR_FILE)
    lsr_reports = impacts.read_ibw_lsrs(LSR_FILE)

    # Define a Unique Identifyer for the LSR file that is being processed
    lsr_uuid = impacts.hash_filename(LSR_FILE)

    # Define batches for batch processing the LSRs, so that it is easier to
    # restart the process, in case of interruptions or failure.
    batches = impacts.define_batches(lsr_reports.shape[0], BATCH_SIZE)

    # Check for pre-existing batch result JSON files, and if found, update
    ↪  the
    # batches dictionary with "processed": True, for the matching batches
    batches = impacts.match_batch_results(file_uuid=lsr_uuid,
                                          batches=batches,
                                          results_path=RESULTS_OUTPUT)

    # Process the LSRs, using the ChatGPT API to classify them

    # Hold the number of total batches for future reference
    num_batches = len(batches)
    # if verbose:
```

```
print(f"Processing LSR file {lsr_uuid}:"
      f"{lsr_reports.shape[0]} reports / "
      f"{num_batches} batches")

# Variables to keep track of the total number of processed reports, as
↪  well
# as the number of processed and skipped batches
total_processed = 0
batches_skipped = 0
batches_processed = 0

# For each batch in batches
for batch_id in batches:
    # if verbose:
    print(f"Batch ID: {batch_id}")
    # If the current batch has been processed
    if batches[batch_id]["processed"]:
        # Do nothing, and move to the next batch
        # Keep track of how many batches were skipped
        batches_skipped += 1
        # if verbose:
        print(f"WARNING: Skipping batch {batch_id + 1}/{num_batches}"
              f"- already processed")
        processed_lsrs = None

    # If not, process the current batch
    else:
        # Get the current batch start and end indices for the LSR
        ↪  DataFrame
        start_idx, end_idx = batches[batch_id]["indices"]

        # if verbose:
        print(f"Indices: ({start_idx},{end_idx})")

        # Subset the LSR reports to only select the reports for this
        ↪  batch.
        # Notice that the end index is incremented by 1, since the top
        # index is always excluded by definition.
        current_lsrs = lsr_reports.iloc[start_idx : end_idx + 1]

        # Process the current batch of LSRs
        try:
            processed_lsrs = \
            gpt.classify_lsr_remarks(current_lsrs['remark'],
                                     impact_defs,
                                     temperature=0,
                                     wait_time=WAIT_TIME,
                                     verbose=True)
        except (RateLimitError,
                ServiceUnavailableError,
                APIError,
                OSError) as e:
            print(f"ERROR! - The following eception occurred:\n\t {e}")
            print("Waiting 10s and retrying once before breaking!")
```

```
            gpt.wait_timeout(10)
            processed_lsrs = \
             gpt.classify_lsr_remarks(current_lsrs['remark'],
                                      impact_defs,
                                      temperature=0,
                                      wait_time=WAIT_TIME,
                                      verbose=True)

        # Write the current batch's result as a JSON file, identified
        ↪  by
        # the lsr_uuid string and the current batch's ID, in the
        ↪  desired
        # output results folder
        batch_filename = f"{lsr_uuid}_{batch_id}.json"
        batch_path = os.path.join(RESULTS_OUTPUT, batch_filename)
        impacts.write_results_json(processed_lsrs, batch_path)

        # if verbose:
        print(f"Wrote partial results file: {batch_path}")

        # Mark batch as processed:
        batches[batch_id]["processed"] = True

        # Keep track of how many batches were processed
        batches_processed += 1

        # if verbose:
        print(f"Processed {len(processed_lsrs)} for"
              f"batch {batch_id + 1}/{num_batches}")

    # Break after MAX_BATCHES, if MAX_BATCHES > 0
    if MAX_BATCHES and batch_id == MAX_BATCHES:
        # if verbose:
        print(f"WARNING: MAX_BATCHES of {MAX_BATCHES}"
              f" reached! HALTING!\n")
        break

