THESIS

A FRAMEWORK FOR RESOURCE EFFICIENT PROFILING OF SPATIAL MODEL PERFORMANCE

Submitted by

Caleb Carlson

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2022

Master's Committee:

Advisor: Shrideep Pallickara
Co-Advisor: Sangmi Lee Pallickara

Henry Adams

ABSTRACT

A FRAMEWORK FOR RESOURCE EFFICIENT PROFILING OF SPATIAL MODEL PERFORMANCE

We design models to understand phenomena, make predictions, and/or inform decision-making. This study targets models that encapsulate spatially evolving phenomena. Given a model M, our objective is to identify how well the model predicts across all geospatial extents. A modeler may expect these validations to occur at varying spatial resolutions (e.g., states, counties, towns, census tracts). Assessing a model with all available ground-truth data is infeasible due to the data volumes involved. We propose a framework to assess the performance of models at scale over diverse spatial data collections. Our methodology ensures orchestration of validation workloads while reducing memory strain, alleviating contention, enabling concurrency, and ensuring high throughput. We introduce the notion of a validation budget that represents an upper-bound on the total number of observations that are used to assess the performance of models across spatial extents. The validation budget attempts to capture the distribution characteristics of observations and is informed by multiple sampling strategies. Our design allows us to decouple the validation from the underlying model-fitting libraries to interoperate with models designed using different libraries and analytical engines; our advanced research prototype currently supports Scikit-learn, PyTorch, and TensorFlow. We have conducted extensive benchmarks that demonstrate the suitability of our methodology.

ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation [OAC-1931363, ACI-1553685] and the National Institute of Food & Agriculture [COL0-FACT-2019].

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

We build models to understand phenomena and inform decision making. These models may be analytical models where we fit a model to the data, or they may be domain theoretic models. We consider spatial models, i.e., models that attempt to capture spatiotemporally evolving phenomena. The class of models we consider in this study are regression models and the data we consider are spatiotemporal datasets. In spatiotemporal datasets, the data are geocoded (with latitude/longitude information) and observations have timestamps associated with them. Spatial datasets continue to be made available in a variety of domains. These datasets encapsulate observational data at diverse timescales and allow scientists to explore interrelated phenomena. A model designer may choose to leverage features (or variables) from diverse collections as the independent variables while choosing a dependent variable (also referred to as the response variable or target). Once a model is constructed, a question that arises is "How is the model performing?". Model validations are a precursor to informing model refinements and targeting specific spatial extents to further scrutiny. For example, if a model predicting the direction of a forest fire works well in Oregon but not so well in Colorado, the model may choose to consider topographical characteristics such as elevation, terrain, etc. into consideration. More importantly, such an analysis is likely to reveal several key factors driving model behavior at different spatial extents. The crux of this paper is to effectively evaluate the performance of spatial models at scale. The performance measures used to assess a model depends on the model type and the measure deemed most suitable by the model designer. For example, for a regression model, a modeler may use RMSE, MAE, MSE, SSIM, or PSNR to assess the model's performance. These performance measures are predicated on access to ground truth. Further, these assessments must utilize resources frugally, minimize network and disk I/O, and interoperate with diverse analytical engines. Ultimately, our goal is to inform targeted model redesign, refinements, and calibration at scale by identifying spatial extents where a model performs well and where it does not.

Non-goals: This study does not focus on training machine learning models. Rather the focus is to validate, at scale, model performance across different spatiotemporal scopes.

1.1 Challenges

Performing model validations at scale introduce challenges stemming from the nature of the datasets and model evaluations.

1. Data volumes: The datasets we consider are voluminous comprising a large number of observations. These observations are multidimensional encapsulating multiple features of interest.

2. Model inferences can be resource intensive: Model inferences trigger disk I/O, have computational overheads associated with them, and can have large memory footprints. In some cases, model inferences may trigger network I/O during data accesses. As such, the evaluations must balance validation coverage with the resource intensive nature of these evaluations.

3. Interoperate with diverse analytical engines: Researchers develop models using diverse analytical engines such as scikit-learn, TensorFlow, PyTorch, Spark etc. Different analytical engines have different model storage formats, encoding formats, and invocation/pipelining schemes that should be reconciled.

4. Parametrization of models: The parameters that serve as inputs and the output of these models may be drawn from different collections. Furthermore, the inputs may have different preprocessing operations such as normalization, encoding format reconciliations, and unit transformations that need to be performed.

1.2 Research Questions

The overarching theme of this study is: How can we perform model validations at scale over voluminous spatiotemporal datasets? Within this broader theme we explore the following research questions:

RQ-1: How can we strike a balance between validation coverage and the resource intensive nature of validations?

RQ-2: How can we ensure that our model validations will scale? Given the large spatial extent

RQ-3: How can we effectively interoperate with diverse analytical engines? [We treat models as black boxes without inspecting the internal structural properties of the models. For instance, the models we consider could be based on partial differential equations, decision trees, matrix multiplications and convolutions, etc.

RQ-4: How can we effectively characterize model performance over large spatial extents? [Visualization results can be streamed incrementally, and the graphs refined as the data become available.]

1.3 Approach Summary

Our methodology encompasses staging of datasets, apportioning of observations for validation, creation and dispersion of model instances, parametrization of models alongside any expected wrangling of features, and visualizing model performance and uncertainty measures. These are accomplished while ensuring effective resource utilizations, reconciling contention, alleviating disk and network I/O, and ensuring timeliness. We allow users to specify the granularities at which model validations should be performed. We rely on a role-based access control scheme to control the scope and the validation budget assigned to a user.

Geocoded observations have $\langle latitude, longitude \rangle$ coordinates associated with them. Our methodology collates disparate observations into smaller spatial extents based on administrative boundaries. The goal of our management scheme is to manage the competing pulls of dispersion and collation. Each spatial extent is represented as a shape file encapsulating the N-sided polygon. We base our preprocessing on shape files for administrative boundaries (from the US Census Survey). Similar shape file exists internationally for provinces, cantons, etc. Each observation is tested for whether it is encapsulated within a shapefile (for the smallest, indivisible aggregation

unit which is a Census tract in our study) and assigned a single dimensional prefix; the prefix assignment is hierarchical allowing aggregation along administrative boundaries.

Models are validated with ground truth observational data. To reduce disk I/O and computational overheads involved in assessing model performance, we introduce the notion of a validation budget. This represents the upper-bound on the total number of observations that are expended. Within this broader concept, we explore three different schemes to apportion the validation budget across model instances: equal, proportional, and uncertainty reduction. To reduce the number of repeated I/O operations that are triggered when observations are retrieved in a piecemeal fashion, we perform batched retrievals of observations per spatial extent.

Partitioning of the model performance based on spatial extents allow us to have a finer grained view of model performance. To profile model performance, we create (or reuse) a model instance for the spatial extent under consideration.

To maximize interoperability with diverse analytical engines, we treat models as black boxes and consider only their parametrization, data preprocessing, and performance characteristics. Data preprocessing involves normalizing features based on the schemes specified by the modeler. Finally, we contrast model outputs with ground truth available from observational data and use that to compute a model performance metric based on the user-specified metric. We reconcile multiple model representation formats and marshalling schemes. Further, we leverage thread pools, locking, and synchronization mechanisms to ensure a high degree of concurrency alongside thread safety during model validations.

Model performance metrics are collated at a controller node, which may decide to allocate an additional budget to reduce uncertainty. Once the model performance results satisfy the stopping criteria, the results are streamed to a dashboard to visualize model performance. Model performance is visualized using a choropleth map with the model performance and variability rendered as a heat map. The choropleth maps can be used to inform targeted model refinements for particular spatial extents.

1.4 Paper Contributions

Here we described our framework to assess model performance at scale. Our specific contributions include the following:

1. Effective identification of spatial extents where a model performs well and where it does not.
2. Our validation budgets can be apportioned using different schemes. Our scheme allows preferentially targeting spatial extents where the model has high variability in performance.
3. Interoperate with diverse analytical engines
4. Validations are performed by preserving data locality.
5. Allows effective surveillance of model performance. Our methodology allows continual validation of model performance. Periodically, it is possible to check model performance at different spatial extents. Furthermore, validation budgets for surveillance can be smaller and expended on spatial extents where there is greater uncertainty.

Spatial extents could be based on: census tracts, counties, states, etc. (though we assert that our methodology should be just as applicable to classification models as well).

Chapter 2

Related Work

Our methodology is broadly applicable to diverse scientific data management frameworks [1, 2]. These management frameworks could be based on DHTs [3], data sketches [4], in-memory systems [5, 6], hybrid systems [7], or those based on peer-to-peer grids [8]. Crucially, because the system is focused on content it is applicable in situations where data may not have ideal colocation properties. This effort is also synergistic with frameworks that train models [9–11] either where these are orchestrations are containerized, over physical machines, or over schemes that leverage the spatially explicit nature of these orchestrations [12, 13].

