Evaluation of Diagonal Confidence-Weighted Learning on the KDD Cup 1999 Dataset for Network Intrusion Detection Systems

Matthew W. Pagano
Johns Hopkins University
3400 North Charles Street
Baltimore, MD 21218, USA
mpagano@cs.jhu.edu

Abstract

In this study, I evaluate the performance of diagonal Confidence-Weighted (CW) online linear classification on the KDD Cup 1999 dataset for network intrusion detection systems (NIDS). This is a compatible relationship due to the large number of instances in NIDS datasets, as well as the constantly changing feature distributions. CW learning achieves approximately 92% accuracy on the KDD dataset when optimized, which is higher than both Perceptron and the Passive-Aggressive algorithm. CW learning also achieves faster convergence rates than both of these algorithms. Moreover, the accuracy of CW learning on the KDD dataset is comparable to several batch-learning algorithms. This challenges the assumption that batch learning should always be used when feasible. Due to shortcomings of the KDD dataset, a full generalization of CW learning to additional NIDS environments cannot yet be made. Nonetheless, this study shows that there is great promise to applying CW learning to future NIDS research.

1 Introduction

My objective is to evaluate the accuracy of diagonal Confidence-Weighted (CW) online linear classification to the KDD Cup 1999 dataset for network intrusion detection systems (NIDS). The motivation and prerequisite background information for this are given below.

1.1 Batch Learning

One of the most fundamental classifications in machine-learning taxonomy is that of batch versus online learning. In the training phase of batch learning, the algorithm considers all training instances at once to form the parameters of the learned model. After training is complete, the model can make predictions on previously unobserved and potentially unlabeled instances.

Training as such makes implicit assumptions about the data. First, batch learning assumes the data is generated from an independent and identical distribution (IID). That is, each instance is generated from the same distribution and is independent of all other instances. This is done to simplify the likelihood equations. Furthermore, batch learning assumes that each instance is generated with the same optimal hypothesis.

These properties have practical implications that influence the performance of batch learning. Because all training instances must be considered at once, generally there must be enough system memory to store the entire training dataset. This size constraint can limit how much training data the system can handle. This is especially problematic because the size of the training data and the subsequent accuracy of the learned model are often directly proportional. In addition, once the model parameters are set, the model cannot malleably adapt to distribution changes without expensive re-training.

1.2 Online Learning

Nonetheless, it is commonly held that batch learning offers higher accuracy than online learning when

batch learning can be applied (Dredze, 2008). Recent advances in online learning, however, question this theory.

One of the main advantages of online learning is that it avoids the assumptions made by batch learning (Dredze, 2010). Rather than having a clear distinction between training and testing phases (and separate datasets for each), an online learner is only given access to continuous stream of instances. These instances are not necessarily generated from an IID and are not required to be generated from a single hypothesis.

In online learning, the learner considers each instance individually, makes a prediction using the model it has learned up to that point, receives the true label, and then updates its model parameters accordingly. That instance is not recorded and is never used again; all that remains are the updates to the model parameters for future use.

This strategy offers many benefits in performance and applicability. Because only one instance is considered at a time, there are no resource constraints on the amount of data from which the model can learn. Similarly, the model can be updated easily without an expensive re-training phase. The removal of the assumptions of batch learning also enables online learning to be applied to a wider variety of environments.

1.3 Confidence-Weighted Linear Classification

Confidence-Weighted (CW) linear classification (Dredze, 2008) is a recently developed online-learning algorithm that provides these advantages. In addition, recent research has shown that CW learning outperforms batch learning with certain datasets (Ma, 2009).

1.3.1 Overview of Online Learning

To understand CW learning, it is necessary to review online learning for linear classification. The following tutorial is based on that given by Dredze *et al.* (2008).

An online learner receives one instance at a time. Label that instance $\mathbf{x_i} \in \mathbb{R}^\mathbf{d}$ since our instance $\mathbf{x_i}$ is a d-dimensional vector with real-valued coefficients. These d dimensions are the features of our model. The learner then uses its current model parameters to generate a prediction $\hat{y}_i \in \{-1, +1\}$ (i.e., binary classification) for the label of instance $\mathbf{x_i}$. Next, the learner receives the true label $y_i \in \{-1, +1\}$ of $\mathbf{x_i}$. Based on the loss suffered between the predicted label $\hat{y}_i$ and the true label y_i (determined by the loss function used in the model), the learner updates its model parameters.

The objective of linear classification is to find the hyperplane that optimally separates the d-dimensional space into two spaces, where one space contains only instances of label -1 and the other contains only instances of label $+1$. If the data is linearly separable, then such a hyperplane exists. This hyperplane can be represented by the following linear equation:

$$f_\mathbf{w}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$$

Here $\mathbf{w}$ is a d-dimensional weight vector that is used for classification by representing the decision boundary between -1 and $+1$ labels. Each component i in $\mathbf{w}$ represents the weight associated with the i-th feature of the model (where d is the total number of features). These weights w_i are the parameters of the model.

Given an instance $\mathbf{x} \in \mathbb{R}^\mathbf{d}$, the online learner generates its prediction $\hat{y}$ by taking the sign of the dot product of $\mathbf{x}$ and $\mathbf{w}$ (thus the prediction will be either -1 or $+1$):

$$\hat{y} = sign(\mathbf{w} \cdot \mathbf{x})$$

The final step is to update the weight vector $\mathbf{w}$ based on the loss suffered after receiving the true label y. How $\mathbf{w}$ is updated depends on the classifier being used.

1.3.2 Specifics of CW Learning

In CW learning, each of the weight values w_i is assigned confidence information that represents the certainty of that value. Confidence is quantified using a multivariate Gaussian distribution with real-valued means and standard deviations. Each weight value w_i has its own Gaussian with mean μ_i and standard deviation σ_i. The covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ is represented differently in the CW variants (to be discussed in Section 4.4.3). For each feature i, the mean μ_i is the weight value w_i, and the standard deviation σ_i is the confidence of that weight

value, where standard deviation and confidence are inversely proportional.

During the update phase (which occurs after each new instance), the Gaussian distribution for each feature is updated. This in turn updates both the weight value and the associated confidence for each feature. Rather than applying updates equivalently, weight values with higher standard deviations (thus lower confidences) are updated more aggressively. After the update, the new Gaussian distribution across all features must predict the true label for the current instance with probability greater than a pre-specified confidence level.

The optimization objective and constraint used for CW learning differ for each variant. The details for each are given in Section 4.4.3.

1.3.3 Applications of CW Learning

Because of the confidence scores associated with each feature, CW learning often shows higher accuracies than other online algorithms. This is most acute with datasets containing large numbers of binary features that are usually zero. Many online algorithms have problems with these datasets because the algorithms only update their weight values when the corresponding feature values are non-zero.

