Final Report

Credit Card Fraud

Team Members:

Utkarsh Agarwal (uagarwal@purdue.edu)
Dimple Dhawan (dhawand@purdue.edu)
Kalpan Jasani (kjasani@purdue.edu)
Jonathan Poholarz (jpoholar@purdue.edu)

Introduction

With the rise of the internet and net-banking, card transactions are common, everyday occurrences to many citizens and companies in the world today. Though, as commonplace as these transactions may be, they simultaneously have the potential to wreak havoc on the financial system and lives of those impacted by them. As such, identifying fraudulent transactions is an important area of study for banks and financial companies in order to protect themselves and their users from disaster. This is especially important as many institutions have hundreds or thousands of transactions each day. Think of stock trading and investment companies for example, managing millions of accounts.

By obtaining a dataset, courtesy of Kaggle (https://www.kaggle.com/mlg-ulb/creditcardfraud/home), consisting of credit card transaction data by European credit cards in 2013, we analyze various approaches for detection of fraudulent transactions. We investigate the application of machine learning algorithms in order to develop a classification system that can distinguish between fraudulent and non-fraudulent transactions.

Our solution is to use and analyze two Support Vector Machine (SVM) classifiers – linear, primal SVM and non-linear, dual SVM classifiers. Hence, we developed a binary classifier – fraudulent vs. non-fraudulent. In this process, we trim the data to a computable size, train the classifiers, and tune hyperparameter variables (values we may adjust to produce a better result).

First, we transform our dataset into a usable size. Next, we present visualizations of the input data. Then, we perform hyperparameter tuning. Last, we discuss our results and conclude this report.

Outline

Data transformation

Given the large size of our dataset (~300,000 transactions), we obtain a smaller sample in order to train our classification in a reasonable amount of time and without over-fitting the classifier to our data.

There are significantly fewer fraudulent data points in this dataset (0.17%), so taking a simple random sample would leave us with very few fraudulent transactions as well as contribute to over-fitting. Hence, we have decided to select all of the fraudulent transactions as well as a small subset of normal transactions. By doing this, we hope to build a strong classifier well trained at detecting fraudulent transactions. While this does not reflect the true proportion of such transactions in the real world, taking advantage of all of the fraudulent transactions will help us build a better classifier. We take a total of 1000 transactions (from the ~300,000), out of which 492 transactions are fraudulent and 508 are non-fraudulent.

Does the size of our subset affect our results? To determine this, we have also analyzed the effects of our training data through k-fold cross validation. Hence, we have two more datasets, one of 250 samples and another of 500 samples.

To summarize, our sample files include:

1. 250.csv
2. 500.csv
3. 1000.csv

Shuffling and Cleaning Data

The non-fraudulent samples will be randomly selected from the entire data set. Then, from this 1000 point sample set, we select a random subset of 500 points (250 from each class) and a random subset of 250 points (125 from each class).

We first perform an initial filtering of the 284,807 samples to obtain a set of 1000 samples where 492 of them are fraudulent and the remaining 508 are non-fraudulent. The initial filtering procedure is implemented in the python scripts getSamples.py followed by shuffleData.py.

getSamples.py generates a list of 508 random numbers representing different points. Then, it traverses all 284,807 points, selecting the 508 randomly chosen non-fraudulent points as well as the 492 marked fraudulent points.

shuffleData.py performs additional tuning from the selected subset. This includes transformations such as changing the values of the fraudulence classifier from 0s and 1s into 1s and -1s. This file also shuffles the points to produce a more random ordering than was left from the initial selection process. A similar process is performed for the 500 and 250 point subsets.

In summary:

1. A sample of 1000 points, 492 fraudulent and 508 non-fraudulent, are obtained.

2. These points are shuffled and finally stored in 1000.csv in our dataset folder. This is the frozen dataset that we plan on using for the rest of our analysis.

These steps are also followed for the 500 point and 250 point samples.

Note on Loading Data

We use the function "getData" to load input, specified by the filename. As we reduce the dataset, we use the file /dataset/1000.csv.

To use the function:

```
import getData
X, y = getData(<filename>)
```

For replicating results and performing tests, please perform the following.

In a Python console, run:

```
import project
import getData
X, y = getData.getXY("../dataset/1000.csv")
# use functions such as project.makePCAGraph(X, y)
```

Visualizing input data

Because our data points possess a significant number of features (30 features), we employ principal component analysis in order to visualize the data. This allows us to display data points in two dimensions, convenient for a graphical representation.

