An Extensible, Scalable, and Continuously Adaptive Machine Learning Layer for the Internet-of-Things

Prahaladha Mallela

CMU-CS-19-106

April 2019

School of Computer Science

Computer Science Department

Carnegie Mellon University

Pittsburgh, PA

Thesis Committee:

Yuvraj Agarwal, Chair

Vyas Sekar

Submitted in partial fulfillment of the requirements

for the degree of Master of Science.

Keywords: Machine Learning, Continuous Learning, Internet of Things (IoT), Model Selection, Online Prediction Serving, Distributed Systems

Abstract

Internet of Things environments are widespread and often include different sources of sensor data, which can be used for machine learning applications. However, in IoT settings, the ambient environment itself is not static but changes over time, leading to variations in the sensor data and thereby decrease in accuracy for any ML application using that data. Furthermore, in complex environments, applications are exposed to various new conditions over time. Each IoT environment also has unique sensors and devices present. All of these special cases, which are symptomatic of IoT environments, make ML based applications very challenging.

This thesis presents an IoT centric, end-to-end Machine Learning Layer which addresses these challenges. Our ML Layer architecture enables each of the aspects (training and prediction serving) to provide feedback to each other leading to a continuous cycle. Our ML Layer includes a flexible model definition that allows us to incorporate any type of model or ML framework, and we initially implement several common models for frameworks such as Scikit Learn and Keras. Initially, the ML Layer optimizes models, and performs ensemble model selection, based on expressive policies. Over time, as a part of its autonomous feedback loop our ML Layer is able to automatically identify different patterns in environmental data, and continuously adapt these models based on this feedback solicited from users. In addition, our system performs dimensionality reduction using environmental data over longer periods to improve prediction efficiency. The system is designed to be general purpose to accommodate any type or combination of IoT data sources.

Our Machine Learning Layer for IoT is also a fully managed service, designed to be flexible and adaptive to facilitate ease of use and deployability. It can be deployed in a variety of settings (including smart homes and buildings) which require specialized learning on the spot to fit the environment and continuously improve accuracy.

Acknowledgements

I would like to thank my advisor Yuvraj Agarwal for his support. He has provided valuable guidance and suggestions throughout the entire process. Furthermore, he spent valuable time both in and outside our weekly meetings, which was very helpful for my project. I appreciate the opportunity to work with him very much.

I would also like to thank Vyas Sekar for agreeing to be the second faculty member for my thesis committee, reviewing my thesis document, and taking the time to come to my presentation.

Finally, I would also like to thank Peter Steenkiste and Tracy Farbacher for the help related to the program.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

In the last several years, the Internet of Things has become very widespread and popular. The idea of the Internet of Things is essentially to create a seamless network of digitally connected physical objects. These objects interact and share physically relevant information that can allow systems to perform intelligent actions for end users or provide them with useful, actionable information.

Figure 1.1: An illustrative diagram about the overall idea of smart cities, which encompasses many different aspects [image from [1]]

There are many different applications of IoT that have emerged for multiple scenarios such as residential, industrial, and retail use cases. There are estimated to be 20 billion connected IoT devices by 2020 [2]. There is also growing interest in creating larger scale deployments of IoT that expand to entire cities as presented in Figure 1. These smart cities aim to make public spaces everywhere connected and deliver new types of smart interactive experiences for people [3, 4].

Such expansion of IoT is leading to the creation of many different types of IoT environments. This includes smart homes, smart cars, smart public transportation, and much more, which contain a variety of sensor information. All of this presents a unique opportunity for the application for machine learning in this area. Machine Learning based methods can identify important occurrences, conditions, and events that arise in the environment. These can be used both to notify users as well as perform actions in the environment which can help users.

In this thesis, we focus on smart homes and smart buildings, though this work can also be applicable in other settings. Within the context of smart homes and buildings, Machine Learning is especially useful to help people in the context of their activities, by inferring the state and activities in the environment. However, there are many unique features of IoT that make the use cases for IoT different especially with respect to machine learning, as described below.

Unique Features and Challenges of IoT

Some of the unique challenges that arise in the IoT Machine Learning setting, especially for state and activity inferences in smart homes/spaces, include:

- *Using data from a unique set of different sensors in each environment*
  Each environment has a unique set of data including data from appliances, smartphones, and various individual sensing devices. These different sensors can capture different types of input, in different physical locations, which the system has to use.
- *Different types of sensor outputs*
  In different environments, even similar types of data (for example accelerometer data) can have different levels of quality and format so models for the data do not transfer.

As a result, the above implies that to fully leverage the data in each environment for best performance, the models that are used to make predictions for the end user task should be completely learned and optimized from all the available custom environmental data, rather than depending only on a particular type of data, or on prior/global information which may not be applicable.

Challenges in the IoT setting further include:

- *Not much initial data*
  In each unique environment, initially there will not be any training data, or only a limited amount of training data.
- *Data addition over time*
  Since the data comes from the physical environment, the system is continually exposed over time to all the possible data variations, and has to keep learning over time.
- *Dynamic environments*
  Temporary or permanent changes in the environment can occur at any time, and fast adaptation is required based on these changes.

As a result, the above implies that any system for the smart space IoT setting has to be able to get more data that will help it learn in the new environment, and should continuously adapt over time as it gains more experience in the environment and is exposed to new conditions, in order to perform well and be useful to end users.

In this thesis, a novel Machine Learning Layer is presented, which is designed for the IoT setting and which addresses the unique challenges that arise in smart space IoT use cases as previously described.

1.2 Background

The two fundamental tasks in using machine learning are **training** and **serving**. Training refers to the process of building up models based on a set of example data. Serving is the process of using those trained models to respond to live requests, and make predictions.

Each machine learning model has two types of parameters, which are *learnable* parameters and *hyperparameters*. Learnable parameters are optimized during the training process based on the data. In contrast, hyperparameters are set before the training process.

Machine Learning tasks are, in general, divided into several different paradigms. Supervised learning involves developing models that learn from a set of samples which are labeled. Unsupervised learning learns from unlabeled data. Here we use a combination of both supervised and unsupervised techniques.

Background for different techniques relevant to the aspects used in the Machine Learning Layer will be presented in each of the sections.

1.3 Related Work

The Machine Learning Layer for IoT that is presented in this thesis is a complete machine learning system that can work with any IoT application. Below, several different types of commonly used Machine Learning systems and IoT systems are outlined, along with a brief description of how the Machine Learning Layer is different.

1. **Offline Large Scale Training Systems**

   There are many different large scale training systems in use. They are designed to optimize for very expensive models such as deep neural networks, with many hyperparameters, and for which each training iteration itself is a very long process. As such they not only use hyperparameter tuning techniques, but also techniques to monitor progress over time during training which can take many hours.

   Once such example is Google Vizier [5], which is a framework that is designed for the optimization of deep neural networks for internal Google projects. It is designed as a scalable framework for datacenters to optimize the tuning of networks which can take many hours or days to train. For this use case, it employs a variety of specialized techniques and algorithms. Google Cloud Hyperparameter Tuning [6] is an external service allows for hyperparameter tuning of such models based on Google Vizier.

   Yet other systems such as Google AutoML [7] use completely different techniques to build high accuracy deep neural network models for complicated problems (for example in

computer vision). They view the problem of learning to build a network architecture itself as machine learning problem. They apply a variety of techniques such as reinforcement learning to learn how to create networks. This domain involves very large amounts of data. These systems are designed to put a lot of effort into producing very high-quality and complicated deep networks for domains such as object recognition, and translation.

2. Online Serving Systems

Another set of systems focuses on serving of models. Many such systems have been created and studied, as presented in [8]. They use static models which are pre-trained and manually packaged to be ready for serving. The models do not change during the course of serving. Within this there are specialized model serving systems, for example LASER [9]. They focus on optimizing model serving performance for a particular domain like advertising and are tied to particular, restricted model types.

There are also more general serving systems. One recent example is Clipper [10], which is a serving system that optimizes performance using different strategies such as batching, and caching, and scaling to ensure that performance demands can be met even at large data center scales. While Clipper does some weighting of the models, it treats the models as black-boxes.

3. IoT and Big Data Solutions

There are many different Cloud platforms that are commonly used today. Most of these large scale Cloud platforms also support IoT, such as Google Cloud [11], Microsoft Azure [12], Amazon AWS IoT [13], etc. These IoT platforms are mainly targeted towards analytics, large scale operations management for business and industrial applications, etc. Though some of them do support smart home automation, they usually focus on rule-based systems, and not tasks like custom localized activity recognition model development.

4. Application Specific IoT Systems

There are many application specific systems that are in use for IoT. These systems are characterized by a tight coupling to a specific type of data. They cannot operate with other data sources, and often use pretrained models using offline data. One such example is

Ubicoustics [14], which is a state-of-the-art deep network for sound classification, which relies on huge amounts of training data and is completely pretrained. There are also many different systems targeted towards their own sensor platforms for example the MITES sensor [15], which uses 13 different hardware sensors, and couples that with a ML system to do inferences. In this thesis, we present quantitative and qualitative comparisons in both of these domains.

Limitations of Existing Systems

Most types of current machine learning systems have assumptions and goals that do not align fully with this setting of smart space IoT.

For example, the large scale training systems assume the availability of large sets of training data for very expensive one-time training tasks. In contrast, the serving systems assume that very high accuracy models are already available and do not need to adapt, which is an incorrect assumption. The wide range of platform and sensor specific systems are tightly coupled to particular tasks and uses. This makes them inflexible and very limited, since in real IoT deployments those specific data sources may not be present, or there may be many additional data sources available which are very valuable are crucial to good performance. Additionally, such systems are static in models they use, and rely on pretraining and out of context data to choose the best models. Most of these systems require a large amount of manual work to refresh the models.

Our ML Layer Differentiation

In contrast, our Machine Learning Layer for IoT is a general and flexible system that can work with any combination of data sources and produce models optimized for both the available data and the end-user task. The ML Layer aims to build up a high accuracy system from scratch, and one that adapts over time in any new and unique environment. The ML Layer is a unique end-to-end system that combines the training and serving aspects in an autonomous loop which can significantly improve the performance over time. The various components, architecture, and techniques that were developed and put together to create this system are designed to enable continuous improvement of the ML models over time.

1.4 Outline

The rest of this thesis is organized as follows:

- Chapter 2 provides the overall structure of our end-to-end Machine Learning Layer. The additional details of the various subcomponents and methods used in the Machine Learning Layer are described in the following chapters.
- Chapter 3 focuses on the components that enable the continuous improvement of the Machine Learning Layer over time with additional data
- Chapter 4 describes the integration of dimensionality reduction techniques into the Machine Learning Layer to further increase the efficiency of the serving module
- Chapter 5 provides details for the ML Layer Management System which allows many different applications to use the Machine Learning Layer for different tasks, and also allows deployment across different physical machines, in a distributed setting
- Chapter 6 concludes this thesis, providing some avenues for future research

1.5 Unique Features and Contributions

This thesis has several different unique features and contributions as listed below.

Overall Idea and Framework:

Firstly, this thesis introduces the overall concept of a Machine Learning Layer for IoT, with the goal of adapting to different sorts of data and environments, and specifically introduces a:

- System design that handles model training and prediction serving as independently functioning components that continuously communicate with each other, and the concept of a continuously improving system using an autonomous feedback loop, producing optimized models over time for the specific environment in question;
- An extensible design that is flexible enough to support different types of Machine Learning frameworks, different types of data, and different training and serving deployment settings

Algorithms and Components:

This thesis introduces algorithms and components and applies existing methods, across the Machine Learning Layer to enable a dynamic and adaptive system. Specifically, it

- Presents an algorithm for timely automated feedback which allows our system to learn from new data, by automatically suggesting the appropriate feedback without supervision. This enables dynamic improvement of ML models in different environments over time;
- Integrates components for model ensemble selection, and continuous model updates to improve accuracy over time based on that feedback;
- Introduces an algorithm for integrating and optimizing dimensionality reduction with model selection to make the system scalable, and the models more efficient over time, by reducing the computational requirements of prediction, while maintaining accuracy.

Implementation and Evaluation of the End-To-End System:

The thesis work not only presents the framework and algorithms, but also the complete system implementation incorporating the various aspects into an end-to-end system:

- A training and serving system, along with a set of expressive APIs that allows a user or a developer to use it with a variety of IoT sensor data
- A distributed management system that enables easy deployment of the training and serving components on multiple compute nodes, supporting and managing the prediction tasks for different applications, and enabling horizontal scalability
- Evaluation of the improvement of the accuracy of the system over time, along with efficiency making use of Smart Home Audio data scenarios, and the MITES data

Chapter 2

Machine Learning Layer Training Serving System Design

2.1 Introduction

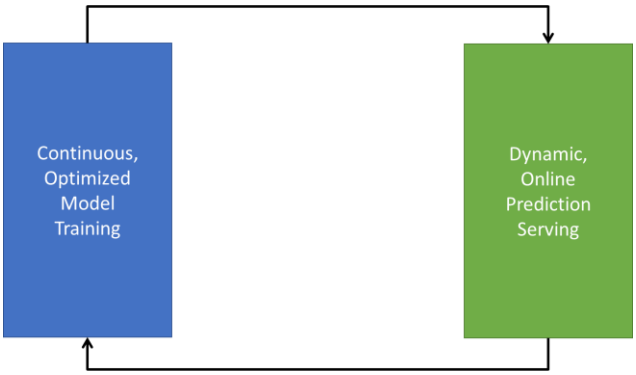

Figure 2.1: A representation of the continuous feedback loop, with the training and serving components providing feedback to each other.

The core of our Machine Learning Layer is the training and serving components. The architecture features an independent Training Module and Serving Module which work together in a closed loop, which continues to enhance the performance of the system over a period of time.

The ML layer is not tightly coupled to a particular application or a specific type of data unlike the wide range of application specific IoT systems as mentioned before including [14, 15]. Instead it can work with any kind of IoT application and be used with a variety of environmental data. The

system aims to create a powerful plug and play interface and continuously improve performance through an autonomous feedback loop wherever it is deployed. As a result, it can deliver much better user experience, for applications like context recognition by being more accurate and supporting improvement over time.

The interfaces to the ML Layer are designed to be easy to use by any client application and general enough to train on different sensor data, request predictions, and provide feedback.

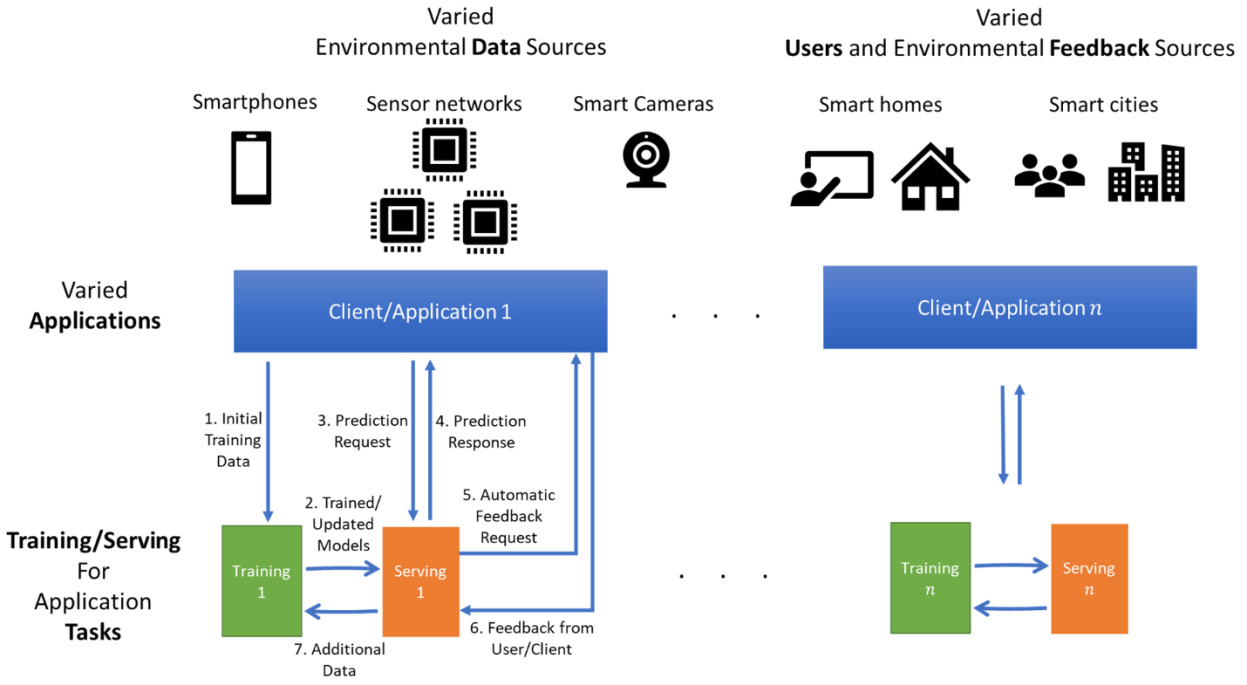

Figure 2.2: Each client application can use the training and serving system for its own tasks. Initially the client sends a set of training data to the Training Module (step 1). Based on this, models are trained and sent to the Serving Module (step 2). The client can continuously request predictions based on the environmental data and the Serving Module responds with the predictions (steps 3, 4). The system can generate automatic feedback requests (step 5), and the user/client can provide feedback (step 6). The additional data collected in the Serving Module is sent to the Training Module (step 7), which updates the models (step 2).

