**Comparative Analysis of Label Poisoning Attack Methods on SVMs**

Case Western Reserve University

CSDS 440 with Dr. Soumya Ray

July 27, 2021

Aaron Smith

axs2082

Kevin Galvan Cuesta

kxg320

**Introduction**

A commonly adopted form of machine learning algorithm is known as a Support Vector Machine (SVM). Their ability to maximize the margin between two cross sections of data makes them a powerful tool in learning scenarios. However, as with any algorithm, they can suffer as a result of adversarial attacks. For this report, we investigate adversarial learning concepts, where machine learning algorithms perform in an adversarial environment. This is a fairly broad research area and can be applied to many scenarios, but for our case, we will focus primarily on Support Vector Machines and the binary classification problem. In our scenario, a defender attempts to separate examples into one of two classes. The adversary attempts to manipulate the defender by deploying either exploratory or causative attacks. An exploratory attack is one that attempts to exploit a blind-spot or a weakness in a classifier, but doesn’t affect the training process. A causative attack is one that attempts to affect the classifier by altering the training data.

For this work, we focus on causative attacks, which primarily attack either the features, or the labels. In a data poisoning attack, the malicious party creates fake entries in the training data. Generally, a poisoning attack can be seen as a noise in the feature data. In contrast, a label flip attack maintains the original feature data and does not add any new examples, but instead impairs the classifier by flipping some number of training labels. Either of these attacks can be truly problematic for many systems in use today. For instance, in recommender systems, users are directed towards products and services that the learner has determined are of most interest to the user. However, an entity may be financially motivated to instead cause the recommender system to direct more users to a specific set of products. In an overt attack, the managers of the recommender system may notice that the classifier is performing poorly, but, as is described later, some attacks are established such that the classifier performance is not degraded in general, but instead has been altered to contain a key blindness or weak point that can be exploited by the attackers.

In the sections that follow, we describe, deploy, and evaluate two SVM attack algorithms. The algorithm described by Aaron Smith is a Label Flipping algorithm that maximally degrades the classifier performance with a limited budget of label flips. This algorithm works by iteratively solving Quadratic Programming and Linear Programming problems. The other algorithm, described by Kevin Galvan Cuesta, is a gradient ascent algorithm that modifies the example features instead of the labels. These two approaches have similar objectives, but are quite different in approach.

**Individual Report: Aaron Smith**

**Paper Summaries**

In an adversarial environment, machine learning algorithms face security obstacles that have the potential to degrade performance. This broadly applies to many different types of learning algorithms. In (Li, 2016), the authors describe several methods for attacking factorization-based collaborative filtering. These attacks occur in e-commerce applications which use recommendation and filtering systems to direct a user to a particular product. In this work, the authors assume that the attacker has full access to the learner and is also able to alter the training data. What initially drew me to this work was the focus on producing erroneous data, while mimicking normal user behavior. This seems particularly nefarious as it would be harder to detect an attack if the classifier appears to behave correctly. The authors produce attack solutions to the “alternative minimization” and “nuclear norm minimization” methods, which are noted to be common in practice.

In (Li, 2016), the authors claim three primary contributions: a comprehensive characterization of attacker utilities, novel gradient computations, and mimicking normal user behaviors. The attack model assumes there is a training data matrix of m users and n items. The attacker is allowed to add some number of extra users to the dataset and each added user is allowed to express a preference on a limited number of items. In an *availability attack*, the attacker seeks to maximize the total error of the collaborative filtering system. This is an overt attack and the operators of such a system would likely notice that the results have been diminished. An *integrity attack* seeks to raise the likelihood that a subset of items is promoted, while minimizing the overall degradation of the system so that the intrusion is less likely to be detected. In both cases, the authors use a projected gradient ascent method to attack the learner. The authors note that a possible defense strategy against these attacks is to identify changes in the correlation among features, as the attack will alter these correlations during the attack. Another possible solution includes combinatorial models, such as bagging, to mitigate the effects of the attack.

The other paper I explored, and later implemented, is (Xiao, 2012), where the authors seek to diminish a binary classifier by selectively flipping labels in the training data. Similarly to (Li, 2016), the authors in (Xiao, 2012) assume that the attacker has full access to the learner and is able to modify the training data used by the learner. In this work, the authors are only interested in maximally degrading the accuracy of the classifier and they additionally focus their efforts on Support Vector Machines. Additionally, the authors restrict themselves using a budget, such that they can only flip a limited number of training labels. The solution presented then maximizes accuracy loss for a given number of label flips. The dataset that is the result of the attack algorithm is called the tainted dataset. In the following paragraphs, I will go into detail on how this algorithm is performed and how it was implemented for this project.