Principal component analysis

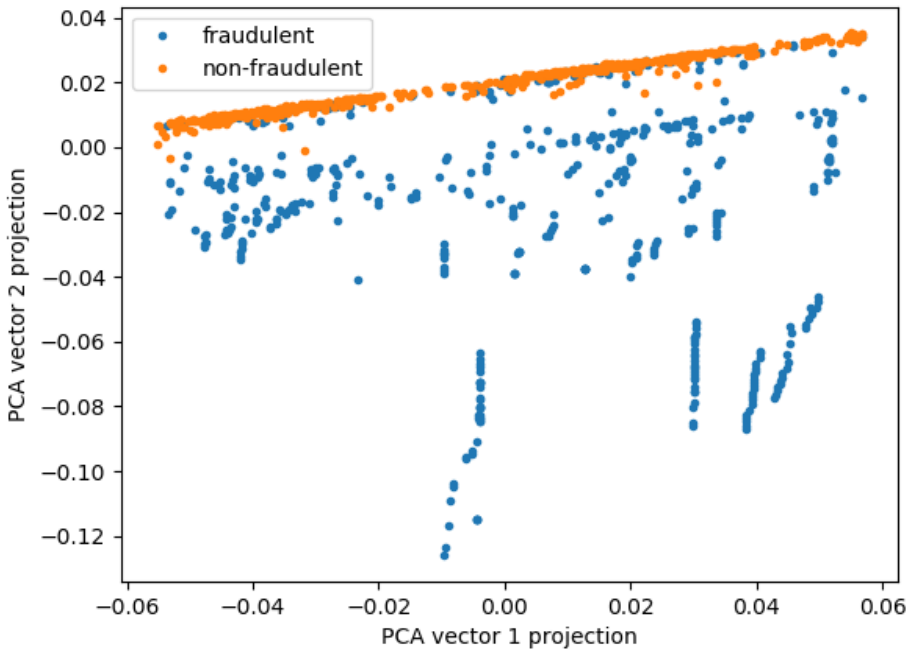


Figure 1: Orange – Non-Fraudulent data points, and Blue – Fraudulent points.

The graph shows the projected data on the first two principal components. The graph depicts some separation or clustering of the two sets of samples.

To replicate results, use:

```
project.makePCAGraph(X, y)
```

We decided to perform Myopic Forward Fitting on our set of 30 features to determine if there were dominant features useful for classification of the data and for best visualizing the data. The two features ranked most important were features 0 and 3. Surprisingly, feature 0, which we know to be the time of the transactions, seems to be rather important. This is in contrast to our initial hypothesis that time would not be useful in classifying the fraudulence of transactions. Feature 14 is the third most important feature, and, when plotted against feature 3, results in an interesting visualization of the data.

In regards to our data, it should be noted that feature 0 (time) is defined as the "number of seconds elapsed between a transaction and the first transaction in the dataset" (From the description of the dataset online)

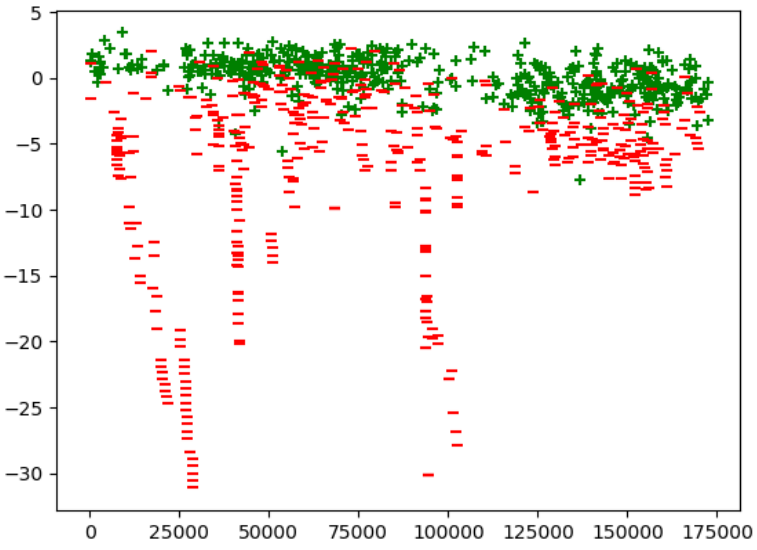


Figure 2: Plot of feature 0 (x axis) vs feature 3 (y axis).
"-" → Fraudulent Transactions
"+" → Non-fraudulent Transactions