The system design is flexible so that the Training and Serving can be deployed in different locations, not necessarily on the same machine. The system is also very flexible as it can support many types of models. It features a model definition that can be used to implement models in a

variety of frameworks, so that they can be used seamlessly inside the Machine Learning Layer. Furthermore, it can support different prediction tasks including binary classification and multiclass classification tasks defined by the client. Though the focus is on classification, the framework can be extended to handle tasks like regression as well.

2.2 System Design and Interface

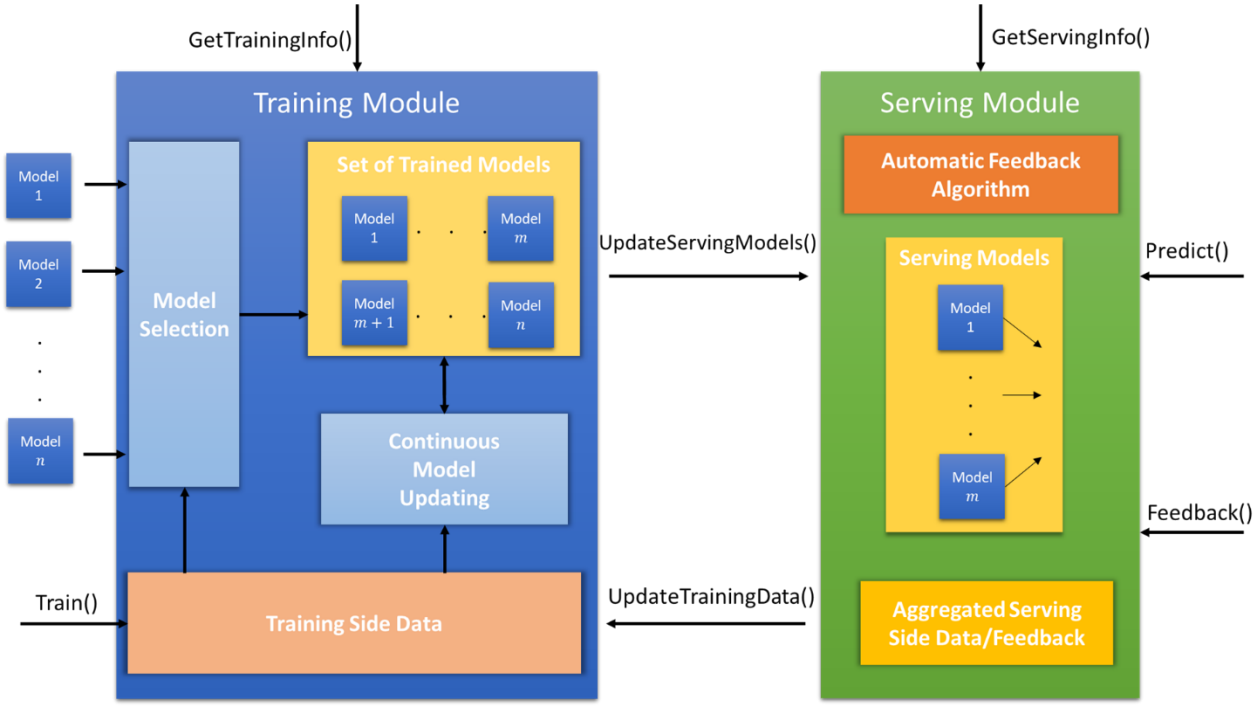

Figure 2.3: This figure illustrates the ML Layer Training and Serving Modules, along with the interface functions, and how the modules interact with each other. Note that the Serving Module uses a subset of *m* of the models from the *n* Training Module models.

The Machine Learning Layer Training and Serving System interface supports the following functions:

- *Train()* which allows the client to provide the initial training data, so that the ML Layer can train and tune all the models we support, before serving begins.

- *Predict()* which allows the client to provide input data, and get a prediction from the Serving Module.
- *Feedback()* which allows the client to provide feedback to improve the system. This feedback for example specifies whether the prediction was correct or not.
- *UpdateServingModels()* which is used to transfer the best models from the Training Module in a standardized format, for the Serving Module to use subsequently.
- *UpdateTrainingData()* which is used to transfer the aggregated environmental feedback and data from the Serving Module to the Training Module so that the models can be retrained with the new data.
- *GetTrainingInfo()* which allows the client to access the relevant state and information in the Training Module like the set of trained models, parameters, etc.
- *GetServingInfo()* which allows the client to access information about the state in the Serving Module like the serving ensemble and the model weights.

2.3 Training Module

The Training Module handles all the training and updating of models. Initially it performs model selection, and then over time it continuously updates the models as more data is added.

Model Selection

To start, the Training Module is initialized with set of MLLayer models that are untrained, which satisfy the interface described in Section 2.5.

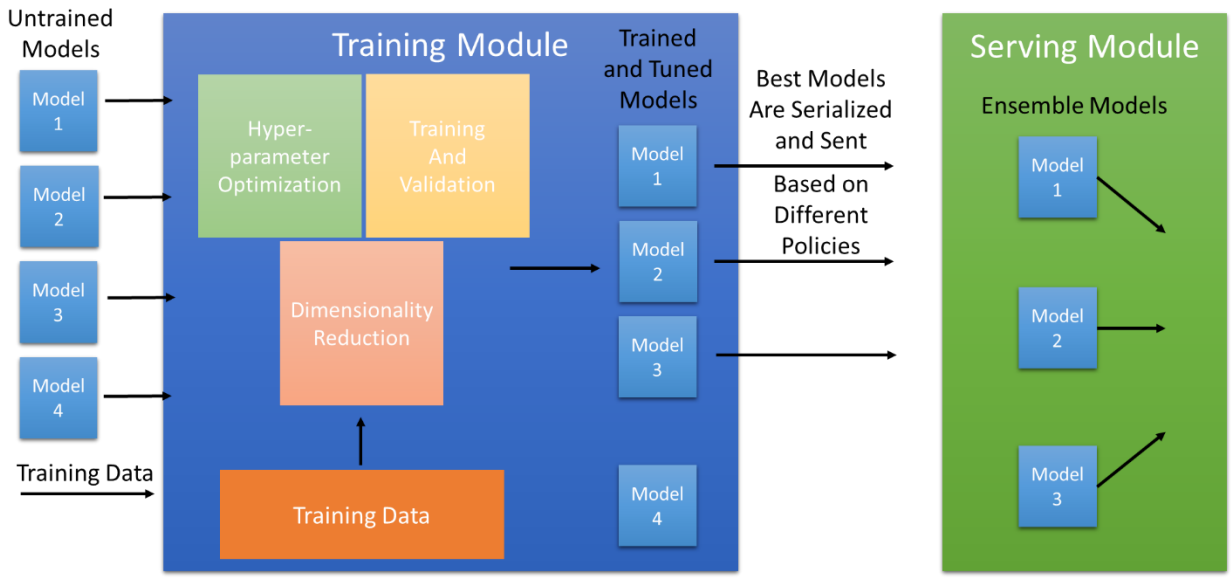

Figure 2.4: This diagram shows the process of model selection in the Training Module. All the models and hyperparameters are trained and optimized based on the input training data, in parallel. Based on different policies, an ensemble of models is selected and sent to the Serving Module.

All the models are trained using the training data, and during this the hyperparameters are optimized based on cross validation. So, for each model, the hyperparameters that work best (empirically) are found along with their respective validation accuracies. The models are then trained with these hyperparameters on the entire training dataset. The system does this in parallel for each model for efficiency.

After all the models are optimized, the best models on the training side are chosen to create a model ensemble and serialized according to the format specified in the model definition which is different for each model. These models are then sent to the Serving Module.

The Training Module allows for different model selection policies. These policies allow for different ensembles to be selected and are customizable based on the accuracy and latencies of the models. For example, we have implemented a policy that choses the top 3 models based on accuracy. Another example is a policy that choses the top 3 models based on lowest latency.

Dimensionality reduction is also incorporated into model selection using an algorithm described in Chapter 4, but only after the system has enough data from the environment. Note that dimensionality reduction is done to reduce model complexity, to improve prediction performance.

Overall, the Training Module maintains the training data, along with the trained models, and the outputs of the model selection like the validation accuracies, etc. so that it can use this later when updating the models.

So far, we have described the model selection process. The Training Module also continuously updates models in order to support learning based on automatic feedback. This is described in more detail in Chapter 3.

2.4 Serving Module

The Serving Module performs online predictions, using the models that it receives from the Training Module. It also provides a certainty estimate for each prediction based on the models that are included in the ensemble. The Serving Module features components for automatic feedback that enable the continuous learning.

Ensemble Predictions

Initially after the model selection, the Serving Module gets the models in a serialized way from the Training Module. The Serving Module then continuously responds to prediction requests and performs weighted ensemble prediction with a set of weights that are determined based on the validation accuracies, as explained below:

Let the best models that are sent to the serving layer be $m_1, m_2, \ldots, m_n$

The ensemble prediction for classification tasks is

$$prediction = \underset{c}{\operatorname{argmax}} \left(\sum_{i=1}^{n} w_i \ (if\ f_{m_i}(x) = c) \right)$$

Where $f_{m_i}(x)$ is the prediction that model m_i makes for input x, and w_i is the weight for model m_i.

For regression tasks, this can be modified to be

$$prediction = \sum_{i=1}^{n} w_i f_{m_i}(x)$$

The static weights for the models $m_1, m_2, \ldots, m_n$ are $w_1, w_2, \ldots, w_n$ are given by

$$w_i = e^{x_i} \left(\sum_{j=1}^{n} e^{x_j} \right)^{-1}$$

Where $x_1, x_2, \ldots, x_n$ are the validation accuracies for models $m_1, m_2, \ldots, m_n$.

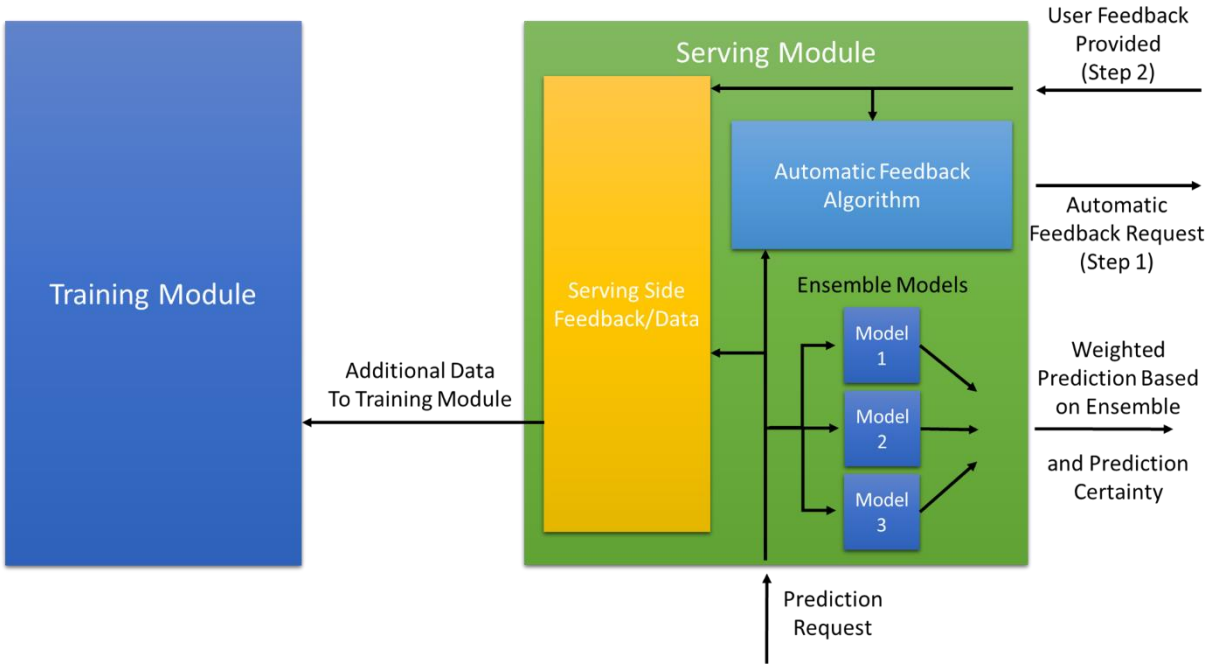

Figure 2.5: This diagram illustrates the functions within the Serving Module, which include ensemble model predictions, prediction certainty estimation, automatic feedback requests, and aggregation serving side data and feedback.

Prediction Certainty Estimation

The Serving Module also provides estimated certainty for each prediction. In IoT applications, there are several advantages to providing certainty estimation:

- In many IoT settings, ML based predictions are used to take actions that affect the physical environment. When the certainty is low, these can be avoided.
- By using certainty estimation, notifications to the user can be prioritized.
- It also allows more visibility for the end user into how certain the ML system is for that prediction

There are multiple ways to get an ensemble based certainty estimate.

1. The first method to get an ensemble certainty is to just take the different predictions together with the model weights. This method does not rely on the models being able to estimate any class based certainties so it is the most general method. It works as follows:
   Let the possible classes be c_1, c_2, …, c_m, and $certainties$ be the vector estimation of class certainties.

$$certainties = \left(\sum_{i=1}^{n} w_i \left(if\ f_{m_i}(x) = c_1 \right), \sum_{i=1}^{n} w_i \left(if\ f_{m_i}(x) = c_2 \right), \ldots, \sum_{i=1}^{n} w_i \left(if\ f_{m_i}(x) = c_m \right) \right)$$

2. There are classifiers that can provide some kind of estimation of certainty over a distribution of the classes. The second method combines this into the certainty estimation by taking the weighted average of the probability in each class from each model.

$$certainities = \left(\sum_{i=1}^{n} w_i f_{m_i,c_1}(x), \sum_{i=1}^{n} w_i f_{m_i,c_2}(x), \ldots, \sum_{i=1}^{n} w_i f_{m_i,c_m}(x) \right)$$

   Where $f_{m_i,c_j}(x)$ is the probability that model m_i predicts for the class c_j on the input x.

To be able to support different types of classifiers the first strategy is used, which just uses the predictions of the models to output class based certainties. This has the advantage that no change to any of the models is required. Also, many commonly used models, e.g. nearest neighbors, do not support probability estimation, so the first method is more general purpose.

In either method, the output $certainties$ is still a vector with a value for each class. There are many ways that a certainty estimate could be generated from the vector. A simple method is to take the maximum value of the vector:

$$certainty = \max_i \ certainities_i$$

This can range from a minimum value of $1/n$ to a maximum value of 1.

Another approach is to take the magnitude or the squared magnitude:

$$certainty = \|certainties\|^2$$

This can also range from a minimum value of $1/n$ to a maximum value of 1.

The second method with the magnitude uses the entire vector rather than a single value, so we use that.

Feedback and Updating

The Serving Module also enables continuous learning using automatic feedback. We describe this in more detail in Chapter 3.

2.5 Machine Learning Layer Model Definition

Design goals: Our Machine Learning Layer supports using various common models from different frameworks, such as the SVM model from Scikit Learn and LR from Weka. There are various advantages to this approach:

- It allows our system to leverage the capabilities of any framework which may include faster training, computationally efficient data transformations, etc. It also allows for deployment in a wider range of settings, for example on different hardware which require custom optimized frameworks;
- It allows the system to be independent of any particular framework which enables it to be used in the same way as frameworks and models change;

However, our machine learning layer does not treat the models as black-boxes, so we need to balance uniformity across models along with customizability for each model. The Machine Learning Layer was designed to have a flexible ML Layer Model Definition so that it can support any types of models and frameworks. The interface was designed to have the following properties:

- To allow different types of models to be used with an interface that exposes as many details as are necessary for both training and serving tasks;
- To allow the ML Layer to interact with a common abstract interface, while allowing individual models to maintain their expressiveness and allowing flexibility for model and framework specific methods to be implemented;

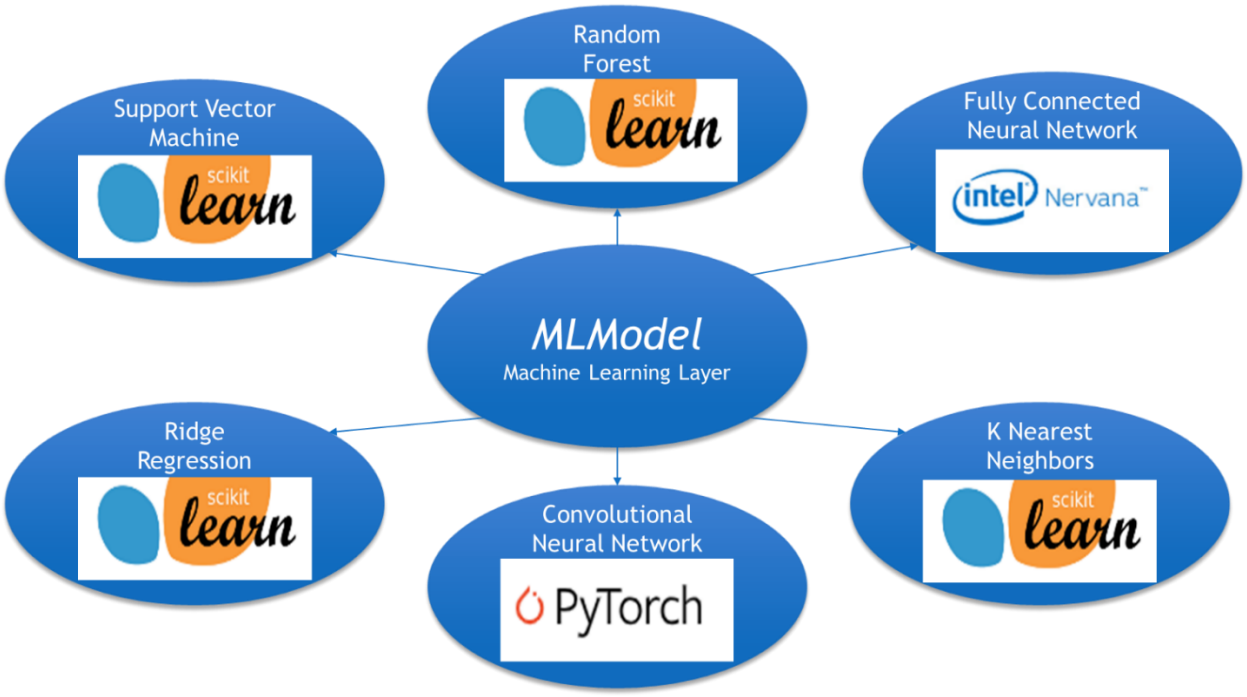

Figure 2.6: The Machine Learning Layer Model definition allows different models to be supported using different frameworks. All machine learning layer models are derivatives of the base class MLModel.