Spatio-temporal data contain attributes related to time and space dimensions. Voluminous spatio-temporal datasets come from a variety of disciplines such as sociology, atmospheric sciences, ecology, and social media. These observations are collected through a variety of mechanisms, such as continuous remote sensing from satellites, aerial imaging, and distributed observing stations and sensors. In the era of data-driven analytics, there is growing demand to build and deploy predictive models over spatio-temporal domains to discover interesting and previously unknown patterns over large-scale data can be used to extract actionable insights [14]. Doing so requires efficient storing, querying, and object modeling of geospatial data using system architectures that are capable of supporting specific user requirements [15].

Model validation is the process of determining the accuracy of a trained model by evaluating its accuracy against a testing dataset. Validation allows us to understand, compare, and interpret the performance of our trained models. Performance of an end-to-end model, in terms of its accuracy, is governed by a range of factors and models built over a set of labels with complex relationship often have significantly differing performance over different sections of the overall data domain [16, 17]. This variability in performance can be caused by factors such as class imbalance, sparse training data, noise or significant difference in the distribution of the training and test data-domains.

Overall model performance computed using typical validation techniques in such cases will not reflect the true performance over these smaller sub-regions [18].

Identification of sub-regions showing low prediction accuracy, i.e., sub-domains where the average loss is higher than a defined threshold, has been explored in various works through the application of statistical techniques [19–21]. They facilitate analysis of the model performance at a more granular level. Additionally, they help provide a measure of confidence in a model based on the region over which it is applied so that they can be applied with a more trust [22] and help interpret and explain the model decision. These techniques, however, either require domain expertise and/or are not designed to handle voluminous datasets, which would involve handling the scalability challenge of exploring a large number of potential sub-domains.

In their survey on machine learning model validation using correlated behavior data [23] Ferdinandy et al. have looked at ways to validate classification models used in ethological studies. These models were trained using data related to a particular animal species and validated using a different species. They have discovered that it is necessary to create a standardized set of best practices in evaluating and reporting results of behavioral classification models.

Machine learning is increasingly being applied to time series data, as it constitutes a better alternative to forecasting based on traditional time-series models. As stated by Matthias Schnaubelt's survey on using machine learning model validation [24], special methodologies such as forward validation schemes (schemes that keep the temporal order of observations) are required for evaluating time-series models.

High dimensional datasets with a small number of samples are used in neuroimaging, genomic, eye-tracking and other biomedical studies. Andrius Vabalas et al. have demonstrated that when training classification models using these datasets, the type of validation scheme used could introduce a bias which would lead to inaccurate assessment of the model [25]. They have shown that it is necessary to use multiple types of validation schemes to accurately assess a given model.

Machine learning models are designed using ScikitLearn [26], Tensorflow [27], Keras, and PyTorch [28], which are some of the most popular machine learning libraries. These libraries

differ in performance, difficulty of prototyping new models, deployment, and community support. Therefore, scientists utilize either of these libraries for model-building based on the problem that needs to be solved and the complexity of the relationship between the features involved [29, 30]. As a result, building a unified platform for model evaluation requires us to adapt and interpret these different frameworks, which rely on their own set of configurations.

IBM's PAIRs AutoGeo is a recent attempt at consolidating the end-to-end process of pre-processing diverse sources of geospatial data for models to be trained on, and allow cropping and selecting of spatial areas or points to validate the trained model on [31].

Chapter 3

Systems Architecture

In this section, we describe the architecture of the entire system, as well as the technology stacks used in the individual components and across the cluster. Our framework is implemented as an overlay containing a *Coordinator* overseeing and tracking many *Workers*, with a *Proxy* server sitting on top providing a RESTful API for cients. The high-level visual representation of this can be seen in "Fig. 3.1". As is standard in most distributed frameworks, GNU/Linux is being used as the baremetal OS. In our case, we specifically use AlmaLinux, a free-and-open-source distribution that's fully compatible with the latest Red Hat Enterprise Linux platform (8.5 at time of writing). We've chosen to implement our software stack in Python 3, since it provided the most seamless integration with diverse range of data science, machine learning, and analytical frameworks. This decision was made to support the maximum number of research applications as possible. While there is a native performance trade-off in terms of when compared to languages like Java or C++, many of the computational libraries like `Numpy` and `SciPy` compile to C code, giving us fast performance when it's really needed. Furthermore, Python supports most common object-oriented design patterns, allowing for solid Software Engineering principles to be implemented so future developers can easily extend our base functionality. As our inter-component communication framework, we used Google's Remote Procedure Call (gRPC). This gives us the flexibility of defining message and service types easily in `protobuf` files, and having efficient message marshaling as a result.

3.1 Proxy Server

The Proxy server serves a couple of different purposes, the first being to provide clients with a RESTful API to send HTTP/s requests to. Client requests are sent as HTTP multipart/form requests with 2 parts: a JSON *request* string and a *model* file. The JSON request string is very flexible, and supports a range of optional fine-grained controls. A minimal client request contains the fields

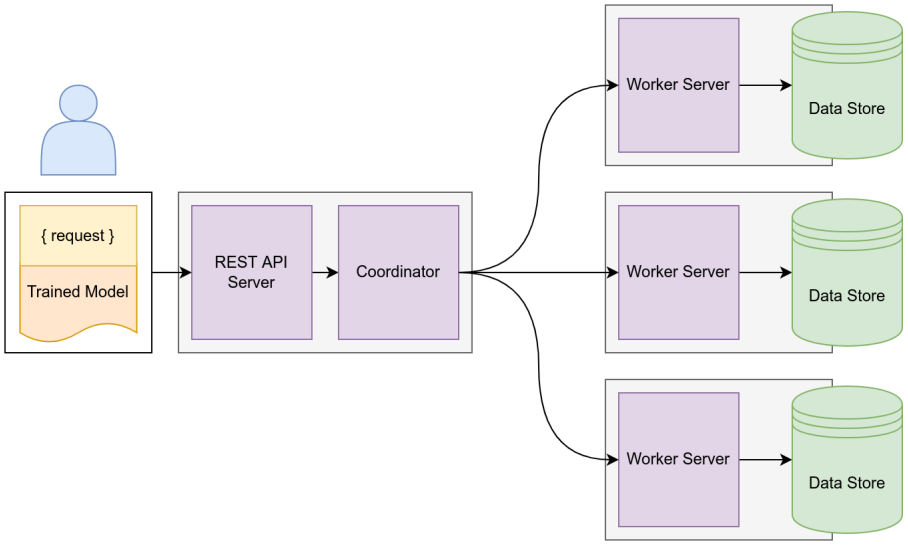

Figure 3.1: High level overview

shown in “Fig. 3.2”, but allows for many more knobs to be dialed like Worker or Coordinator job processing modes (See *Methodology, §E*), validation budgets (*Methodology, §B*), or even database read configuration preferences.

```
{
  "model_framework": "TENSORFLOW",
  "model_category": "REGRESSION",
  "database": "<database_name>",
  "collection": "<collection_name>",
  "feature_fields": [ "<feature_0>", "..." ],
  "label_field": "<label_field>",
  "normalize_inputs": true,
  "loss_function": "MEAN_SQUARED_ERROR",
  "spatial_coverage": "ALL",
  "spatial_resolution": "COUNTY"
}
```

Figure 3.2: Minimal client request

In addition to the request string, the input file is checked against a list of supported upload file-types based on the requested model framework, and extracted out as a stream of bytes. Both the request and the file bytestream are parsed into a custom gRPC message and forwarded to

the Coordinator server. While the Proxy server is currently relatively simple, it stands as a placeholder for planned future functionality like managing Role-based Access Control (RBAC) policies, user-specific budget allocations, load-balancing across different Coordinators, and even executing threat-detection models on uploaded files.

3.2 Coordinator

The Coordinator oversees an overlay of Worker nodes and keeps track of registered Workers and their respective metadata, including what data and specific spatial extents they store locally. This metadata is stored in a radix tree for fast lookups, insertions, and hierarchical aggregation for spatial extents at different resolutions. It also tracks ongoing jobs, and the completion statuses with respect to the individual Workers. The Coordinator endpoint itself is implemented as a gRPC server with multiple services like *RegisterWorker*, *DeregisterWorker*, *SubmitValidationJob*, etc. The services sit on top of a thread pool executor, allowing many incoming requests to be processed concurrently.

Upon receiving a gRPC validation job request from the Proxy, the Coordinator infers any necessary fields based on the validation budget specified, and routes it to the appropriate job execution function which handles load-balancing. Load-balancing is done using round-robin with respect to the spatial extents and resolution requested, and their locations on the tracked Workers. Individual jobs are created for each Worker with a subset of the spatial extents local to it, and the entire set of jobs is submitted to the Workers in a non-blocking, asynchronous fashion using Python's `Asyncio` library. A copy of the model is sent with each Worker job request, as well as many of the original request parameters specifying the data query configuration, model framework, etc. Similar to the communication between the Proxy and the Coordinator, communication between the Coordinator and Workers uses gRPC. The Coordinator's relationship to the Worker can be seen in "Fig. 3.3".