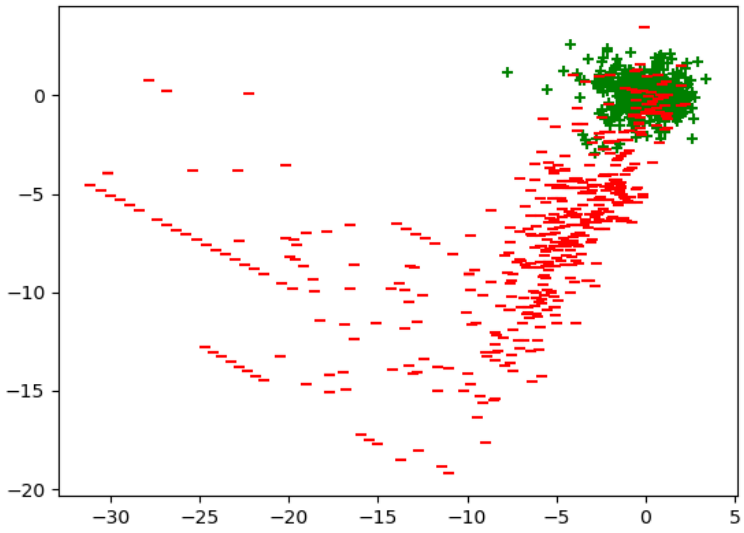
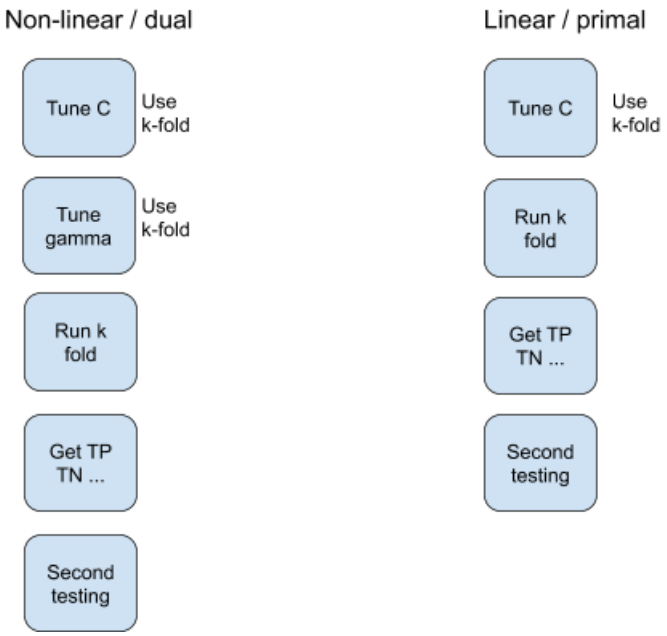


Figure 3: Plot of feature 3 (x axis) vs feature 14 (y axis). Although producing a slightly lower accuracy score, plotting features 3 vs 14 gives a good visualization of the clumping of non-fraudulent data points. Many of the fraudulent points are separable from non-fraudulent transactions.
"-" → Fraudulent Transactions
"+" → Non-fraudulent Transactions

Outline for hyper parameter tuning and testing procedures



Hyperparameter tuning

Primal SVM

We will use linear primal svm for data classification model building. To do this, we will utilize the inbuilt machine learning library for the Python programming language named Scikit-learn. Specifically, the function svm.LinearSVC with dual set to false will be used.

To run the code and find errors, you can use our kfoldcv.py function, with the appropriate keyword arguments. Details will be provided below.

Dual SVM

We also use radial basis kernel dual svm for non-linear data classification and model building. Again, Scikit-learn will be used here.

k Folds Cross-Validation

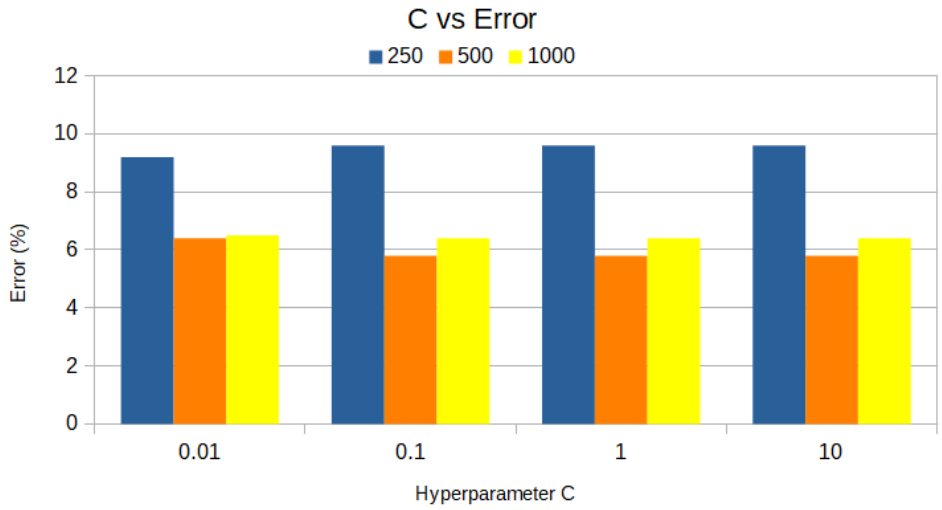
We have implemented a run function in kfoldcv.py which performs our k-fold cross validation, returning an array z containing the error percentages for each trial.

