Formalized Generalization Bounds for Perceptron-Like Algorithms

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Robin J. Kelby

August 2020

This thesis titled

Formalized Generalization Bounds for Perceptron-Like Algorithms

by

ROBIN J. KELBY

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Gordon Stewart

Assistant Professor of Electrical Engineering and Technology

Mei Wei

Dean, Russ College of Engineering and Technology

Abstract

KELBY, ROBIN J., M.S., August 2020, Computer Science

Formalized Generalization Bounds for Perceptron-Like Algorithms (38 pp.)

Director of Thesis: Gordon Stewart

Machine learning algorithms are integrated into many aspects of daily life. However, research into the correctness and security of these important algorithms has lagged behind experimental results and improvements. My research seeks to add to our theretical understanding of the Perceptron family of algorithms, which includes the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron algorithms.

In this thesis, I will describe three variants of the Kernel Perceptron algorithm and provide both proof and performance results for verified implementations of these algorithms written in the Coq Proof Assistant. This research employs generalization error, which bounds how poorly a model may perform on unseen testing data, as a guarantee of performance with proofs verified in Coq. These implementations are also extracted to the functional language Haskell to evaluate their generalization error and performance results on real and artificial data sets.

Acknowledgments

Acknowledge later

Table of Contents

List of Figures

1 Introduction

The field of machine learning research has advanced very quickly in the past decade. Machine learning describes the class of computer programs that automatically learn from experience, often employed for classification, recognition, and clustering tasks. One of the classic problems in machine learning is handwritten digit recognition to classify written numbers automatically. Computers have historically struggled to interpret handwritten information because handwriting can vary drastically between writers. While humans can be taught to read as well as learn to read on their own, handwriting recognition can be challenging for computers to accomplish. Several datasets have been created specifically for the problem of handwriting analysis for numerical digits. For example, the MNIST dataset [LBBH98] is one of the primary datasets for computers to learn how to classify handwritten digits into the numbers 0-9. This dataset allows researchers to compare the performance of multiple models, trained and tested on the same data, but using different machine learning algorithms. Some systems have achieved a near-perfect performance on the MNIST dataset for the problem of handwritten digit classification, and this technology is valuable for processing documents, such as ZIP codes on letters sent through the U.S. Postal Service.

Increasingly, machine learning has been heavily integrated into our daily lives. As described by the authors of "Social media big data analytics", social media companies such as Facebook, Twitter, and YouTube learn from our digital data in order to serve individuals with targeted information and often advertising [GHHA19]. Retailers track customer purchases to learn about individuals' habits and entice them with specific offers and coupons. The pages we visit, profiles we create, and products we buy are used to predict our future actions and monetize our attention. This kind of task would be almost impossible for a human to complete, due to the vast amounts of data involved per person or account. In addition to social media, retailers, and advertisers, machine learning

techniques are also being employed in critical systems, such as healthcare and infrastructure, where failure can lead to the loss of time, money, and lives. Research to evaluate the use and oversight of machine learning algorithms [Var16] has shown that there are few existing safety principles and regulations for critical systems that rely on machine learning components. Machine learning drives more than websites and commerce; its algorithms are also responsible for the well-being and safety of people around the world, and regulation has largely not caught up with machine learning advances.

The development of machine learning tends to be experimentally driven in most applications. New or finely tuned configurations for internal components can lead to increased accuracy and efficiency or decreased training time compared to other algorithms for a specific problem or dataset. Such refinements can have enormous impacts for researchers studying machine learning problems and algorithms. The process of machine learning differs from algorithm to algorithm, but for most methods, machine learning algorithms learn models from training data to encode the program's knowledge. Models consist of learned parameters, which represent different kinds of data depending on the encoding of the model, and hyperparameters, a small number of variables directly specified by the programmer. Learned models are able to take a new piece of data as input and produce a result or judgment from that data. In the case of handwritten digits, the input to the model is the handwritten digit, and the output is the classification of that digit as a number from 0-9.

Machine learning algorithms can produce models that have millions of learned parameters, and small changes to model training, configuration, or hyperparameters can have enormous impacts on performance. Because of the complexity of the models produced by many machine learning algorithms, most new papers published in the field describe results found through experimentation, as opposed to examining the underlying

theory responsible for these advances. Additional research in understanding the theory behind machine learning may help to understand why some techniques are better suited for some problems than others, as well as potential avenues for exploration.

Finding errors in machine learning algorithms or models can be very difficult. With thousands or millions of parameters learned by the computer, not specified by the programmer, algorithms can easily get stuck in small, local solutions instead of finding the optimal solution. For example, gradient descent is an algorithm tasked with finding the lowest, or global, minimum of a multi-dimensional hillside with many peaks and valleys. Through many iterations, gradient descent travels downward along the gradient until a place is reached where descent is no longer possible. If the algorithm cannot find a deeper valley, this depth is returned as the overall solution. However, gradient descent can fail to find the global solution when the hyperparameters are not tuned correctly by the programmer or deeper valleys take too long to find. Techniques have been developed to mitigate the limitations of gradient descent, such as momentum, but the programmer usually has to experiment with multiple techniques to achieve peak performance. Additionally, few machine learning algorithms have theoretical properties that can be verified, such as a theorem that a learning algorithm will always terminate or find the global solution. Research into verifying machine learning to produce models with optimal behavior is limited due to these difficulties.

