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 Computer Science

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 (70 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 synthetic data sets.

Acknowledgments

In this opportunity to sit and think back on all those who have made this thesis possible, I am incredibly grateful. Dr. Gordon Stewart has been a faithful advisor for my research, and his insight and care has been instrumental in my success. His encouragement in research was a major factor in my decision to pursue my Master's degree and I am thankful for his time and attention over the years.

I would also like to thank Dr. David Juedes, Dr. Razvan Bunescu, and Dr. Peter Jung for serving on my committee. I am grateful for their input and analysis of my work and how they have pushed me to do my best.

This thesis would not be possible without a research team that has given me support and focus over the past two years. Nathan St. Amour, David Masters, and Tim Steinberger have been fantastic lab partners and I am glad to have worked closely with them to navigate academia together. Their input and willingness to listen have helped me through many challenges.

Finally, I would like to thank my family for their support and love. Mom, Dad, Russ, and Shannon, thanks for all the time I've spent talking about my research, even when I didn't do a good job explaining along the way.

Table of Contents

List of Tables

List of Figures

1 Introduction

The field of machine learning research has advanced rapidly 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 digit recognition to classify handwritten 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 to provide a common source of handwritten digit data so that the performance of different machine learning algorithms can be directly compared. 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 [MKB17].

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 details of machine learning differ from algorithm to algorithm, but for most methods, machine learning algorithms learn models from training data to encode the knowledge implicit in the training data. 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 that may control the speed of training or other high level details. Models tend to be complex, and can require thousands or millions of learned parameters for high accuracy on a given problem. Learned models are able to take a new piece of data as input and produce a result or judgment from that data. For the recognition 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.

The development of new machine learning algorithms or advances in training and validation techniques 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. Small refinements to algorithms and hyperparameters can have enormous impacts on training time, model size, and performance on unseen testing data. Because of

the complexity of the models produced by many machine learning algorithms, many 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 the standard training method for neural networks that minimizes the error in the network's model over the training data. To visualize the process of gradient descent, the algorithm seeks to find 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, which can occur for nonconvex optimization problems. 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, machine-checked

proofs. Formal verification often employs proof assistants, such as Coq, which 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 the correctness of the specifications and theorems proved. Proofs are implemented as portable programs, which allows for others to verify proofs independently. Because proofs and implementations are written in the same environment, 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 realistic 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, one might argue that 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, 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. Coq has been used to specify and verify a diverse range of algorithms, data structures, and applications beyond these three projects.

In this thesis, I will describe my additions to the verification framework MLCert [MLC]. MLCert provides software tools and libraries in the Coq proof assistant for verified machine learning in Coq, such as generic definitions and proofs for machine learning algorithms, example algorithms such as the Perceptron, and extensions for the implementation and training of neural networks. 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 motivation for the Budget Kernel Perceptron and Description Kernel Perceptron algorithms. 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 = w + training_set[j].
```

The most basic Perceptron algorithm has the following steps, as shown by the pseudocode in Figure 2.1. For this algorithm, we require as input the weight vector paired with its bias, the number of epochs, and the training set. Before training begins, each parameter in the weight vector w and the bias is initialized to zero. The training set is formatted to contain training examples paired with labels, where the label is either 0 or 1. The Perceptron algorithm consists of two nested loops. 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: prediction and update. 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 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 misclassified training example is added to the weight vector. This update step shifts the hyperplane in n-dimensional space to improve the Perceptron's classification with each mistake. The Perceptron is able to find a linear hyperplane to separate two classes of data because its model represents the hyperplane in n-dimensional space with each value in the weight vector corresponding to the coefficient

for each dimension. Every update of the model shifts the hyperplane away from training examples that were misclassified, and over time, the number of mistakes decreases.

The Perceptron algorithm is powerful despite its simplicity. However, there are limitations to the Perceptron's classification. As outlined by Minsky and Papert [MP69], 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 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. Minksy and Papert's work led to a decline in Perceptron and neural network research due to these limitations.

While the Perceptron is usually 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.

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.
```

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 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.

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:
            let (training_set, weights) = w in
            weights[j] += 1.
```

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 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 that produce models 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. Models with high generalization error tend to overfit to the training set, such as models that simply memorize the entire training set. When such models are presented with unseen examples in testing that are different from the training set, the model will have poor performance. 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, when apply typical

statistical guarantees, to the tightness of the generalization bounds. Verified generalization bounds guarantee worst-case expected 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 and 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, regardless of whether the algorithm has converged to a solution. 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. Reducing the number of support vectors to a size sufficiently smaller than the training set slows the growth of the parameter space as the size of the training set increases. 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 some number of mistakes, bounded by some value L. Using L, the number of support vectors can be capped at less than or equal to the number of mistakes. 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 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 provides 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 follow 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 correspond to the types of training data, training labels, hyperparameters, and parameters, respectively. 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 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 Learner module will later be instantiated with specific types and functions to implement the Perceptron family of algorithms.

The format of Learner.t 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 their prediction and update functions can be written using the specified type signature.

When implementing a machine learning algorithm, there are four required sections in Coq which organize and define its methods and data structures. 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 and its Learner.t instantiation are 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 in Figure 3.2:

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 32 bit 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 as 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 float32 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 float32 array in KernelParams is used for the kernel_predict calculation of the training example x's label, as each float32 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 float32 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 float32 array is updated so that the float32 value for that training example is incremented by one.

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 unit record. The Kernel Perceptron Learner can be defined as follows in Figure 3.5. The variable *K* is the generic type of the kernel function, and *F* provides a proof that the support vectors are a Foldable type, compatible with the parameter definitions in the Kernel Perceptron functions. 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 float32 array, the parameters for the Budget Kernel Perceptron are built as an axiomatized vector of size *sv* containing support vectors paired with a float32 value for that support vector. Axiomatized vectors are defined with extraction to Haskell lists in mind. 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 float32 array of size n, which is the dimensionality of the data, and does not need an index for each example for Budget classification.

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.
```

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 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 float32 value associated with this support 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 with respect to when it was added to the support set, 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 for the Budget Kernel Perceptron called kernel_update shown in Figure 3.9, the update rule used is the generic function U so that the budget update rule can be changed.

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).
```

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 empty record. K and F are defined in the same way as the Kernel Perceptron to specify the type of the kernel function and provide a proof that the support vectors are Foldable. 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 4.2.1, 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

The Description Kernel Perceptron is the final implementation located in "kernelperceptron.v". Many of its functions and definitions modify the Budget Kernel Perceptron, as both implementations use a fixed number of support vectors less than the number of training examples. The first major change for the Description Kernel Perceptron is the definition of its parameters. Figure 3.11 lists the definitions necessary to create dparams, the Description parameters. The definition of support vectors is the same as for the Budget Kernel Perceptron, relying on Akd and Bkd, which define the type of training examples and labels, respectively. However, the set of support vectors, dsupport_vectors, is simply an axiomatized vector containing des support vectors, where des is the maximum number of misclassifications. The parameters for the Description Kernel Perceptron are the set of support vectors paired with a float32 value, which keeps track of the number of misclassifications.

Figure 3.11: Parameter Definition in KernelClassifierDes

```
Definition dsupport_vector: Type := Akd ∗ Bkd.
Definition dsupport_vectors: Type := AxVec des dsupport_vector.
Definition dparams: Type := float32 ∗ dsupport_vectors.
```

Prediction for the Description Kernel Perceptron is also very similar to the Budget Kernel Perceptron, but the calculation is simpler. The kernel_predict_des function is shown in Figure 3.12. There are no float32 values paired with individual support vectors in dparams, as each support vector has the same weight. The reason for this will be shown in the kernel_update_des function. Once the float32 value is separated from the set of support vectors, the calculation over the support vectors is the same as the Budget Kernel Perceptron.

Figure 3.12: kernel_predict_des function in KernelClassifierDes

