DISSERTATION

LEVERAGING ENSEMBLES: BALANCING TIMELINESS AND ACCURACY FOR MODEL TRAINING OVER VOLUMINOUS DATASETS

Submitted by

Walid Budgaga

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2020

Doctoral Committee:

Advisor: Shrideep Pallickara
Co-Advisor: Sangmi Lee Pallickara

Asa Ben-Hur
F. Jay Breidt

ABSTRACT

LEVERAGING ENSEMBLES: BALANCING TIMELINESS AND ACCURACY
FOR MODEL TRAINING OVER VOLUMINOUS DATASETS

As data volumes increase, there is a pressing need to make sense of the data in a timely fashion. Voluminous datasets are often high dimensional, with individual data points representing a vector of features. Data scientists fit models to the data—using all features or a subset thereof—and then use these models to inform their understanding of phenomena or make predictions. The performance of these analytical models is assessed based on their accuracy and ability to generalize on unseen data. Several existing frameworks can be used for drawing insights from voluminous datasets. However, there are some inefficiencies associated with these frameworks including scalability, limited applicability beyond a target domain, prolonged training times, poor resource utilization, and insufficient support for combining diverse model fitting algorithms.

In this dissertation, we describe our methodology for scalable supervised learning over voluminous datasets. The methodology explores the impact of partitioning the feature space, building models over these partitioned subsets of the data, and their impact on training times and accuracy. Using our methodology, a practitioner can harness a mix of learning methods to build diverse models over the partitioned data. Rather than build a single, all-encompassing model we construct an ensemble of models trained independently over different portions of the dataset. In particular, we rely on concurrent and independent learning from different portions of the data space to overcome the issues relating to resource utilization and completion times associated with distributed training of a single model over the entire dataset. Our empirical benchmarks are performed using datasets from diverse domains, including epidemiology, music, and weather. These benchmarks demonstrate the suitability of our methodology for reducing training times while preserving accuracy in contrast to those obtained from a complex model trained on the entire dataset. In particular,

our methodology utilizes resources effectively by amortizing I/O and CPU costs by relying on a distributed environment while ensuring a significant reduction of network traffic during training. Finally, I deeply express my sincere appreciation to those who have contributed to this thesis and supported me in one way or the other during my study journey.

ACKNOWLEDGEMENTS

First and foremost, I am extremely grateful to my parents, Saeed and Sabah, who have been a constant source of strength and inspiration to me in my life, especially during the difficult moments I am facing in my study period. I would also like to thank my wife, Areij, for the support and assistance in every step of my thesis. At the most, I would like to thank my sons, Malak and Yousef, for their patience during the busy moments of my study.

I would like to express my sincere gratitude to my advisor Shrideep Pallickara and co-advisor Sangmi Pallickara for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better mentors for my Ph.D. study. Besides, I would like to thank the rest of my doctoral committee: Asa Ben-Hur and Jay Breidt for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also go to the members of the Distributed Systems Group and Big Data Lab for being a source of great ideas and useful suggestions throughout my study journey. Also, I would like to take this opportunity to thank Wayne Trzyna, Paul Hansen, and Abhimanyu Chawla for all their cooperation and support during my time at Colorado State University. Finally, I would like to thank all those who have made any contribution in some way towards my Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Over the past several decades the volume of data has continuously and rapidly increased in different fields. According to a survey on big data [1], in 2011 the overall amount of data in the world had reached 1.8 zettabytes (a zettabyte is 10^{21} bytes). In that year, around 72 hours of videos are uploaded to YouTube every minute, and more than 10 petabytes of log data are generated in Facebook per month. Besides, the International Data Corporation predicted that there will be 44 zettabytes of data in 2020 and 175 zettabytes of data in 2025 [2]. These numbers illustrate the continuous growth in data volume.

A confluence of factors has contributed to the exponential growth in data volumes. Improvements in disk densities, capacities, and quality, along with their falling costs, have made it cheaper to store increasing amounts of data. Miniaturization and improvements in sensing equipment have led to a proliferation of observational devices and continuous sensing environments, such as the Internet of Things (IoT), that are prolific sources of data. The proliferation of consumer devices (smartphones, tablets, and so on) also adds substantially to increased data volumes. Simulations, sales tracking, and commercial data collection and harvesting activities including social media are other sources of data. [3, 4]

Big data also creates value for several sectors and across large scopes through deep analysis. Big data, which is not only massive in volume but also in variability, enables exploring hidden structures in data and improved decision making. That is, voluminous data offer opportunities to extract insights from them. These insights can be used to understand phenomena, inform planning, and make forecasts based on current conditions. Often these insights are based on leveraging algorithms that fit models to the data. Industries and a variety of organizations, such as health-care systems, show an interest in using big data to improve efficiency and quality of performance. A recent big data survey [1] reports that fully using big data can improve profit by 60% and help organizations save billions of dollars.

The model-fitting process involves multiple steps. Features are extracted from the data, pre-processed, and transformed. A key component of the model-fitting process is the iterative training process where weights are assigned to features, the weights incrementally are refined, and hyper-parameters that are associated with the model-fitting algorithm are tuned. Model performance is assessed by using cross-validation (by generating different folds of the dataset) or a separately held validation dataset. The model-fitting process underpins our understanding of how features interact with each other, the relative importance of features, and the influence a set of features on particular phenomena. A performant model can then be used as a surrogate for the data. Model fitting (or training) is time-consuming. However, once trained, the model evaluations that assess the influence of feature changes is performed fast.

A sampling approach can be used as an easy and fast solution for gaining insights from voluminous datasets. However, it becomes infeasible for datasets that are large in volume and variability. Typically, a complex model that is trained with massive data is more accurate than a simple model or a complex model that has been built on small training data [5]. While a simple model fails to learn well and capture interesting and hidden patterns in data, an excessively complex model that is trained on small training data will overfit and cause poor predictive performance on new unseen data examples. That is, voluminous data allow building highly accurate models.

Processing big data is always associated with different challenges including scalability and storage limitations [6]. The continuous growth in data volume requires an equivalent increase in computation and storage resources that are beyond the capacity of a single machine. A realistic solution for this issue involves employing a large number of commodity machines and using the accumulated computation and storage resources for addressing large-scale problems.

Processing big data on a distributed system involves a lot of effort toward providing efficient computations. Adding more machines expands the system capability, but it also increases the probability of failure. In such an environment, the failing of nodes is expected rather than exceptional. It is inefficient to repeat an entire expensive computation every time a single node fails. In addition, the system should be scalable so that adding more machines facilitates the orchestration of more

computations. Efficient resource management in a heterogeneous environment is a desirable feature for balancing workload and maximizing resource utilization. Depending on the problem that is being solved, the system should maintain a proper level of consistency to guarantee correctness that could be hurt in the concurrent executions.

To process voluminous datasets, several systems have emerged to hide all of the complexity that is associated with data processing in a distributed environment. Pregel [7], Dryad [8], and PowerGraph [9] are examples of frameworks that rely on data-flow graphs to enable the expression of computation dependencies. Some other frameworks have been developed to enable computations on streaming data such as Granules [10] and Storm [11]. Piccolo [12] is a framework that enables parallel computations in memory for problems that involve sharing states across nodes and iterations. Hive [13] and Pig [14] are some of the frameworks that have been built on Hadoop to provide an easy way for everyday users to perform many common tasks on Hadoop by using a high-level interface that hides Hadoop's complexity. In addition, some frameworks [15–18] have been specially designed to apply machine learning algorithms on large-scale data in a distributed environment.

Distributed systems that allow iterative computations can enable building a complex model on voluminous data. Usually, these systems adopt a data-centric approach to reduce bandwidth consumption and improve performance. Iteratively, the learning algorithm is locally applied to the data portions that are stored on distributed workers in parallel to find subsolutions that are synchronized and aggregated to update the global model. Optimizing the global model in such a manner involves exploiting the natural decomposability of the objective function over the training examples or, if possible, modifying the learning algorithm to become system friendly. Although employing the distributed system seems to be the most dominant approach to learning from large-scale data and building a complex model, there are some inefficiencies that are associated with it:

- The performance can severely suffer from high network traffic because all of the machines that are processing the training data have to fetch and update the distributed model's parameters, iteratively.
- It is an infeasible solution for problems that involve sequential training.
- It does not provide a support for leveraging and combining diverse learning methods for gaining insights from voluminous data.
- There is an expensive synchronization barrier between the refining steps. In this case, the progress is controlled by the slowest machine.
- Concurrent updating of the model's parameters can cause inconsistency, a state that can hurt the convergence of the training algorithm.
- Fault tolerance is a necessity in real-world deployments to deal with failed machines. Otherwise, some (or all) of the computations need to be re-executed from scratch.

1.1 Research Challenges

Our goal in this study was to enable learning from large-scale data in a short time without sacrificing predictive accuracy. Because training on an entire set of training data captures all of the patterns and relationships in one complex model, it is most likely that the complex model generalizes well and outperforms an equivalent model that is trained on a sampled data. Because we are dealing with continuously growing data, scalability is the main challenge and the root of many other challenging problems. Increasing the training data should only involve adding more machines, without a significant increase in the training time. Regardless of the data size, making the training time nearly consistent is a desirable feature, but it is also challenging.

We can maximize the prediction accuracy of a model by increasing the size of the training data [1, 19]. However, the training time will increase as the size of the training data gets larger. In this case, the data volume represents a trade-off between accuracy and training time. The challenge

here is to create a process that maximizes the prediction accuracy while keeping the training time small. Maintaining the complex model's accuracy with reduced training time involves the ability to extract the hidden patterns and relationships from the data in a short time. To accomplish this goal, we must targeted the following objectives:

- All training observations of voluminous data must be included in the training process.
- The training procedure must be issued in parallel without synchronization barriers.
- The network I/O must be minimized.
- The process must have the ability to extract the hidden structures and patterns that could appear as outliers in a smaller version of the data.

Fitting a global model on the entire dataset by using a distributed system limits the number of learning algorithms to be applied. It is infeasible to apply sequential algorithms (such as gradient boosting) on voluminous data that is hosted by multiple machines in parallel. Also, there is no easy and obvious way to allow applying several learning algorithms to the same problem toward achieving better prediction accuracy. Our goal is to allow applying an arbitrary algorithm in its natural way without the need for modification. Additionally, we need to support applying various algorithms to the same problem when they enable gaining more insights from the data.

1.2 Research Questions

The primary objective of this study was to facilitate the effective creation of regression models over voluminous data. The specific research questions that we explored included the following:

- RQ-1: *How can we reduce model training times?* Training times increase with increased data volumes, and our objective is to reduce model training times.
- RQ-2: *How can we ensure that this reduction in training times is not at the expense of accuracy?* We will contrast the accuracy of the models using our methodology with that of

a canonical model that was trained without regard for training times, and with a focus on accuracy.

- RQ-3: *How can we ensure that the methodology scales with increases in data volumes and the number of available machines?* Coping with the increase in data volume by adding extra machines must keep the training times nearly constant.

1.3 Approach Summary

The crux of this study is to ensure faster model creation/training over voluminous data. In particular, we focus on regression models. Our methodology targets: (1) the alleviation of I/O constraints, (2) the creation of independent models in a distributed environment, (3) ensuring data locality during the model-creation process, and (4) employing and combining several methods to preserve prediction accuracy.

Training a single all-encompassing model may involve network I/O and synchronization barriers for updating the model's parameters. This is because the data volumes are often far too big to be entirely resident on a single disk. In such cases, the learning algorithm is locally applied to data that is hosted by worker machines in parallel to find subsolutions that are synchronized and aggregated to update the global model. The process is iterative and continues until a stopping condition is reached.

Rather than build a single all-encompassing model, we simultaneously launch multiple training processes to fit several models. In our setting, data is partitioned and distributed over a collection of machines. Each model has data locality—the model is restricted to, and locally trained on, the data that are available on a machine, without data transmission involved. Each model instance trains independently of other instances, ranks features, performs feature selection, and so on. Individual model instances do not communicate with each other, nor are there any synchronization barriers that span model instances during their training.

Model creation leverages the distributed environment by ensuring that the training process executes concurrently on multiple machines; thus, model training proceeds concurrently. If there

are N models, and assuming that the data is distributed more or less evenly, then we can achieve an N-fold speed up in model training times. The key reason for the speedup is that the I/O and CPU costs that are associated with model training are amortized over the distributed collection of machines.

The created models are used to construct an ensemble whose behavior depends on how the training data were partitioned. The ensemble exploits the insights that are gained from the partitioned data and generates a prediction for a given observation. The final prediction is emitted based on the cooperation that occurs during the training phase.

1.4 Contributions

In this study, we describe our methodology for the fast creation of models over voluminous datasets while preserving the accuracy of these models. We validate our methodology in the context of datasets from multiple domains. The methodology is demonstrably scalable while preserving accuracy. Our specific contributions include the following:

1. Amortizing I/O and CPU costs by orchestrating model creation in a distributed environment.

2. Preserving the accuracy of the created models by considering all of the observations in the training process and combining the learned pieces.

3. Efficient resource utilization and scalability by relying on building independent models that do not require synchronizations barriers.

4. Ensuring simple and efficient fault tolerance by relaunching only the failed learning processes on new machines without affecting the remaining processes.

5. Combining the strength of diverse algorithms to solve a single problem.

6. Supporting algorithms working in a sequential manner—the methodology should allow applying the algorithms in their natural way, without the need to modify them.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we explore and summarize the related work. In Chapter 3, we show our motivating experiments that reveal the importance of learning from voluminous datasets and investigate the training of independent models. Chapter 4 introduces the proposed methodology and the involved tasks. Following this, we demonstrate the use of geospatial partitioning to enable scalable anomaly detection over spatiotemporal data streams in Chapter 5. Chapter 6 illustrates the applicability of the methodology for evaluating the epidemiological impact of livestock disease outbreaks at a national scale. Chapter 7 provides benchmark results for an extensive evaluation of ensembles that are constructed by using input- and output-based partitioning. Finally, we conclude the dissertation and show future research directions in Chapter 8.

Chapter 2

Related Work

In this chapter, we start explaining the iterative computation (§2.1) because it is the main component required in the learning process. Then, we show some relevant solutions (§2.2 and §2.3) that rely on several models for gaining insights from datasets. Finally, some state-of-the-art approaches (§2.4) that enable learning from large-scale datasets are discussed. Solutions for large-scale datasets are classified into three main approaches: (a) sampling, (b) divide and conquer approaches, and (c) distributed learning approaches. Besides enabling learning from large datasets, these approaches try to reduce training costs and time without sacrificing accuracy.

2.1 Iterative Computations

A broad spectrum of machine learning algorithms attempt to fit a model to a given dataset (training data). Model fitting, often called the learning process, usually involves iterative processing to boost the model's efficiency gradually. In the learning process, the model is evaluated on the training data, and based on the evaluation result the model's parameters are adjusted to improve its accuracy. This process involves several refining steps and multiple passes over the training data to reach the desired level of the model's accuracy. That is, the learning process improves the model's efficiency in an iterative manner. In each iteration, the current model's parameters and the training data are used to compute the new model parameters. The learning process employs an objective function (loss function) to measure the degree to which the ideal case is satisfied. There are many algorithms that are designed to refine the model's parameters and optimize the objective function. Gradient descent is a very commonly used optimization algorithm.

Gradient descent is a generic optimization algorithm that finds a model's parameters $\vec{w}$ that minimize a given loss function. Usually, it is used in the training process of supervised learning to refine the model's parameters towards reducing the prediction error of the training examples. In general, the algorithm optimizes the following objective function:

$$\min_{\vec{w}} \sum_{i=1}^{n} \mathcal{L}(h(\vec{x_i}, \vec{w}), y_i) + \lambda \mathcal{P}(\vec{w}),$$

where n represents the number examples in the training data, $\vec{x_i}, y_i$ are the i^{th} input feature vector and the associated dependent variable, respectively, $h()$ is the hypothesis function that uses the model's parameter $\vec{w}$ to predict the target, $\mathcal{L}()$ is the loss function that returns how close the predicted value is to the actual value, λ is a hyperparameter that controls the importance of the regularization term, and $\mathcal{P}(\vec{w})$: is the regularization term.

Each refining step uses the current state of the model's parameters to compute the new state. The employed learning method identifies the number of refining steps in each iteration that represents one pass over the full set of training examples. While online learning involves one refining step for each example in the training data, batch learning requires performing only one refining step after iterating over the entire set of training examples. Gradient descent is applicable in three different learning forms:

Batch Gradient Descent

Batch gradient descent is a batch-learning method that involves iterating over all of the training examples to make one update to the model's parameters. This approach is very slow to reach the neighborhood of the optimal solution, especially for large datasets. However, it can find the parameters that lead to high prediction accuracy. This learning method is parallelizable because the objective function is amenable to operating on distributed data and involves few communication messages to aggregate the model's parameters, which are refined on different data partitions.

Stochastic Gradient Descent

The stochastic gradient descent is tailored to enable applying the gradient descent algorithm to a large dataset with improved performance. Stochastic gradient descent is an online learn-

ing procedure that updates the model's parameters after each example in the training data. For this reason, the data must be randomly shuffled (reordered) before iterating over the training examples. This method can reach the neighborhood of an optimal solution very fast, but it often does not optimize the loss function error as well as in the case of batch gradient descent. Parallelizing this learning method is prohibitively difficult because it involves a large number of communication messages to update the global state of the model's parameters.

Mini-Batch Gradient Descent

The mini-batch gradient descent updates the parameter once for a fixed number of examples. This approach divides the whole dataset into small batches, each of which is used to make one update to the model's parameters. Mini-batch learning represents a trade-off between efficiency (online) and algorithm convergence (batch learning). Also, it provides a trade-off between efficiency and the number of refining steps, and it enables parallel implementation of the learning process. This approach allows processing all of the mini-batches in parallel on different machines with fewer communication messages compared with online learning.

2.2 Ensemble Methods

Ensemble methods rely on the creation of multiple models instead of a single model for a given problem. The models are usually trained on different subsets of the training dataset, with different settings, or by using various algorithms. Often, the individual models of the ensemble are trained with the same learning algorithm (referred to as the base learner). Such an ensemble is called a homogeneous ensemble. An ensemble whose individual models are trained with different base learners is referred to as a heterogeneous ensemble. The ensemble learning process for regression can be divided into three phases. The first phase involves ensemble generation, where the individual models are created. In the ensemble pruning phase, some of the created models are eliminated based on quantifiable exclusion criteria. The last phase is ensemble integration, where the models are combined to generate a final prediction for a given input features vector. It is well known that

an ensemble produces predictions that are more accurate and robust than any of its constituent models, and ensemble methods have shown great success in many real-world tasks. The most commonly used ensemble approaches are boosting, bagging, and stacking. Broader studies about ensemble methods and their applications can be found in [20–25].

2.2.1 Bagging

Bagging (**b**ootstrap **agg**regat**ing**) [26] builds ensembles with stable predictions by combining unstable models that are highly accurate for different parts of the input feature space and less accurate for others. It relies on bootstrap sampling to create several training sets that are used to build models with diversities. The bootstrap sampling performs random sampling with a replacement to create a new version of the original dataset [27]. It has been used for determining the confidence intervals of some statistic parameters and quantifying uncertainty by calculating standard errors and confidence intervals [28].

Bagging applies bootstrap sampling on the original dataset to create different training sets that are the same size as the original dataset. The base learner is employed to build complex models with low bias and high variance on the generated training sets. Bagging reduces the variance of the complex models in an ensemble. Bagging computes the final prediction for a given observation by averaging the predictions of the individual models to reduce the prediction variance, resulting in improved generalization. Bagging low-bias models that are not highly correlated reduces the variance without increasing the bias, which in turn reduces the generalization error [29]. Bagging relies on bootstrap sampling to reduce the correlation among the individual models. Another advantage of bagging is that the individual models of the ensemble can be trained independently and in parallel, resulting in reduced training time, especially for small datasets.

Random forest [30] is a well-known bagging approach that uses a relatively deep decision tree as a base learner to build complex models with a low bias to capture trends in data. Random forest improves the generalization ability over bagging by adding randomness during trees induction to decorrelate the tree models, resulting in better variance reduction and lower generalization error.

Random forest has already achieved great success in many regression and classification problems. For example, the authors in [31] compared random forest classifiers with support vector machines (SVMs) in terms of classification accuracy, training time, and user-defined parameters. It was found that both approaches show similar performance in terms of classification accuracy and training time. However, the number of user-defined parameters involved in random forest classifiers is less than the number that is needed for SVMs. The work in [32] shows that random forest classifiers show a robust performance even in the presence of noise in the training data, and they achieve similar performance to that of boosted and bagged decision-tree classifiers. Besides, it has been used for different domain problems. For example, it has been applied to classification problems in remote sensing [31, 33, 34], deep learning problems [35–38], and time-series problems [39–41].

2.2.2 Boosting

Boosting [42, 43] refers to a number of techniques that iteratively convert weak predictors to strong ones. A weak predictor is only somewhat effective at forecasting outputs, whereas a strong predictor will produce results with low relative error. Unlike bagging, boosting builds the models in a sequential stagewise fashion. Boosting methods assign each observation a weight and then update the weights at each iteration of the training process, based on the observed prediction errors. Weights that are associated with weak predictions are increased, while those that are associated with strong predictions are decreased. This process guides the subsequent training stages by shifting the focus of the models toward observations that were inaccurately predicted. In general, boosting methods are most applicable in situations where the base model is highly biased. Gradient boosting [44], an exemplar of a boosting ensemble, targets the optimization of a differentiable loss function, which iteratively increases the expressive power of the base prediction model. It employs shallow decision trees (low variance and high bias) and iteratively reduces the bias, resulting in a reduction of the relative prediction error.

2.2.3 Stacking

Stacking is a type of ensemble learning that builds an analytical model on the predictions of the other models to provide a prediction that is more accurate than any of the combined predictions are [45]. It involves two training phases. In the first phase, the training dataset is used to fit several models, referred to as first-level learners. Then, the first-level learners are applied to a given data to generate a new training dataset where the input features are represented by the learners' outputs, along with the original target values. In the second phase, the generated data are used to train a new model, referred to as second-level learner or meta-learner. Usually, the first-level learners are trained by using different learning algorithms, resulting in a heterogeneous ensemble. For a given features vector, all of the first-level learners are used to generate their predictions, which are then forwarded to the meta-learner that makes the final prediction. That is, the meta-learner takes as input the predictions that are made by the first-level learners and generalizes over them. Compared with a single model, building a stacked ensemble increases the complexity of the training process, which involves more computation resources but promises better prediction accuracy.

Using the same dataset to train both the first-level learners and the meta-learner increases the risk of overfitting [23]. For training the meta-learner, it is recommended to use test observations that have not been used to train the first-level learners. Usually, a leave-one-out or a cross-validation procedure is followed to create a training dataset for the meta-learner. Employing the leave-one-out procedure involves building the first-level learners several times in multiple phases. In each phase, one different observation is left out for testing and the rest are used for training the learners. The learners are then invoked on the test observation to generate one features tuple for the new training dataset. This procedure is very time-consuming because the first-level learners need to be built n times to generate a training dataset with n observations. For reducing the time complexity, a cross-validation procedure has been applied to generate the training dataset for the meta-learner.

Ting et al. [46] studied the conventional stacking method that was introduced by Wolpert [45] and tried to improve its performance by combining the confidence instead of the predictions of

the first-level learners for classification problems. For the creation of the input features in the new training set, the authors used the classes' probabilities that were generated by each of the first-level classifiers instead of a single predicted class. The class probabilities acted as the confidence measure for the prediction made. In their work, it was also found that multiresponse linear regression was the most proper algorithm to use to fit the meta-learner compared with a C4.5 decision tree learning algorithm [47], a reimplementation of a Naive Bayesian classier [48], and a variant of a lazy learning algorithm [49]. The authors reformulated the class label (the target) into m binary variables, where m was the number of the classes. The multiresponse linear regression algorithm was then used to train the meta-learner to predict the m variables. The value of these variables was either 1 or 0. In each observation, the variable that represented the observation's class was set to 1, and the other variables were set to 0. In such a case, the linear model makes regression predictions for all of the binary variables. Additionally, the authors could show that the use of multiresponse linear regression for the meta-learner and the other three algorithms for first-level learners led to better predictive accuracy than the majority vote and the selected model based on cross-validation did. Employing the predicted probabilities of each first-level learner increases dimensionality (*number of classes* $\times$ *number of classifiers*) in the new training dataset, which could also lead to overfitting, especially for quite small datasets.

Another work [50] has empirically shown that heterogeneous ensembles that are built by the stacking method that was proposed by [45] perform comparably to a single classifier that was chosen by cross-validation. In this work, the authors developed two separate extensions of the stacking approach that relies on probability distributions for training the meta-learner with multiresponse linear regression. In the first extension, the authors added two sets of input features to the training dataset that was generated for the meta-learner. The first set of features was created by multiplying the probability distributions by the maximum probability, and the entropies of the probability distributions represented the second set. The motivation behind adding the new features was to capture the certainty that the meta-learner can combine with the predictions information that is represented by probabilities to achieve better performance. In the second extension, the authors used multire-

sponse model trees as an alternative to multiresponse linear regression to train the meta-learner. The authors used 30 datasets to evaluate both extended approaches. The results showed that the latter extended approach performs better than the conventional stacking approaches and the best classifier selected from the stacking ensemble by cross-validation do.

The stacking method has been applied in different domains of machine learning and has shown an improvement in prediction accuracy and generalization. Ravi et al. [51] investi d the use of the stacking method for activity recognition, formulated as a classification problem, and contrasted its performance with single classifiers and other ensembles. The authors found that the plurality-voting approach outperformed the stacking ensembles. The authors in [52] introduced an online and dynamically weighted stacked ensemble for multi-label classification. Their proposed approach relies on spatial modeling to optimize the weights to the ensemble's classifiers. The conducted experiments have shown better performance with this model on most of the used datasets than that which was obtained with other prominent ensemble models. Ding et al. [53] proposed a stacking ensemble to enhance the performance of multi-label classification. The proposed ensemble relies on the **a**rtificial **b**ee **c**olony (ABC) algorithm [54] to search base classifier configurations and then sets the outputs of the chosen classifiers as inputs for a meta-classifier. Singh et al. [55] proposed a stacking-based ensemble for predicting the onset of Type-2 diabetes mellitus (T2DM) within five years. The authors used L-SVM, RBF-SVM, Poly-SVM, and decision trees algorithms to build the first-level learners. The authors employed NSGA-II, an evolutionary algorithm, to optimize the ensemble size and accuracy. The meta-learner was built using a *k*-nearest neighbor algorithm. The evaluation results have shown that the proposed approach significantly outperformed several individual machine learning approaches and traditional ensemble approaches.

2.3 Mixture of Experts Approaches

The **m**ixture **o**f **e**xperts (MoE) is a powerful combining method that shows great potential to improve the performance and reliability of machine learning models. It relies on the divide-and-

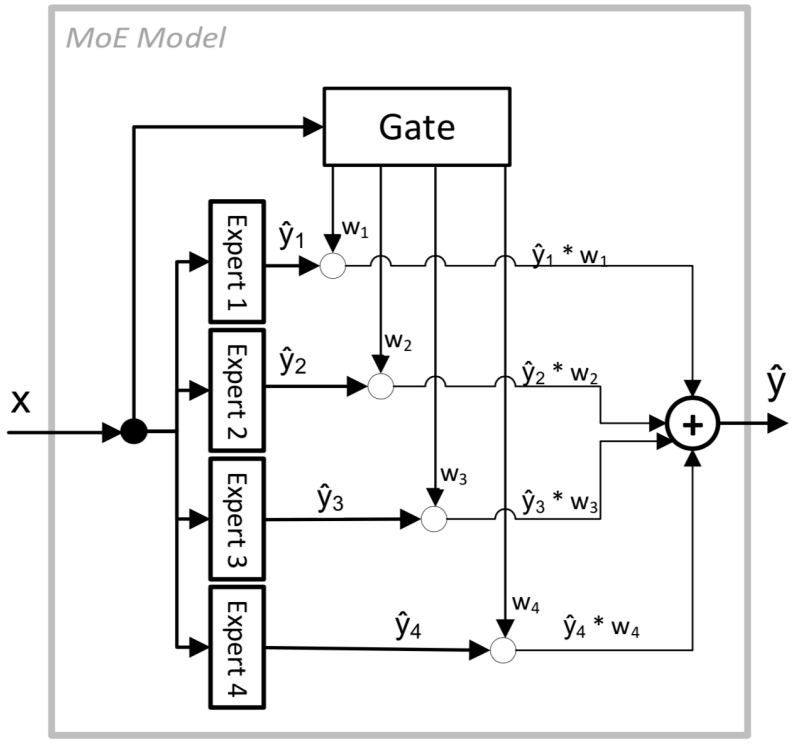


Figure 2.1: A simple example of a mixture of experts approach that relies on the weighted sum of the experts' outputs.

conquer principle in which a complex problem is partitioned into several problems that can be efficiently solved efficiently. Mainly, the input space is divided into regions that are used to train expert models that are supervised by a gate. For solving a nonlinear supervised-learning problem, this approach can divide the problem into linear problems, each of which is simple to learn and interpret. Detailed information on MoE approaches can be found in [56–59].

The conventional MoE approach proposed by Jacob et al. [60] composes a modular architecture, with several competing experts. In this work, a multilayer neural network was used to build the experts and the gate. The proposed approach contains three main components: (1) several experts, each of which is specialized for a subset of input cases, (2) one gate that relies on soft partitioning of the input space to create regions on which the experts can specialize, and (3) a probabilistic model that combines the experts and the gate. The authors investigated many error functions and their influence on system performance. Their study showed that the model's perfor-

mance could be improved by competitive and localized training. For accomplishing this goal, the experts must be trained to minimize their local errors, and their outputs must be weighted for each input observation based on the ratio of local errors to the total error. The final prediction can be seen as a weighted sum of the experts' predictions, where the weights are produced by the gate for each input vector. The weights of the gate network are estimated by maximizing the likelihood of the training set by assuming a mixture of Gaussian models, where each expert is responsible for one component of the mixture. Figure 2.1 shows a simplified example of this approach for making predictions.

The conventional MoE approach has been evaluated by contrasting its performance with that of single back-propagation networks on a vowel recognition problem [60, 61]. The evaluation results have shown better performance and generalization properties on the part of the MoE approach over single networks. Additionally, the MoE approach could converge significantly faster than the single networks do.