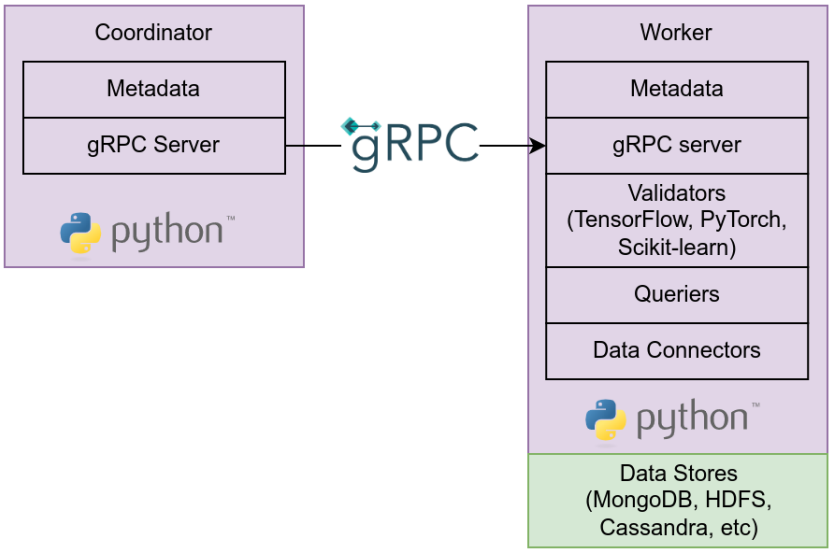

Figure 3.3: Coordinator-Worker component stacks

3.3 Worker

The Worker is responsible for the actual inference of the model on ground truth. Once a job containing spatial extents and a model to validated arrives, the Worker extracts the model bytes from the request and saves it as the correct file type to a directory specified by the unique job ID. It then creates an instance of a *Validator* with the request parameters, selecting the job mode (in most cases, this will be multiprocessing by default), and launching the validation job. If multiprocessing is requested, a shared executor is used with a fixed pool size to launch many inference jobs concurrently. The amount of simultaneous child processes that are forked in this case are tied to the number of physical cores available. Lastly, the child processes are recycled back into the shared executor which retains the imported libraries for future executions, greatly reducing overhead. Results collected for each spatial extent inference include variance corresponding to the selected loss function, the loss as requested, accuracy if it's a classification job, the total number of observations used, and internal metrics like the duration of the execution. Once all results have been collected, they are returned to the Coordinator which might apply a final aggregations or allocate a new budget for a final pass before routing the results back to the client.

We refer to our wrappers around individual modeling frameworks as *Validators*, which implement a common interface for validation making these modules extensible. Frameworks currently supported are TensorFlow, PyTorch, and Scikit-learn, but this list can be easily expanded in the future. Under the hood, Validators use the modeling framework library APIs to actually load and execute the model, but this first requires data to be loaded for a given spatial extent. This brings us to what we call *Queriers*, essentially our implemented wrappers around data connectors like `PyMongo`. Just like before, we implemented this as a generalized interface that allows new connectors to be supported in the future. Using a Querier, data can be loaded into efficient formats using PyTorch Tensors, Pandas DataFrames, or Numpy Arrays to be fed into the model, and similarly to calculate the aforementioned loss and variance estimates.

Chapter 4

Methodology

4.1 Dataset Staging and Sharding

In order to assess the performance a model efficiently, it is critical to achieve data locality to minimize or eliminate network I/O. In many cases, geospatial datasets are too large to be contained on a single machine. Instead, the data must be fragmented and distributed across many machines and their disks, but this poses the problem of data retrieval for model evaluation, since data have to traverse the network to be fed into the model. This approach is inefficient, since it pulls large amounts of data to the model which is small in size. The concept of data locality addresses this problem by pushing the computation (model) to the data, where it is evaluated directly on top of the data fragment in memory or on disk. With geospatial datasets, our approach is to group records within spatial boundaries together, then distribute these groups evenly across a cluster. For an administrative boundary like counties, this would look mean grouping records together by county and balancing the counties across the storage machines.

Records which come in a gridded format with just a latitude/longitude must be associated or tagged with the administrative boundary in which it lies. We accomplish this by using the shapes of counties, states, or census tracts to query which one the record's latitude/longitude pair falls into. Once the records have been tagged with their associated boundary, they can be grouped together as a single data shard and placed in a round-robin fashion around the cluster. In this last step, metadata can be recorded about where that data was placed in order to assist with later spatially-targeted computations.

4.2 Validation Budgets

In this section we describe our methodology for determining how to reduce the compute and I/O load for a given job. Remember that the purpose of model evaluation is to understand the *true*

error structure of a model, as given. Visualizing the error structure via choropleth maps gives a user the ability to quickly spot where their model performs poorly, as seen by both the variance and loss maps in "Fig. 4.1". Unfortunately, with voluminous data, this can be an infeasible task especially when under time or cost constraints. We attempt instead to *estimate* the error structure of a model by using only a sample, or subset, of the entire ground truth dataset. The sample size, called the *validation budget* (denoted *n*), is a total number of records a model is allowed to be evaluated on and can be chosen based on available data size, compute resources, time/cost constraints, or any other factors that come into play with inference-as-a-service. This validation budget is allocated in such a way that maximizes our understanding of the model's true performance across all geospatial extents, while minimizing the cost of achieving these estimates.

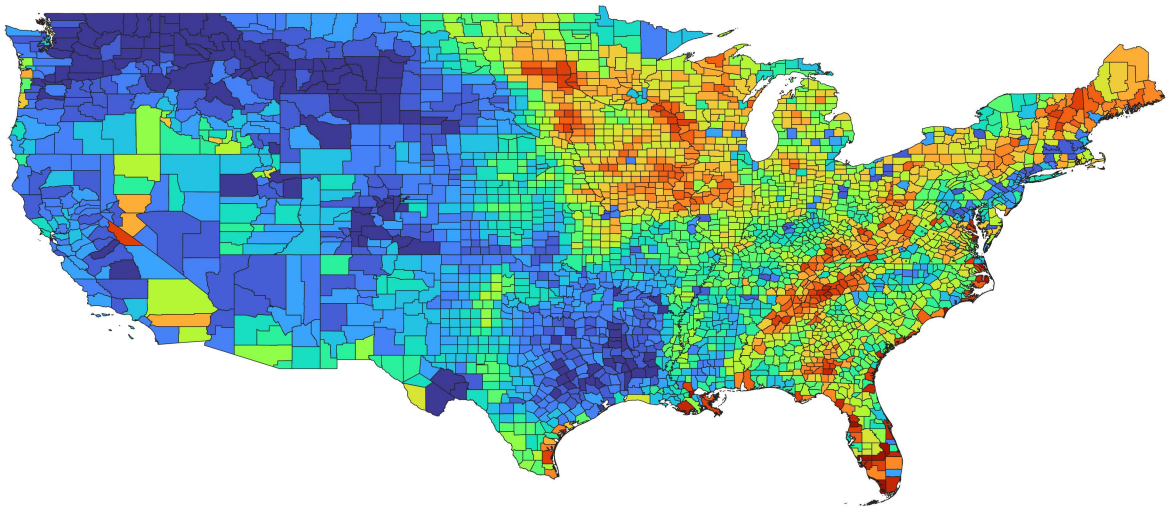

Figure 4.1: County choropleth maps for true model loss. Cooler colors are lower loss values, and hotter colors are higher values.

To explain this in terms of stratified sampling, let $D = \{1, 2, ..., N\}$ denote a database of N units, g_k denote ground truth for unit k, and m_k denote a model prediction that could be computed for unit k. Suppose that the database is divided into H mutually exclusive and exhaustive strata, $D = \cup_{h=1}^{H} D_h$ where the size of D_h is N_h and $N = \sum_{h=1}^{H} N_h$. In our example above, counties would be the strata. We wish to compare model output to ground truth using a total validation budget of $n = \sum_{h=1}^{H} n_h$ units, with a sample $d_h \subset D_h$ of n units randomly selected from the N_h units

in stratum h. If the goal is to understand the mean squared error (MSE) over the entire database, then we use $MSE = N^{-1} \sum_{h=1}^{H} \sum_{k \in D_h} (g_k - m_k)^2$. An unbiased estimator of MSE based on the stratified random sample is $mse = N^{-1} \sum_{h=1}^{H} \sum_{k \in D_h} (g_k - m_k)^2 (n n_h^{-1})$. Furthermore, an optimal allocation would make the variance of mse as small as possible subject to the validation budget, n. In some cases, it might make sense to consider unequal costs c_k of model evaluation in different strata, so that the validation budget would be replaced by the total cost $\sum_{h=1}^{H} c_h n_h$, which reduces to the validation budget n if all costs equal one.