Each model is a *subclass of the base class MLModel* as shown in Figure 2.6. The functions that are part of the ML Layer Model Definition, which are *implemented separately for each MLModel subclass*, include:

- **Train():** Allows for the definition of the model specific training algorithm that is to be used in the ML Layer.
- **Validate():** Allows the use of custom validation methods that are appropriate for the model, as different methods are useful for different models, and that are also optimized for the framework.
- **Tune():** Allows for the training and initial tuning of the hyperparameters using methods that are specific to the model and optimized for the framework.
- **Set_params():** Sets the specific values of the parameters to the model. The parameters are consistent with the definition in the parameters function.
- **Parameters():** Allows for the definition and specification of the set of custom model parameters for tuning along with their ranges and can specify different types of parameters such as integer or real. It returns the list of parameter ranges.
- **Predict():** The predict function takes the input data and returns a prediction from the model.
- **Copy():** Does a complete deep copy of the model and returns it, which is used as needed when either the Training or Serving Module needs to create separate copies of the models
- **Store():** Stores the model as a string that is completely portable and serializable. It also returns the name of the class, which is a subclass of the MLModel class. This is used as needed, including when the models are sent from the Training Module to the Serving Module
- **Load():** Loads the model from the model string. This is used as needed, including when the models are sent from the Training Module to the Serving Module

Chapter 3

Continuous Improvement of the ML Layer

3.1 Overview

The previous section presented an overview of the components of our Machine Learning Layer. This next section describes the components related to continuous improvement of the system over time, in further detail.

There are different aspects that work together to enable the enhancement of our system over time with more data, including automatic feedback solicitation and continuous model improvement. Together, they enable an autonomous feedback loop that allows the system to intelligently learn in the environment over time.

The automatic feedback functionality also makes our system much more robust to change. It allows our system to identify differences in the data distribution and focus on those differences even without manual guidance from the client or application. By using focused feedback, our Machine Learning Layer adapts fast and improves its accuracy on previously unseen data.

By automatically identifying groups of data, our system can also gather more feedback with a smaller number of label requests to the user, improving usability. However, to make use of a continuous stream of data, the models have to be updated. Our Machine Learning Layer training and serving modules interact to keep improving the models. Since the updates to the models are continuous, the system does not have to perform full static model selection on every update, and instead uses a less computationally expensive model update method.

3.2 Introduction and Requirements for Automatic Feedback

This section introduces the overall setting for soliciting automated feedback from users, before we present the algorithm in Section 3.5. There is a constant stream of feature vectors of incoming data, $x_1, x_2, x_3, \ldots$, that is continuously being provided to the Machine Learning Layer. For any given x_n, the algorithm can predict and output a set of points which are likely to be associated with x_n. There are several key properties that the algorithm has to satisfy, specifically:

1. The algorithm should be able to identify a group of multiple instances of data at once. For example, identifying a group of 4 datapoints at once gives more examples than only identifying 2 datapoints at once.
2. Each set of data points that the algorithm outputs should belong to the same class as much as possible. This means the algorithm is accurate.
3. The algorithm has to run online, so the algorithm should be efficient to run interactively as new data is acquired.
4. The feedback that the algorithm identifies should not be limited to a particular area of the data space, since the feedback is used by the system to learn over time.
5. The algorithm should avoid repeatedly identifying sets of points that are close to already labeled feedback points.

There were many different approaches which we experimented with. The first type involves clustering algorithms, including those like DBSCAN [16], Affinity propagation [17], Mean-shift [18], etc. These approaches have several limitations. First, they are iterative and too slow to be used effectively in the online setting especially with lots of data. Also, they are unstable with respect to additional data. Since they depend on having a full view of all the data, even small additions of data can significantly change the output. As a result, they are not suitable for this continuous IoT data setting. Also, approaches that use small windows of data lose the global view these methods need for identifying different clusters. We also tried using dimensionality reduction and then performing clustering or other techniques on the reduced dimensionality data, but this has similar problems of being slow and unstable online.

For the online setting, the algorithm has to be fast enough to respond in real time to be effective for feedback, in addition to satisfying all the other requirements that were outlined above.

3.3 Background for Hyperparameter Optimization

Each model has hyperparameters which need to be tuned based on the data the system receives to get proper results. Since the space of hyperparameters for each model is a large space that is often continuous, it cannot be all enumerated. There are several different methods for hyperparameter tuning, which are described here.

Grid Search

Grid Search is a deterministic search algorithm [19]. For each parameter x_i, there is a set of hyperparameter values s_i. Then the search space is the product of all the individual sets $s_1 \times s_2 \times s_3 \times ... \times s_n$ which grows exponentially with parameters. The advantage of grid search is that it allows targeting the search towards important values, but the main disadvantage is that it cannot exhaustively search the entire space which may contain much better values.

Random Search

Another commonly used method for hyperparameter tuning is random search [19]. This is a very simple method which just randomly samples the overall hyperparameter space. This method has advantages over other methods as its complexity does not increase with the number of dimensions of the set of hyperparameters. It has also been shown to perform well for hyperparameter tuning given enough iterations [19]. However, the disadvantages of this method include lack of use of previous estimates, and also a lack of targeting towards important values of hyperparameters.

Evolutionary Methods

While grid search and random search are useful techniques, there are other methods that use previous history which can improve efficiency. Evolutionary methods such as Differential Evolution [20] are iterative methods that keep a set of possible parameters at each stage, and then perform modifications to this current best set at each step.

Bayesian Optimization

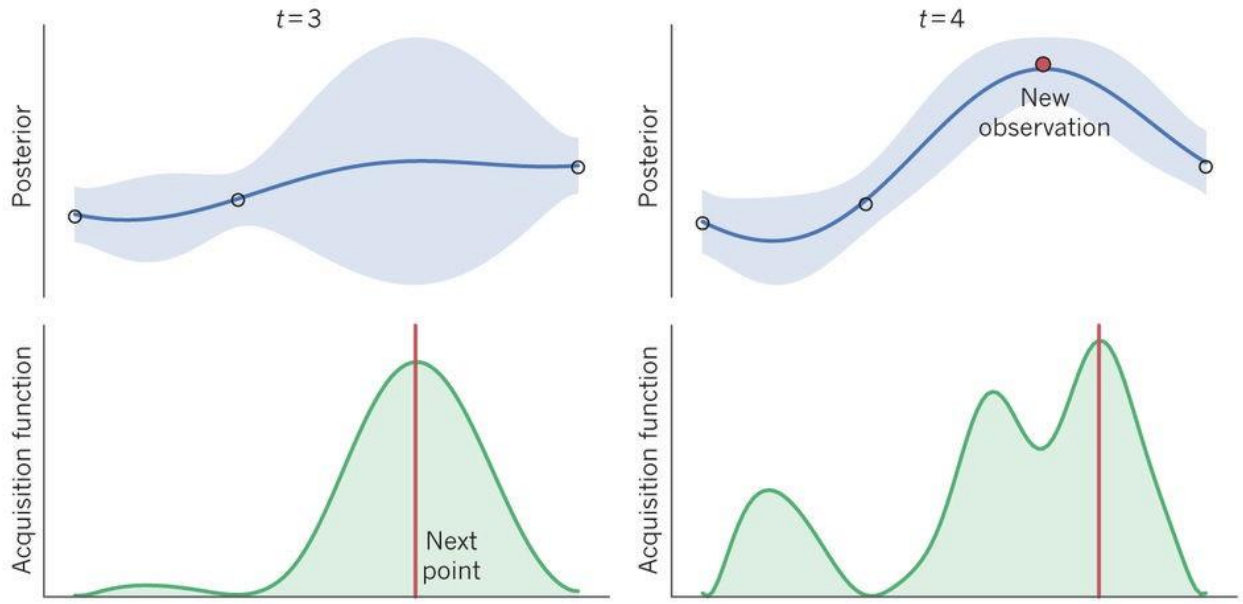

Figure 3.1: This shows an abstract example of how Bayesian optimization choses the next point to try while optimizing the objective function [image from [21]]. The posterior distribution is the internal model of the objective function based on the previous evaluations. The acquisition function is the uncertainty of the internal model, based on which the next point is chosen by taking the maximum.

Bayesian Optimization [22] is a completely black-box optimization technique that has been effectively used to optimize high dimensional problems. Bayesian Optimization is meant to be sample efficient and fully uses information gathered from all the previous evaluations. It is a much more advanced technique than the other methods described earlier, and operates by creating a separate model of the objective function, which is the function being optimized, from the evaluations. Every iteration, it choses the point to evaluate based on some metric based on this internal model, such as the uncertainty or the information gain. This process is shown in Figure 3.1 on abstract example, which shows how the Bayesian optimization creates an internal model (posterior), and choses the next point with the maximum acquisition function value (in the example, this is the uncertainty of the internal model).

Within the Bayesian Optimization itself, there are a variety of methods that can be used. Many of these methods vary in the estimation of the internal model. Examples include:

1. **Gaussian Process Based Bayesian Optimization**
   The first method uses gaussian processes to create an estimation of the objective function based on the previously evaluated points. A Gaussian Process estimates a multivariate Gaussian distribution for the function values at each point, which indicates the current expected value of that point along with the uncertainty.
2. **Gradient Boosted Regression Tree Based Bayesian Optimization**
   The second method uses gradient boosted trees to create an estimation of the objective function based on the previously evaluated points.

In our actual implementation, we found gradient boosted tree optimization to be faster per iteration in terms of execution time, and was preferred.

3.4 Algorithm for Automatic Classification and Feedback

We now describe the actual algorithm for automatic feedback, designed to be responsive enough for the online setting, while providing good performance for the different criteria outlined before.

Firstly, the algorithm includes a data transformation component. Each time a new data example is added, it is transformed using a function $f(x)$ that can be any transformation including a complex neural network embedding which is appropriate for the data. This depends on the data and can also be the identity transformation.

The next important component of the algorithm is the distance measure that is used between points in this transformed space. This measure can be any metric including cosine, correlation, etc. This distance metric need not adhere to the triangle inequality (unlike the standard Euclidean metric) so the algorithm does not make incorrect optimizations based on this assumption.

Within this transformed space, and under the particular distance metric, the algorithm aims to find large enough subsets of points that are all close to each other with respect to that distance. The algorithm maintains a graph that contains information about close points. When a new data point is added, that point is transformed, and the distance from that point to all the other points is computed using a vectorized parallel distance computation operation, which is linear time complexity. This is very important since the number of datapoints increases dramatically over

time. Then, the graph is updated by adding this new vertex v, and edges to the all the points that are within a distance d.

The next step is to be able to determine if there exists a subset of the points that are all within a radius of distance d, include the newly added point, of size greater than n. This allows the algorithm to identify points which are likely to belong to the same class. Doing an exhaustive search for this can be inefficient, so our algorithm does a more efficient search by looking at the vertices which are within a distance of 2 neighbors from the current vertex. For each neighbor v_i of vertex v, the size of the set of neighbors of v_i is checked and the maximum such set is kept. If a particular neighbor v_i is marked, then the current vertex v is marked and the search is stopped.

After the search is completed if the maximum sized set exceeds n then the group is identified and output. To maintain this graph efficiently both in terms of time and space, the adjacently list representation is used which only stores the vertices that are within the distance d. This makes searching neighbors efficient and minimizes the space that is used for the graph. This space efficiency is especially important as over time the number of datapoints increases.

When feedback is provided, for a group of datapoints, those vertices are marked along with all the vertices in the neighborhood of that subgraph.

Note, the various parameters we use have tradeoffs, and are empirically chosen, depending on the type of data. The parameter d controls the distance between the points. The number of neighbors in the graph is dependent on the distance. Intuitively, the distance should be as small as possible to ensure the accuracy of the algorithm. However, d should be large enough so that similar points can actually be identified as belonging to the same class. Similarly, n should not be too large, since the algorithm will not identify opportunities for soliciting feedback, to improve accuracy.

Our algorithm also avoids re-identifying points that are close to points that have already been identified before by automatic feedback. This has an additional benefit, since it allows the algorithm to continue focusing on and covering different areas of the data with less feedback. It also prevents the end users from repeatedly being asked about very similar feedback, improving usability.

3.5 Approaches for Model Updating

As a part of the continuous feedback loop in the Machine Learning Layer, the feedback obtained actually has to be utilized, so the models also have to be continuously updated. For our IoT use case, we focus on continuous improvement, so each iteration is not independent of another. Here, more details are presented about the approaches used for continuous model updates. We present the design space of potential solutions we explored, before we describe the approach we chose since it is more appropriate for the continuous setting.

Preliminary Solutions

One solution is to simply train and tune all the models initially, and choose the ensemble of the best models at the start. As more data is added, only retraining occurs without changing the model ensemble at all. This approach is efficient, but is limited by the parameters based entirely on initial data, which in many practical scenarios is limited.

Another solution is to completely retrain and retune all the models every time there is additional data, and choose an ensemble of the best models. Unlike the previous approach, this approach does not depend on the selection of the best hyperparameters in the beginning itself. It allows the model hyperparameters to change over time, along with the model ensemble. However, it is not suited for this use case as rerunning model selection, along with hyperparameter exploration, every time is expensive and does not leverage the continuous nature of the optimization.

Hybrid Approach: Incremental Retuning with Bayesian Optimization

Instead we chose a hybrid approach to enable model updating, which avoids the unnecessary evaluations over full repeated model selection.

Initially model selection is performed by completely training and tuning all the models.

Then models are continuously updated using different levels of retraining

1. The models that are in the current best ensemble are retrained with the new data
2. The models that are part of the current best ensemble are retuned with the new data, using the previously identified best hyperparameters. To do this retuning, for each

model in the current ensemble, Bayesian Optimization is run using the previously identified best hyperparameters for that model as starting points for the optimization.

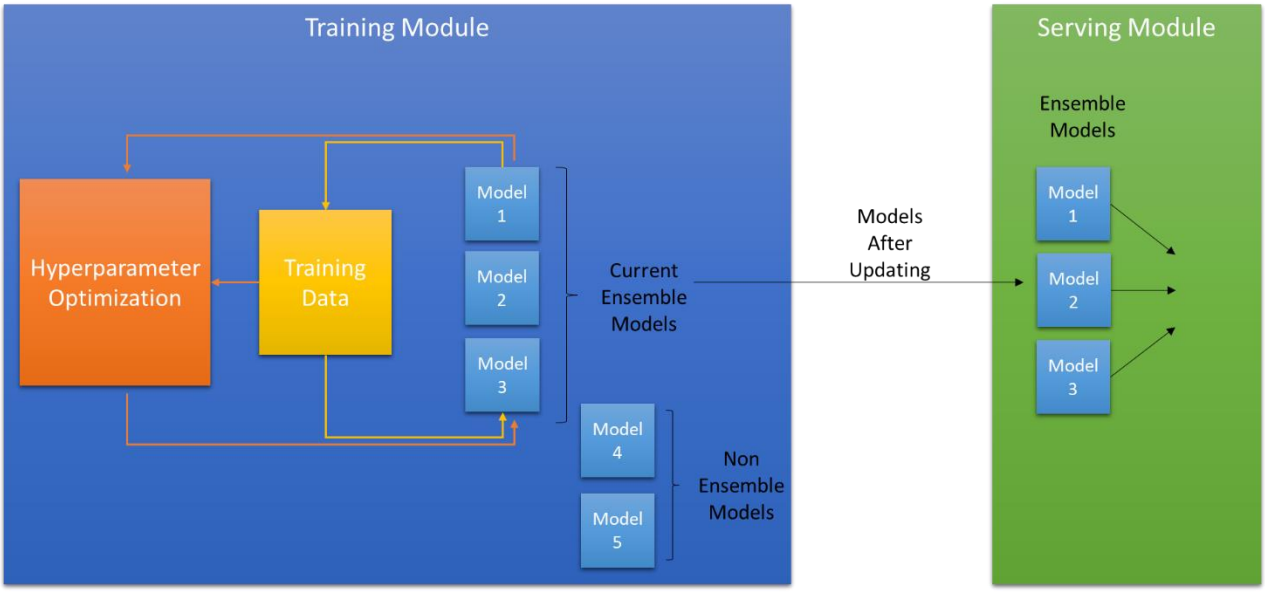

Figure 3.2: A representation of the model updating in the ML Layer.