The learning process in the MoE approach attempts to estimate the parameters of both components, the individual experts and the gate. The gate's parameters are estimated in a way that enables partitioning the input space towards maximizing the overall performance. An expectation-maximization algorithm (EM) has been used to learn the parameters of the experts and the gate. Training the MoE model with an EM represents cross-entropy between the experts and the gate. Such training is referred to as competitive learning between the experts because each expert gets rewarded or penalized based on its performance on each training case. Additionally, the parameters of the individual experts and the gate can be estimated separately, which enables leveraging diverse algorithms and developing a hierarchical mixture of experts. [57]

The approach of training a MoE with an EM has been extended to a hierarchical structure that allows assigning multiple levels of responsibilities to the experts [62]. The extended approach, referred to as a hierarchical mixture of experts, has a tree structure where the experts exist at the leave nodes and the gates sit on all of the other nodes. Similar to the original approach, the gates make soft partitioning of the input space as a result of which the same input vector could

be assigned to multiple experts with different responsibilities. For a given input vector, the gates produce values that assign responsibilities to the child components, and the experts make their predictions, which are passed to the parent nodes. The experts' predictions are adjusted by the gate's outputs and forwarded to the parent nodes until the final prediction is obtained at the root node. The predictions are conditioned on multiple mixture levels. This approach can be seen as a kind of soft decision tree model with a fixed number of nodes.

The authors in [63] proposed a three-level hierarchical mixture model for classification problems. They formulated a log-likelihood function to consider two assumptions: (1) the data are generated from several sources (clusters) and (2) within each source (cluster), data are generated by multiple sub-sources that constitute the more massive cluster. Bishop et al. [64] proposed a Bayesian treatment based on variational inference to optimize the topology of the hierarchical mixture of experts. The authors claimed that the maximum-likelihood framework that is used to estimate the parameters in the original approach [62] has the following limitations: (1) it is vulnerable to overfitting because of the large number of parameters being optimized compared with the sizes of the training sets, (2) it includes singularities in the likelihood function, and (3) it does not provide a mechanism for determining the topology and the number of nodes in the hierarchical structure. The proposed approach relies on a fully Bayesian treatment of the original approach to eliminate its limitations.

Recently, Zhao et al. [65] proposed a binary tree-structured version of the MoE approach (referred to as hierarchical routing mixture of experts) for regression problems. In the tree structure, the classifiers operate as nonleaf node experts and simple regression models operate as leaf node experts. Together, the classifiers soft-partition the input-output space based on the natural separateness that occurs in multimodal data. This enables the expert models (simple regression models) to provide accurate predictions.

The modular architecture of the conventional MoE approach has drawn researchers to develop different approaches to training the MoE model by minimizing the MoE error. The majority of

the proposed approaches treat the MoE approach as a mixture model, with conditional densities as mixture components that are represented by experts and mixing coefficients that are emitted by the gate [66]. The conventional approach assumes a mixture of Gaussian models that is very sensitive to noise in that data and unsuitable for skewed data [67]. Some research [68–71] has been conducted to deal with heterogeneities, heavy-tailed distributions, and the skewness in the data.

In this work [69], the authors used the skew-*t* distribution to build a robust mixture framework that efficiently deals with heavy tails and extra skewness. The authors presented some modern EM-type algorithms for the maximum-likelihood estimation. Lin et al. [68] proposed a robust mixture modeling framework using multivariate skew-*t* distributions. To estimate the mixture parameters, the authors developed two variants of Monte Carlo EM algorithms for maximizing the likelihood. The authors in [71] employed *t* distribution and developed a dedicated Expectation–Maximization (EM) algorithm to estimate the model's parameters by monotonically maximizing the likelihood. Nguyen et al. [70] used a Laplace approximation instead of a Gaussian process to model the distribution of the errors of the regression experts in the mixture of linear experts to improve the model robustness. The authors derived a minorization–maximization algorithm that monotonically increases the likelihood of the process of the maximum-likelihood estimation.

Masoudnia et al. [58] studied different MoE approaches that use neural networks as expert models for classification problems. According to the partitioning strategy that was employed and how the gate was involved, the authors categorized MoE approaches into two groups, a mixture of implicitly localized experts and a mixture of explicitly localized experts.

The approaches that use a mixture of implicitly localized experts can be seen as extensions of the conventional MoE approach. The gate uses a particular error function to determine which of the experts should be used for training on each observation during the training process. In the training process, the gate partitions the input space according to the experts' performance and then allocates a subspace to each expert to learn. Different approaches have emerged to improve the partitioning strategy and the overall performance by modifying the error function of the gating

network. According to Masoudnia et al.'s survey, some approaches investigated the trade-off between *selection* (i.e., one or more experts are used to predict a given observation) and *fusion* (i.e., all experts are used for making predictions) strategies. In these approaches, it was found that the selection approach reduces the bias term in the prediction error, while the fusion approach leads to variance reduction. For achieving an optimum balance between the two strategies, the error function of the gate was extended to include two terms for the selection and fusion strategies. In such a case, the gate learns the switching conditions between the two strategies, which enables a better performance than does the corresponding MoE that relies on a single strategy.

With a mixture of explicitly localized experts, the proposed approaches have a similar structure as do the methods that use a mixture of implicitly localized experts, including experts and a gate, but they rely on a different partitioning principle. While the data are partitioned into nested and stochastic regions of the input space during the training process in the methods that use implicitly localized experts, the approaches that use explicitly localized experts partition the input space into separate regions before training the experts. The experts are trained with different regions of the input space.

Several partitioning methods have been used. For instance, the authors in [72] used Self-Organizing Maps to partition the input space into several clusters. An expert was fit on each data partition. In other work [73], the fuzzy C-means clustering algorithm [74] was employed to cluster the data into overlapped training sets, each of which was used to train a multilayer perceptron expert. The authors in [75] developed a new method that was based on cooperative coevolution and a mixture of experts to automatically partition the input space into several regions. In this method, the experts' models are assigned to the produced regions. Adding the cooperative coevolution layer to the MoE enables a better investigation of the weights of the experts. The authors showed that their approach could, on average, outperform the conventional MoE approach on several classification problems. The authors in [76] developed a new MoE system for view-independent face recognition. The developed MoE composes multilayer perceptrons experts and a radial basis function gating network. In their work, the face view space was manually decomposed into views.

They developed a teacher-directed learning method that forces the experts to specialize in their corresponding views.

In this work [77], the authors developed mixture of experts (MoE) and AdaBoost models by using different algorithms and then contrasted the performance of both approaches. Using different learning algorithms, they investigated three MoE and AdaBoost approaches. The MoE model approaches that were investigated are those that use (a) a multilayer perceptron as an expert and a gating network (mlp), (b) a general linear model as an expert and a gating network (glm), and (c) a support vector regression as an expert (svr) and glm as a gating network. For building the AdaBoost ensemble, the authors used mlp, glm, and regression trees as base learners. The evaluation experiments showed that all of the variants of MoE and AR2 provided better performance than their single models did, and there was no significant difference between MoE and AR2.

2.4 Large-Scale Data Processing

Several approaches have emerged to enable the execution of iterative computations that are needed by a large class of machine learning applications. Some approaches [19, 78–80] focus on extending Hadoop by adding features that allow the execution of iterative computations in a more efficient way. Some others [7, 81, 82] provide graph abstraction that can be used to express iterative computations. Also, there are frameworks [5, 18, 83–86] that are specially developed to solve iterative problems or a broader class of machine learning applications. In this section, we will introduce some approaches that enable gaining insights from voluminous datasets.

2.4.1 Sampling Approaches

Sampling is a commonly used strategy for dealing with voluminous datasets when the computational resources are insufficient to create analytics models efficiently. Sampling relies on a random process to create a smaller version of the original data. The created subset can then be used as a surrogate for building the analytical models. For many domain problems, different strategies [87–93]

have been proposed to create a smaller representative subset of the original dataset in a way that enables building analytics models with performance that is similar to those that are created using the entire dataset. Borovicka et al. [94] discussed different ways for creating and evaluating the representative subsets.

Although sampling enables the analysis of voluminous data, it becomes infeasible for optimizing problems with a large number of parameters that involve voluminous data to achieve an acceptable solution. Full use of big data that is both large in volume and variability promises opportunities to explore more hidden structures in such data [1]. Agarwal et al. [19] have shown that increasing the sampling size of such data usually leads to better accuracy.

2.4.2 Divide and Conquer Approaches

Divide and conquer is another technique for enabling learning from voluminous data. To overcome the scalability issues that prevent applying classical learning methods to voluminous datasets, this approach relies on partitioning the original dataset into subsets, finding a solution for each subset independently, and combining the local solutions to provide a global solution for the original dataset. Several approaches [95–98] rely on this technique to allow diverse solutions for different domain problems. Some approaches [99–101] employ ensemble techniques to learn from voluminous data.

Zhang et al. [98] used the divide and conquer technique for solving a classification problem by using kernel ridge regression. In this work, the authors randomly partitioned a dataset into equally-sized subsets, employed kernel ridge regression to compute a local solution for each subset, and averaged the local solutions to obtain the global solution. Averaging the estimated subsolutions into a global solution tends to find an average solution instead of maximizing the performance of the global solution. Qi Guo et al. [96] applied the divide and conquer concept by decomposing the feature space into a number of subspaces and locally building a classifier for each subspace. The authors built a global classifier based on the outcomes from the local classifiers. For a given observation, the local classifiers are applied, and their predictions are fed to the global classifier to

produce a final prediction. The authors illustrated that their approach could reduce the prediction error compared with a solution employing all of the data for building a single model. However, the proposed solution might fail to achieve the same performance with different datasets because their solution incorporates all of the captured knowledge in a single model.

Farrash et al. [99] proposed a framework that enables building ensembles for voluminous learning. The proposed framework was applied to a classification problem using core vector machine (CVM) by partitioning a given dataset into disjoint subsets and fitting a classifier for each data portion. The final prediction was obtained by relying on majority voting. In this work, the authors mainly investigated the effects of some partitioning strategies and subset size on ensemble performance. The tested partitioning methods divide the whole dataset into equally-sized portions, which means that the obtained results could be influenced by two factors, the number of models within the ensemble and the size of the subsets. Basilico et al. [100] proposed methodology for building a random forest ensemble on a partitioned dataset. The authors employed the MapReduce framework to create one mega-ensemble in a single MapReduce job that builds a random forest model for each data block in the map phase and then aggregates the random forest models to a mega-ensemble in the reduce phase. In the proposed approach, the time spent to build the final ensemble can be significantly influenced by two factors. In the map phase, the map task uses IVoting [102] to build the random forest model in a sequential way that is similar to that used in boosting ensembles. Also, the shuffling phase has to transfer huge amounts of data (random forest models) that could exceed the size of the original dataset because it transfers the deep-growing trees from all of the map nodes to the reduce node. Panda et al. [101] developed a scalable distributed framework, PLANET, for building tree models over voluminous datasets. This framework relies on MapReduce jobs to find a set of candidate splits for the trees' nodes to parallelize the extensive computations that are involved in trees induction. In this approach, building the trees could be extremely time-consuming because the MapReduce jobs scan the entire dataset even if small portions of data are needed and each massive tree is built sequentially.

2.4.3 Data Processing Frameworks

The problem of processing and fully using large-scale data has motivated several industrial and academic organizations to build an infrastructure to cope with continuous data growth and enable multipurpose computations on big data by using a large number of machines. For instance, Google File System (GFS) [103] and Hadoop Distributed File System (HDFS) [104] have emerged to enable transparent storage and access for large-scale data that is distributed across a huge number of commodity machines in a reliable manner. Several computing frameworks have been proposed to apply computations on large-scale data in parallel by relying on a large number of networked machines. Good examples of these approaches include Hadoop [105], which implements the MapReduce programming model, and the original implementation of MapReduce by Google [106], which enabled processing 20 PB of data per day.

Among the proposed distributed computing systems, Hadoop has gained widespread use for addressing any problem that is expressible as a MapReduce job. Hadoop is the most commonly used framework that uses a large cluster of unreliable commodity machines for large-scale data processing. In addition to its simplicity and availability, Hadoop hides all of the complexity that is associated with the parallel processing of big data and lets the users focus only on the actual problem. Hadoop autonomously performs all of the needed work for distributed computations such as distributing data across machines, balancing the workload, the parallel processing of big data, performing I/O operations, transferring data between nodes, and handling failure. These are some of the features that lead to its widespread use in industry and academia. Although it is still possible to use Hadoop to execute an iterative application by launching one MapReduce job for each iteration, such an approach is inefficient for two main reasons:

1. Launching the same MapReduce job for each iteration involves reloading and reprocessing the input data even if the data stays unchanged throughout the iterations. Besides, the intermediate outputs are usually stored in the HDFS after each iteration to enable the following iteration (MapReduce job) to use them, resulting in replicating the intermediate outputs on

different machines. Executing iterative applications in this way increases the execution time and wastes the cluster's resources.

2. Stopping the iterative computation often involves checking whether a given termination condition has been satisfied or not after each iteration. Because Hadoop does not support such a feature, users are usually involved in developing a new MapReduce job to check convergence status or executing the MapReduce job for a fixed number of iterations.

2.4.4 Iterative MapReduce-Based Approaches

Some approaches [19, 78, 79] focus on extending Hadoop by adding features that allow the execution of iterative computations in a more efficient way. A framework developed by Bu et al. [78], *HaLoop*, extends Hadoop to enable iterative computation by executing the job's map and reduce tasks multiple times until convergence is satisfied. HaLoop caches the input data in the first iteration to avoid frequent disk access, and it does the same with the outputs of each iteration to assess the termination condition more efficiently. HaLoop is a strict consistency model and makes use of all of Hadoop's optimizations.

Agarwal et al. [19] extended Hadoop to enable iterative computations. These authors showed that it is feasible to extend Hadoop to support iterative computations because Hadoop is a data-centric computing platform, and a broad class of ML algorithms is amenable to working on partitioned data. The main contribution of this work is the use of the AllReduce technique to add a mechanism of state synchronization and share to Hadoop. The proposed approach implements AllReduce by using a tree structure to support parallel computations of many statistical queries. The execution of the AllReduce operation involves two phases, a reduce phase and a broadcast phase. In the reduce phase, the computing nodes pass subsolutions that are optimized on their local data to their parent nodes where the subsolutions are combined and given to their parents again until the global solution is obtained at the root node. In the broadcast phase, the global solution is propagated to all of the nodes by sending it back down to all of the other nodes. The job includes only map tasks whose hosted machines are nodes forming the spanning tree that the system uses

for the AllReduce operations. A running spanning-tree server on the gateway node to the Hadoop cluster creates the spanning tree and supplies each map node with the IP addresses of its parent and children. In addition, the authors proposed a distributed algorithm, a hybrid, that combines the desirable features of online and batch algorithms. The hybrid algorithm initially uses an online learning algorithm to reach the neighborhood of an optimal solution and uses batch learning to improve the online learning solution towards an optimal solution. The online learning is locally applied to each data portion without involving communications between the nodes. Then, the refined weights of the local solutions are averaged by using the AllReduce operation, and the globally resulted weights are broadcast to all nodes. After a number of passes over the training data, the optimized weights are used to initialize the batch learning, which is also locally applied to each data portion, and the fitted weights are summed up by using the AllReduce operation in each iteration. To achieve a good trade-off between reliability and scalability, the creation of a spanning tree is delayed until all of the nodes have completed the first pass over the data, and it uses only the speculative execution survivors.

Zaharia et al. [80] developed a system called *Spark* to support computations of a large dataset in distributed memory on a large cluster in an efficient and fault-tolerant fashion. Spark provides a distributed memory abstraction that relies on resilient distributed datasets (RDDs) to enable users to perform efficient processing of problems involving data reuse such as iterative computations and interactive queries. The RDDs are read-only data objects that the system constructs using deterministic operations called transformations from either stable storage datasets or other existing RDDs. The framework lazily executes the transformations to create the RDDs when it needs to for computations. Once the system creates the RDDs, they are partitioned and cached in memory across the cluster's nodes to provide efficient data access and reuse. The main advantage of the RDDs concept is that it addresses the inefficiency that occurs by employing Hadoop for an iterative computation, which involves launching a new MapReduce job for each iteration, resulting in frequently reloading the input data from the disk and replicating the intermediate outputs across multiple nodes. Spark uses the RDDs to cache the input data and intermediate results in memory

and uses them in iterative computations, without accessing the disk and replicating the data. In Spark, only the first iteration involves accessing the disk to load the data, and the remaining iterations access the RDDs that represent the input data and intermediate results in memory. In Spark, the user can execute each iteration in the map and reduce phases. Additionally, Spark provides efficient fault tolerance by logging and replicating for each RDD a lineage graph that contains all of the transformation operations that were used for its construction. Recovery from failure involves only the re-creation of the missing RDDs by executing the transformation operations in the lineage graph. Logging only the transformation operations, instead of the actual data, results in gaining efficient fault recovery with low overhead.

2.4.5 Graph-Based Approaches

Low et al. [82] proposed a framework, *GraphLab*, to execute iterative applications by using a directed data graph that enables expressing data and computational dependencies. In GraphLab, a user implements two different computations through the *update function* and the *sync mechanism*, which are analogous to the map and reduce tasks in MapReduce, respectively. GraphLab applies the update function to graph vertices in parallel for local computations and the sync mechanism for aggregating the global state. This framework provides a shared data table (SDT) to support a globally shared state. GraphLab supports three different consistency models (fully, edge, and vertex), each of which determines different extents to which computations can overlap. The framework does not provide any mechanism for fault tolerance, and such a feature is considered as ongoing research.

Abadi et al. [83] proposed an open-source software library for machine learning. It relies on a data-flow graph (referred to as TensorFlow graph) to enable expressing computations and executing them on different computational devices (such as CPU cores and GPU cards). In a TensorFlow graph, the nodes represent mathematical operations and the edges represent the data flow between the graph nodes in the form of multidimensional data arrays known as tensors. The TensorFlow

engine uses the available computational devices to execute the graph nodes. The authors developed a placement algorithm to map different graph nodes to different computational devices for execution. The algorithm runs a simulated execution of the graph to identify all applicable devices for each node, and then the algorithm chooses the one that provides the best performance.

The TensorFlow architecture is composed of three components: client, master, and worker processes. The client can instantiate a TensorFlow graph and create a session to communicate with the master and the worker processes for executing the graph. The master process coordinates the executions of the worker processes. Each worker process is responsible for one or more computational devices for executing the mapped graph nodes. The three components can run on a single physical machine or different machines for distributed executions. The authors' ambition with the proposed approach was to build a flexible system that uses a TensorFlow graph to execute any computations that are expressible with a directed acyclic graph. In addition to the execution of the whole TensorFlow graph, the user can execute a portion of the graph by passing the names of desirable outputs to the system. In this case, the system uses the output nodes and moves backward using the dependencies to determine all of the nodes that the system executes.

The TensorFlow graph automatically supports distributed and parallel execution of the graph nodes. The master needs only to send a *run* command to all of the worker processes. The nodes that do not have direct dependencies can be independently scheduled for parallel execution, while node dependencies that can be expressed by data edges or special edges, called control dependencies, enforce a synchronous execution of the operations. The TensorFlow graph programming model enables using common programming idioms to speed up the training process. For example, to parallelize the computation of gradient descent, the user can divide mini-batches into smaller splits, and the system processes them on different devices in parallel. The parallel execution involves having many replicas of a computational subgraph and using a single client thread to coordinate the entire training loop of those replicas. After all of the splits have been processed, the gradients are combined and the model's parameters are synchronously updated. To enable asyn-

chronous updates of the model's parameters, a user has to follow the same steps, but with creating one client thread for each replica.

The TensorFlow engine rewrites the user graph by inserting some special nodes including nodes that feed the graph with data and store the outputs as well as the ones that enable data transfer between the graph nodes across different devices and machines. The TensorFlow graph also supports many operators that allow expressing a simple conditional statement and an iterative loop.

2.4.6 Parameter Server Approaches

Li et al. [5] proposed a parameter server framework for solving distributed machine learning problems. The framework design is composed of parameter servers and worker machines. The framework partitions the model's parameters across the parameter servers and the training data across the worker machines. The system organizes the workers in groups, each of which runs an optimization for an individual application. The globally shared parameters that the system keeps as (key, value) pairs on the parameter servers can be accessed by using their keys. In every learning iteration, each worker pulls the model's parameters from the servers, refines them using its data, and pushes them back to the servers. The servers aggregate the received parameters and update the model.

To avoid the inefficiency that results from expensive global synchronization barriers between the iterations, the framework supports a flexible consistency to allow the workers to pull and push the model's parameters in an asynchronous fashion that the system controls with a bounded delay. In the supported bounded-delay consistency model, the proper choice of the maximum time that limits the parameters' staleness between the fastest and slowest worker increases the algorithm's performance and does not hurt the algorithm's efficiency. Achieving such a trade-off depends on several factors including the algorithm's sensitivity to data inconsistency, feature correlation in the training data, and heterogeneity between the workers. Also, to reduce the network traffic between the workers and servers, each worker filters the parameters and only pushes those that more likely

affect the model.

Xing et al. [85] proposed a framework called *Petuum*, which exploits several properties of machine learning programs to address data- and model-parallel challenges in large-scale machine learning applications. The authors claim that a minor error in a model's parameters does not influence the algorithm convergence, and the framework allows asynchronously updating the model's parameters with a bounded staleness guarantee to improve the iterations throughput. Also, Petuum exploits the nonuniform convergence of the model's parameters and prioritizes the slow-converging parameters to speed up the training process. Additionally, the framework scheduler groups the model's parameters into independent blocks to enable parallel updates of the parameter groups and enforce sequential updates inside each group.

Dean et al. [18] developed a framework called *DistBelief* that relies on data and model parallelism to train large deep learning models on computing clusters with thousands of machines. The user needs to define the computations that should take place at each node in each layer of the neural network and the messages that the framework should pass between the layers, and the framework automatically parallelizes the computations and manages the messages transfer and synchronization. This framework relies on model parallelism to parallelize the computations within a single model and on models' replicas to speed up the training time and achieve efficient failure handling. The framework employs the concept of a centralized shared parameter server to enable the models' replicas to share their states.

2.5 Discussion

The majority of the surveyed approaches have to sacrifice some aspects of performance. For example, sampling methods reduce the training time significantly but become infeasible for optimizing problems with a large number of parameters or for data with high variability. All of the studied work enables iterative computation, which is a key component for gaining insights from

data. While iterative computation is efficiently applicable to data that are loaded in memory, it can lead to poor resource utilization and scalability issues when the data are hosted by multiple machines. Some proposed approaches try to enable iterative computation on voluminous distributed data by enabling iterative computation on Hadoop, using a directed data graph, or relying on parameter servers. These approaches can reduce the training time for voluminous data, but their performance can suffer severely when the computations include numerous dependencies or involve high network traffic. To improve the perfomance, some of these approaches attempt to relax the synchronization barrier that is involved between iterations, restrict some behavior of the algorithms, or modify the algorithms to become system friendly.

Scalability improvement requires enabling concurrent learning from distributed data, reducing the dependencies involved, and minimizing the network traffic. These objectives are achievable by relying on and combining some aspects of ensembles, mixtures of experts approaches, and divide and conquer methods. Ensembles methods have been developed to improve the prediction and generalization for small datasets. However, the concept of the ensemble can be used as the divide and conquer approach to enable learning from distributed data at large scales. In other words, building multiple models on distributed data and combing those models for making predictions. Concurrent and independent training of the ensemble's models enables efficient resource utilization and addresses the scalability issues, but it can hurt the prediction accuracy.

Improving the prediction accuracy can be accomplished by adopting the concept of the mixture of experts approaches. These approaches improve the prediction performance by relying on a gate for partitioning the data to build models that are expert for different subsets of the data—usually, the gate partitions the data during the models training based on the models' performance, which represents a bottleneck in the system. By partitioning the data based on common characteristics before the training, we can improve the models' predictions without hurting the scalability.

Chapter 3

Motivating Experiments

We studied the main challenges that are associated with training models on voluminous datasets (§3.2). Next, we evaluated the effect of volume increase of training data on prediction accuracy and generalization (§3.3). Finally, we investigated the training of independent models and contrasted their performance (§3.4). We relied on the training of independent models to construct an ensemble that uses all models for making a prediction for a given observation. For this study, datasets from different domains were used to conduct several experiments (§3.1).

3.1 Experimental Setup and Datasets

In our experiments, we trained the analytical models on separate machines with the same capability (HPDL60 with Xeon E5-2620 CPU, 16 GB of RAM). The models that were built on data that were drawn from the entire training datasets are referred as global models. Ensembles that were built in this chapter comprise instances of individual models that were trained on different even-sized portions of the dataset, hosted by multiple machines. The individual models forming the ensembles and the global models were ensembles that were trained by using the gradient boosting algorithm, implemented in scikit-learn [107].

Unlike the global models, the individual models of the ensembles needed to deal with quite small training datasets, which in turn spent less time for the data loading and validation processes. However, to contrast the training times of both approaches, we reported only the times that were needed for the training process, without considering the time spent for reading the data from the disk and the validation process that is involved to adjust the model's complexity. We feel that is a fair comparison in all of the benchmarks. The best and fastest global models were contrasted with the ensembles. We used mean squared error (MSE) to measure the prediction accuracy.

To evaluate the independent models training, we used three different multidimensional datasets. The datasets were from different domains and had varying levels of dimensionality and variability

Table 3.1: The datasets that were used to evaluate the efficiency of our methodology.

Dataset	Training Obs.	Test Obs.	Features	Variation	Target (Unit)
Weather Data (NOAA)	1,299,854	150,000	8	Low	Temperature (Kelvin)
Million Song Data	463,715	51,630	90	Medium	Release Year (Year)
Epidemiology Data	900,000	100,000	2,595	High	Disease Duration (Day)

to profile the suitability of our ensembles. The datasets were also well-suited for us to investigate the relationship between the increase/decrease of the variability in the dataset and corresponding gains in performance from our ensembles. Table 3.1 briefly summarizes these datasets.

The first dataset was a real-world climate dataset that was obtained from the National Oceanic and Atmospheric Administration North American Mesoscale Forecast System [108]. This dataset includes readings (surface pressure, surface temperature, snow cover, snow depth, relative humidity, wind speed, etc.) that are regularly collected from various weather and climate stations. We randomly sampled about 1.45 million observations from an 11-year period between 2004 and 2014. We used climate features to predict the surface temperature values measured in Kelvin. The dataset is described with a small number of climate features, resulting in low variability data.

The million song dataset was downloaded from the UCI machine learning repository [109]. The dataset includes audio features of the songs released in the years ranging from 1922 through 2011. We used audio features to predict the release year of a song. The variability in this dataset is higher than that of the previous one because of the high dimensionality of audio features.

The third dataset consisted of epidemic simulation data that captures the spatiotemporal spread dynamics of disease outbreaks among livestock populations in Texas. This dataset represents the outcomes from executing one million variants of a disease spread scenario, describing information about herds and their interactions with the environment. We relied on a stochastic discrete event simulation, the North American Animal Disease Spread Model (NAADSM), to execute these variants and produce outputs. Details on how to generate such data can be found in §6.1. Each observation includes input parameters that describe a particular scenario variant and output param-

eters that describe the outcomes for that particular scenario. Our dataset comprised one million scenario variants and their corresponding outcomes. The model that we considered used input parameters to predict one of the output parameters, disease duration (in days). The epidemiology dataset includes data with high variability because Latin hypercube sampling was used to sample the high-dimensional feature space to create the scenario variants.

3.2 Challenges of Learning from Voluminous Datasets

Traditional machine learning algorithms have been designed to fit models on quite small datasets by loading the entire training set in memory and making multiple passes over it to refine the model's parameters gradually. However, the rapid growth in data volumes represents a unique challenge to applying such algorithms on training data that is too large to fit in the memory of a single physical machine.

Gaining insights from a voluminous dataset is achievable by applying a learning algorithm on distributed data to build a global model. Iteratively, the learning algorithm is locally applied to the data portions that are stored on distributed workers in parallel to find subsolutions that are synchronized and aggregated to update the global model. All of the patterns and relationships in the data are captured in a single complex model that usually generalizes well. However, building a global model in such a manner involves sacrificing some degree of desirable properties.

Optimizing the global model involves exploiting the natural decomposability of the objective function over the training examples or, if possible, modifying the learning algorithm to become system friendly. Additionally, the nature of some algorithms makes the parallelism inefficient or even impossible, which limits the number of applicable techniques, especially the ones that work sequentially. Also, training a global model on voluminous data in parallel usually requires a global synchronization barrier for each update step of the shared model's parameters, resulting in poor resource utilization and extension of the training time that will mainly depend on the slowest machine.

3.3 Effect of Data Volume on Performance

We investigated the effect of data size and variability on the prediction accuracy. Furthermore, we analyzed the prediction error to see how its components (bias and variance) are affected by such factors.

3.3.1 Effect of Data Volume on Prediction Accuracy

Voluminous data, which is not only massive in volume but also variability, enables exploring hidden structures in data and improves decision making [1]. That is, voluminous data offer opportunities to extract insights from them. We investigated how the size of training data with different variability affects predictive accuracy. For this test, we used different sampling sizes of the epidemiology (higher variability) and weather (lower variability) datasets.

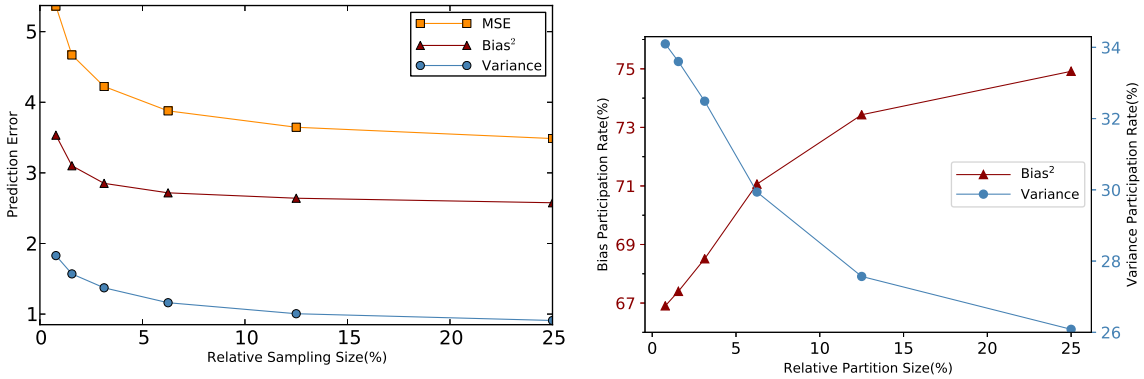
Table 3.2: The MSE as a function of the sampling rate for datasets with different variability

Dataset	1%	10%	50%	100%
Epidemiology Data (high variability)	5.36	3.88	3.46	2.46
Weather Data (low variability)	31.47	26.01	25.20	25.10

Table 3.2 lists the accuracy that was obtained for each sampling size. In both datasets, increases in the sampling size improved the prediction accuracy. However, the improvement ratio has a positive correlation with the variability in the dataset (i.e., the increases in the size of training set drawn from high-variability datasets add extra information to the training set, resulting in better predictive accuracy). The results illustrate that including massive number of training observations from a high-variability dataset becomes necessary for gaining better accuracy, while sampling from a low-variability dataset could be sufficient to build accurate models in a relatively short time.