Primal SVM

Each step of our training process utilizes our three subsets of the data (the subsets of 250 points, 500 points, and 1000 points) to see if the sample size made a difference in any of the results. The following graphs will color code these samples.

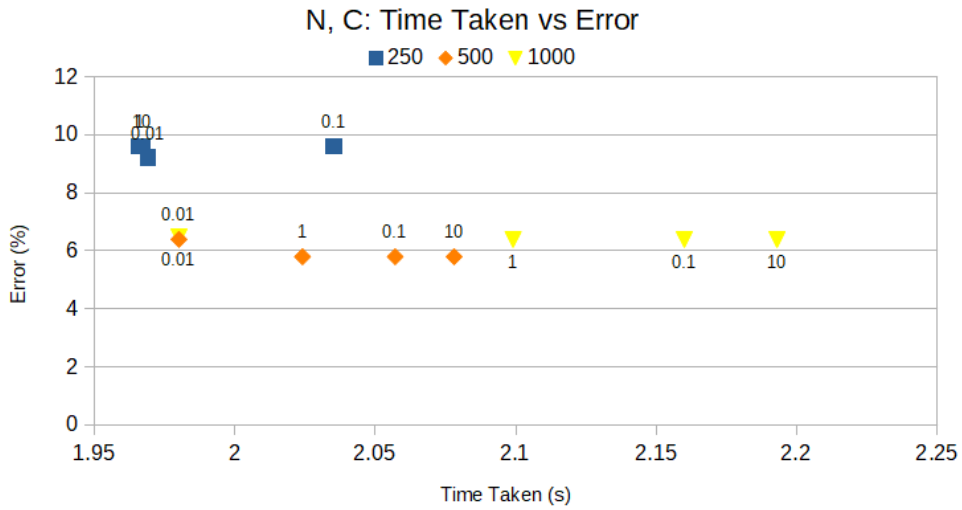
Starting with the Primal SVM method, we work through our training and testing process. We test 4 different values for the hyperparameter C: 0.01, 0.1, 1.0, and 10.

Note: Tabular format of graph data is available in the separate file titled 'Appendix' as well as sample output that yielded the tabular data. Graphs were generated with the spreadsheet program LibreOffice Calc from the data we collected.



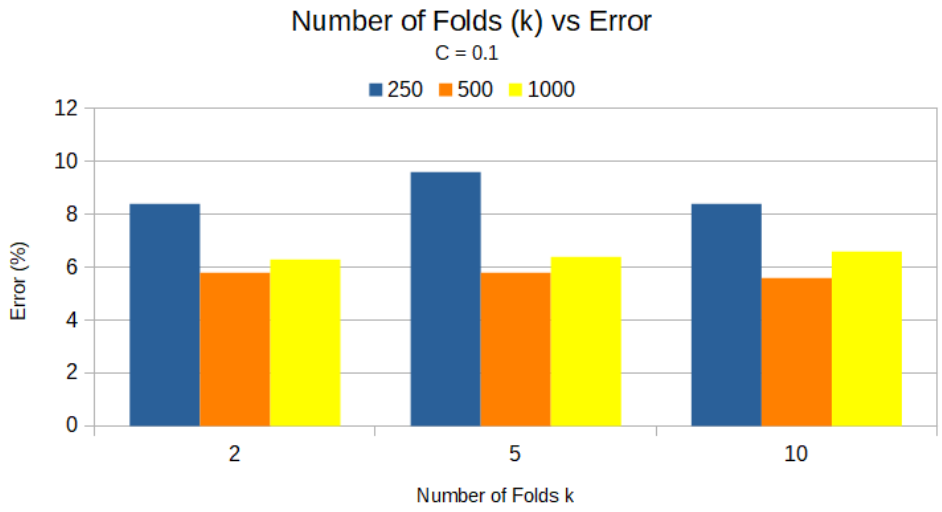
The value of C did not have too large of an impact on the error output. The error for a C value of 0.1 was a bit less than 0.01 and about the same as 1 and 10. These results are fairly consistent across the sample sizes with slightly larger error for the 250 sample size. The smaller sample may not have enough points to effectively classify the data when compared to the larger samples. Since the C value does not impact the results to a large degree, we deduce that the data may be linearly separable.