One way to increase our knowledge in the theory of machine learning is to verify the correctness of machine learning algorithms through mathematical proofs. Formal verification often entails machine-checked proofs of correctness, where software is built or translated into a proof assistant, such as Coq. Proof assistants allow for the integration of proofs with software specifications and implementations. Mathematical proofs in Coq are guaranteed to be as valid as the proof assistant itself, and because these proofs are portable programs, access to the proofs can allow others to verify proofs as well. Because

implementations are written in the same environment as their proofs, the proofs directly correspond to the implementation verified. The Coq environment also provides libraries containing both implementations of data structures and proofs to aid in the development of verified systems.

Researchers have used the Coq proof assistant to verify many different software systems and prove correctness properties. The CompCert compiler for the C language [Ler09] is the first verified compiler, proving that the behavior of a C program compiled with CompCert will not be changed in the transformation of compilation. Verified compilers ensure that the executable program produced by the compiler does not contain errors produced in compilation. For safety-critical applications, executables created by a verified compiler are more secure than executables created by unverified compilers. Another verified system written in Coq is Verdi [WWP+15], a framework for specifying and implementing distributed systems with tolerance for node faults. In a network of computers, connections can be dropped, packets lost or sent out of order, and nodes can fail or restart. Verdi allows the programmer to specify the fault conditions their distributed system should be resilient against, and the Verdi system itself mechanizes much of the proof process and code extraction for deployment in real-world networks. Distributed system software written with Verdi has been verified to handle faults and errors that may occur. Finally, Coq has also been used to implement microkernels, which are the basis for operating systems. The CertiKOS project [GSC+16] has developed several microkernels with security properties and proofs of correctness, including mC2, a verified concurrent microkernel. Operating systems allocate memory and computer resources and must defend against malicious processes. As demonstrated by these research projects, the Coq proof assistant can be extended for a diverse range of verified systems.

In this thesis, I will describe my additions to the verification framework MLCert. Building on the Perceptron implementation and existing proofs in MLCert, I present a

verified implementation of the Kernel Perceptron algorithm, as well as two variants on the Kernel Perceptron algorithm: a Budget Kernel Perceptron and a Description Kernel Perceptron. Background information for this thesis is provided in Chapter 2, with an introduction to the Perceptron and Kernel Perceptron algorithms, a more extended discussion of the challenges and tactics of machine learning verification, and the specifications for Budget Kernel Perceptrons and Description Kernel Perceptrons. Chapter 3 describes the methodology for implementing these algorithms in Coq. The proofs for these implementations and their performance results are detailed in Chapter 4. Finally, future work and conclusions are discussed in Chapter 5.

2 Background

This chapter aims to provide necessary background information in order to understand the remainder of this thesis. Sections 2.1 and 2.2 describe the Perceptron algorithm and its descendant, the Kernel Perceptron algorithm. Next, the challenges and methods of formal verification of machine learning are discussed in sections 2.3 and 2.4. Finally, modifications of the Kernel Perceptron algorithm, such as Budget Kernel Perceptrons in section 2.5 and Description Kernel Perceptrons in section 2.6, are detailed as improvements for the Kernel Perceptron.

2.1 The Perceptron Algorithm

The Perceptron algorithm was initially published in 1957 by Frank Rosenblatt. Highly influential in the early growth and development of the field of artificial intelligence, the Perceptron [Ros57] provided one of the first methods for computers to iteratively learn to classify data into discrete categories. In order to classify n-dimensional data, the Perceptron learns a weight vector with n parameters as well as a bias term. Both the weight vector and bias consist of positive integers greater than or equal to zero which encode a linear hyperplane separating two or more categories in n-dimensional space.

Figure 2.1: Perceptron Pseudocode

```
Definition Perceptron (w:Params) (epochs:nat) (training_set:list (Label * Data)) :
for i in epochs:
    for j in size(training_set):
        (example, true_label) = training_set[j]
        predict = Predict(example, w)
        if predict != true_label:
            w = Update(w, training_set[j]).
```

The most basic Perceptron algorithm has the following steps. Before training, each parameter in the weight vector w is initialized to zero. The algorithm consists of two nested loops, as shown by the pseudocode in Figure 2.1. For this algorithm, we require the weight vector, the number of epochs, and the training set as input. The training set consists of labeled training examples, where the label is either 0 or 1. The outer loop uses the number of epochs to control the number of iterations over the entire training set. The inner loop executes for every training example in the training set and has two main steps. First, the n-dimensional data inside the training example and the weight vector are used to calculate the Perceptron's predicted label for this example, without using the training example's true label. The calculation for Perceptron prediction is shown in pseudocode in Figure 2.2 takes as input the weight vector and a single training example to produce a predicted label for the given example.

Figure 2.2: Perceptron Prediction Pseudocode

```
Definition Predict (example:Data) (w:Params):
    (bias, weight) = w
    bias + dot_product(weight, example).
```

The true label and the calculated predicted label are then compared. If both labels are the same, the Perceptron correctly classified this training example. However, if the predicted label is different, the weight vector is updated using the example to improve classification over time. This update is the second step of the inner loop.