3.3.2 Effect of Data Volume on Bias-Variance Components

We investigated the effect of data volume on the bias-variance components. We built models on data with different sampling sizes, computed their prediction errors (MSEs), and decomposed



(a) **(b)**

Figure 3.1: The bias-variance decomposition of prediction errors for different sampling sizes of the epidemiology dataset. (a) The MSE along with its bias and variance components. (b) The proportion of bias and variance in the MSE.

them into bias and variance. We performed this analysis with the epidemiology dataset and plotted the results in Figure 3.1.

As we can see in Figure 3.1a, both components (bias and variance) participate in the reduction of the prediction error. To find out which part contributes more to the accuracy improvement, we plotted the proportions of these components in MSEs in Figure 3.1b. As we can see, the variance proportion gets smaller once the sampling size becomes larger. It indicates that an increase in training set size reduces the chance of overfitting and leads to better generalization capability of a model.

3.4 Random Ensemble

Independent models training takes advantage of the distribution of data and collocates a learning process on each machine holding a data partition to train a predictive model. The trained models were used to form an ensemble (referred to as *random ensemble*). Like bagging, the individual models are built with high complexity to generate predictions with low bias, and the constructed ensemble averages the models' predictions to reduce the variance. The random ensemble approach enabled learning from voluminous data in a short time without sacrificing too much predictive accuracy. This type of ensemble should produce predictions that are more accurate and robust than

any of its constituent models are. We applied the random ensembles in different work [110, 111] and observed similar results.

The ensemble performance depends on the volume of the training sets that are used to build the ensemble's models. The predictions of the individual models get better when the size of data partitions becomes larger, resulting in improvements of the ensemble's predictions. This relationship illustrates the trade-off between training time and prediction accuracy. That is, accuracy improvement can be accomplished at the cost of the training time. In this section, we illustrate the construction of the ensemble and assess its effectiveness by conducting several experiments, using datasets from diverse domains. In particular, the experiments attempt to answer the following questions:

1. How does the base learner affect the ensemble's performance?
2. How does the ensemble affect generalization in comparison with its best model?
3. How does the ensemble influence bias and variance of the prediction error?
4. What is the most effective ensemble size for different datasets?
5. How well does an ensemble of gradient boosting models work with different datasets?

3.4.1 Ensembles Construction

We assume that the dataset is hosted by several machines. We sample from the distributed data to create a test dataset that is not included in the learning process but used for assessment purposes. We launch a training process on each machine that is hosting data partitioning to build several models in parallel. Only a subset of the trained models is used to construct an ensemble. For a given input feature vector, the ensemble uses its constituent models to make predictions that are averaged and emitted as the final prediction. The use of independent and concurrently trained models instances comprising the ensemble enables learning from a voluminous dataset in a timely manner. Figure 3.2 illustrates the phases for constructing an ensemble over a voluminous dataset and applying it to make predictions.

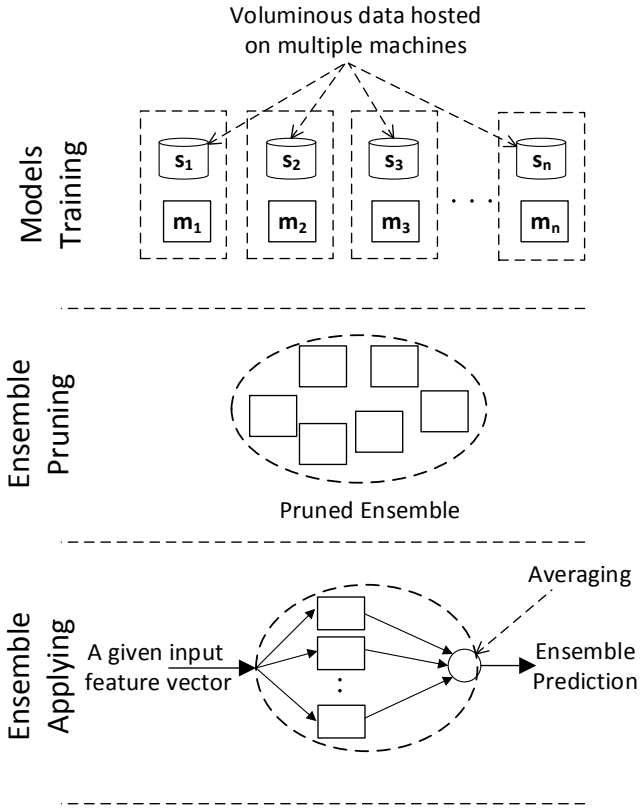


Figure 3.2: The creating and applying phases of the ensemble that was built on randomly partitioned data.

For building this ensemble, multiple techniques are leveraged to preserve predictive accuracy with high generalization. This involves choosing the most appropriate learning algorithm for the problem that is being solved to achieve good predictive accuracy. We tune the hyperparameters of individual models to increase their complexity. Each of the individual models approaches the underfitting of its training set and becomes very sensitive to data variations. We rely on 10-fold cross-validation to train, assess, and tune the models' complexity. The tuning process generates model instances whose predictions have slightly lower bias and high variance. The prediction variance is then reduced by averaging the models' predictions. Like the bagging ensemble, the created ensemble improves the predictive accuracy over unseen data by reducing the prediction variance with a slight increase of the bias, if any. Improvements are predicted based on the following factors:

i. **Choice of the base learner**

   Because the ensemble tends to reduce the variance of the base learner, the base learning algorithm should generate models with a low bias to maximize the improvement ratio. Building low-bias models results in an ensemble with low bias *and* variance.

ii. **Correlation among the models**

   In order to reduce the prediction variance of the base learner (e.g., reducing the generalization error of the ensemble), the construction of ensembles must account for the correlation properties of model instances. The ensemble's models must not be highly correlated. Unlike bagging, which relies on several bootstrap samples of the same size as the original training data, our ensemble fits the individual models on different subsets of the voluminous data to reduce the correlation between them.

iii. **Composition of the ensemble**

   We select a subset of trained model instances to comprise an ensemble such that we achieve better generalization than with an ensemble comprising all of the available model instances.

This process is known as ensemble pruning. It has been shown that an ensemble-pruning phase improves generalization [112].

3.4.2 Ensemble Pruning Algorithm

Different techniques [112] have been proposed for performing ensemble pruning. We developed a greedy algorithm 1 that initializes the ensemble with a model that has the best predictive accuracy. In this algorithm, additional model instances are incrementally added to the ensemble only if they improve the ensemble performance. The pruning process completes when the algorithm does not find an instance that improves the generalization when added to the ensemble. We use the test dataset to assess pruning. Our algorithm is fast because it does not use the actual individual models and only relies on their predictions of the test data.

3.4.3 Effect of Base Learner on the Performance

We investigated the effect of the base learner on the performance of the ensemble by using two different base learners. To that end, we employed the epidemiology dataset to create ensembles by using two different base learners, one using ridge regression and the other using a gradient boosting ensemble.

The training set was evenly partitioned into 128 portions, each of which was hosted by one machine. For each base learner, 128 individual models were built on the corresponding partitions with data locality. Then, the test data were used for assessing the generalization capability of both ensembles.

As the results in Figure 3.3a demonstrate, both ensembles improved the generalization ability over their models. However, we obtained higher accuracy by using the gradient boosting as a base learner than we did by using the ridge regression approach. Ridge regression builds a simple model with slightly high bias. Gradient boosting employs a shallow decision tree with low variance as a base learner and reduces the bias, resulting in a lower generalization error.

We computed the expected test MSE and decomposed it into bias and variance for the ensembles and their models (see Figure 3.3b). Both ensembles reduce the variance of their base learner.

Algorithm 1: Ensemble pruning: reducing the ensemble size

```
Input: actValues ;                  // Actual values of the test data
       predictions ;                // List with predictions of each
                                    // model for the test data
Output: mIndices ;                  // Indices of models constructing
                                    // the pruned ensemble
idx, pred = findBestPred(actValues, predictions);
mIndices = { idx } ;
mPred    = { pred } ;
Delete predictions {idx} ;
while predictions != { } and idx != -1 do
    enRMSE = rmse(actValues, average(mPred));
    idx    = -1;
    for index in predictions do
        enPred    = average(mPred , predictions{index});
        tmpEnRMSE = rmse(actValues, enPred) ;
        if tmpEnRMSE < enRMSE then
            idx    = index ;
            enRMSE = tmpEnRMSE;
        end
    end
    if idx != -1 then
        mIndices ← idx ;
        mPred    ← predictions{idx} ;
        Delete predictions {idx} ;
    end
end
```

However, the ensembles' performance relies not only on variance reduction but also on the choice of the base learner and the tuning of its complexity. The ensemble with better predictions takes advantage of the low-biased models that are built by using gradient boosting to achieve very low generalization error compared with ridge-regression-based models, which are not complex enough to capture the trends in data. We can see in the figure that the ridge regression model has high bias and low variance.

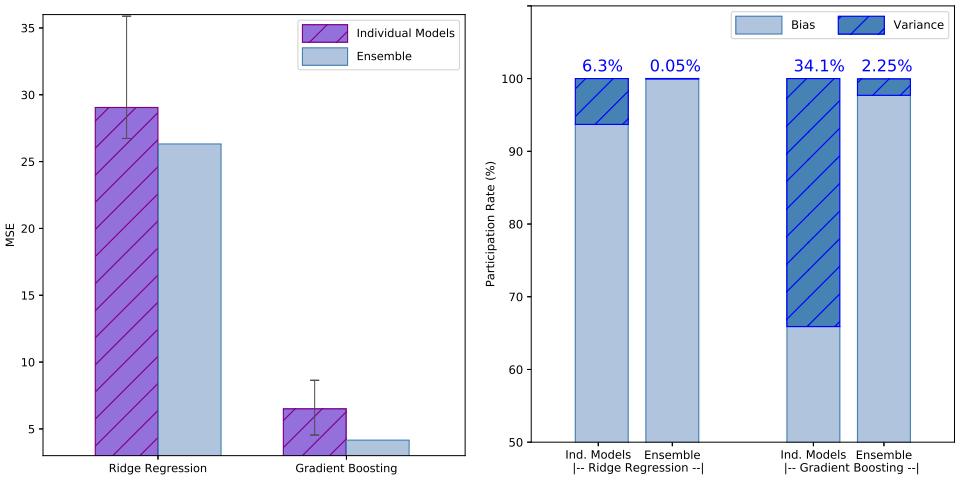


Figure 3.3: The effect of two different base learners, ridge regression and gradient boosting, on ensemble performance. (a) The MSE for both ensembles along with their individual models. (b) The contribution of bias and variance to the MSE.

3.4.4 Ensemble Performance

We assessed the performance of the ensembles by using a different number of data partitions. We considered ensembles with 2, 4, 8, 16, 32, 64, and 128 individual models. Note that the increases in the number of data partitions reduce the sizes of these partitions. In this case, both the number of individual models and the size of the data partitions influence the accuracy of the constituent models and the correlations between these models. To take this point into consideration, we also computed the improvements that we gained from the ensembles over their best individual models.

As we can see in Figure 3.4, the ensembles became more accurate as the size of the data partitions increased. The key reason behind such improvement is that the performance of the individual models within the ensembles improved as the size of the data partitions increased. As the second y-axis shows, the ensembles provided a further improvement over their individual models by reducing their prediction variance. This improvement became smaller as the partition size got larger because raising the size of a data partition reduced prediction variance, resulting in building highly correlated models. Also, the improvement ratios of the ensembles over their individual

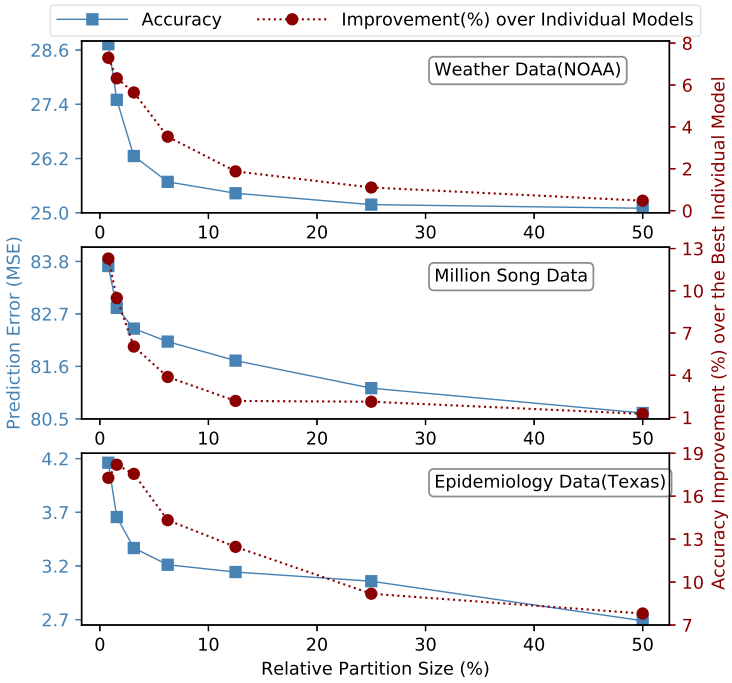


Figure 3.4: Ensemble's accuracy and percentage of accuracy improvement over its best individual models for different partitions' sizes.

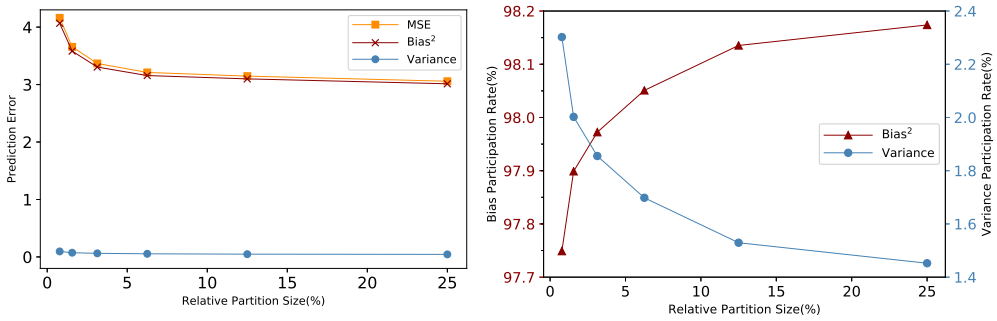
models were different in the three datasets. It seems that datasets with high variability (such as the epidemiology dataset) benefitted more from our ensemble.

3.4.5 Bias-Variance Decomposition of the Ensemble

We computed the bias and variance components in the ensembles by using the epidemiology dataset and contrasted their proportions in the MSE. As we can see in Figure 3.5, the variance proportions in the MSE of the ensembles approximates zero, while it is considerably high in the sampling approach Figure 3.1.

3.4.6 Ensemble versus Global Model

In this experiment, we randomly drew observations from each training dataset to create subsets that were 50% of the original training datasets. For each dataset, we used the created subset to build one global model and ensembles with different sizes. The number of investigated data partitions



(a) **(b)**

Figure 3.5: The bias-variance decomposition of prediction errors in the ensemble for different partitions' sizes of the epidemiology simulation dataset. (a) The MSE along with its bias and variance components. (b) The proportion of bias and variance in the MSE.

was 2, 4, 8, or 16. The results in Figure 3.6 illustrate that the ensembles enabled gaining a high speedup of training time over the global models, without sacrificing too much on accuracy.

3.4.7 Ensemble versus Global Model by Using Spark

In this experiment, we used Spark to build random ensembles and global models over the epidemiology and million song datasets. Spark was deployed as a standalone with the Hadoop Distributed Filesystem (HDFS), using 80 machines. The global models were built on the entire training dataset by using the entire cluster. For the epidemiology dataset, we built one random ensemble with 15 individual models, while we built 3 ensembles with 4, 8, and 17 individual models for the million song dataset. The models within each ensemble were built simultaneously by using the Spark cluster. To avoid performance degradation that can result from high data shuffling and CPU contention, we relied on static partitioning of the cluster resources and grand two nodes for each model

Figures, 3.7a, 3.7b, 3.8a, and 3.8b, show the accuracy and training times of the global and random ensembles. The global models achieved better accuracy than the random ensembles did because the individual models comprising the random ensembles trained on less data. In general, the random ensembles did not sacrifice too much on accuracy but gained a high speedup of train-

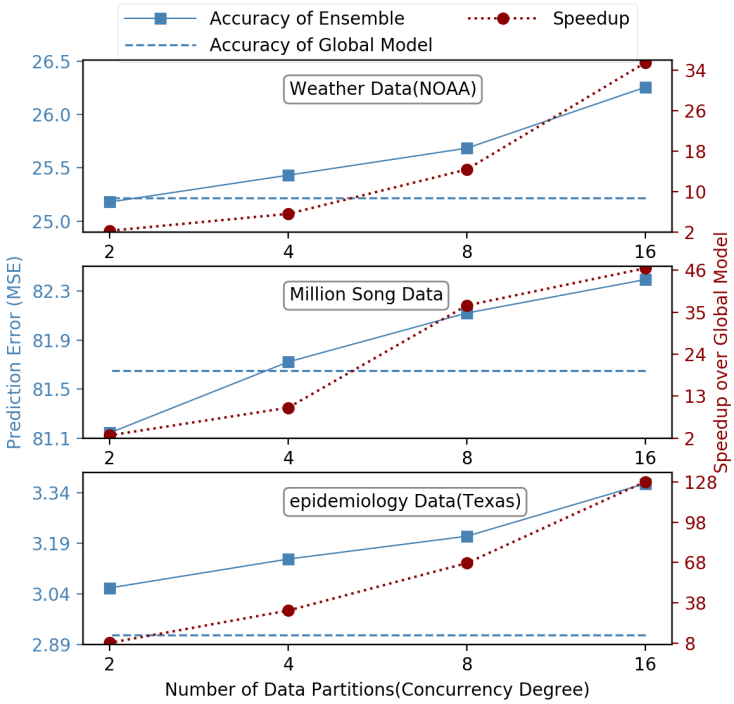
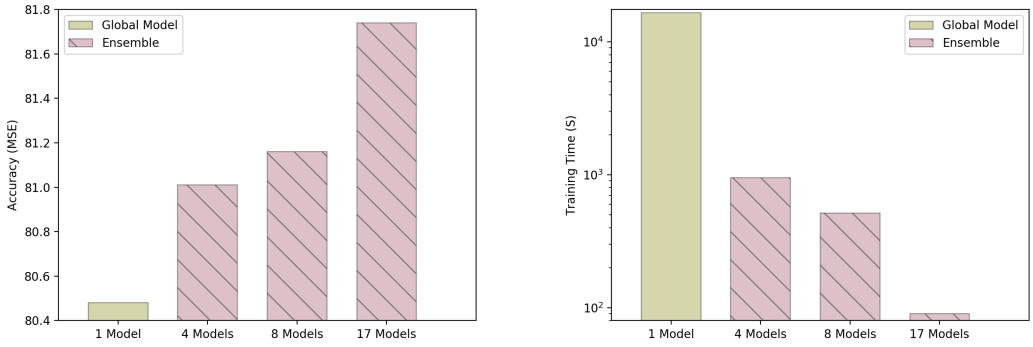
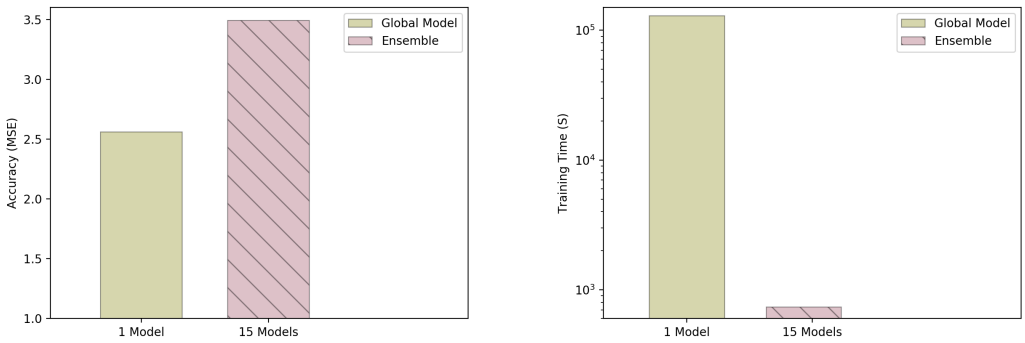


Figure 3.6: Ensembles' accuracy and speedup of training time over the global model. Both approaches were built on the 50% of the original datasets.



(a) **(b)**

Figure 3.7: Ensembles versus global models, using the million song dataset. (a) The accuracy (MSE). (b) The training time.



(a) **(b)**

Figure 3.8: Random ensembles versus global models, using epidemiology dataset. (a) The accuracy (MSE). (b) The training time.

ing time over the global models. Compared with the global models, our methodology enabled constructing random ensembles with reasonable performance in a very short time.

3.4.8 The Effective Ensemble Size

We applied our greedy algorithm for reducing the size of the random ensembles for different partition sizes to the three datasets. For each dataset, we investigated three relative partition sizes, 0.78%, 1.6%, and 3.125%, which resulted in partitioning the data into 128, 64, and 32 blocks, respectively. Figure 3.9 shows the accuracy improvement of the random ensembles for each individual model that the algorithm chose to be a member of the pruned ensemble. The end of each curve in the figure shows the final number of individual models in a pruned ensemble.

Our algorithm demonstrated the ability to reduce the ensembles' size and improve generalization. This indicates that certain models contribute to variance reduction, which in turn improves ensemble performance. Furthermore, the figure also shows a relationship between the reduced ensemble size and data variability. Random ensembles that were built on 128, 64, and 32 partitions of the epidemiology dataset (high variability) were reduced to become 88%, 80%, and 75% smaller. Using the weather dataset (with comparably lower variability), the random ensembles of sizes 128, 64, and 32 were reduced to become 32%, 72%, and 84% smaller.

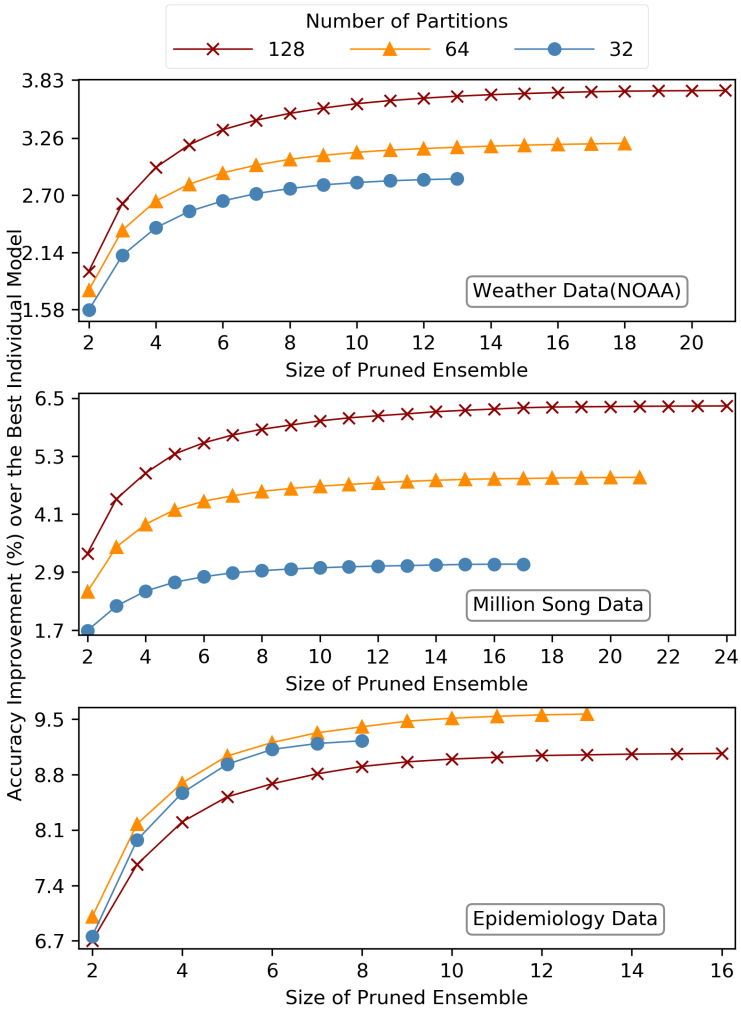


Figure 3.9: The effective ensemble size for different numbers of partitions.

Chapter 4

Methodology

We propose a methodology that enables gaining insights from large datasets for supervised problems. The methodology tries to address the main inefficiencies that are associated with the training process of the global model on a large dataset that is hosted by multiple machines. Mainly, the methodology attempts to reduce the long training times, allow leveraging various learning algorithms for solving a single problem, efficiently use computational resources, and ensure scalability by just adding more machines to deal with the increases in data volumes. To accomplish our objectives without sacrificing the prediction accuracy, we build an ensemble with several models that need to be independently trained on different subsets of the given dataset.

In our methodology, there are two training phases involved. However, the second phase is optional and only useful for providing further improvement to the ensemble. In the first phase, we construct the base ensemble by training its models and building its gate function. In the second phase, we improve the performance of each model in the ensemble by relying on the stacking technique, resulting in an extended ensemble that outperforms the base one.

For building the base ensemble, we perform three main tasks to construct the base ensemble for a given dataset. The first task involves partitioning a given dataset into several subsets with manageable sizes and dispersing them over multiple machines. In the second task, a learning process is collocated with each data partition to train the models independently and in parallel. The trained models are then used to construct an ensemble. The partition information that was used to partition the original dataset is used to build a gate function of the ensemble. The ensemble uses the gate function to find and apply the proper model for each given observation. The last task is the integration phase, where the formed ensemble relies on its gate function and models for predicting a given observation.

Before data partitioning, we sample from the given dataset to create a test dataset that is not included in the learning process but used for ensemble assessment. The main tasks that are involved

in our methodology for constructing the base ensembles are shown in Figure 4.1. As mentioned above, the performance of the base ensemble can be extended by applying the stacking method on each model in the ensemble. The extended ensemble adds a little more complexity, but it usually provides better prediction accuracy than the original ensemble does.

The methodology enables the construction of scalable ensembles that have the potential to generalize well. The main intention beyond the use of partitioning methods is to enable concurrent learning from voluminous datasets and to achieve good prediction accuracy by capturing the patterns that are associated with different regions of the input space. In such a case, several models participate in gaining insights from large-scale datasets, and each model is specialized for particular subsets of observations. For a given input feature vector, the created ensemble uses the gate function to find the specialized model to maximize the prediction accuracy.

In this work, the ensembles that are constructed by using our methodology are referred to based on the employed partitioning method. For instance, an ensemble that is built by using randomly partitioned data is referred to as a *random ensemble*. An ensemble that is built on data that is partitioned based on the similarities in input space is referred to as an *input ensemble*.

4.1 Data Partitioning

Several partitioning methods can be used to provide high-quality training sets to the learning algorithms for building accurate models. Data partitioning is the main component that many learning algorithms rely on to extract patterns and trends in data (such as in CART algorithms [113] and support vector machines [114, 115]). Besides, it has been used to simplify a complex problem by dividing it into simpler problems. For instance, the mixture of experts approaches [58, 59], divide and conquer methods [116, 117], and other proposed approaches [118, 119] rely on partitioning strategies to solve complex problems.

These methods not only control the contents of the training sets that are used for building the models but also guide how the learned pieces (models) should be used for making a prediction for a given observation. Additionally, data partitioning enables training the individual models in

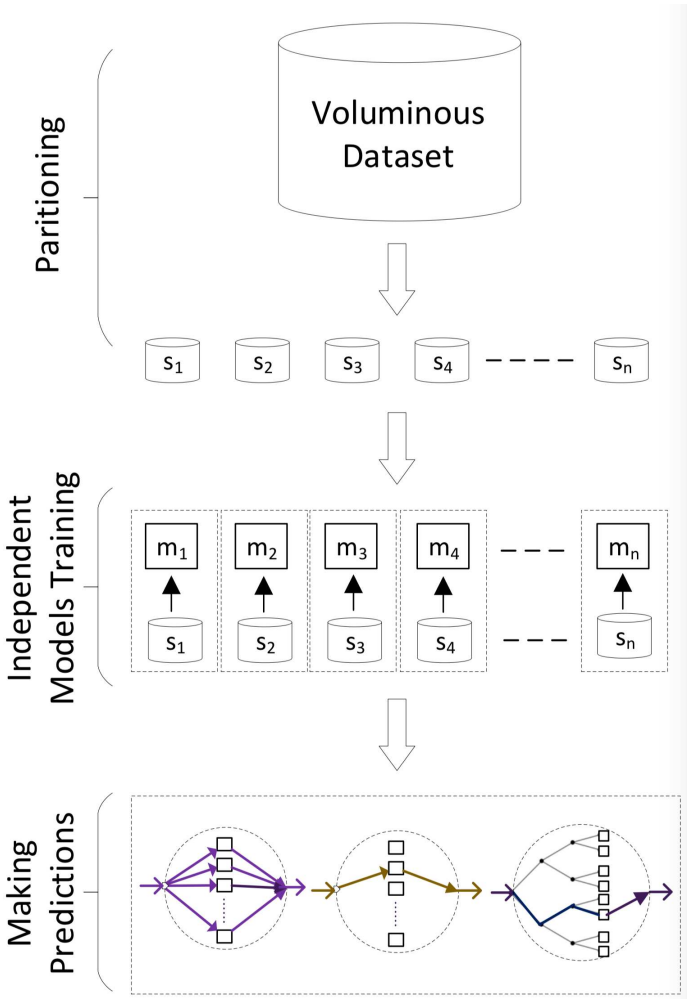


Figure 4.1: The main tasks involved in the methodology for scalable models training and accurate ensemble construction.

their natural way without involving any modification, which will support all of the appropriate algorithms, even the ones that are working sequentially. Some partitioning methods guarantee a scalable solution and exceptional performance without encountering the training challenges that are associated with global models. Additionally, these methods can be combined to increase scalability and enable finer-grained learning from voluminous datasets.

4.1.1 Random Partitioning

The given voluminous dataset is randomly partitioned into subsets with manageable sizes to enable building each model on a single machine. Each training set can be seen as a sampled version from the original dataset. Because a voluminous dataset is usually collected on several machines, the models can be directly built on the data partitions without the need to shuffle the data. This approach enables building ensembles with a significant reduction in training time without sacrificing too much accuracy.

Random partitioning enables building models on relatively small training sets, which in turn leads to the creation of models that are not very accurate. The generated training sets will include similar but not identical data observations—building models with high complexities on the assigned training sets will result in overfitting. The created models will provide accurate predictions for seen data but poor predictions for unseen data. These models will produce predictions that are not highly correlated, with low bias but high variance. Such a property will enable building a bagging ensemble (referred to as random ensemble) that averages the predictions of all of the models to reduce the prediction variance and provide an accurate forecast for a given input vector.