A naïve approach would be to use equal allocation. This would assign $n_h = nH^{-1}$, which effectively breaks the total budget into equal size chunks to assign to every stratum. We will refer to this allocation scheme as the **equal allocation budget**, and have implemented this as a choice for users. A slightly better approach would be proportional allocation, which would assign $n_h = nN_h N^{-1}$, rounding to integers if necessary. This scheme is referred to as the **proportional allocation budget**, and is implemented as a sampling rate of the underlying strata. Various compromises between equal and proportional allocations begin with a minimum allocation in every stratum, say n_0, then allocates the remaining sample $n - Hn_0$ in proportion to a power of the stratum size: $n_h = n_0 + cN_h^a$ where $a = 1/2$ or $1/3$ are typical choices. These allocations have the advantage of "protecting" the smaller strata by not making large strata overly precise. While both of these allocations are simple, none yield optimal information about overall MSE if either the costs are unequal across strata or the prediction errors have different behavior in different strata. In the case of our county example, many may have wildly different sizes, and the model will likely perform differently based on the data within.

An improvement over the simple allocation schemes would be to proportionally allocate the budget based on the model's empirical variance. This tells us where the model is predictable, and where it varies based on the input data. We write the empirical variance of the squared prediction errors in stratum h as $S_h^2 = \{\sum_{k \in U_h} e_k^2 - \frac{(\sum_{k \in U_h} e_k)^2}{N_h}\}/(N_h - 1)$, where $e_k = (g_k - m_k)^2$. Obviously, e_k must reflect the loss function used, so if we were using the MAE to validate the model, then $e_k = |g_k - m_k|$. The optimal allocation subject to the cost constraint is then easily shown to

be $n_h \propto \frac{N_h S_h}{\sqrt{c_h}}$. When costs are constant across strata, this is known as the Neyman allocation. The optimal allocation reduces to proportional allocation if costs and variances are constant across strata, and reduces to equal allocation if costs, variances, and stratum sizes are all identical. In general, though, the optimal allocation assigns more samples to larger, more variable, and cheaper strata.

We next introduce a simple model to illustrate the behavior of the optimal allocation. Suppose that the model prediction errors satisfy $g_k - m_k = \mu_h + \tau_h \varepsilon_k$ for $k \in D_h$ all , where μ_h is the model bias in stratum h, τ_h^2 is the prediction error variance in stratum h, and the ε_k are independent and identically distributed normal random variables with mean zero and variance one, across all strata. Then assuming N_h values are large in every stratum, $S_h^2 \simeq Var((g_k - m_k)^2) = Var((\mu_h + \tau_h \varepsilon_k)^2) = Var(\mu_h^2 + 2\mu_h \tau_h \varepsilon_k + \tau_h^2 \varepsilon_k^2) = 4\mu_h^2 \tau_h^2 + 2\tau_h^4$, using properties of the normal distribution. The optimal allocation would then assign more samples to strata with high model bias, high prediction error variance, or both.

In practice, S_h^2 is unknown, since that would require evaluating the model on the full dataset. In this case, we first assign some of the budget to obtain initial estimates s_h^2 of S_h^2 in every stratum, using a simple allocation such as equal allocation of n_0 units in every stratum h, then use the approximately optimal allocation of the remaining sample $n - Hn_0$, plugging in the estimated s_h in place of the unknown S_h. This gives us some idea of how close we are to understanding its true error structure had it been evaluated on the full dataset (population, instead of sample). The proportional allocation equation used for a given stratum h is $n_h = \frac{(n - Hn_0)s_h}{\sum_{k \in s} s_k}$. Using this strategy, we are able to proportionally hand out the remaining samples of the budget, based on the estimated variances, for a final evaluation round. We call this allocation strategy the **incremental variance budget**. When using this budget, it's important to capture both the variance for the initial allocation *and* the variance for the new allocation together.

To provide an example of this, let's say a stratum h is evaluated with $n_0 = 2000$ observations, resulting in some variance s_h, and is then allocated an additional $n_h = 500$ records for re-evaluation. After the new evaluation, it wouldn't be sensible to throw away the variance re-

sulting from the initial run in favor of the final evaluation variance; the first probably had a better estimate of the population variance for h! This leaves us with two options. The first option is to re-evaluate the original 2,000 records plus the additional 500, giving us a sample variance for the full allocated 2,500. However, stratum h was not allocated 2,500 records for the second go-around; it was only allocated 500. This would be exceeding the total budget, and incurring more resource usage than originally assigned. A second and better approach that we have implemented is to use the Welford's method, which is a single-pass method for calculating the online variance across allocation iterations by looking at the differences between the sum of squared differences for N and $N - k$ samples. Not only does it completely eliminate the need for re-evaluating the initial variances (or saving them somewhere), it's also more numerically stable [32] and only requires each stratum to save three variables throughout all iterations: $\bar{x}$, s_x^2, and N throughout all iterations.

After evaluating global performance with this approach, we noticed that this helps us better understand the model *as a whole* given the total validation budget, but a drawback is that strata with high variability do not get a large enough share of the remaining budget in order to effectively improve their local estimate of the model's variance (S_h) and error (MSE_h), had the model been evaluated on the stratum population. We quickly realized that when s_h is low after the initial allocation round, that means we already have a good idea of how that model behaves for that stratum, and do not need to further allocate any more observations for a final round from the remaining budget. Sorting the variance estimates s returned from the initial round, we can see they follow a normal distribution. Thus, we place a threshold such that only variance estimates greater-than-or-equal-to two standard deviations above the mean of the variance estimates are considered for the Neyman proportional allocation, otherwise that stratum's evaluation is considered complete. In the example shown by "Fig. 4.2", only the counties with variance estimates above the red threshold would be proportionally allocated the rest of the budget. This has the benefit of reserving the entire remaining validation budget for the strata with high variance estimates, greatly improving their estimate of the true model variance and loss after the final evaluation round. It also has the benefit of reducing the computational load, since only a fraction of the strata are re-evaluated,

kicking off fewer jobs which in turn incurs less processing overhead. Additionally, this parameter can be tuned or removed altogether to meet user requirements for specific needs.

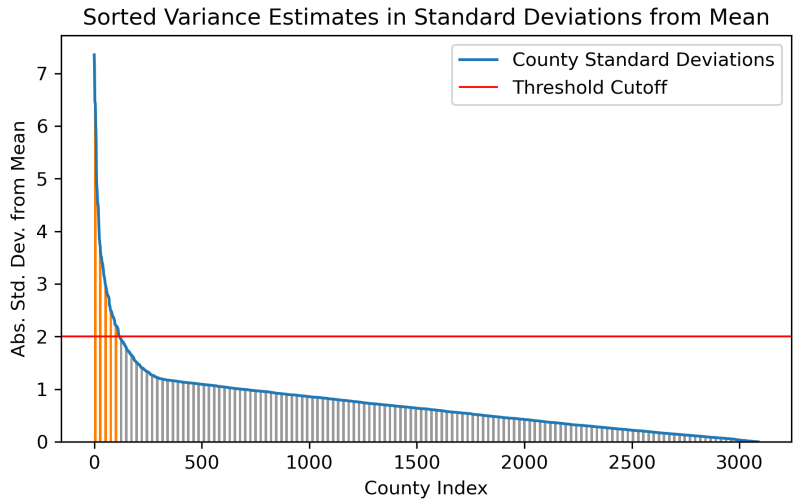

Figure 4.2: Variance estimate allocation threshold. Only counties above 2 std. dev. from mean, shown in orange, are chosen for further examination. The other counties, shown in gray, are considered complete.

4.3 Models as Black Boxes

The term "model" has recently become a heavily overloaded term; here we mean any function or set of functions that take input(s) and produce output(s) which can be compared against ground truth to produce a residual. Our goal is for the inference service to be agnostic to the contents of the model. Instead, we only care about how inputs are fed into the model, how outputs are retrieved, and how to evaluate the results against the spatial extent's ground truth. The format or type of the model *does* matter, as it determines how the model is loaded and executed, but beyond this, a user may provide an arbitrarily complex model so long as they specify how to use it (for example, a model saved using TensorFlow is loaded/executed in a completely different way than a model saved from PyTorch).

In the initial stages of this project, support for the most popular machine learning frameworks is added, and expanded to more niche frameworks or user-defined options to accommodate a diverse range of user-submitted model options. Investigating which frameworks to support in the initial

stages reveals a disparity between framework usage in industry versus research and academia. We opted to target the research field, since that's our target user-base. A recent survey which scraped all the papers (and their respective GitHub repositories) published at the top computer vision, NLP, and general ML conferences from 2017 to the end of 2019 shows that TensorFlow/Keras and PyTorch make up for the vast majority of ML projects [33–35]. Another survey showed Scikit-learn/SciPy being used in over half of the scraped ML projects in 2019, totaling 110,000 open-source projects [36,37]. These three model frameworks, PyTorch [28], TensorFlow [38], and Scikit-learn [26] are the first we have implemented support for black-box model inference on.

Submitting a black-box model for validation is simplified through model request parameterization. Everything from the input that is retrieved and fed into the model, to how to evaluate the model's performance can be specified through the request's JSON parameters. In a request, a user must minimally specify the data source, spatial resolution, input features, output labels, and type of model. The backend framework infers any other needed information, and performs end-to-end the model evaluation job.