# Keep track of how many total LSRs were processed
if processed_lsrs:
    total_processed += len(processed_lsrs)
else:
    total_processed += 0

# Notify the user processing is done, and provide some counts on
↪  results
print(f"DONE!\n\t"
      f"LSR file: {LSR_FILE}\n\t"
      f"UUID: {lsr_uuid}\n\t"
      f"processed {batches_processed}/{num_batches} batches\n\t"
      f"skipped {batches_skipped}/{num_batches} batches \n\t"
      f"processed {total_processed} LSRs")

# Consolidate processed JSON results into a single CSV file
```

```
# Check if there are any JSON files in the current output path
result_files = impacts.get_files_in_dir(RESULTS_OUTPUT,
↪  extension=".json")
# Make sure that the JSON files found match the current UUID
result_files = list(filter(lambda x: lsr_uuid in x, result_files))

# If the list of result files in the output folder, with the requested
↪  uuid
# is empty, notify that no batch results were found
if not result_files:
    print("ERROR: No batch result files were found for current uuid!")

# Else if there are result files in the output folder
else:
    # Create a copy of the original LSR DataFrame
    output_lsrs = lsr_reports.copy()
    #output_lsrs.index = output_lsrs.remark
    # Create the new columns for the new results data in the DataFrame
    output_lsrs["MINOR"] = None
    output_lsrs["MODERATE"] = None
    output_lsrs["SERIOUS"] = None
    output_lsrs["SEVERE"] = None
    output_lsrs["CATASTROPHIC"] = None
    output_lsrs["FFSI"] = None
    output_lsrs["EXTRA"] = None
    # For each of these files
    for json_file in result_files:
        # Read the results JSON file
        batch_results = impacts.read_json_results(json_file)
        # Get the current batch ID from the batch results file
        batch_id =
        ↪  json_file.split("/")[-1].split('_')[-1].split('.')[0]
        # Get the LSR dataframe ID corresponding to the beginning of
        ↪  the
        # current batch
        start_index = int(batch_id) * BATCH_SIZE
        # Iterate over each result, and add the data to the
        ↪  corresponding
        # columns in the output LSR dataframe, for the corresponding
        ↪  LSR
        for batch_index in batch_results:
            idx = start_index + int(batch_index)
            p_min = batch_results[batch_index][1]["MINOR"]
            p_mod = batch_results[batch_index][1]["MODERATE"]
            p_ser = batch_results[batch_index][1]["SERIOUS"]
            p_sev = batch_results[batch_index][1]["SEVERE"]
            p_cat = batch_results[batch_index][1]["CATASTROPHIC"]
            score = batch_results[batch_index][2]
            extra = batch_results[batch_index][3]
            output_lsrs.loc[idx]["MINOR"] = p_min / 100
            output_lsrs.loc[idx]["MODERATE"] = p_mod / 100
            output_lsrs.loc[idx]["SERIOUS"] = p_ser / 100
            output_lsrs.loc[idx]["SEVERE"] = p_sev / 100
            output_lsrs.loc[idx]["CATASTROPHIC"] = p_cat / 100
```

```
                output_lsrs.loc[idx]["FFSI"] = score
                output_lsrs.loc[idx]["EXTRA"] = extra

        output_lsrs.reset_index(drop=True)
        output_lsrs.to_csv(f"./results/{lsr_uuid}_classified.csv",
        ↪  index=False)

        print("DONE!")


# Block of code which will be executed when this file is executed as a
↪  script
if __name__ == '__main__':
    # Run the main() function
    main()
    # Terminate with exit code 0!
    sys.exit(0)
```

Listing C7: ChatGPT Classifier: Common GPT API functionalities, defined for reusing in the main file.

```
#!/usr/bin/env python3

import json
from sys import stdout
from time import sleep
from numpy import isnan

from openai import ChatCompletion

from impacts_common import ffsi_score


def read_api_key(json_file_path):
    ''' Read the OpenAI key stored as a dictionary in a txt file.