3.6 Results

In this section, we evaluate different metrics, including the performance of the automatic feedback algorithm, along with results for the improvement in accuracy of the system over time, as more data is added.

3.6.1 Introduction to Datasets/Comparisons

Smart Home Audio Dataset

The primary dataset used in this section is a high-resolution audio dataset that we manually collected. The audio data consists of a variety of sounds that are representative of those found in a smart home, in different scenarios (longer term and shorter term) which are described in the following sections. We collected audio data sampled at 16000 Hz to capture a wide range of audio.

The data input to the system is in the form of frequency spectrograms. Each input spectrogram corresponds to 0.96 seconds of audio, and the complete format is described in [24].

Pretrained Audio Based Classification System

Specifically for the audio based context recognition scenario, we compare accuracy with a state-of the art pre-trained system based on a deep neural network model [14]. This deep neural network model is a convolution neural network that has been trained on very large amounts of environmental audio data. It has been shown to achieve high accuracy on different environmental sound datasets. The Ubicoustics model takes the same spectrogram input and can predict the class of the sound [14], from a total of 30 sound classes and the system also uses a threshold to predict when there is no sound class.

3.6.2 Smart Home Audio Data: Long Term Use Case

For this use case, we use a smart home scenario in a kitchen. The prediction task is a multiclass classification task to identify various activities. The data is a representative sample of many different kitchen sounds including a blender, running microwave sounds, cell phone ringing, cutting or chopping, water faucet running, and the pantry door. It includes background data that was captured with lots of noises. It also includes a couple of additional classes including typing and knocking. Overall our scenario has 9 total classes.

The data transformation we use in the automatic feedback algorithm is a network model described in [24]. This is a convolutional neural network, called vggish, which produces a 128 dimensional embedding of the input. Through lots of experimentation, this was found to be better than using the raw data or using other transformations such as the spectrograms directly.

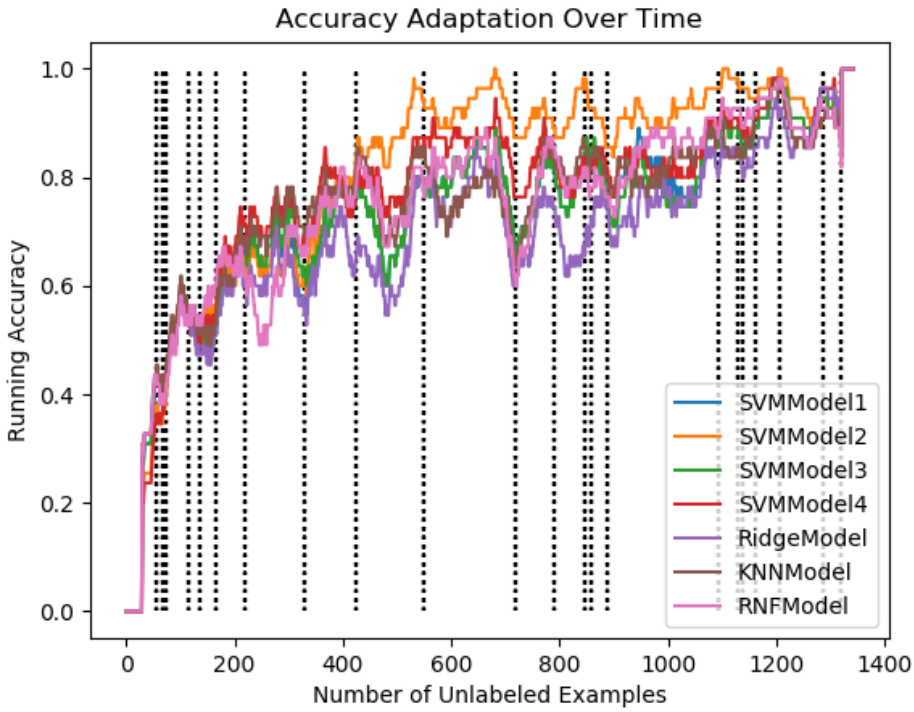

Figure 3.3: The running accuracy improvement for the Long Term Smart Home Audio Use Case using the automatic feedback functionality. The graph shows how the system continuously improves over time. Each vertical line indicates when the system automatically asks the user for feedback. The system is able to dynamically adapt to events in the environment.

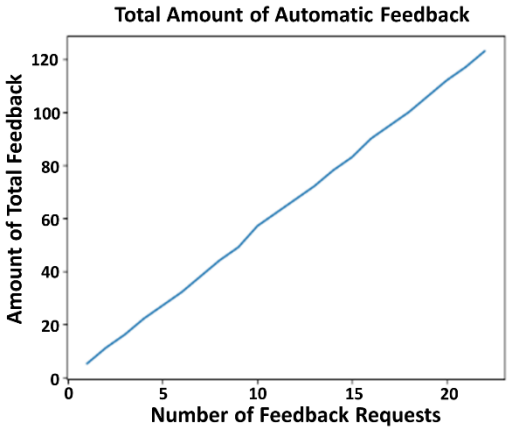

Figure 3.4: The total amount of automatic feedback for the Long Term Smart Home Audio Use Case as a function of the number of feedback request iterations. The system results in each iteration of feedback giving a 6x increase in the number of labeled examples which can be used by the system to improve the models.

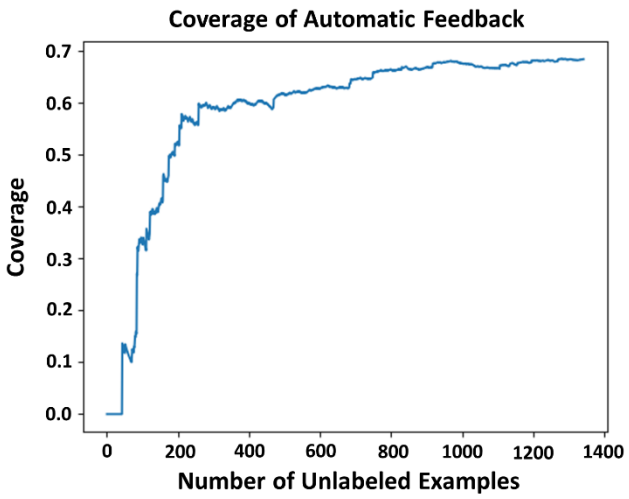

Figure 3.5: The amount of data that is covered by the automatic feedback for the Long Term Smart Home Audio Use Case. The coverage is over 60% while using only 20 iterations. This shows the system is able to identify feedback that spans the data space, and the obtained feedback is not concentrated in a small part of the space. By collecting feedback across the data space, the system is making good use of feedback requests, which is also demonstrated by the improvement in accuracy of the models over time.

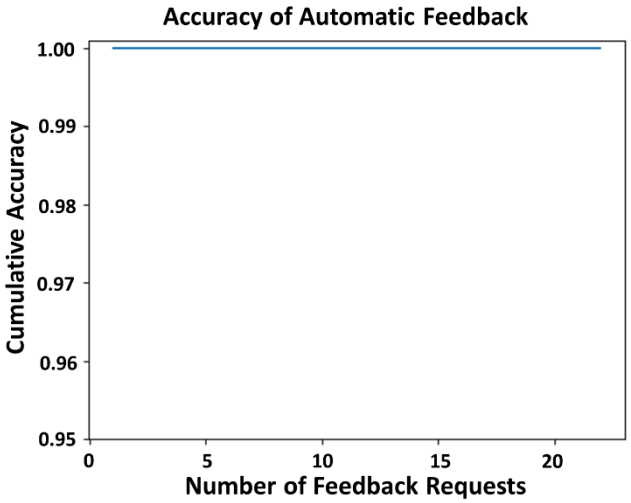

Figure 3.6: The accuracy of the automatic feedback for the Long Term Smart Home Audio Use Case. Since the accuracy is 100% across the feedback requests, this shows that whenever the system automatically identifies a group of feedback, all the data points belong to the same class. This makes it easy and valuable for the user since no corrections to the automatic feedback are required.

The graph in Figure 3.3 shows how the system continuously improves using the automatic feedback over time. Each vertical line indicates when the system automatically asks for feedback.

The graphs in Figure 3.4, Figure 3.5, and Figure 3.6 show that our algorithm is able to cover over 60% of the data automatically, while only using 20 instances of user feedback for that specific use case. The system results in each example giving a 6x increase in the number of examples added, and the accuracy of the algorithm is 100% so it is very effective and valuable, for that specific use case.

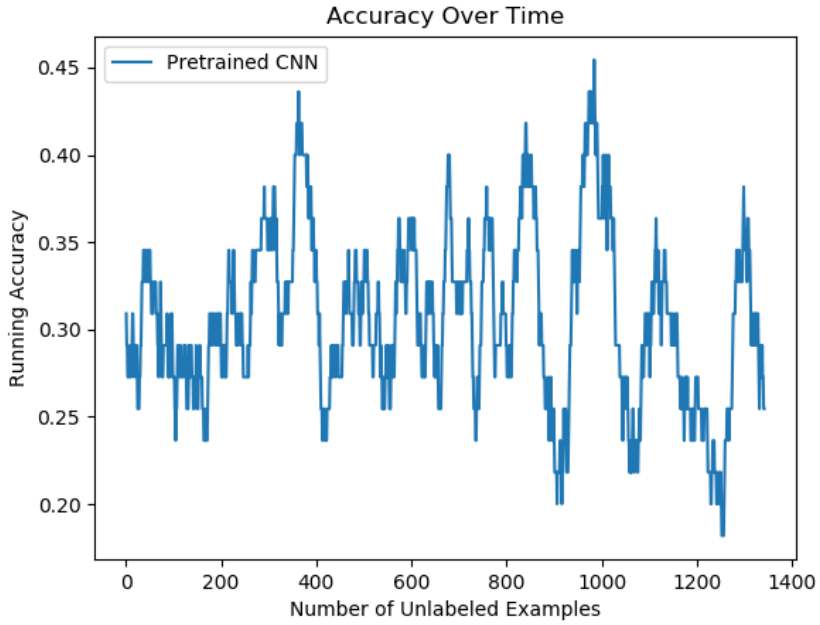

Figure 3.7: Accuracy of a pretrained audio based recognition system, Ubicoustics [14], for the Long Term Smart Home Audio Use Case. This shows that the pretrained system is not able to perform with high accuracy since it is not optimized for the local environment, and it is also not able to automatically adapt to changes in the environment unlike our ML Layer. The accuracy of Ubicoustics ranges from 25% to 45%.

We compare with the state-of-the-art deep network model in Ubicoustics [14], showing that the pretrained system overall does not have high accuracy, nor is it able to adapt to the unique environment in which it is deployed. As a result, our ML Layer which dynamically optimizes the ML models for each environment, in Figure 3.3, significantly outperforms the deep network model as shown in Figure 3.7. Furthermore, Ubicoustics [14] is tightly coupled to one particular application, while the Machine Learning Layer is much more general and not limited to audio data only.

3.6.3 Smart Home Audio Data: Short Term Adaptation Use Case

For this use case, a smaller scenario is used, which is specifically designed to show the short term adaptation. This is for a prediction task in a room. Our overall scenario has 5 total classes (background, knocking, door1, door2, door3), where there are three doors in the room. In this scenario, our system is initially trained to distinguish between background noises and a knocking event. Incrementally, the system is introduced to different door sounds (door1, door2, and door3) that are in the same area and quite similar to each other. The door sounds are intermixed with the background class that was a mix of different types of environmental noise.

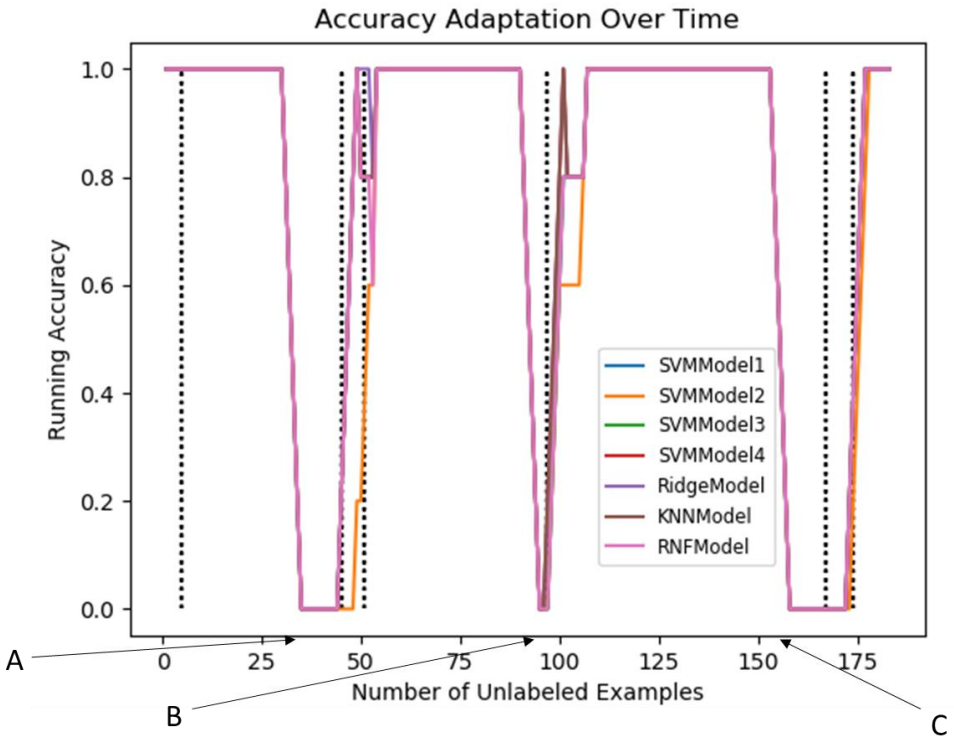

Figure 3.8: This graph shows the running accuracy (moving average accuracy for the last few predictions) for the Short Term Smart Home Audio Use Case. The dotted lines show times at which our system solicits user feedback. The points A, B, C show when the new events door1, door2, door3 are introduced respectively.

The graph in Figure 3.8 shows the running accuracy (accuracy of the last few predictions) of each of the models as a function of prediction requests (unlabeled data). Each vertical line indicates when the system automatically asks for feedback. There is a sharp drop in accuracy when a new 'door1' event occurs at point A in the graph, which has not been seen before. After some events

of ‘door1’ opening and closing, the system automatically recognizes this as a new event and solicits feedback for this new class, and the accuracy quickly rebounds to 100%.

When the ‘door2’ class is introduced at point B in the graph, initially all the models confuse it with ‘door1’. However, the automatic feedback algorithm is able to quickly identify this change and get user feedback. Once again, the models become accurate quickly as they are updated. A similar adaptation occurs when ‘door3’ is introduced at point C in the graph.

This shows the power of the system in being adaptive to short term change. It is quickly and effectively identifying new different classes even with small amounts of unlabeled data, and automatically soliciting user feedback.

The graph in Figure 3.8 also shows how initially when a new class is encountered, the system identifies feedback, but after a short while (one or two interactions) it stops asking for feedback for that class. This shows the capability of the algorithm in asking feedback only as needed, and not repeatedly for the same class.

In this case of different doors, the classes are actually very similar to each other, but our algorithm is still able to differentiate between them and identify separate feedback. On receiving feedback, our system is able to update and retrain the models, ensuring that accuracy is maintained and that there is fast adaptation to a new event.

In this case since the state-of-the-art deep models, like Ubicoustics [14] are static, they do not have the ability to distinguish between events like individual doors. In contrast our Machine Learning Layer dynamically optimizes the models over time for the environment leading to much better end application and user experiences.

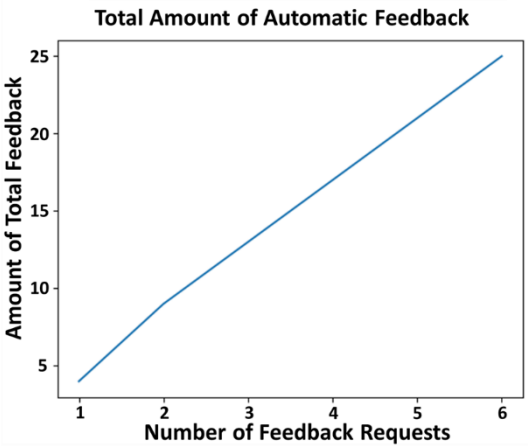

Figure 3.9: The total amount of automatic feedback for the Short Term Smart Home Audio Use Case as a function of the number of feedback request iterations. The system results in each iteration of feedback giving a 4x increase in the number of labeled examples which can be used by the system to improve the models.