The Perceptron algorithm is powerful despite its simplicity. However, there are limitations to the Perceptron's classification. The Perceptron cannot classify data that is not linearly separable with 100% accuracy, such as points classified by the exclusive-OR function, a binary operator that returns TRUE when its two inputs are the opposite of each

other. Despite the simplicity of exclusive-OR, the Perceptron cannot produce a model, or linear hyperplane, such that all the points classified by exclusive-OR as TRUE are also classified by the Perceptron as TRUE, and all the points classified by exclusive-OR as FALSE are also classified by the Perceptron as FALSE. The Perceptron can achieve at best 75% accuracy for the exclusive-OR function. This restriction on the Perceptron in part caused the first AI Winter, a severe decline in artificial intelligence research, due to unreasonable expectations for the Perceptron in fields where data is not linearly separable.

While the Perceptron is limited to classification of linearly separable data, the Perceptron Convergence Theorem states that the Perceptron is guaranteed to converge to a solution on linearly separable data. This property of the Perceptron algorithm was first proven on paper by Papert in 1961 [Pap61] and Block in 1962 [Blo62]. However, this proof was not verified by machine until 2017 through the work of Murphy, Gray, and Stewart [MGS17] in the Coq proof assistant.

2.2 The Kernel Perceptron

The Kernel Perceptron improved on the Perceptron algorithm with the introduction of the kernel trick by Aizerman, Braverman, and Rozoner [ABR64]. Using kernel functions, the classification of the Perceptron can be expanded to include non-linearly separable data. There are four main modifications for the Kernel Perceptron: prediction, kernel functions, parameter space, and weight vector update. Prediction for the Kernel Perceptron uses kernel functions to produce non-linear hyperplanes instead of linear hyperplanes. Because of kernalization, the prediction function changes so that in addition to the weight vector w and the current training example, the training set and training labels are required as well. The bias term is no longer necessary.

In the pseudocode KernelPredict function shown in Figure 2.3, K represents an arbitrary kernel function. Kernel functions form a class of functions that take two

Figure 2.3: Kernel Perceptron Prediction Pseudocode

```
Definition KernelPredict (example:Data) (w:KernelParams)
        (training_set:list (Label * Data) (K:Kernel):
    for i in size(training_set):
        (label, data) = training_set[i]
        sum += w[i] * label * K(example, data)
    return sum.
```

examples as input and produce a single value. By using non-linear kernel functions, the Kernel Perceptron can classify data that is not linearly separable. For example, the Kernel Perceptron can classify the exclusive-OR function with 100% accuracy using a quadratic kernel. By using kernel functions in prediction, the parameters used by the Kernel Perceptron have different cardinality compared to the parameters of the Perceptron. The Kernel Perceptron requires one parameter per training example for its classification, regardless of the dimensionality of the data. Therefore, the size of the weight vector is dependent on the size of the training set.

Finally, the weight vector update for the Kernel Perceptron is somewhat different from that of the Perceptron. When a training example is misclassified by the Kernel Perceptron, its parameter is incremented and the rest of the weight vector is unchanged. The full Kernel Perceptron algorithm is shown in Figure 2.4.

The Kernel Perceptron improves upon the Perceptron, but the Kernel Perceptron has its own limitations. The size of the parameter space for the Kernel Perceptron limits its usefulness in applications where memory is at a premium, as the size of the weight vector is dependent on the number of training examples, not the dimensionality of the training data. Also, the Kernel Perceptron, due to the use of kernel functions, is not guaranteed to

Figure 2.4: Kernel Perceptron Pseudocode

```
Definition KernelPerceptron (w:KernelParams) (epochs:nat)
        (training_set:list (Label * Data)) (K:Kernel):
for i in epochs:
    for j in size(training_set):
        (example, true_label) = training_set[j]
        predict = KernelPredict(example, w, training_set, K)
        if predict != true_label:
            w = Update(w, j).
```

converge to a solution or terminate, unlike the Perceptron algorithm. This means that the Perceptron Convergence Theorem cannot be used to prove the correctness of an implementation of the Kernel Perceptron.

2.3 Approaches to Machine Learning Verification

Verifying machine learning algorithms is a difficult problem in software engineering. Machine learning algorithms can produce thousands or millions of parameters in their models, which interact to classify data. The learning process for machine learning models can be tedious for humans to trace, and the model parameters generated during training are often not human-interpretable for manual verification of correctness. The authors of [BF16] describe how machine learning researchers do not agree on a standard definition of what human interpretability is or how models should be able to be interpreted by humans. Interpretability varies between algorithms and tends to be more difficult for neural algorithms, including the Perceptron family of algorithms. Some formal verification in the field of machine learning has been performed, as shown by [TD05], but

many algorithms have not been verified correct. Even for implementations with paper proofs of correctness, few have been proven correct by machine.

2.4 MLCert Framework

To facilitate the verification of machine learning algorithms, Bagnall and Stewart developed MLCert [BS19], an open-source tool built in the Coq proof assistant. MLCert employs generalization error to prove correctness for machine learning algorithms. Generalization error, as described by Levin, Tishby, and Solla [LTS90], is an important indicator for the robustness of a machine learning model; algorithms with low generalization error can generalize from the training examples used in training to correctly classify unseen examples from the same domain of data in testing. Instead of trying to verify the model directly, MLCert verifies the generalization bounds for machine learning implementations built in its framework. Bounds on the generalization error indicate that an algorithm has bounds on mistakes made during testing, and the size of the parameter space contributes heavily to the tightness of the generalization bounds. Verified generalization bounds guarantee worst-case performance for a model. Previous work in the MLCert framework [BS19] has resulted in an implementation of the Perceptron algorithm with proofs to verify its generalization bounds. However, to the best of our knowledge, no one has implemented the Kernel Perceptron in Coq or formally proven its correctness of generalization bounds using machine-checked proofs.