4.4 Scalability

We designed our framework from the ground up with scalability as our main concern. The ability to painlessly scale resources horizontally is critical in cloud computing and distributed systems. Throughout our implementation, we've employed techniques like remote component and data discovery, data replication, parallelism, asynchronous I/O, threadpooling, and multiprocessing, while paying close attention to use efficient message marshaling and remote procedure calls, as well as appropriate data structures for storing both metadata and computational values. Our overlay is deployed as a Coordinator-Worker relationship, with one *Coordinator* and many *Workers*.

Coordinator: Since there is only one Coordinator for many Workers, it is crucial that this does not become a bottleneck for the entire cluster. Ideally (and in our experiments), a Coordinator is on its own distinct node, not co-located with a Worker. To ensure the Coordinator has as light of a load as possible for a given inference job, we've designed it so that it is only responsible for

(1) keeping track of registered workers and metadata about them, and (2) splitting up an incoming request into multiple job requests which get load-balanced as requests to individual Workers.

The entirety of a Coordinator's state resides in-memory only; nothing is persisted to the disk. Additionally, Workers are responsible for registering themselves with the Coordinator, and providing all metadata through self-discovery such as what data/spatial extents they have stored on their local disks, and the size/counts of them. This relieves the Coordinator from having to initiate any data or Worker discovery procedures. As the cluster needs to scale, more Workers can be dynamically added or removed.

For load-balancing requests, a Coordinator maintains a radix tree with the individual node keys being the hierarchical spatial extent identifier (in our experiments, this was the state and county IDs), and the node values storing metadata about where the data for that spatial extent is located (Worker hostname, IP address, port, etc). This allows all search operations to be performed in $O(k)$, where k is the number of hierarchical prefixes the spatial ID space maintains. For the example with state and county, $k = 2$, since an ID for a county is prefixed with the state ID. Regardless, this is not a memory-intensive data structure, especially when used with the state/county/census-tract spatial triad which have roughly 50/3,088/85,058 IDs respectively. Metadata about the Workers was stored in an inverted fashion, as a hashmap from registered Worker hostname to an array of the spatial extent IDs they store. The load-balancing effectiveness is heavily dependent on the placement of data– obviously, if one machine stores the majority of the data, the Coordinator has no choice but to assign the majority of the inference workload to that machine.

When a request comes in to validate a model for all spatial extents at a given resolution, the Coordinator creates a set of Worker jobs W, each containing the subset of spatial IDs that Worker w has locally. In the case where each spatial extent is replicated multiple places, like a MongoDB replica set, the Coordinator uses a round-robin policy to assign each ID to the replica set member with the lowest number of IDs in their job. Part of the load-balancing responsibility means that the Coordinator infers any validation budget information from the user request, and translates it to individual job budget parameters. For example, if the incoming request had specified a spatial

resolution of "county" and an equal allocation budget with a total limit of 10M records, the Coordinator will divide the 10M budget by the number of known counties (3,088), and assign every county ID in the individual worker jobs a random-sample allocation of $\lfloor \frac{1E7}{3088} \rfloor = 3238$ records per county. Finally, the constructed jobs are concurrently submitted to their respective Workers using asynchronous I/O so no blocking occurs waiting for a submitted job to return. We should also note that only one copy of the submitted model is held in memory for the Coordinator while transmitting; since no writes are happening here, it can be simultaneously read by multiple *send()* threads which include it as a byte stream in the job requests to the Workers.

Workers: On startup, Workers discover the spatial extent IDs of the data available to them locally, storing this metadata in a radix tree similar to the Coordinator, and report this to the Coordinator in a registration request. In addition to spatial metadata, a Worker maintains a shared thread- and process-pool executor for handling incoming jobs. Multiple incoming jobs can be processed concurrently using multiple threads, and within a job, multiple child processes are forked to validate the model on each of spatial extents in the request. Since model validation is *both* CPU-intensive and I/O intensive, some performance was gained by using multi-threading, but only on the I/O side of things as Python's Global Interpreter Lock (GIL) prevents two threads from running simultaneously. Thus, forking child process allows each to have its own GIL, giving us drastic improvements in terms of both CPU and I/O concurrency. We created the size of the process pool to match that of the available CPU cores on the system, and ensured that the child processes were being recycled between validation runs to eliminate process creation overhead. For incoming Worker job requests, the bytes of the model are saved once to disk for persistence and to allow for loading by the model's framework from within the confines of a child process. One caveat to this is each child process needs to have its own copy of the model loaded for running inference on its stratum sample, so this incurs a memory overhead and might be a limiting factor for large models. The exact limitation is characterized by the size of the model, memory available, and number of child processes running concurrently: Given available memory a, model size m, and number of child processes p (which corresponds to the number of physical cores c), the total copies of models being

inferred concurrently must not exceed $\frac{a}{pm}$ without performance penalties. For example, with $a =$ 64GB of memory, $c = p = 8$ cores, a model up to $m =$ 8GB in size can be inferred concurrently without using disk swap-space.

4.5 Software Extensibility

The entire project was carefully implemented using Software Engineering principles to not only protect the codebase's maintainability, but allow for future features to be added, more frameworks to be supported, and original components to be extended for specific use cases. Since the project is implemented with Python 3, we used an object-oriented approach for both the Coordinator and Workers and the data structures therein. This allowed us to reuse common structures, which in turn decreases the risk of introducing bugs to the system by having copies of components "fall behind" in terms of interface updates.

Supported model inference frameworks (i.e. PyTorch, TensorFlow, Scikit-learn) and data retrieval frameworks (i.e. MongoDB) were implemented as *Validator* and *Querier* objects in their own respective Python modules. These objects follow the "Wrapper" design pattern and inherit common functions, helper methods, and member fields from an abstract superclass. Implementing common construction and execution function signatures maintains a consistent use pattern from the Worker's perspective. A concrete example of this with the Querier object is the `spatialQuery` function, which takes as arguments the spatial ID you want to query for, the feature fields and label field you wish to project out of the results, and an optional limit and sample rate. With different Querier implementations, the way this data is retrieved can be modified to use different connectors or sampling strategies all while staying transparent to the user of the interface. With this approach, a new framework can easily be added and supported via a new Python module with minimal changes to the Worker request routing. For example, one could easily plug in a supporting module for Apache HDFS as an underlying data store, or MathWorks' MATLAB as a new model inference framework.

4.6 Incremental Evaluation

In most cases, datasets do not stay static. Observations are added over time. When this is the case, models that have already been evaluated on previous data may become "stale", where their error structure as understood before is not reflective of how it behaves with newer unseen data. We plan to address this by incrementally allocating some additional budget for each of the spatial extents receiving new data to re-evaluate the model on. By persisting the variance, count, and mean of the values the model has been evaluated on so far, we can capture a new up-to-date variance that accurately estimates performance for both the old and new records together. This is a classic online strategy, and is implemented using Welford's method for calculating a running variance. Since only a couple of parameters have to be remembered for each spatial extent, instead of the entire set of residuals calculated thus far, it is a computationally inexpensive operation to keep model validation estimates current.

Chapter 5

Performance Benchmarks and Discussion

5.1 Experimental Setup

Our experiments were run on a cluster of 25 commodity machines, each with an 8-core Intel Xeon E5-2620 v4 CPU running at 2.10GHz, 64GiB of DDR3 RAM, and 5400RPM hard disks. Three of the 25 machines were set aside as a replica set to manage the sharded/replicated database configuration, and one of the machines was dedicated as the Master node. The remaining 21 machines all housed the data shards locally and ran the Worker processes, responsible for the actual model validation workloads. A sharded, replicated MongoDB cluster was set up across these machines, with WiredTiger as the storage engine, but any other distributed storage frameworks could have easily been used in its place like Apache Cassandra or HBase. Local *mongod* processes which had direct access to the shard data on disk were connected to by the Worker processes.

We trained both linear and deep neural network (DNN) regression models for each of the supported analytical engines (TensorFlow, PyTorch, Scikit-learn), using *soil temperature 0.1 meters below surface* as the label, and 10 features related to *wind*, *pressure*, *dewpoint*, and *temperature above the surface*, and a loose hyperparameter search to achieve optimal model performance for the entire experiment dataset.

5.2 Datasets

In our experiments, we use a 1-year subset of NOAA's [39] North American Mesoscale Forecast System (NAM) dataset from 2021. This dataset is downloaded as gridded-binary 2 (.grb2) files, containing a latitude/longitude record for each of the 12km grids in the observed North America range, and each file is for a 6-hour interval snapshot. With over 400 observed variables per 12km grid, this ends up being millions of observations for a single file. We selected 36 observed variables, resulting in just over 56,000 observations for a given file. With these obser-

vations, we tag the record with the county GIS ID (GISJOIN) it belongs to using a *$geointersects* query in MongoDB with county shapefiles, and ingest it into our sharded, replicated MongoDB cluster in BSON format. Even with a greatly reduced subset of observations and time-range, this resulted in over 120GB of records in MongoDB before replication. Finally, an index is created on the records' GISJOIN field, and the county shards are distributed evenly across our cluster's 21 Worker machines. With 3,088 counties, this results in the data just under 150 counties being located on each machine. An example of our load-balancing policy in action can be seen in "Fig. 5.1".