4.1.2 Controlled Partitioning

Controlled partitioning relies on some partitioning criteria to divide the input space into disjoint groups. Fitting models on the generated regions enables the construction of an ensemble with models that are specialized for different subsets of observations. For a given observation, the ensemble uses the partition information to identify the observation's region and then uses the

responsible model for making a prediction. Ensembles with specialized models promise accurate predictions and high generalization capability.

Spatial Partitioning

The dataset with spatial properties is partitioned into spatial extents. Observations belonging to the same spatial area usually share common characteristics and enable the learning process to learn the associated patterns and trends efficiently. Spatial observations include features pointing to locations in space; for instance, the locations could refer to pixels in an image or points on the earth (geographic). In the presence of geographic features (latitude and longitude), the use of the Geohash algorithm can enable the effective geographic partitioning of large datasets. The Geohash algorithm is fast and applicable to partitioning streamed data. We applied it in one of our studies to partition streamed geospatial data (see §5.1.3 for more details about Geohash).

Input-Based Partitioning

In the absence of spatial properties in the datasets, the dataset can be divided into different groups based on their similarity in the input space. Each group includes observations occurring in proximate portions of the feature space. To simplify the partitioning process, we do not attempt to find the optimal number of groups (clusters). The number of groups is determined based on the size of the given dataset and the available computational resources. The employed partitioning method should seek to produce disjoint groups, where the similarity between observations within a group is higher than that existing in different groups.

Several simple techniques (e.g., locality-sensitive hashing methods [120–122], ball tree algorithms [123, 124], and clustering methods [125–128] can be applied to achieve the desired objective, but they become inefficient for high-dimensional datasets. The analysis of high-dimensional data poses a challenging problem that has a significant influence on the efficiency of computational and statistical operations [6]. A high number of features leads to high consumption of storage space, an increase in the processing time, and an efficiency reduction for the statistical methods.

The *curse of dimensionality* causes this problem and makes clustering high-dimensional data a challenging task.

The curse of dimensionality refers to the data analysis problems that occur in high- but not in low-dimensional space [129]. The problem can be seen as fixed data points become increasingly sparse as the dimensions increase, and in turn the data become insufficient to cover the space. To illustrate how the data becomes sparse in high-dimensional space, let us assume that we have generated 1,000 uniform distributed data in a two-dimensional space. If each dimension is divided into only 10 equal intervals, we will have at least one point in each cell of all available 100 cells. However, if we perform the same task in 10-dimensional space, less than 0.00001% of the cells will contain one data point and the remaining cells will be empty. The authors of [130] have shown that the number of occupied cells in high-dimensional space is equivalent to the data size. As a result of such a phenomenon, the difference between the nearest and farthest neighbor of a random point approaches zero as dimensionality gets higher [131]. That is, the Euclidean distances between all of the data points are approximately the same and tend to be indistinguishable. Addressing this issue involves exponentially increasing the number of data points with the dimensionality growing, which is impossible to achieve for high-dimensional space.

Several methods [132–136] have been proposed to overcome the curse of dimensionality. Dimensionality reduction techniques are commonly used to minimize the effect of the curse of dimensionality. In this process, the input features that contribute to the largest portion of the explanatory power are identified and selected to represent the data. Usually, the input features are ranked based on their analytical strength to the target. Input features with the highest rank are then selected to represent the most influential features. Methods such as principal component analysis, correlation analysis, least absolute shrinkage and selection operator, and random forest ensemble can be used for dimensionality reduction.

Output-Based Partitioning

The input space of datasets can also be partitioned based on the similarities in the output space. The involved process attempts to find the best split points in the input space to create regions

with maximum purity in terms of the output space. The algorithm that is used for decision tree induction [137] can be used to achieve such partitioning. Iteratively, the algorithm partitions the data by finding the split points in input features to minimize the variance of a continuous target (in a regression case) and separating the values (classes) of the discrete target (in a classification case) within the generated data portions. The process will continue partitioning the data until a stop criterion (such as reaching the minimum number of observations in the data portion) is reached. Setting the minimum number of observations for stopping partitioning can help to obtain the desired number of training sets.

This partitioning method puts all of the observations that lead to similar target values in a separate subset, which in turn, enables the construction of an ensemble with models that are specialized for different areas of the output space. The ensemble whose models trained on the generated data portions uses the thresholds of the split points that are found in the partitioning process to identify the region of a given observation and then employs the associated model for making a prediction.

Custom Partitioning

The dataset can also be partitioned in a way that maximizes the prediction accuracy of the formed ensemble. The data must be partitioned before starting the training process of the ensembles' models to enable a scalable construction of ensembles. One way to achieve such partitioning is by relying on one or more features that have the highest correlations with other features. The generated subsets will include observations that are associated with different intervals of the partitioning features. For a given observation, the composed ensemble can quickly identify and use the responsible model. Another way to achieve this data partitioning is by finding boundaries in the input space to divide the given dataset into subsets. The partitioning boundaries should separate the subsets that share some common characteristics. For example, data for a face recognition problem can be divided into subsets that cover different regions of the face. Building models on such subsets will enable having models that are specialized in different areas of the face.

4.2 Independent Models Training

The models are trained independently in a conventional way on a collection of machines. We divide each data portion into two subsets (the training set includes 80% of data, and the test set contains 20% of the data). We use the training sets to train the models, whereas the test sets are used to create the new training sets for training the stacked models that form the extended ensemble. Before training the models, the most influential input features are determined independently for each model. The training processes are launched on the machines hosting the training sets to ensure data locality during the model creation process. We rely on cross-validation to tune the hyper-parameters and choose the best model for each training set. Training the models independently on subsets that are created by controlled partitioning promises the following advantages:

- Concurrent creation of independent models in a distributed environment significantly speeds up the training process. Consequently, gleaning insights from the underlying datasets involves a *short training time*.

- Individual model instances do not communicate with each other, nor are there any synchronization barriers that span model instances during their training. As a result, independent model training enables *efficient resource utilization* and *high scalability*.

- Independent models training facilitates a *simple fault-tolerance mechanism* because failing learning processes do not affect the other running processes and any failed process can be relaunched independently.

- Training models independently enables leveraging different algorithms for solving a single problem. Employing the proper learning algorithm for each training set leads to *prediction improvement* of the individual models and the formed ensemble.

- Training each model on observations that share some common characteristics promotes the capturing of hidden patterns and the trends in data. In such a condition, the observations that seem to be outliers in the entire dataset will be evident in their subsets. Therefore, learning

from subsets that are generated by controlled partitioning promises *efficient learning* from the voluminous datasets.

- The models can be trained by applying the algorithms in their natural way, without the need to modify them. Independent models training supports *leveraging any applicable algorithms* (that are even the ones working in a sequential manner) without constraints.

4.3 Extended Ensemble

The base ensemble is constructed by using specialized models that could learn the local patterns and trends, but none of them could efficiently learn the global trends and the patterns that stretch over multiple regions. Each specialized model might partially learn these patterns. Therefore, the accuracy and generalization for each region could be improved by exploiting the gained insights. Such an objective can be accomplished by relying on some cooperation between the specialized models, which could be consulting neighboring models or relying on the stacking. Models that are specialized for neighboring regions can vote or average their predictions to provide an accurate prediction for a given observation coming from one of their regions.

Relying on the stacking technique is another way to incorporate the insights of the specialized models to build a meta-learner (referred to as a stacked model) for each region. Stacking is a type of ensemble learning that builds a model on the predictions of the other models to provide a prediction that is more accurate than any of the combined predictions are [45] (see §2.2.3 for more details). By applying the stacking method, the formed extended ensemble will compose stacked models that are specialized for each region. Each of the stacked models will improve the prediction accuracy of the assigned region, resulting in an improved extended ensemble.

After building the specialized models for the base ensemble, creating new training sets is needed to build the stacked models for the extended ensemble. We create a training set for each region by applying all of the specialized models to the region's data for making predictions. To avoid overfitting, the region's data should not be used for training specialized models. For each region, the newly created training data will encompass the models' predictions as input features

among the values of the original target. The new training sets can be created in parallel by using the machines that host the original data. Subsequently, a training process is collocated with the new datasets to build the stacked models in parallel independently.

4.4 Hybrid Ensemble

For some partitioning methods, our methodology does not provide a mechanism for controlling the sizes of the resulting partitions. To avoid building a global model on a large data partition that is hosted by several machines, the partitioning method can recursively be applied to large data partitions until a desirable partition size is met. Additionally, the partitioning methods can be combined to construct a hybrid ensemble. Combined partitioning enables more finer-grained learning and avoids distributed training of a model on data that is hosted by multiple machines. The following scenarios illustrate some examples of how the partitioning methods can be combined to construct a hybrid ensemble:

- *Combining controlled and random partitioning methods to reduce the construction time of the hybrid ensemble*. For example, we can partition the data based on the similarities in the input space and then partition the large data portions. In such a case, each specialized model built on a large training set is a random ensemble.

- *Combining multiple controlled partitioning methods for finer-grained learning and constructing a hybrid ensemble with better performance than that of an ensemble that has been built by using one partitioning method*. For example, we can apply spatial partitioning to divide the original data into spatial extents. Next, each spatial extent can be partitioned based on similarities in the output space. In this case, the hybrid ensemble will have only ensembles that are specialized for different spatial extents, and each ensemble includes models that are specialized for different regions of the output space.

- *Combining multiple controlled partitioning methods with a random partitioning for faster construction of a hybrid ensemble that performs better than an ensemble that is built by*

using a single partitioning method. For example, the data is partitioned based on similarities in the input space, every data portion is then partitioned based on similarities in the output space, and finally, each large training set is randomly partitioned. In the hybrid ensemble, we will have ensembles that are responsible for different regions of the input space, and each ensemble includes specialized models for different areas of the output space.

Chapter 5

Enabling Scalable Anomaly Detection over Spatiotemporal Data Streams

We have built an ensemble that is based on geospatial partitioning to enable a scalable and real-time detection of anomalies in spatiotemporal data streams [138]. A massive number of models were trained on different geospatial extents to construct the ensemble. A real-world petabyte climate dataset was used to construct and evaluate the geospatial ensemble.

The datasets that we considered were composed of streams that continually report readings from observational devices. Some of these observational devices, such as radars and satellites, can remotely sense features of interest, while other features may require in situ measurements by devices such as piezometers and barometers. The measurements are reported as observations in discrete packets that comprise a stream. Each observation comprises n-dimensional tuples with each dimension representing a feature of interest. Examples of such features include temperature, air pressure, humidity, etc. Features may also have linear or non-linear relationships with each other; for instance, there may be a relationship between temperature and precipitation at specific geographic locations. Ultimately, these relationships result in a classification by our ensemble as either normal or anomalous with a corresponding degree of irregularity. An *anomaly* may constitute an irregular event, inconsistent sensor readings, or other types of situations that result in data points that are outside of the expected norm.

Anomaly detection is a precursor to the discovery of impending problems or features of interest. Timely detection of anomalies is critical in several settings. Often such detection needs to be made in real-time to be able to detect potential emergencies. Our specific problem relates to voluminous data streams and anomaly detection that accounts for the evolution of the feature space over time. Also, given the rate of data arrivals and the volumes that are involved, human intervention is rendered infeasible. This work is applicable in domains where observations have geospatial

and chronological components, including atmospheric sciences, meteorology, environmental and ecological modeling, epidemiology, and traffic monitoring.

The range of values that each feature takes on may be rather large, and simple checks for breaching the upper and lower bounds are generally not viable. Other dimensions, such as location and time, may determine whether a particular feature value is considered anomalous. For example, temperatures at night are often lower than those in the day are for a particular location. Also, in the case of geospatial data, what is considered normal will vary by region, and anomaly classifications must account for this as well.

We developed an anomaly detection framework that partitions the streamed data into geospatial extents and trains the ensemble's models to be specialized for different geospatial areas. The framework was developed to enable a scalable and real-time detection of anomalies in streaming data. The developed framework with the geospatial ensemble enables tackling the following challenges:

1. Streams have no preset lifetimes, and readings arrive continually. This makes it infeasible to inspect all previous records when making a classification.

2. Observations are multidimensional. Individually, feature values (i.e., values along a dimension) may be normal, but when collectively accounting for all of the dimensions, the tuple may be anomalous.

3. There are spatial and chronological dimensions that are associated with feature values. How features evolve is spatiotemporally correlated. What is considered normal for a particular geographical extent would be considered anomalous for another. Anomaly classifications must take these into account.

4. What is considered anomalous evolves over time. A good exemplar of this is temperature readings. Over the past several years, average temperatures at various geographic locations have trended higher overall, but they have fluctuated from year to year.

5. The combination of legitimate feature values is very large. Building a single model of what constitutes anomalous data is infeasible and impractical.

6. Anomaly detection must be done in real time despite the constant feature space evolution and data volumes that are involved.

The framework constructs the ensemble and employs its specialized models for making predictions. Because the purpose here is to evaluate the scalability of our methodology, it is more practical to integrate our framework with a distributed system. For this study, we used Galileo [139,140], a distributed storage system, which was tasked with supplying new data to the anomaly detector framework as it was streamed into the system for storage. When an anomaly is found, offending records are flagged for further analysis.

In the remainder of this chapter, we will briefly introduce Galileo, illustrate the anomaly detection framework and its integration with Galileo, show the construction of the geospatial ensemble, and show some evaluation results of the framework.

5.1 Distributed Storage Framework

The distributed storage system Galileo is responsible for managing incoming observations across its storage nodes. Galileo offers spatiotemporal partitioning functionality and distributed indexing and query support, and it ensures that our approach can scale up as resources are added. In this work, we added several anomaly detection features to Galileo, including support for visualization. However, it is worth noting that the components in the anomaly detection framework are loosely coupled and could be used with a variety of distributed storage systems.

5.1.1 Galileo

Galileo is a high-throughput distributed storage framework that was designed for managing multidimensional data. The system's network design is modeled as a hierarchical DHT, which allows incremental assimilation of storage resources and the use of multitiered hash functions to enable the development of novel partitioning schemes. By focusing on spatiotemporal datasets,

Galileo provides functionality that is generally not provided by standard DHTs, such as expressive query support, time series analysis capabilities, and polygon- or proximity-based geospatial retrieval functions. Galileo is decentralized, and it is composed of a network of storage nodes that facilitate data management.

5.1.2 Storage Nodes

Each storage node in Galileo manages a single instance of the anomaly detection framework. Based on the classification output by the framework, the storage node can take appropriate action (which may vary across problem domains). The storage node does not need to wait for the result of each evaluated observation during the storage process, and it will assume that the observations are normal until informed otherwise.

The storage node treats the anomaly detection framework as a black box. It cooperates with the coordinator asynchronously to detect anomalies without knowledge of the particular detection algorithm that is being used. However, some of the framework's behavior can be controlled. Configurable parameters include enabling or disabling adaptive classifications as well as how fast adaptations should be made. Additionally, the storage node can also modify the size of the geospatial area that is assigned to each detector instance, which is critical in situations where the geographic scope of the node changes because of fluctuations in the underlying resource pool.

5.1.3 Geospatial Data Partitioning

Data partitioning in Galileo is done based on the observed Geohash prefixes of incoming records [141]. The Geohash algorithm is a geocoding scheme that divides the earth into a hierarchy of spatial bounding boxes that are referenced by Base32 strings. Figure 5.1 provides an example of how the Geohash algorithm works. For example, the latitude and longitude of the coordinates N 28.8927, W 81.9796 would map to the Geohash string *djjsqeb2*. Longer strings result in higher precision coordinates, and two Geohash strings with the same prefixes will be located in a similar geographic region; a Geohash of *djjs* would describe a broader area that also encompasses

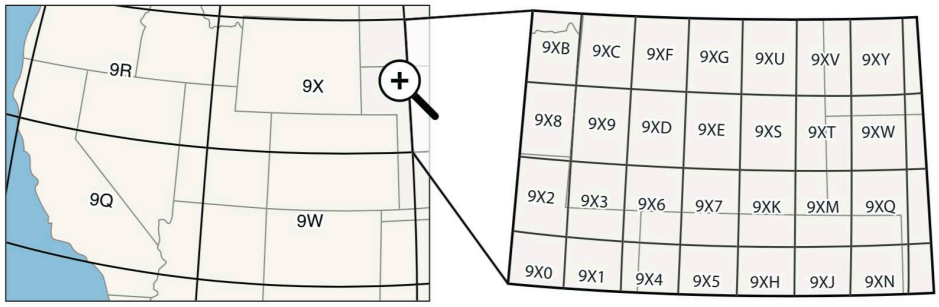


Figure 5.1: A visual overview of the Geohash geocoding scheme showing two hierarchical divisions of the western United States.

the coordinates mentioned above. These prefixes ultimately determine where data records will be routed, processed, and stored.

The prefix length that is used for partitioning in Galileo is configured based on the size and scope of the particular deployment. The default length, four characters, assigns incoming records to 39.1km $\times$ 19.5 km regions. This naturally divides a broad geographic region into manageable pieces that can be maintained by individual nodes in the system, and it also allows future reconfiguration to scale up and down to meet changing problem requirements. Moreover, this scheme enables data partitioning to be carried out in a decentralized manner while facilitating parallel computations.

Classification accuracy is also influenced by how the data are partitioned across nodes in the system. If a small number of partitions are used, classifications must be made across a broader range of geography and ambient conditions, whereas a fine-grained partition allows for similarly fine-grained classifications. The data are partitioned once again at each node by using a longer (finer-grained) Geohash string to divide the workload up among anomaly detector instances, making analysis across a hierarchy of spatial regions possible.

5.2 Anomaly Detection Framework

Our design simplifies the integration of multiple anomaly detection algorithms. The framework assists these algorithms by controlling the amount and type of observations that are made visible to them. Rather than having an all-encompassing instance of the anomaly detection model,

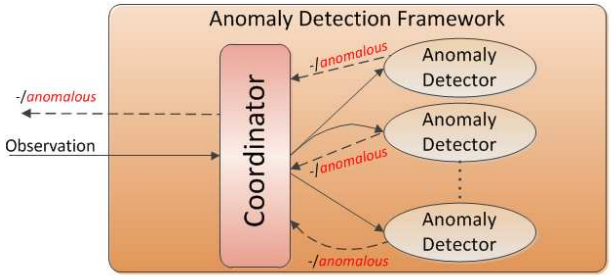


Figure 5.2: Overview of the architecture of the anomaly detection framework (a single coordinator and multiple anomaly detectors).

our approach allows building ensembles that comprise multiple model instances that can be dispersed over the collection of storage nodes; each model instance is responsible for a particular geospatial scope, allowing for specificity in the classification of anomalies. This also assists the algorithms in accounting for the spatial correlations between features because each model instance learns from the data in a particular geographical region. The use of multiple model instances ensures two features: (1) scalability: as the number of storage nodes increases, the number of model instances scales as well, with each responsible for a specific geographical scope and (2) high throughput: there are no bottlenecks because observations do not need to be funneled through an all-encompassing model. Incidentally, properties (1) and (2) contribute to faster turnaround times for classifications. Finally, our approach allows multiple algorithms to operate on the same set of observations for a geographical scope. Such combinations allow us to exploit the properties of different techniques for minimizing misclassifications—specifically, observations that are classified as anomalous by multiple algorithms are likely to be truly anomalous. However, a particular algorithm may capture subtle variations that were missed by the others.

The framework contains two main components, a single coordinator and multiple anomaly detectors. The coordinator supervises the training processes and the use of the detectors on a single machine without involving network I/O. Figure 5.2 illustrates an overview of the architecture of the framework and how the predictions are made. For enabling a scalable anomaly detection, multiple machines need to be used, and an instance of the framework has to be created on each machine.

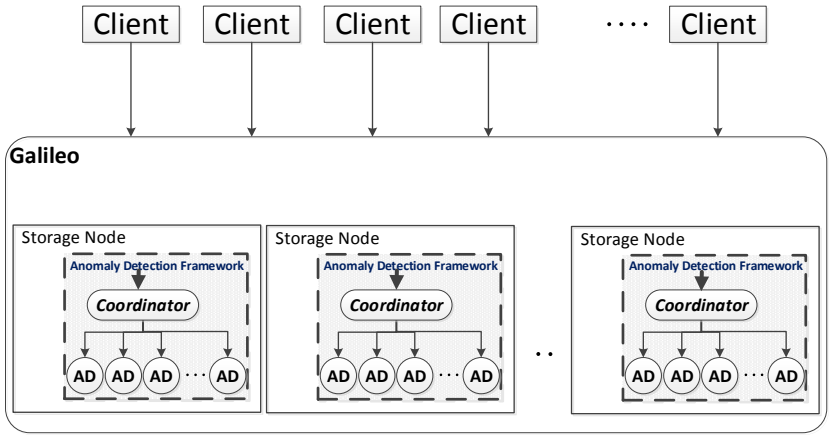


Figure 5.3: Integration of the anomaly detection framework with Galileo. The components include the storage nodes, detection coordinators, and anomaly detectors (AD).

5.2.1 Storage Node Integration

Each distributed storage node manages a single instance of the detection framework, which contains a single coordinator process and multiple detector instances. Figure 5.3 demonstrates the integration of the anomaly detection framework with Galileo. As observations are streamed into the system, the storage node forwards them to the framework that directs them to the appropriate anomaly detectors based on their geographical location. Figure 5.4 depicts how events are passed between each of the components; observations sent from the coordinator to the anomaly detectors are queued and classified in order. The detector tags observations that are classified as anomalous by triggering a callback, anomalousAlarm. This method can be overridden to perform postprocessing steps, which generally includes flushing the identifiers of the anomalous observations to disk for further analysis.

To ensure high throughput and minimize I/O, data management tasks in our framework are performed asynchronously. Upon receipt of a new observation, the storage node updates its in-memory metadata graph and generates a unique file identifier for the data. Once this step is complete, the observation is encapsulated in an observation container and passed to the anomaly detection coordinator for processing. Note that no disk I/O has occurred up until this step; only after the anomaly detection process begins is the observation placed in a queue for on-disk storage. This

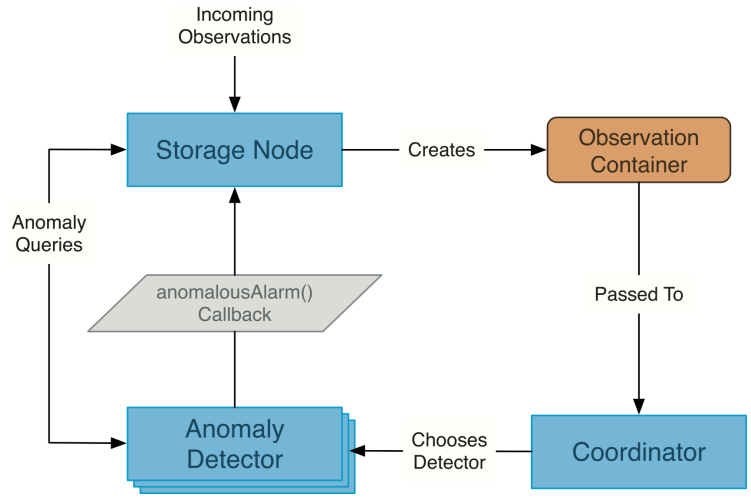


Figure 5.4: The anomaly detection framework processing cycle.

approach helps to interleave I/O and processing operations on separate threads. Further, when an observation is classified as anomalous and assigned a degree of irregularity, the metadata graph is updated without requiring any changes to files that are stored on the disk.

5.2.2 The Coordinator

The primary task of the coordinator is creating and managing anomaly detector instances. The coordinator receives observations from the node and then forwards them to the appropriate detector instances based on the spatial partitioning scheme in use. If a detector instance for the region does not exist, the coordinator creates a new instance. The coordinator uses an abstract class called AnomalyDetector to load and manage detectors that can use different detection algorithms. This makes it possible to add, manage, and combine different anomaly detection techniques concurrently. Also, because of the loose coupling between the coordinator and detectors, any changes made to either component will not require corresponding changes in the other.

Our approach ensures that available cores and execution pipelines are used efficiently by managing a thread pool to facilitate parallel detection activities. During initialization, the coordinator creates a thread pool of a configurable size and provides its reference to the anomaly detectors. Detector instances submit classification tasks that contain a queue of incoming observations to be processed concurrently, and the queues can be updated as more observations are assimilated.

5.2.3 Anomaly Detector

The primary concern of the anomaly detector is to collect training data from the coordinator, build a model (or multiple models) for the recevied data that cover a finer-grained geospatial scope, and then use the appropriate model to detect observations whose behaviors are outside the observed norm. Collecting the training data and training a model is done automatically for each detector regardless of the actual implementation of the anomaly detector instances. Each anomaly detector operates in three phases. It starts in the data collection phase, where the detector collects observations in memory and transitions to the training phase when the amount of data collected reaches a configurable threshold. The coordinator can also override the threshold to begin training immediately if the particular problem warrants such an action. In the training phase, a training task is created and queued to the thread pool. While the training task is running in a separate thread, observations are buffered for classification until the training process is complete. Finally, in the classification stage, the models are used to classify incoming data.

Our framework allows anomalies to be detected in an online manner, enabling evaluations to occur on the fly without retraining the underlying models. Further, the detectors can adapt to changes that occur over time. For example, temperature values at a particular region that were unusual 30 years ago may be considered normal now. In such a case, detector implementations are given the opportunity to adapt to these changes and not tag such data as anomalous. However, this feature can be disabled to handle situations where adaptation is not beneficial; for instance, models dealing with heart rate measurements do not need to be adapted because valid measurements do not change over time. When an anomaly is detected, implementations can include a degree of irregularity to demonstrate the relative magnitude of the anomaly. Finally, the detector can serialize and deserialize its model to and from disk to allow migration to other systems and to cope with failures or planned outages.

5.2.4 Anomalies: Group and Experts Approach

The anomaly detector instances managed by the coordinator at a storage node work independently on different geospatial data, but they can also cooperate with one another to achieve a consensus in situations where classifications may be suspect. Because observations from bordering geospatial regions should have similar behavior, instances operating on nearby regions are consulted to confirm suspect classifications. If a detector instance receives an outcome indicating a suspect anomaly, it informs the coordinator, which forwards the suspect outcome to detector instances in nearby geographic regions. In this situation, the observation is considered anomalous if a majority of detectors confirm the anomaly.

Using different detection algorithms is another scenario in which a group of experts approach proves to be useful. In such a case, the coordinator passes data samples to anomaly detectors, each of which implements various detection algorithms. At each anomaly detector, the final classification for the sample is then made based on the results coming from each instance, where the observation is deemed anomalous only if more than half of the instances confirm the anomaly.

5.3 Reference Anomaly Detection Implementations

We designed our reference anomaly detectors with three primary goals in mind: (1) real-time classification, (2) minimizing human intervention, and (3) domain neutrality, allowing the detectors to be used for different problems without extensive adjustment. While achieving real-time classification is mostly a function of the efficiency of the algorithms that are used, avoiding user-defined thresholds was a critical factor in achieving goals (2) and (3). An overview of the design is shown in Figure 5.5; the classifier has access to the model's parameters and can update them at run time, making adaptation possible. This feature was exploited in our implementation by modifying the model parameters at a configurable time interval. We incorporated support for two different algorithms—density- and distance-based multidimensional clustering—to detect anomalies in an n-dimensional feature space. We support a wide variety of functionality in our reference imple-

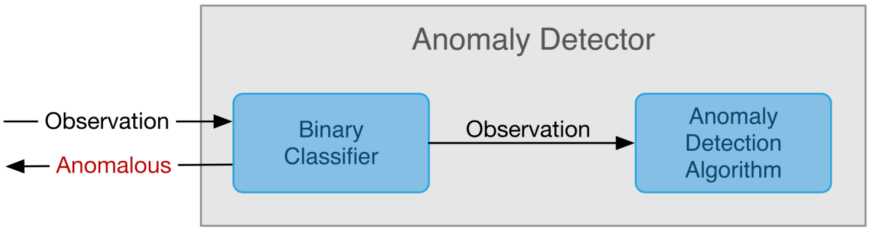


Figure 5.5: An overview of our reference anomaly detector designs, which are composed of an Anomaly Detection Algorithm and a Binary Classifier.

mentations to ensure that the interfaces in our framework are generalizable and can be leveraged by other anomaly detector designs.

5.3.1 Anomaly Detection Algorithms

We implemented four anomaly detection methods to demonstrate the flexibility of our framework: density, distance, Bayesian, and ensemble. Our density, distance, and Bayesian methods iterate over two main steps: (1) assigning observations to clusters and (2) updating cluster parameters based on the assignments. These steps are repeated until the models converge. The density-based approach assumes that the data points were generated by a finite number of mathematical distributions and focuses on iteratively improving the distributions' parameters to fit the training data better. On the other hand, the distance-based algorithm clusters observations that are in close proximity on the multidimensional hyperplane and then refines the cluster centroids to minimize the overall distance between related observations. Our Bayesian approach, based on Dirichlet process mixture models (DPMMs), supports an infinite number of mixture components. As a result, the number of clusters does not need to be predefined. Finally, our ensemble approach uses isolation forests to generate decision trees based on incoming observations and uses path lengths to determine which data points are anomalous.

The density-based anomaly detector employs GMMs built with EM, while the distance-based anomaly detector uses a clustering model built with K-means. These techniques, as well as the ensemble approach (isolation forest), rely on implementations that are provided by the WEKA

software package [142]. Our Bayesian approach was supported by functionality that was sourced from the Datumbox Framework [143].

Density-Based Anomaly Detection: GMMs and EM

In general, mixture models are used to represent the densities of multidimensional data. Gaussian mixture models can approximate almost any continuous density distribution with high accuracy [144]. Gaussian mixture models achieve this by (1) applying a sufficient number of Gaussians, (2) fitting them to the data by adjusting parameters such as the means and covariances, and (3) choosing the coefficients for the linear combinations of different densities. Gaussian mixture models are also used by many techniques that build models based on class densities. For instance, a mixture of Gaussian densities is used in a linear and quadratic discriminant analysis to identify nonlinear decision boundaries in data [144].

We estimate GMMs parameters by using EM. Expectation maximization is an iterative refinement technique that is used for estimating the maximum likelihood of parameters in probabilistic models. For GMMs, an EM is used to determine the means and covariance matrices for each Gaussian component.