    Function that reads a JSON file containing an OpenAI API key, which is
    stored in a dictionary of the form:
        { "secret_key": "<OpenAI API Secret Key String>" }
    '''
    # Read the JSON file
    with open(json_file_path) as key_file:
        openai_key = json.load(key_file)

    # Access and return the value of the API key
    return openai_key["secret_key"]


def query_gpt(query, role="user", system_task={}, temperature=1, top_p=1):
    ''' Query the GPY API and return the first completion result.

    Function that queries the GPT API, and returns the first completion
```

```
    produced as a response to the query.

    OpenAI's default values for temperature and top_p are maintained here
    as default values. From the official documentation:

    - temperature:
        What sampling temperature to use, between 0 and 2. Higher values
        like 0.8 will make the output more random, while lower values like
        0.2 will make it more focused and deterministic.

        We generally recommend altering this or top_p but not both.

    - top_p:
        An alternative to sampling with temperature, called nucleus
        sampling, where the model considers the results of the tokens with
        top_p probability mass. So 0.1 means only the tokens comprising
        the top 10% probability mass are considered.

        We generally recommend altering this or temperature but not both.
    '''
    # Process valid non-empty, non-blank string queries
    if query and query != " ":
        # Message list for the API query
        message_list = []

        # If a system task is passed as parameter, make sure to first
        ↪  append it to
        # the message_list
        if system_task:
            message_list.append(system_task)


        # Append the query to the message_list
        message_list.append({"role": role, "content": query})

        # Generate and receive back a ChatGPT completion for the GPT API
        completion = ChatCompletion.create(
            # Make sure to use GPT-3.5-turbo v.0301, and not the updated
            ↪  version
            # GPT-3.5-turbo v.0613 (which will be default on 06-27-2023,
            ↪  when using
            # the default string "gpt-3.5-turbo")
            model="gpt-3.5-turbo-0301",
            messages=message_list,
            temperature=temperature,
            top_p=top_p
        )

        # Return the first completion produced by our query to the API
        result = completion.choices[0].message.content
    # Handle EMPTY LSR remarks, by returning a classification dictionary of
    # zero probabilities, and an EXTRA string reporting the missing remark
    else:
        result = """{"MINOR": 0,
```

```
                    "MODERATE": 0,
                    "SERIOUS": 0,
                    "SEVERE": 0,
                    "CATASTROPHIC": 0
                    }NO REMARK FOR THIS LSR!"""

    # Return the result
    return result


def wait_timeout(seconds=20):
    '''Wait (sleep) for a determined number of seconds and show countdown.

    This function sleeps for a determined number of seconds, and logs to
    the console a countdown progress of how much time remains in the
    waiting period.

    GPT API Request rate limits are defined in terms of both Requests Per
    Minute (RPM) and Tokens Per Minute (TPM):

    Free trial users: 3 RPM (20s timeout) / 150,000 TPM
    Pay-as-you-go users: 60 RPM (1s timeout) / 250,000 TPM
    '''
    print("Waiting before querying the API again...")
    for remaining in range(seconds, 0, -1):
        stdout.write("\r")
        stdout.write("{:2d} seconds remaining.".format(remaining))
        stdout.flush()
        sleep(1)
    print("\n")


def classify_lsr_remarks(lsr_remarks_list, impact_defs,
                         temperature=1, top_p=1, wait_time=20,
                         starting_idx=0, limit=0, verbose=False):
    '''Classify a list of LSR remarks using an impact definition
       and ChatGPT.

    This function queries ChatGPT for a classification based on a specific
    impact definition (FFSI), for each LSR remark contained in the input
    list received.