```
Definition kernel_predict_des
                (aw: dparams)
                (x: Akd) : Bkd :=
  let (a, w) := aw in
  foldable_foldM
    (λ wi_xi r ⇒
      let: (xi, yi) := wi_xi in
      r + (float32_of_bool yi) ∗ (K xi x))
    0 w > 0.
```

The update procedure for the Description Kernel Perceptron is likewise simplified from the Budget Kernel Perceptron. The Description Kernel Perceptron can make a maximum of *des* mistakes. When a misclassification occurs, the float32 value in dparams is compared to a hyperparameter *alpha*, which is set to the maximum number of misclassifications. If the Description Kernel Perceptron has already made the maximum number of mistakes, then the parameters are not updated for the rest of training. However, if the float32 value is not equal to *alpha*, then the des_update function runs using the

current parameters and the misclassified example. As described in Figure 3.13, the des_update function increments the float32 value by one to update the number of misclassifications. The misclassified example is added onto the front of the support set, and a zero example is removed. The parameters are initialized with *des* zero vectors, and as misclassfications occur, these zero vectors are replaced. For this update procedure, if an example is misclassified multiple times, multiple copies of the support vector will be added to the support set, increasing its influence on the hyperplane without using a float32 value per support vector. The full Description Kernel Perceptron is in 3.14.

Figure 3.13: des_update function in KernelPerceptronDes

```
Definition des_update (ap: Params) (yj: A*B): Params :=
    let (a, p) := ap in
    ((f32_add f32_1 a), (AxVec_cons yj) (AxVec_init p)).
```

Figure 3.14: kernel_update function in KernelPerceptronDes

```
Definition kernel_update
    (K : float32_arr n → float32_arr n → float32)
        (h:Hypers) (example_label:A*B) (ap:Params) : Params :=
    let: (example, label) := example_label in
    let: (a, p) := ap in
    let: predicted_label := kernel_predict_des K ap example in
    if Bool.eqb predicted_label label then ap
    else if f32_eq a (alpha h) then ap
        else des_update ap example_label.
```

Using these definitions, a Learner instantiation can be created, as shown in Figure 3.15.

Figure 3.15: Learner Definition in KernelPerceptronDes

