CS185C:
Final Project
Malware Classification

Jordan Conragan, Brett Dispoto

May 18, 2020

Contents

I Abstract and Introduction **3**

1 Abstract **3**

II Methodology and Dataset **3**

2 The Dataset **3**

3 Preprocessing the Dataset **3**

III Experimental Setup **4**

4 K-Nearest Neighbors **4**
4.1 KNN Procedure 4

5 Support Vector Machines **5**

6 K-Means Clustering **5**

7 HMM Boosting **5**
7.1 HMM Boosting Procedure 5

8 Stacking **5**
8.1 Stacking Procedure 5
8.2 A Stacking Framework 6

9 HMM Bagging **7**
9.1 Bagging Procedure 7

IV Experimental Results **8**
9.2 KNN Results 8
9.3 (Primitive) Support Vector Machine Results 9

10 K-Means Clustering **10**
10.1 Boosting Results 11
10.2 Stacking Results 12
10.3 Bagging Results 13

V Discussion and Future Work **14**

Part I
Abstract and Introduction

1 Abstract

In this work we provide experiments and analysis of various multiclass classification algorithms to identify families of malicous software (malware). The best result was achived using stacking as an ensembler method, with an average of 98.2% accuracy on the three largest malware families: winwebsec, zbot, and zeroaccess. Additionally, the ROC curve for each family have an area under curve of at least 0.998. The stacking framework is made up of one support vector machine (SVM) per malware family, each trained on scores from $N = 5$ "boosted" HMMs per Malware family. Each SVM was trained and tested using a one-versus-rest methodology. Further hyperparameters for this model include: `max_unique_opcodes`= 35, `c_budget`= 2, `max_iters`= 100, and `gamma`= 2^{-16}. In addition to the stacking experiments and tuning, several experiments were performed with varying methodologies and relative levels of success. Clustering experiments were performed using a reduced feature set, using K-Means (init=KMeans++, 46.66% accuracy) and K-Nearest-Neighbors ($K = 1$, 94.99% accuracy) algorithms. Multiclass supervised classification experiments were performed on this reduced feature set using a SVM, and best results of 75.8% accuracy was achived with a radial kernel of degree 3. Finally, bagging and boosting experiments were performed using Hidden Markov Models as classifiers to ensemble. Bagging achived high accuracy of 99.7%; however, the results cannot be mathemathical derived for reasons described in the paper. Boosting experiments were not successful because it is too difficult to naively discover a threshold for scoring when random seeds are used across HMMs.

Part II
Methodology and Dataset

2 The Dataset

The dataset used is the Malicia dataset published by the International Journal of Information Security. The dataset is comprised of several malware family subsets all of which contain several opcode sequences of malware from that family. The only malware families with enough samples to perform the machine learning techiniques describe here are:

1. WinWebSec
2. ZBot
3. ZeroAccess

The dataset is unbalanced, with there being about a $1:1$ ratio for the sample count for WinWebSec as compared to the number of samples from ZeroAccess and ZBot conbined. Class imbalance is addressed according to which model is in use. For example, when using K-Means algorithm, class weights are used.

3 Preprocessing the Dataset

Most of the machine learning techniques used in this report use variants of the same preprocessing steps. Here are the preprocessing steps taken for the methods described in this report. Preprocessing was done using both python and bash scripts. The relevant files can be found in the `preprocessingScipts` directory of our submission.

1. Download the dataset (`malicia` — provided by Fabio Di'Troia).

2. Split the dataset into directories based upon their family label. (Already completed by the dataset provider.)

3. For each malware family, the following steps were then taken:

   (a) Read through all of the files, count the occurrence of each unique opcode across all files.

   (b) Take the n (turning parameter) most common Opcodes, and convert them to an ASCII symbol for observation symbols for our HMMs. The Opcodes which are not within the n most common will be converted to an "other" symbol. This will reduce noise in our model.

   (c) Once each opcode is assigned a symbol, we again read through the files and convert the opcodes to symbols.

      i. If bagging is being used, make copies of **each** converted malware file, which will later be split up accordingly during training.

      ii. Otherwise, if boosting or stacking is being used, we can simply dump the converted opcodes (symbols) for the entire family into one huge file. This file will be our observation sequence.