    This function returns a dictionary containing the original remarks,
    their associated probabilistic classifications produced by ChatGPT
    using the impact definitions, and its associated FFSI score.
    '''

    # Empty dictionary which will hold the results of the processed LSRs
    processed_lsrs = {}

    # Initialize the ChatGPT system task based on the impact definitions
    initialization_task = {"role": "system", "content": impact_defs}

    # Print out limits, the system task and impact definitions if verbose
```

```
if verbose:
    # If limit is provided, show a warning and wait a bit
    if limit:
        print("WARNING: only %i LSRs will be processed! \n" % limit)
        sleep(2.5)

    # Print out the system task
    print("SYSTEM TASK:\n", impact_defs, '\n')

# Keep track of total LSRs that need to be processed
total_lsrs = len(lsr_remarks_list)

# If the starting_index is not 0, slice the lst list to start at the
# requested initial index
if starting_idx:
    # If verbose, print a warning notifying of new starting point
    if verbose:
        print("WARNING: starting at index %i, out of %i" %
              (starting_idx,
               total_lsrs))
        sleep(2.5)

    # Slice the LSR remarks to start at the new index
    lsr_remarks_list = lsr_remarks_list[starting_idx:]
    # Update the number of total LSRs to be processed
    total_lsrs = len(lsr_remarks_list)


# Query the ChatGPT API prepending the system task each time
for index, remark in enumerate(lsr_remarks_list):

    if verbose:
        print(f"Remark sent: {remark}")

    # Construct and send the query, receiving the result
    result = query_gpt(remark,
                       role="user",
                       system_task=initialization_task,
                       temperature=temperature,
                       top_p=top_p)

    if verbose:
        print(f"Result received: {result}")

    # In case the GPT API response includes more text than the
    ↪  requested
    # JSON dictionary formatted answer, separate the dictionary portion
    # from the rest of the response, and add the remaining response as
    ↪  a
    # "extra" in the result's dictionary

    # Extract any extra output GPT may have generated
    response_extra = result.split('}')[-1].split("\n\n")[-1]
```

```
        # Extract the classification results
        response_classes = result.split('{')[-1].split('}')[0]

        # Handle non-classification outputs, which have only EXTRA GPT
        ↪  outputs
        if response_classes == response_extra:
            response_classes = """"MINOR": 0,
                                  "MODERATE": 0,
                                  "SERIOUS": 0,
                                  "SEVERE": 0,
                                  "CATASTROPHIC": 0"""
        response_json = '{' + response_classes + '}'

        if verbose:
            print(f"Response JSON: \n\t{response_json}")
            print(f"Response EXTRA: \n\t{response_extra}\n")

        # Read the resulting probabilities in JSON format as a dictionary
        probs_dict = json.loads(response_json)

        # Calculate the FFSI score for the current classification results
        score = ffsi_score(probs_dict)

        # If verbose, print out the original remark, the classification,
        # probabilities, and its corresponding score
        if verbose:
            print("INDEX: %s\n" % str(index),
                  "PROMPT: %s\n" % remark,
                  "PROBS: %s\n" % result,
                  "SCORE: %s\n" % str(score),
                  "EXTRA: %s\n" % response_extra)

        # Add the current results to the output LSRs dictionary
        processed_lsrs[index] = [remark, probs_dict, score, response_extra]

        # If a limit is provided, stop classifyig LSRs after x reports
        if limit:
            # if the current index is equal to the limit, stop the loop
            if index == limit - 1:
                break

        # If there are still remarks to process, wait some time to comply
        ↪  with
        # the API's request rate limits
        if index < total_lsrs - 1:
            # Wait wait_time seconds (X requests/min rate limit)
            wait_timeout(wait_time)

    # Return the processed LSRs
    return processed_lsrs
```

Listing C8: ChatGPT Classifier: Common Impacts functionalities, defined for reusing in the main file.

```
#!/usr/bin/env python3

import csv
import json
import os

from numpy import round, nan
from pandas import read_csv, to_datetime
from shortuuid import uuid


def read_textual_definition(text_file_path):
    ''' Read a text file containing an FFSI definition.