We also track the running time of the algorithm to compare the trade-off between the error and the hyperparameter.



Increasing the value of the hyperparameter, although not improving our error values, does increase the overall time for the algorithm. For this reason, we felt that a C value of 0.1 is our best option as it minimizes both the running time and error results.

With a C value chosen, we move on to optimizing the value of k, the number of folds in our cross validation. We test k values of 2, 5, and 10.

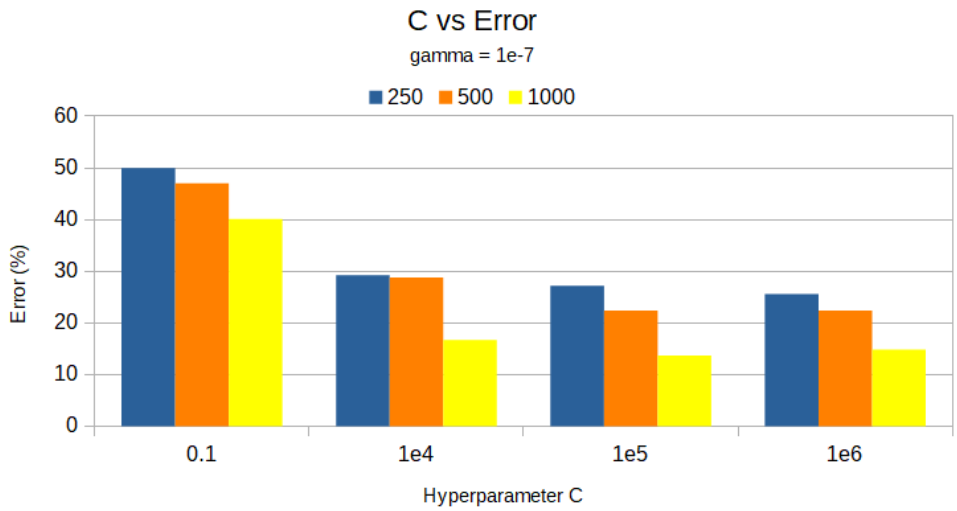


The error rates are fairly consistent across the number of folds. With this information, we know that when working with Primal SVM for classifying our data, we should use a C value of 0.1 and any number of folds.

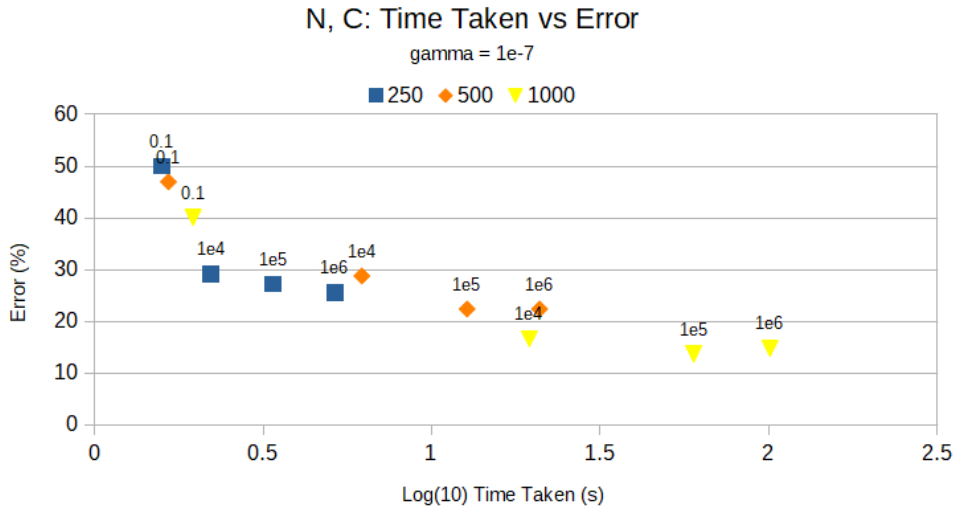
Dual SVM

Moving on to the Dual SVM method, we have additional parameters to test. To avoid having to work with x^4 or x^5 parameter combinations, we validate the best value of one parameter before moving onto the next and the next. Starting with C, we chose reasonable values of gamma (1e-7) and k (5 folds) that did result in long run times or unreasonable error values. We test values of C starting at 0.1, then increasing by powers of 10 until reaching 1,000,000 (1e6). Values of C between 10,000 and 1 are omitted, but 0.1 has been left in to show the relative error values.