**Algorithm Summary**

As noted above, I have chosen to implement the Adversarial Label Flips Attack described in (Xhao, 2012). The basis of this attack is find the tainted dataset S’ that maximizes classifier error when the classifier is training on S’, instead of the original dataset S, where . The authors note that their derived optimization problem is a bilevel problem due to the conflicting objectives between the classifier and the attacker. Additionally, a greedy algorithm that flips labels based on the current classifier is noted to be ineffective. Solving the bilevel optimization problem is known to be NP-hard and an exhaustive search is prohibitively expensive. Due to this, the authors instead attempt to find a near-optimal number of label flips. To do this, the algorithm allows the attacker to foresee the reaction of the defender to flipped labels, and the attacker only maximizes empirical loss of the classifier on the original dataset.

The Adversarial Label Flips Attack on SVMs (ALFA) algorithm accepts an original training set S, a vector of adversarial costs (or risks for each example changed), a budget parameter C, and an SVM hyperparameter . First, the function is calculated using the dual method described earlier. Then, for each example, the hingle loss is calculated and stored, while the loss of the attacker is set to a vector of zeros. The next step of this algorithm is to iteratively solve two different optimization problems. The first attempts to minimize the difference in errors between the attacking and defending models. This problem

is a Linear Program (LP), where is a value used to select either the original training label, or a flipped label. The next problem is a Quadratic Program (QP) which minimizes the attacker model by selecting a set of weights and biases that perform well under the new dataset S’. The optimization function is shown below.

The newly calculated attacker error is calculated and then fed back into the optimization problem above. This process iterates until the q vector converges to a solution. Once the vector q converges, the values are used to identify which example labels to flip by sorting the indices of the vector q according to the values of q in descending order and flipping labels starting from highest to lowest, until we’ve reached our budget C. At this point, we have our new tainted training set S’.

As was shown in class, SVMs can be used on data that is not linearly-separable by including a slack term. This form is generally called the soft-margin problem. Fortunately, this problem can be written as a Quadratic Programming (QP) in the dual. This is useful because there are many efficient tools available to solve QPs and at the optimal point, the dual maximizes to the primal’s minimum. To derive the dual form, we start with the primal problem

.

In simple terms, the primal problem attempts to minimize the width of the margin plus the sum of the slack terms, or examples within the margin, where gamma is a hyperparameter that chooses how heavily we penalize mistakes. From the primal, we can calculate the lagrangian by bringing the constraints up into the minimization as

.

From this, we can calculate the partial derivative of the Lagrangian with respect to w, , b. The Karush-Kuhn-Tucker (KKT) conditions tell us that the optimal point will happen at a location where the partial derivatives are zero. Additionally, the KKT conditions tell us that our slack variables will only be non-zero when we are at the boundary. Setting the partial derivatives to zero, and then plugging the solution back into the Lagrangian results in

.

Where J(is the dual function and is maximized over alpha such that each alpha value is between [0, ] and the sum of the products of . Due to the complementary slackness condition in KKT, we know that either (the Lagrange multipliers) or . This means that our support vectors are the examples where . The classification function can then be defined using only the results of the kernel function K, the slack variables , the support vector labels , and a bias b, as shown below. The class prediction is then. In this project, we only implemented the linear kernel, but in general, a SVM projects the input features onto a feature space via a function phi. However, we can use the kernel trick to circumvent ever knowing the possibly high dimensional function phi, if we assume that a kernel K is a Mercer Kernel, which satisfies .

**Implementation Details**

For this project, we were restricted to the libraries that were provided in the environment.yml file. When reading (Xhao, 2012), I noticed that all three of the major aspects of their algorithm were either Linear or Quadratic Programs. This seemed like a good fit since we were provided CVXOPT, which is a convex optimization library that efficiently solves LP and QP problems. The three problems are: the SVM dual form (QP), Equation 10 from (Xhao, 2012) (LP), and Equation 11 from (Xhao, 2012) (QP). The first problem was solved in class and the process of converting the primal problem to a dual problem was described above. However, CVXOPT uses the standard QP form and so in the file SVM.py, in the fit() function, I work through the algebra of converting the lecture results into the standard QP structure.

Equation 11 from (Xhao, 2012) is an LP problem and again, CVXOPT assumes the standard form. Optimization function in this problem is to minimize over q . Since the error terms are fixed here, and gamma is a scalar, this is simply where where . Then, collecting the inequalities, there are three inequality expressions that need to be gathered into a matrix G and a constraint vector h. Similarly, there are equality constraints that CVXOPT summarizes in the form Ax=b. As was done with G and h, some matrix stacking is implemented to summarize the equality constraint in matrix A and vector b. Once this is complete, the function calls the CVXOPT LP solver to solve the optimization problem. The implementation of this function can be found in the file *LabelFlipAttack.py* in the function *\_solve\_lp\_eq11()*.