Consequently, weight values of more frequent features receive more updates, leading to models in which only frequent features have accurate representations. This can in turn decrease the convergence rate because more iterations are needed to determine the weight values of the rarer features. Ironically, the same property that enables online learning to handle large datasets quickly (the fact that it only needs to remember one instance at any time), also prevents it from understanding all features quickly (especially the less frequent ones) (Dredze, 2008).

CW learning addresses this by maintaining a confidence score for each of its weight values. These scores influence the rate at which a given weight value is updated, where weight values with lower confidences are updated more aggressively.

Ma *et al.* (2009) demonstrate that this approach is highly effective in classifying a URL as either benign or malicious from the properties of the URL string. These properties include such features as hostname, primary domain, geographic information, and connection speed, but not that actual web-page content. The authors observe that their corpus of labeled URLs is appropriate for online learning because the corpus is large (much larger than batch algorithms can handle) and because the feature distribution changes considerably over time.

In the study by Ma *et al.* (2009), online algorithms produce higher accuracies than batch algorithms because the number of instances on which batch algorithms can train is limited by system resources. Although other online algorithms such as Perceptron (Rosenblatt, 1958), Logistic Regression with Stochastic Gradient Descent, and the Passive-Aggressive Algorithm (Crammer, 2006) classify the dataset well, CW learning achieves the highest accuracy (up to 99%). The authors believe this is because CW adjusts the rate of learning on a per-feature basis, which addresses the constantly changing feature distribution in their data. The authors further note that continuous re-training of the model is necessary in order to account for the regularly changing and newly created features.

The motivation for my project came after reading this paper. I noticed that the authors' rationale for using online learning (and specifically CW learning) for malicious-URL detection also applied to network intrusion detection systems. I discuss this link further in the next section.

2 Network Intrusion Detection Systems

First, I discuss the fundamental properties of network intrusion detection systems (NIDS). My taxonomy largely comes from established divisions in the field, but I am also loosely following the classification scheme given by Small (2010).

2.1 Fundamentals

The goal of NIDS as it pertains to this study is to properly characterize network traffic as benign or malicious. In this sense, it can be reduced to a binary-classification problem. From a high level, NIDS tactics fall into two categories: 1) signatures; 2) statistical anomalies.

Signatures are patterns that are mostly found in malicious activity, but rarely found in benign activity. For example, this might be a succession of system calls; a string of opcode bytes in shellcode; or a series of queries made in an application, service,

or networking protocol. Network traffic is examined against these patterns, and if a match is found, that traffic is labeled malicious. If the traffic does not match one of the patterns, the traffic is labeled benign. Signatures are usually written manually after the detailed steps of an attack are determined, and as such, do not involve many components of machine learning. Consequently, I will not discuss them further in this study.

On the other hand, NIDS based on statistical anomalies borrows heavily from machine learning. In the typical setting, anomaly detection is based on a signal-to-noise ratio in a given network. First, a baseline of normal network activity is established. It is realistically quite challenging to maintain a fully functional network that is free of attacks for establishing this baseline (if this could be done, there would be no need for further security measures). However, it is assumed that a reasonable attack-free yet completely functional network can be measured to form a baseline.

After this step, all network traffic is compared against the baseline. If the traffic being evaluated falls within baseline parameters, the traffic is labeled benign; otherwise, it is labeled malicious. The baseline serves as the noise floor, and effectively sets the rate of false alarms. If the baseline is set to be too large, then malicious traffic might fall within the baseline parameters and thus wrongly labeled as benign (false negative). If the baseline is set to be too small, then normal traffic might fall outside and thus wrongly labeled as malicious (false positive).

2.2 Machine Learning and NIDS

There is much overlap between anomaly-based detection for NIDS and recent advances in machine learning. Accordingly, there exists a large body of research on applying machine-learning algorithms to the NIDS problem (Zhang, 2001; Hu, 2003; Sinclair, 1999).

I was motivated to apply CW learning to NIDS data because the two main justifications given by Ma *et al.* (2009) (discussed in Section 1.3.3) for malicious URLs also apply to NIDS. First, the size of training data for NIDS classification can be very large. The modern network consists of many workstations, servers, and mobile devices (e.g., smart phones), each of which generates its own network traffic. Logs of this traffic can easily exceed gigabytes per day. In addition, these devices usually run multiple applications that generate their own network traffic. This leads to very large datasets.

Second, the distribution and number of features is constantly changing. The modern network is a highly dynamic environment that is constantly in flux. Between dynamic forms of host addressing (e.g., Dynamic Host Configuration Protocol, or DHCP), wireless and remote workers (which have broken the standard perimeter model of network security), and the recent increase in mobile devices capable of standard networking, the shape of the modern network is constantly changing. On top of that, these devices and workstations are communicating over a multitude of different application protocols, each with its own language and specifications.

Because of these similarities with the study by Ma *et al.* (2009), I hypothesized that CW learning would also be effective with NIDS datasets.

3 DARPA Intrusion Detection Evaluation Datasets

In 1998, the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory funded the first official dataset specifically designed for intrusion detection systems (Lippmann, 2000). The actual implementation and data collection was carried out by the Information Systems Technology Group (ISTG) of MIT Lincoln Laboratory. Another dataset of the same objective was collected in 1999 as well.

To create the dataset, Lincoln Labs simulated a local area network of the U.S. Air Force. Simulations were preferred over real networks due to privacy concerns. Lincoln Labs subjected the network to multiple, diverse attacks whose traffic instances were later labeled. The result was seven weeks of network traffic for training data and two weeks for testing data. The data was stored in compressed tcpdump format, and totaled 5 million connection records for training data and 2 million connection records for testing data (Tavallaee, 2009).

Since then, these datasets have been used by many researchers to evaluate their NIDS methods. They have come to be quite ubiquitous in the intersection between machine learning and NIDS. Although

some researchers have criticized their applicability to NIDS research (see Section 6), they have been heavily analyzed and discussed in the years that followed their release. It was for these reasons that I chose to use a distillation of these datasets, the KDD Cup 1999 dataset, for my project.

4 Evaluation Procedures

This section describes the procedures I followed to evaluate CW learning and other online algorithms on the KDD Cup 1999 dataset.

4.1 Data Collection

The Association of Computing Machinery (ACM) Special Interest Group on Knowledge Discovery and Data Mining (KDD) organizes an annual competition for practitioners of this field. Each year a dataset from a diverse subject (e.g., breast cancer, consumer recommendations, or particle physics) and a machine-learning task based on that dataset are made publicly available. Researchers develop their algorithms to solve this task and then submit their solutions. These solutions are evaluated by the KDD group, and the winners are announced afterward.