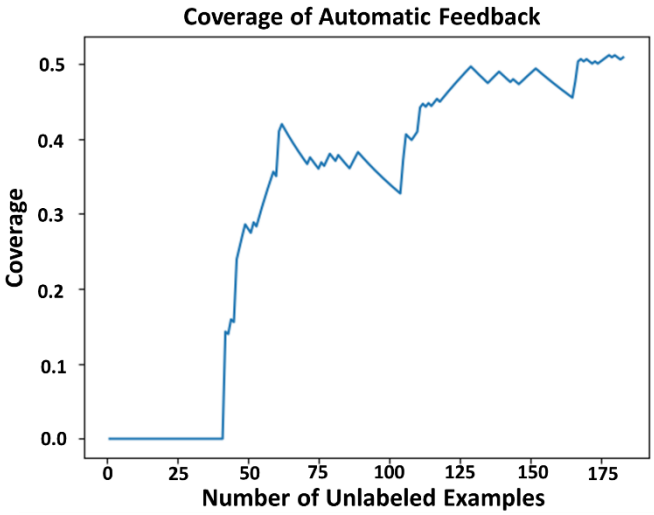

Figure 3.10: The amount of data that is covered by the automatic feedback for the Short Term Smart Home Audio Use Case. The coverage is around 50% after just 6 interactions, which shows that the automatic feedback is not concentrated in a small area, but rather is spread across the data space. This is confirmed by the accuracy graph in Figure 3.8 which shows that the automatic feedback is identifying the different events as they occur, allowing accuracy to improve.

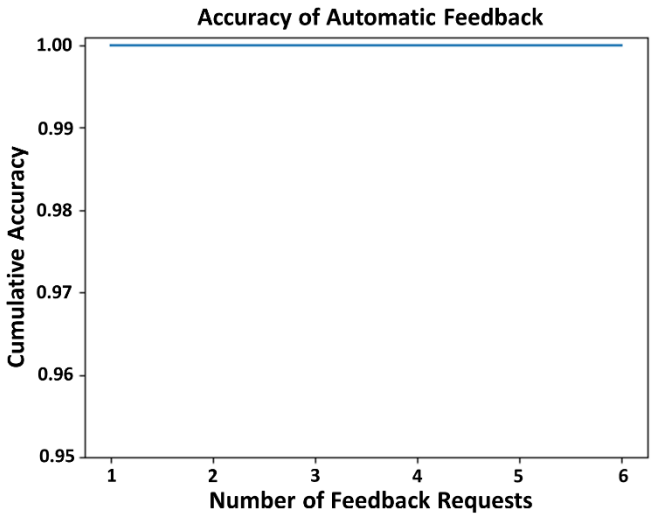

Figure 3.11: The accuracy of the automatic feedback for the Short Term Smart Home Audio Use Case. Since the accuracy is 100% across the feedback requests, this shows that whenever the system automatically identifies a group of feedback, the datapoints all belong to the same class. This makes it easy and valuable for the user since no corrections to the automatic feedback are required.

The graph in Figure 3.9 shows the total amount of feedback that the automatic feedback algorithm identifies as function of the number of requests. This graph demonstrates that each interaction

leads to 4 times the feedback which is a significant improvement in feedback efficiency, for this specific use case.

The graph in Figure 3.10 shows the coverage of the automatic feedback, which is around 50% after just 6 interactions for this specific use case.

The graph in Figure 3.11 shows that the feedback suggested by the system is completely accurate, which makes the automatic feedback very easy and valuable for the end user.

Chapter 4

Optimized Dimensionality Reduction

4.1 Motivation for Dimensionality Reduction

Our Machine Learning Layer focuses on improving accuracy and adapting to continuous environmental changes. However, a secondary goal for our ML Layer is to be efficient and computationally low cost to facilitate deployment. To this end, we integrate dimensionality reduction techniques and show that we can improve serving efficiency of the models which are optimized for the environment, without compromising accuracy.

For the IoT setting there are several benefits to dimensionality reduction. Firstly, dimensionality reduction can produce models that are less computationally demanding and consume less resources when used for continuous predictions, in an online setting. In addition, dimensionality reduction allows the Machine Learning Layer to also run on more energy constrained platforms such as an edge IoT gateway. Also dimensionality reduction can result in models that take less space to store, which can also be useful for resource constrained devices.

To integrate dimensionality reduction into the Machine Learning Layer, we present another algorithm which enhances the Model Selection aspect in the Training Module of our ML Layer. It is very important that the system has been exposed to all possible data before applying dimensionality reduction, since otherwise the dimensionality reduction cannot be done without losing classification relevant information. So, the model selection with dimensionality reduction is only applied once the system has collected a large amount of data. It is important to note that the system uses all the data, not just the data that the user provides feedback.

Using our algorithm, the amount of dimensionality reduction achievable can vary from model to model. This allows our Machine Learning Layer to create a customized data transformation optimized on a per model basis that is based on the data collected in the actual environment, and adapt models to improve efficiency in the long run.

4.2 Techniques for Dimensionality Reduction

There were many different techniques for dimensionality reduction that we experimented with. Some of these techniques are described in this section. Since our goal was not just to do dimensionality reduction, but integrate it with our Machine Learning Layer, we had several design criteria in mind:

- It should perform well with different types of high dimensional data that can occur in IoT settings. In this case, the performance refers to the classification accuracy which is important to the end result, and not information preservation metrics used for typically used dimensionality reduction
- It should perform relatively efficiently during the training phase of the dimensionality reduction
- It should not have many hyperparameters to tune, so that there is little need for additional tuning to happen. These are the hyperparameters of the dimensionality reduction algorithm, and not the model hyperparameters.

Some of the dimensionality reduction techniques that we explored are described below.

Principal Component Analysis

Principal Component Analysis or PCA [25] is a widely used dimensionality reduction technique. It is based on identifying linear subspaces of the original data that capture most of the variation present in the data. PCA computes principal components, which are the basis of the transformed space which is of lower dimensionality than the original space. PCA has the advantage that is computationally very simple, but since it is limited to linear transformations, other methods are more effective where nonlinear structure is present, which is often the case in high dimensional sensor data as shown in Figure 4.1.

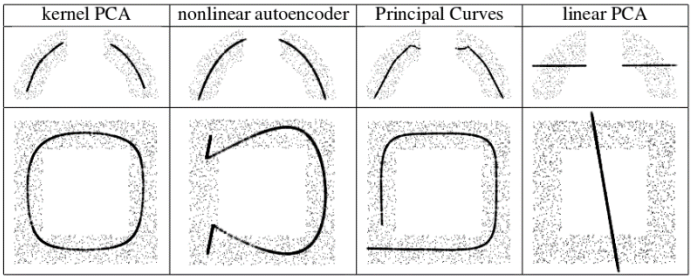

Figure 4.1: This image shows the first principal component for two different datasets. This shows that Kernel PCA is effective when apply to non-linear data, while PCA is very limited since it is only based on linear subspaces. [Image from [30]]

Manifold Learning T-SNE/ISOMAP, UMAP

Another class of dimensionality reduction techniques is based on manifold learning [26] or related techniques. These techniques are very different than PCA. They do not compute a simple transformation of the data, but instead create a complex nonlinear mapping of the data based on the overall structure and distribution of data under different distance metrics. These algorithms do not try to minimize reconstruction error, and instead seek to preserve the structure of the data with respect to the distance metric. There are several algorithms that do this, including t-Distributed Stochastic Neighbor Embedding (t-SNE) [26] which has been very widely used, isometric mapping (ISOMAP) [27], and the more recent Uniform Manifold Approximation and Projection (UMAP) [28] as illustrated in Figure 4.2. t-SNE is a non-parametric method, so it does not compute a transformation that can be applied to serving data once it is calculated on the training data. In contrast, UMAP is a parametric method, so it can then be applied to serving data. Unlike PCA, UMAP has many different tunable parameters to tradeoff preserving the global structure of the data and preserving the local structure of the data. Furthermore, each iteration of UMAP takes much more time than PCA, so it can be much more expensive when it is used with large amounts of environmental data.

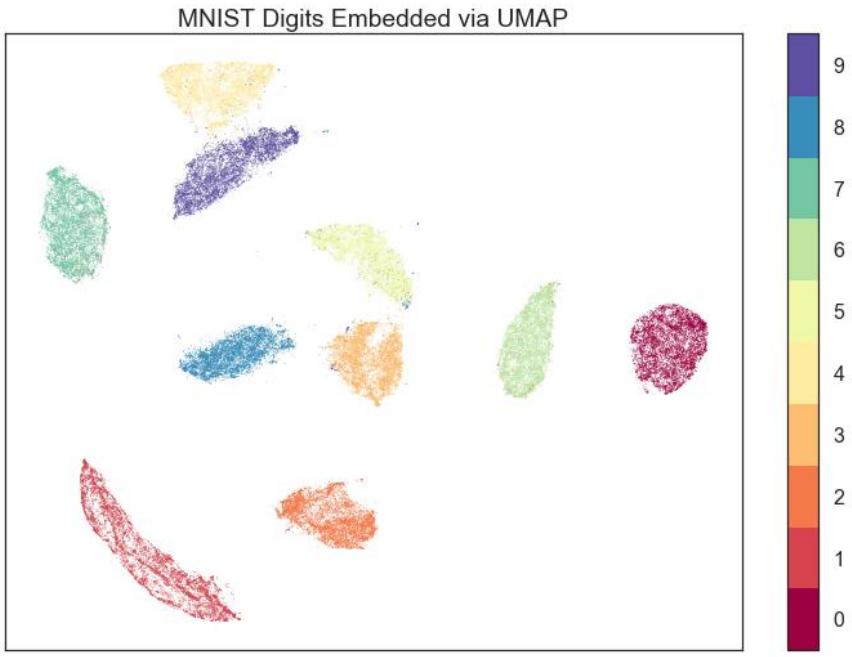

Figure 4.2: This is an example image of the output of UMAP on the full MNIST digit recognition dataset. UMAP can reduce dimensionality to 2 dimensions by trading off the global structure of the data with the local structure of the data. [Image from [31]]

Kernel Principal Component Analysis

Kernel Principal Component Analysis [29] is based on the regular PCA. However, like the manifold learning techniques like UMAP, it can learn nonlinear data transformations as shown in Figure 4.1. It does this by introducing a kernel function into the PCA algorithm. This kernel function is a function that is a dot product of data after a data transformation:

$$K(x, y) = \phi(x) \cdot \phi(y)$$

There are many different types of kernels such as polynomial, radial basis function, cosine etc. Dimensionality reduction techniques do not always preserve classification relevant information. Across different datasets, we found that the cosine kernel was effective at preserving information relevant to the prediction tasks.

4.3 Algorithm for Optimized Dimensionality Reduction

In this section, we describe the algorithm for optimized dimensionality reduction.

First, the algorithm runs Kernel PCA on the unlabeled data collected from the environment. The algorithm runs this with the maximum number of dimensions possible, which is the minimum of the number of samples in the unlabeled data and the dimensions in the unlabeled data.

Then, the algorithm optimizes the dimensionality reduction for each model. This optimization is over the range of possible dimensions. For each setting of the number of dimensions, the model is trained and tuned with its hyperparameters which are further optimized. This search can be expensive especially when there is a large amount of data, and the original dimensionality of the data is high.

The algorithm does a couple of things to make this more efficient.

Firstly, the algorithm uses a smart search strategy to optimize the number of dimensions for each model, by using Bayesian Optimization, described earlier, with an initial set of exponential starting points across the range of dimensions. Secondly, the algorithm does not rerun Kernel PCA for each setting of dimensions during the optimization, but rather makes use of the initially trained Kernel PCA and then takes the appropriate range of dimensions at each iteration.

Figure 4.3 illustrates how this algorithm works to train the full set of models while optimizing dimensionality reduction in the Training Module.

It can be noted that while this algorithm is tightly integrated with Kernel PCA, it can be further adapted to include other types of dimensionality reduction methods.

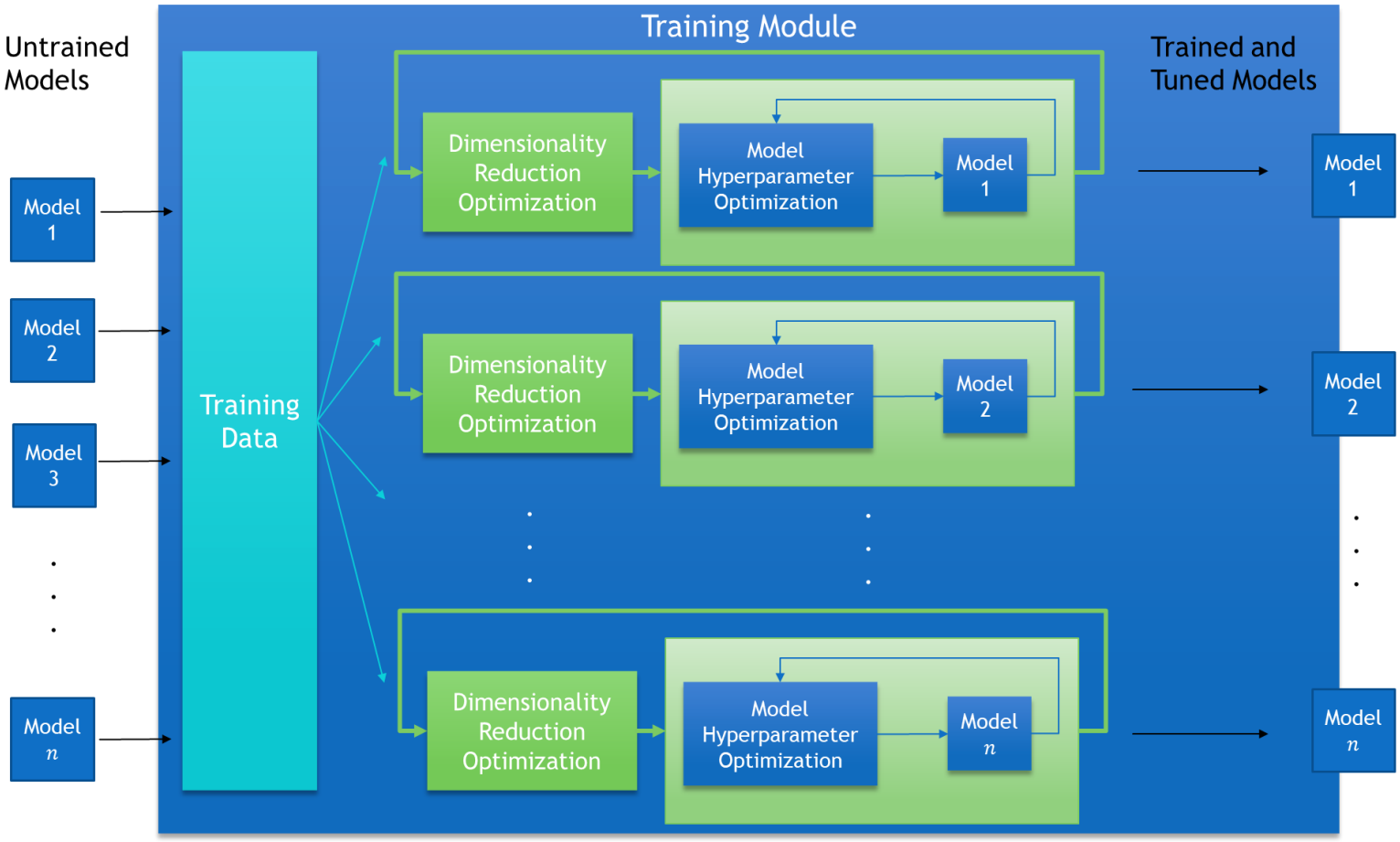

Figure 4.3: A representation of the algorithm for optimized dimensionality reduction in the Training Module. There are different models, for example Model 1 can be the Scikit Learn SVM model, which has its own set of hyperparameters. For each model, there is an outer level of dimensionality reduction optimization (which consists of several iterations), and an inner level of hyperparameter optimization (which itself consists of several iterations). Each iteration of the dimensionality reduction leads to retuning the model. This is done for all the models to produce the final set of trained and tuned models with dimensionality reduction.

4.4 Results with Dimensionality Reduction

Smart Home Audio Dataset: For the Smart Home Audio Data that was described before, previously the accuracy comparison with the pretrained audio system was provided. Here a comparison of execution time performance is provided with the same state-of the art pre-trained system, Ubicoustics [14].

Combined Sensor Dataset: We also use a dataset collected from MITES sensors [15] for evaluation. The MITES sensor is a combined sensor package that has many different streams from different heterogeneous sensors including temperature, light, acceleration, microphone, etc. The overall size of the input that is provided from the MITES sensor is 1172. This dataset consists of 6 classes corresponding to common events in a smart home.

These two data sets are useful to evaluate the dimensionality reduction as both of them have large feature vectors, with many different correlated signals. The Machine Learning Layer dimensionality reduction algorithm is able to effectively identify nonlinear structure and reduce dimensionality, based on environmental data. Our results show the efficiency benefits when dealing with complex and high dimensional sensor data from different sources while maintaining high accuracy.

4.4.1 Accuracy for Serving

We first present results that show the changes to the prediction accuracy before and after dimensionality reduction.

Figure 4.4 illustrates the accuracy results for the Smart Home Audio Dataset. Figure 4.5 shows the accuracy results for the Combined Sensor Dataset, using the MITES data we collected.