Distance-Based Anomaly Detection: K-means

K-means is a clustering technique that assigns N observations to K clusters. The algorithm starts with an initial set of clusters that are represented by their centroids and then iterates over two main steps: (1) assigning observations to clusters based on their proximity to cluster centroids and (2) updating each cluster's centroid by computing the mean of the assigned observations. The two steps are repeated until no changes occur in the clusters. We use canopy clustering [145] as a preprocessing step to determine the initial cluster centroids and reduce the overall number of distance calculations that are required for the K-means to converge. Our K-means detector implementation also supports two functions to measure the distances of observations from the cluster centroids: Euclidean and Mahalanobis. Both distance measures use Formula (5.1) to compute the distance

between two points (x and y). The distance is scaled in different dimensions based on the weight matrix ($\boldsymbol{W}$):

$$D(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})\mathbf{W}^{-1}(\mathbf{x} - \mathbf{y})^T. \tag{5.1}$$

The Euclidean distance uses the identity matrix as its weight matrix, which results in the features being equally weighted. Thus, Euclidean distances do not consider the correlations between features. On the other hand, the Mahalanobis distance [146] considers the correlation between features by weighting the equation with the covariance matrix Σ. In this case, features that are correlated will have a more substantial influence on the distance measure.

Bayesian Anomaly Detection: Dirichlet Process Mixture Models

Dirichlet process mixture models is a Bayesian unsupervised learning technique that does not require predefining the number of clusters, unlike many other approaches [147–149]. This property is particularly useful because it allows for dynamic adaptation in the number of clusters as the data points evolve. Dirichlet process mixture models construct a single mixture model in which the number of mixture components is infinite, and extensions can allow incoming data points to be assigned to multiple clusters. In our implementation, we used a Gaussian base distribution and a scaling parameter (α) of 0.01.

Ensemble Anomaly Detection: Isolation Forests

Isolation forests represent a particularly unique approach to anomaly detection. The algorithm generates decision trees by inspecting incoming observations and splitting them randomly based on the observed minimum and maximum values of each feature. This process isolates abnormal observations, as only a few features must be deviated from the norm to produce an irregular path through the trees. Path lengths are then used to determine an anomaly score (which is, in turn, used to calculate the degree of irregularity in our framework). Because of the stochastic nature of this approach, the process must be repeated several times to gain confidence in the resulting anomaly

scores. However, isolation forests have linear time complexity and low memory requirements, allowing them to scale up to handle large datasets [150].

5.3.2 Binary Classifiers

The clustering techniques that we have described are used to create binary classifiers, which are responsible for making final decisions about whether an observation is anomalous or not. The classification process begins by clustering incoming observations, deciding whether they are anomalous, and then updating model parameters for adaptation (if enabled). Our reference implementation computes thresholds that determine whether a given input sample is anomalous or not, with raw outputs used to assign degrees of irregularity to the data. The following subsections describe how the classifiers carry out anomaly detection and adaptation.

Anomaly Detection without Threshold Parameterization

Given the volumes of data considered in this work, human intervention in the anomaly classification process must be avoided. Additionally, manually choosing thresholds that accurately classify observations as normal or anomalous is often challenging [151, 152]. Further complicating matters, even if an optimal threshold could be found offline, it may not remain valid as the incoming data evolves. To address these issues, we autonomously select classification thresholds.

In our density and Bayesian clustering approaches, log-likelihood values are produced to describe the possibility that a given observation could be generated by the underlying distributions. Since the likelihood of GMMs is the average of the log-likelihood observed in "normal" training data, it can be used as a baseline reference for determining a threshold for normal observations. Observations with likelihood values that are equal to or greater than the GMMs' likelihood will not be tagged as anomalous. However, further investigation is required for observations whose likelihood is less than the GMMs' likelihood. We use the following formula to compute fitness scores, f_{score}, that will have negative values only for suspect observations:

$$f_{score} = \frac{obs_{llk} - model_{llk}}{|model_{llk}|}, \tag{5.2}$$

where obs_{llk} is the log-likelihood that an observation and $model_{llk}$ is the log-likelihood of the model.

We consider the distance of an observation from its assigned cluster centroid to adjust the values of f_{score} such that the observation's likelihood increases as its distance to the cluster centroid decreases, and vice versa. The fitness scores are adjusted by multiplying them by a distance factor X_{dist}, which is derived from a normalized Euclidean distance and always has a positive value:

$$X_{dist} = \sqrt{\sum_{i=1}^{d} \frac{(x_i - \mu_i)^2}{\sigma_i}}, \tag{5.3}$$

where d is the dimensionality of the observation and x_i is the value of feature i of observation X, μ_i is the mean of the Gaussian distribution that models the feature density in space i, and σ_i is the standard deviation of feature i.

The value of X_{dist} will decrease as long as the distance from observation to its cluster decreases, and vice versa. The final resulting value obtained from $f_{score} \times X_{dist}$ is used to classify an observation. Empirically, we found that comparing this value with -1.0 gives accurate classification results, even across different application domains. Therefore, an observation is tagged as anomalous if the computed result is less than -1.0 and as normal if it is greater than -1.0.

In our distance-based clustering approach, we rely on the Mahalanobis distance, which gives us the scaled distance in terms of standard deviations. Based on Chebyshev's rule, 93.75% of the data of any distribution lies inside the range of four standard deviations of the mean. Thus, we compare the distance value with four to classify the observation.

Finally, the ensemble-based approach with isolation forests does not require parameterization. However, iForests produce anomaly scores in the range [0, 1] based on path lengths, where a score of 0 represents normal data with high confidence, and a score of 1 represents certain anomalies. Path lengths are determined by the number of edges that are traversed from the isolation tree root node to the terminal edge node. While determining a threshold for what should be considered anomalous data is problem-specific, observations with an anomaly score much less than 0.5 are presumably normal. Additionally, if all observations return a score of 0.5, there is a high likelihood

that no anomalies exist in the data. Given these factors, a threshold of 0.6 is generally acceptable for most problem types. We allow this default value to be changed by users, and alternatively they can compute an appropriate score based on the dataset contamination (estimated number of probable outliers).

The Adaptive Algorithm

Since we are dealing with continuous data streams, our goal is to allow adaptive behavior without affecting performance. This is ensured for our distance, density, and Bayesian detectors by incrementally updating the clusters' parameters, μ and Σ, without the need to have all of the observations resident in memory. How often the cluster's parameters should be updated depends on a configurable adaptation speed that can be changed at run time.

Incremental updates of the current mean μ_{t-1} are performed by retrieving the previous sum and using it with the new observation to compute the new mean μ_t. Equations (5.4) and (5.5) update the means and sizes of a cluster to which the observation x_i was assigned:

$$n_t = n_{t-1} + 1. \tag{5.4}$$

$$\mu_t = \frac{\mu_{t-1} * n_{t-1} + x_i}{n_t}. \tag{5.5}$$

In both of our approaches, each element σ^2 in the covariance matrix is i, j incrementally updated by following the same concept that was used for updating the mean. Equation (5.6) computes the new σ_t^2 of the features x_i and x_j (We use Bessel's correction to estimate variance.):

$$\sigma_t^2 = \frac{(n_t - 2)\sigma_{t-1}^2 + (n_t - 1)(\mu_{i,t-1} - \mu_{i,t})(\mu_{j,t-1} - \mu_{j,t}) + (x_i - \mu_{i,t})(x_j - \mu_{j,t})}{n - 1} \tag{5.6}$$

By updating these values, we ensure that our models will adapt to evolutions in the dataset over time. This is particularly important in a variety of fields, such as atmospheric science, environmental modeling, and epidemiology.

Unlike our other anomaly detection methods, isolation forests do not require additional logic to handle adaptation as the observations evolve. Path lengths through the isolation trees will vary over time, responding to changes in the incoming data points. As a result, the isolation score that is produced by the algorithm for the same input data may change in response to transformations in the underlying data stream. Conversely, data streams that do not evolve over time will produce similar sets of isolation trees and path lengths.

5.3.3 Evaluation of the Anomaly Detection Metric

To evaluate our reference anomaly detector, we used five classification datasets, sourced from the UCI Machine Learning Repository [109]. The datasets were selected from different application domains and have varying amounts of observations and features. The datasets that we used are shown in Table 5.1.

Table 5.1: UCI datasets used to evaluate the efficiency of our anomaly detection approaches.

Dataset	Observations	Features
Cardiac Arrhythmia	452	279
Breast Cancer Wisconsin	569	32
Cardiotocography	2,126	23
Seeds	210	7
Statlog (Shuttle)	58,000	9

To establish ground truth for testing purposes, we considered data from the most abundant class as nonanomalous and used it for training, with the rest of the classes used for testing. The K-means and EM-based detectors were separately applied to each dataset. Binary classification results represented by sensitivity and specificity are listed in Table 5.2. As illustrated by the results, both detectors perform well in most cases. The lowest-performing dataset, Arrhythmia, contained a large number of attributes but very few observations, which may explain its performance.

Table 5.2: Binary classification evaluation for UCI datasets.

	EM		K-means	
Dataset	**Sensitivity**	**Specificity**	**Sensitivity**	**Specificity**
Cardiac Arrhythmia	0.978	0.33	0.983	0.162
Breast Cancer Wisconsin	0.740	0.972	0.917	0.642
Cardiotocography	0.820	0.466	0.860	0.509
Seeds	0.919	0.951	0.961	0.793
Statlog (Shuttle)	0.974	0.976	0.983	0.826

5.4 Evaluation of the Anomaly Detection Framework

We designed a series of experiments to evaluate the performance of each part of our distributed anomaly detection framework. The experiments answered the following questions:

1. Does the framework scale out as more resources are added, and is data partitioned efficiently among the resources?

2. Can our framework operate in real-time while processing continuous data streams? While each anomaly detector implementation will have its own resource requirements, the framework itself should be lightweight.

3. Can anomalies be detected based on geospatial locations? Detector instances may be applicable to some regions but not others.

4. Given our adaptation interfaces, is the framework able to accurately adapt its model as the dataset evolves over time?

5. Can the framework detect anomalous observations whose features have normal values, but an unusual combination of some features' values?

5.4.1 Building the Models: Test Dataset

This study used real-world climate data that were obtained from the National Oceanic and Atmospheric Administration (NOAA) North American Mesoscale Forecast System [108]. The

readings in this dataset are collected regularly from various weather and climate stations and stored in the self-describing NetCDF format [153]. Each file contains spatiotemporal information as well as several climate feature readings that include surface pressure, surface temperature, snow cover, snow depth, relative humidity, and wind speed. The particular data that were used in this study were collected over an 11-year period from 2004 to 2014. Each year comprised roughly 1,000,000,000 observations on average, which were stored alongside compressed geographical map tiles. The entire dataset spanned over 1PB of raw data and 20 billion files.

Observations from the first 3 years (2004 through 2006) were used as training data to build the models that were used in this section. Observations from the remaining years were used for verification and classification.

5.4.2 Experimental Setup

Our framework was run on a 78-node cluster with 48DL160 servers (Xeon E5620 CPU, 12 GB of RAM) and 30HP DL320 servers (Xeon E3-1220 V2 CPU, 8GB of RAM). Each server was equipped with four hard disks. Ten clients running on machines outside of the cluster were used to read the observations from the National Oceanic and Atmospheric Administration dataset and send them to Galileo, which operated under the OpenJDK Java Runtime, version 1.7.0_65.

Each storage node maintains an anomaly detection framework instance that receives incoming observations and partitions them spatially by using the Geohash-based scheme that was described earlier. Each storage node is responsible for a number of regions that are subdivided by the coordinator based on a four-character prefix to assign a 39.1×19.5 km region to each anomaly detector instance. Figure 5.6 shows the distribution of anomaly detection instances across the cluster (note the relatively uniform distribution of load). In these benchmarks, the total number of anomaly detector instances was 60,922, and the training data for each model consisted of 49,241 observations on average.

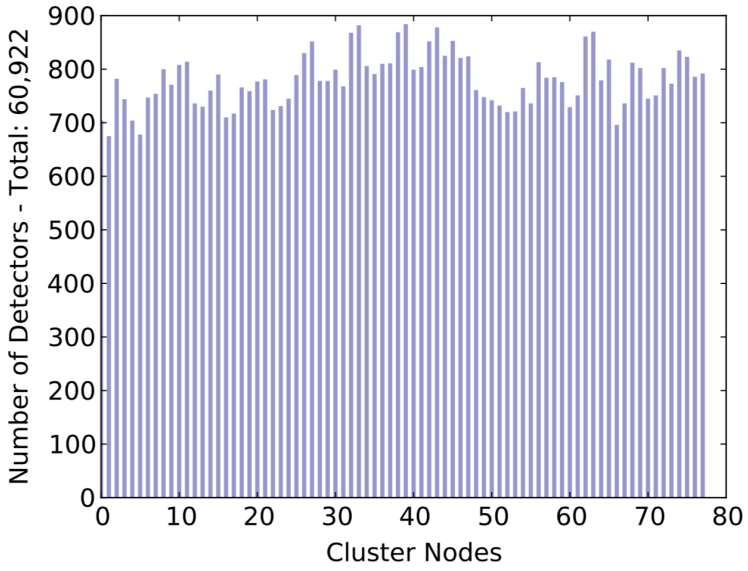


Figure 5.6: Anomaly detector instances at each storage node. Note the relatively uniform distribution of detectors due to our Geohash-based spatial partitioning scheme.

5.4.3 Anomaly Detection Throughput

Our partitioning scheme and network layout help facilitate real-time anomaly detection at scale. To evaluate the scalability of our framework, we conducted a throughput test across all 78 machines in our test cluster, with varying anomaly detection adaptation speeds. With adaptation disabled, the 78-node cluster could process about 1 million observations per second in parallel. Figure 5.7 illustrates the number of classified observations per second using both the EM and K-means detectors at different adaptation speeds. Even when the model parameters were recomputed for every other new record, our framework was able to achieve an overall throughput of about 500,000 classifications per second. This efficiency could be leveraged in cloud-based settings with low-power or low-cost resources or used to process live data streams.

5.4.4 Influence of Geospatial Scope on Model Accuracy

In this experiment, we demonstrated the effect of the geographic regions being managed by anomaly detector instances. We trained two models for each approach, one built with observations that were taken from Florida in the United States, and another built from observations belonging

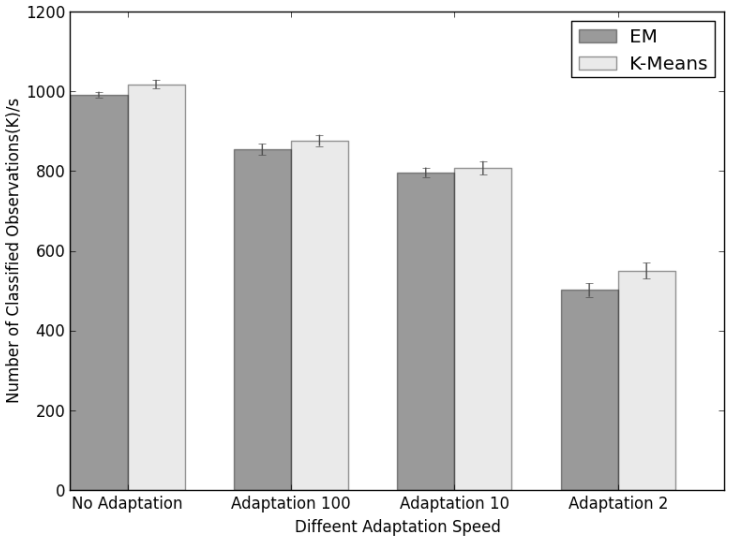


Figure 5.7: Observations classified per second for a variety of adaptation speeds; an adaptation rate of 100 means that the model parameters are updated each time 100 new observations have been received.

to Hudson Bay in Canada. The Figure 5.8 shows the geographic differences between the two locations.

Random observations were sampled from both regions and used to test the models. If the models have specialized in their particular regions correctly, then being fed with observations from the other region should produce classifications that are definitively anomalous. On the other hand, observations that were taken from the model's own spatial region should be considered normal. Table 5.3 outlines the results of this experiment; observations from a different region were correctly tagged as anomalous, while those from the model's own region were considered normal. This shows that each model instance captured fine-grained details within its own spatial region and reinforces our decision to use multiple detection instances.

5.4.5 Evaluating Model Adaptation

In this experiment, we tested how the framework copes with changes occurring in observed behavior when the adaptation feature is turned on. We drew a random observation from the test data and created 300,000 copies of it. In each copy, the temperature was set to a value between 330K and 360K that was generated randomly, which would constitute anomalous temperatures.

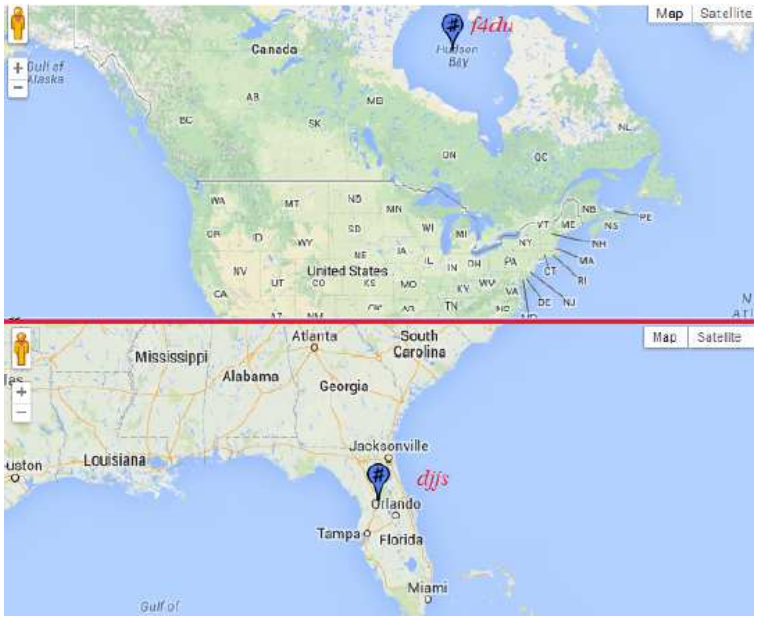


Figure 5.8: Two different geographic regions with correspondingly different climates: Hudson Bay, Canada, and Florida, USA.

Table 5.3: Classification results obtained by applying EM and k-means detectors to observations taken in January 2013 for both Hudson Bay (Geohash: f4du) and Florida (Geohash: djjs). To demonstrate that the models have specialized for their particular geographic region, we fed in observations from the other region, which were correctly labeled as anomalous. An observation was tagged as anomalous if the score was less than -1 in EM and greater than 4 in K-means.

	EM		K-means	
Sample Location	**Florida**	**Hudson Bay**	**Florida**	**Hudson Bay**
Hudson Bay	-1.1E22	-0.5	419.04	2.3
Hudson Bay	-1.1E22	-0.5	343.9	2.16
Hudson Bay	-1.1E22	-0.4	329.2	1.13
Florida	0.017	-538577	1.03	38.4
Florida	-0.089	-708725	1.27	46
Florida	0.007	-538253	0.34	61

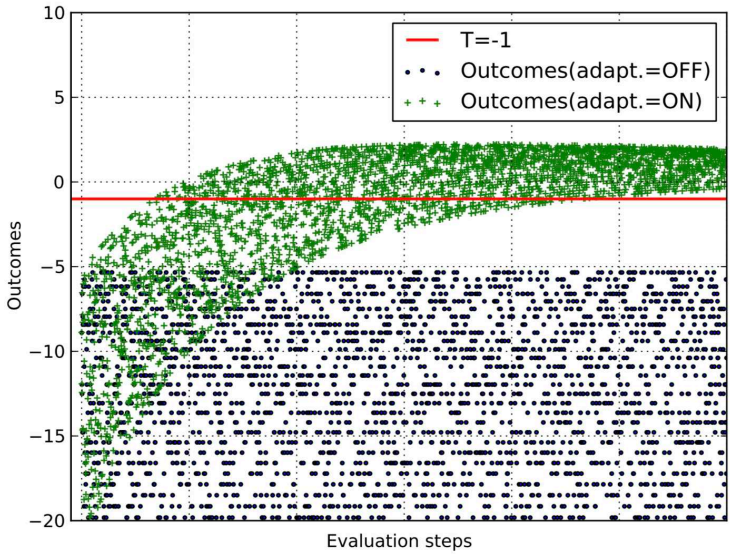


Figure 5.9: Outcomes obtained by running the EM-based approach twice, with and without the adaptation feature.

Adaptation speed was set to 100, which means that adaptation was performed once for every 100 classification operations. Because both EM and K-means use the same adaptation algorithm, we report results for only the EM-based approach, which was used to classify the 300,000 copies of the observation, followed by another 300,000 classifications with adaptation turned off.

The outcomes of two cases can be seen in Figure 5.9. In the figure, the x-axis represents the evaluation steps, and the y-axis represents the outcomes. When the adaptation feature was disabled, the observation outcomes shown in blue were not changed over the 300,000 evaluation steps and were always considered anomalous based on our threshold of -1.0. However, when the adaptation feature was enabled, the same observations were evaluated as anomalous early on in the test but began being considered normal over time. Consequently, we can conclude that our adaptation feature allows continuously anomalous observations to eventually become normal over time, whereas the nonadaptive model would not be able to handle such changes.

5.4.6 Comparison of Anomaly Detection Methods

After the anomaly detectors had been built on observations from the training data using each of our approaches, the detectors were used to classify observations from the remaining 9 years

Table 5.4: Number of anomalies detected by each of our classification methods.

Region	EM	K-Means	DPMM	iForest
9pkv	49,207	48,974	35,737	26,870
f2y8	329	308	1,153	276
dk6m	22,473	21,799	12,446	29,133
dk63	9,776	3,882	2,517	3,156
c3w6	1,014	982	1,897	1,071
c1f2	11,874	1,571	333	2,888
9wwh	430	429	2,160	929
9s0j	25,895	25,616	27,752	37,307

Abbreviations: EM, Expectation Maximization; DPMM, Dirichlet Process Mixture Models.

(2006 through 2014). Table 5.4 lists the number of anomalies that was detected by each approach across eight different spatial locations, underscoring the influence of spatial regions on our climate dataset. Table 5.5 lists the number of common anomalies that was detected by both approaches (set intersection). This experiment illustrates that there are indeed corner cases in the dataset that can be detected by some approaches but not others. Isolation forests, for instance, have been shown to be robust to the effects of swamping and masking that cause the number of outliers to be overestimated or underestimated [150]. In general, it is essential to be able to evaluate several approaches because each dataset contains unique properties that may be best suited for a particular approach.

Table 5.5: Intersection of anomalies detected by each of our classification methods.

Region	EM ∩ KM	KM ∩ DPMM	EM ∩ iForest	DPMM ∩ iForest
9pkv	48,974	34,580	9,675	10,246
f2y8	308	63	40	53
dk6m	21,799	11,079	4,640	4,659
dk63	3,882	1,472	321	436
c3w6	982	531	79	220
c1f2	1,571	271	289	198
9wwh	429	295	84	174
9s0j	25,616	20,644	6,196	9,735

Abbreviations: EM, Expectation Maximization; DPMM, Dirichlet Process Mixture Models; KM, K-means.

5.5 Discussion

This chapter presented our approach, encompassing algorithms and system design, for scalable detection of anomalies in multidimensional, geospatial data streams. To deal with observations that are spatiotemporally correlated, each of our models is responsible for a particular geographical extent and includes timestamps that are associated with incoming data points. Each model instance tunes itself independently based on the data that it observes. Rather than employing a single model that attempts to preserve such correlations, using individual instances that span smaller geographical extents improves accuracy. This approach is well-suited for situations where there is variability in the data streams for particular geographical extents, allowing the regions to be adaptively refined with corresponding additions of model instances. Ultimately, our approach allows fine-tuning of the specificity of the classifications by controlling the geographical scope that is associated with the classification models.

Associating model instances with particular geographical extents results in a scalable design; that is, we can scale with increases in data volumes and the number of machines that is available. This approach is also amenable to dispersion, with models executing on multiple machines and performing concurrent, distributed classifications as observations arrive. Multiple model instances are also maintained at each node by using a thread pool, which allows for concurrent classifications of data streams within a particular geospatial area. Our empirical results validate the scalability and throughput of our approach at both the individual nodes and in a distributed cluster, which is capable of reaching an overall throughput of over 1,000,000 classifications per second. The specificity of our models—controlled by the granularity of the Geohash—allows us to account for spatiotemporal correlations that are associated with the observations and achieve greater accuracy.

Our anomaly detector interfaces are agnostic to their underlying algorithms and implementations. The interfaces automate three key phases in the detection of anomalies: collection, training, and classification. The training phase buffers and prepares the data for training, while the training phase creates and trains the models on the collected data. Finally, the classification phase uses the trained models to classify observations as either normal or anomalous. Our reference imple-

mentation incorporates support for distance (Euclidean and Mahalanobis), density, Bayesian, and ensemble-based clustering algorithms. The feasibility of this approach was verified with well-known datasets.

Given the associated data volumes, dimensionality, and the rates at which observations arrive, it is infeasible to employ human intervention in the anomaly classification process. Our approach does not require human intervention for the adjustment of anomaly detection thresholds and provides degree of irregularity scores to help users direct their efforts. Our classifiers continually and autonomously adapt themselves based on the data that are observed by models; that is, both the adaptation and the classification are performed concurrently. Rather than requiring explicit coordination amongst detector instances, we allow a consensus to be reached through distributed anomaly queries. Our empirical benchmarks demonstrate both the efficiency (per-packet classifications) and accuracy of these adaptations.

Chapter 6

Assessing the Epidemiological Impact of Livestock Disease Outbreaks at the National Scale

Our goal was to build analytical models that enable assessing the epidemiological impact and economic costs of livestock disease outbreaks at the national scale while reconciling the heterogeneity of the U.S. mainland. These models are used by the what-if tool that allows planners to experiment with the outcome of their planning activities. The tool uses the analytical models to give real-time responses to the planners' explorations.

Our effort focused on taking scenarios covering geospatial regions, generating scenario variants, orchestrating each variant for several iterations that are statistically valid, and learning from the outputs—both intermediate and final—that were generated by executing these variants. Figure 6.1 illustrates the main tasks that are involved in building models for the what-if tool. Instead of building a single analytical model, we built 1,161 analytical models to account for heterogeneity across the continental U.S. Here, heterogeneity refers to differences in the factors that contribute to disease spread, such as the density and distribution of farms, the probabilities associated with the types and frequency of contacts, surveillance, and biosecurity. The efficient management of the voluminous data that is generated during variants' generation and execution strengthens the effective generation of the analytical models.

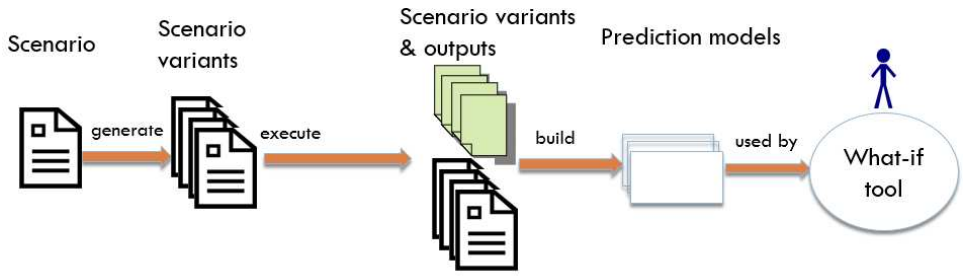


Figure 6.1: The workflow of models construction for the what-if tool.

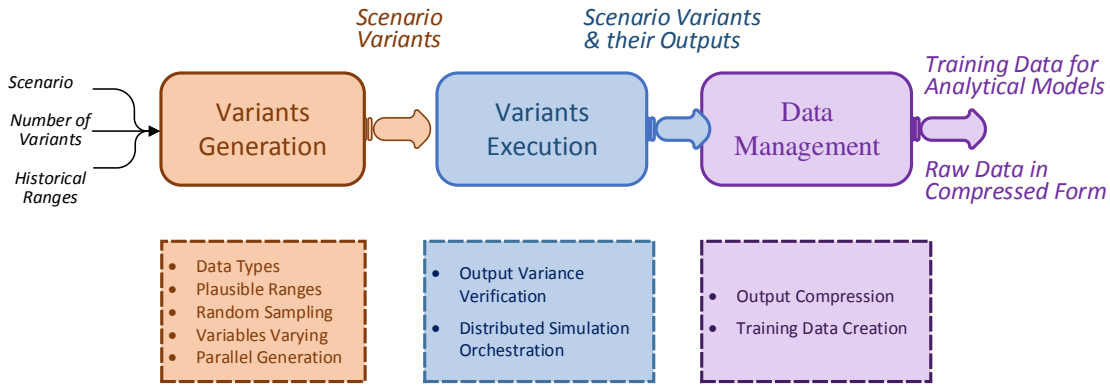


Figure 6.2: The phases and tasks that are involved for the creation of the epidemiology datasets.

In the next sections, the creation of the epidemiology datasets is described in detail in §6.1, investigation of the most proper learning algorithm and encoding of the trained models are introduced in §6.2, and the training data creation and models construction for disease outbreaks at the national scale are illustrated in §6.3.

6.1 The Creation of Epidemiology Datasets

To understand the relationship between a scenario (input parameters) and its outcome (output parameters), we need to generate a massive number of variants and execute them to produce their outputs. The generated data (variants and their outputs) is prepossessed and then used to fit analytical models that are used for making quick predictions. The process for data generation involves performing three stages on the variant basis, as shown in the Figure 6.2:

For a given scenario, we generate variants, execute them, and process the generated data. The stages must be executed sequentially for each variant. The following subsections describe these stages and the involved tasks in more detail.

6.1.1 Variants Generation

A given scenario uses input parameters to describe disease properties and outbreak characteristics. Generating a scenario variant involves randomly varying the values of the input parameters in their plausible ranges. For producing plausible ranges for the input parameters, our framework

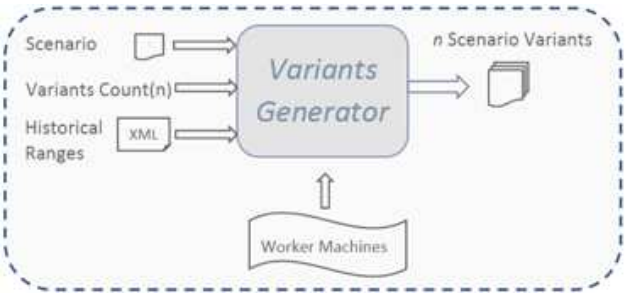


Figure 6.3: The variants generator framework.

consults historical data that are available in previous scenarios as a preliminary step. We developed a variants generator framework that performs all of the needed tasks for generating variants. The framework orchestrates the process of variants generation on multiple machines in parallel to enable generating a large number of variants in a relatively short time. Figure 6.3 shows the inputs and the produced outputs of the variants generator framework.

For generating the variants, the variants generator receives the given scenario, along with the number of the variants that need to be generated, the historical ranges, and a list of machines where the workload of variants generation is to be distributed. The workflow of variants generation (shown in Figure 6.4) is as follows:

1. The input parameters are extracted from the given scenario. The extracted input parameters can have simple data types (such as numerical data types) or complex data types (such as probability density functions [PDFs] and charts).

2. Variables having simple data types are created to describe the complex input parameters. Each extracted PDF and chart is described by a number of numerical variables. This task generates numerical variables (The numerical input parameters and the ones that describe the input parameters of complex data types).