In 1999, the KDD Cup datasets were distillations of the 1998 DARPA Intrusion Detection Evaluation datasets discussed in Section 3. The original DARPA datasets were still in the file formats used by their passive collection devices (e.g., tcpdump, ufsdump, and Windows NT audit event logs). As such, they were not immediately conducive to machine-learning algorithms. Some of these datasets were converted into machine-readable form to serve as the KDD Cup 1999 datasets.

4.2 Data Formatting

The format used by the KDD Cup 1999 was not SVM, which is my preferred format. I therefore wrote a Python script to convert the existing training and testing KDD datasets to SVM. In addition, the dataset was originally comma-separated and did not list feature numbers. I converted it to use features numbers followed by colons, as in the SVM format.

The original labels were either "normal" or the name of the attack corresponding to that instance. There were a total of 22 attacks in the dataset. I converted this labeling system to a binary scheme so that I could use the linear classification tools described in this paper. Moreover, the conversion to binary classification allowed me to run the Passive-Aggressive (PA) Algorithm (Crammer, 2006) on the data. I used PA to better compare my results with that of Ma *et al.* (2009).

To perform the binary conversion, I configured my Python script to replace a "normal" label with $+1$ and any other label with -1. This is valid because the objective of most NIDS implementations is to detect *any* malicious activity. Granularly classifying the malicious activity by type is usually not a priority. Tavallaee *et al.* (2009) also performed the same binary conversion in their analysis on the KDD dataset.

Next, the original data contained zeroes for feature values. To improve efficiency, I converted this into a sparse-vector scheme in which features with zero values were omitted from the data (their omission implied a zero value). The software I used was capable of properly interacting with sparse vectors.

Lastly, three of the original features were symbolic, but non-binary. For example, the "protocol_type" feature value could be one of the following: "tcp", "udp", or "icmp." For all such non-binary symbolic features, I created new binary features for each of the possible options. For a given instance, all of these new features would be zero except for the new feature that represented the instance's original feature value. For example, the original Feature 2 was the "protocol_type" feature discussed above. I configured my Python script to create new binary features 2 ("tcp"), 3 ("udp"), and 4 ("icmp"). If the protocol type of a given instance were "icmp," that would be represented in my new scheme as zeroes for Features 2 and 3, and one for Feature 4[1].

This is similar to the high-feature space, sparse-vector datasets for which CW learning is well suited. It is interesting to note that many of the services used in the KDD dataset are not commonly seen in network traffic today. This shows how dynamic the feature distributions of NIDS datasets are.

[1]The "service" feature had 71 possible options. This in turn created 70 new binary features. 70 of these 71 features were thus always zero for every instance. The large number of services in today's networks indicates that the features of NIDS datasets will continue to grow and become increasingly sparse. These are two excellent criteria for using CW learning.

4.3 Dataset Properties

As described by Tavallaee *et al.* (2009), the features of the KDD dataset belong to the following four groups: 1) Features that can be extracted from a TCP/IP connection; 2) Features from connections in the past two seconds that share a destination host with the connection of the current instance (labeled "same host" features); 3) Features from connections in the past two seconds that share a service with the connection of the current instance (labeled "same service" features); 4) Features that are based on content, such as the number of failed login attempts.

Furthermore, the training and testing datasets are not generated from the same probability distribution. The testing dataset also contains attack vectors that are not included in the training dataset. The purpose is to provide a realistic challenge to NIDS developers. As discussed in Section 2.2, both the feature distribution and attack vectors of NIDS data are constantly changing, so it is possible that the distribution of an earlier dataset will be different from that of a dataset collected later. Online learning is particularly attractive to NIDS research because online learning does not make assumptions about the distribution of the data (see Section 1.2).

Finally, the statistics on the KDD datasets are as follows:

- Number of training instances: 4,898,431
- Number of testing instances: 311,029
- Number of features: 122
- Number of labels: 2 (+1 for normal traffic and -1 for malicious traffic)

A disadvantage to this dataset is the low number of features (122). Ma *et al.* (2009) state that CW learning is beneficial in their dataset because their total number of features (3,231,961) is large and constantly growing. This is one of the parallels I see between malicious URL detection and NIDS research. It is thus disappointing that I cannot test CW learning on a NIDS dataset with a large number of features, but I leave this for future work. My belief is that CW learning will show high performance on this type of data.

4.4 Evaluation of CW Learning on the KDD Datasets

The final step was to run CW learning on the KDD datasets. As did Ma *et al.* (2009), I compared the results of CW learning against other online algorithms. I discuss each algorithm I use in the next sections.

4.4.1 Perceptron

Perceptron (Rosenblatt, 1958) is a linear classifier that follows the description of online algorithms given in Section 1.3.1. The update rule to the weight values of Perceptron is applied when the classifier makes a mistake, and is as follows:

$$\mathbf{w_{i+1}} \leftarrow \mathbf{w_i} + y_i \mathbf{x_i}$$

where i is a given instance. As noted in Ma *et al.* (2009), the update rule in Perceptron is fixed and thus cannot adapt its adjustments of the weight values to the degree of the mistake.

4.4.2 Passive-Aggressive Algorithm

The Passive-Aggressive (PA) Algorithm (Crammer, 2006) is the binary-classification form of the multiclass algorithm named the Margin Infused Relaxed Algorithm (MIRA) (Crammer, 2003). After receiving an instance, MIRA computes a *similarity-score* between each possible label and the instance given. The prediction is the label that has the highest similarity-score. MIRA is an *ultraconservative* algorithm in that it only updates labels with similarity-scores that were higher than the similarity-score of the correct label for the current instance.

When limited to binary classification, MIRA is the basic PA algorithm. The main difference between MIRA and PA is that MIRA is designed for separable data, whereas PA can be applied to nonseparable data. In addition, the loss bounds of PA are more general and effective than that of MIRA (Crammer, 2006).

To understand PA, it is first necessary to review the concept of the *margin* in binary classification. Building on the definitions of online learning given in Section 1.3.1, the margin m is defined as follows:

$$m = y(\mathbf{w} \cdot \mathbf{x})$$

If the binary classifier correctly predicts the true label y, the sign of the margin will be positive; otherwise, it will be negative. The margin is the distance between the instance $\mathbf{x}$ and the classification decision-boundary defined by the weight vector $\mathbf{w}$. The goal of the classifier is to maximize the margin, as this increases the distance between the instances and the decision boundary. Doing so ensures that the decision boundary does not run too close to the instances, thereby risking being on the wrong side of future data points. This is why the absolute value of the margin is labeled the *confidence* (not to be confused with the confidences of CW learning), where a higher positive value means greater confidence.

Rather than using a fixed update as does Perceptron, PA adapts its updates to the severity of the mistake by taking the margin into account. Specifically, PA updates force the margin of the current instance $\mathbf{x_i}$ to be greater than or equal to one.