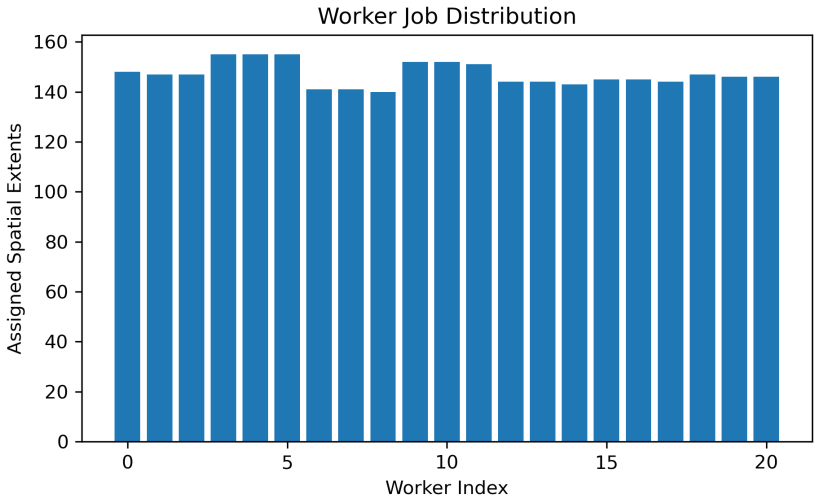

Figure 5.1: Assigned Worker spatial extent distributions, using a MongoDB sharded replica set with 3 replicas per shard.

5.3 Metrics

We first began by running validation jobs with *no budget*, which defaulted to using all available observations for every county. Three separate modes were used at the Worker for processing these jobs, synchronous, multi-threaded, and multi-processing. Both the total job duration and individual worker duration results can be seen in the "Fig. 5.2". As our baseline, some of the serially-executed jobs on the Workers took almost 9 minutes. Using Python's thread pool executor helped relieve I/O blocking, but did not achieve true concurrency due to the GIL. The biggest

performance improvement was found with using a process pool executor, where true parallelism was achieved both with CPU-bound and I/O-bound tasks. For the remaining experiments, we use multiprocessing as the default job mode.

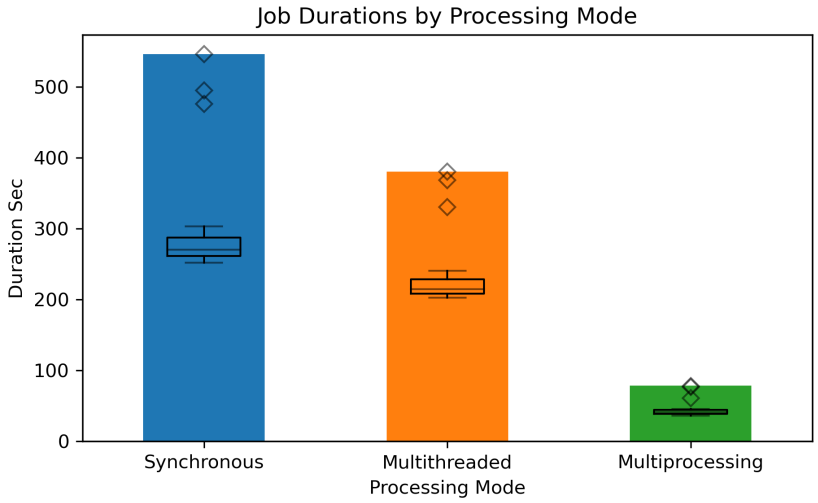

Figure 5.2: Job duration by processing mode (no validation budget). Bars denote total job duration, box plots denote individual Worker job durations.

Next, we captured cluster metrics for the duration of several jobs: memory, disk and network I/O, CPU usage. The next few figures are the results of **Benchmark A**, which uses a TensorFlow deep neural network as the model to validate, multiprocessing as the default Worker job mode, and four different executions, each with a different budget:

1. A *no-budget* (population) validation job, using all available observations,
2. A *proportional sampling* budget job that used a sample rate of 20 percent,
3. An *equal allocation* job that had 10M total records equally allocated to 3,088 counties,
4. An *incremental variance* job that had an initial allocation of 500 records to all counties, a total budget of 10M, using the allocation threshold of 2 standard deviations above the mean for initial variance estimates.

"Fig. 5.3" shows average Worker CPU usage across these four jobs. We can see around 50 percent of the CPU being used on average, due to some of the parallel child processes being in their data retrieval stage (I/O intensive), and others being in their model validation stage (CPU intensive). The variance budget job finishes its first validation round quicker than the other schemes, but has to complete a second pass using the remaining budget, hence the spike we see at around 25 seconds into the job. The memory usage from this benchmark can also be seen in "Fig. 5.4", where we see that the variance budget uses less memory than the other allocation schemes on average.

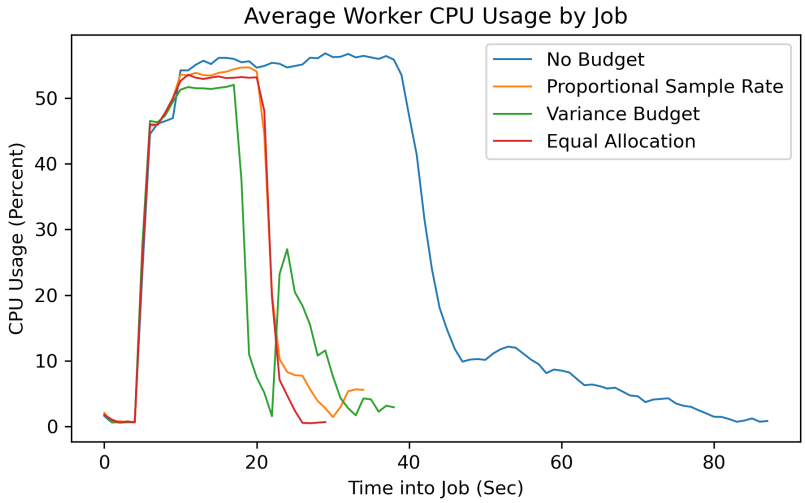

Figure 5.3: Cluster average Worker CPU usage, see budget descriptions for Benchmark A.

The total job durations can be seen in "Fig. 5.5", where we can see all budget types significantly reduce the total amount of time it takes to complete a job. Variance budgeting takes slightly longer than equal allocation or proportional sampling due to its second pass of the model on counties with high variability. While there is a small penalty to processing time, the second variance pass provides much better estimates of both the population loss and variances in the cases where the initial estimates have a high variance. This can be seen in "Fig. 5.6" where we compare population loss and variance to estimates provided by all three of the budgeting schemes outlined in Benchmark A. The final estimates provided by the variance budget are much more accurate to the

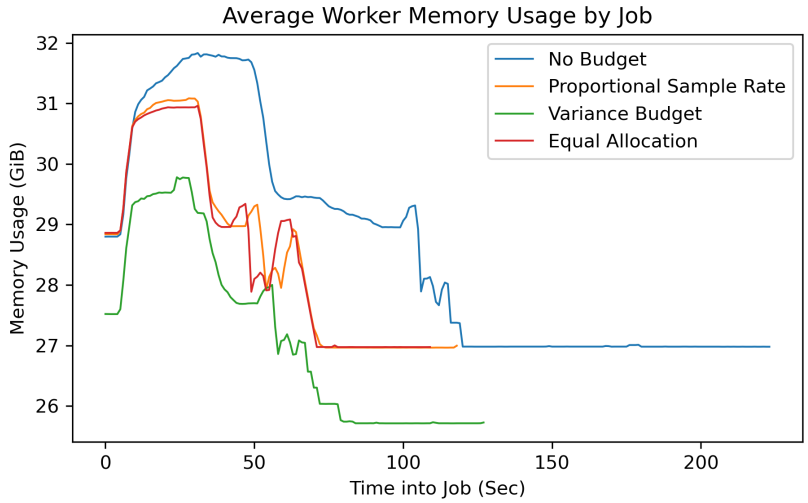

Figure 5.4: Cluster I/O and memory usage, see budget descriptions for Benchmark A.

population values than the estimates generated by the equal allocation strategy or the proportional sample rate strategy. With this said, however, we did note that over all 3,088 counties, the initial estimates that did not receive a second allocation from the remaining budget did not do as well as the proportional sampling, which was effective at gaining a decent overall picture of the model's error structure.