The parameter space for the Kernel Perceptron is dependent on the number of training examples. This means that, as compared to the Perceptron algorithm, the Kernel Perceptron has very loose generalization bounds due to the increased size of the parameter space. The tightness of the generalization bounds matters because tighter bounds provide a stronger guarantee for performance. To tighten the generalization bounds of the Kernel Perceptron, one approach is to limit the number of parameters.

2.5 Budget Kernel Perceptron Algorithms

Budget Kernel Perceptrons are a family of algorithms which modify the Kernel Perceptron to limit the size of the parameters for the model while minimizing the impact on the accuracy of the model. Budget Kernel Perceptrons are often employed in areas where computer memory or resources are at a premium, and their modifications are customized for the requirements of their field. One strategy for Budget Kernel Perceptrons is to keep a set number of training examples for classification called support vectors, with specific rules for updating this set over time to maintain its size as the classification boundary changes. For the base Kernel Perceptron algorithm described in Section 2.2, every training example is a support vector. An example of a budget update rule is described in the article "Tracking the best hyperplane with a simple budget Perceptron", where the authors describe an update procedure where one support vector is selected at random for each replacement [CCBG07]. Another update rule is to always select the oldest support vector for replacement, as this support vector may no longer be necessary for correct classification.

Other strategies minimize the impact of removed support vectors through more creative means. Dekel, Shalev-Shwartz, and Singer present the Forgetron, where each support vector is "forgotten" over time by decreasing its impact on the model, which means that there is always an oldest support vector to be removed with the least influence on the model [DSSS07]. Another set of strategies include the Projectron and Projectron++ algorithms described by [OKC09], which store both a support set and a projection onto the support set to reduce the overall size of the model. Both these methods balance model size with increased classification error compared to the base Kernel Perceptron.

Of these three studies, none discuss or provide proofs of their Budget Kernel Perceptron's generalization bounds. The nature and function of Budget Kernel Perceptrons complements our research in proving generalization error for machine

learning algorithms. By implementing a Budget Kernel Perceptron, the bounds on the size of the parameter space can improve the bounds on generalization error compared to the base Kernel Perceptron algorithm.

2.6 Description Kernel Perceptrons

In contrast to Budget Kernel Perceptrons, another method of encoding the Kernel Perceptron parameters involves description-length bounds. During training, the Kernel Perceptron will make a set number of mistakes, bounded by some value L. Using L, the number of support vectors is less than or equal to the number of mistakes, which will always be less than or equal to the size of the training set. This method requires a record of every misclassification made during training. Only training examples that were misclassified are included in the set of support vectors and used to calculate the hyperplane.

One approach for a Description Kernel Perceptron is described by Cramer, Kandola, and Singer [CKS03] in their paper, "Online Classification on a Budget." In their approach, when a misclassification is made, there are two phases: insertion and deletion. When a new example is misclassified, this example is inserted into the support set. The algorithm then examines the entire support set and searches for any redundant support vectors that are no longer necessary by examining the distance of each support vector from the decision hyperplane. Examples that are never misclassified are never added to the support set. This approach blends the Budget and Description Kernel Perceptron, as the authors prove on paper that the size of their support set is dependent on the margin for misclassification and distance from the hyperplane, which bounds both the number of mistakes and the overall size of the support set.

The generalization error for a Kernel Perceptron using description-length bounds is dependent on the number of misclassifications, which provides a bound on the size of the

parameter space. Cramer, Kandola, and Singer designed their algorithm with generalization error in mind and provide the generalization error of their algorithm on experimental data sets [CKS03]. The generalization bounds for a Description Kernel Perceptron should improve on the bounds for the base Kernel Perceptron as long as the number of mistakes is significantly less than the number of training examples.

2.7 Chapter Summary

This chapter summarizes the background of this thesis, discussing the Perceptron and Kernel Perceptron algorithms, as well as variants of the Kernel Perceptron algorithm with improved generalization bounds. Chapter 3 will next describe my extensions to the MLCert framework to implement three Kernel Perceptron algorithms: the base Kernel Perceptron algorithm, a Budget Kernel Perceptron, and a Description Kernel Perceptron, with generalization proofs for each implementation written in Coq.

3 METHODS

In this chapter, I will describe my methods for implementing the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron in the MLCert framework. First, Section 3.1 describes the pipeline in MLCert for building the specifications of machine learning algorithms. Next, Sections 3.2, 3.3, and 3.4 outline the Coq sections for each implementation, which consist of a prediction section and an section for the entire algorithm. The Coq sections for the proofs of these implementations and their extraction to Haskell follows in Chapter 4.