The PA update rule of instance $\mathbf{x_i}$ is as follows. l_i is the loss suffered at round i after receiving the true label y. τ_i is the update factor at round i.

$$l_i = max\{0, 1 - y_i(\mathbf{w_i} \cdot \mathbf{x_i})\}$$

$$\tau_i = \frac{l_i}{|\mathbf{x_i}|^2}$$

$$update : \mathbf{w_{i+1}} = \mathbf{w_i} + \tau_i y_i \mathbf{x_i}$$

If the margin $m_i = y_i(\mathbf{w_i} \cdot \mathbf{x_i}) >= 1$, then the decision boundary is far enough away from the instance $\mathbf{x_i}$ for us to be confident in our prediction. l_i will then be zero, thus τ_i will be zero, thus $\mathbf{w_{i+1}} = \mathbf{w_i}$. This means no update is made, and is referred to as the *passive* element of the algorithm (i.e., change the classifier as little as possible).

If the margin is less than one, this means the decision boundary is too close to the instance to be confident in the prediction, or the instance is on the wrong side of the decision boundary completely. In this case, PA forces the size of the update to $\mathbf{w_{i+1}}$ to be proportional to the margin, regardless of how large the update might be. This is the *aggressive* component.

4.4.3 Confidence-Weighted Learning

A description of Confidence-Weighted (CW) learning can be found in Section 1.3.2. There are multiple variants of CW learning that address the inherent accuracy-versus-efficiency tradeoff in any learning algorithm. These variants differ in how they represent the covariance matrix of the multivariate Gaussian across all features.

As discussed by Ma *et al.* (2010), using the full covariance matrix is quadratic in the number of features, which is computationally expensive. A viable alternative is to use only the diagonal elements of the covariance matrix. This approximation is linear in the number of features, which is much more tractable. The disadvantage is that diagonal approximations lose information about the correlation between features, which can be used to decrease convergence rates.

The variants used in this study implement the diagonal approximation of the covariance matrix. They differ based on whether the optimization constraint is based on the variance or standard deviation, as well as how the covariance matrix is diagonalized. Similar to PA, the optimization constraint in CW is the value to which the margin $y_i(\mathbf{w} \cdot \mathbf{x_i})$ must be greater than or equal. The original form of CW, named *CW-Var* (Dredze, 2008) uses the variance in the optimization constraint because using the standard deviation would not be convex in the covariance matrix. The second form of CW, named *CW-Stdev* (Crammer, 2008), uses the standard deviation and maintains convexity with the covariance matrix by replacing the covariance matrix with an equivalent quantity that uses its eigenvalues.

Below are the variants of diagonal CW that I use in this study.

1) **Diagonal Standard Deviation Drop** (Crammer, 2008). The matrix is made diagonal by simply dropping off the off-diagonal elements after the update to the covariance matrix.

2) **Diagonal Standard Deviation Exact** (Crammer, 2008). The elements of the covariance matrix are projected onto the diagonal with an exact solution after the update. As specified in version 1.4 of the CW library (Dredze, 2010), the details of this algorithm are not given in the study by Crammer *et al.* (2008), so I cannot provide further details.

3) **Diagonal Standard Deviation Project** (Crammer, 2008). The full covariance matrix is projected onto the diagonal elements of the matrix after the update. This projection is performed by incorporat-

ing into the update a diagonal matrix $diag^2(\mathbf{x_i})$, in which the diagonal elements are the squares of the elements of instance $\mathbf{x_i}$.

4) **Diagonal Variance Drop** (Dredze, 2008). Same as 1), except the optimization constraint is based on the variance.

5) **Diagonal Variance Exact** (Dredze, 2008). Same as 2), except the optimization constraint is based the variance.

6) **Diagonal Variance Project** (Dredze, 2008). Same as 3), except the optimization constraint is based on the variance, and the diagonal matrix of the projection is $diag(\mathbf{x_i})$, not $diag^2(\mathbf{x_i})$.

4.4.4 Optimizations

Version 1.4 of the CW Library (Dredze, 2010) includes two command-line parameters for optimizations, "optimize_iterations" and "optimize_parameters." The iteration optimizer determines the optimal number of iterations using the following algorithm: 1) Create random splits of the data; 2) Make the first split the training data and the second split the testing data; 3) Train and test for the default number of iterations, which is 10; 4) Compute the accuracy on the testing data after each iteration; 5) Return the iteration number that had the highest accuracy.

The parameter optimizer uses a similar algorithm to determine the optimal parameter values for a given algorithm and update type. More on the parameter values used for the tests in this study is given in Section 5.2.

I evaluated each online algorithm with optimizations enabled and disabled to observe the effect of the optimizations.

4.4.5 Additional Considerations

The CW Library (Dredze, 2010) has two main options: *RunStreamTest* and *RunTrainTest*. According to the code comments, *RunStreamTest* reads in the data as a stream. This prevents the data from being loaded into memory, which can be a problem if the training data is too large for system resources.

Conversely, *RunTrainTest* loads the data into memory. Additional features of *RunTrainTest* include k-way cross validation, parameter optimization, loss output, and prediction output. I chose to use *RunTrainTest* to take advantage of the parameter optimization.

Of course, using *RunTrainTest* meant the data had to be loaded into memory. The training data was 470MB in size, which was too large for the original configuration of my Java Virtual Machine (JVM). When I first began running the code, I received JVM errors stating that the JVM did not have sufficient heap space. I resolved this problem by increasing the maximum memory available to the JVM to 1.5GB using the `-Xmx1536m` command line parameter.

My workstation only had 2GB of system memory, so this slowed it down considerably. Nonetheless, I was able to run all tests successfully (albeit very slowly, especially for the CW variants with expensive update types).

I redirected the output from these tests to text files stored on my local hard drive. After studying the output for relevant information, I wrote a Python script to parse these files and format them for easy analysis. I have synthesized the output of this Python script into the tables and figures shown in the next section.

5 Results and Analysis

The results of my tests are given in Tables 1 - 8. An "N/A" entry in an "Optimized" column means the optimal number of iterations comes before the iteration of that row. For example, iterations 2 - 10 are "N/A" in Table 1 because the optimal number of iterations for that test was 1.

There are a number of observations to draw about these results, as discussed below.

Iteration	Non-optimized	Optimized
1	0.862376177141	0.923032900469
2	0.862376177141	N/A
3	0.862376177141	N/A
4	0.862376177141	N/A
5	0.862376177141	N/A
6	0.862376177141	N/A
7	0.862376177141	N/A
8	0.862376177141	N/A
9	0.862376177141	N/A
10	0.862376177141	N/A

Table 1: Diagonal CW with `std dev drop` updates.