```
Definition Learner : Learner.t A B Hypers Params :=
  Learner.mk
    (λ _ ⇒ @kernel_predict_des n (S des) K F')
    (@kernel_update K).
```

3.5 Chapter Summary

This chapter describes the implementation of the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron in Coq. The Learner.t definitions for each of the algorithms ensure that these implementations are correctly formatted for the rest of the MLCert framework. Using the definitions in this chapter, Chapter 4 will discuss proofs and Haskell extraction.

4 Proofs and Experimental Results

The results of my research include 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. Generalization proofs for the three Kernel Perceptron variants can be found in subsections 4.1.1, 4.1.2, and 4.1.3. The experiments run on the Kernel Perceptron variants were designed with three main questions in mind, including the difference between experimental and empirical generalization error, the generalization error and runtimes of the implementations on both real and synthetic datasets, and the difference in runtime between the Haskell implementations and unverified implementations in Python, the language most commonly used for machine learning research and applications. The extraction directives from Coq to Haskell for the three implementations are detailed in subsection 4.2.1. Subsections 4.2.2, 4.2.3, 4.2.4 describe the testing methodology for three datasets that were used to evaluate the experimental generalization error and analyze the runtime of training and testing each implementation. Finally, in subsection 4.2.5, the trends and results seen across datasets and implementations are outlined. 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 proven. 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. Most of the variables found in the Learner_bound proof can be found in the Learner.t instantiation, but *eps* is not part of the implementation. The value *eps* is the difference between expected accuracy and empirical accuracy, which is usually a value between zero and one. The specific value of *eps* is chosen to ensure that the resulting bound is nontrivial, as a bound greater than one is no longer useful for bounding generalization error. 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 two 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 of performance or correctness. However, the Kernel Perceptron bound serves as a baseline for the generalization bounds of 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 _ ⇒ 1) ≤
  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 the size of the support set. In code and proofs, the size of the support set is listed as $(S\ sv)$, which is equivalent to $sv + 1$. The successor of sv is used 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. The INR term before the cardinality of the parameter set is an injection from the encoding of natural numbers to the real numbers as part of the proof of this bound. 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 grows much less slowly 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 _ $\Rightarrow$ 1) $\leq$

INR 2ˆ((32∗(S sv) + ((1 + n ∗ 32)∗(S sv)))) ∗ exp (−2%R ∗ epsˆ2 ∗ mR m).

4.1.3 Description Kernel Perceptron Generalization Proofs

The generalization bound for the Description Kernel Perceptron is similar to that of the Budget Kernel Perceptron. First, we must define the cardinality of the parameter space used by the Description Kernel Perceptron. The parameters store a single float32 value paired with ($S\,des$) support vectors. The successor of des is used as the size of the support set, so that support vector replacement is always possible. Like with the Budget Kernel Perceptron, the cardinality of each support vector in the support set is 2^{1+n*32}, which stores a single Boolean label as well as n float32 values. However, because there is not a float value for every support vector, only the cardinality of a single float32 must be added to the cardinality of the entire support set. The cardinality of the Description parameters is shown in Equation 4.3.

$$\#|Params| = 2^{((32+((1+n*32)*(S\,des))))} \tag{4.3}$$

Figure 4.4 shows the generalization bound of the Description Kernel Perceptron in the lemma Kperceptron_bound_Des. This bound is similar to the bound of the Budget

Kernel Perceptron, but is tighter because there is only one float32 value instead of a float32 value per support vector. This small difference means that if the budget is the same as the number of mistakes for a specific dataset, the Description Kernel Perceptron will have a lower bound.

Figure 4.4: Generalization Bound for the Description Kernel Perceptron

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

4.2 Haskell Extraction and Performance Experiments

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 two sets of Haskell files located in the directory hs/KernelPerceptron/ 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/KernelPerceptron/ directory.

4.2.1 Details of 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", a Haskell module that can be included by a Haskell driver program. This file is extracted to the two locations in MLCert where Haskell driver programs reside: hs/KernelPerceptron/ and hs/KernelPerceptron/timing_drivers/. The Budget Kernel Perceptron and Description Kernel Perceptron each have their own extraction directives to extract "KPerceptronBudget.hs" and "KPerceptronDes.hs" to these same locations.

There are four different kinds of Haskell driver programs for a variety of different purposes. All driver programs report the training accuracy, test accuracy, and generalization error for the dataset run by that driver. Several also print the model produced by training. The drivers are differentiated by their file names, which identify the purpose of the drivers.

"KPerceptronXOR.hs" tests 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. This is the only driver that uses nonlinear data and a nonlinear kernel function for prediction.

Each implementation has a driver that generates a new linearly separable dataset using a random number generator to test that the implementation can execute. These

drivers have Test in their file names and first randomly generating n floating point values between negative one and one to determine a linear hyperplane. Training and testing examples are generated by this method and classified using the hyperplane, creating a synthetic dataset that is linearly separable by construction. These synthetic datasets test that the implementations have been set up correctly and can classify linear data. However, because the datasets are generated differently for each run of the program, these drivers cannot be used to compare implementations.

To compare the accuracy of an implementation on a specific dataset, each implementation has a driver which reads a dataset from files and performs training and testing. These drivers have RunFile in their file names. The drivers require that the input dataset be stored in two files, one containing the training set and the other containing the test set. The dataset files must also be formatted with one example per line and values separated by commas. The first value of the line must be a positive unique integer, which is used by the Kernel Perceptron to differentiate between the training examples for updates. The last value must be either True or False, corresponding to the label for the example. As long as the dataset files are formatted correctly, the driver will train and test on this dataset and report the accuracy and generalization error for this dataset. Only one dataset is run by this driver.