The last of the three optimization functions is from Equation 10 in (Xhao, 2012). This function is written in *LabelFlipAttack.py* in *\_solve\_qp\_eq10().* This QP problem is similar to the SVM objective function, except that each error term is multiplied by a variable . Writing down the Lagrangian, and taking the partial derivative of the Lagrangian w.r.t. the error and setting it equal to zero results in . Due to the similarity of Equation 10 and the SVM, I simply used the SVM class to solve the SVM problem, with the modification that this gamma was a vector formed by , where q is the vector of values.

This project was written using Python and primarily used CVXOPT, Numpy, argparse, os, and Matplotlib. The data was obtained from LIBSVM at <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>. The datasets we tested were ‘breast-cancer,’ ‘breast-cancer\_scaled,’ ‘heart,’ ‘elections,’ and ‘foodstamp.’ To convert the LIBSVM datasets to numpy arrays, I used the function *libsvm\_to\_numpy().*  Additionally, a synthetic dataset was constructed using the code in *make\_synthetic\_data().*

**Experiment description**

Evaluations were performed as follows. First, the Evaluator.py main function is run using the command-line. The arguments in the command-line establish the dataset, the attack algorithm, an optional feature normalization, and the flip budget C. Once the data is loaded into feature and label numpy arrays, the data is optionally normalized and cleaned of nan values, then I perform cross validation using 5 folds. For each fold, the training data is run through the Adversarial Label Flip Attack algorithm (or passed without alteration if ‘none’) and the tainted data is used to train a new SVM. This new SVM is then used to classify the fold test data and the accuracy is measured and stored. After all folds are complete, the accuracies are averaged together and the variance of the accuracy is calculated. The results are formatted and logged in the ‘logs’ folder.

**Research Extension**

The algorithm implemented in this project only sought to attack a classifier. The authors in both (Li, 2016) and (Xhao, 2012) expressed that learning to optimally attack a classifier could help us better understand how to build resilient training and operation strategies. An obvious extension would be to tackle the objective of being resilient to such attacks. This analysis would extend beyond the theory of machine learning and should include an analysis of the systems and processes used by the enterprise deploying the learned models. Both of these works were based on the assumption that the attacker has full knowledge of the algorithm’s response to data and is capable of modifying the data prior to training. This is a steep requirement that can be subverted by obfuscating the learner from the outside world. Additionally, security protocols need to be in place to prevent unauthorized users from accessing the data used to train the learners. I would be interested in studying the application stack of such a system to better identify the physical and electronic vulnerabilities to the enterprise.

**Conclusions**

The Adversarial Label Flip Attack generally degrades the classification ability of the classifier. Additionally, due to efficient solvers present in the CVXOPT library, the algorithm ran quickly, taking only a few minutes to complete on a small Macbook Air. However, the authors in (Xhao, 2012) note that the algorithm should converge monotonically, which did not happen in all cases. It was assumed that this is because the QP solver stopped early due to reaching a maximum iteration count.

It was found that scaling significantly improved both the performance of the SVM and the performance of the attack algorithm. This may have been obvious, as it was discussed in class, but it was nice to see that scaling the feature values to be within [0,1] had a positive effect on training. Additionally, scaling the features greatly improved the runtime of the LP and QP solvers. When features were not scaled (i.e. ran without the --normed flag), the CVXOPT solver would tend to run to max iterations and would in general take significantly longer to complete.

To demonstrate the results of the LabelFlipAttack algorithm, I generated a synthetic dataset of 1000 examples on a feature space of size 2. The function that created this dataset is defined in utils.py make\_synthetic\_data(). In the Figure 1 below, I show the original data, which contains no label noise. I then split the data into a training (200 examples) and testing (800 examples) datasets. I then passed the training dataset through the label flip algorithm and flipped 25 examples. The examples that were flipped are shown in Figure 2. Finally, I trained an SVM on the real data and another SVM on the tainted data. I compare the accuracy and the decision boundaries in Figures 3 and 4. The script that was used to generate these figures is in the jupyter notebook “Demo LabelFlipAttack.ipynb”.

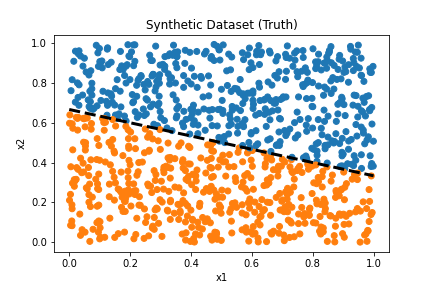


Figure 1

A synthetic dataset with no label noise.