Iteration	Non-optimized	Optimized
1	0.825456790202	0.921901173202
2	0.825411778322	N/A
3	0.825395702651	N/A
4	0.825392487517	N/A
5	0.825392487517	N/A
6	0.825395702651	N/A
7	0.825402132920	N/A
8	0.825395702651	N/A
9	0.825395702651	N/A
10	0.825395702651	N/A

Table 2: Diagonal CW with `std dev exact` updates.

Iteration	Non-optimized	Optimized
1	0.385230959171	0.921306373360
2	0.385224528902	N/A
3	0.385224528902	N/A
4	0.385224528902	N/A
5	0.385224528902	N/A
6	0.385224528902	N/A
7	0.385224528902	N/A
8	0.385224528902	N/A
9	0.385224528902	N/A
10	0.385224528902	N/A

Table 3: Diagonal CW with `std dev project` updates.

Iteration	Non-optimized	Optimized
1	0.920309681733	0.918628166505
2	0.919882068874	0.918856441039
3	0.920319327136	0.919271193361
4	0.920669776773	N/A
5	0.920615119490	N/A
6	0.920200367168	N/A
7	0.919830626726	N/A
8	0.919538049506	N/A
9	0.919264763092	N/A
10	0.919187599870	N/A

Table 4: Diagonal CW with `variance drop` updates.

Iteration	Non-optimized	Optimized
1	0.809850528407	0.918653887579
2	0.807953599182	0.918239135257
3	0.807497050114	0.918390246568
4	0.807474544174	0.918721405399
5	0.807474544174	0.918814644293
6	0.807487404711	N/A
7	0.807487404711	N/A
8	0.807487404711	N/A
9	0.807487404711	N/A
10	0.807487404711	N/A

Table 5: Diagonal CW with `variance exact` updates.

Iteration	Non-optimized	Optimized
1	0.78789759154	0.918981831276
2	0.78787187046	0.918200553646
3	0.78846667031	0.918338804420
4	0.78846345517	0.918502776268
5	0.78846345517	0.919190815004
6	0.78846345517	N/A
7	0.78846345517	N/A
8	0.78846345517	N/A
9	0.78846345517	N/A
10	0.78844416437	N/A

Table 6: Diagonal CW with `variance project` updates.

Iteration	Non-optimized	Optimized
1	0.809072465911	0.809194641014
2	0.809004948091	0.809667265753
3	0.809062820508	0.809940552167
4	0.809426130682	0.810364949892
5	0.809570811724	0.810583579023
6	0.809734783573	0.810911522719
7	0.810040221329	0.811014407016
8	0.810117384552	0.811094785373
9	0.810098093746	0.811178378864
10	0.810127029955	0.811133366985

Table 7: Accuracies for PA.

Iteration	Non-optimized	Optimized
1	0.715078015233	0.715078015233
2	0.721865163698	N/A
3	0.756537171774	N/A
4	0.772024473602	N/A
5	0.771159602480	N/A
6	0.772944002006	N/A
7	0.776361689745	N/A
8	0.778200746554	N/A
9	0.779779377485	N/A
10	0.781917441781	N/A

Table 8: Accuracies for Perceptron.

5.1 General Accuracies

As expected, the accuracies from best to worst are: 1) CW; 2) PA; 3) Perceptron. The highest accuracies attained by each of these three algorithms are shown in Figure 1. These results mirror that of Ma *et al.* (2009).

Perceptron is the simplest algorithm because it applies a fixed update rule. As such, it cannot adjust its weight values to the degree of the mistake. As stated by Ma *et al.* (2009), Perceptron is unable to make fine-grained updates because: 1) it makes no updates after correct predictions; 2) it applies updates equally among all features. This is most likely why Perceptron has the lowest accuracy of the three algorithms.

PA improves upon Perceptron because PA's update considers the size of the margin. This avoids fixed updates by taking into account the severity of an incorrect prediction. It therefore makes sense that

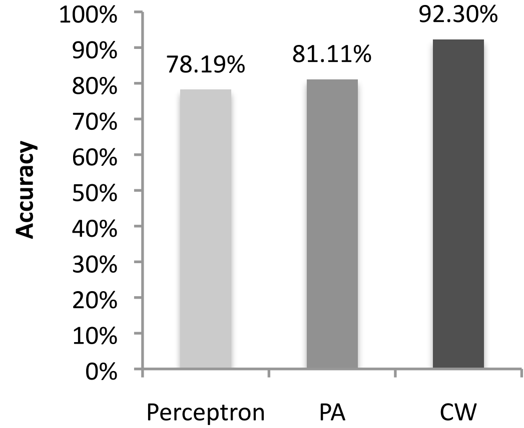

Figure 1: The highest accuracies achieved by each of the three online algorithms tested in this study.

PA's accuracy is higher than that of Perceptron.

However, both Perceptron and PA update individual features equally, regardless of how the features differ from each other with respect to the dataset. Only CW adapts its feature updates to the confidence it has in its weight values. This is presumably why CW has higher accuracies than both Perceptron and PA. These results are quite similar to those shown by Ma *et al.* (2009).

5.2 Optimizations

As discussed in Section 4.4.4, I ran all tests with optimizations enabled and disabled. The results shown in Tables 1 - 8 show the optimized number of iterations for each test. The optimized parameters for each test are shown in Table 9. Note that if the test was not optimized, the parameter values would be the defaults, which was 1.0 for both ϕ (CW) and C (PA).

The parameter ϕ for CW represents the confidence parameter, which is defined as:

$$\phi = \Phi^{-1}(\eta)$$

Here Φ represents the cumulative function of the standard normal distribution and η represents the confidence hyperparameter, defined in $\eta \in [0, 1]$ (Dredze, 2008).

The parameter C represents the aggressiveness parameter in the PA-I variant of PA (Crammer,

Algorithm	Parameter	Value
CW St. Dev. Drop	ϕ	0.1281
CW St. Dev. Exact	ϕ	0.0803
CW St. Dev. Project	ϕ	0.1351
CW Var. Drop	ϕ	0.0100
CW Var. Exact	ϕ	0.0055
CW Var. Project	ϕ	0.0055
PA	C	0.00325
Perceptron	N/A	N/A

Table 9: The optimized parameter values for each test. Note that if optimizations were not enabled, the default values for these parameters would be used. The default value was 1.0 for both ϕ and C.

2006). This parameter governs how aggressive the aggressive update step of PA should be (see Section 4.4.2).

The results shown in Tables 1 - 6 for CW tests show that both iteration and parameter optimization greatly improve CW's accuracies. This might be because the optimized ϕ values of the CW tests are all much smaller than the default value of 1.0.