3.1 Structure of Perceptron Implementations in MLCert

The MLCert framework provides data structures and proofs that can be instantiated with the specifics of a machine learning algorithm, as well as extraction directives which facilitate the process of extracting Coq code to Haskell for execution. MLCert requires four sections in Coq for the complete implementation of a machine learning algorithm. Before discussing the four required sections, I will first give the structure and type signature MLCert uses to encode a machine learning algorithm. This module is located in the file "learners.v":

Figure 3.1: Learner Module

```
Module Learner.
Record t (X Y Hypers Params : Type) :=
  mk { predict : Hypers → Params → X → Y;
       update : Hypers → X∗Y → Params → Params }.
End Learner.
```

The Learner module defines the general form of parameters and functions for machine learning algorithms. Four types are listed in the definition of Learner.t: X, Y, Hypers, and Params. X is the type of the training data, and Y is the type of the training labels. The type of hyperparameters is Hypers, and finally the type of parameters is Params. These four types are used to define the type signatures for the required predict and update functions. A prediction function requires Hypers, Params, and a training example of type X to return the predicted label of type Y. The update function requires Hypers, an example paired with its label, and Params to return updated Params. This module will later be instantiated with specific types and functions to implement the Perceptron family of algorithms.

This format is not universal for all machine learning algorithms. For example, unsupervised learning algorithms do not use training labels, as with the k-means clustering algorithm. Because the predict function is meant to return a predicted label as classification, unsupervised algorithms would not be able to be implemented using Learner.t. However, Learner.t is sufficient for implementing the Perceptron family of algorithms because Perceptrons are supervised, using labeled training data, and rely on prediction and update for their classification.

When implementing a machine learning algorithm, the first required Coq section implements the prediction function according to the type signature of the predict function in Learner.t. The second section defines the update function for the machine learning algorithm, as well as instantiating Learner.t with the specific types, prediction, and update functions for the algorithm. This second section is directly used by the third and fourth sections. Generalization proofs in the third section prove the cardinality of the Params used by the algorithm as well as the generalization bounds for the entire algorithm. Finally, the fourth Coq section defines how the algorithm should be extracted, a process which translates the algorithm in Coq to the functional language Haskell for experimental

results. The rest of this chapter describes the first two Coq sections for each implementation.

3.2 Kernel Perceptron Coq Implementation

The Kernel Perceptron implementation in MLCert is located in the file "kernelperceptron.v". Section KernelClassifier contains the predict function for the Kernel Perceptron. The definition of kernel_predict is shown below:

Figure 3.2: kernel_predict function in KernelClassifier

```
Definition kernel_predict (K : float32_arr n → float32_arr n → float32)
        (w : KernelParams) (x : Ak) : Bk :=
    let T := w.1 in
      foldable_foldM
        (λ xi_yi r ⇒
          let: ((i, xi), yi) := xi_yi in
          let: (j, xj) := x in
          let: wi := f32_get i w.2 in
          r + (float32_of_bool yi) ∗ wi ∗ (K xi xj))
        0 T > 0.
```

The kernel_predict function takes three inputs: a kernel function K, the current Kernel Perceptron parameters w, and an example x of type Ak. The type Ak representing training data is defined in KernelClassifier as an index paired with an array of floating point values of size n. The type of labels Bk is defined as Boolean. In the kernel_predict function, the kernel function can be specified for one of several kernel functions.

KernelClassifier contains two kernel functions corresponding to the linear and quadratic kernels, as shown in Figure 3.3.

Figure 3.3: Kernel functions in KernelClassifier

```
Definition linear_kernel {n} (x y : float32_arr n) : float32 :=
    f32_dot x y.
Definition quadratic_kernel (x y : float32_arr n) : float32 :=
    (f32_dot x y) ** 2.
```

The KernelParams for the Kernel Perceptron are defined as the training set paired with a float array of size m, where m is the number of training examples. In the basic Kernel Perceptron, every training example in the training set is a support vector. The float array in KernelParams is used for the kernel_predict calculation of the training example x's label, as each float value corresponds to a support vector. The kernel_predict function folds over the support vectors in KernelParams so that for each support vector, the result of the kernel function applied to the support vector and x is multiplied with the float value for the support vector and the label for the support vector. The result of this calculation is compared with zero to return the predicted Boolean label for x.

The Coq section KernelPerceptron completes the Kernel Perceptron implementation, containing the kernel_update function and instantiating Learner.t with the Kernel Perceptron parameters and functions. The kernel_update function is defined in Figure 3.4. Using the kernel_predict function and kernel function K, kernel_update compares the predicted label to the actual label. If the predicted label is correct, the parameters p are returned without change. However, if the predicted label is incorrect, the float array is updated so that the float value for that training example is incremented by 1.

Figure 3.4: kernel_update function in KernelPerceptron

```
Definition kernel_update
    (K : float32_arr n → float32_arr n → float32)
        (h:Hypers) (example_label:A∗B) (p:Params) : Params :=
    let: ((i, example), label) := example_label in
    let: predicted_label := kernel_predict K p (i, example) in
    if Bool.eqb predicted_label label then p
    else (p.1, f32_upd i (f32_get i p.2 + 1) p.2).
```

With kernel_update implemented, Learner.t can be instantiated with the necessary types and functions. As the Kernel Perceptron does not use hyperparameters in its algorithm, the type Hypers is defined as the empty record. The Kernel Perceptron Learner can be defined as follows in Figure 3.5. This Learner definition is used in the other two Kernel Perceptron sections which will be discussed in Sections 4.1.1 and 4.2.1.