Finally, to time the execution of training and testing on a dataset, drivers with FileIO in their name read in one or more datasets. The FileIO drivers run training and testing five times and report the time in seconds. Each implementation has a FileIO driver which can run multiple datasets with varying dimensions. The drivers do not time reading the dataset files and use the same data format as the RunFile drivers.

In the following subsections, I will detail my testing methodology on three datasets, two real datasets downloaded from the UCI Machine Learning Repository [DG17] and a synthetic dataset created from randomly generated linear hyperplanes. The Iris Data Set

[Fis36] and Sonar Mines vs. Rocks Data Set [SG88] were formatted to be more easily read into the driver programs. Each of the following subsections describes the dataset under test, the experimental setup and drivers used, the calculated generalization error, and the timing results for training and testing.

4.2.2 Synthetic Dataset Performance Results

A synthetic dataset was created specifically to test the performance of the Kernel Perceptron and its variants. There are 20 independent trials in this dataset with training and testing sets based on randomly generated data separated by a randomly oriented linear hyperplane. Each trial contains 1000 training examples and 1000 test examples, each with three dimensions. These datasets were created by recording and formatting the output of 20 runs of "KPerceptronTest.hs".

Because the Kernel Perceptron variants are all deterministic, the generalization error results were found using the RunFile driver for each implementation. All implementations ran for five epochs. The Budget Kernel Perceptron limited the size of the support set to 100 examples, 10% of the training set, and the Description Kernel Perceptron was similarly limited to 100 mistakes.

The generalization error of the three implementations is graphed in Figure 4.5. The Kernel Perceptron and Budget Kernel Perceptron have the same training and testing error for all 20 synthetic trials, while the Description Kernel Perceptron tends to have slightly worse training and testing accuracy and increased generalization error on some trials. The greatest observed generalization error is 1.4% for the Description Kernel Perceptron in Trial 8.

Table 4.1 provides statistics across the synthetic trials for average training and testing accuracy and generalization error, along with the 95% confidence interval for each implementation. Because the Kernel Perceptron and Budget Kernel Perceptron had the

same performance across all trials, their averages and confidence interval are the same. The Description Kernel Perceptron's averages reflect its tendency to have lower accuracy and increased generalization error, and its confidence interval is larger than the other two implementations. For many of these trials, the Description Kernel Perceptron would require several more mistakes to have the same performance as the Kernel Perceptron and Budget Kernel Perceptron.

Using the generalization bounds as proven in Section 4.1, the number of training examples, dimensionality of the data, and size of the support set can be input to determine if the observed generalization error compares to the calculated bound. Table 4.2 shows the generalization bound for the two extreme values of *eps*, which are greater than zero and less than or equal to one. Unfortunately, for both of these values, all the implementations have vacuous bounds far greater than one. With the settings used in the above experiments, the calculated generalization bounds are unable to be compared to the observed generalization error.

Worse still is the fact that the Kernel Perceptron always produces vacuous bounds. Because all training examples are support vectors, the smallest possible training set is a single example. The smallest possible dimensionality of data is one dimensional. Therefore, the generalization bound for the smallest possible dataset of a single one-dimensional training example stored as a float32 value is shown in Equation 4.4. This bound is calculated to be 1.84467E+19 for *eps* = 0.001 and 2.49650E+18 for *eps* = 1. The bound becomes even more vacuous as the dimensionality of the data and number of examples increases. Because of this, the Kernel Perceptron cannot be used to compare calculated generalization error to generalization error observed in experiments.

$$2^{(m*n*32+m*32)} * e^{-2*eps^2*m} = 2^{(1*1*32+1*32)} * e^{-2*eps^2*1} = 2^{64} * e^{-2eps^2} \tag{4.4}$$

Figure 4.5: Synthetic Generalization Error

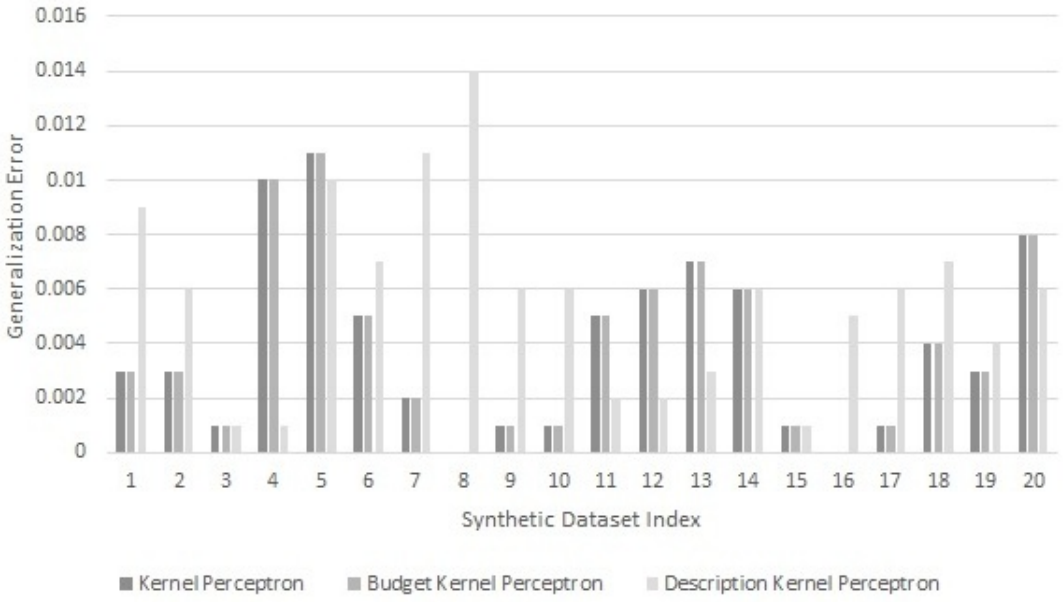


Table 4.1: Average Synthetic Generalization Error and Confidence Intervals

	KP	**Budget KP**	**Description KP**
Average Training Accuracy	98.61%	98.61%	96.915%
Average Testing Accuracy	98.58%	98.58%	96.87%
Average Generalization Error	0.39%	0.39%	0.565%
95% Confidence Interval	0.001435352	0.001435352	0.001539834

Table 4.2: Synthetic Generalization Bound Calculations

	KP	**Budget KP**	**Description KP**
Training Examples	1000	1000	1000
Dimensionality	3	3	3
Support Vectors	1000	100	100
eps = 0.001	6.89567E+38531	1.93230E+3883	4.19807E+2929
eps = 1	1.78025E+37663	4.98862E+3014	1.08381E+2061

To determine if the Budget Kernel Perceptron and Description Kernel Perceptron produce nonvacuous bounds, different support set sizes and values of eps were input into their generalization bounds. With the settings listed in Table 4.3, nonvacuous bounds were found for both variants. The synthetic dataset was used to evaluate the effect of a limited budget for the Budget Kernel Perceptron and a limited number of mistakes for the Description Kernel Perceptron. Both implementations have a generalization bound of about 9% for a three-dimensional dataset containing one thousand training examples and two support vectors or mistakes for Budget and Description, respectively. To produce these values, *eps* must be set to 0.301 for the Budget Kernel Perceptron and 0.282 for the Description Kernel Perceptron. The generalization error produced by these implementations with limited budget and mistakes is graphed in Figure 4.6. Overall, the training and testing accuracy for both implementations is far lower than the accuracy produced with larger support sets, but the difference between training and testing accuracy remains small. The greatest observed generalization error observed for the Budget Kernel Perceptron is 5.7% and for the Description Kernel Perceptron is 4.8%, well below the 9% calculated bound, which validates the theoretical bound proven in Coq for these implementations.

The runtimes of the three implmentations were also evaluated to compare training and testing performance. The runtime for each trial was tested five times to determine average training and testing. Figure 4.7 shows the synthetic timing results for the Kernel Perceptron. All runtimes were around 200 seconds, with little variation between trials. The averages for each trial and their 95% confidence intervals are listed in Table 4.4.