Interestingly, when optimizations are enabled, all three CW `std dev` update types display the same behavior, which is 1 iteration yielding an accuracy around 0.92. Similarly, with optimizations enabled, all three CW `variance` updates types display the same behavior, which is 3-5 iterations yielding accuracies around 0.919. When viewed together, all forms of CW with optimizations enabled yield an accuracy around 0.92. This is surprising because the optimized ϕ values among the CW tests vary widely, from 0.0055 to 0.1351. I interpret this to mean there is strong consistency and similarity in how the CW variants behave when each is optimized to its fullest extent.

Conversely, optimizations do not improve the accuracies of PA or Perceptron. With PA, this makes partial sense because the non-optimized and optimized number of iterations is the same, with roughly equivalent accuracies for each. However, it is unexpected that the accuracies would be so similar with such different parameter values for C (1.0 for non-optimized and 0.00325 for optimized). It appears that C did not much influence how PA classified the KDD dataset. With Perceptron, there are no parameters to optimize because the update rule is fixed. This is why the first iteration of the optimized and non-optimized tests have the same accuracy. Consequently, there is nothing to optimize for Perceptron.

My interpretation is that the granularity by which an algorithm can adjust its updates governs how well optimization can improve it. Perceptron has by far the least scalable updates, as its updates are fixed. This is why there is nothing to optimize with Perceptron. PA improves upon this by adjusting its updates proportionally to the margin, or confidence of prediction. This might explain why it is the middle of the road between Perceptron and CW with respect to optimization benefits. Lastly, CW is by far the best in terms of granularly adjusting its updates, thus it shows the highest optimization benefit.

5.3 Convergence Rates

For this study, I define convergence rate as the number of iterations required before the accuracy stops changing. This represents the time at which the weight values no longer get updated after further iterations. Faster convergence rates are desirable for increased efficiency.

My hypothesis was that CW would demonstrate faster convergence than both PA and Perceptron because of CW's ability to selectively update its model. The data seems to support this. Neither PA nor Perceptron converges within 10 iterations. Conversely, 4 of the 6 CW variants (`std dev drop`, `std dev exact`, `std dev project`, and `variance exact`) converge within 10 iterations. Of those 4, `std dev drop` converges after the first iteration and `std dev project` converges after the second iteration. In general, all of the `std dev` variants converge, with two of them converging immediately. However, only 1 of the 3 `variance` variants converges. Strangely, `variance project` seems to converge after the fourth iteration, but then its accuracy changes on the tenth and final iteration.

The conclusion that CW converges faster than other online algorithms matches that found by Dredze *et al.* (2008). In their study, they find that multiple CW variants converge faster than PA on three popular datasets for Natural Language Processing.

5.4 CW Variants

I tested all six variants of diagonal CW, as described in Section 4.4.3. Tables 1 to 6 show several interesting trends.

Overall, accuracies for the diagonalization methods from highest to lowest were `drop, exact`, and `project`. A true diagonal covariance matrix for a multivariate Gaussian distribution signifies that the distribution is actually composed of independent Gaussians in each dimension, all of which are independent of each other. For CW, a diagonal covariance matrix means that each feature's Gaussian is completely independent from all other features' Gaussians. As discussed in Section 4.4.3, the use of the diagonal covariance matrix is an approximation designed for efficiency, but at the risk of losing information (thus accuracy) about the correlation between different features. The three different diagonalization methods differ in where they fall in the accuracy-versus-efficiency tradeoff.

Because `drop` updates have the highest accuracies, it appears there was not much correlation between features. Rather, the features appear to be independent of each other. This is because `drop` simply eliminates all off-diagonal elements of the covariance matrix. That is, the off-diagonal elements in `drop` have no influence on the model. The fact that `drop` still has the highest accuracies might mean the off-diagonal elements were not meaningful to the data. This would in turn mean the features were not highly correlated with each other[2].

`project` updates show the lowest accuracies. It appears that the projection method of diagonalization (see Section 4.4.3) is not appropriate for this dataset.

In general, `variance` updates show higher accuracies than `std dev` updates. This confuses me because `std dev` updates outperform `variance` updates in the study conducted by Crammer *et al.* (2008). It appears that for this dataset, constraints on the margin based on variance, as well as based on the covariance matrix in its original form, provide more accurate weight values during updates.

Finally, I am perplexed as to why the non-optimized accuracies of `std dev project` are so low. Perhaps it is because the default value of ϕ (1.0) was not appropriate for this dataset. The weakness of this theory is that the ϕ value of `std dev project` is closer to the default value than any other CW variant. The most plausible argument is that `std dev` performed worse than `variance`, and `project` performed worse than `drop` and `exact`, so it would make sense that their combination would produce low accuracies. This might also explain why `variance drop` has the highest accuracies when both optimized and non-optimized iterations are considered, as `variance` and `drop` have the highest accuracies in their respective families.

I attempted to step through the code with a debugger to determine why `std dev project` was performing so poorly, but I ran into a number of problems. Because the training and testing datasets were so large, I had to increase the maximum available memory to the Java Virtual Machine (JVM) to 1.5GB in order to begin running my tests. Despite many repeated efforts and querying of Internet resources, I could not get my Eclipse Integrated Development Environment to increase its maximum JVM memory. This is why I had to run my Java applications from the command line. However, I was not savvy enough with debugging Java applications from the command line (e.g., jdb) to make progress with this approach in the time I had remaining to complete the project. I could not debug the full data in Eclipse because I could not expand the maximum available memory to the Eclipse JVM.

I also attempted to debug the data in Eclipse with `RunStreamTest` (see Section 4.4.5) so that I would not have to load all of the data into memory. However, I received error messages that I had not seen when running `RunTrainTest`, to which there did not appear to be an immediate solution.

Rather than debug these errors, I chose to divide the data into smaller random splits, and then run the debugger in Eclipse on these smaller datasets using `RunTrainTest`. The downside to this was that I would be changing the statistical properties of the

[2] The counterargument would be that several of my features are binary-feature representations of the same multiclass symbolic feature, as described in Section 4.2. That is, if a feature was originally represented by more than two discrete values, I represented them by creating binary features for all possible values, then for each instance one of those features would be one and the rest would be zero. These features would not be independent of each other as a result of this expansion.

data. I proceeded anyway because it was one of my few remaining options.

I noticed a large number of messages that stated, ”Making no update because of invalid alpha,” where alpha was infinity or Not a Number (NaN). These seemed like a good place to start in debugging the low accuracies for `std dev project`. Unfortunately, I found it very difficult to precisely examine the properties of sample instances that generated this error because there were many instances that did not generate the error. I am confident that I could deduce more information on this given more time, but time did not fully permit in this study.

5.5 Comparison to Batch Algorithms

In their study, Ma *et al.* (2009) show that recently developed online algorithms such as CW can be equally as accurate as batch algorithms. This contradicts the general advice to use batch algorithms whenever possible (Dredze, 2008).