Part III
Experimental Setup

4 K-Nearest Neighbors

K-Nearest Neighbors (KNN) was one of the first algorithms implemented in this work. KNN makes a wise first choise because it allows us to examine the dataset, potentially for intuitive solutions to building a model.

KNN was implemented following according to the following features of our samples:

1. Entropy of opcode sequence, and

2. Number of distinct opcodes

KNN is performed using the files `KNN.py` and `knnPreprocesor.py`. The procedures and results are described in the sections below. *sklearn* was used as a KNN library for python.

4.1 KNN Procedure

Procedures for running the KNN algorithm on the dataset is as follows:

1. Preprocess the dataset. This includes:

   (a) Convert all opcodes to symbols,

   (b) Count the number of distinct symbols in each sample,

   (c) Compute the entropy $H(X)$ of the sequence of opcodes. This is given by Shannon's Entropy:

$$H(x) = E[I_X] \tag{1}$$

   (d) Write the above information in a file for each sample.

2. Use 90% of the dataset as our "training" samples (there really is no training in KNN).

3. Test KNN using 10% of the dataset.

4. Repeat steps 2 and 3 for different values of K, ranging from 2 to 10. Keep the best result.

5 Support Vector Machines

Given that the dataset had already been preprocessed to be represented in two dimentions from the KNN experiments, we figured we might as well feed this into support vector machines and see if they might be able to better partition the classes when the dataset is viewed from the perspective of other dimensions/ coordinate systems. In addition to reducing the observations to two features, we also:

1. Normalize our dataset because it makes computing distances for SVM much faster.
2. Use class weights inversely proportional to their frequencey in the dataset. This will help deal with the unbalanced dataset.

Finally, SVMs were tested with several kernels, each for multiple degrees. Degrees were tried in the range $[2, 5]$. The kernels tested were:

1. linear,
2. polynomial, and
3. radial

6 K-Means Clustering

We also figured that we might as well experiment with K-Means clustering in addition to SVMs and KNNs in our experiments consisting of the two reduced feature sets. The K-Means clustering algorithm was supplied by sklearn, and provides hyperparameter tuning on attributes of the model such as the number of random resets of centroids, number of clusters, and initial position of centroids.

7 HMM Boosting

7.1 HMM Boosting Procedure

Boosting was done according to the procedure described in lecture. That is, all HMMs of a malware family were trained using the same observation, only with different training parameters (random seeds) varying across models. *HiddenMarkovModel.java* and *Boosting.java* are the files where boosting is implemented.

8 Stacking

8.1 Stacking Procedure

Beause stacking yielded the best results among all tested models, most extensive hyperparameter tuning was performed using stacking. Stacking is implemented using the files *Stacking.java*, *Stacking.python*, *HiddenMarkovModel.java*, in addition to the library LibSVM. Stacking is implemented using HMMs as weak classifiers, then using SVMs as metaclassifiers, using "one-vs-everything else" methodology to achive multiclass classification using support vector machines.

The procedure for each component of the classifier:

HMM Training Below is the methodology of training each pre-ensembled HMM for stacking.

Model	**Input**
HMMs for winwebsec	1000 files
HMMs for zbot	1067 files
HMMs for zeroaccess	651 files

8.2 A Stacking Framework

In order to efficently perform experiments based on empricial results, we have developed a framework (desktop application) which allows for rapid hyperparameter tuning and testing.

Classifying The program is Stacking.py in the Stacking folder of the project. When you run the program on Windows, either in the command line with python Stacking.py or by running the main function with an IDE, a file dialog window opens up and you must choose a malware file to classify. The program will output the family that it thinks the virus most likely belongs to, or 'None' if it doesn't believe that the malware belongs to a family that the program has been trained on.

Training Before the file classification can be used, however, the models used by the program must first be trained. This can be achieved by running `python Stacking.py --train` in the command line. By default, the model will be trained on 3 families with the highest number of files. However, one can specify the number of families that the model is trained on by using the `--train <number_of_files>` option. For example, to train the model on 5 different families, one should run `python Stacking.py --train 5`. Depending on the performance of the machine, it could take several hours to completely train the model. If it is interrupted, it unfortunately has to be retrained from scratch. The more families included in training, the longer it will take to train the model used by the program.