The Budget Kernel Perceptron and Description Kernel Perceptron execute much faster than the Kernel Perceptron because their models are significantly smaller. Figure 4.8 shows the synthetic timing results for these two implementaitons. The Budget Kernel Perceptron took around 0.35 seconds to train and test, while the Description Kernel

Figure 4.6: Synthetic Generalization Error using Limited Budget and Mistakes

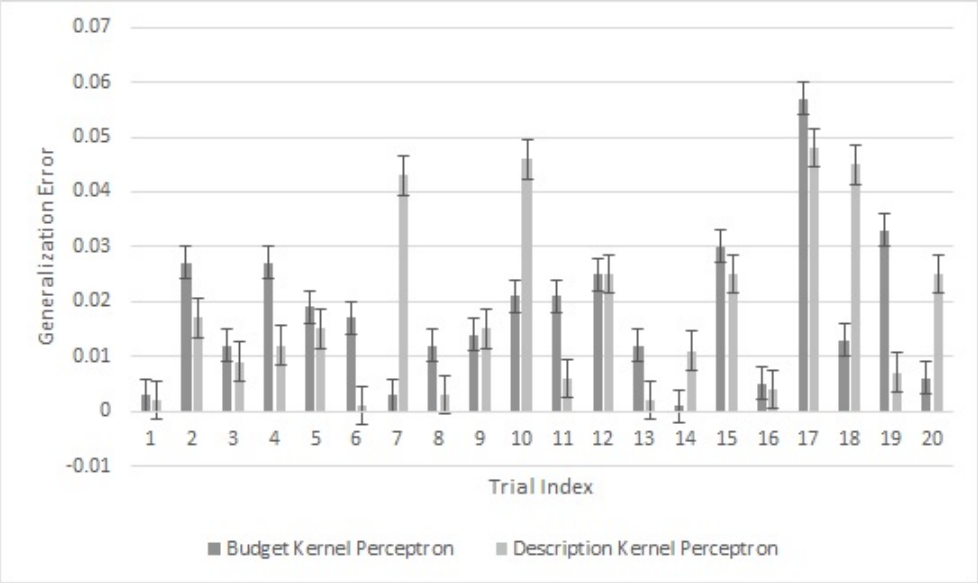


Table 4.3: Synthetic Generalization Error and Confidence Intervals using Limited Budget and Mistakes

	Budget KP	**Description KP**
Training Examples	1000	1000
Dimensionality	3	3
Support Vectors	2	2
eps	0.301	0.282
Calculated Bound	9.35%	9.10%
Greatest Observed Generalization Error	5.7%	4.8%
Average Training Accuracy	76.89%	78.625%
Average Testing Accuracy	76.6%	78.11%
Average Generalization Error	1.79%	1.805%
95% Confidence Interval	0.005794861	0.007007097

Perceptron took around 0.29 seconds. The Description Kernel Perceptron has the smallest model of the three implementations, causing it to have the fastest runtime. Table 4.4 shows the averages and confidence intervals for each trial.

The confidence intervals for the Kernel Perceptron trials are much larger than the Budget and Description Kernel Perceptron trials because the runtimes for the Kernel Perceptron vary by seconds, as opposed to hundredths of seconds for the Budget and Description Kernel Perceptrons. These runtimes show that it is possible for the Budget and Description Kernel Perceptrons to have the same or similar accuracy as the Kernel Perceptron while training and testing in a shorter amount of time.

The timing analysis of the Kernel Perceptron variants on the same datasets demonstrates how the runtimes for training and testing compare against implementations in the same language. However, because Haskell is not commonly used for machine learning tasks, the previous timing results do not indicate how the Haskell implementations compare against implementations in another language. Python is currently one of the languages of choice for machine learning research and applications because of libraries such as Numpy and Scipy that optimize scientific computing and machine learning algorithms. To compare the performance of Haskell to Python, the Budget Kernel Perceptron was chosen to be implemented in Python because it tends to have higher accuracy than the Description Kernel Perceptron with less required resources than the Kernel Perceptron.

Two unverified Python scripts were written to be run on synthetic data. The first script, “BudgetPython.py”, is close to a direct translation of the Haskell implementation into Python without modifications to improve performance in Python. This script will be referred to as the naive Python implementation. Many of the function and variable names were kept the same and the overall data structures and steps are the same. However, several changes were necessary. For example, the Haskell implementation uses 32-bit

Figure 4.7: Kernel Perceptron Synthetic Timing

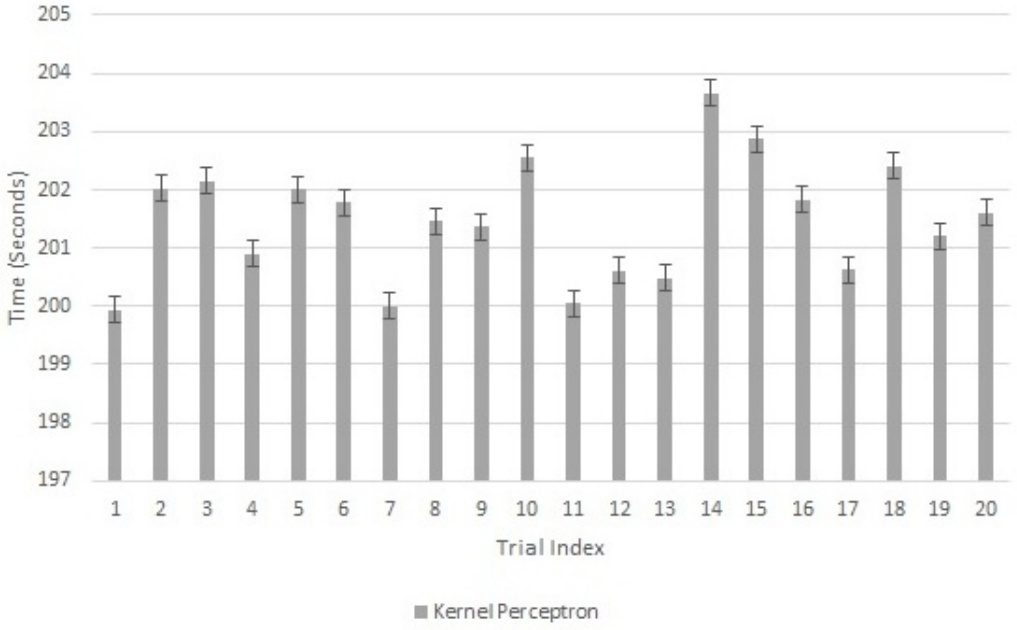


Figure 4.8: Budget and Description Kernel Perceptron Synthetic Timing

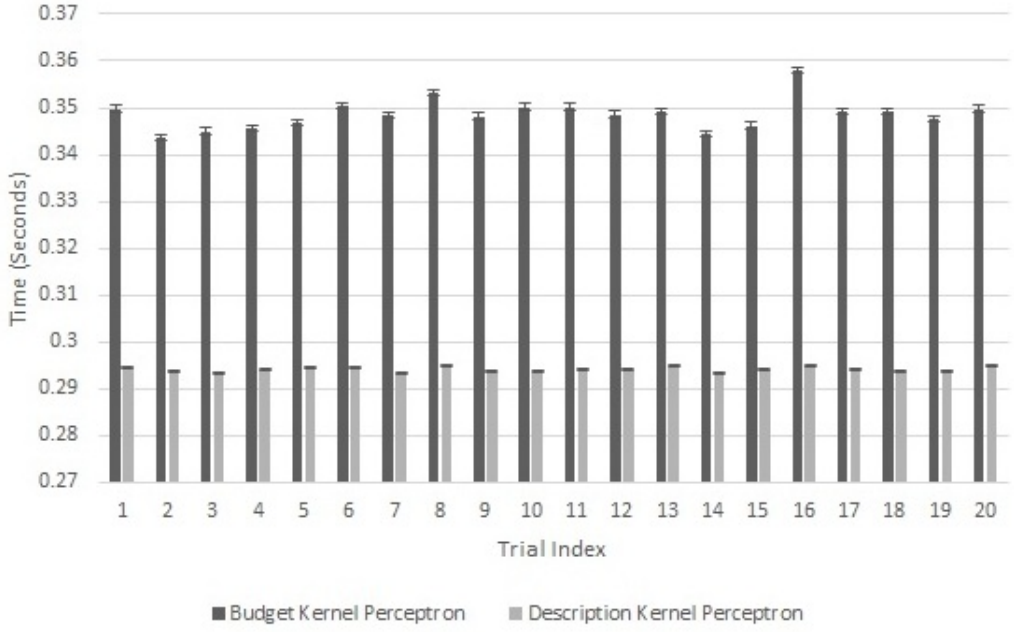


Table 4.4: Synthetic Average Runtimes (Seconds) and Confidence Intervals

Trial	KP	95% CI	Budget KP	95% CI	Description KP	95% CI
1	199.943	3.17258	0.3498	0.05499	0.2946	0.05606
2	202.025	2.44561	0.3436	0.05116	0.2938	0.05499
3	202.145	2.78125	0.345	0.05245	0.2932	0.05529
4	200.901	2.62953	0.3456	0.05263	0.294	0.05441
5	202.012	2.97056	0.3468	0.05351	0.2946	0.05557
6	201.786	2.46189	0.3504	0.05567	0.2944	0.05666
7	199.996	2.66681	0.3484	0.05175	0.2934	0.05422
8	201.455	3.07707	0.3532	0.05527	0.295	0.05440
9	201.362	2.92524	0.3482	0.05479	0.2936	0.05558
10	202.546	2.92950	0.3502	0.05332	0.2938	0.05452
11	200.051	2.58570	0.3502	0.05332	0.2942	0.05432
12	200.609	3.10256	0.3486	0.05215	0.294	0.05537
13	200.479	3.28990	0.3492	0.05331	0.2948	0.05647
14	203.655	3.41870	0.3444	0.05226	0.2934	0.05323
15	202.866	4.18980	0.3462	0.05384	0.294	0.05489
16	201.827	3.75820	0.358	0.05232	0.295	0.05587
17	200.627	3.80374	0.3492	0.05333	0.2942	0.05528
18	202.409	3.09442	0.3492	0.05381	0.2938	0.05450
19	201.203	2.51938	0.3476	0.05410	0.2938	0.05646
20	201.605	2.93406	0.3498	0.05351	0.295	0.05442

floating point values, but the naive Python implementation uses the built-in float type, which is 64-bit. Instead of recursion or folds, the naive Python implementation uses for loops to iterate through the parameters or the training and testing sets. Finally, the Python scripts do not use global variables, which are used in the Haskell driver files to specify the size and shape of the training set, so that multiple shapes and sizes of data sets can be run by the same functions. These changes simplify the Budget Kernel Perceptron implementation while not changing many of the underlying functions and data structures.

The second script, "BudgetPythonNumpy.py", improves on the naive Python implementation by using Numpy data structures and functions. Several types were modified to be more compatible with Numpy, such as the type of parameters. Instead of a list of tuples, where each tuple contains the float32 value, data, and label for a support vector, the Numpy Python implementation uses a tuple containing three Numpy arrays, so that the float32 values, data, and labels are in separate arrays. This change to the structure of the parameters makes vectorization much easier and facilitated the use of Numpy functions for computation. The Numpy Python and naive Python implementations produce the same results for accuracy and generalization error, but through calculations with different structures.

For both scripts, the timing procedure in Python was the same as in Haskell. For each trial of the synthetic dataset, the file input to read and format the dataset was not timed, but training and testing were timed five times to average the results. For each trial, the training and testing accuracy was compared to the Haskell implementation to ensure that all three Budget Kernel Perceptron implementations produced the same generalization error.

Figure 4.9 and Table 4.5 display the results of timing the Python implementations on the same datasets as the Budget Haskell implementation. The Haskell implementation was faster than both of the Python implementations training on the same synthetic datasets. The Numpy Python implementation was faster than the naive Python due to vectorization

Table 4.5: Python Budget Kernel Perceptron Runtimes and Confidence Intervals

Trial	**Budget Haskell**	**95% CI**	**Naive Python**	**95% CI**	**Numpy Python**	**95% CI**
1	0.3498	0.05499	0.82	0.00139	0.4266	0.00380
2	0.3436	0.05116	0.8216	0.00220	0.4024	0.00078
3	0.345	0.05245	0.819	0.00107	0.4088	0.00073
4	0.3456	0.05263	0.8198	0.00169	0.4208	0.00073
5	0.3468	0.05351	0.8216	0.00274	0.4236	0.00048
6	0.3504	0.05567	0.822	0.00107	0.4218	0.00073
7	0.3484	0.05175	0.8206	0.00118	0.4218	0.00114
8	0.3532	0.05527	0.8248	0.00209	0.4192	0.00243