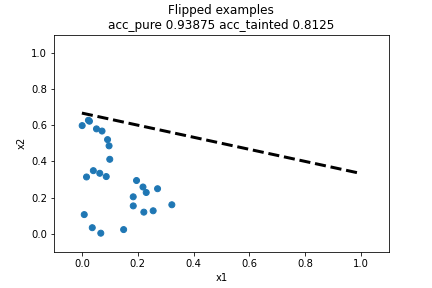


Figure 2

The 25 examples that were flipped by the Adversarial Label Flip Attack algorithm.

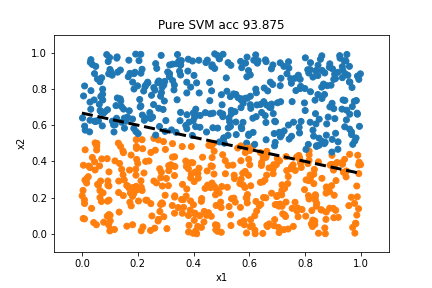


Figure 3

The test data which was classified by SVM trained on untainted data.

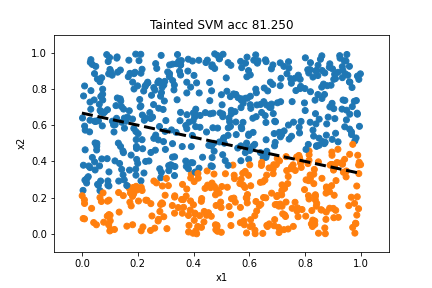


Figure 4

The test data which was classified by the SVM trained on tainted data.

In the tables below, I report the results of the experiment on each of the five datasets. For each dataset, an SVM was trained on the real dataset, an SVM was trained on the tainted dataset, finally another SVM was trained on a dataset that was tainted via randomly selecting C=25 labels to flip. From these results, we can see that the ALFA algorithm performs better on some datasets. For example, the breast-cancer dataset does not see any degradation from the ALFA algorithm and random flipping performs equally. Contrast this with the breast-cancer\_scale dataset results and we see that scaling plays an important role in the ALFA algorithm and the SVM in general. In the elections\_1000 and heart datasets, the AFLA algorithm degrades performance by more than 5% with only 25 label flips and the AFLA algorithm degrades the classifier performance by more than simple random flipping. Finally, in the foodstamp dataset we see that the results are 87.4% in all cases, which is likely because the classifier is always picking the same label. The foodstamp dataset has 87% positive labels, which is also the resulting classifier accuracy.

| DATASET: breast-cancer |  |  |
| --- | --- | --- |
| No data manipulation | ALFA tainted dataset | Randomly flipped labels |
| Mean Accuracy: 0.6500  Accuracy Variance: 0.0018 | Mean Accuracy: 0.6526  Accuracy Variance: 0.0023 | Mean Accuracy: 0.6500  Accuracy Variance: 0.0018 |

| DATASET: breast-cancer\_scale |  |  |
| --- | --- | --- |
| No data manipulation | ALFA tainted dataset | Randomly flipped labels |
| Mean Accuracy: 0.9707  Accuracy Variance: 0.0001 | Mean Accuracy: 0.9269  Accuracy Variance: 0.0005 | Mean Accuracy: 0.9707  Accuracy Variance: 0.0002 |

| DATASET: elections\_1000 |  |  |
| --- | --- | --- |
| No data manipulation | ALFA tainted dataset | Randomly flipped labels |
| Mean Accuracy: 0.7010  Accuracy Variance: 0.0009 | Mean Accuracy: 0.6691  Accuracy Variance: 0.0027 | Mean Accuracy: 0.6980  Accuracy Variance: 0.0052 |

| DATASET: foodstamp\_1000 |  |  |
| --- | --- | --- |
| No data manipulation | ALFA tainted dataset | Randomly flipped labels |
| Mean Accuracy: 0.8740  Accuracy Variance: 0.0000 | Mean Accuracy: 0.8740  Accuracy Variance: 0.0000 | Mean Accuracy: 0.8740  Accuracy Variance: 0.0000 |

| DATASET: heart |  |  |
| --- | --- | --- |
| No data manipulation | ALFA tainted dataset | Randomly flipped labels |
| Mean Accuracy: 0.6037  Accuracy Variance: 0.0111 | Mean Accuracy: 0.5296  Accuracy Variance: 0.0053 | Mean Accuracy: 0.5889  Accuracy Variance: 0.0079 |

**Individual Report: Kevin Galvan Cuesta**

**Biggio Poisoning Attacks background**

While I personally analyzed a few articles from Battista Biggio’s works, my main focus was on his 2012 paper, “Poisoning Attacks against Support Vector Machines”. Biggio has a wide range of expertise in poisoning different kinds of learning algorithms and so I will summarize the extent of his works in relation to poisoning attacks in adversarial environments:

**Pre-2012:** Biggio has some previous work on poisoning different, more simpler algorithms (Logistic Regression, Naive Bayes, etc.) and here is where he begins his theoretical work in the design and creation of adversarial environments for learning networks. (Biggio, Fumera, and Roli, 2011 & 2012).