It would be quite insightful to see if the results of this study support or reject this finding by Ma *et al.*. I did not have enough time to run several batch algorithms on the KDD datasets, but fortunately Tavallaee *et al.* (2009) ran these tests already, so I can compare accuracies with their results.

It should be noted that a direct comparison between the results of Tavallaee *et al.* (2009) and this study cannot be made. Tavallaee *et al.* randomly divided the KDD training dataset into three subsets, each of 50,000 instances. Their batch algorithms were then trained on all of these subsets. Presumably the authors did this because their batch algorithms would not have been able to process the entire KDD training dataset (which is one of the main advantages of using online learning). Even though Tavallaee *et al.* and I were not training on exactly the same training data, in theory the training sets were close enough for a comparison to be valid.

As pointed out by Mark Dredze, I can attempt to train and test on the same datasets as Tavallaee *et al.* (2009) by randomly creating different splits until I observe the same accuracies on the batch algorithms as in their study. This would provide a more level comparison between CW and the batch algorithms used in their study. I leave this as future work for my project.

One further point should be made about the study by Tavallaee *et al.*. The authors note that the distribution of the KDD testing dataset is skewed, leading to inflated accuracies that prevent generalization to a realistic NIDS performance (discussed further in Section 6). The authors therefore provide an alternate dataset with a modified distribution to correct these problems. Future work of this study is to test CW and other online algorithms on the modified dataset. However, for the purposes of this study, the fact that my tests and that of Tavallaee *et al.* use (approximately) the same training and testing datasets means that our results can still be compared.

Accordingly, the accuracies for the batch algorithms of Tavallaee *et al.* and for CW are shown in Figure 2. Tavallaee *et al.* implemented their tests using the Weka (2008) collection, and used the default values for all parameters. The batch algorithms they implemented were the following: J48 Decision Tree (Quinlan, 1993), Naive Bayes (John, 1995), NBTree (Kohavi, 1996), Random Forest (Breiman, 2001), Random Tree (Aldous, 1991), Multilayer Perceptron (Ruck, 1990), and Support Vector Machine (Chang, 2001).

It can be seen from Figure 2 that CW performs comparably with all batch algorithms used by Tavallaee *et al.* This supports the results found by Ma *et al.* (2009).

6 Problems with the KDD Cup 1999 Dataset

Since it was made available, several researchers have criticized the validity of the KDD dataset for NIDS research. Tavallaee *et al.* (2009) compile the various arguments that have been made against it, which I summarize in this section.

The KDD dataset has many duplicate records. The training and testing datasets have 78% and 75% duplicate records, respectively. Tavallaee *et al.* (2009) state the following on how this affects training:

> This large amount (sic) of redundant records in the train set will cause learning algorithms to be biased towards the more frequent records, and thus prevent it from learning unfrequent (sic) records which (sic) are usually more harmful to network such as U2R attacks.

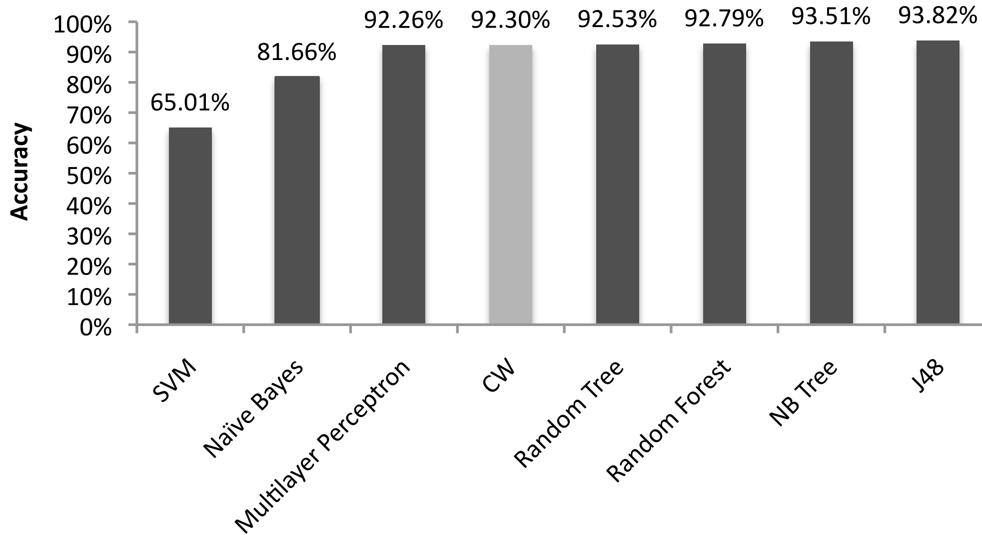

Figure 2: Accuracies of the batch algorithms used by Tavallaee *et al.* (2009) and CW. As shown, CW (an online algorithm) performs comparably to all batch algorithms used in the Tavallaee *et al.* study. This supports the results found by Ma *et al.* (2009).

It is interesting that this is exactly what CW prevents (see Section 1.3.3). This might explain the high accuracies that optimized CW shows on the KDD datasets.

Nonetheless, for algorithms that do not make adjustments similar to CW, this large number of duplicate records is indeed problematic. Furthermore, learning algorithms that have better prediction on the duplicate records will show inflated and misleading accuracies as a result.

6.1 Criticisms of the 1998 DARPA Intrusion Detection Evaluation Datasets

The following criticisms (Tavallaee, 2009) pertain primarily to the NIDS datasets generated in 1998 by MIT Lincoln Labs on behalf of DARPA (see Section 3).

Realism. The data was generated as a simulation of network traffic typical of U.S. Air Force bases. Unfortunately, there has been no substantial validation that this simulation was realistic. Moreover, the data is not similar in nature to network traffic from other production networks.

Overloading. Tcpdump was one of the primary tools to capture data. Due to the massive size of network traffic (see Section 2.2), capture tools such as tcpdump often become overloaded and begin dropping packets. However, there was no study conducted as to whether this was a problem during data collection.

Definitions. What constitutes an attack in network security varies among researchers and across environments. No rigorous definitions or constructions were made as to the nature of the labeled attacks.

Lastly, Mahoney and Chan (2003) identify simulation artifacts in the DARPA dataset by comparing its network traffic with real network traffic from a university departmental server. They show that these artifacts allow NIDS implementations to detect attacks purely from artifacts of how the simulation was performed, rather than actual characteristics of real network traffic. This leads to higher accuracy rates than can be expected in realistic environments. In addition, the simulated data does not contain the same level of garbled network traffic (e.g., IP fragments, bad checksums, and malformed arguments) that is normally observed in real network traffic. This would lead to fewer false alarms than can be expected in most production systems.

6.2 Criticisms of the KDD Cup 1999 Dataset

As described in Section 4.1, the KDD dataset was a distillation of the 1999 DARPA dataset. The KDD dataset inherited some (but not all) of the criticisms of the DARPA dataset, and earned a few of its own.