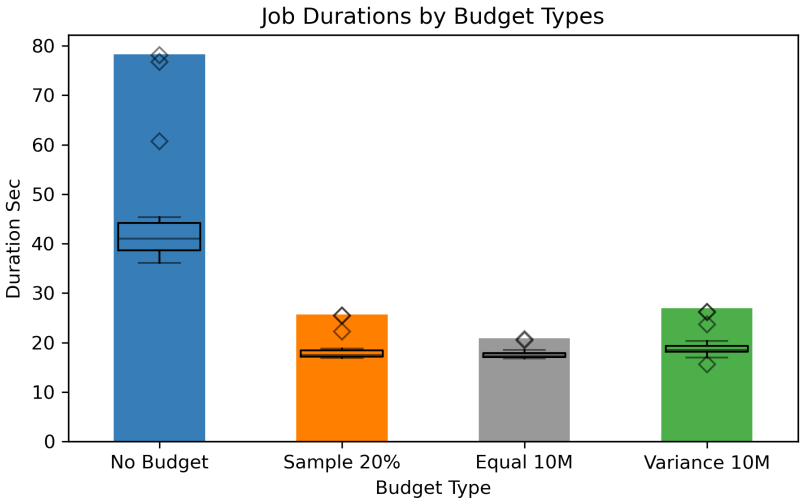

Figure 5.5: Total and Worker job durations by budget type, see description for Benchmark A.

To address the poor estimates by the counties with lower variances, we adjusted the incremental variance budget parameters for the initial allocation and the threshold cutoff. **Benchmark B** introduces two new executions:

1. An *incremental variance* job that had an initial allocation of 1000 records to all counties, a total budget of 10M, using the allocation threshold of 1 standard deviation above the mean for initial variance estimates.

2. An *incremental variance* job that had an initial allocation of 1000 records to all counties, a total budget of 10M, *with the allocation threshold completely removed*.

It also reuses the results from the equal allocation and proportional sample rate executions in Benchmark A for comparison. As shown in "Fig. 5.7", the equal allocation (bottom) and proportional sample rate (second from bottom) budgets estimate the error structure of the model generally well, but have many outliers counties which do not capture the population variance at all. The incremental variance budgets with thresholds (top three) specialize in "snapping" counties with abnormally high variances to the population variance, removing outliers, but generally struggle with counties that have lower variances. Removing the threshold (middle) gives us a happy compromise between no extreme outliers, and a decent estimate of the big picture model. A choropleth map of the estimated loss using the middle strategy is provided in "Fig. 5.8", which can be compared against the true population loss values for the experiment model in "Fig. 4.1".

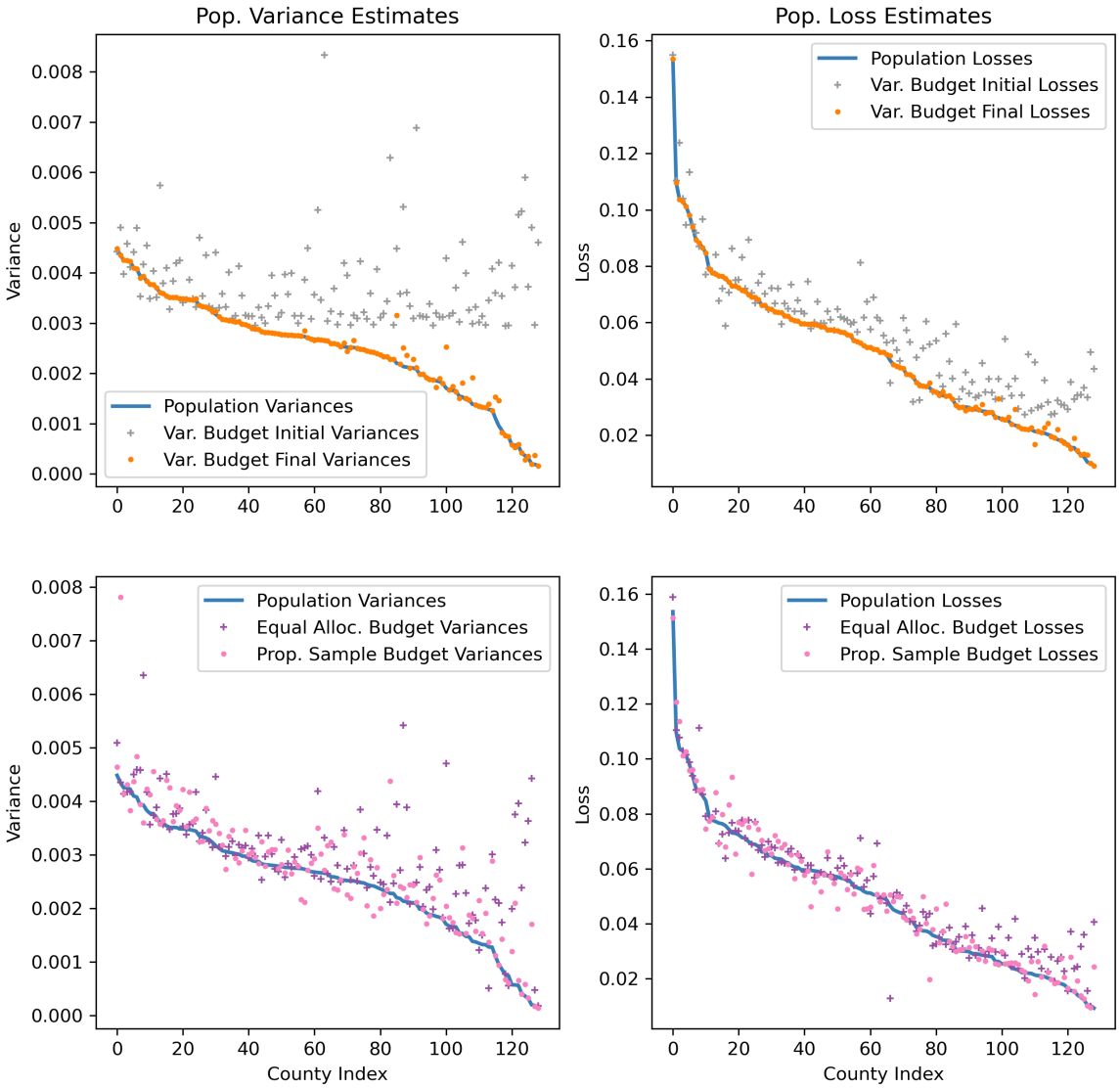

Figure 5.6: Loss and variance estimate accuracy against true population loss and accuracy, see description for Benchmark A. First column values sorted by population variances, descending. Second column sorted by population losses, descending. Only counties with high variances considered. *"Var. Budget. Initial"* denotes the initial variances of the variance budget pass that were found to be greater than 2 std. deviations away from the mean.

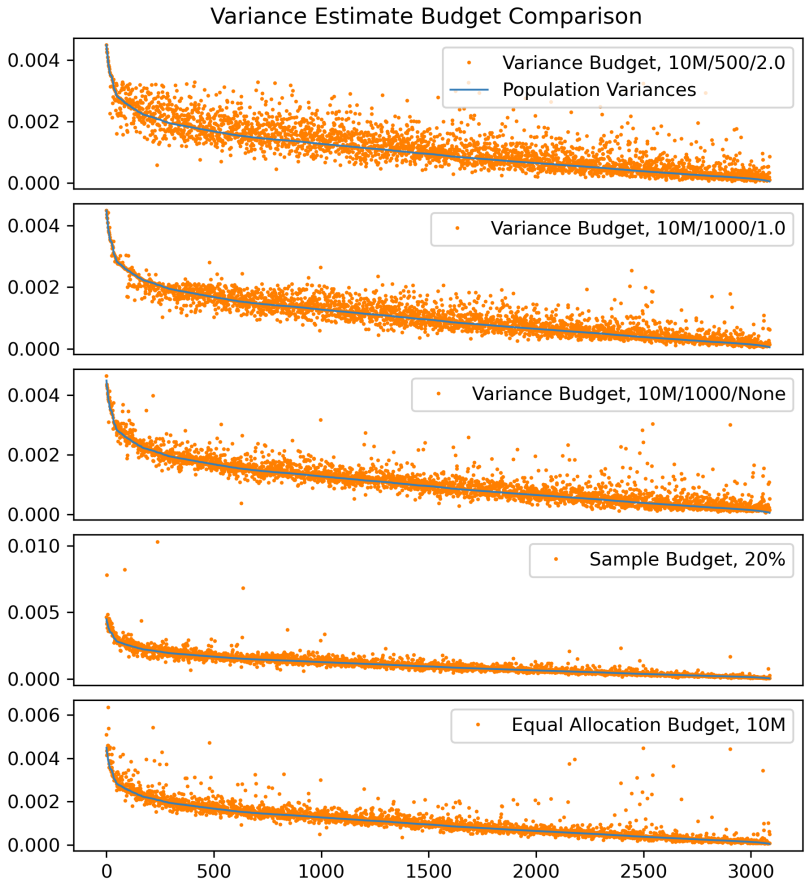

Figure 5.7: Variance estimates by budget type. For *Variance Budgets*: 3 numbers A/B/C denote total allocation/initial allocation/threshold in standard deviations from mean. Middle experiment has threshold completely removed.