3. Historical ranges are used to create a plausible range (bounding box) for each numerical variable. The historical ranges are shrunk to become closer to and around the variables' values being varied. The bounding boxes will enable generating scenario variants in the neighborhood of the input space of the original scenario.

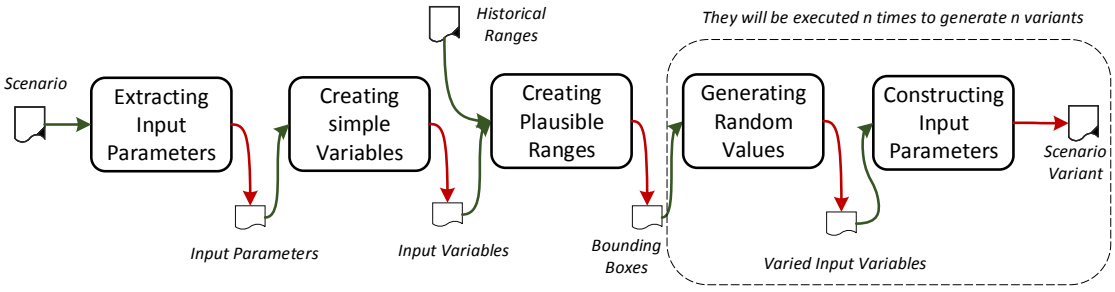


Figure 6.4: The workflow of variants generation.

4. The random values are generated within the bounding boxes for all of the numerical variables.

5. The input parameters with complex data types are reconstructed by using the varied values of their numerical variables.

Steps 4 and 5 are repeated for each new scenario variant. Multiple machines are used to generate the scenario variants in parallel.

Data Types

While most input parameters represent a numerical value or discrete state, simulations also frequently employ two-dimensional line charts or probability density functions to describe complex behavior. For instance, Figure 6.5 below contains several probability density functions that were used previously for the "cattle latent period" input parameter (amount of time between infection and the onset of infectiousness, in days).

The complex data types play a vital role in simulation outcomes and must be varied to enable the exploration of a scenario's parameter space. There are two main challenges here. First, how can we find plausible ranges for the individual input parameters of different data types? Second, how can we vary the input parameters of complex types. In addition to the simple data types, we need to support varying the data types that are listed in Table 6.1.

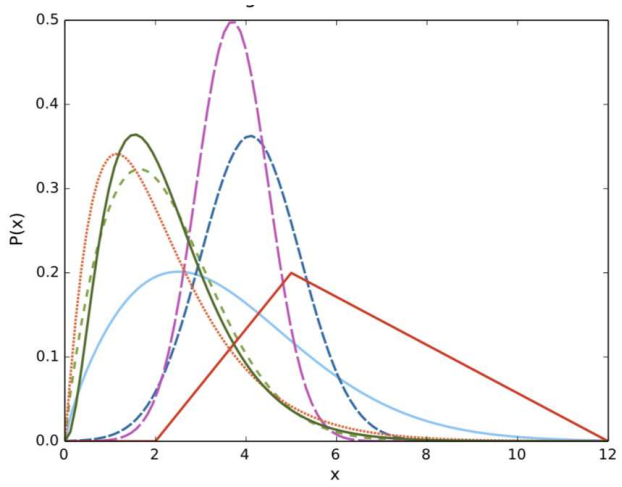


Figure 6.5: A variety of PDFs that were used to describe the "cattle latent period" parameter in previous scenarios. Historical data provided by modelers and epidemiologists is used to determine the upper and lower bounds of valid parameter ranges.

Table 6.1: The probability distribution functions that the variants generator supports as data types.

Bernoulli	Beta	Beta Pert	Bimodal	Binomial	Chart
Chi	Exponential	Gamma	Gaussian	Histogram	Inverse Gaussian
Logistic	Log Logistic	Log Normal	Negative Binomial	Pareto	Pearson3
Pearson5	Piecewise	Poisson	Triangular	Uniform	Weibull

Plausible Ranges

While some input variables have predefined ranges of valid values (such as a percentage ranging from 0% to 100%), others are completely unconstrained. In both cases, a value may be valid but not plausible, that is, extremely unlikely to occur due to environmental conditions or other factors. To produce plausible ranges for the input variables, we rely on the historical ranges that are available in previous scenarios as a preliminary step. Next, the ranges that are determined by this process are inspected and refined by subject-matter experts if necessary. This helps reduce or potentially avoid user intervention while maintaining accuracy.

We use the historical ranges to create a plausible range (bounding box) for each variable that is being varied. Our framework is given the historical ranges in an XML-format file that is used to put a range constraint on each value that is being varied (Figure 6.6 shows examples of historical

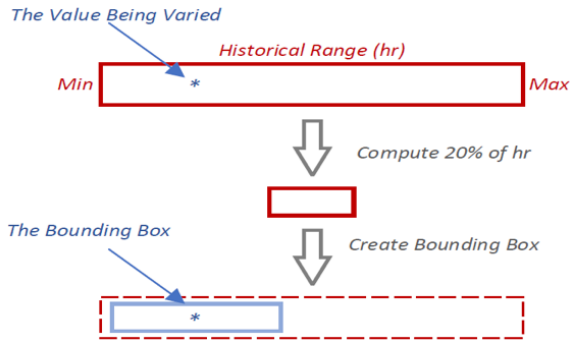
```
.....
<param name="direct distance" from-production-type="Cattle" .....
  <bounds>0,2110.27</bounds>
  <mean>41.5283,166.74</mean>
  <variance>846.675,560464</variance>
  <skewness>0.508,103.98</skewness>
</param>

<param name="indirect movement" production-type="Sheep/Goats" ....
  <xBounds>0,100</xBounds>
  <yBounds>0.03,1</yBounds>
</param>

<param name="max spread" from-production-type="Feedlot"  to-pr.....
  <bounds>3,6</bounds>
</param>
.....
```

Figure 6.6: Examples of historical ranges that can be found in the XML-based file.

ranges). The file includes the historical ranges for the input parameters, and the variants generator uses the historical ranges to create new ranges for all of the values that are being varied.

The figure shows examples of the historical ranges that are defined in the historical range file. The first element defines the historical ranges for the variables that describe the input parameter named direct distance of distribution type (i.e., the PDF). The second element defines the historical ranges for the variables describing an input parameter named *indirect movement* of chart type. The third element defines the historical ranges for the input parameter, *max spread*, which is the numeric type. For example, the variants generator uses the ranges [3,6] defined in the third element to vary all of the max spread of infections between feedlots. More information could be included in each historical-range element to distinguish between the same input parameters that are defined for different production-types.

The variants generator uses historical ranges with a given scenario to create plausible ranges that enable generating variants in the neighborhood of the input space of the source scenario. Accomplishing this task requires that the framework generates random values around the actual values existing in the source scenario. For this purpose, the variants generator uses the historical ranges to create a bounding box around each value in the source scenario. Each input parameter of



(a) The creation of a bounding box for a given value.

Example: Creating bounding box for the value 3 using the historical range [1.98418, 4.1]

len0.20= (4.1 - 1.98418) * 0.20 = 0.423

Bound box: [3 - len0.20, 3 + len0.20]

Bound box: [3 - 0.423, 3 + 0.423] = **[2.577, 3.423]**

(b) Example of the creation of the bounding box for the value 3

Figure 6.7: The creation of the bounding box.

a complex data type is dismantled into multiple simple variables. The generator uses the historical ranges to create bounding boxes for each of its simple variables. The bounding boxes are created to control the length of the intervals that are used to generate random values around the actual values that are extracted from the source scenario. Figure 6.7 shows the steps that are needed to create bounding boxes and an example of building a bounding box for the value 3.

Random Sampling

After establishing plausible ranges for all of the numerical variables, one might elect to generate new scenario variants with random samples across the parameter space. However, simple random sampling assigns an equal probability to each possible input without considering the plausibility of the values. For circumventing this limitation, weighted sampling draws values from a probability distribution. This preserves the plausibility curve of potential inputs, but it also increases the likelihood of choosing highly probable values; if a small number of samples are drawn from a probability distribution, a correspondingly small portion of the input space will be represented. This means that we would have to generate (and execute) a large number of scenario variants to explore the parameter space with weighted sampling adequately.

Unlike simple random sampling or weighted sampling methods, Latin hypercube sampling (LHS) [154] stratifies the input probability distributions to better represent their underlying variability. This reduces the number of samples that is required for our algorithms to adequately

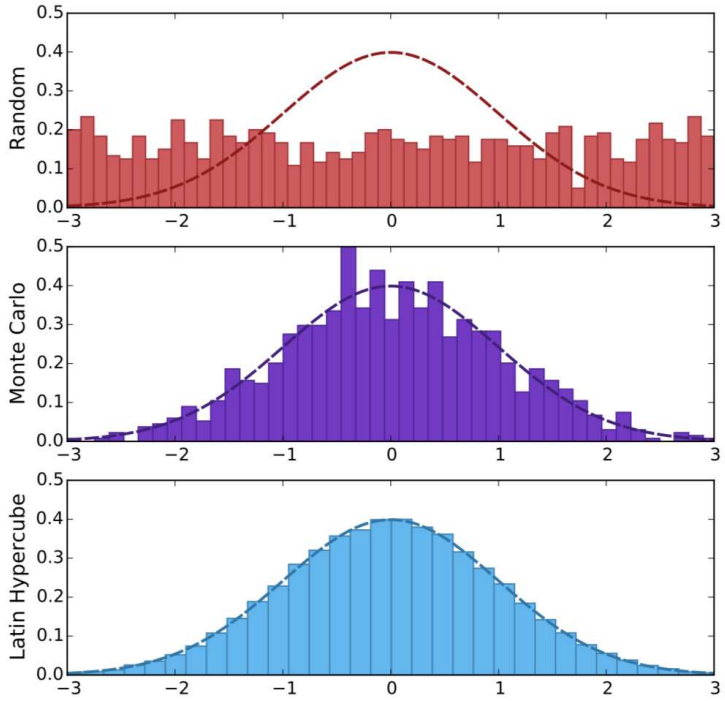


Figure 6.8: Sampling performed over a normal distribution using random, Monte Carlo, and Latin Hypercube sampling. In each case 1,000 samples were taken and are represented by 50 bins.

explore the scenario parameter space, which in turn decreases the overall computational footprint of the framework. Latin hypercube sampling owes its name to Latin squares, which are $N \times N$ arrays that contain N different elements. Each element in a Latin square occurs exactly once in each row and column. When this concept is applied in a multidimensional setting, the elements occur once in each hyperplane, forming a Latin hypercube. This allows us to produce samples across all of the variables in the parameter space in a single sampling step. Based on the number of samples required, each stratum represented by the elements in the hypercube is divided into equal intervals. These produce samples in the range [0, 1], which are converted back to the original units by using the corresponding bounding boxes.

Figure 6.8 provides a visual comparison of unweighted, weighted, and Latin hypercube sampling. The sampling was performed over a normal distribution. In each case, 1,000 samples were taken and are represented by 50 bins. Latin hypercube sampling provided the best overall representation of the underlying distribution (a standard normal distribution) when the number of samples was held constant across methods.

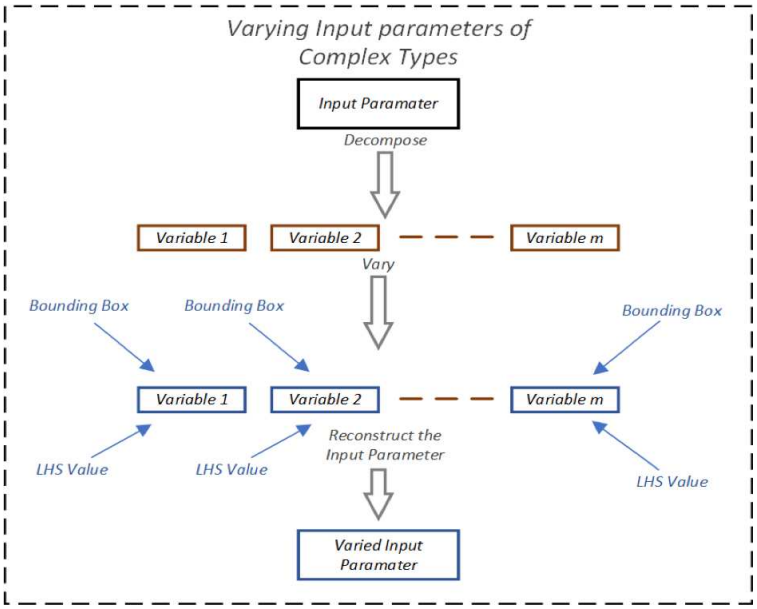


Figure 6.9: Varying input parameters that are associated with complex types.

Variables Varying

To generate a scenario variant, we need to use LHS for generating n random values, where n is the number of the numerical variables (not the number of input parameters). Each numerical input parameter is seen as a single numerical variable. The other input parameters that have complex data types are decomposed into a different number of variables. Therefore, the number of numerical variables is always higher than the number of input parameters. We need to generate n LHS values to vary n variable values. Figure 6.9 shows how we vary the input parameters that are associated with complex data types.

To vary the numerical variables, we need the random values that are generated by LHS and the corresponding bounding boxes. The LHS generates random values between 0 and 1 that need to be scaled to the bounding boxes. We use the following formula to generate a random value within a bounding box:

$$\mathbf{randomValue} = \mathbf{b_0} + (\mathbf{b_1} - \mathbf{b_0}) \times \mathbf{lhsValue}, \tag{6.1}$$

where b_0, b_1 are the lower and upper bounds of the bounding box, respectively.

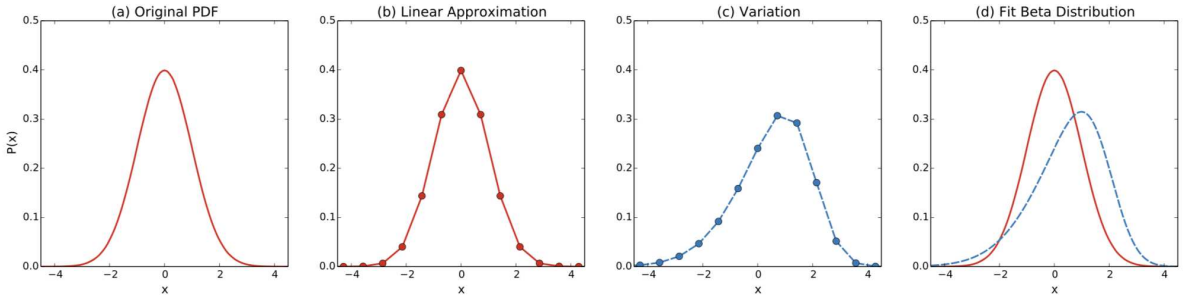


Figure 6.10: A visual overview of our PDF variation algorithm. The original PDF (a) is converted to a piecewise linear approximation (b). Next, key attributes (upper and lower bounds, mean, variance, and skewness) are modified to create a linear approximation of a new PDF (c). Finally, a beta distribution is fit to the modified linear approximation to create a new PDF variant (d).

Varying the PDF Parameters

Probability density functions can often be decomposed into a few different variables (such as mean, variance, and skewness) that can all be manipulated to create a new PDF variation. Unfortunately, each type of distribution has its own formulas for modifying these attributes, and the fact that our simulation supports 24 different types of distributions only further complicates matters. Instead of dealing with this issue as 24 separate problems (or possibly more for other simulations), we generalize the PDFs by transforming them into piecewise linear approximations.

Once this step has been completed, we inspect the resulting upper and lower bounds, mean, variance, and skewness. These attributes are modified to create a new linear approximation of a curve. Next, a beta distribution is mathematically fit to the curve. Beta distributions are described by two shape parameters, α and β, which can be adjusted to model a wide range of distributions. An overview of our PDF generation algorithm is visually described in Figure 6.10.

The fitting algorithm relaxes the bounds and tries to find a beta distribution that has the same values as do the varied mean, variance, and skewness. We validated our fitting algorithm by applying it to different PDFs. In the validation test, we find a beta distribution for a given PDF (without variation) and compare the shapes of both distributions. For a given PDF, we use the characteristics of its corresponding piecewise approximation to find a beta distribution with similar characteristics. Figure 6.11 shows different PDFs and their fitted beta distributions.

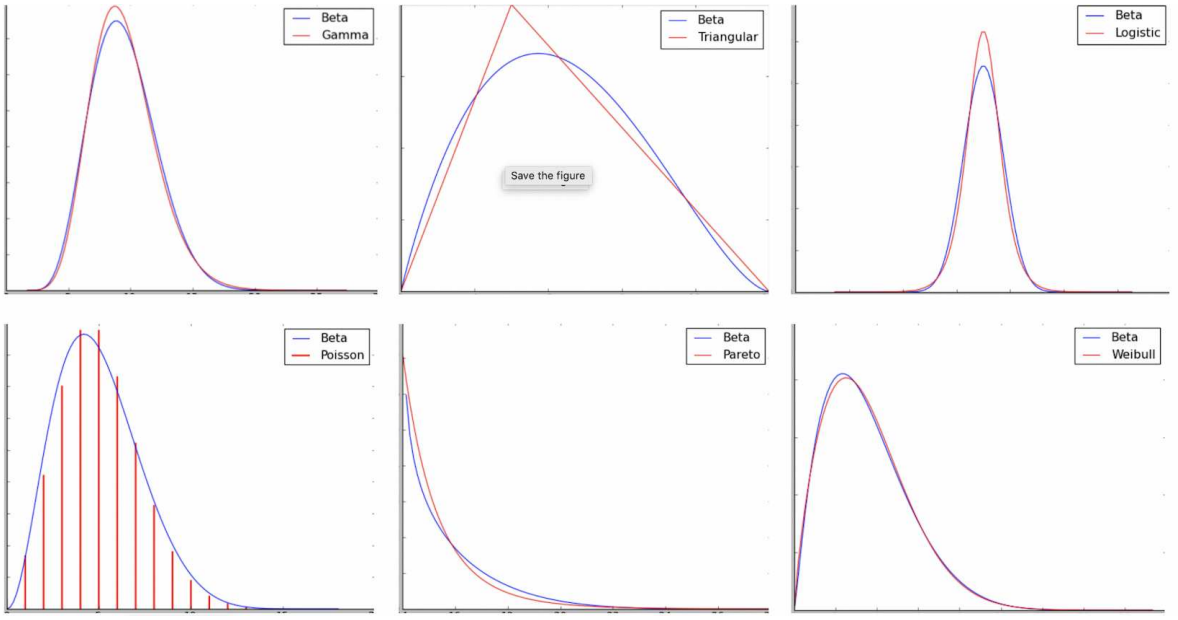


Figure 6.11: Different PDFs and their fitted beta distributions.

Varying Relational Chart Parameters

The modeler can choose x-y charts for some input parameters. We use the minimum and maximum values of x and y of the chart to find the smallest rectangle that covers the modeler's chart. The variants generator uses the bounding boxes and the LHS values to vary the four values (minimum and maximum values of x and y). The varied values will form another rectangle. Finally, the generator scales the x-y values in the first rectangle based on the second rectangle. Figure 6.12 shows the concept of varying charts. Each of the following figures shows a scenario chart (in red) along with its variations (in gray).

6.1.2 Parallel Variants Generation

Our variants generator supports two types of simulations, NAADSM and ADSM. The NAADSM scenario is stored in two XML files, and an ADSM scenario is stored in one SQLite3-formatted file. Because we are attempting to generate millions of variants, it becomes necessary to use a large number of machines to generate the variants in parallel. The good thing here is that the generation of variants can be done independently and does not involve any communication. To speed up the variants-generation process, we created a distributed version of the variants generator. Our design

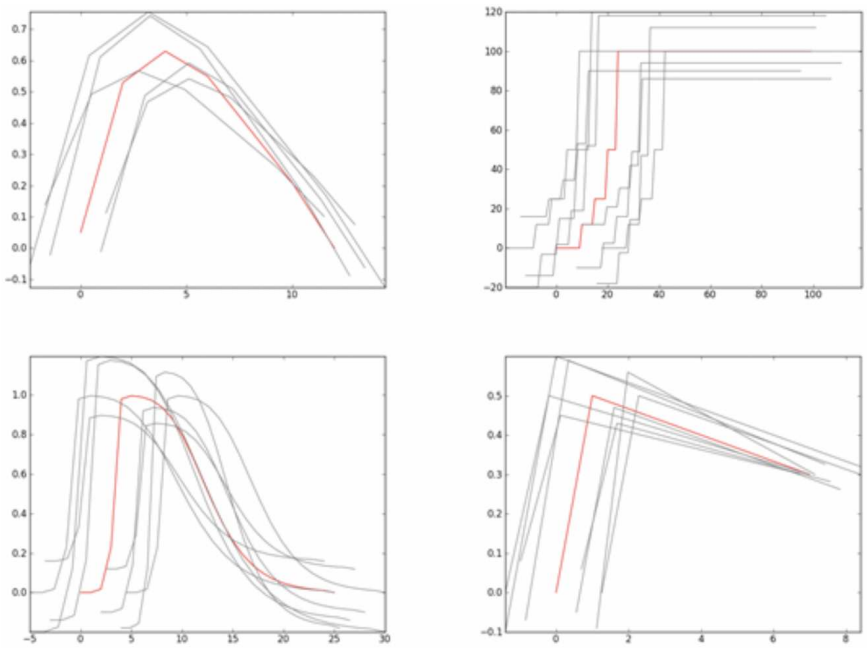


Figure 6.12: Different charts and their generated variants.

decision of the distributed version was made to generate variants on a large number of machines in an effective way. The distributed version includes two main parts, *coordinator* and *remote variants generator*.

Coordinator

The coordinator reads the input parameters from a given scenario (XML-based and SQLite3-based scenarios), converts them into simple variables, creates bounding boxes, and generates LHS values based on the created variables. Additionally, it divides the generated LHS values into chunks and assigns them to remote variants generators that are launched on several machines. The coordinator performs the following tasks:

1. Reading the input parameters from a given scenario: The coordinator has two readers to support reading the input parameters from two different scenario formats (XML- and SQLite3-based scenarios).

2. Creating simple variables: Once the input parameters have been extracted, they are converted into simple variables that need to be varied. The number of these variables represents the number of random values that are needed for generating one scenario variant.

3. Creating the bounding boxes: The coordinator uses the historical ranges and the variables' values to create a bounding box for each variable. The bounding boxes limit the ranges where the variables' values can vary.

4. Generating the LHS values: The coordinator uses LHS to generate random values that enable varying the source scenario in the neighborhood of its input space. Two pieces of information are needed to generate the LHS values, the number of random values needed for generating each variant (referred to as d) and the number of variants (referred to as n). The coordinator generates an $n \times d$ array with LHS values.

5. Dividing the LHS array into chunks: The LHS array is horizontally partitioned into several chunks, each stored in a separate file. Each chunk includes random values that enable generating a particular number of variants. The number of chunks depends on the number of remote generators that will be launched in parallel on several machines.

6. Launching remote generators on several machines: The coordinator creates instances of the remote generator and launches them on several worker machines. The coordinator delivers the original scenario, the extracted variables, and the assigned LHS chunk to each remote generator.

Remote Variants Generator

Based on the capabilities of the worker machines, the framework launches several remote generators on each machine. Each remote generator is assigned to one chunk file that includes many lines, and each line carries random values for generating one scenario variant.

The remote generator uses the assigned chunk to create variants and locally store them on-disk. Each remote generator performs the following tasks:

1. Loading the contents of the chunk file in a 2D array.

2. For each line of random values existing in the array, the remote generator performs the following steps:

   (a) scaling the random values to their corresponding bounding boxes to create new values for the numerical variables, (b) using the generated variables to create the input parameters, and (c) using the new input parameters to construct scenario variants. The remote generator uses to two writers; one is responsible for writing the input parameters to the NAADSM-based scenario (XML-based file), and the other is for inserting the input parameters into the ADSM-based scenario (SQLite3-based file).

6.1.3 Variants Execution

Once a scenario variant has been created, it must be executed several times to obtain an understanding of its output distribution and behavior.

Verifying Output Variance

For determining whether the overall variation of the output variables is significant, 32 pilot runs are executed for each variant. Doing so requires two pieces of information: (1) the outputs that are most meaningful from an analytical standpoint and (2) the minimal significant difference in output variances that must be achieved. Determining the minimal significant difference in output parameters requires the knowledge of a subject-matter expert. These values can be expressed as percentages, numeric ranges, or confidence intervals, and they tell the framework whether or not there is enough variation in the output variables. If the minimum variance is met, the execution of the scenario variant is complete. Otherwise, the observed variance is used to calculate how many more executions of the scenario must be carried out to achieve the required minimum variance.

Distributed Simulation Orchestration

We generate a large number of variants that need to be executed to produce their outputs. The execution of the scenario variants has to be done in parallel to speed up the creation process

of the epidemiology data. For this purpose, Forager [155], a distributed execution engine, has been employed for orchestrating the variants executions on multiple machines in parallel. Forager enables fast execution of the scenario variants, and each needs to be executed at least 30 times to account for uncertainty in the results. However, executing each variant produces a massive amount of raw output.

6.1.4 Data Management

Once the scenario variants and their outcomes become available, we make use of them for extracting knowledge and establishing an understanding of the relationship between the scenario variants (the input parameters) and the outputs (the output parameters). We extract the considered output parameters and concatenate them with their corresponding variants. Additionally, we compress the raw outputs to save a substantial amount of disk space and greatly increase the exploration capabilities on a given set of hardware [155].

Training Data Creation

For the understanding of how the disease evolves in the study region, we generate the variants with one infected farm. We choose a random farm and set its state as infected in the given scenario and generate variants with one infected farm. In such a case, the produced outputs describe and track disease spread in the entire study region covered by the scenario as a result of the initially infected farm and its production-type.

6.2 Building the Analytical Models

We need to train the analytical models to understand the relationship between the input and output parameters. Building the models is a one-time process that initializes our real-time forecasting engine. Building models is time expensive. However, once the models have been trained, applying them for making predictions is fast. We had to investigate various learning algorithms to find the one that provides the best prediction accuracy. We tested an artificial neural network, multivariate linear regression, deep learning, random forests, and gradient boosting [155, 156]. Multiple test

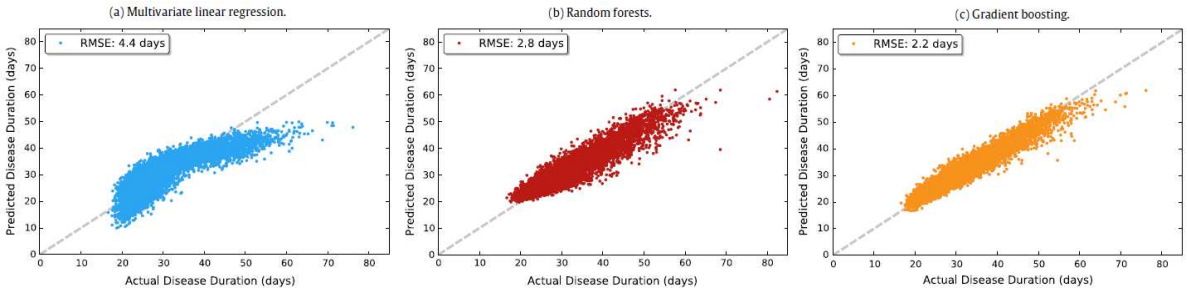


Figure 6.13: A comparison of predictions for the disease duration NAADSM output variable produced by multivariate linear regression, random forests, and gradient boosting on our 100,000-variant dataset.

scenarios were employed to evaluate the considered learning algorithms on different output parameters. Our evaluation results showed that the models that were trained by using gradient boosting outperformed all of the models that were trained by the other algorithms. Figure 6.13 shows the prediction accuracy of the linear regression, random forest, and gradient boosting models that were built to predict disease duration in the Texas scenario.

For enabling the analytical models to be used by the what-if tool that runs on internet browsers, the models must be loaded without the need for the libraries that were used for building the models. Each created prediction model is stored in a small JavaScript Object Notation (JSON) file to allow it to be retrieved and used by the what-if tool. JavaScript Object Notation is a lightweight data-interchange format, and it is supported by many different programming languages. The JSON-based model includes only the information that is necessary to enable the what-if tool to make predictions. The what-if tool should be able to load the analytical models and use them to predict the values of output parameters of unseen scenario variants created by the modelers.

6.3 National-Scale Disease Spread Model

We analyzed petabytes of epidemiology data on multiple machines to build analytical models that allow assessing the epidemiological impact and economic costs of livestock disease outbreaks at the national scale. This involved training 1,161 analytical models that were specially and temporally coupled to model the disease dynamics. We generated millions of scenario variants and executed the created variants to create training data where an observation contains the input pa-

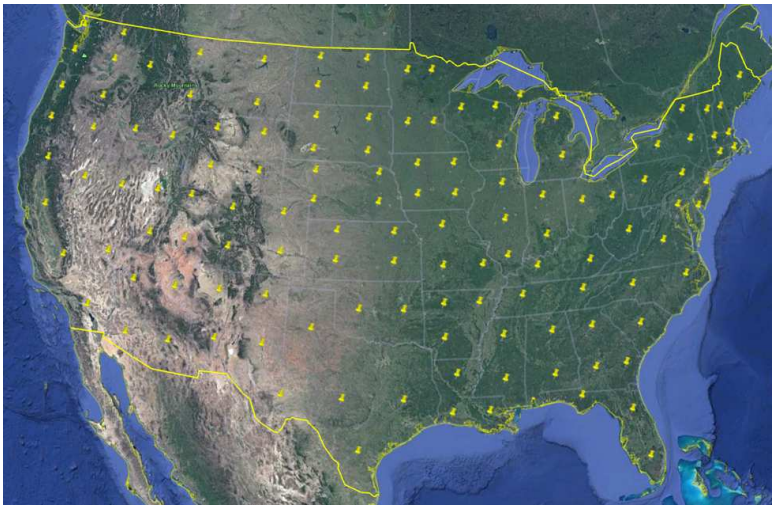


Figure 6.14: 129 sectors covering the contiguous United States.

rameters of one variant as input features and its outputs as targets. The models were trained to predict the intermediate and final outputs for given input parameters (i.e., a scenario variant). The models were built to capture phenomena that are associated with smaller geospatial areas. The trained models were used by a what-if tool to enable real-time visual analysis of disease-spread dynamics.

In this study, we worked on five regional scenarios that cover the central, northcentral, northeastern, southeastern, and western regions of the United States. There are 48 states covered by these scenarios. To consider the heterogeneity, we subdivided the simulation area into *sectors* to model at a scale that is finer than states but coarser than counties. Figure 6.14 shows the centers of the sectors and their distribution accorss the contiguous United States. . For each sector, we built nine models, one final model and eight daily models. For a given variant, the final model predicts disease duration in the sector, and each of the other models predicts one of the considered output parameters on a daily basis. For a given variant, the daily models use a given current state and the disease duration predicted by the final model to predict a new state. Additionally, we built a probability model (referred to as *shipment component*) that connected the models within different sectors.