One of the main KDD-only criticisms was reported by Portnoy *et al.* (2001) and Leung and Leckie (2005). These researchers found that the distribution of attacks across the datasets was very uneven. This made their attempts at cross-validation difficult, as some of their folds would contain only one type of attack. Furthermore, Leung and Leckie discovered that 2 of the attacks in the testing data set, `smurf` and `neptune` (both Denial-of-Service attacks), were responsible for 71% of the entire testing data set. This creates a hugely uneven distribution during testing.

7 Comparison to Proposal

Overall, most of the hypotheses from my proposal were correct. CW does indeed perform well on NIDS datasets. Furthermore, CW outperforms other online algorithms, and is comparable (if not better) than many batch algorithms. This supports the findings of Ma *et al.* (2009).

Due to time constraints, I was unable to perform the same granular study as do Ma *et al.*. I wanted to test continuous versus interval-based training, as well as variable versus fixed number of features. However, testing and analyzing the results of all six diagonal CW variants required a substantial amount of time. Performing the remaining tests is future work of this study.

In addition, I wanted to test on other NIDS datasets to improve the confidence of my results. Doing so would allow me to generalize them more convincingly to the NIDS community. This is additional future work.

Lastly, I wanted to run tests with full and factored (Ma, 2010) versions of CW (not just diagonal). Once again, time did not permit, and thus this is left as future work.

8 Conclusion

The results of this study show a high potential for applying CW to NIDS research. This is due to large sizes of NIDS datasets and dynamic feature distributions. CW outperforms two commonly used online algorithms, Perceptron and PA, in both accuracy and convergence rates. In addition, CW performs comparably to many batch algorithms. These results question the general counsel that batch algorithms should always be applied when feasible.

Unfortunately, there are several issues with the KDD dataset that prevent a wide generalization to broader NIDS environments. However, the results from this study justify further inquiry into the application of CW to NIDS research.

Acknowledgments

I would like to thank Mark Dredze for providing me with implementations of the Perceptron, PA, and CW algorithms, as well as several insightful conversations on the subjects of this study.

References

ACM Special Interest Group on Knowledge Discovery and Data Mining 1999. *KDD Cup 1999: Intrusion Detector Learning*. http://www.sigkdd.org/kddcup/index.php?section=1999&method=info.

C. Chang and C. Lin 2001. *LIBSVM: A Library for Support Vector Machines*. 2001.

Chris Sinclair and Lyn Pierce and Sara Matzner 1999. *An Application of Machine Learning to Network Intrusion Detection*. Proceedings of the Computer Security Applications Conference, 1999.

D. Ruck and S. Rogers and M. Kabrisky and M. Oxley and B. Suter 1990. *The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function*. IEEE Transactions of Neural Networks, Volume 1, Number 4, 1990.

David Aldous 1991. *The Continuum Random Tree I*. The Annals of Probability, 1991.

Frank Rosenblatt 1958. *The Perceptron: A Probabilistic Model for Information Storage and Optimization in the Brain*. Psychological Review, 65.

George John and Pat Langley 1995. *Estimating Continuous Distributions in Bayesian Classifiers*. Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, 1995.

J. Ross Quinlan 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

Justin Ma and Alex Kulesza and Mark Dredze and Koby Crammer and Lawrence K. Saul and Fernando Pereira 2010. *Exploiting Feature Covariance in*

High-Dimensional Online Learning. Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy.

Justin Ma and Lawrence K. Saul and Stefan Savage and Geoffrey M. Voelker 2009. *Identifying Suspicious URLs: An Application of Large-Scale Online Learning*. Proceedings of the 26th International Conference on Machine Learning, Montreal, Canada.

Kingsly Leung and Christopher Leckie 2005. *Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters*. Proceedings of the Twentheigth Australasian Conference on Computer Science-Volume 38, 2005.

Koby Crammer and Mark Dredze and Fernando Pereira 2008. *Exact Convex Confidence-Weighted Learning*. Neural Information Processing Systems (NIPS) Foundation.

Koby Crammer and Ofer Dekel and Joseph Keshet and Shai Shalev-Shwartz and Yoram Singer 2006. *Online Passive-Aggressive Algorithms*. Journal of Machine Learning Research, 7.

Koby Crammer and Yoram Singer 2003. *Ultraconservative Online Algorithms for Multiclass Problems*. Journal of Machine Learning Research, 3.

Leo Breiman 2001. *Random Forests*. Machine Learning, Volume 45, Number 1, 2001.

Leonid Portnoy and Eleazar Eskin and Sal Stolfo 2001. *Intrusion Detection with Unlabeled Data Using Clustering*. Proceedings of ACM CCS Workshop on Data Mining Applied to Security, Philadelphia, PA, November, 2001.

Mahbod Tavallaee and Ebrahim Bagheri and Wei Lu and Ali A. Ghorbani 2009. *A Detailed Analysis of the KDD CUP 99 Data Set*. Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Security and Defense Applications (CISDA 2009).

Mark Dredze 2010. *Confidence-Weighted (CW) Library Version 1.4*.

Mark Dredze 2010. *Online Learning of Linear Classifiers*. Lecture Slides from Machine Learning course offered at Johns Hopkins University, Fall 2010 Semester.

Mark Dredze and Koby Crammer and Fernando Pereira 2008. *Confidence-Weighted Linear Classification*. Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland, 2008.

Matthew V. Mahoney and Philip K. Chan 2003. *An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection*. Technical Report, Computer Science Department, Florida Institute of Technololgy, February 2003.

Richard Lippmann and Joshua W. Haines and David J. Fried and Jonathan Korba and Kumar Das 2000. *The 1999 DARPA Off-line Intrusion Detection Evaluation*. Computer Networks, Volume 3, Issues 4, Recent Advances in Intrusion Detection Systems.

Ron Kohavi 1996. *Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid*. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 1996.

Samuel K. Small 2010. *The NIDS Aren't Alright*. Lecture Slides from Network Security course offered at Johns Hopkins University, Spring 2010 Semester.

Wenjie Hu and Yihua Liao and V. Rao Vemuri 2003. *Robusy Anomaly Detection Using Support Vector Machines*. Proceedings of the International Conference on Machine Learning, 2003.

Weka Collection 2008. *Waikato Environment for Knowledge Analysis (Weka) Version 3.5.7*. http://www.cs.waikato.ac.nz/ml/weka, June 2008.

Zheng Zhang and Jun Li and C.N. Manikopoulos and Jay Jorgenson and Jose Ucles 2001. *HIDE: a Hierarchical Network Intrusion Detection System Using Statistical Preprocessing and Neural Network Classification*. Proceedings of the IEEE Workshop on Information Assurance and Security, 2001.