    This function reads an textfile into a multiline string object.
    '''
    with open(text_file_path) as text_file:
        ffsi_definition = text_file.read()

    return ffsi_definition


def read_lsr_remarks(lsr_file_path, column_name='REMARK'):
    ''' Read a single column from CSV file containing Local Storm Reports.

    This function reads a CSV file containing Local Storm Reports, and
    returns a list of 'remarks' contained in the 'REMARK' column.
    '''
    # Open the CSV file
    lsr_file = open(lsr_file_path)

    # Create a DictReader object
    lsrs = csv.DictReader(lsr_file)

    # List to hold the LSR remarks
    remarks = []

    # Iterate over each report, and append its remark to the remarks list
    for report in lsrs:
        remarks.append(report[column_name])

    # Return the LSR remarks
    return remarks


def read_standard_lsrs(lsr_file_path, no_index=True, no_category=True):
    """ Read a standard CSV file containing a collection of LSRs

    This function reads a "standard" CSV format containing LSR reports, as
    they are provided by the Iowa State Univeristy Local Storm Report
    Archive: https://mesonet.agron.iastate.edu/request/gis/lsrs.phtml
```

```
    """
    # Read the file, and define standard names for each columns, as well as
    # approrpiate data types
    if no_category:
        standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                                 names=["valid", "valid2", "lat", "lon",
                                        "mag", "wfo", "typecode",
                                        ↪  "typetext",
                                        "city", "county", "state",
                                        ↪  "source",
                                        "remark", "ugc", "ugcname"],
                                 index_col=False,#"valid2",
                                 dtype={"valid": str, "valid2": str,
                                        "lat": str, "lon": str,
                                        "mag": str, "wfo": str, "typecode":
                                        ↪  str,
                                        "typetext": str, "city": str,
                                        "county": str, "state": str,
                                        ↪  "source": str,
                                        "remark": str, "ugc": str,
                                        ↪  "ugcname": str})
        standard_lsrs["category"]=None
    else:
        standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                                 names=["valid", "valid2", "lat", "lon",
                                        "mag", "wfo", "typecode",
                                        ↪  "typetext",
                                        "city", "county", "state",
                                        ↪  "source",
                                        "remark", "category", "ugc",
                                        ↪  "ugcname"],
                                 index_col=False,#"valid2",
                                 dtype={"valid": str, "valid2": str,
                                        "lat": str, "lon": str,
                                        "mag": str, "wfo": str, "typecode":
                                        ↪  str,
                                        "typetext": str, "city": str,
                                        "county": str, "state": str,
                                        ↪  "source": str,
                                        "remark": str, "category": str,
                                        ↪  "ugc": str,
                                        "ugcname": str})

    if not no_index:
        standard_lsrs.set_index('valid2', inplace=True)

        # Make sure that dates and times are represented appropriately
        standard_lsrs.index = to_datetime(standard_lsrs.index)

        # Make sure that report sources, categories, and WFOs are
        ↪  represented
        # appropriately as categorical data
        standard_lsrs.source = standard_lsrs.source.astype('category')
        standard_lsrs.category = standard_lsrs.category.astype('category')
```

```
        standard_lsrs.wfo = standard_lsrs.wfo.astype('category')

    # Remove all NaNs from the remarks column (empty remarks) and replace
    ↪  them
    # with blank strings " "
    standard_lsrs.remark.replace(nan," ",regex=True, inplace=True)

    # Return the loaded data in a Pandas DataFrame
    return standard_lsrs


def read_ibw_lsrs(lsr_file_path, no_index=True):
    """ Read a CSV file containing a collection of Expertly-Classified LSRs