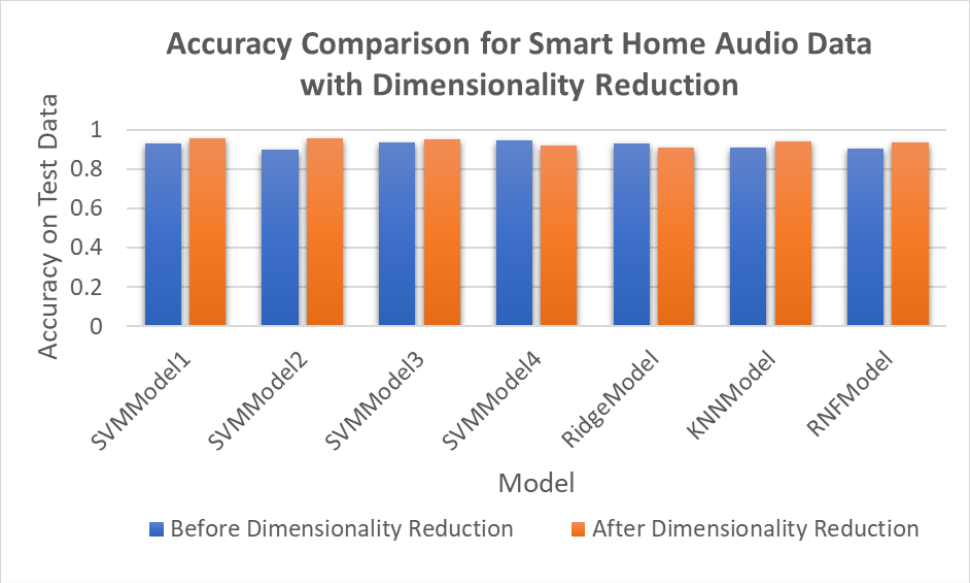

Figure 4.4: Accuracy comparison using dimensionality reduction for the Smart Home Audio Dataset with the various classes. Before dimensionality reduction, the number of dimensions was 6144. As can be seen, the dimensionality reduction allows the accuracy to be preserved.

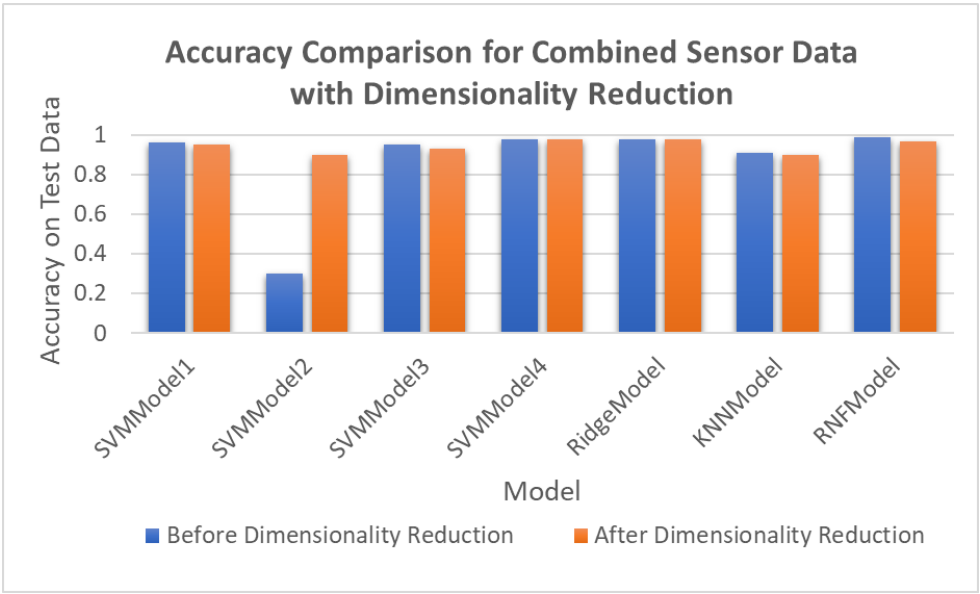

Figure 4.5: Accuracy comparison using dimensionality reduction for the Combined Sensor Dataset, based on the MITES platform with 13 physical sensors. Before dimensionality reduction, the number of dimensions was 1172. As can be observed, the accuracy with dimensionality reduction is mostly unchanged, except for the second model, for which accuracy increases significantly.

These results demonstrate the ability of the algorithm to apply dimensionality reduction in a way that allows accuracy to be preserved. In the case of the Smart Home Audio Data, for many of the models, the accuracy can even increase slightly as a result of dimensionality reduction.

4.4.2 Performance for Serving with Dimensionality Reduction

This section compares the serving performance before and after dimensionality reduction. We use the prediction latency as a proxy for performance as has been done in prior work, like Clipper [10].

The graph in Figure 4.6 shows the latency results for the Smart Home Audio Dataset

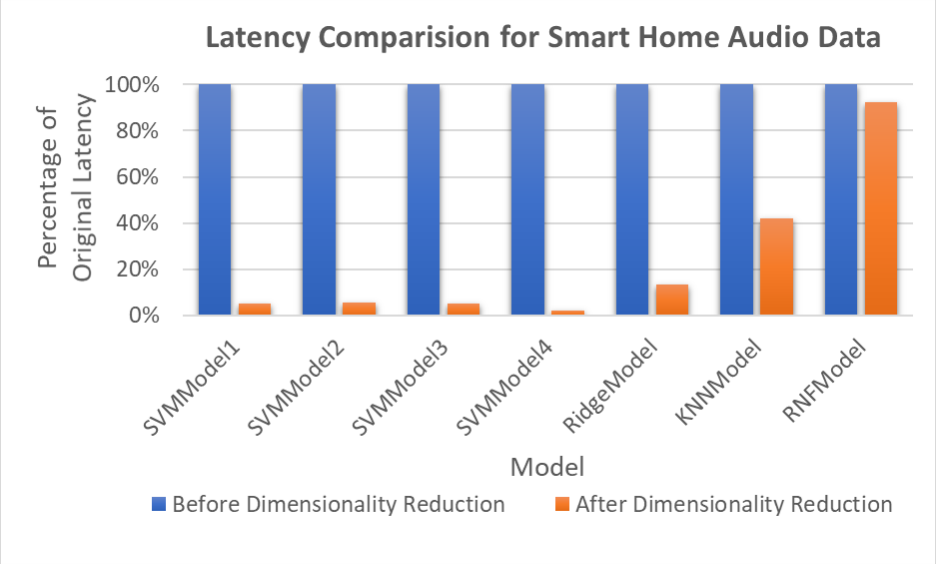

Figure 4.6: Latency comparison using dimensionality reduction for the Smart Home Audio Dataset, running on the same hardware configuration. This shows that the dimensionality reduction results in latency improvement for all the models, and significant latency improvements for most of the models.

This shows that the prediction latency for all the models improves, and there is significant improvement in latency for almost all the models except for the random forest, by using dimensionality reduction.

In comparison, the latency per prediction for the state-of-the-art pretrained deep learning model, Ubicoustics [14], is 0.080434 seconds. The latency of each of the models shown in the graph after

dimensionality reduction, relative to the latency of the state-of-the-art system are 1444x, 1259x, 1545x, 394x, 2126x, 93x, 58x faster, respectively.

Our results show a significant improvement in computational performance over the state-of-the-art system. Furthermore, combined with the previous results which showed better accuracy for our ML Layer, as well as the fact that our models are highly localized to each environment, this highlights that our ML Layer is able to deliver much better results for this audio based context recognition use case in all aspects.

Next, the graph in Figure 4.7 shows the latency results for the Combined Sensor Dataset.

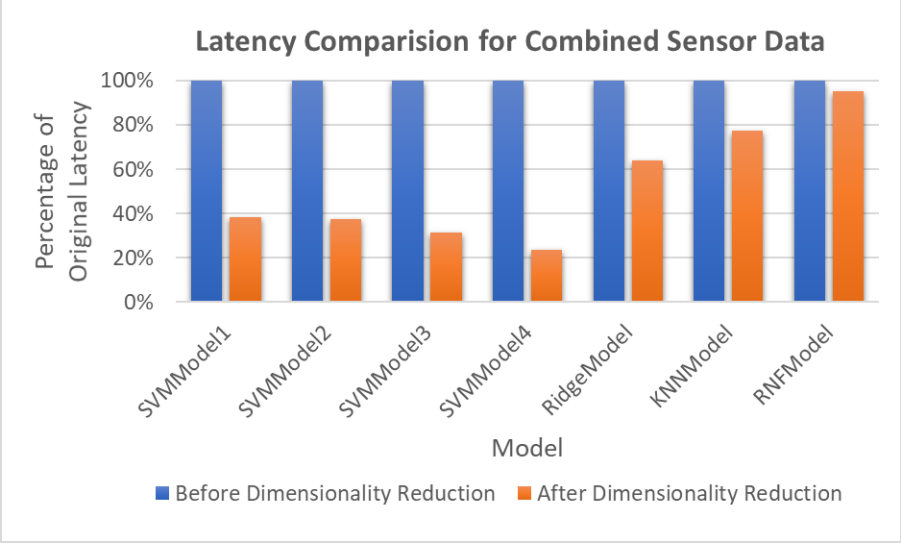

Figure 4.7: Latency comparison using dimensionality reduction for the Combined Sensor Dataset, running on the same hardware configuration. This shows that the dimensionality reduction results in latency improvement for all the models, and significant latency improvement for the SVM models.

4.4.3 Per Model Optimization Results

The graph in Figure 4.8 shows that the model selection algorithm uses different amounts of dimensionality reduction for each model for the Smart Home Audio Dataset.

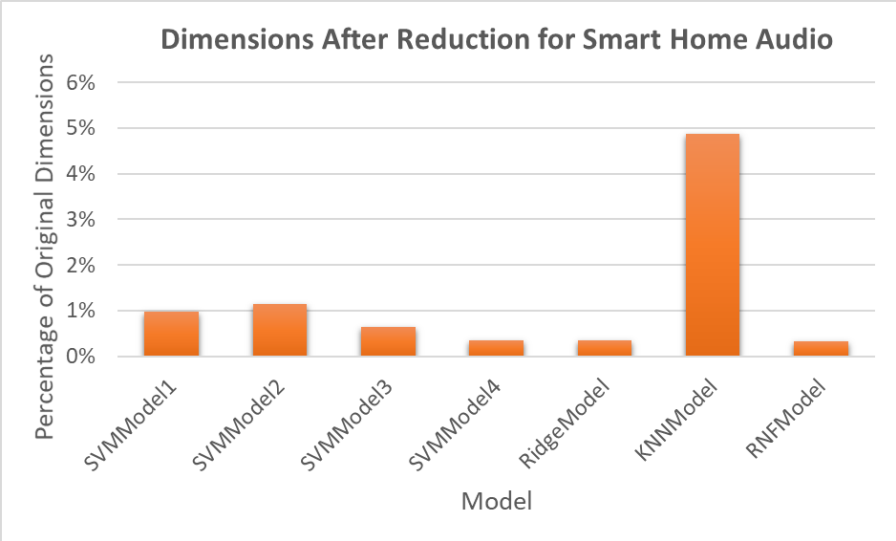

Figure 4.8: Results for per model dimensionality reduction optimization for the Smart Home Audio Dataset. The number of original dimensions was 6144. The graph shows that for all the models, the number of dimensions is reduced to less than 5% of the number of original dimensions.

The graph in Figure 4.9 shows that the model selection algorithm uses different amounts of dimensionality reduction for each model for the Combined Sensor Dataset.

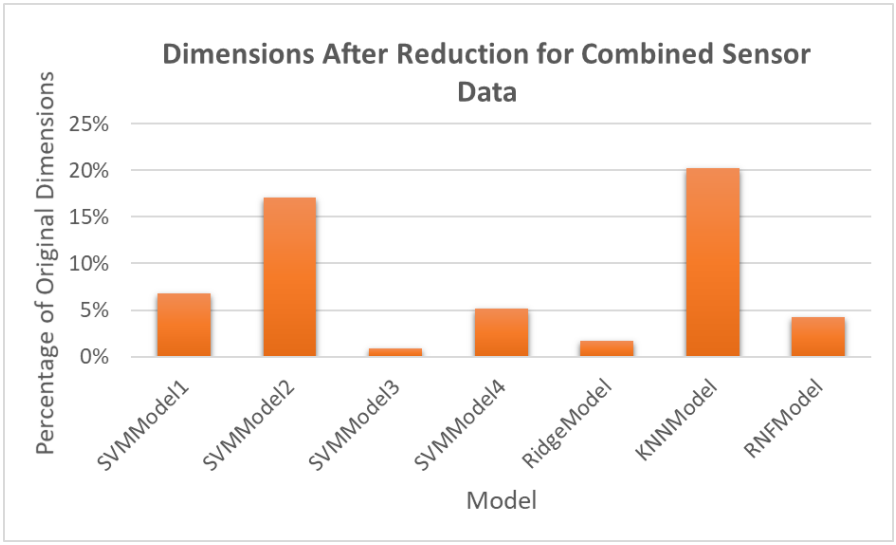

Figure 4.9: Results for per model dimensionality reduction optimization for the Combined Sensor Dataset. The number of original dimensions was 1172. The graph shows that for all the models, the number of dimensions is reduced to less than 25% of the number of original dimensions, and to less than 10% for many of the models.

These results show the importance of this algorithm which optimizes on a per model basis. This has both accuracy and efficiency advantages. Since it optimizes for accuracy for each model it is able to keep the best amount of information for that model. This leads to further efficiency advantages for some models which perform better with more dimensionality reduction, as the number of dimensions is optimized independently for each model.

Chapter 5

ML Layer Management Module

5.1 Overview

So far, the Machine Learning Layer Training Serving system, with its algorithms and components has been presented. Each instance of the core training/serving system modules are designed for a specific prediction task, with a specific set of sensor inputs. It is trained to optimize for that task using that data and improve it with feedback over time. In large scale IoT deployments, there are many prediction tasks that have to run simultaneously. For example, in a smart home, one prediction task may involve face identification using front door smart camera data, while at the same time inside the house there may be prediction tasks associated with different rooms, using the set of sensors within each room.

For each of these prediction tasks, there are different instantiations of both the training and serving aspects. From here on, the term *prediction stream* is used to refer all the components that are involved in performing a particular prediction task, including the training and serving modules. In this section, we describe the design of a management layer to handle the dynamic management of many different prediction streams. This also allows different clients and applications to simultaneously make use the features of our Machine Learning Layer.

The system is designed to run in a distributed setting, and takes efficiency and security into account. Just like the training and serving system presented so far, the ML Layer Prediction Stream Management System was also designed to have a clear and easy to use interface for any type of

client application. Our Management Module consists of several components, as shown in Figure 5.1.

- The Central Management Service (CMS) which manages all the different prediction streams
- The Starter Service (SS) which is a local service that manages training and serving servers on each machine.
- The Training Module which does training and tuning and improvement of models
- The Serving Module which does dynamic prediction serving

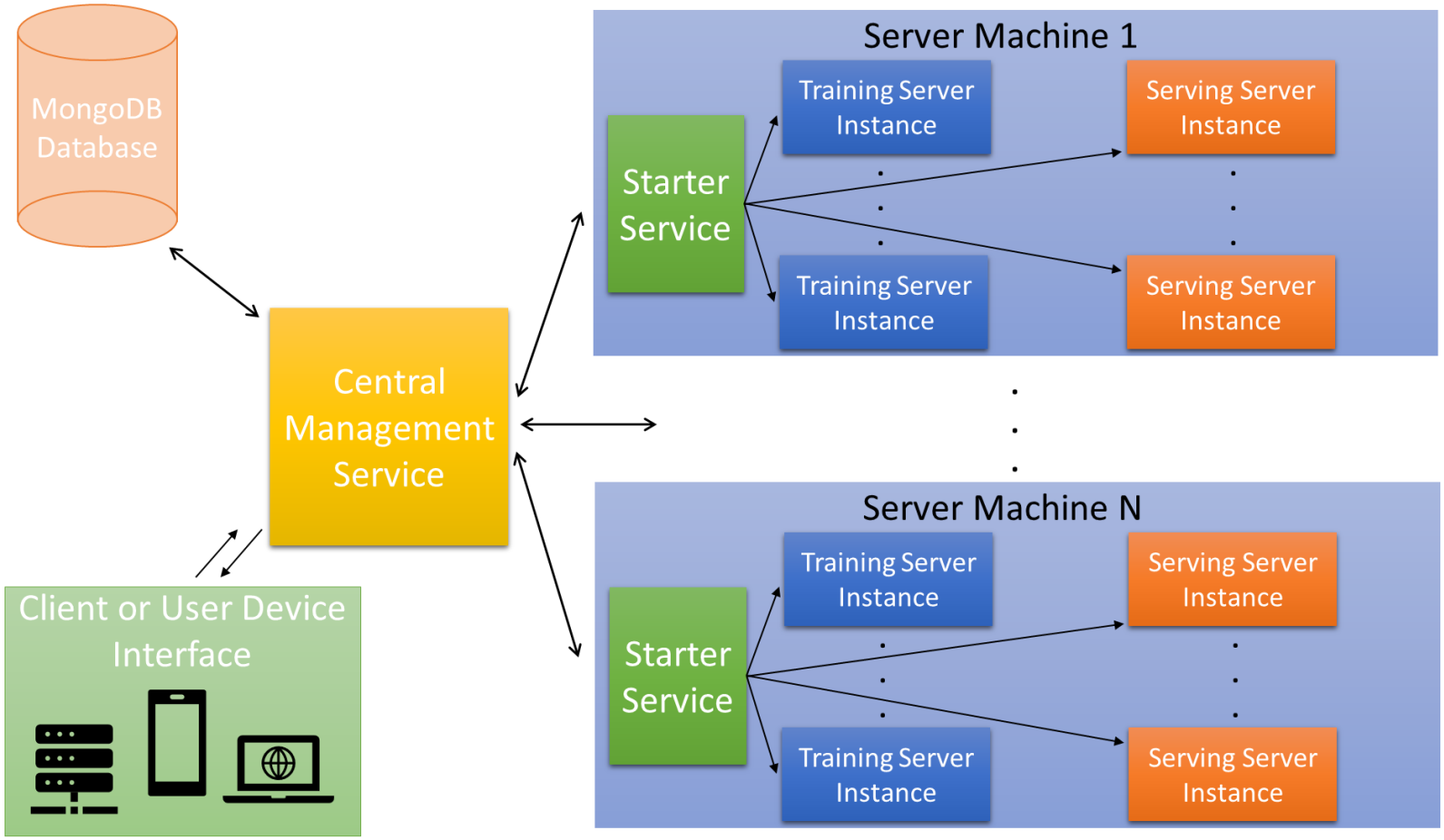

Figure 5.1: Overall diagram of the components of the ML Layer Prediction Stream Management System.