9	0.3482	0.05479	0.82	0.00107	0.4146	0.00078
10	0.3502	0.05332	0.8232	0.00157	0.4184	0.00048
11	0.3502	0.05332	0.822	0.00196	0.421	0.00062
12	0.3486	0.05215	0.8206	0.00147	0.4188	0.00114
13	0.3492	0.05331	0.8232	0.00293	0.4326	0.00048
14	0.3444	0.05226	0.819	0.00164	0.396	0.00206
15	0.3462	0.05384	0.8234	0.00202	0.4136	0.00078
16	0.358	0.05232	0.8236	0.00237	0.4104	0.00048
17	0.3492	0.05333	0.8208	0.00073	0.4204	0.00133
18	0.3492	0.05381	0.82	0.00164	0.424	0.00062
19	0.3476	0.05410	0.818	0.00186	0.4226	0.00048
20	0.3498	0.05351	0.819	0.00186	0.4222	0.00073

and Numpy functions, although the Numpy Python implementation was more variable in times across trials than the naive Python implementation. The Python implementations validate that our Haskell implementations are not only comparable to the runtimes of Python implementations, but are slightly faster.

4.2.3 Iris Dataset Performance Results

The Iris Dataset [Fis36] contains 150 examples of four-dimensional data which represents 50 members each of three Iris species: Iris Setosa, Iris Versicolour, and Iris Virginica. Iris Setsosa is linearly separable from Iris Versicolour and Iris Virginica when Iris Versicolour and Iris are combined into a single class. This dataset is not divided into a training set and a testing set, which is required for determing generalization error as the difference between training and testing accuracy. To divide this dataset into a training set and a testing set, a random number generator was used to place each example either into the training set or the testing set. Two splits were used for this division, one with roughly 50% training examples (77) and 50% testing examples (73), and another with roughly 75% training examples (113) and 25% testing examples (37). These ratios were chosen to see if an increased number of training examples also increases the training and testing accuracy.

Once each training example was divided into separate training and testing files, each file was further formatted to replace the original labels of the dataset with Boolean values. Iris-setsosa was replaced with True, and both Iris-versicolour and Iris-virginica were replaced with False. A unique integer identifier was also added to the front of each example to conform to the specifications for file IO. With these preprocessing steps, the Iris data could be run by the Kernel Perceptron variants.

The generalization error results for the two Iris datasets were found using the RunFile driver for each implementation. All implementations ran for five epochs. The Budget Kernel Perceptron limited the size of the support set to 10% of the training set, and the

Figure 4.9: Python and Haskell Budget Kernel Perceptron Synthetic Timing

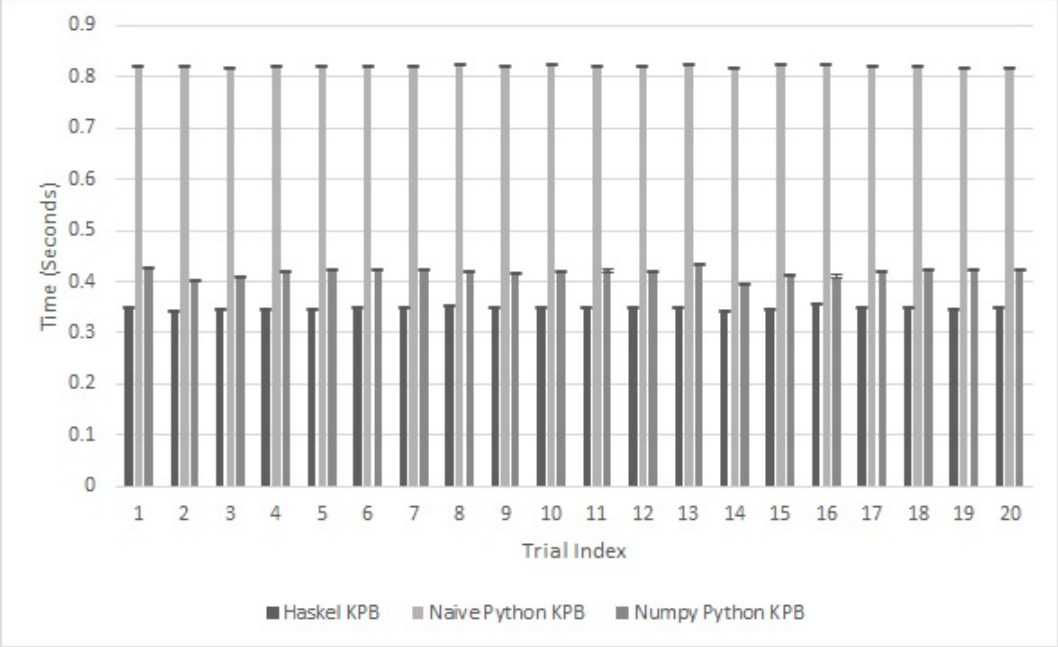


Table 4.6: Iris 50/50 Dataset Observed and Calculated Generalization Error

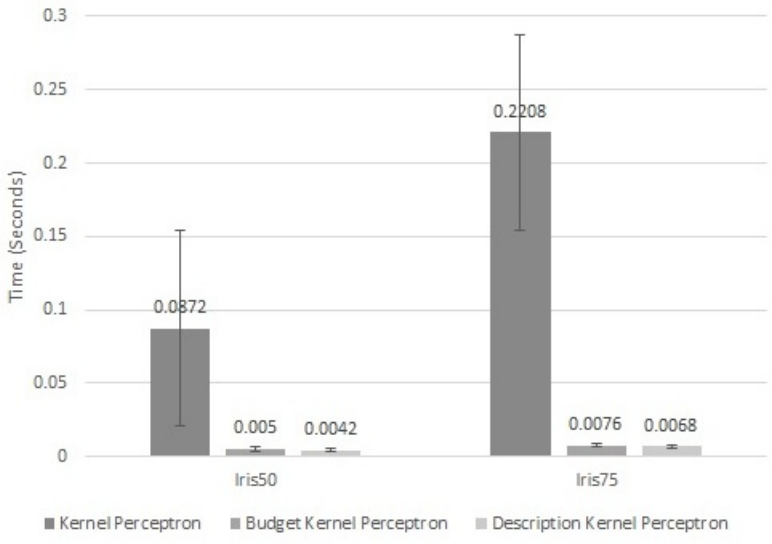
	Kernel Perceptron	**Budget KP**	**Description KP**
Training Examples	77	77	77
Testing Examples	73	73	73
Dimensionality	4	4	4
Support Vectors	77	7	7
Training Accuracy	100%	100%	100%
Testing Accuracy	100%	100%	100%
Generalization Error	0%	0%	0%
eps = 0.001	3.08674E+3036	6.76113E+271	1.07711E+214
eps = 1	4.05710E+2969	8.88660E+204	1.41572E+147

Description Kernel Perceptron was similarly limited to 10% of the size of the training set for the number of mistakes.

The results for the 50/50 split of the Iris dataset are shown in Table 4.6. All implementations had perfect training and testing accuracy, with zero generalization error. However, all of the implementations had vacuous bounds for this dataset, as no setting of *eps* produced a bound less than one. The results for the Iris 75/25 split are shown in Table 4.7. All implementations again had perfect training and testing accuracy, with no generalization error, and no implementation had nonvacuous bounds.

The timing results for the Iris datasets are shown in Figure 4.10. As with the synthetic dataset, the Kernel Perceptron took significantly longer to train and test than the Budget and Description Kernel Perceptrons, with the Description Kernel Perceptron slightly faster to train and test than the Budget Kernel Perceptron. Table 4.10 presents average runtimes and confidence intervals for the Iris datasets.

Figure 4.10: Iris Data Set Timing



4.2.4 Sonar Mines vs. Rocks Dataset Performance Results

The Sonar Mines vs. Rocks Dataset contains 208 examples of 60-dimensional data, representing sonar pings of either underwater explosive mines or roughly cylindrical rocks. This dataset is linearly separable. Like the Iris dataset, the Sonar dataset is not divided into a training set and testing set. Using the same method as the Iris dataset, two Sonar datasets were created with a roughly 50/50 split training and testing (116/92) and a roughly 75/25 split (157/51). The Sonar datasets were further preprocessed by adding a unique integer identifier to the front of each example and changing the labels from M to True and R to False.

The values for each example in the original Sonar dataset are decimals ranging from zero to one. The authors of [MGS17] stated that on the full Sonar dataset, their Perceptron took over 275,000 epochs to converge to a solution. Due to limitations on my machine used for performance experiments, it was not realistic to run my implementations for more than 10,000 epochs. Because of this limitation, the Sonar dataset was normalized from values from zero to one to values between negative one and one to center the examples closer to the zero vector used at the start of training. The normalized data files were used for the generalization error and timing experiments described below.

Again, the generalization error results were found using the RunFile driver for each implementation. All implementations ran for 10,000 epochs. The Budget Kernel Perceptron was limited to 10% of the training set and the Description Kernel Perceptron to 10% of the training set as the number of mistakes.

The results for the 50/50 split of the Sonar dataset are shown in Table 4.8 and for the 75/25 split are in Table 4.9. Only the Kernel Perceptron had results with higher than about 50% accuracy for both Sonar Datasets. The results for the Budget Kernel Perceptron and Description Kernel Perceptron were disappointing due to the fact that the Sonar datasets violate their underlying assumptions. To produce a model with high accuracy, thousands

of misclassifications must be made to incrementally move the hyperplane. Because the dataset contains a maximum of just 208 examples, variants that rely on making a number of mistakes significantly smaller than the number of training examples are bound to have poor performance. The accuracies of the Budget and Description Kernel Perceptron only increase when the number of misclassifications or support vectors is set to the minimum necessary, which means that such a setting will have much worse generalization error compared to the Kernel Perceptron. The Sonar datasets also do not produce nonvacuous bounds due to the size of the training set and the dimensionality of the data.

The timing results for the Sonar Datasets are given in Table 4.11. These results were found by running 10,000 epochs for each implementation, and the confidence intervals for each implementation are also listed.