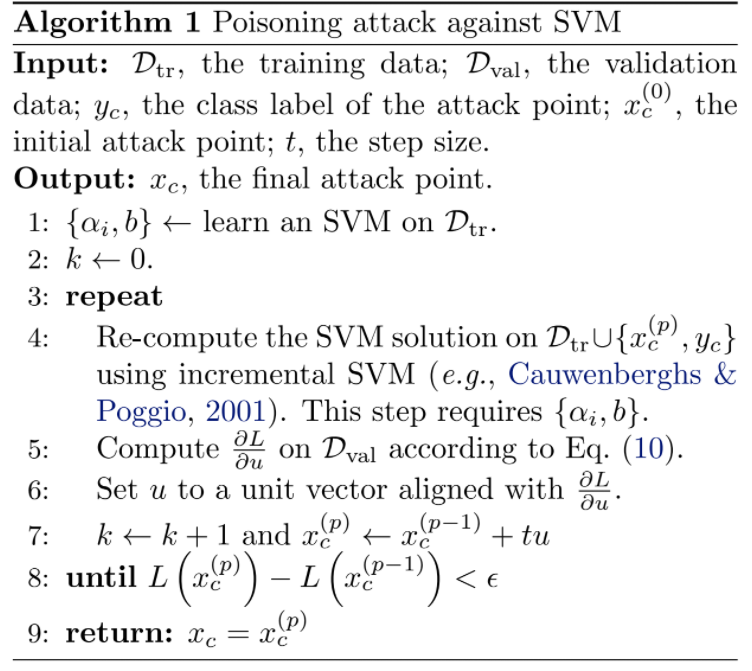
**2012:** This is likely his most influential paper of Biggio’s work at the time. Studying Support Vector Machines, he created the robust 2012 paper covering several kernel forms in this work (Biggio, Nelson, and Laskov, 2012). While the article itself was not released until 2013, I will reference it as the ‘2012 paper’ as it was first presented at the ICML 2012 conference.

**Post-2012:** Biggio continues his work oftentimes in more difficult and more adversarial settings. He drops some of the assumptions made in earlier papers about the knowledge an adversarial agent has on the defending algorithm where his earlier works included complete information. He notes that his earlier works may not always be the most realistic interpretation of common scenarios but does produce the upper bound on possible attacks in poisoning settings. Moreover, he extends his works into neural networks via deepfool, and works in Support Vector Machine extending his model from gradient ascent where the kernel of the classifier is known, to Projected Gradient Descent (Biggio and Roli, 2018).

While Biggio’s works are by no means a complete library of the work done in the area of poisonings on Support Vector Machines, he is one of its most extensive contributors, often working with top authors.

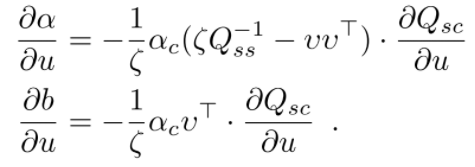
As we are the creators of the Support Vector Machine in our paper, our previously obtained knowledge about its functionality made the 2012 paper a prime pick. Moreover, since our objective is the comparison of algorithms, an algorithm under some of the best assumptions is a good place to start. As he stated in the ICML 2012 presentation, these assumptions should set a relative upper bound on the performance of algorithms

**Algorithm Implementation**

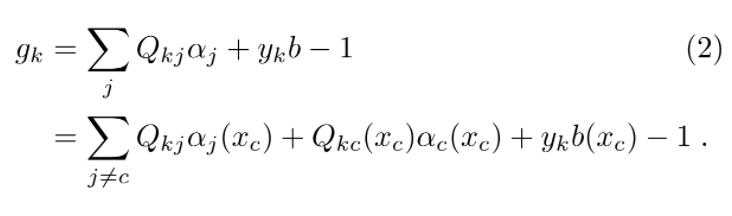
The algorithm implemented is labeled as “Algorithm 1” in the 2012 paper (below). Using both the training and validation data sets, the algorithm begins by first computing the alpha values on our SVM in reference to the original, unpoisoned data set. It then begins to iterate through the examples, continuously removing the previously used example vector and replacing it with a newly generated, tainted vector. This technique is known as incremental SVM and was outlined in Cauwenberghs & Poggio’s 2001 paper. However, it should be noted that the computational cost of this algorithm is quite high as it recomputes the SVM at every iteration inserting the . The next step is to compute the maximizing hinge loss gradient vector with respect to a single example. Normally, this is a minimization problem, but as we are trying to decrease the accuracy, we must maximize the hinge loss. We then add this vector back into our feature matrix multiplied by the hyperparameter, t or step size.