    The 'magnitude' field corresponds to IBW categories for each LSR
    """
    # Read the file, and define standard names for each columns, as well as
    # approrpiate data types
    standard_lsrs = read_csv(lsr_file_path, delimiter=',', header=0,
                             names=["time", "office", "local_time",
                                    "county", "location", "state",
                                    "event_type", "magnitude", "source",
                                    "lat", "lon", "remark"],
                             index_col=False,  # "valid2",
                             dtype={"time": str,
                                    "office": str,
                                    "local_time": str,
                                    "county": str,
                                    "location": str,
                                    "state": str,
                                    "event_type": str,
                                    "magnitude": str,
                                    "source": str,
                                    "lat": str,
                                    "lon": str,
                                    "remark": str})
    standard_lsrs["category"] = None

    if not no_index:
        standard_lsrs.set_index('time', inplace=True)

        # Make sure that dates and times are represented appropriately
        standard_lsrs.index = to_datetime(standard_lsrs.index)
        standard_lsrs.local_time = to_datetime(standard_lsrs.local_time)

        # Make sure that report sources, categories, and WFOs are
        ↪  represented
        # appropriately as categorical data
        standard_lsrs.source = standard_lsrs.source.astype('category')
        standard_lsrs.magnitude = standard_lsrs.source.astype('magnitude')
        standard_lsrs.category = standard_lsrs.category.astype('category')
        standard_lsrs.office = standard_lsrs.wfo.astype('office')
```

```
    # Remove all NaNs from the remarks column (empty remarks) and replace
    ↪ them
    # with blank strings " "
    standard_lsrs.remark.replace(nan, " ", regex=True, inplace=True)

    # Return the loaded data in a Pandas DataFrame
    return standard_lsrs


def ffsi_score(probs, normalize=False):
    ''' Calculate a single score from FFSI class probabilities.

    This function converts a dictionary or list of FFSI class
    probabilities (in percents), into a single score between 1 and 5,
    representing a continuum of values across the total number of classes.
    This score can also be normalized to values between 0 and 1.
    '''
    # If the input probabilities are passed in as a dictionary
    if type(probs) is dict:
        # Calculate the score, assume probabilities are in percent
        ffsi_score = (probs["MINOR"] / 100 * 1 +
                      probs["MODERATE"] / 100 * 2 +
                      probs["SERIOUS"] / 100 * 3 +
                      probs["SEVERE"] / 100 * 4 +
                      probs["CATASTROPHIC"] / 100 * 5)

    # Else if the input probabilities are passed as a list of numbers
    elif type(probs) is list:
        # Calculate the score, assume probabilities are in percent
        ffsi_score = (probs[0] / 100 * 1 +
                      probs[1] / 100 * 2 +
                      probs[2] / 100 * 3 +
                      probs[3] / 100 * 4 +
                      probs[4] / 100 * 5)

    # If varues are to be normalized
    if normalize:
        # Divide the score by the number of classes
        ffsi_score /= 5

    # Return the calculated FFSI score
    return ffsi_score


def write_results_json(results, fname="./results/test.json", indent=2):
    ''' Write a JSON file containing LSR remarks, FFSI classification
        and score.

    This function writes out a dictionary of Classified LSRs into a JSON
    file, including the remarks, the probability classes, and their FFSI
    scores.
    '''
    # Open a new file to be written
    with open(fname, "w") as outfile:
```

```
        # Dump the dictionary as JSON
        json.dump(results, outfile, indent=indent)


def define_batches(num_reports, batch_size=100):
    '''Define a dictionary of batches to batch process a dataframe of LSRs

    This function takes the total number of reports, and calculates a
    number of batches using a specific batch size. These batches are
    assigned an ID, and index ranges are associates to them, so that the
    number of records in each batch of the input DataFrame's total size
    corresponds to at most the defined batch size.