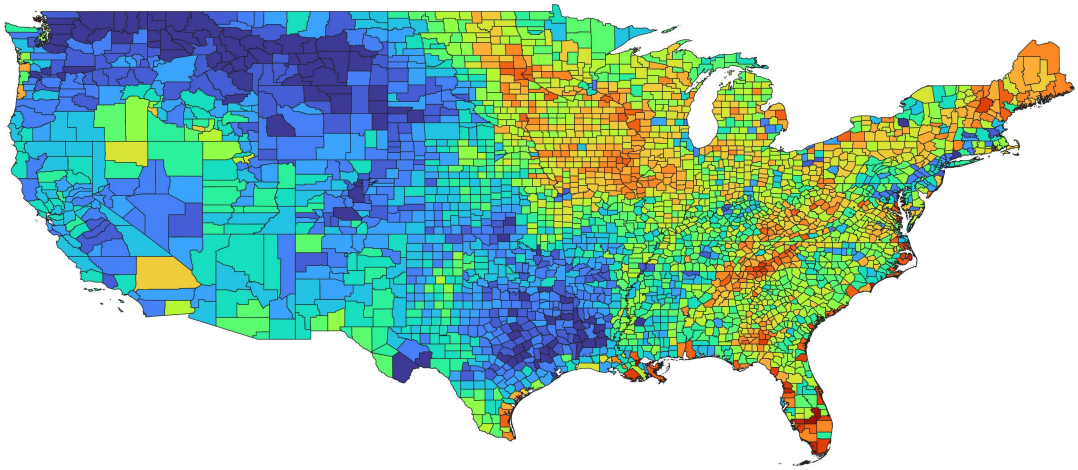

Figure 5.8: County choropleth maps for estimated model loss. Cooler colors are lower loss values, and hotter colors are higher values.

Chapter 6

Conclusions and Future Work

Here we described our methodology to support model validations at scale. Our benchmarks demonstrate the suitability of our methodology to facilitate model evaluations that are resource efficient and timely.

RQ-1: Using a validation budget allows us to ensure spatial coverage while conserving resources (both computing and I/O) that are expended during model validations. Our methodology allows us to estimate the error structure of the model M without having to rely on validations involving a large number of observations.

RQ-2: To ensure scaling we complement our validation budget with several mechanisms such as data locality, data dispersions based sharding schemes, and conserving memory by reducing the number of model instances that are memory resident. At each worker node, we also leverage thread pools and concurrency to ensure that processing cores available at a node are utilized effectively.

RQ-3: To ensure broader applicability, we treat models as black boxes without inspecting the internal structural properties of the models. For instance, the models we consider could be based on partial differential equations, decision trees, matrix multiplications and convolutions, etc. Our SE design pattern allows us to support assimilation of a wide range of analytical engines.

RQ-4: To characterize model performance we render visualizations of results as choropleth maps allowing users to explore spatial variations in model performance. The results from validation workloads can be streamed incrementally, and the choropleth maps refined as the data become available.

As part of future work, we propose to incorporate support for models based on classification and clustering. In the case of classifications, we will explore interactivity of our choropleth maps to contrast the AUC of the ROC at different spatial extents for different classifier thresholds. The challenge in the case of clustering is determining the feature vector associated with each spatial extent (regardless of their granularity) so that they may be clustered and rendered effectively. We

also propose to supplement these with a model surveillance scheme to continuously assess the performance of models as new observations are made available.

Bibliography

[1] S. L. Pallickara, S. Pallickara, and M. Pierce. Scientific data management in the cloud: A survey of technologies, approaches and challenges. *Handbook of Cloud Computing*, 1, 2010.

[2] S. L. Pallickara, S. Pallickara, M. Zupanski, and S. Sullivan. Efficient metadata generation to enable interactive data discovery over large-scale scientific data collections. *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 573–580, 2010.

[3] M. Malensek, S. L. Pallickara, and S. Pallickara. Evaluating geospatial geometry and proximity queries using distributed hash tables. *IEEE Computing in Science and Engineering (CiSE)*, 16(4):53–60, 2014.

[4] T Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2566, 2017.

[5] S. Mitra, P. Khandelwal, S. Pallickara, and S. L. Pallickara. Stash: Fast hierarchical aggregation queries for effective visual spatiotemporal explorations. *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[6] D. Rammer, T Buddhika, M. Malensek, S. Pallickara, and S. L. Pallickara. Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines. *IEEE Transactions on Big Data*, 8(1), 2022.

[7] M. Malensek, S. L. Pallickara, and S. Pallickara. Minerva: Proactive disk scheduling for qos in multitier, multitenant cloud environments. *IEEE Internet Computing*, 20(3):19–27, 2016.

[8] G. Fox, S. Lim, S. Pallickara, and M. Pierce. Message-based cellular peer-to-peer grids: foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21(3):401–415, 2005.

[9] Menuka Warushavithana. Containerization of model fitting workloads over spatial datasets. Master's thesis, Department of Computer Science, Colorado State University, Fort Collins, CO, 2021.

[10] M. Warushavithana, C. Carlson, S. Mitra, D. Rammer, M. Arabi, J. Breidt, S. L. Pallickara, and S. Pallickara. Distributed orchestration of regression models over administrative boundaries. *EEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*, 2021.

[11] M. Warushavithana, S. Mitra, M. Arabi, J. Breidt, S. L. Pallickara, and S. Pallickara. A transfer learning scheme for time series forecasting using facebook prophet. *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 809–810, 2021.

[12] D. Rammer, K. Bruhwiler, P. Khandelwal, S. Armstrong, S. Pallickara, and S. L. Pallickara. Small is beautiful: Distributed orchestration of spatial deep learning workloads. *Proceedings of the IEEE/ACM Conference on Utility and Cloud Computing*, 2020.

[13] S. Mitra, M. Warushavithana, M. Arabi, J. Breidt, S. Pallickara, and S. L. Pallickara. Alleviating resource requirements for spatial deep learning workloads. *The 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. (CCGrid 2022)*, 2022.

[14] Shashi Shekhar, Zhe Jiang, Reem Y Ali, Emre Eftelioglu, Xun Tang, Venkata MV Gunturi, and Xun Zhou. Spatiotemporal data mining: A computational perspective. *ISPRS International Journal of Geo-Information*, 4(4):2306–2338, 2015.

[15] Xiaoyu Wang, Xiaofang Zhou, and Sanglu Lu. Spatiotemporal data modelling and management: a survey. In *Proceedings 36th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Asia 2000*, pages 202–211. IEEE, 2000.

[16] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. Slice finder: Automated data slicing for model validation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1550–1553. IEEE, 2019.

[17] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th international conference on intelligent user interfaces*, pages 126–137, 2015.

[18] Josua Krause, Adam Perer, and Kenney Ng. Interacting with predictions: Visual inspection of black-box machine learning models. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 5686–5697, 2016.

[19] Minsuk Kahng, Dezhi Fang, and Duen Horng Chau. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–6, 2016.

[20] Zheguang Zhao, Lorenzo De Stefani, Emanuel Zgraggen, Carsten Binnig, Eli Upfal, and Tim Kraska. Controlling false discoveries during interactive data exploration. In *Proceedings of the 2017 acm international conference on management of data*, pages 527–540, 2017.

[21] Jae Won Lee, Jung Bok Lee, Mira Park, and Seuck Heun Song. An extensive comparison of recent classification tools applied to microarray data. *Computational Statistics & Data Analysis*, 48(4):869–885, 2005.

[22] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[23] Bence Ferdinandy, Linda Gerencsér, Luca Corrieri, Paula Perez, Dóra Újváry, Gábor Csizmadia, and Ádám Miklósi. Challenges of machine learning model validation using correlated behaviour data: evaluation of cross-validation strategies and accuracy measures. *PloS one*, 15(7):e0236092, 2020.

[24] Matthias Schnaubelt. A comparison of machine learning model validation schemes for non-stationary time series data. Technical report, FAU Discussion Papers in Economics, 2019.

[25] Andrius Vabalas, Emma Gowen, Ellen Poliakoff, and Alexander J Casson. Machine learning algorithm validation with a limited sample size. *PloS one*, 14(11):e0224365, 2019.

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[27] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[29] Xingzhou Zhang, Yifan Wang, and Weisong Shi. {pCAMP}: Performance comparison of machine learning packages on the edges. In *USENIX workshop on hot topics in edge computing (HotEdge 18)*, 2018.

[30] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, and Valentino Zocca. *Python Deep Learning: Exploring deep learning techniques and neural network architectures with Pytorch, Keras, and TensorFlow*. Packt Publishing Ltd, 2019.

[31] Wang Zhou, Levente Klein, and Siyuan Lu. Pairs autogeo: an automated machine learning framework for massive geospatial data. *IEEE International Conference on Big Data*, pages 1755–1763, 2020.

[32] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[33] The state of machine learning frameworks in 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/. (Accessed on 04/11/2022).

[34] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Alvaro Lopez Garcia, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.

[35] Jeff Hale. Deep learning framework power scores 2018 | towards data science. https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a. (Accessed on 04/11/2022).

[36] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[37] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4), 2020.

[38] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[39] Russell S. Vose et al. Improved historical temperature and precipitation time series for u.s. climate divisions. *Journal of Applied Meteorology and Climatology*, 53(5):1232–1251, May 2014.