Clients interact with the Central Management Service to manage prediction streams. Clients interact with the training and serving modules directly to provide training data, request predictions, provide feedback, etc. as described previously.

The various components of the system and implementation are described in the next sections.

5.2 Implementation Choices

We had multiple goals when implementing the ML Layer. These include cross platform usability, flexible deployment, scaling across machines, along with efficiency and security.

To allow the system to be cross-platform, and integrate well with many ML libraries, etc., we implemented the entire system in Python [32]. Many underlying libraries are already well supported Python.

The communication between the various components occurs through RPC. There were different RPC methods that we attempted initially. However, based on the implementation goals for the ML Layer, gRPC [33], which is based on the RPC mechanisms developed at Google, was chosen as the final RPC method. Since it supports cross-platform RPC calls from many different programming languages, it provides the required flexibility to handle clients written in many different languages simultaneously. It is also scalable, lightweight, and battle tested.

The ML Layer frequently exchanges large amounts of information like models, and environmental data. Also, reducing prediction request latency for Serving is important, especially since the system is used for high dimensional sensor data. As a result, gRPC's use of protobufs for data transfer along with other performance optimizations, makes it an effective choice.

Finally, as described in section 6.5, the ML Layer requires security in the various interactions. As a result, gRPC is a useful RPC method as it supports standard security methods, such as asymmetric key cryptography.

5.3 Central Management Service (CMS)

5.3.1 Central Management Service Details

The Central Management Service creates and manages different prediction streams. A prediction stream is meant for a particular prediction task.

The functionality that is supported includes:

- Creating training serving streams dynamically based on requests from the client
  The Stream Management Service can choose between different machines and supports managing Training and Serving instances on different machines (described next)
- Performing cryptographic key sharing as needed
- Deleting the Training and Serving server instances given the stream id
- Providing information to the client about a particular stream
- Managing all the database information on a per stream basis
- Periodically checking the performance and utilization metrics of different machines and instances, creating a central system view which is useful for both system management and performance analysis in real deployments

Clients of the training and serving interact directly with the Training and Serving instances once they are created. The CMS is used for the creation, and management of instances and it does not receive any training or serving related data from the client.

Policies for Stream Creation

One of the goals of the management system is to be able to scale horizontally, making use of different machines to do training and serving. Machines can also include smaller edge nodes including sensor devices, with different computational capabilities. As a result, the way in which work is distributed is important.

The CMS creates a new Training and Serving instance for each new stream. For this purpose, it supports different policies to choose which machines to use when a client requests a new prediction stream to be created. There are different strategies which were implemented including:

- Round Robin based scheduling to available machines

- Dynamic load-balancing based on the ratio of the number of instances a machine has to the number of cores that it has, to allows more powerful machines to share higher loads

Strategies that balance the instances using system metrics, like the number of cores, are important since most of the machine learning operations on both the Training and Serving instances are computationally intensive.

Internal Mappings

To enable this, internally the CMS maintains several data structures:

- The unique identifiers (provided by the client) of all the current prediction streams
- All the available registered machines and the Starter Services (SS)
- The mapping between streams and the training and serving instances which are associated with those streams

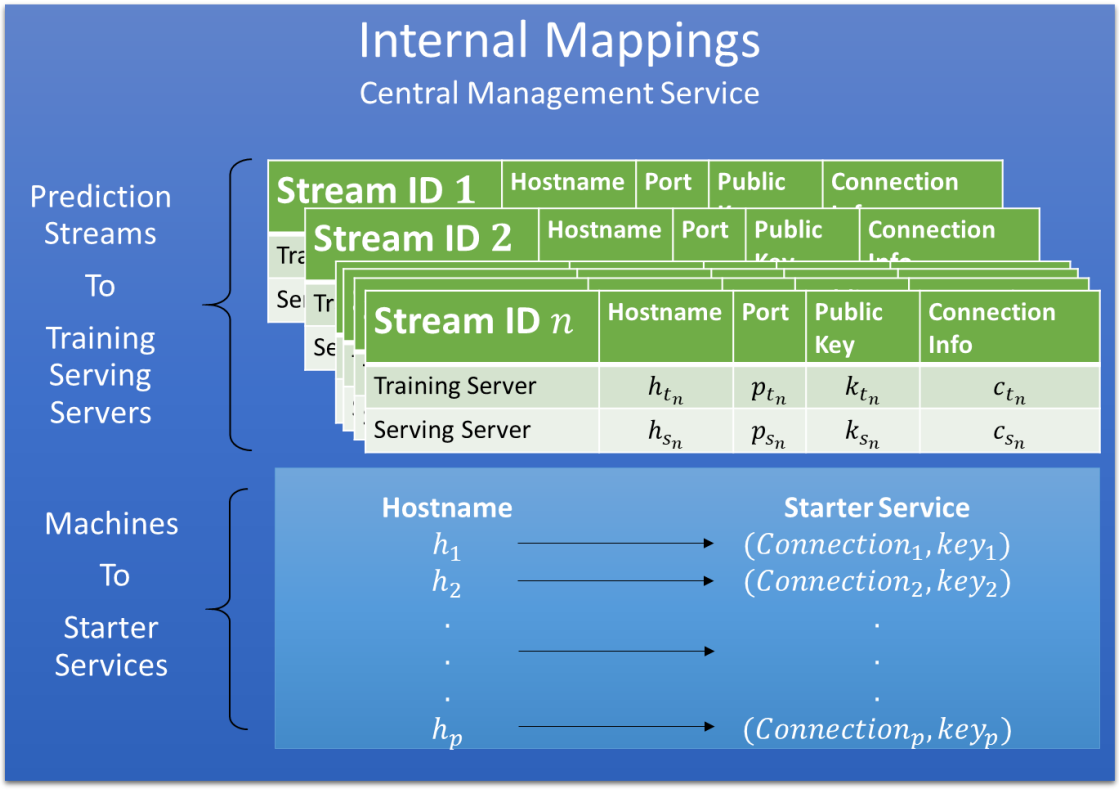

Figure 5.2: This diagram represents the information for mappings that the Central Management Service holds internally to manage Training and Serving across several different machines.

5.3.2 Central Management Service Interactions and API

The client interacts with the Central Management Service using the functions:

- *Create_stream()*

  This creates a new stream given a stream id from the client.

  The Starter Service (SS) instances are used to create training and serving instances on the available machines. If the Stream id already exists, then the existing stream information is returned so that the client can continue to use the training and serving instances that are already in use. It also adds to the new stream to all the internal mappings.
- *Delete_stream()*

  This deletes a stream given an id. The Starter Service instances are used to delete the associated training and serving instances for that stream. It also removes the stream from all the internal mappings.

The Starter Service (SS) interacts with the CMS using the function:

- *Add_server()*

  This registers the machine with the CMS when the Starter Service is first started on that machine

The Training and Serving instances interact with the CMS using the functions:

- *Update_training_data()*

  This updates the training database information for the given stream
- *Update_serving_data()*

  This updates the serving database information for the given stream

These API functions and interactions are shown in Figure 5.3.

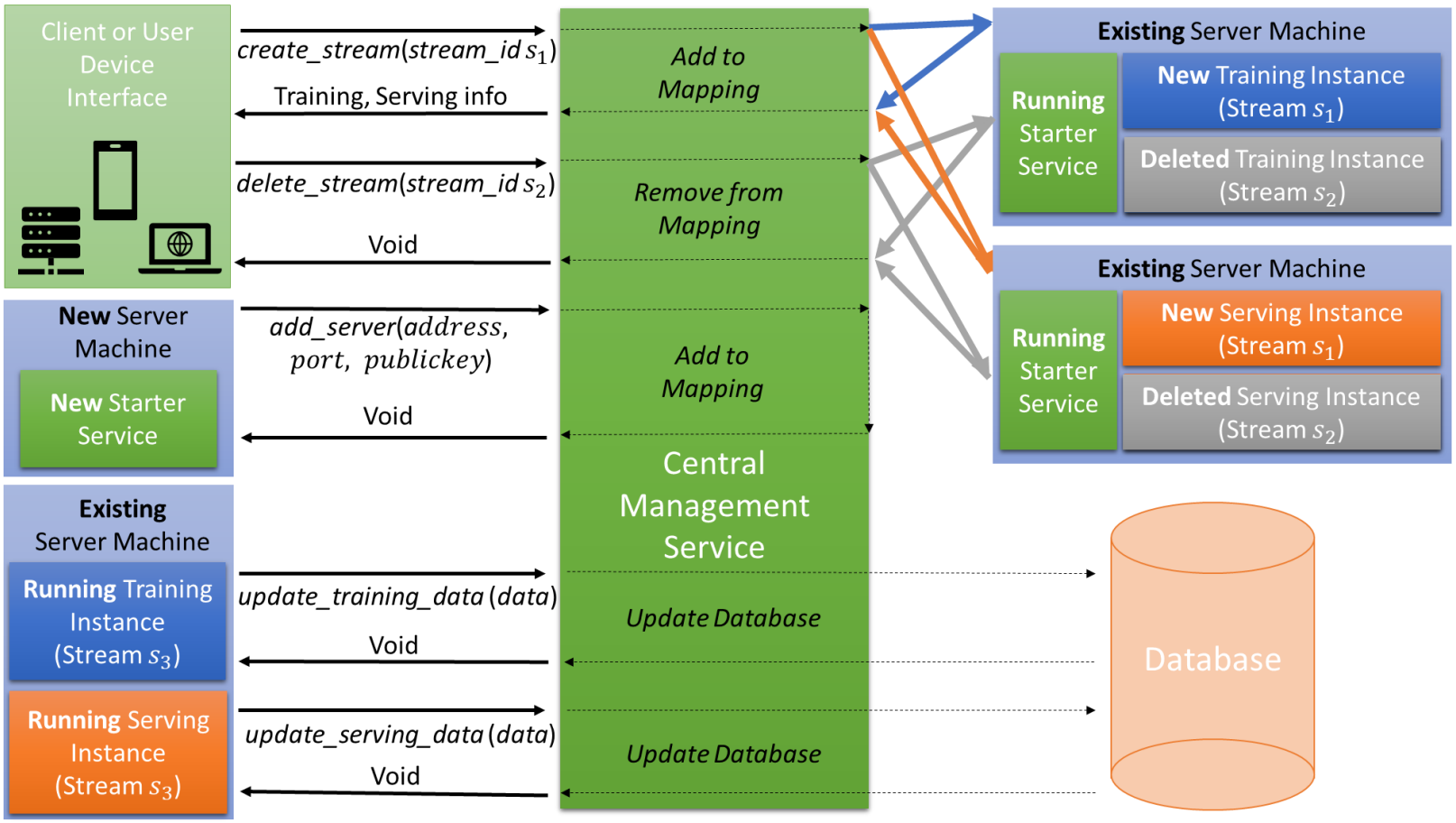

Figure 5.3: An illustration of how the Central Management Service handles requests and interacts with components which uses its API

5.4 Database Storage

The system also has a database component that stores the data for the entire system in a central location, which is updated periodically.

The data for the training side includes the

- Training Data
- Training Module Models
- Hyperparameters for each model
- Training accuracies for each model
- Validation accuracies for each model

The data for the serving side includes the

- Weights for the ensemble of chosen models
- Serving Module Models
- Aggregated Serving Feedback/Data and Metrics

The system does not require a particular database or implementation of storage. It can easily be used with any type of database depending on the scale of the deployment and setup. For example, if the system is dealing with very high dimensional data from a large number of sensors and many prediction streams, then enterprise scale databases like Google Spanner [34], can be used. On the other hand, if the use case is much smaller scale, then a simple database can be used.

For our current implementation, we use a MongoDB [35] database as our data store. Everything is stored on a per stream basis and the training and serving data is stored in different sets of objects. The data is updated periodically, and the training and serving instances send the updated data to the CMS to update the database.

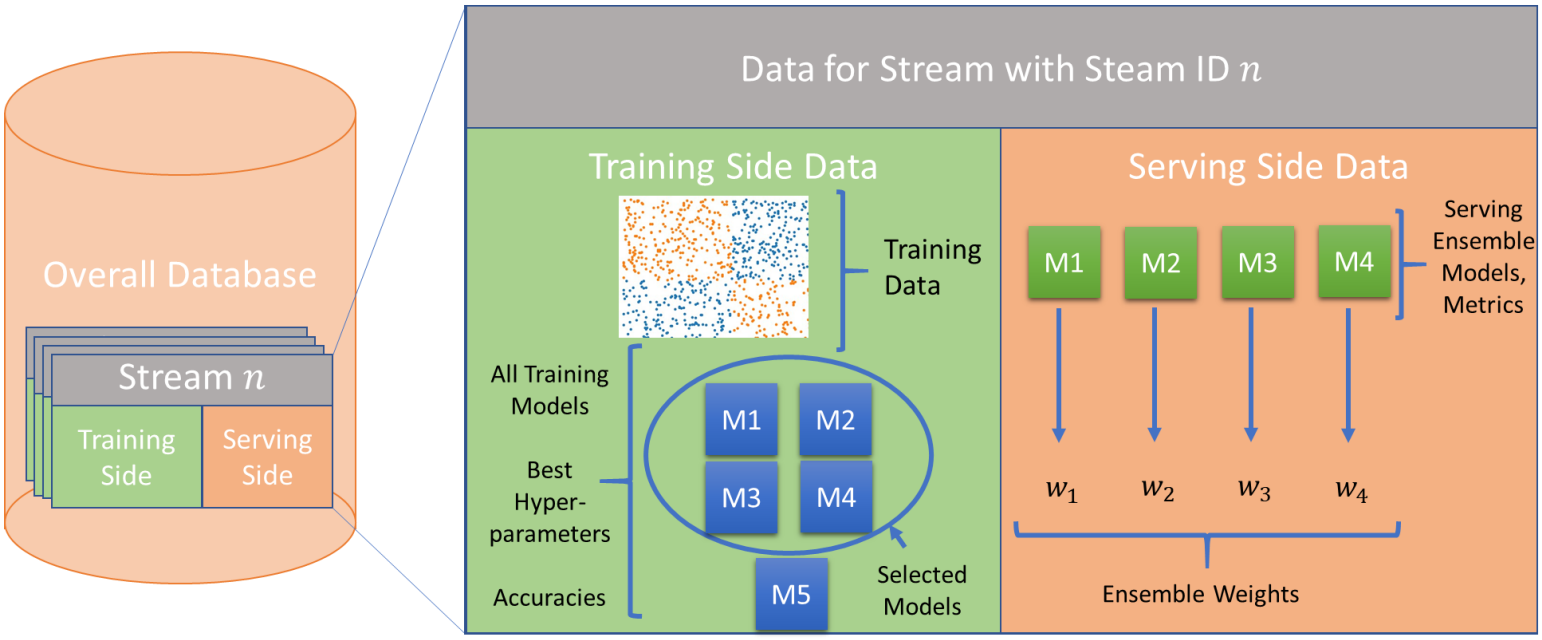

Figure 5.4: An example of the per stream data in the overall database for the system

The system needs to be able to store large files, combinations of models, etc. As a result, in the current implementation, the data is stored using the GridFS API within MongoDB to ensure that large files can be stored and loaded efficiently. The models and all the data are stored in a standardized format. Specifically, the models are stored using the same standard method that is used when they are serialized and communicated between the Training and Serving Layers.

Updating

The database updating was implemented in different ways. Initially there was a pull mechanism that would allow the CMS to pull the information from the training and serving instances on a regular basis.

However, this can be inefficient when there are lots of streams. Instead, in the current implementation, the training and serving instances push updates to the database using the Central Management Service API with the functions *update_training_data()* and *update_serving_data()*. This is much more efficient since the training and serving instances can only update as needed, on demand.

5.5 Starter Service (SS)

5.5.1 Starter Service Details

The Starter Service runs on each machine configured to be used for training and serving. There is only a single instance of the service on each machine which supports the management of all training and serving instances on that machine. The CMS has access to all of the different machines and it interacts with them using the Starter Service.

The Starter Service can do the following tasks:

- Launch new training server instances and serving server instances as different processes on the machine;
- End training server instances and serving server instances;
- Provide metrics about the system to the CMS which are used to balance across machines or create an overall view of the system

Initialization

Initially, on each machine the Starter Service instance is launched. After it is launched, it automatically registers with the CMS. After it is registered, the new machine is available to the CMS for running training and serving instances.