Note: Tabular format of graph data is available in the separate file titled 'Appendix' as well as sample output that yielded the tabular data.

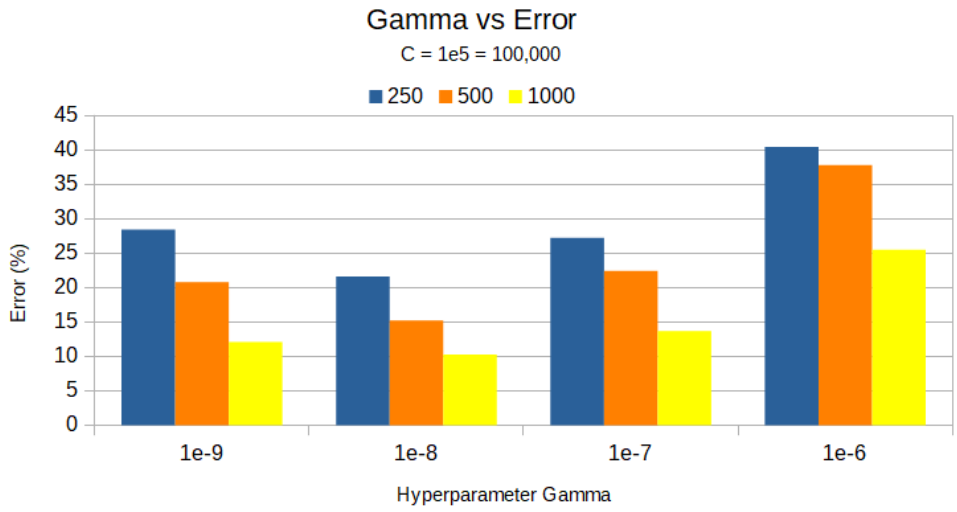


A C value of 1e5 gives the best error results at just less than 1e4 and 1e6. Compared to Primal SVM, the 1000 point data set seems to perform significantly better than the 500 point data set. This raises initial concerns about potential over-fitting to the data. As running time is also a concern, we also measure these values and plot them for analysis.

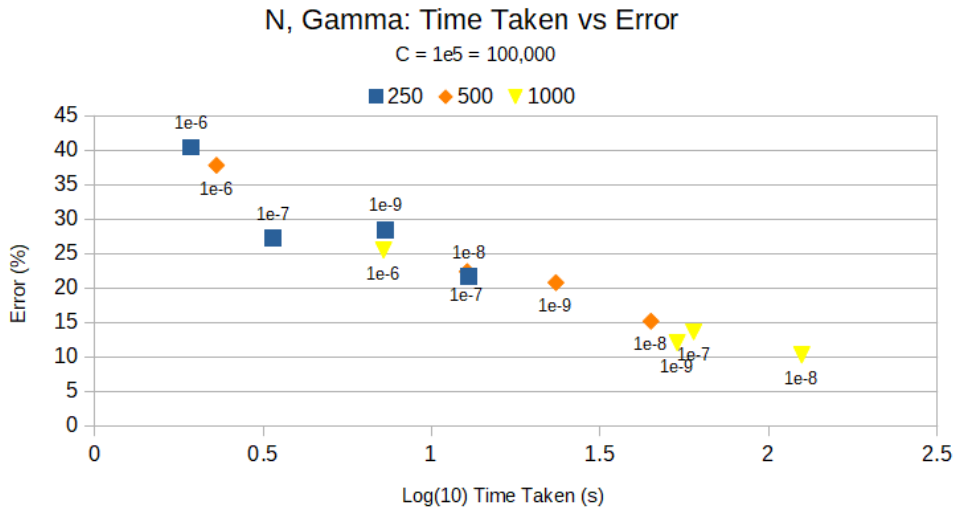


For viewing purposes, the time taken has been transformed logarithmically (base 10). Although it gives a much better error rate in testing, the 1000 point data set takes around a minute and a half for its best C value of 1e5. This is much larger than the 500 point data set but still reasonable to work with. The time taken for an even larger sample to train the data would likely be a concern in a real world scenario.