4.2.5 Discussion of Generalization Error and Timing Results

There are several alternatives to the generalization bound formulation used in the MLCert framework. Mohri and Rostamizadeh [MR13] give the following formulation for their bound, expressed in Equation 4.5. For this bound, the authors posit that there are T labeled training examples used to train an on-line algorithm such as the Perceptron, and that these training examples are drawn i.i.d. from the distribution D. L represents the loss function used by the algorithm. The authors record the sequence of hypotheses generated by the algorithm as $h_1, ..., h_T$. Therefore, the bound aims to minimize the expected error of the algorithm by finding $\hat{h}$. The δ variable is similar to eps in MLCert's generalization bound. The mathematics behind this bound are based on the average accuracy of each hypothesis in the seqence to find the expected accuracy.

$$E_{(x,y)\sim D}[L(y\hat{h}(x))] \leq \frac{1}{T}\sum_{i=1}^{T} L(y_i h_i(x_i)) + 6\sqrt{\frac{1}{T}\log\frac{2(T+1)}{\delta}} \tag{4.5}$$

Table 4.7: Iris 75/25 Dataset Observed and Calculated Generalization Error

	Kernel Perceptron	**Budget KP**	**Description KP**
Training Examples	113	113	113
Testing Examples	37	37	37
Dimensionality	4	4	4
Support Vectors	113	11	11
Training Accuracy	100%	100%	100%
Testing Accuracy	100%	100%	100%
Generalization Error	0%	0%	0%
eps = 0.001	4.19009E+5442	1.33053E+533	6.22910E+436
eps = 1	2.96325E+2969	9.40956E+434	4.40525E+338

Table 4.8: Sonar 50/50 Dataset Observed and Calculated Generalization Error

	Kernel Perceptron	**Budget KP**	**Description KP**
Training Examples	116	116	116
Testing Examples	92	92	92
Dimensionality	60	60	60
Support Vectors	116	11	11
Epochs	10,000	10,000	10,000
Training Accuracy	100%	55.17%	55.17%
Testing Accuracy	69.57%	51.09%	51.09%
Generalization Error	30.43%	4.08%	4.08%
eps = 0.001	6.66619E+68162	1.06487E+6467	4.98537E+6370
eps = 1	Would not compute	1.86671E+6366	8.73934E+6269

Mohri and Rostamizadeh do not directly give generalization bounds for the Kernel Perceptron, but they do list modifications that can be made to their theorems that account for kernelization. However, this bound requires the storage of the entire sequence of hypotheses through testing. As our system only stores the most recent hypothesis, the MLCert bound cannot look into past hypotheses. Unfortunately, our generalization bounds cannot directly be compared to Mohri and Rostamizadeh's because of this method of formulation.

Cesa-Bianchi, Conconi, and Gentile [CBCG04] formalized a generalization bound also based on the sequence of hypothesis created during training. In the authors' bound, where they use risk in place of generalization, the number of training examples is n such that Z^n represents the entire set of training examples. K represents the kernel function used, and the dual kernel, different from the kernel defined in our research as well as by Mohri and Rostamizadeh, is provided in Equation 4.6. $\mathcal{M}$ is defined in Equation 4.7 as the set of all misclassifications, and t represents the index of a single misclassification. f is the norm of the kernel space. This bound uses hinge-loss instead of the usual zero-one loss function for the Kernel Perceptron, and γ defines the margin for the hinge-loss function. The full bound is shown in Equation 4.8.

$$\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j K(x_i, x_j) \geq 0 \tag{4.6}$$

$$\mathcal{M} = 1 \leq t \leq n : H_{t-1}(X_t) \neq Y_t \tag{4.7}$$

$$risk(\hat{H}) \leq \inf_{f\in\mathcal{H}_K:\|f\|\leq 1}\inf_{\gamma>0}\left(D_{\gamma,n}(f, Z^n) + \frac{1}{\gamma n}\sqrt{\sum_{t\in\mathcal{M}} K(X_t, X_t)}\right) + 6\sqrt{\frac{1}{n}\ln\frac{2(n+1)}{\delta}} \tag{4.8}$$

As with the bound by Mohri and Rostamizadeh, Cesa-Bianchi, Conconi, and Gentile define their bound in terms of minimizing $\hat{H}$ from the full set of hypotheses found during

training. Again, our generalization bound cannot be directly compared to the above bound because we do not store past hypotheses. Additionally, as our implementations do not use the dual kernel, the kernelization and kernel space described in this paper do not directly correlate to our kernel and parameter spaces.

These bounds do, however, show that generalization is an important performance metric, although the exact formulation may vary even for the same or similar algorithms. Our generalization bound calculations are simpler to calculate and rely on fewer variables.

Some trends between datasets and implementations are shown in the generalization and timing analyses. Many of the generalization bounds calculated in this thesis are vacuous. Nonvacuous bounds rely on a very small support set compared to the full training set. Unfortunately, the computer I have had to use for my research is not powerful enough to run experiments with a larger training set size to examine more nonvacuous bounds due to its age and computational capabilities. Vacuous bounds provide no real guarantee for performance, even though the theorems for the generalization bounds have been proved in Coq. Future work to examine the nonvacuous bounds of the Budget and Description Kernel Perceptrons could determine guidelines for the dimensions and size of datasets that will provide nonvacuous bounds.

MLCert uses 32-bit floating point values to represent data, matching precision in Haskell. Although the use of fewer bits would lower the accuracy of the model, 8-bit or 16-bit floating point representations would have tighter generalization because the cardinality of the parameters would significantly decrease. With an 8-bit floating point representation, the Kernel Perceptron's generalization bound is shown in Equation 4.9 and the Budget Kernel Perceptron's bound follows in Equation 4.10.

$$2^{(m*n*32+m*32)} * e^{-2*eps^2*m} \tag{4.9}$$

$$2^{(8*(S\,sv)+(1+n*8)*(S\,sv))} * e^{-2*eps^2*m} \tag{4.10}$$

Even with an 8-bit floating point representation, the Kernel Perceptron's generalization bound is still vacuous for all datasets. Equation 4.11 shows the calculation for the Kernel Perceptron's generalization bound on the smallest possible dataset of a single, one-dimensional example. With the two extreme values of *eps* of 0.001 and 1, the Kernel Perceptron's bounds are 65,535.9 and 8,869.3, respectively. A comparison of the generalization bounds of the Budget Kernel Perceptron with 32-bit and 8-bit precision are detailed in Table 4.12. The Budget Kernel Perceptron has nonvacuous bounds on the synthetic dataset as long as the training set is much larger than the support set, as shown by the limited budget experiments. However, the use of 8-bit precision would allow for a larger budget of support vectors with a nonvacuous bound. For example, with eight support vectors and *eps* set to 0.304, the 8-bit Budget calculated generalization bound is 15.86%. In the limited budget experiments, the support set must be limited to two 32-bit support vectors to calculate nonvacuous bounds.

$$2^{(m*n*8+m*8)} * e^{-2*eps^2*m} = 2^{(1*1*8+1*8)} * e^{-2*eps^2*1} = 2^{16} * e^{-2eps^2} \tag{4.11}$$