Figure 3.5: Learner Definition in KernelPerceptron

```
Definition Learner : Learner.t A B Hypers Params :=
  Learner.mk
      (λ _ ⇒ @kernel_predict n m support_vectors F K)
      (kernel_update K).
```

3.3 Budget Kernel Perceptron Coq Implementation

The Budget Kernel Perceptron is also located in the file “kernelperceptron.v” in MLCert. The predict function is located in the section KernelClassifierBudget, and

modifies the Kernel Perceptron predict function so that a budget on the size of the set of support vectors can be enforced. In KernelClassifierBudget, the variable sv is the size of the support set. As opposed to the KernelParams which contain the entire training set and a float array, the parameters for the Budget Kernel Perceptron are built as an axiomatized vector of size sv containing support vectors paired with a float value for that support vector. The definition of the type of support vectors and the type of the support set are given in Figure 3.6.

Figure 3.6: Support Vector Definitions in KernelClassifierBudget

Definition bsupport_vector: Type := Akb $*$ Bkb.

Definition bsupport_vectors: Type := AxVec sv (float32 $*$ (bsupport_vector)).

The predict function for the Budget Kernel Perceptron shown in Figure 3.7 is very similar to Kernel Perceptron prediction, with the main difference being the size of the support set. Again, the kernel function K used in prediction can be specified as any kernel function with the correct type signature. Like kernel_predict, kernel_predict_budget folds over the support set with the same calculation as in kernel_predict. However, the type of the training data, Akb, is different for the Budget Kernel Perceptron. Akb is defined as a float array of size n, which is the dimensionality of the data, and does not include an index for that example as it is not necessary for Budget classification.

The Budget Kernel Perceptron implementation is located in the module KernelPerceptronBudget, and this module contains several functions necessary for the budget update rule to maintain the size of the support set. The update rule for the Budget Kernel Perceptron is more complex than for the Kernel Perceptron. If the current example has been misclassified, we first need to determine if this example is already a support vector. If the example is a support vector, then the float value associated with this support

Figure 3.7: kernel_predict_budget function in KernelClassifierBudget

```
Definition kernel_predict_budget
        (w: bsupport_vectors)
        (x: Akb) : Bkb :=
    foldable_foldM
        (λ wi_xi r ⇒
            let: (wi, (xi, yi)) := wi_xi in
            r + (float32_of_bool yi) ∗ wi ∗ (K xi x))
        0 w > 0.
```

vector should be incremented by one. However, if this example is not a support vector, then we need to add this example to the support set and remove a support vector. As discussed in Section 2.5, there are several methods for selecting the support vector to be removed. In our implementation, we choose the oldest support vector, as removing this support vector will likely have the least impact on the decision hyperplane. When a new example is added to the support set, it is added on to the front of the vector, while the support vector at the end of the vector is removed. This replacement procedure ensures that the oldest support vector is always stored at the end of the vector for safe removal. The logic for the kernel budget update rule is defined in the function budget_update shown in Figure 3.8. In the update function for the Budget Kernel Perceptron called kernel_update shown in Figure 3.9, the update rule used is given as an argument to kernel_update so that the budget update rule can be changed.

Finally, Learner.t can be instantiated using kernel_predict_budget and kernel_update. No hyperparameters are used for the Budget Kernel Perceptron, again implemented as the

Figure 3.8: budget_update function in KernelPerceptronBudget

```
Definition budget_update (p: Params) (yj: A∗B): Params :=
    if existing p yj then upd_weights p yj.1
    else add_new p yj.
```

Figure 3.9: kernel_update function in KernelPerceptronBudget

```
Definition kernel_update
    (K : float32_arr n → float32_arr n → float32)
        (h:Hypers) (example_label:A∗B) (p:Params) : Params :=
    let: (example, label) := example_label in
    let: predicted_label := kernel_predict_budget K p example in
    if Bool.eqb predicted_label label then p
    else (U p example_label).
```

empty record. Figure 3.10 shows the Budget Kernel Perceptron's Learner definition that is used in the proof and extraction sections of 4.1.2 and 4.2.2, respectively.

Figure 3.10: Learner Definition in KernelPerceptronBudget

```
Definition Learner : Learner.t A B Hypers Params :=
    Learner.mk
        (λ _ ⇒ @kernel_predict_budget n (S sv) K F)
        (kernel_update K).
```

3.4 Description Kernel Perceptron Coq Implementation

This is a placeholder until the Description Kernel Perceptron is implemented.

3.5 Chapter Summary

This chapter describes the implementation of the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron in Coq. Each of the sections for these algorithms conform to the specifications given by the module Learner.t to ensure that these implementations can be proved to have generalization bounds and extracted to Haskell, which will be discussed in Chapter 4.

4 Proofs and Experimental Results

The results of my research consist of proofs and experimental performance for the implementations described in Chapter 3. This chapter consists of two sections, one for generalization proofs and the other describing extraction and performance. The Kernel Perceptron results can be found in sections 4.1.1 and 4.2.1. Sections 4.1.2 and 4.2.2 provide the Budget Kernel Perceptron results. Finally, the Description Kernel Perceptron results are located in sections 4.1.3 and 4.2.3. The conclusions and future work for this research follow in Chapter 5.

4.1 Generalization Proofs

In the MLCert framework, much of the proof burden has been automated. For a new Learner representing a machine learning algorithm, there are two new lemmas that need to be proved. The first lemma proves the cardinality of the parameters used by the algorithm, which corresponds to the size of the parameter space. The second lemma applies the first in order to prove a generalization bound for the Learner as a whole. An example of the second lemma for a generic Learner is shown in Figure 4.1.