6.3.1 Sectors

A sector is defined by a point location. A farm belonged to the sector if both existed within the same state and if the sector's location was the closest to the farm location. The sector locations were chosen by hand by placing more sectors in larger states. Table 6.2 shows the number of sectors, along with other information for the study regions that were covered by the five scenarios.

Table 6.2: The number of sectors in the study regions.

Region	Farms	Production Types	States	Sectors
Central	311,736	9	6	23
Northcentral	253,393	9	8	24
Northeastern	75,558	8	11	15
Southeastern	292,892	9	12	25
Western	156,983	9	11	42
Total	**1,090,562**	**9**	**48**	**129**

6.3.2 Shipment Component

The shipment component was created based on collected data describing county-to-county shipments. It was created such that it describes sector-to-sector shipments. The county-to-county shipments were used to compute the proportions of outgoing shipments on the sector level. Based on the counties' locations, we used the information from shipments between 3,222 counties to compute the shipments between 129 sectors.

The shipment component was used to connect the sectors and allow tracking of disease spread between the sectors. For a given sector, the shipment component shows the proportions of outgoing shipment to all sectors. The total shipments were predicted by the daily models, and the shipment component showed where shipments from a sector were likely to go. Figure 6.15 shows an example of the highest shipments from one of Texas sectors.

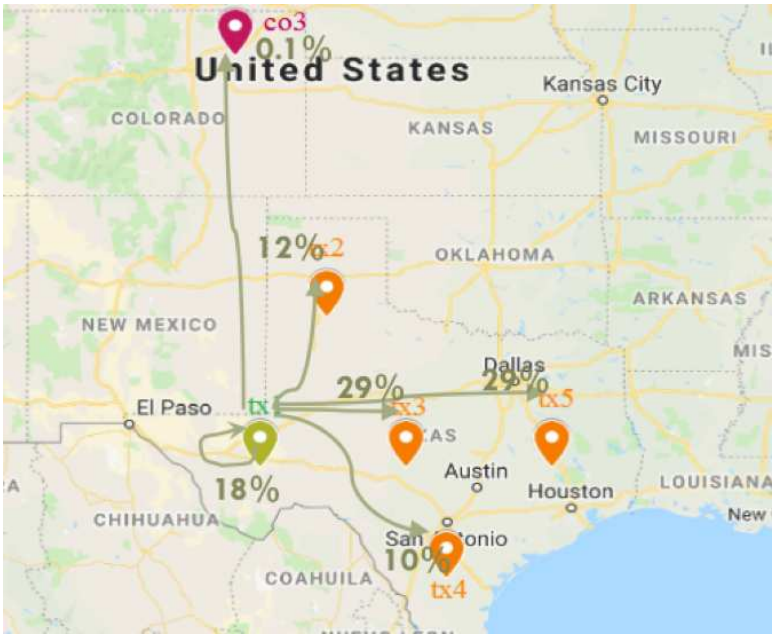


Figure 6.15: Example of sector-to-sector shipments from one of the Texas sectors.

6.3.3 Training Data for the Sectors

Building analytical models for predicting the behavior of disease spread among 48 U.S. states involved generating massive data. Training data needed to be created for each sector to capture the local characteristic. Additionally, the models had to make predictions based on the production type that was used as an initially infected farm. To support such a feature, the training data needed to be generated for each production type that was available in every sector. For each production type in every sector, we had to choose the initially infected farm of that type, generate variants, and execute the variants. Finally, the data (variants and their outputs) that were generated by using different production-types in the same sector had to be merged and used to train the sector's models. We created one replica of the sectors' scenario for each production type that was available within each sector and set an initially infected farm in each replica. We needed to generate and execute variants by using an initially infected farm for every production type in each of 129 sectors (i.e., scenario replicas). Such a process needed to be repeated 1,161 times (nine production types in 129 sectors).

In addition to the generation of massive data, the generated raw outputs are considerably large and can fill the hard disks of all of the worker machines in a few hours. Moreover, the involved tasks

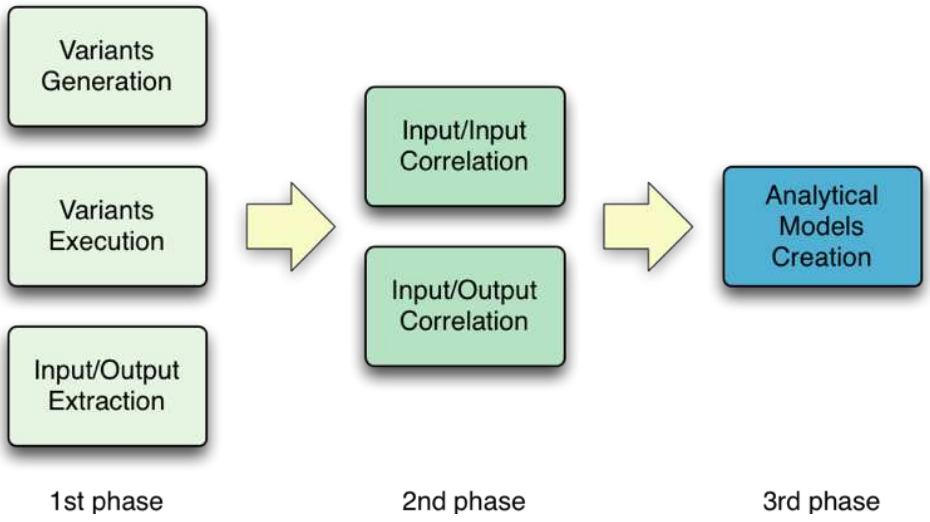


Figure 6.16: Resource maximization is accomplished by overlapping all of the tasks in three phases.

(such as variants generation, variants execution, inputs and outputs extracting, and raw outputs compression) need to be executed in a sequential manner. Such circumstances introduce a unique challenge and can lead to significantly long time to create the training data. To overcome this challenge, we had to maximize resource utilization.

6.3.4 Maximizing Resource Utilization

Maximal resource utilization could be achieved by overlapping the executions of the tasks from different phases. This solution enabled us to maximize the parallelism of the tasks' executions to reduce the overall running time. In this case, all of the involved tasks needed to be executed in three tasks, as shown in Figure 6.16.

In the first phase, the tasks are overlapped for each variant. The remote generator generates variants and appends each generated variant in one local file that will represent the input parameters. Additionally, the generator launches three new processes for each created variant. The first process is created to execute the generated variant. Once the execution is finished, two processes are automatically launched, one process to store the produced outputs in a binary and compact format to reduce the needed storage and the other process to extract the output parameters and add them to the training observations. The variants generator does not require too much mem-

ory and spends less than two minutes for creating every variant. The first process requires a low memory footprint but spends a relatively long time, while the remaining two processes need a high memory footprint and spend less execution time. After the successful execution of all of the involved processes for each variant, the variant and its output files are deleted to reduce the storage consumption.

To avoid potential resource contention that is caused by increasing the number of the generated processes that are running on one machine, we need to monitor and limit the running processes and their used resources. An upper bound of the number of generated processes is automatically set based on the machine capability. It means that each remote variants generator constantly generates variants and creates processes for them until the number of created processes reaches the upper bound. Once the upper bound of created processes is met, the remote variants generator will stop generating variants and their involved processes and resume when the number of its running processes decreases. In such a case, the processes that are generated for compressing the raw outputs and preparing outputs for the training observations will start using a part of the hard disk as a memory when real RAM fills up and more space is needed. In such a condition, these processes will proceed very slowly, and their number will increase accumulatively. For addressing this issue, the remote generator checks the available memory with respect to the number of running processes before new variants and processes can be generated. The generator automatically pauses until the two conditions are satisfied.

In second phase, the correlations between the inputs and between the inputs and the outputs are computed to reduce dimensionality and find the most important input parameters for building the analytical models. In the third phase, the most important input parameters for each output parameter are used to build the analytical models. For the execution of all of the phases, we made full use of 436 machines with tens of thousands of cores.

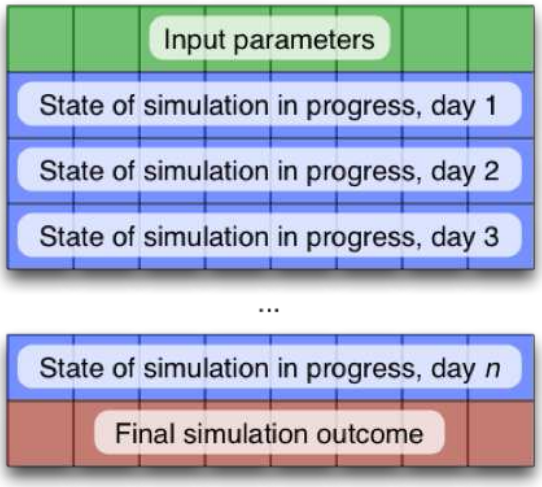


Figure 6.17: Simulation scenario: input parameters and the intermediate and final outcomes.

6.3.5 Analytical Models for the Sectors

We constructed analytical models to accurately forecast outbreaks at a region with a given set of parameters. Our approach for the construction of analytical models copes with the high data dimensionality and supports interactions between different analytical models corresponding to different regions. Furthermore, our approach synchronizes the passage of simulation days between models. Our analytical models are flexible so that they can take a set of *current* conditions and project what is likely to happen a few days out. This is different from having an analytical model that takes a set of initial conditions and produces the final output. This allows the analytical models (for a given region) to account for what is taking place in a different region. A simulation can be viewed as following (a) a set of numbers representing the input parameters, (b) a set of numbers representing the state of the in-progress simulation on each simulation day, and (c) a set of numbers representing the final simulation outcome (see Figure 6.17).

We built an analytical model for each considered output parameter associated with each sector. To reduce dimensionality and improve the models accuracy, we used the most influential input parameters for each output parameter to train the corresponding analytical model. We used the correlations between inputs and between inputs and outputs to find the most influential input. We built one analytical model for each daily output parameter for every sector. Also, we built an extra

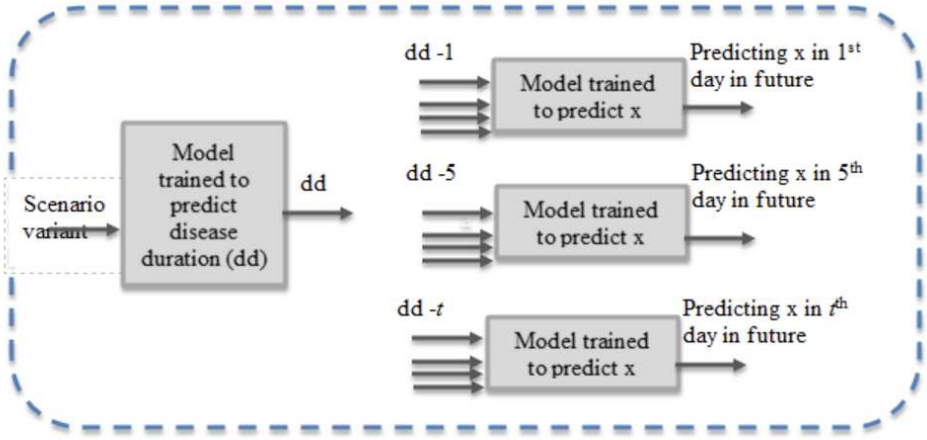


Figure 6.18: Using diease duration predicted by the final model for making predictions of intermediate outcomes of a scenario variant in a sector.

final model for each sector to predict disease duration in that sector. We built 1,161 analytical models in total. For predicting the farms' states on a daily basis, these models employed the outbreak duration that was predicted by the final model. For a given observation (variant), we used the final model to predict the number of days that the outbreak could last, adjusted it to become the number of remaining days, and then fed the daily models with the adjusted value to make predictions for the units' states. Figure 6.18 shows how the two kinds of models (daily and final models) were used together to make predictions for different days.

Chapter 7

Input and Output Ensembles

We conducted a series of system benchmarks to evaluate the efficiency of our methodology. We used the evaluation dataset to construct two ensembles with different sizes and measured system performance. Our evaluation dataset was partitioned into 1,000 portions to evaluate the performance of building ensembles on relatively large training sets. For assessing the performance of constructing ensembles on small training sets, the evaluation dataset was partitioned into 5,357 subsets. To evaluate the efficiency of our methodology, we used Apache Spark to train a global model on the evaluation dataset and contrasted the performance with that of both ensembles. In our experiments, we contrasted the training time, CPU consumption, memory usage, network I/O, and Disk I/O. Additionally, we used different learning algorithms in our experiments to assess their influence on the performance of building the global model and the ensembles.

Additionally, two partitioning methods, partitioning based on the similarity in the input space and output space, were investigated. We partitioned the data based on the similarity in the input and output space to create input and output ensembles; each was composed of 1,000 models to contrast the performance of input and output ensembles. Partitioning based on the similarity in the input space was employed to construct an ensemble with instances of models that were specialized in different regions of the input space. The output-based partitioning method was used to build an ensemble with models that were specialized for regions of the output space. We evaluated the ensembles' performance by contrasting their prediction accuracy with that of the global model, using test data that were not included in the training process.

7.1 Experimental Setup and Datasets

To build the global model and the ensembles, we used 126 machines running Fedora 30. The capabilities of these machines are as follows:

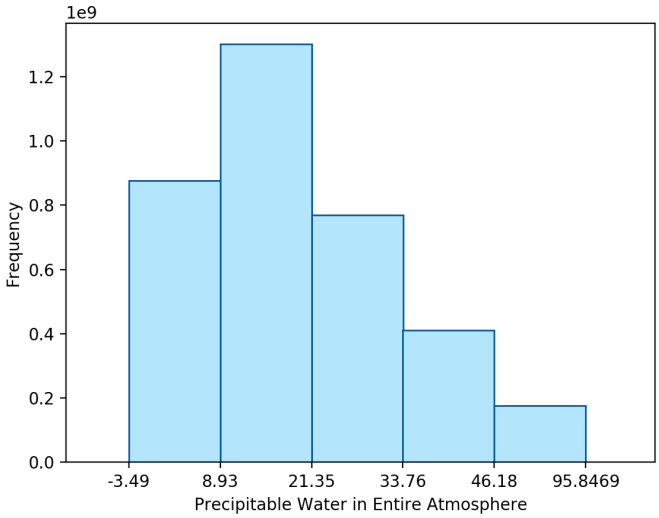


Figure 7.1: Histogram showing the values distribution of the target.

- 34 HP-DL60-G9 machines (Six-Core Intel Xeon E5-2620v3, 16GB),
- 42 HP-DL60-G9 machines (Eight-Core Intel Xeon E5-2620v4, 16GB), and
- 50 SYS-5018R-M machines (Eight-Core Intel Xeon processor E5-2620V4, 64GB).

We developed a framework to train the individual models and construct the ensembles, while PySpark was used to build the global models using a standalone cluster (Spark cluster). We have used observations from 2 years (2013 and 2014) of the NOAA dataset that is described in §5.4.1. The evaluation dataset contains 3,532,225,177 observations with 59 features. Fifty-eight features were used as input features and *precipitable_water_entire_atmosphere* was chosen as target (see Figure 7.1 for the values distribution of the target). The evaluation dataset was divided into test and training data. The test data, including 5 million observations, were used for assessing purposes, while the remaining data were used to train the global model and the ensembles.

7.2 Learning Algorithms and Models Building

We leveraged four learning algorithms (linear regression, decision tree, random forest, and gradient boosting) in our experiments. Some essential characteristics of these algorithms follow:

- Model training using linear regression can be entirely parallelized.
- In the induction of decision trees, the algorithm can simultaneously work on several independent nodes. Further, the split node operation, which is a pervasive operation, can proceed in parallel.
- Building a random forest model involves inducing independent deep trees that can be built in parallel.
- For building a gradient boosting model, shallow trees have to be sequentially created.

We developed a framework that trains the individual models and uses them to construct an ensemble. It efficiently orchestrates the learning processes of a large number the models on given machines in parallel. Once the models have been trained, our framework uses the models' instances and the partitioning information to construct the ensemble. The gate function of the ensemble uses the partitioning information that was used to partition the original data. The ensemble uses its gate function to the specialized model for a given observation to provide an accurate prediction.

7.2.1 Building the Global Model

We used Spark to training the global model on the entire dataset. Spark was deployed as a standalone with the Hadoop Distributed File System on the 126 machines. The replication factor in HDFS was set to 3. We used 12 external machines to stage the evaluation data into HDFS. The staging operation took 168 minutes. To build the global model, we tried to apply all four algorithms on the whole training data. While we could successfully train the global model on the entire data by using linear regression and decision trees, we were unable to do that by using random forest and gradient boosting. By trying to build the global model with random forest, we observed that Spark attempted to shuffle massive data that caused many executors to fail because there was no space left on the disk (250 GB was the maximum free space available on some machines.). By building a gradient boosting model, the training process never ended because Spark created long RDD lineage and tried to recompute the intermediate RDDs when some executors failed. Gradient

boosting is a sequential algorithm and becomes improper for distributed learning, especially with a large number of trees. Figure 7.1 summarizes the experimental settings for leveraging the learning algorithms to train the global model.

Table 7.1: The relative data sizes and the configurations of the learning algorithms that were used to train the global models.

Algorithm	Training Data (%)	Depth	Maximum Iterations|trees
Linear Regression	100%	-	10,000
Decision Tree	100%	30	1
Random Forest	10%	9	1,000
Gradient Boosting	20%	4	1,600

7.2.2 Building the Input Ensemble

Before we could construct the input ensemble, we had to partition the data based on the similarity in the input space. For this purpose, we used Spark to apply the K-means algorithm on 20% of data to create 1,000 clusters. The time that was needed to find the centroids was 12.1 minutes, and the time needed to partition the data was 68 minutes. The centroids were used in parallel to partition the data and disperse the data portions among the machines. The clusters' centroids were used to construct the gate function of the input ensemble. The input ensemble employed its gate function that relies on the centroids for finding the input space region to which a given observation belonged. Then, it applied the responsible models on the observation to make a prediction.

We leveraged linear regression and gradient boosting algorithms to train the models for the input ensemble. For training the individual models by using linear regression, the maximum number of iterations was set to 10,000. In this case, the algorithm stops when it converges, or when the number of iterations reaches 10,000. Additionally, the algorithm was configured to work in parallel, using the available cores to train each individual model. For the gradient boosting algorithm, we relied on the early stopping concept where the learning process will automatically stop when no accuracy improvement on validation data can be observed by adding more trees. This feature is not available for the distributed version of gradient boosting in Spark.

7.2.3 Building the Output Ensemble

For building the output ensemble, we partitioned the data based on the similarities in the output space. To achieve such partitioning, we used the decision tree algorithm that recursively finds the best split points in the input features to reduce the variance of the target values in the produced data partitions. Using Spark, the algorithm was applied to 15% of data and took 38 minutes. The split thresholds were used to build the gate function used by the ensemble. The gate function was used in a distributed manner to partition the data and distribute the data portions across the 126 machines. The partitioning and dispersing operations took 46 minutes.

All four algorithms were leveraged for training the models that constructed the output ensemble. The linear regression algorithm was configured to run in parallel with 10,000 as a maximum number of iterations. The decision tree algorithm sequentially built each trees model with a maximum depth of 5, while each random forest model was built in parallel with 1,000 trees with a maximum depth of 10. By building gradient boosting models, the early stopping concept was used to automatically find the optimum number of trees with a maximum depth of 3. The early stopping concept allowed building gradient boosting models with a number of trees that enabled the best prediction accuracy on the validation data.

7.2.4 The Framework for Ensemble Construction

The framework was developed to train individual models and construct the ensembles. The framework has two components, a driver and multiple executors. The driver accepts as input the ensemble type (input or output), the path to the partitioning information (i.e., centroids or split thresholds), and a list of machines that host the training data. When the driver starts, it sends a copy of the partitioning information to all machines and launches an executor on each machine. Each machine hosts multiple training sets. The executor works as a local scheduler that runs multiple processes in parallel on the hosted machine based on its capability. It collects information about the hosted machine (such as memory and number of cores) and the sizes of the local training sets. It then starts with the largest training sets by running processes that perform correlation analysis to

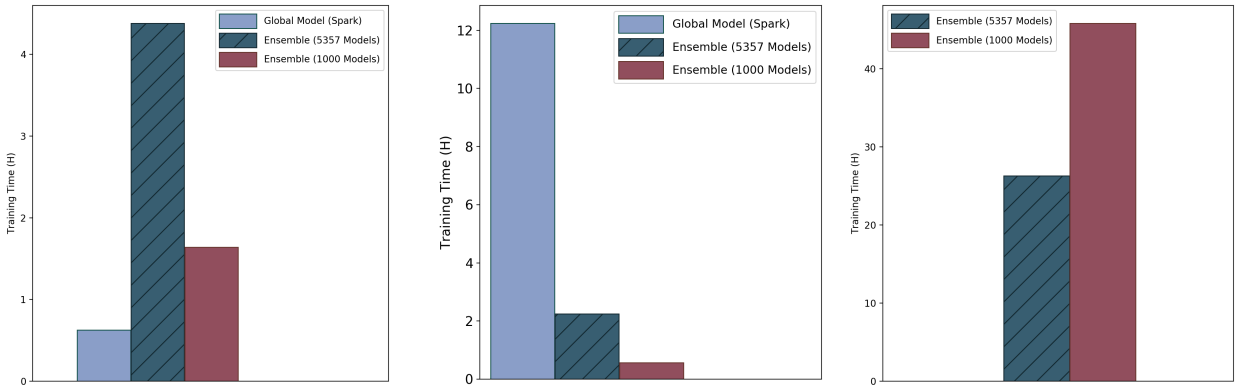
find the most influential features. The executor launches the model training process for any training set whose correlation analysis has been completed. Once a model has been trained, it is sent to all cluster machines. When all of the executors have trained their models locally, each machine receives a copy of the ensemble's models. After all of the executors are completed, the driver runs a light process on each machine to construct an ensemble of a given type and dump it to the local disk. The purpose of replicating the ensemble on all machines is to enable extending the ensemble by using the stacking method.

Each executor monitors the local resource usage and attempts to maximally use the available computation resources by starting with large training sets and adding new processes for other training sets based on the available resources. On each machine, there is usually a mix of processes running on training sets of different sizes. Once one of the running processes fails, the executor writes the relevant information on a disk to enable a manual debugging and fixing of the associated problem.

The benchmarks were performed to consider only the training processes by excluding the processes that determined the most influential input features and the processes that found the best optimum setting of the hyperparameters of the algorithm. Before running the benchmarks, we used our framework to identify the most important input features and the hyperparameters values that led to the best model. For our benchmarks, we used the framework only to train the models that were previously determined. We did the same thing with Spark to contrast the aspects that were associated with actual training processes.

7.3 Ensembles Performance using Small and Large Training Sets

We evaluated independent model training for small and large ensembles and contrasted the system performance with that of the global model that was built by using Spark. The small ensemble was built on 1,000 large data portions, while the large ensemble was built on 5,357 small data



(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

Figure 7.2: The training time of building the global model and ensembles of different sizes by using various learning algorithms.

portions. We contrasted the training time, disk I/O and network I/O, CPU utilization, and memory usage for the global model and both ensembles.

7.3.1 Training Time

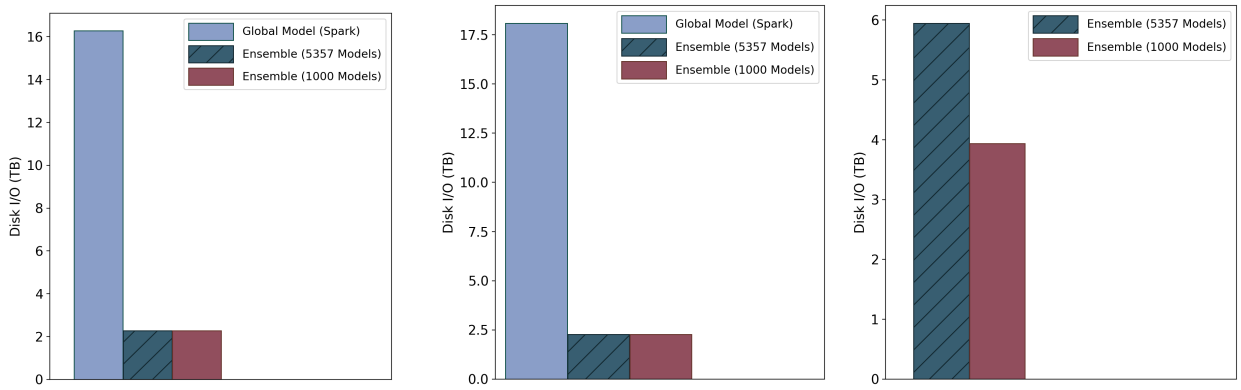
We contrasted the training time of the global model and two ensembles with different sizes for different learning algorithms. As we can see in Figure 7.2a, building the global model by using linear regression required less time compared with our methodology, which was represented by two ensembles of different sizes. The parallel training enabled by Spark achieved very short training time because linear regression parallelized with minimum overhead. By parallelizing the training process of the global model and each model of both ensembles, the training time was essentially influenced by the size of the training sets. It reveals that fitting a model on a large training set benefits more from the parallelism. The same effect can also be observed by contrasting the training times of both ensembles. Constructing ensemble (with 1,000 models) whose models were fitted to larger training sets took less time than the ensemble (with 5,357) whose models were trained on smaller training sets. From these results, we can conclude that applying parallelizable algorithms in a distributed manner to build a global model on voluminous data significantly reduces the training time.

Figure 7.2b illustrates the training time for building the decision trees for the global model and both ensembles. For building the global model by using the decision tree algorithm, Spark concurrently created the tree's nodes that were at the same level. For constructing the ensemble, each tree model was sequentially trained. As the figure shows, insights could be gained from the same dataset faster by using multiple models rather than one global model that was sequentially trained. We also observed that the construction of ensemble with 1,000 models involved less time than the construction of the ensemble with 5,357 models. In both ensembles, our methodology significantly reduced the training time of the algorithms that were working in a sequential manner.

Figure 7.2c shows the time that was needed to construct both ensembles whose models were trained by using the gradient boosting algorithm. For both ensembles, the algorithm was applied to train each of the individual models in a sequential manner without even parallelizing the expensive split operation involved. In such a case, we contrast the training time of a sequential training of models on small and large data sets. As we can see, building the models on smaller training sets required less time than did training the models on larger datasets. This was caused by the split operation, which takes a significantly longer time as the data get larger. This operation searches the input space to find the best split thresholds to partition the data.

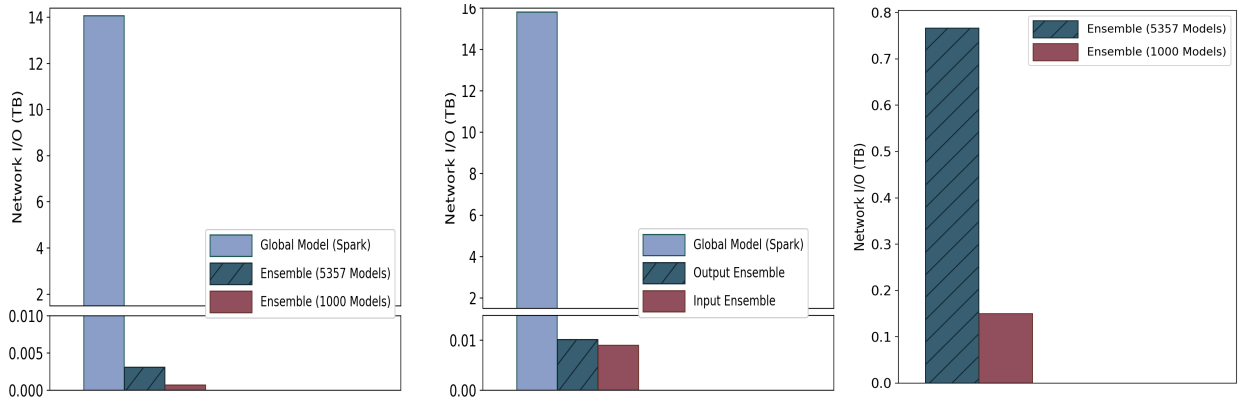
7.3.2 Disk I/O and Network I/O

We plotted the cumulative disk I/O and network I/O that were observed during the training of the global model and the models for our ensembles. Figures, 7.3a and 7.4a, demonstrate the disk I/O and network I/O that resulted by applying the parallel version of linear regression to train the models. Compared with our methodology, training a global model by using Spark involved shuffling, reading, and writing massive data. The high disk I/O and network I/O resulted from different sources. Examples of these sources are sending data into tasks that are running on different machines, reading from and writing to HDFS, shuffling data across all of the machines in each iteration, and dealing with distributed RDDs that need to be read from and flushed to disk. Our methodology involves only shuffling the trained models and some instruction data between



(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

Figure 7.3: The cumulative disk I/O of building global model and ensembles with different sizes using various learning algorithms.

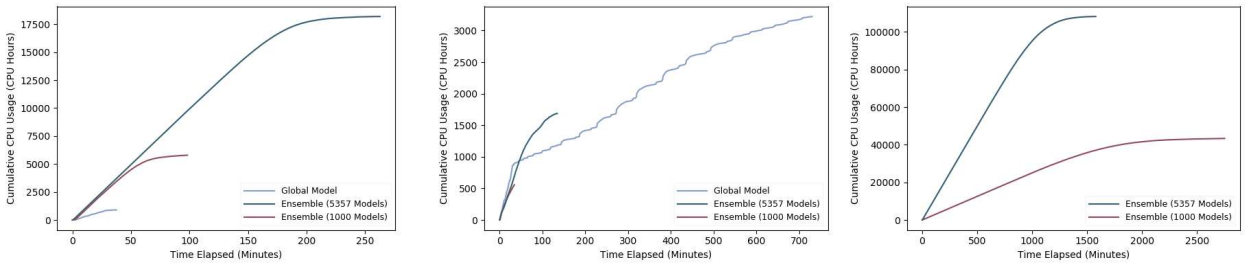


(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

Figure 7.4: The cumulative network I/O of building global model and ensembles with different sizes using various learning algorithms.

the driver and the executors. Additionally, the methodology involves reading the local data from disks and storing the models on all of the machines. In the figure, we can see that constructing the small and large ensembles shows a similar result. The results indicate that our methodology significantly reduces the disk I/O and network I/O compared with a global model that was trained in a distributed manner.