With a C value chosen of 1e5, we move on to selecting an optimal Gamma value. The best values of Gamma are found to be fairly small. We will show data for 1e-9, 1e-8, 1e-7, and 1e-6.

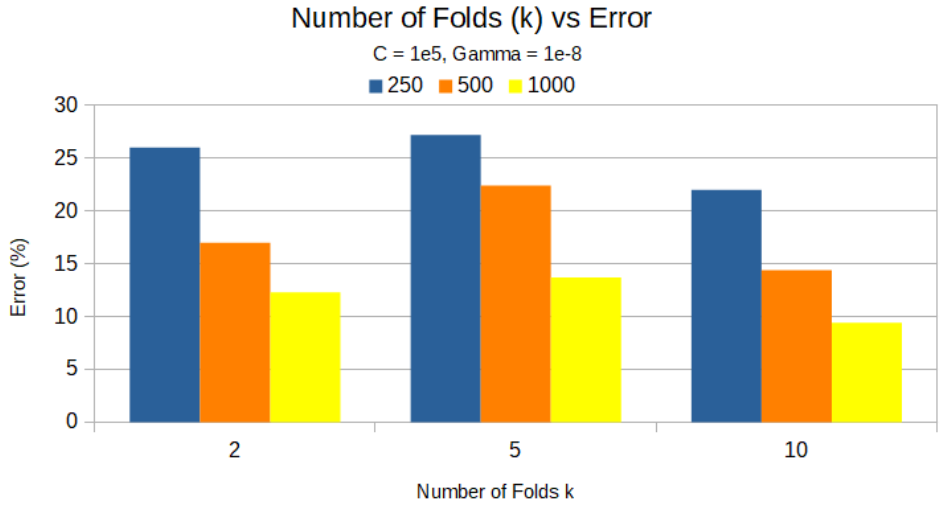


Similar to testing for C, the 1000 point sample performs better than the smaller samples when comparing the error rates. Of these tested values and utilizing our chosen C value of 1e5, Gamma value of 1e-8 produces the best results, beating out its neighboring values of 1e-9 and 1e-7.



In regards to time, our best value of 1e-8 takes longer for all three of our data subsets. Again, the time scale has been transformed logarithmically. Interestingly, the two adjacent gamma values tested both take less time in their computations than our best choice 1e-8 at around two minutes.

After selecting both C and Gamma, we experiment with different values of k, the number of folds in our cross-validation. As with the Primal SVM, we test values 2, 5, and 10.



However, unlike Primal SVM, the number of folds makes more of a difference on the error values. Using 10 folds yields the best results. We hypothesize that more folds may produce even lower error values, but increasing folds beyond 10 produces a significant hindrance in run-time. For this reason, we recommend sticking to a lower fold count of 10 or 5.

In summary, the best values for the Dual SVM parameters are 1e5 for C, 1e-8 for gamma, and 10 folds (k).

Results

Results on k-fold cross validation procedures and hyper parameter tuning

Algorithm 1 (Linear, Primal SVM):

C (regularization parameter): 0.1

Fold size selected for k-fold cross validation: 5

Size	Error
250	9.6
500	5.8
1000	6.4

Algorithm 2 (Non-linear, Dual SVM):

C (regularization parameter): 100000

Gamma for radial basis kernel: 0.00000001

Fold size selected for k-fold cross validation: 5

Size	Error
250	21.6
500	15.2
1000	10.3

Performance Measures

Algorithm 1 (Linear, Primal SVM):

		Predicted	
		Non-fraudulent (+1)	**Fraudulent (-1)**
Actual	**Non-fraudulent (+1)**	True Positive = 503	False Negative = 5
	Fraudulent (-1)	False Positive = 60	True Negative = 432

Table: Performance on same dataset with primal svm

Accuracy: 0.935
Precision: 0.893428063943
Recall: 0.990157480315
F1 Score: 0.939309056956

Algorithm 2 (Non-linear, Dual SVM):

		Predicted	
		Non-fraudulent (+1)	**Fraudulent (-1)**
Actual	**Non-fraudulent (+1)**	True Positive = 505	False Negative = 3
	Fraudulent (-1)	False Positive = 33	True Negative = 459

Table: Performance on same dataset with dual svm

Accuracy: 0.964
Precision: 0.938661710037
Recall: 0.994094488189
F1 Score: 0.965583173996

		Predicted	
		Non-fraudulent (+1)	**Fraudulent (-1)**
Actual	**Non-fraudulent (+1)**	True Positive = 245	False Negative = 5
	Fraudulent (-1)	False Positive = 20	True Negative = 222

Table: Performance on half testing and half training with Primal SVM, C=0.1

Conclusion

From our results, we determine that we can successfully apply a classification algorithm for fraudulent and non-fraudulent transactions. The error for the Linear/Primal SVM is smaller than the Non-Linear/ Dual SVM. In addition, the former takes significantly less time to run. This allows us to conclude that a Linear classifier is a better fit for this use case.