The gradient direction was calculated via the following method:

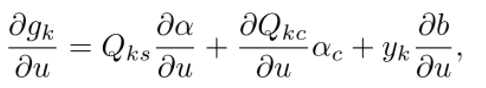
By using the following inverted matrix independent of the sample xc, a particular example vector that is the target of poisoning,



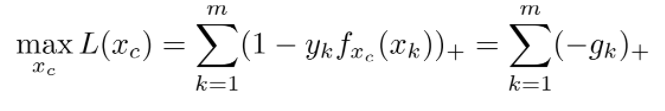
And substituting it into the derivative of the following equation that accounts for all the changes that the poisoning target may have on the margins of the Support Vector Machine:



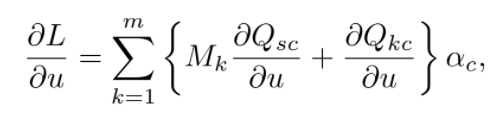
However, one might notice that this function is not differentiable over all points. As a result, the above equation has to change form to perform iterative gradient ascent (hence the need for iterative SVM). By using the norm vector, u, as a representation of the attack direction we can find the following derivative:



Finally, we can maximize the hinge loss using the previous component and substituting:



And we get the gradient attack:



The iterations continue until the difference between the current tainted set and its last update becomes too small. Due to the size of our datasets, this point never reached and I instead set the maximum number of iterations roughly equal to the length of the array of alpha produced from the validation set on SVM.

The implementation of the algorithm itself only used the numpy library, but it did rely heavily on the SVM implementation which additionally used cvxopt. All the code was produced in Python and results were reported in text logs.

**Research Extension**

The objective of SVM is to minimize both the hinge loss and level of inaccuracy that is present in the final algorithm. The algorithm at play attempts to do exactly the opposite by calculating the hinge loss present at a given example in the training set and appending the changed, poisoned vector in place of the original. However, if the model were changed to instead minimize the hinge loss with respect to a particular example, and then go in the direction of the negative gradient, would it be possible to increase the accuracy of the SVM? While quite unlikely, it does follow the similar gradient descent methods in the feature space that are typically reserved for the Support Vector Machine to compute its outputs. I hypothesize a minimal marginal increase is possible if I run the reverse algorithm. In essence, this algorithm should be performing a data cleaning procedure that pushes data points that are outside of a linearly separable space into their respective side of the hyperplane.

While I did not get time to implement these ideas, I did consider changes to the step size. One idea I had was to remove the step size all together, and instead use the length of the gradient vector in its place. Since the gradient direction tells us which direction results in the most change, it should push us most towards the desired loss. Since the gradient vectors were converted into unit vectors in the direction of the gradient, this effect was not present. If this were the case, we would expect the inverse to have the opposite effect, allowing the SVM to perform better at classification. My idea was to try both the gradient length and the inverse and compare the results to the step sizes shown below.

**Experiments**

I ran quite a few experiments tweaking small parts of the algorithm to see how much of a difference they would cause. The change that seemed to have the most effect was changing the step size over the same sample. As it is a pre-defined hyper parameter in the algorithm and is used to calculate the change for any specific one of the examples, it was a clear place to compute experiments. For example, with elections\_1000 I attained the following results:

| DATASET: elections\_1000 |  |  |
| --- | --- | --- |
| step\_size = 0.1  m\_examples: 1000  label 0.0: count 413  label 1.0: count 587  Fold 0 SVM with 661 SV.  Fold 1 SVM with 661 SV.  Fold 2 SVM with 662 SV.  Fold 3 SVM with 663 SV.  Fold 4 SVM with 664 SV.  mean acc. 0.6953  var acc. 0.0007 | step\_size = 0.5  m\_examples: 1000  label 0.0: count 413  label 1.0: count 587  Fold 0 SVM with 663 SV.  Fold 1 SVM with 666 SV.  Fold 2 SVM with 662 SV.  Fold 3 SVM with 664 SV.  Fold 4 SVM with 669 SV.  mean acc. 0.6350  var acc. 0.0005 | step\_size = 1  m\_examples: 1000  label 0.0: count 413  label 1.0: count 587  Fold 0 SVM with 662 SV.  Fold 1 SVM with 662 SV.  Fold 2 SVM with 663 SV.  Fold 3 SVM with 664 SV.  Fold 4 SVM with 664 SV.  mean acc. 0.6360  var acc. 0.0006 |