    Batches are defined as dictionary entries with the following
    structure:
        {
            <batch_id> : {
                            "indices": (<start_index>, <end_index>),
                            "processed": False
                         }
        }
    where <batch_id> is an integer (starting at 0), <start_index> is the
    first index of this batch, and <end_index> is the last index of the
    batch. Note that the last index in the last batch will be equal to
    num_reports-1 since the indices start at 0, and not at 1! The
    'processed' key will be used to keep track of whether this specific
    batch has been processed successfully or not.
    '''
    # Output dictionary of batches based on the input DataFrame
    batches = {}

    # If batch size is == 0, assume no batching is desired, and output a
    ↪  single
    # batch holding the entirety of the num_reports
    if batch_size == 0:
        batches[batch_size] = {"indices": (0, num_reports - 1),
                               "processed": False}
    # Else if the batch size is not zero, do the appropriate calculations,
    ↪  and
    # define the number of full and partial batches corresponding to the
    # requested parameters
    else:
        # Number of "full" batches
        full_batches = num_reports // batch_size

        # Remainder for "partial" batch
        partial_batch = num_reports % batch_size

        # Add the "full" batches to the dictionary
        for batch in range(full_batches):
            batches[batch] = {"indices": (batch_size * batch,
                                          batch_size * (batch + 1) - 1),
                              "processed": False}
```

```
        # If available, add the partial batch to the dictionary
        if partial_batch > 0:
            batches[full_batches] = {"indices": (batch_size * full_batches,
                                                 (batch_size *
                                                 ↪  full_batches)
                                                 + partial_batch - 1),
                                     "processed": False}

    # Return the batches dictionary
    return batches


def hash_filename(file_path, verbose=False):
    ''' Strip the filename from a path, and generate a 22 digit UUID
        from it.

    This function receives a complete file path, remove any route paths
    out of it, and then generates a unique 'short' identifier (using
    shortuui) based on the stripped filename (including the file's
    extension).
    '''
    # Remove any paths from the file name
    stripped_filename = str(file_path).split('/')[-1]

    # If verbose, print the original path and the stripped file name
    if verbose:
        print(f"FILE_PATH: {file_path}\nFILE_NAME: {stripped_filename}")

    # Hash the file name and return a unique identifier
    return uuid(stripped_filename)


def get_files_in_dir(dir_path="./", extension=""):
    '''Return a list of files with the same extension in a given
       directory.
    '''
    # Output list holding the complete file paths
    found_files = []

    # Iterate over the list of files in the directory
    for file in os.listdir(dir_path):
        # If the file's extension matches what we are searching for
        if file.endswith(extension):
            # Add the file to the output file list
            found_files.append(os.path.join(dir_path, file))

    # Return the list of found files
    return found_files


def match_batch_results(file_uuid, batches, results_path="./results/"):
    '''

    The expected format for the batch result files is:
```

```
        <results_path>/<file_uuid>_<batch_id>.json
    '''

    # Check if there are any JSON files in the specified path
    result_files = get_files_in_dir(results_path, extension=".json")

    # Make sure that the JSON files found match the specified UUID
    result_files = list(filter(lambda x: file_uuid in x, result_files))

    # If the list of files in the requested folder, with the requested uuid
    ↪ is
    # empty, notify that no previous batch results were found, and return a
    # None value
    if not result_files:
        print("WARNING: No previous batch results found for current uuid!")

    # Else if there are valid files that match the uuid
    else:
        # For each of these files
        for json_file in result_files:
            # Determine its batch ID
            batch_id =
            ↪ json_file.split("/")[-1].split("_")[-1].split(".")[0]
            # Set this batch's processed variable as True
            batches[int(batch_id)]["processed"] = True

    # Return the batches dictionary, with updated "processed" values for
    ↪ all
    # the JSON batch results files found
    return batches


def read_json_results(json_file_path):
    ''' Read a JSON file containing LSR remarks, FFSI classification and
        score.

    This function reads in a JSON file of Classified LSRs into a
    dictionary, including the remarks, the probability classes, and their
    FFSI scores.
    '''
    # Open the file to be read
    with open(json_file_path, 'r') as j:
        # Load the dictionary as JSON
        contents = json.loads(j.read())

    # Return the contents of the file
    return contents
```