In the real world, False Negatives (in which a transaction is incorrectly believed to be real when it was actually fraudulent) are a significant concern as people may be scammed out of thousands or millions of dollars. From our results, we not that our classifier produces a very small False Negative error ratio and conclude that we have produced a beneficial and rather safe classifier.

The number of False Positives is also rather low. Although being wary of transactions may produce a few extra alerts to users, it will also make it seem that our system is doing work to protect them from fraud. This benefit should outweigh the annoyance of verifying with customers if their purchases are accurate.

Appendix

Data collection

The 1eX format refers to some number times 10^X power.

Primal SVM - Tuning C				
N	**C**	**Folds**	**Error (%)**	**Time (s)**
1000	0.01	5	6.5	1.980
500	0.01	5	6.4	1.980
250	0.01	5	9.2	1.969
1000	0.1	5	6.4	2.160
500	0.1	5	5.8	2.057
250	0.1	5	9.6	2.035
1000	1.0	5	6.4	2.099
500	1.0	5	5.8	2.024
250	1.0	5	9.6	1.966
1000	10	5	6.4	2.193
500	10	5	5.8	2.078
250	10	5	9.6	1.967

Primal SVM - Tuning (k) Folds				
N	**C**	**Folds**	**Error (%)**	**Time (s)**
1000	0.1	2	6.3	2.079
500	0.1	2	5.8	2.017
250	0.1	2	1.941	8.4
1000	0.1	5	6.4	2.160
500	0.1	5	5.8	2.057
250	0.1	5	9.6	2.035
1000	0.1	10	6.6	2.090
500	0.1	10	5.6	2.029
250	0.1	10	8.4	2.019

Dual SVM - Tuning C					
N	**C**	**Gamma**	**Folds**	**Error (%)**	**Time (s)**
1000	0.1	1e-7	5	40.1	1.957
500	0.1	1e-7	5	47.0	1.655
250	0.1	1e-7	5	50.0	1.587
1000	1e4	1e-7	5	16.7	19.489
500	1e4	1e-7	5	28.8	6.201
250	1e4	1e-7	5	29.2	2.214
1000	1e5	1e-7	5	13.7	59.877
500	1e5	1e-7	5	22.4	12.739
250	1e5	1e-7	5	27.2	3.385
1000	1e6	1e-7	5	14.8	100.854
500	1e6	1e-7	5	22.4	20.899
250	1e6	1e-7	5	25.6	5.152

Dual SVM - Tuning Gamma					
N	**C**	**Gamma**	**Folds**	**Error (%)**	**Time (s)**
1000	1e5	1e-9	5	12.1	53.493
500	1e5	1e-9	5	20.8	23.37
250	1e5	1e-9	5	28.4	7.308
1000	1e5	1e-8	5	10.3	125.303
500	1e5	1e-8	5	15.2	44.723
250	1e5	1e-8	5	21.6	12.901
1000	1e5	1e-7	5	13.7	59.877
500	1e5	1e-7	5	22.4	12.739
250	1e5	1e-7	5	27.2	3.385
1000	1e5	1e-6	5	25.5	7.193

500	1e5	1e-6	5	37.8	2.297
250	1e5	1e-6	5	40.4	1.936

Dual SVM - Tuning (k) Folds					
N	**C**	**Gamma**	**Folds**	**Error (%)**	**Time (s)**
1000	1e5	1e-7	2	12.3	32.404
500	1e5	1e-7	2	17.0	13.847
250	1e5	1e-7	2	26.0	4.227
1000	1e5	1e-7	5	13.7	59.8777
500	1e5	1e-7	5	22.4	12.739
250	1e5	1e-7	5	27.2	3.385
1000	1e5	1e-7	10	9.4	341.265
500	1e5	1e-7	10	14.4	162.458
250	1e5	1e-7	10	22.0	29.917

Files submitted

Our zip folder submitted has the following two folders – dataset and scripts.

dataset/

1. 1000.csv
2. 500.csv
3. 250.csv

scripts/

1. getSamples.py
2. shuffleData.py
3. getData.py
4. kfoldcv.py
5. Myopicfitting.py