The results of the timing analysis, especially for the synthetic datasets, shows that the Budget and Description Kernel Perceptrons are significantly faster to run when the size of the budget or number of mistakes are a fraction of the full size of the dataset. Storing fewer nonessential support vectors saves computation time and resources compared to the Kernel Perceptron.

Timing results with the Budget Python implementations also demonstrate that the Haskell implementations have slightly faster run times than Python implementations, even the Python implementation which was optimized using Numpy functions and data

structures. While Haskell is not often used for machine learning, it has been demonstrated to be viable for running machine learning algorithms.

The Sonar dataset highlights that the Budget and Description Kernel Perceptrons rely on the either a small number of necessary support vectors or a small number of mistakes compared to the size of the training set. In cases where neither of these assumptions is true, the Budget and Description Kernel Perceptrons will have worse accuracy compared to the Kernel Perceptron. The Kernel Perceptron works best for datasets with a high number of necessary support vectors of many mistakes compared to the size of the training set. Because of this, the Budget and Description Kernel Perceptrons are not always improvements on the performance of the Kernel Perceptron.

Finally, the Iris dataset validates that the Kernel Perceptron variants can perform well on a real-world dataset. Because few misclassifications are necessary to produce a hyperplane with perfect accuracy, all variants of the Kernel Perceptron had high performance on the Iris dataset. Overall, these results show that the variants of the Kernel Perceptron can perform well on real and synthetic data.

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.

Table 4.9: Sonar 75/25 Dataset Observed and Calculated Generalization Error

	Kernel Perceptron	**Budget KP**	**Description KP**
Training Examples	157	157	157
Testing Examples	51	51	51
Dimensionality	60	60	60
Support Vectors	157	15	15
Epochs	10,000	10,000	10,000
Training Accuracy	84.08%	54.14%	54.78%
Testing Accuracy	82.35%	50.98%	50.98%
Generalization Error	1.73%	3.16%	3.8%
eps = 0.001	7.18546E+92254	4.71614E+8818	6.48856E+8683
eps = 1	Would not compute	2.01955E+8682	2.77855E+8547

Table 4.10: Iris Average Runtimes (Seconds) and Confidence Intervals

Trial	**KP**	**95% CI**	**Budget KP**	**95% CI**	**Description KP**	**95% CI**
Iris 50/50	0.0872	0.00342	0.005	0.00392	0.0042	0.00431
Iris 75/25	0.2208	0.00563	0.0076	0.00462	0.0068	0.00501

Table 4.11: Sonar Average Runtimes (Seconds) and Confidence Intervals

Trial	**KP**	**95%CI**	**Budget KP**	**95%CI**	**Description KP**	**95%CI**
Sonar 50/50	1045.839	1.49470	79.0278	2.06799	66.1688	1.28507
Sonar 75/25	2121.471	6.43889	138.6656	4.08308	121.6616	3.08207

Table 4.12: Synthetic Generalization Bound Calculations, 32-bit versus 8-bit Budget

	Budget KP	
Training Examples	1000	
Dimensionality	3	
Support Vectors	100	
Precision	32-bit	8-bit
eps = 0.001	1.93230E+3883	2.50102E+993
eps = 1	4.98862E+3014	6.45687E+124

5 Conclusions

The previous chapters of this thesis outline the methodology and results of verifying the Kernel Perceptron family of algorithms in the MLCert framework. The main contributions of this work include Coq implementations of the Kernel Perceptron, Budget Kernel Perceptron, and Description Kernel Perceptron with proofs in Coq of their generalization error. The accuracy, generalization error, and runtime of the Coq implementations were tested through extraction into Haskell. The Haskell Budget Kernel Perceptron's timing was also compared to two Python versions to demonstrate the efficiency of the Haskell implementations against implementations in a language used by most machine learning researchers.

However, there are limitations to the Kernel Perceptron implementations. Because the calculation of generalization error relies on the cardinality of the parameter space and its relationship to the size of the training set, the Kernel Perceptron will always produce vacuous bounds, regardless of the dimensionality of the data. Therefore, the generalization proof in Coq for the Kernel Perceptron does not guarantee its generalization performance. In contrast, the Budget Kernel Perceptron and Description Kernel Perceptron can produce nonvacuous bounds, as long as the size of the support set is sufficiently smaller than the number of training examples. Experimental tests of the Budget and Description Kernel Perceptrons with limited numbers of support vectors or mistakes, respectively, had much lower accuracy, but their generalization error was lower than the calculated generalization bound. Unfortunately, I was unable to test larger datasets due to limitations on my machine or determine if both high accuracy and low generalization could be achieved on larger datasets.

Future work for this research in MLCert could involve the implementation of additional machine learning algorithms to evaluate their generalization error. There are other variants of the Perceptron algorithm that could be investigated. Modifications to the

Budget and Description Kernel Perceptrons may impact or improve the generalization error of these algorithms. For the Budget Kernel Perceptron, the parameter update procedure could be improved to remove the support vector with the least impact on the hyperplane, which may not be the oldest support vector. The Description Kernel Perceptron could be improved by changes to the MLCert framework itself. The current implementation will run for the specified number of epochs, even if the maximum number of mistakes is reached in the first epoch. Modifying some of the MLCert functions could make it possible to stop training early, improving the timing of the Description Kernel Perceptron. Fortunately, MLCert can be customized in a variety of ways to make future exploration of machine learning algorithms possible.

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.

[CBCG04] Nicolo Cesa-Bianchi, Alex Conconi, and Claudio Gentile. On the generalization ability of on-line learning algorithms. *IEEE Transactions on Information Theory*, 50(9):2050–2057, 2004.

[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.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml

[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.

[Fis36] R. A. Fisher. Iris data set, 1936. URL: https://archive.ics.uci.edu/ml/datasets/iris

[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.

[MKB17] Mohssen Mohammed, Muhammad Badruddin Khan, and Eihab Bashier Mohammed Bashier. *Machine Learning: Algorithms and Applications*. CRC Press, 2017.

[MLC] Mlcert: Certified machine learning. URL: https://github.com/OUPL/MLCert

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons: an Introduction to Computational Geometry*. M.I.T. Press, 1969.

[MR13] Mehryar Mohri and Afshin Rostamizadeh. Perceptron mistake bounds. *arXiv*, 2013.

[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.

[SG88] Terry Sejnowski and R. Paul Gorman. Connectionist bench (sonar, mines vs. rocks) data set, 1988. URL: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+%28Sonar,+Mines+vs.+Rocks%29

[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.

Thesis and Dissertation Services