Figures, 7.3b and 7.4b, demonstrate the disk I/O and network I/O that resulted by applying the decision tree to construct a global model and both ensembles. As we can see, Spark shuffles massive data and performs read and write operations for huge data. Besides, we can notice that



(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

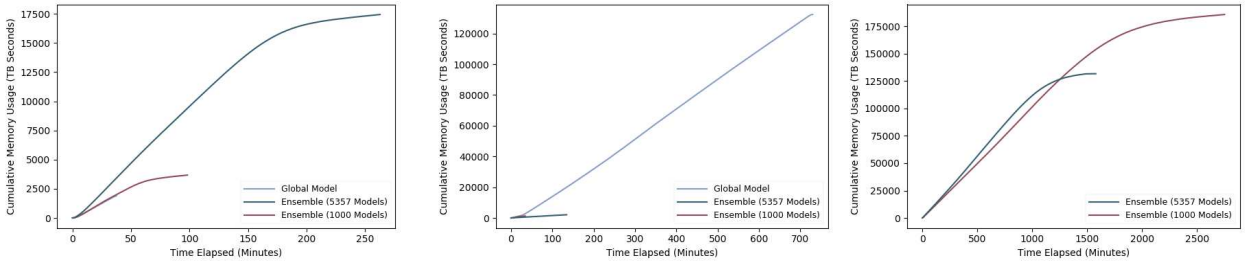
Figure 7.5: Cumulative CPU usage (CPU hours) that was observed while the decision tree algorithm was used to train the global model and the ensembles of different sizes.

both ensembles involved a comparable Disk I/O. Figures, 7.3c and 7.4c, demonstrate the disk I/O and network I/O that resulted by applying the gradient boosting algorithm to construct the ensembles with different sizes. It is notable that the large ensemble (5,357) reads and writes more data than the small ensemble (1,000) does. The reason for that is the ensemble with a large number of models that involved shuffling and storing more models.

7.3.3 CPU Utilization

Figure 7.5a demonstrates CPU utilization when linear regression was used to train the global model and both ensembles. While the building of both ensembles involved using approximately 100% of the CPU resources, training the global model using Spark used less than 50% of the CPU resources. The construction of both ensembles consumed high CPU resources because each model of these ensembles was built in parallel. Parallelizing a learning process on a single machine leads to leveraging the available cores on that machine.

Figure 7.5b shows the CPU consumption during training the global model and both ensembles by using the decision tree algorithm. Spark needed to synchronize after building each level of global tree that was built with 30 levels, resulting in a reduction of CPU utilization. Compared with the global model, building the models for the ensembles used higher CPU resources. However, the CPU utilization of the ensembles construction was not high because building each decision tree on a small training set required a short time, which increases the overhead between the completed and started processes.



(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

Figure 7.6: Cumulative memory usage (TB seconds) that was observed while the gradient boosting algorithm was used to train the global model and the ensembles of different sizes.

Figure 7.5c describes CPU utilization as the gradient boosting algorithm was employed to train the models for both ensembles. As we can see, building ensemble with 5,357 models on small training sets enabled nearly 100% CPU consumption, while building the ensemble with 1,000 models on large training sets used roughly 25% of the CPU resources. Using large training sets to train models in parallel prevented using all of the CPU resources on each machine because each training process requires high memory usage, which limits the number of concurrently running processes.

7.3.4 Memory Usage

Figure 7.6a shows the memory usage during training the global model and the models for both ensembles using linear regression. As we can see, Spark reserved nearly 1 TB of memory for the entire training time of the global model. On the other hand, building the models for the ensembles involved using about 1 TB during the beginning part of the training process. Subsequently, the memory usage gradually became lower as the training process proceeded.

Figure 7.6b shows the memory usage by the training the global model and the small and large ensemble using decision tree. As the figure shows, Spark consumed high memory for most of the training process, while building decision tree models for both ensembles used low memory.

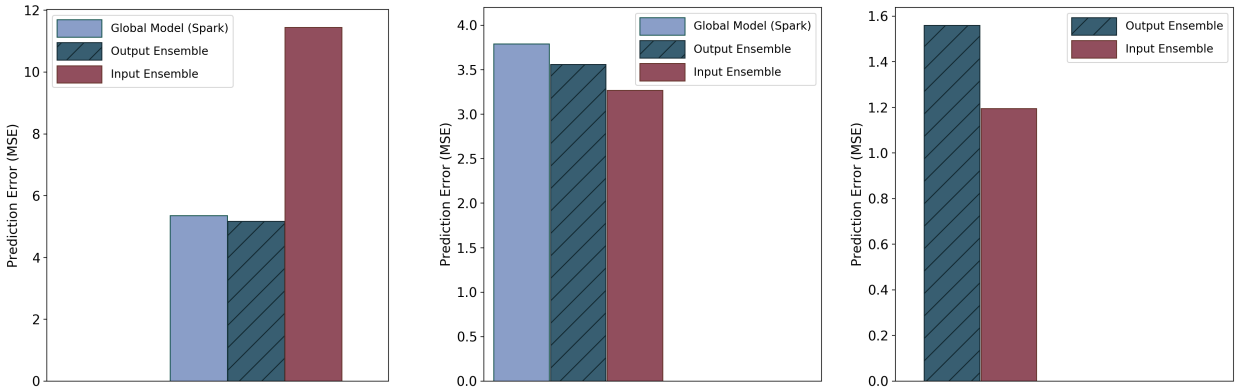
Figure 7.6c exhibits the memory that was used during the construction of both ensembles using gradient boosting. The results in figures show the same pattern of memory usage for building the ensembles. However, building the ensembles with 5,357 models that were fit on smaller train-

ing sets required more memory footprint. For the reason that more small training sets can fit in memory, memory usage was higher for building gradient boosting models on the smaller data sets.

7.4 Prediction Performance

To evaluate the accuracy of input and output ensembles, we built each ensemble on 1,000 data portions and contrasted the ensembles' accuracy with the accuracy of the global model. We measured MSE by applying the global model and the constructed ensembles on test data that have not been used in the training processes. Figures, 7.7a, 7.7b, and 7.7c, shows the prediction error measured by MSE for the linear regression, decision tree, and gradient boosting. In general, input and output ensembles enabled more accurate predictions than the global model. The reason for that is the global model trained on whole data cannot efficiently capture local patterns, while our methodology enables capturing the local patterns using multiple models.

There is one corner case where the input ensemble constructed using linear regression provided the worse prediction accuracy. The reason could be that the training sets produced by partitioning the data based on the similarities in input space include patterns that were inefficient to be captured by linear regression. On the other hand, gradient boosting models seemed to be specialized for such patterns. Applying gradient boosting models on the same test data provided the best prediction accuracy overall.



(a) Linear Regression **(b)** Decision Tree **(c)** Gradient Boosting

Figure 7.7: The prediction error (MSE) of the global model, input ensemble and output ensemble built using different learning algorithms.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

As data volumes keep growing, there are increasing interests in gaining insights from the data and leverage them to understanding phenomena or make predictions. However, rising in date volumes makes efficient learning from such data more challenging. We studied the related approaches and tried to combine their strength in our proposed methodology. Our methodology merges the predictive power of ensembles and a mixture of experts approaches and relies on partitioning methods to maximize the gained insights in a timely fashion. While partitioning is used by several algorithms to learn from data, our methodology exploits some partitioning methods to divide the problem into multiple problems that are sufficiently simple to learn from them more efficiently.

Additionally, the methodology uses such partitioning methods to make independent and concurrent learning on large-scale data achievable. Such training enables our methodology to maximize resource utilization and provide a scalable solution for effective learning from voluminous datasets. By relying on independent models training using manageable subsets of the data, our methodology allows leveraging and combining desirable algorithms to exploit their strength for solving a given problem.

In this dissertation, we described the proposed methodology for partitioning voluminous datasets and building a scaleable ensemble in a distributed environment. We applied the methodology to a real-world petabyte climate dataset by relying on geospatial partitioning and constructing ensemble with 60,922 models that are specialized for different geospatial areas. Additionally, we analyzed petabytes of epidemiology data to build analytical models that allow assessing the epidemiological impact and economic costs of livestock disease outbreaks at the national scale. This involved training 1,161 analytical models that were specially and temporally coupled to model the

disease dynamics. The analytical models were built to capture phenomena associated with smaller geospatial areas.

Besides, we used a dataset containing 3,532,225,177 observations to build a global model using Spark and to construct 2 ensembles, one with 1,000 models and the other with 5,357 models, using our methodology. Different aspects (such as training time, disk I/O, network I/O, and CPU and memory usage) associated with the training process of the global model and the ensembles were measured and contrasted. The results have shown that ensembles enable better resource utilization for most of the benchmark experiments. Additionally, we contrasted the prediction accuracy of the global model, input ensemble, and output ensemble and observed that the ensembles built using our methodology provide more accurate predictions than the global model.

RQ1: *How can we reduce model training times?*

For enabling a significant reduction in training times and achieve high throughput in terms of the number of evaluated observations in a second, the methodology supports independent and concurrent training of models on multiple machines with data locality. Additionally, we use only the most influential input features for training the models.

RQ2: *How can we ensure that the reduction in training times is not at the expense of accuracy?*

The methodology relies on different strategies to preserve good accuracy and ensure generalization. The methodology relies on feature ranking with collinearity removal to identify the most influential input features and uses them to build accurate models. Further, the ensemble models are trained on high-quality data sets, each of which contains observations that share some common characteristics. In such a case, each model will learn the local patterns and hidden trends in the assigned data. Each of the ensemble models will be specialized for a subset of observation to provide high prediction accuracy. Additionally, the methodology applies a stacking method to improve the accuracy further for the specialized models.

RQ2:*How can we ensure that the methodology scales with increases in data volumes and the number of available machines?*

Coping with the rise in data volume by adding new machines should keep the training times nearly

constant. Since our methodology relies on independent models training, and there are no bottlenecks, our methodology is scalable and can deal with data increase by adding more machines to make the time constant. Additionally, adding more machines without data growth will reduce the training time.

8.2 Future Work

There are increasing interests in developing frameworks that enable iterative computations on distributed data in a distributed manner. The frameworks try to reduce the training time and increase the scalability by collocating the computations with distributed data portions. Using these frameworks to apply the algorithms that run sequentially or involve shuffling massive data leads to poor resource utilization, unscalability, and high training time. For improving the performance in such cases, the learning algorithm has to be restricted or modifying to become system friendly. For instance, Spark limits the maximum depth of the decision trees into 30 and requires frequent checkpointing for the intermediate computations in a long dependency graph.

Our methodology provides an alternative solution for scalable learning from large-scale datasets and requires more attention and investigations. Our future work will focus on developing and optimizing a framework for partitioning the data, dispersing the partitioned data across a collection of machines, training models, and constructing an ensemble. Two main components in the framework need to be optimized. One optimization requires finding how the data partitions should be distributed over clusters of commodity hardware to reduce the train time and enable better resource utilization. This entails finding the optimal data distribution and optimizing the local scheduling on each machine. Furthermore, we plan to add a feature to the framework that enables moving some computations to the GPU to maximize resource utilization.

Partitioning is a crucial feature in our methodology and requires investigating its effect on accuracy improvement. Therefore, the second optimization could be finding the most proper partitioning for a given dataset to improve the prediction and classification performance. We will study different partitioning methods, including the ones used by some learning methods, and try

to optimize one that partition the data to separate the patterns and relations in different data sets. Additionally, the framework should allow experimenting with different partitioning methods and ensemble structures to find new learning structures. Because our framework does not support distributed computations, we will work on enabling our framework to use a distributed framework (such as Spark) to perform computations on data in parallel.

Bibliography

[1] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: the digitization of the world from edge to core," *Seagate, https://www. seagate. com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf*, 2018.

[3] P. B. Berra, C.-Y. Chen, A. Ghafoor, and T. D. Little, "Issues in networking and data management of distributed multimedia systems," in *High-Performance Distributed Computing, 1992.(HPDC-1), Proceedings of the First International Symposium on*, pp. 4–15, IEEE, 1992.

[4] M. Parashar, "Big data challenges in simulation-based science.," in *DICT@ HPDC*, pp. 1–2, 2014.

[5] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 583–598, USENIX Association, Oct. 2014.

[6] J. Fan, F. Han, and H. Liu, "Challenges of big data analysis," *National science review*, vol. 1, no. 2, pp. 293–314, 2014.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 59–72, ACM, 2007.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 17–30, 2012.

[10] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–10, IEEE, 2009.

[11] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting started with storm*. " O'Reilly Media, Inc.", 2012.

[12] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables.," in *OSDI*, vol. 10, pp. 1–14, 2010.

[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive-a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 996–1005, IEEE, 2010.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, ACM, 2008.

[15] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, pp. 1223–1231, 2013.

[16] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, 2014.

[17] K. Canini, T. Chandra, E. Ie, J. McFadden, K. Goldman, M. Gunter, J. Harmsen, K. LeFevre, D. Lepikhin, T. Llinares, *et al.*, "Sibyl: A system for large scale supervised machine learning," *Technical Talk*, 2012.

[18] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

[19] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1111–1133, 2014.

[20] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*, pp. 1–15, Springer, 2000.

[21] O. Okun, *Supervised and unsupervised ensemble methods and their applications*, vol. 126. Springer, 2008.

[22] Z.-H. Zhou, "When semi-supervised learning meets ensemble learning," *Frontiers of Electrical and Electronic Engineering in China*, vol. 6, no. 1, pp. 6–16, 2011.

[23] Z.-H. Zhou, *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC, 2012.

[24] J. Mendes-Moreira, C. Soares, A. M. Jorge, and J. F. D. Sousa, "Ensemble approaches for regression: A survey," *Acm computing surveys (csur)*, vol. 45, no. 1, pp. 1–40, 2012.

[25] H. Mitchell, "Ensemble learning," in *Data Fusion: Concepts and Ideas*, pp. 295–321, Springer, 2012.

[26] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[27] B. Efron, "Bootstrap methods: another look at the jackknife," in *Breakthroughs in statistics*, pp. 569–593, Springer, 1992.

[28] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.

[29] P. M. Domingos, "Why does bagging work? a bayesian account and its implications.," in *KDD*, pp. 155–158, Citeseer, 1997.

[30] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[31] M. Pal, "Random forest classifier for remote sensing classification," *International Journal of Remote Sensing*, vol. 26, no. 1, pp. 217–222, 2005.

[32] P. O. Gislason, J. A. Benediktsson, and J. R. Sveinsson, "Random forests for land cover classification," *Pattern Recognition Letters*, vol. 27, no. 4, pp. 294–300, 2006.

[33] M. Belgiu and L. Drăguţ, "Random forest in remote sensing: A review of applications and future directions," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 114, pp. 24–31, 2016.

[34] A. E. Maxwell, T. A. Warner, and F. Fang, "Implementation of machine-learning classification in remote sensing: An applied review," *International Journal of Remote Sensing*, vol. 39, no. 9, pp. 2784–2817, 2018.

[35] Z.-H. Zhou and J. Feng, "Deep forest," *arXiv preprint arXiv:1702.08835*, 2017.

[36] K. Miller, C. Hettinger, J. Humpherys, T. Jarvis, and D. Kartchner, "Forward thinking: Building deep random forests," *arXiv preprint arXiv:1705.07366*, 2017.

[37] Y. Kong and T. Yu, "A deep neural network model using random forest to extract feature representation for gene expression data classification," *Scientific reports*, vol. 8, no. 1, pp. 1–9, 2018.

[38] S. Kim, M. Jeong, and B. C. Ko, "Interpretation and simplification of deep forest," *arXiv preprint arXiv:2001.04721*, 2020.

[39] M. Kumar and M. Thenmozhi, "Forecasting stock index movement: A comparison of support vector machines and random forest," in *Indian institute of capital markets 9th capital markets conference paper*, 2006.

[40] M. J. Kane, N. Price, M. Scotch, and P. Rabinowitz, "Comparison of arima and random forest time series models for prediction of avian influenza h5n1 outbreaks," *BMC bioinformatics*, vol. 15, no. 1, p. 276, 2014.

[41] H. Tyralis and G. Papacharalampous, "Variable selection in time series forecasting using random forests," *Algorithms*, vol. 10, no. 4, p. 114, 2017.

[42] M. Kearns, "Thoughts on hypothesis boosting," *Unpublished manuscript*, vol. 45, p. 105, 1988.

[43] R. E. Schapire, "The strength of weak learnability," *Machine learning*, vol. 5, no. 2, pp. 197–227, 1990.

[44] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[45] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[46] K. M. Ting and I. H. Witten, "Issues in stacked generalization," *Journal of artificial intelligence research*, vol. 10, pp. 271–289, 1999.

[47] J. R. Quinlan, "C4. 5: Programming for machine learning," *Morgan Kauffmann*, vol. 38, p. 48, 1993.

[48] B. Cestnik *et al.*, "Estimating probabilities: a crucial task in machine learning.," in *ECAI*, vol. 90, pp. 147–149, 1990.

[49] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine learning*, vol. 6, no. 1, pp. 37–66, 1991.

[50] S. Džeroski and B. Ženko, "Is combining classifiers with stacking better than selecting the best one?," *Machine learning*, vol. 54, no. 3, pp. 255–273, 2004.

[51] N. Ravi, N. Dandekar, P. Mysore, and M. L. Littman, "Activity recognition from accelerometer data," in *Aaai*, vol. 5, pp. 1541–1546, 2005.

[52] A. Büyükçakir, H. Bonab, and F. Can, "A novel online stacked ensemble for multi-label stream classification," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pp. 1063–1072, 2018.

[53] W. DING and S. WU, "Abc-based stacking method for multilabel classification," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 27, no. 6, pp. 4231–4245, 2019.

[54] D. Karaboga, "An idea based on honey bee swarm for numerical optimization," tech. rep., Technical report-tr06, Erciyes university, engineering faculty, computer . . . , 2005.

[55] N. Singh and P. Singh, "Stacking-based multi-objective evolutionary ensemble framework for prediction of diabetes mellitus," *Biocybernetics and Biomedical Engineering*, vol. 40, no. 1, pp. 1–22, 2020.

[56] S. Frühwirth-Schnatter, *Finite mixture and Markov switching models*. Springer Science & Business Media, 2006.

[57] S. Yuksel, J. Wilson, and P. Gader, "Twenty years of mixture of experts," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 23, pp. 1177–1193, 08 2012.

[58] S. Masoudnia and R. Ebrahimpour, "Mixture of experts: a literature survey," *Artificial Intelligence Review*, vol. 42, no. 2, pp. 275–293, 2014.

[59] I. C. Gormley and S. Frühwirth-Schnatter, "Mixtures of experts models," 2018.

[60] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, G. E. Hinton, *et al.*, "Adaptive mixtures of local experts.," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[61] S. J. Nowlan and G. E. Hinton, "Evaluation of adaptive mixtures of competing experts," in *Advances in neural information processing systems*, pp. 774–780, 1991.

[62] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the em algorithm," *Neural computation*, vol. 6, no. 2, pp. 181–214, 1994.

[63] M. K. Titsias and A. Likas, "Mixture of experts classification using a hierarchical mixture model," *Neural Computation*, vol. 14, no. 9, pp. 2221–2244, 2002.

[64] C. M. Bishop and M. Svenskn, "Bayesian hierarchical mixtures of experts," in *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, pp. 57–64, Morgan Kaufmann Publishers Inc., 2002.

[65] W. Zhao, Y. Gao, S. A. Memon, B. Raj, and R. Singh, "Hierarchical routing mixture of experts," *CoRR*, vol. abs/1903.07756, 2019.

[66] P. Moerland, "Some methods for training mixtures of experts," tech. rep., IDIAP, 1997.

[67] G. J. McLachlan and D. Peel, *Finite mixture models*. John Wiley & Sons, 2004.

[68] T.-I. Lin, "Robust mixture modeling using multivariate skew t distributions," *Statistics and Computing*, vol. 20, no. 3, pp. 343–356, 2010.

[69] T. I. Lin, J. C. Lee, and W. J. Hsieh, "Robust mixture modeling using the skew t distribution," *Statistics and computing*, vol. 17, no. 2, pp. 81–92, 2007.

[70] H. D. Nguyen and G. J. McLachlan, "Laplace mixture of linear experts," *Computational Statistics & Data Analysis*, vol. 93, pp. 177–191, 2016.

[71] F. Chamroukhi, "Robust mixture of experts modeling using the t distribution," *Neural Networks*, vol. 79, pp. 20–36, 2016.

[72] B. Tang, M. I. Heywood, and M. Shepherd, "Input partitioning to mixture of experts," in *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290)*, vol. 1, pp. 227–232, IEEE, 2002.

[73] J. Goodband, O. C. Haas, and J. A. Mills, "A mixture of experts committee machine to design compensators for intensity modulated radiation therapy," *Pattern recognition*, vol. 39, no. 9, pp. 1704–1714, 2006.

[74] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.

[75] M. H. Nguyen, H. A. Abbass, and R. I. Mckay, "A novel mixture of experts model based on cooperative coevolution," *Neurocomputing*, vol. 70, no. 1-3, pp. 155–163, 2006.

[76] R. Ebrahimpour, E. Kabir, and M. R. Yousefi, "Improving mixture of experts for view-independent face recognition using teacher-directed learning," *Machine Vision and Applications*, vol. 22, no. 2, pp. 421–432, 2011.

[77] T. Lasota, B. Londzin, Z. Telec, and B. Trawiński, "Comparison of ensemble approaches: Mixture of experts and adaboost for a regression problem," in *Asian Conference on Intelligent Information and Database Systems*, pp. 100–109, Springer, 2014.

[78] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[79] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[80] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[81] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[82] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[83] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[84] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly, "Efficient iterative processing in the scidb parallel array engine," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, p. 39, ACM, 2015.

[85] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: a new platform for distributed machine learning on big data," *Big Data, IEEE Transactions on*, vol. 1, no. 2, pp. 49–67, 2015.

[86] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[87] R. Zhu, P. Ma, M. W. Mahoney, and B. Yu, "Optimal subsampling approaches for large sample linear regression," *arXiv preprint arXiv:1509.05111*, 2015.

[88] H. Wang, R. Zhu, and P. Ma, "Optimal subsampling for large sample logistic regression," *Journal of the American Statistical Association*, no. just-accepted, 2017.

[89] R. Zhu, "Poisson subsampling algorithms for large sample linear regression in massive data," *arXiv preprint arXiv:1509.02116*, 2015.

[90] P. Ma and X. Sun, "Leveraging for big data regression," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 1, pp. 70–76, 2015.

[91] R. Zhu, "Gradient-based sampling: An adaptive importance sampling for least-squares," in *Advances in Neural Information Processing Systems*, pp. 406–414, 2016.

[92] L. Han, T. Yang, and T. Zhang, "Local uncertainty sampling for large-scale multi-class logistic regression," *arXiv preprint arXiv:1604.08098*, 2016.

[93] J. Petrak, "Fast subsampling performance estimates for classification algorithm selection," in *Proceedings of the ECML-00 Workshop on Meta-Learning: Building Automatic Advice Strategies for Model Selection and Method Combination*, pp. 3–14, 2000.

[94] T. Borovicka, M. Jirina Jr, P. Kordik, and M. Jirina, "Selecting representative data sets," in *Advances in data mining knowledge discovery and applications*, InTech, 2012.

[95] C.-J. Hsieh, S. Si, and I. Dhillon, "A divide-and-conquer solver for kernel support vector machines," in *International Conference on Machine Learning*, pp. 566–574, 2014.

[96] Q. Guo, B.-W. Chen, F. Jiang, X. Ji, and S.-Y. Kung, "Efficient divide-and-conquer classification based on feature-space decomposition," *arXiv preprint arXiv:1501.07584*, 2015.

[97] Y. Tian, X. Ju, and Y. Shi, "A divide-and-combine method for large scale nonparallel support vector machines," *Neural Networks*, vol. 75, pp. 12–21, 2016.

[98] Y. Zhang, J. Duchi, and M. Wainwright, "Divide and conquer kernel ridge regression," in *Conference on Learning Theory*, pp. 592–617, 2013.

[99] M. Farrash and W. Wang, "How data partitioning strategies and subset size influence the performance of an ensemble?," in *Big Data, 2013 IEEE International Conference on*, pp. 42–49, IEEE, 2013.

[100] J. D. Basilico, M. A. Munson, T. G. Kolda, K. R. Dixon, and W. P. Kegelmeyer, "Comet: A recipe for learning and using large ensembles on massive data," in *Data mining (ICDM), 2011 IEEE 11th international conference on*, pp. 41–50, IEEE, 2011.

[101] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1426–1437, 2009.

[102] L. Breiman, "Pasting small votes for classification in large databases and on-line," *Machine learning*, vol. 36, no. 1, pp. 85–103, 1999.

[103] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[104] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[105] "Apache hadoop." http://hadoop.apache.org, 2011.

[106] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[107] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[108] National Oceanic and Atmospheric Administration, "The north american mesoscale forecast system," 2016.

[109] M. Lichman, "UCI machine learning repository," 2015.

[110] M. Malensek, W. Budgaga, R. Stern, S. Pallickara, and S. Pallickara, "Trident: Distributed storage, analysis, and exploration of multidimensional phenomena," *IEEE Transactions on Big Data*, 2018.

[111] D. Rammer, W. Budgaga, T. Buddhika, S. Pallickara, and S. L. Pallickara, "Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 468–477, IEEE, 2018.

[112] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st ed., 2012.

[113] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[114] C. Cortes and V. Vapnik, "Support vector machine," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[115] L. Wang, *Support vector machines: theory and applications*, vol. 177. Springer Science & Business Media, 2005.

[116] J. L. Bentley, "Multidimensional divide-and-conquer," *Communications of the ACM*, vol. 23, no. 4, pp. 214–229, 1980.

[117] J. L. Bentley and M. I. Shamos, "Divide-and-conquer in multidimensional space," in *Proceedings of the eighth annual ACM symposium on Theory of computing*, pp. 220–230, 1976.

[118] Y. Hu, S. Luo, L. Han, L. Pan, and T. Zhang, "Deep supervised learning with mixture of neural networks," *Artificial Intelligence in Medicine*, vol. 102, p. 101764, 2020.

[119] J. Wang and V. Saligrama, "Local supervised learning through space partitioning," *Advances in Neural Information Processing Systems*, vol. 1, 01 2012.

[120] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer networks and ISDN systems*, vol. 29, no. 8-13, pp. 1157–1166, 1997.

[121] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[122] H. Koga, T. Ishibashi, and T. Watanabe, "Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing," *Knowledge and Information Systems*, vol. 12, no. 1, pp. 25–53, 2007.

[123] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[124] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition.," in *BMVC*, pp. 789–798, 2006.

[125] L. Rokach and O. Maimon, "Clustering methods," in *Data mining and knowledge discovery handbook*, pp. 321–352, Springer, 2005.

[126] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping multidimensional data*, pp. 25–71, Springer, 2006.

[127] S. Vega-Pons and J. Ruiz-Shulcloper, "A survey of clustering ensemble algorithms," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 25, no. 03, pp. 337–372, 2011.

[128] A. Fahad, N. Alshatri, Z. Tari, A. Alamri, I. Khalil, A. Y. Zomaya, S. Foufou, and A. Bouras, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE transactions on emerging topics in computing*, vol. 2, no. 3, pp. 267–279, 2014.

[129] M. Steinbach, L. Ertöz, and V. Kumar, "The challenges of clustering high dimensional data," in *New directions in statistical physics*, pp. 273–309, Springer, 2004.

[130] A. Hinneburg and D. A. Keim, "Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering," 1999.

[131] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?," in *International conference on database theory*, pp. 217–235, Springer, 1999.

[132] M. Brown, K. Bossley, D. Mills, and C. Harris, "High dimensional neurofuzzy systems: overcoming the curse of dimensionality," in *Proceedings of 1995 IEEE International Conference on Fuzzy Systems.*, vol. 4, pp. 2139–2146, IEEE, 1995.

[133] F. Daum and J. Huang, "Curse of dimensionality and particle filters," in *2003 IEEE Aerospace Conference Proceedings (Cat. No. 03TH8652)*, vol. 4, pp. 4_1979–4_1993, IEEE, 2003.

[134] F. Y. Kuo and I. H. Sloan, "Lifting the curse of dimensionality," *Notices of the AMS*, vol. 52, no. 11, pp. 1320–1328, 2005.

[135] F. Bach, "Breaking the curse of dimensionality with convex neural networks," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 629–681, 2017.

[136] L. Grüne, "Overcoming the curse of dimensionality for approximating lyapunov functions with deep neural networks under a small-gain condition," *arXiv preprint arXiv:2001.08423*, 2020.

[137] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[138] W. Budgaga, M. Malensek, S. Lee Pallickara, and S. Pallickara, "A framework for scalable real-time anomaly detection over voluminous, geospatial data streams," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, p. e4106, 2017.

[139] M. Malensek, S. Lee Pallickara, and S. Pallickara, "Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals," *Future Gener. Comput. Syst.*, vol. 29, pp. 1049–1061, June 2013.

[140] M. Malensek, S. L. Pallickara, and S. Pallickara, "Expressive query support for multidimensional data in distributed hash tables," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, (Washington, DC, USA), pp. 31–38, IEEE Computer Society, 2012.

[141] Wikipedia contributors, "Geohash — Wikipedia, the free encyclopedia," 2016. [Online; accessed 12-June-2016].

[142] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[143] V. Vryniotis, "Datumbox machine learning framework," *Obtenido de Datumbox. com-Documentation-API 1.0 v*, 2015.

[144] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[145] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 169–178, 2000.

[146] P. C. Mahalanobis, "On the generalized distance in statistics," National Institute of Science of India, 1936.

[147] C. E. Antoniak, "Mixtures of dirichlet processes with applications to bayesian nonparametric problems," *The annals of statistics*, pp. 1152–1174, 1974.

[148] M. D. Escobar and M. West, "Bayesian density estimation and inference using mixtures," *Journal of the american statistical association*, vol. 90, no. 430, pp. 577–588, 1995.

[149] D. Görür and C. E. Rasmussen, "Dirichlet process gaussian mixture models: Choice of the base distribution," *Journal of Computer Science and Technology*, vol. 25, no. 4, pp. 653–664, 2010.

[150] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation-based anomaly detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 1, pp. 1–39, 2012.

[151] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[152] M. Raginsky, R. M. Willett, C. Horn, J. Silva, and R. F. Marcia, "Sequential anomaly detection in the presence of noise and limited feedback," *IEEE Transactions on Information Theory*, vol. 58, no. 8, pp. 5544–5562, 2012.

[153] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[154] M. D. McKay, R. J. Beckman, and W. J. Conover, "Comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[155] M. Malensek, W. Budgaga, S. Pallickara, N. Harvey, F. J. Breidt, and S. Pallickara, "Using distributed analytics to enable real-time exploration of discrete event simulations," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 49–58, IEEE Computer Society, 2014.

[156] W. Budgaga, M. Malensek, S. Pallickara, N. Harvey, F. J. Breidt, and S. Pallickara, "Predictive analytics using statistical, learning, and ensemble methods to support real-time exploration of discrete event simulations," *Future Generation Computer Systems*, vol. 56, pp. 360–374, 2016.