Regular System Measurement

The CMS regularly updates the metrics of all the machines by calling the Starter Service on each machine, and using it to measure the utilization, and other information either for the overall system or for a particular process.

5.5.2 Starter Service Interactions and API

The CMS interacts with the Starter Service using the functions:

- *create_training_server()*

  Creates a training server instance on the machine. It returns the port of the training server instance and the public key to the Central Management Service
- *create_serving_server()*

  Creates a serving server instance on the machine. It returns the port of the serving server instance and the public key to the Central Management Service
- *stop_server()*

  Stops the training or serving server instance running on a given port on the machine
- *get_info()*

  Returns the information related to the system. Returns the overall number of cores in the system. Also returns the overall system utilization, etc. It can also return per process utilization, etc.

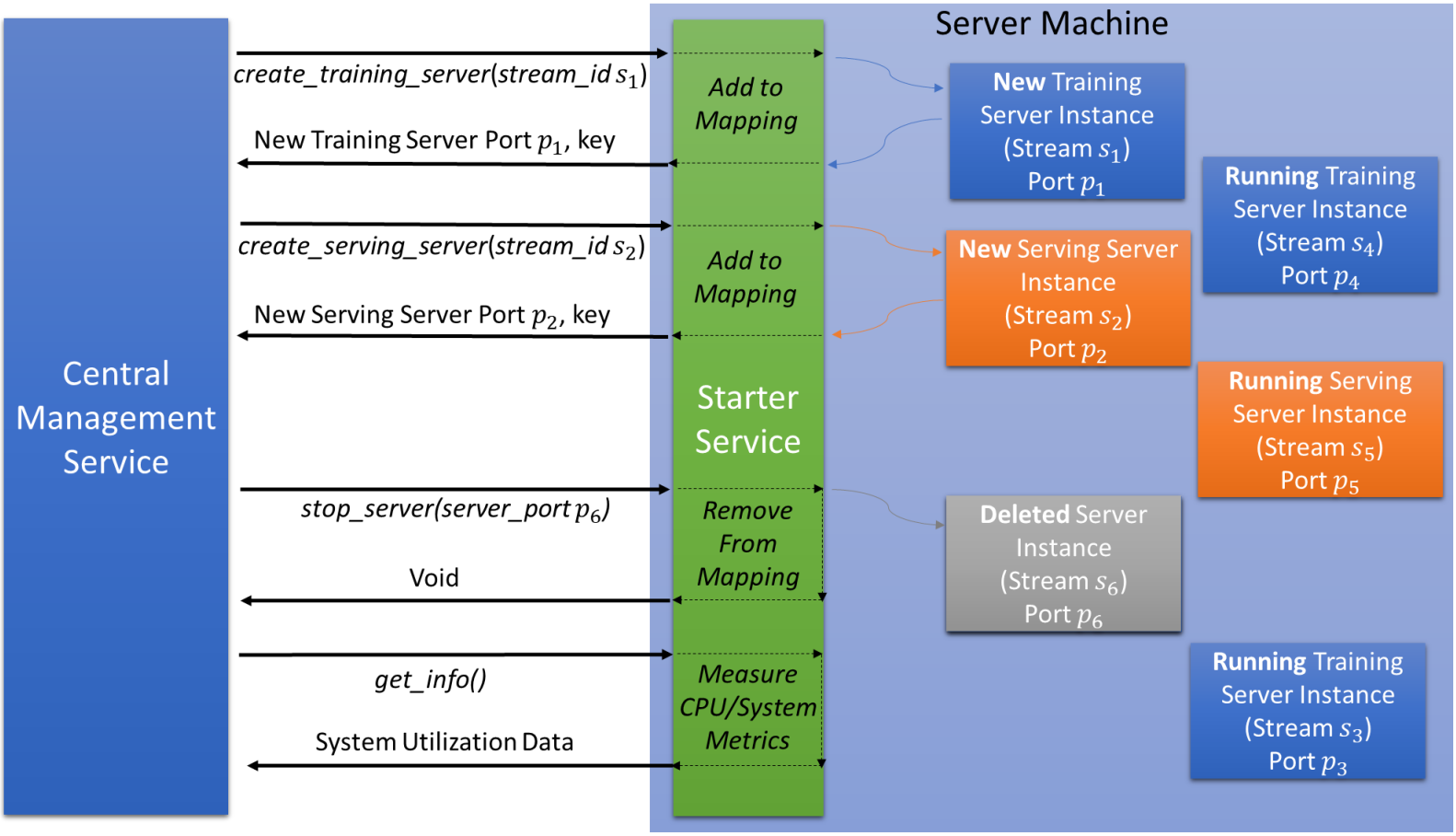

Figure 5.5: An illustration of how the Starter Service handles requests and interacts with components which uses its API

5.6 Security

5.6.1 Motivation

As described in the previous sections, there are many different components that are continuously interacting and communicating. The data they are transmitting consists of valuable information including the training data from the environment and serving data that is continuously streaming in from the environment. It also includes the models that are being trained and being served. In addition to data, the system also performs many different commands to run the various training and serving instances on different machines. As a result, considering a secure design for the communication is important.

Furthermore, the different prediction streams are all separate and they should each have their own security, since the data from one prediction stream should only be accessible to that prediction stream (for example the training server in one stream should not have access to the serving server in another stream). Also, standard authentication is important, so that callers can be sure that they are communicating and sending their data to the correct entity.

To achieve data security and authentication, both symmetric key and asymmetric key based methods can be used. In the current system, the asymmetric key based method is used along with gRPC which supports encryption and authentication for its communication channels.

5.6.2 Details of the Security Setup and Interactions

We now describe the secure communication setup. This includes the interactions between the CMS and the Starter Services on different machines, between the CMS and the training and serving instances, and between the training instance and serving instance for a particular prediction stream.

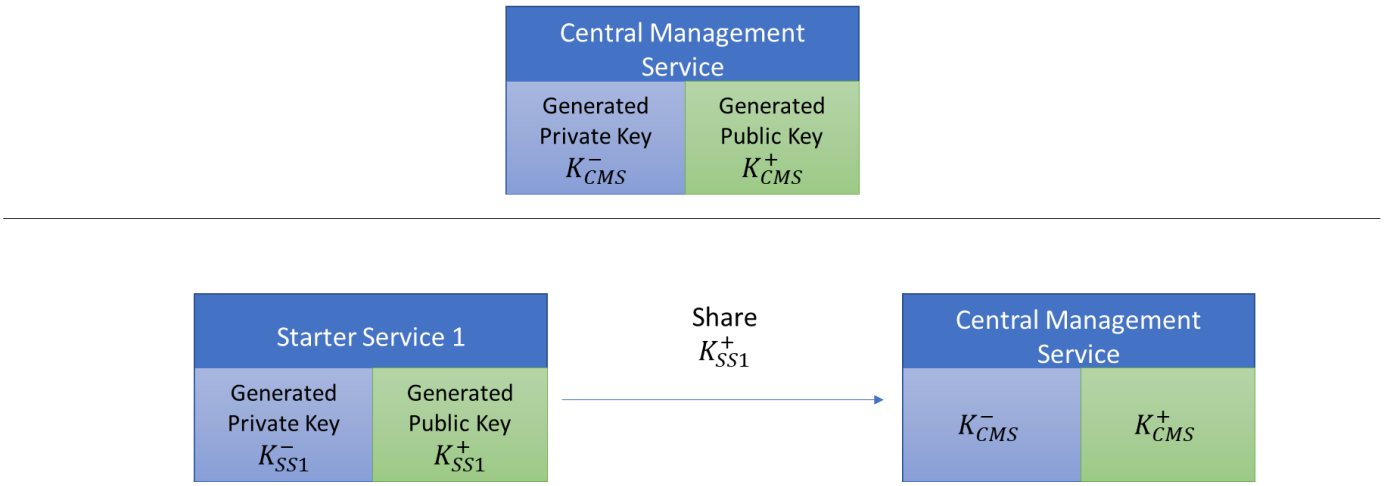

Figure 5.6: Top: Initially the Central Management Service generates public and private keys when it is started. Bottom: A new Starter Service instance generates its own public and private key and shares the public key with the Central Management Service automatically.

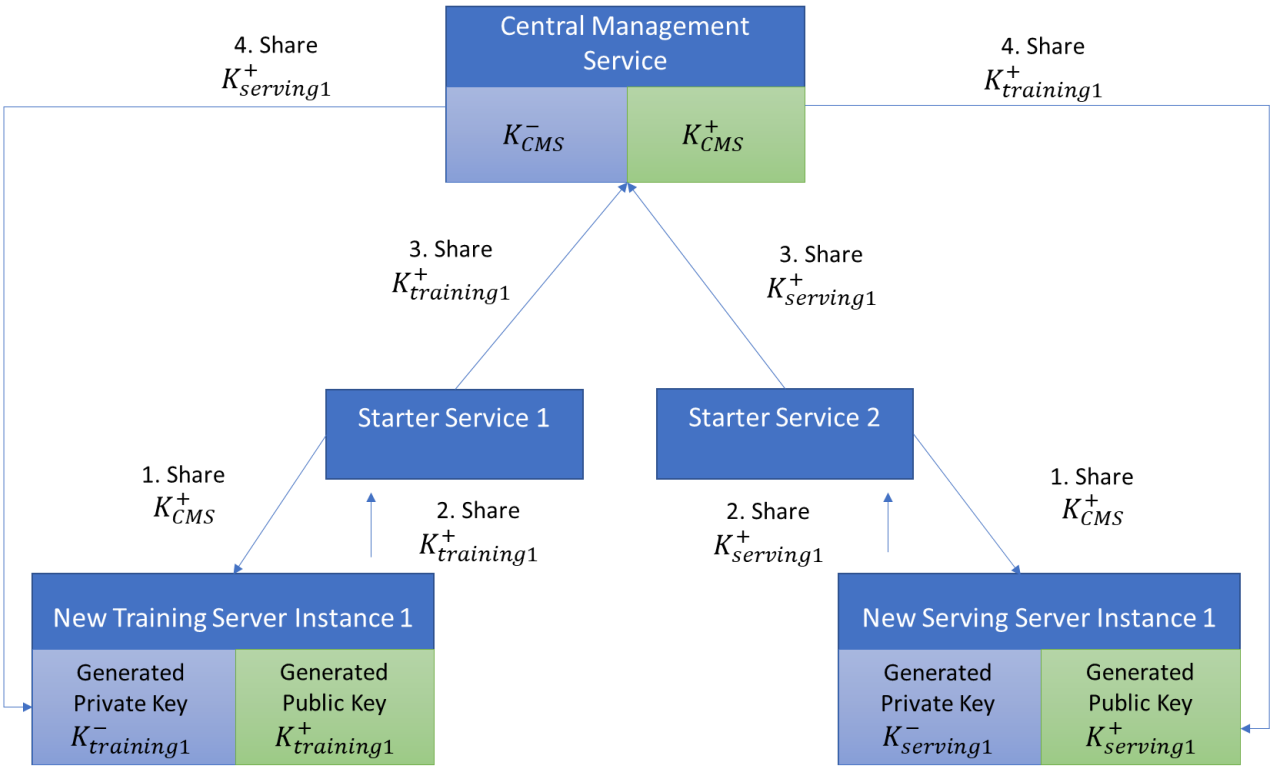

Figure 5.7: This diagram shows how the keys are shared when new training and serving instances are created. Each training and serving instance automatically generates its own public and private keys.

Initially, when the Central Management Service starts running, it automatically generates its own public and private key pair. Whenever a Starter Service instance starts running on a particular machine, it is initialized with this public key of the CMS and the hostname, port of the CMS. The Starter Service instance automatically registers with the CMS. This is illustrated in Figure 5.6.

When a new stream is created, the Central Management Service uses the Starter Service instances to initialize a new training instance and a new serving instance. When these start running, they each automatically generate their own public private key pairs individually. The Starter Service instances also provide them with the public key of the CMS. Then the Starter Service instances together with the CMS share the public keys and other information about the training and serving instances, like the IP and port, with one another before they can communicate with each other. This is illustrated in Figure 5.7.

The client directly communicates with the training serving instances securely using the public keys which it gets from the Central Management Service when the training and serving instances are created.

Chapter 6

Conclusion

This thesis presented a novel Machine Learning Layer specifically for the IoT setting. Our system features a tightly coupled training and serving architecture, that creates an autonomous feedback loop and enables the models to continuously improve based on feedback. We presented details about our implementation, which introduced and integrated many different components, techniques, and algorithms together in a unique way that makes the system adaptive, dynamic, and well suited to handle the use cases that arise in the Internet of Things.

The Machine Learning Layer not only allows for continuous learning, but is also designed to be flexible and general purpose. We are able to demonstrate the results of the system which show that it can learn over time from environmental data to create accurate localized models over time. The system also is able to the use environmental data to create models that are more efficient during serving. The completed system delivers the capabilities in a fully managed service that can be used seamlessly by any application and any IoT environment to deliver highly accurate and adaptive results.

6.1 Future Work

There are several possible avenues for future work. These include:

- Deploying the Machine Learning Layer on a large scale in many different types of environments
    - In large office buildings with many different sensors and many different prediction tasks involving employee activity
    - In smart homes for personal use and extending to smart cities
    - In smart cars and other smart transportation, etc.

- Completely edge machine learning deployments in which the serving instances only run on small devices like sensor platforms
- Adding more fault tolerance capabilities for example migrating training and serving instances to other machines, if a machine goes down
- Adding support for revoking keys if keys get compromised
- Running deployments in which the system scales across large numbers of machines in a high-performance server machine learning setting

- Learning Temporal Models
    - Extending the model training to take into account temporal variations by developing separate models for different temporal contexts
- Learning Correlations Between Prediction Tasks
    - Identifying correlations that exist between different prediction tasks for different sets of events, which could be useful in ensuring the robustness/confidence of predictions by allowing cross verification for correlated predictions

Bibliography

[1] Smart Cities are Getting Smarter, But Challenges Remain. 2019. Retrieved from https://www.roboticsbusinessreview.com/regional/smart-cities-getting-smarter-challenges-remain/.

[2] Leading the IoT. Gartner Insights on How to Lead in a Connected World. 2017. Retrieved from https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf.

[3] Smart Cities – What's In It For Citizens. 2018. Retrieved from https://newsroom.intel.com/wp-content/uploads/sites/11/2018/03/smart-cities-whats-in-it-for-citizens.pdf.

[4] Human-Centric Smart Cities. 2017. Retrieved from https://www.cisco.com/c/dam/assets /sol/sp/ultra-services-platform/hr-cisco-smart-cities-wp-10-27-17.pdf.

[5] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495, New York, NY, USA, 2017. ACM.

[6] Cloud Machine Learning Engine. 2018. Retrieved from https://cloud.google.com/ml-engine/.

[7] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

[8] Dan Crankshaw and Joseph Gonzalez. Prediction-serving systems. *Queue*, 16(1):70:83–70:97, February 2018.

[9] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182. ACM, 2014.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[11] Google Cloud Platform. 2018. Retrieved from https://cloud.google.com/products/#internet-of-things.

[12] Microsoft Azure. 2018. https://azure.microsoft.com/en-us/overview/iot/.

[13] AWS IoT. 2018. Retrieved from https://aws.amazon.com/iot/.

[14] Gierad Laput, Karan Ahuja, Mayank Goel, and Chris Harrison. Ubicoustics: Plug-and-play acoustic activity recognition. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, pages 213–224. ACM, 2018.

[15] Gierad Laput, Yang Zhang, and Chris Harrison. Synthetic sensors: Towards general-purpose sensing. *In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3986–3999. ACM, 2017.

[16] Martin Ester, Hans-Peter Kriegel, Jorg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[17] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[18] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (5):603–619, 2002.

[19] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[20] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[21] Image credit https://cdn-images-1.medium.com/max/1600/1*PF8XTtgVm1UYTuRc3U2ePQ.jpeg.

[22] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811, 2018.*

[23] Athinodoros S Georghiades, Peter N Belhumeur, and David J Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. IEEE Transactions on Pattern Analysis & Machine Intelligence, (6):643–660, 2001.

[24] Shawn Hershey, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, Channing Moore, Manoj Plakal, Devin Platt, Rif A. Saurous, Bryan Seybold, Malcolm Slaney, Ron Weiss, and Kevin Wilson. Cnn architectures for large-scale audio classification. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2017.

[25] IT Jolliffe. Principal component analysis. 1986. *Spring-verlag, New York*, 2:29, 1986.

[26] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[27] Vin D Silva and Joshua B Tenenbaum. Global versus local methods in nonlinear dimensionality reduction. In *Advances in neural information processing systems*, pages 721–728, 2003.

[28] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426, 2018.*

[29] Bernhard Scholkopf, Alexander J. Smola, and Klaus-Robert Muller. Advances in kernel methods. chapter Kernel Principal Component Analysis, pages 327–352. MIT Press, Cambridge, MA, USA, 1999.

[30] Sebastian Mika, Bernhard Scholkopf, Alex J Smola, Klaus-Robert Muller, Matthias Scholz, and Gunnar Ratsch. Kernel pca and de-noising in feature spaces. In *Advances in neural information processing systems*, pages 536– 542, 1999.

[31] Uniform Manifold Approximation and Projection. 2018. Retrieved from https://github.com/lmcinnes/umap.

[32] Python. 2019. Retrieved from https://www.python.org/.

[33] gRPC. 2019. Retrieved from https://grpc.io/.

[34] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al.

Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[35] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.