The data sets above took my algorithm quite a bit of time to run, so I opted out of taking a sample for each. Moreover, the more I conducted experiments, the less information I seemed able to take away from the number of support vectors, so I decided to drop them from the following tablesSince the smaller datasets were able to run at a quicker pace, I ran the following experiments on the heart dataset:

| DATASET: | heart | m=270 |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step size | .1 | .25 | .5 | .75 | 1 |
| MeanAccuracy | 0.6185 | 0.5889 | 0.5259 | 0.5481 | 0.5407 |
| Var Accuracy | 0.0064 | 0.0024 | 0.0146 | 0.0130 | 0.0134 |

The research experiments were done on the heart dataset, so I added them below. The following were computed using the reverse algorithm:

| DATASET: | heart | m=270 |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step size | .1 | .25 | .5 | .75 | 1 |
| MeanAccuracy | 0.6519 | 0.6333 | 0.5926 | 0.5519 | 0.5519 |
| Var Accuracy | 0.0080 | 0.0062 | 0.0059 | 0.0147 | 0.0147 |

Next, are the experiments conducted on the breast-cancer\_scaled dataset:

| DATASET: | breast-cancer\_scaled | m=683 |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step size | .1 | .25 | .5 | .75 | 1 |
| MeanAccuracy | 0.9489 | 0.9177 | 0.8938 | 0.9164 | 0.9003 |
| Var Accuracy | 0.0017 | 0.0008 | 0.0026 | 0.0020 | 0.0018 |

**Conclusions**

What stood out the most was the weird behavior that appeared to occur under step size. I had originally predicted that it would follow somewhat of a quadratic function with the bottom resting at some point around t =0.5. While t= 0.5 did often have the lowest accuracy (meaning the algorithm performed the best), the tailing results did not seem to fit a clear pattern. Instead of a quadratic function, the behavior of the graphs appeared to be asymptotic. Yet, even though appearing to reach a minimum level of accuracy, the behavior didn't seem consistent in that in some cases (i.e. heart) the accuracy would both rise and fall as the step size continued to grow. It seems too large a difference to be a result of randomness, yet I find much difficulty in explaining the observation.

Overall, the poisoning attacks did do quite a good job at producing the tainted datasets and were thus effective at diminishing the ability of an SVM to accurately classify a label given its training data. Some data sets produced almost imperceptible results, so to avoid repetition, they are displayed in the following section. The results for the original SVMs were also included in the final section for space purposes.

The research extension provided probably the most surprising results. I had originally expected each of the graphs to follow an asymptotic pattern or quadratic pattern, but instead would approach its best performance and show inconsistent results thereafter. However, this pattern did seem to be present in the research extension. More surprisingly though is that the step size seems to have maximized the level of error that was present. As a result I received the same exact results for step size 0.75 and 1.

After seeing the totality of the results, it became clear why the graphs showed the pattern that they did. Since we are taking the gradient iteratively, and only in relation to particular examples, it is not the gradient vector of the entire function. My hypothesis is that the large step sizes could cause the algorithm to jump over the maximums for the hinge loss. If the feature was already creating noise that the SVM was struggling to handle, the updated figures could have completely pushed it in a new direction (given a big enough step size), possibly one that increased the performance measure relative to the other losses.

**Comparative Analysis and Discussion**

The AFLA algorithm and the GradientAscent algorithm both attempt to degrade a binary classifier by altering the training data before it is used to train an SVM classifier. However, the AFLA algorithm does this by flipping C labels and it finds the ‘nearly-optimal’ solution using a pair of Quadratic and Linear Programs. The Gradient Ascent algorithm instead attacks the features of the examples by maximizing the hinge loss in relation to a particular example. This difference makes it so that a direct comparison of the algorithms is not possible. For instance, if the budget for the AFLA algorithm is set to C=25 flips, what would an equivalent budget look like for the Gradient Ascent algorithm which modifies features? The gradient ascent algorithm runs iteratively instead, looking to get the difference between two affected SVMs trained on different poisoned data sets below a threshold. In most cases, we expected the gradient ascent procedure to perform better given that the budget was a large limiting factor on the ability of the ALFA algorithm to successfully poison the dataset.

In the tables below we collect the results of our algorithms on the five datasets. It is clear that the degradation caused by the Gradient Ascent algorithm was equal to or greater than the degradation caused by the AFLA algorithm, and this was consistent with our original hypothesis. As we already noted the AFLA algorithm was restricted to 25 label flips, while no such ‘budget’ restriction was placed on the Gradient Ascent algorithm.