Figure 4.1: Generalization Bound for a generic Learner

```
Lemma Learner_bound eps (eps_gt0 : 0 < eps) init :
  @main A B Params Hypers Learner
    hypers m m_gt0 epochs d eps init (λ _ ⇒ 1) ≤
  #|Params| ∗ exp (−2%R ∗ eps^2 ∗ mR m).
```

The definition of main which is used in Figure 4.1 can be found in the file "learners.v". Once main has been instantiated with the specifics of the Learner, such as its particular Params and Hypers, there are proofs in "learners.v" such as the lemma

main_bound which provide the machinery necessary to prove this inequality over the real numbers. As described by Bagnall and Stewart [BS19], MLCert uses Hoeffding's inequality, a type of Chernoff bound, to prove the generalization bound for a Learner. In the following subsections, the lemmas proving the generalization bounds for the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron will be discussed.

4.1.1 Kernel Perceptron Generalization Proofs

The bound for the Kernel Perceptron relies on the size of the parameter space. As proven in the lemma K_card_Params in the section KernelPerceptronGeneralization, the cardinality of the parameters for the Kernel Perceptron is shown in Equation 4.1. This power of 2 is calculated by unfolding the definition of Params, which consists of the training set and a float array of size m. As all values are floating point numbers stored in 32 bits, the cardinality of a single floating point number is 2^{32}. Therefore, as the dimensions of the training set are equal to m training examples multiplied by n dimensions, the cardinality of the training set is equal to 2^{m*n*32}. The cardinality of a float array of size m is 2^{m*32}.

$$\#|Params| = 2^{(m*n*32+m*32)} \tag{4.1}$$

Kcard_Params is central to the proof of the generalization bounds of the Kernel Perceptron in the lemma KPerceptron_bound, which is defined in Figure 4.2. The generalization bound for the Kernel Perceptron is very loose, as growth in the size of the training set causes exponential growth in the generalization error. This limits the usefulness of the Kernel Perceptron's generalization bound, as a loose generalization bound provides few guarantees for performance or correctness. However, the Kernel

Perceptron bound allows for comparison between this result and the generalization bounds for the Budget Kernel Perceptron and the Description Kernel Perceptron.

Figure 4.2: Generalization Bound for the Kernel Perceptron

Lemma Kperceptron_bound eps (eps_gt0 : 0 < eps) init :

@main A B Params KernelPerceptron.Hypers

(@KernelPerceptron.Learner n m KPsupport_vectors H K)

hypers m m_gt0 epochs d eps init (λ _ $\Rightarrow$ 1) $\leq$

2^(m∗n∗32 + m∗32) ∗ exp (−2%R ∗ eps^2 ∗ mR m).

4.1.2 Budget Kernel Perceptron Generalization Proofs

The Budget Kernel Perceptron has a similar bound on the cardinality of the parameter space. However, the parameter space for the Budget Kernel Perceptron is not dependent on m, the size of the training set, whatsoever. Instead, the parameter space relies on (S sv), which is the size of the support set. The successor of sv is used to denote the size of the support set so that the budget update procedure is always possible regardless of the value of sv, as there will be at least one support vector able to be replaced.

Like the Kernel Perceptron, the Budget Kernel Perceptron stores the support set and a float array. The float array is of size (S sv), so its cardinality is $2^{32*(S\ sv)}$. The support set stores (S sv) training examples, which consist of one float value for each of the n dimensions of the data, plus a Boolean value for the label of the support vector. Therefore, the cardinality of each training example in the support set is 2^{1+n*32}. The full cardinality of the Budget Kernel Perceptron Params is given in Equation 4.2. The lemma proving this bound is found in the section KernelPerceptronGeneralizationBudget, named Kcard_Params_Budget.

$$\#|Params| = 2^{((32*(S\ sv)+((1+n*32)*(S\ sv))))} \tag{4.2}$$

Figure 4.3 shows the lemma for the generalization bound of the Budget Kernel Perceptron, which uses Kcard_Params_Budget in its proof. Comparing the bound of the Budget Kernel Perceptron to the Kernel Perceptron, the overall structure of the two bounds is similar when the number of training examples is the same. However, because the support set can be significantly smaller than the number of training examples, the Budget Kernel Perceptron's bound is tighter than that of the base Kernel Perceptron.

Figure 4.3: Generalization Bound for the Budget Kernel Perceptron

```
Lemma Kperceptron_bound_budget eps (eps_gt0 : 0 < eps) init :
  @main A B Params KernelPerceptronBudget.Hypers
    (@KernelPerceptronBudget.Learner n sv F K U)
    hypers m m_gt0 epochs d eps init (λ _ ⇒ 1) ≤
  INR 2^((32*(S sv) + ((1 + n * 32)*(S sv)))) * exp (-2%R * eps^2 * mR m).
```

4.1.3 Description Kernel Perceptron Generalization Proofs

This is a placeholder until the Description Kernel Perceptron is implemented.

4.2 Haskell Performance

In order for the Coq implementations to be run, these implementations must be extracted to Haskell. The file "extraction_hs.v" contains extraction directives for Haskell so that some Coq functions and data structures are extracted properly. The last Coq module for each implementation uses the extractible_main definition, found in

"learners.v", to also provide the necessary machinery that Learner.t relies on. The extracted Coq code is written to a Haskell file located in the directory hs in MLCert.

The extracted Haskell code for a machine learning algorithm does not contain code to initialize the system with training and testing data or functions to display accuracy and generalization error results to the user. Unverified Haskell drivers have been written for these implementations, which include the extracted Haskell code as a module. The Haskell drivers for the Kernel Perceptron implementations can also be found in the hs directory.