After the model has been trained, the program is ready to classify malware files, but it is a good idea to know how accurate the model is. This can be seen by running `python Stacking.py --test` in the command line. It will print the test accuracy for each family, and it will also produce a single graph containing an ROC curve for each family. Testing the model takes significantly less time than training it. Finally, running `python Stacking.py --all` will train the model with the default number of families, test the model as if the `--test` option was set, and then prompt the user to chose a file for classification. After some manual and unscientific testing, I found that the program works well when classifying a malware sample from one of the families that the model was trained on. However, it has difficulties classifying a file that did not come from any of the training families as 'None'.

SVM Training Below is the methodology of training each SVM for stacking.

Model	**Input**
SVM for winwebsec	1. Scores from 5 HMMs of 1089 winwebsec files and 2. 1720 total files from zbot and zeroaccess
SVM for zbot	1. Scores from 5 HMMs of 533 zbot files and 2. 2832 total files from winwebsec and zeroaccess
SVM for zeroaccess	1. Scores from 5 HMMs of 325 zeroaccess and 2. files and 3206 total files from winwebsec and zbot

Testing SVM Below are is the methodology of testing each SVM for stacking.

Model	Input
SVM for winwebsec	1. Scores from 5 HMMs of 1089 winwebsec files and 2. 1719 total files from zbot and zeroaccess
SVM for zbot	1. Scores from 5 HMMs of 533 zbot files and 2. 2831 total files from winwebsec and zeroaccess
SVM for zeroaccess	1. Scores from 5 HMMs of 326 zeroaccess and 2. files and 3247 total files from winwebsec and zbot

9 HMM Bagging

Bagging is performed using the files `Bagging.java` and `HiddenMarkovModel.java`. The procedures and results are described in the sections below.

9.1 Bagging Procedure

The following steps were performed in order to use bagging as an ensemble method:

1. Split assembly instructions into their appropriate family, and translate the instructions to HMM symbols,
2. Within these family folders, use a shell script (included, `makeTesting.sh`) in order to split 10 percent of the samples into a test set.
3. Train x HMMs on each family,
4. Once each HMM for a given family is trained, score the observation sequence which just used to train the model.
5. Write down this score.
6. Once all HMMs of a given family are trained, use whichever ensembler aggregate function to take all the generated scores and aggregate them into one score.
7. Once training is complete for all families, for each family we now have this "aggregate score" written down in a log in the directory where the training samples are located.
8. Finally, to test all of these trained HMMs, we go into all of the test sets, score the test sample using the **each of the three "ensemblers"** (each made up of x HMMs.). Once we have the score from each ensembler, we can then go back to the logs where the original score was written down for the training samples.
9. Now, our sample has one score for each ensembler, denoted as $S_{1..\texttt{num\_ensembler}}(x_{\text{test}})$
10. We classify this sample as whichever has the minimum of $abs(S_{family}(x_{\text{test}}) - score_{\text{family}})$, that is, we classify as whichever family's "original score" score is closest to the score for the sample given by *that family's* "ensembled" HMMs. In other words, we're just doing the KNN algorithm, where $k = 1$

Part IV
Experimental Results

9.2 KNN Results

Below are the results of the KNN experiments:

K	Accuracy
1	94.99%
2	94.44%
3	93.06%
4	91.52%
5	90.02%
6	90.03%
7	89.89%

As can be seen above, the best achived accuracy using KNN is where $K = 1$.