We also found that some datasets were not susceptible to either of the two algorithms. For example, in the tables below, we see that the foodstamp\_1000 dataset had an SVM accuracy of 87% and that did not change when running the attack algorithms. This is likely due to the SVM being unable to classify this data using a linear classifier. The foodstamp\_1000 dataset has unequal priors, with the negative label being present in 87% of the examples. It seems likely that the classifier has learned to simply produce the negative label in all cases and ignores the features. In the breast-cancer dataset, we see the importance of scaling as both algorithms performed poorly on the unscaled version of the data, while performing better on the scaled version.

| DATASET: breast-cancer |  |  |
| --- | --- | --- |
| Gradient Ascent | AFLA | Regular SVM |
| 0.6477 | 0.6529 | 0.6500 |

| DATASET: breast-cancer\_scale |  |  |
| --- | --- | --- |
| Gradient Ascent | AFLA | Regular SVM |
| 0.8938 | 0.9269 | 0.9707 |

| DATASET: elections\_1000 |  |  |
| --- | --- | --- |
| Gradient Ascent | AFLA | Regular SVM |
| 0.6350 | 0.6691 | 0.6980 |

| DATASET: foodstamp\_1000 |  |  |
| --- | --- | --- |
| Gradient Ascent | AFLA | Regular SVM |
| 0.8740 | 0.8740 | 0.8740 |

| DATASET: heart |  |  |
| --- | --- | --- |
| Gradient Ascent | AFLA | Regular SVM |
| 0.5259 | 0.5296 | 0.5889 |

**References**

Biggio, B., Fumera, G., Roli, F.: Multiple classifier systems for robust classifier design in adversarial environments. Int’l J. of Machine Learning and Cybernetics 1(1), 27–41 (2010)

Biggio, B., Fumera, G., Roli, F.: Design of robust classifiers for adversarial environments. In: IEEE Int’l Conf. on Systems, Man, and Cybernetics (SMC), pp. 977–982 (2011)

Biggio, B., Nelson, B., Laskov, P.: Poisoning attacks against support vector machines. In: Langford, J., Pineau, J. (eds.) 29th Int’l Conf. on Mach. Learn. (2012)

Biggio, B., Fumera, G., Roli, F.: Security evaluation of pattern classifiers under attack. IEEE Trans. on Knowl. and Data Eng. 99(PrePrints), 1 (2013)

Biggio, B., Roli, F.: Wild Patterns: Ten Years after the Rise of Adversarial Machine Learning. Pattern Recognition, pp. 317-331 (2018)

Cauwenberghs, G., & Poggio, T. Incremental and Decremental Support Vector Machine Learning. Adv. Neural Inf. Process. Syst.. 1. (2001)

Li, B., Wang, Y., Singh, A. Vorobeychik, Y. (2016). Data Poisoning Attacks on Factorization-Based Collaborative Filtering. 1608.08182

Shoag, D. 2019. *Foodstmps*. Built micro datasets from full sources. Original author(s) unknown. Case Western Reserve University.

Shoag, D. 2019. *Peri\_IV\_data*. Built micro datasets from full sources. Original author(s) unknown. Case Western Reserve University.

Xiao, Han & Eckert, Claudia. (2012). Adversarial label flips attack on support vector machines. 242. 870-875. 10.3233/978-1-61499-098-7-870.

**Appendix**

The following are sampled data sets from Daniel Shoag’s lecture series on Econometrics. Original authors unknown.

**FoodStamps:** FoodStamps.dta has a projected label, y of food stamp recipiency. This variable outputs a 1 if a given observation (a single person) was a recipient of a food stamp assistance, and an output of 0 if they were not.

The post-cleaning data set included feature information such as hours worked in the previous week, the health status ranging from 1-5 (poor to excellent), employment status (working, unemployed, or not looking for work), marital status of 5 options (widowed, married, married but separated, etc.), and binary variables for seeing and hearing difficulties.

Some ‘random’ variables were left in the cleaned data set because we believed the noise they produced would be of interest. One of these variables included household numbers. If 5 people were in a household, each person would be assigned a number at random. Note that this is not completely random, but would still lead to a lot of noise in the dataset.

**Elections:** Elections.dta had a label vector, y stating the outcome of the next governor election. It output a 1 if a democrat won the seat and a 0 if a republican won.

A wide variety of numerical demographic data, such as percentage of black voters, percentage of foreign voters, voters from total population percentage, and adjacent election features such as the most recent election win, the previous election win, bipartisanship in state congress, etc.

This data set was selected as it would be one with a high level of inaccuracy even if there was little noise in the data set. This is a result of the difficulty of predicting elections and we believed it would provide interesting results.