4.2.1 Kernel Perceptron Haskell Extraction

The extraction directives for the Kernel Perceptron can be found in the section KPerceptronExtraction in the file "kernelperceptron.v". This section extracts the Kernel Perceptron to the Haskell file "KPerceptron.hs" in the hs directory. There are two Haskell driver files for the Kernel Perceptron. The first driver is a small driver to test that the Kernel Perceptron using a quadratic kernel can classify the XOR function with 100% accuracy. The linear kernel cannot be used because the XOR function is not linearly separable. The four samples for this function are specified in the driver, along with the quadratic kernel for the prediction function. When run, this driver demonstrates that the Kernel Perceptron behaves as expected with the quadratic kernel and is able to classify data that is not linearly separable.

The second driver, "KPerceptronTest.hs", is a more general test of the Kernel Perceptron. For this driver, the dimensionality of the data is three, and each of the three values is randomly generated in the range (-1.0, 1.0). The number of training examples is 200, and one sample is chosen as the hyperplane which is used to label each sample as either True or False. The Kernel Perceptron runs for five epochs.

4.2.2 Budget Kernel Perceptron Haskell Extraction

The section KPerceptronExtractionBudget in “kernelperceptron.v” contains the extraction directives for the Budget Kernel Perceptron, which extracts the Budget Kernel Perceptron to the Haskell file “KPerceptronBudget.hs”. The driver file “KPerceptronBudgetTest.hs” is similar to the second driver of the Kernel Perceptron. However, while the dimensionality, number of training examples, and epochs are kept the same, the number of support vectors is specified as 20, limiting the size of the support set to 10% of the training set. The support set is initialized to zero vectors, which will be removed as the Budget Kernel Perceptron makes classification errors during training.

4.2.3 Description Kernel Perceptron Haskell Extraction

This is a placeholder until the Description Kernel Perceptron is implemented.

4.3 Chapter Summary

These results from the implementations of the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron demonstrate that the generalization error for the Kernel Perceptron can be improved through limiting the size of the support set or placing a limit on the number of mistakes made during training. The conclusions drawn from these results are described in Chapter 5, along with a discussion of future work to be done in the field of machine learning verification.

5 Conclusions

References

[ABR64] M. A. Aizerman, E. M. Braverman, and L. I. Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[BF16] Adrien Bibal and Benoit Frénay. Interpretability of machine learning models and representations: an introduction. In *ESANN'16 Proceedings*, Bruges, 2016.

[Blo62] Hans-Dieter Block. The perceptron: a model for brain functioning. *Reviews of Modern Physics*, 34(1):123, 1962.

[BS19] Alexander Bagnall and Gordon Stewart. Certifying the true error: machine learning in coq with verified generalization guarantees. In *Proceedings of AAAI'19*, pages 2662–2669, Hawaii, 2019.

[CCBG07] Giovanni Cavallanti, Nicolo Cesa-Bianchi, and Claudio Gentile. Tracking the best hyperplane with a simple budget perceptron. *Machine Learning*, 69(23):143–167, 2007.

[CKS03] Koby Crammer, Jaz Kandola, and Yoram Singer. Online classification on a budget. In *Advances in Neural Information Processing Systems 16*. Proceedings of NIPS 2003, 2003.

[DSSS07] Ofer Dekel, Shai Shalev-Shwartz, and Yoram Singer. The forgetron: a kernel-based perceptron on a budget. *SIAM Journal on Computing*, 37(5):1342–1372, 2007.

[GHHA19] Norjihan A. Ghani, Suraya Hamid, Ibrahim A. T. Hashem, and Ejaz Ahmed. Social media big data analytics. *Computers in Human Behavior*, 101:417–428, 2019.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjoberg, and David Costanzo. Certikos: an extensible architecture for building certified concurrent os kernels. In *OSDI'16*, pages 653–669, Savannah, Georgia, 2016.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86(11), pages 2278–2324, 1998.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009.

[LTS90] Esther Levin, Naftali Tishby, and S. A. Solla. A statistical approach to learning and generalization in layered neural networks. In *Proceedings of the IEEE*, volume 78, pages 1568–1574, 1990.

[MGS17] Charlie Murphy, Patrick Gray, and Gordon Stewart. Verified perceptron convergence theorem. In *MAPL'17*, Barcelona, 2017.

[OKC09] Francesco Orabona, Joseph Keshet, and Barbara Caputo. Bounded kernel-based online learning. *Journal of Machine Learning Research*, 10(11):2643–2666, 2009.

[Pap61] Seymour Papert. Some mathematical models of learning. In *Proceedings of the Fourth London Symposium on Information Theory*, 1961.

[Ros57] Frank Rosenblatt. The perceptron, a perceiving and recognizing automaton. *Report: Cornell Aeronautical Laboratory*, 58(460), 1957.

[TD05] Brian. J. Taylor and Marjorie A. Darrah. Rule extraction as a formal method for the verification and validation of neural networks. In *Proceedings of IEEE International Joint Conference on Neural Networks 2005*, pages 2915–2920, Montreal, 2005.

[Var16] Kush R. Varshney. Engineering safety in machine learning. In *2016 Information Theory and Applications Workshop*, La Jolla, California, 2016.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI'15*, pages 357–368, Portland, Oregon, 2015.