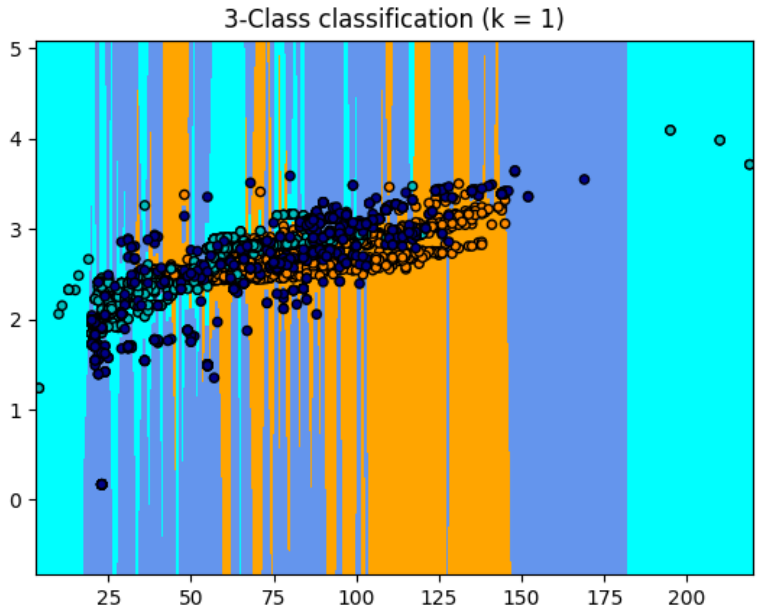

Figure 1: Overfitted family clusters when $K = 1$

When we plot the clusters on a graph, we can see that when $K = 1$, we can see that we're overfitting on the training set. Therefore, if this model were used in software production, we would probably choose a larger K such as $K = 3$ as to reduce overfitting, despite the fact that $K = 3$ yields lower accuracy. The figure below shows the clusters when $K = 3$, and we can see that outliers are not given as much of a "say" in our model, reducing overfitting.

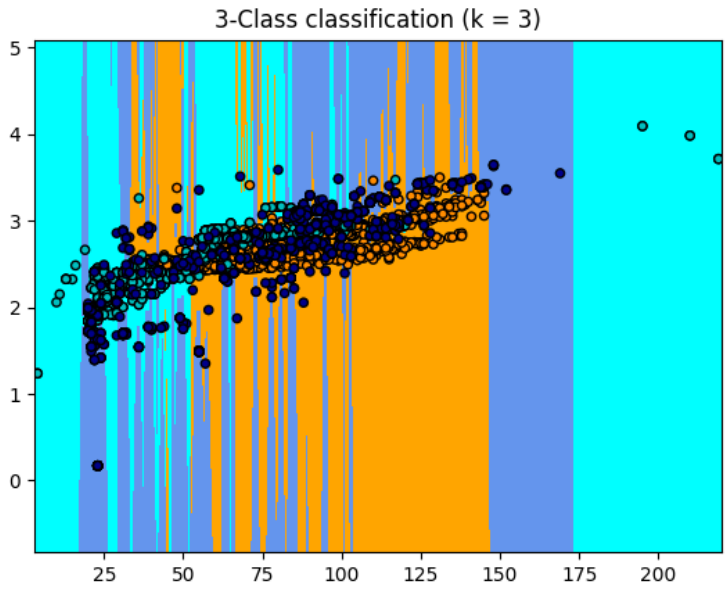

Figure 2: Family clusters when $K = 3$

9.3 (Primitive) Support Vector Machine Results

The best achived accuracy using purely SVMs and the two features (*entropy* and *distinct_opcodes* is given by the model described in the table below:

Key	**Value**
Kernel	*RadialBiasFunction*
Degree	3
C ("budget")	1
Normalized Features?	`true`
Accuracy	75.8%

Based on the results from both these SVM tests and the KNN tests of the pervious section, we conclude that clustering/ partitioning the malware families based on only these two features is not enough to produce a strong classifier. We can see from the plotted SVMs that there is a lot of overlap between classes, and any kernel would not be able to determine a corolation which does not exist to begin with, as can be seen in the zoomed in `rbf` kernel SVM below:

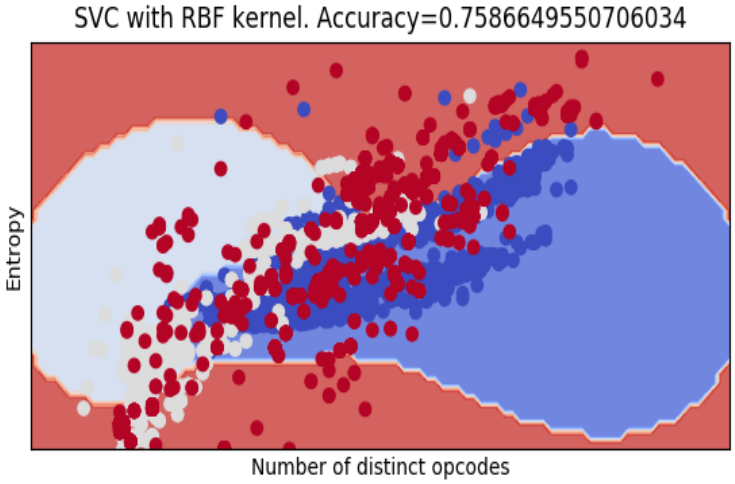

Figure 3: Zoomed in clusters using SVM with RBF kernel of degree 3.

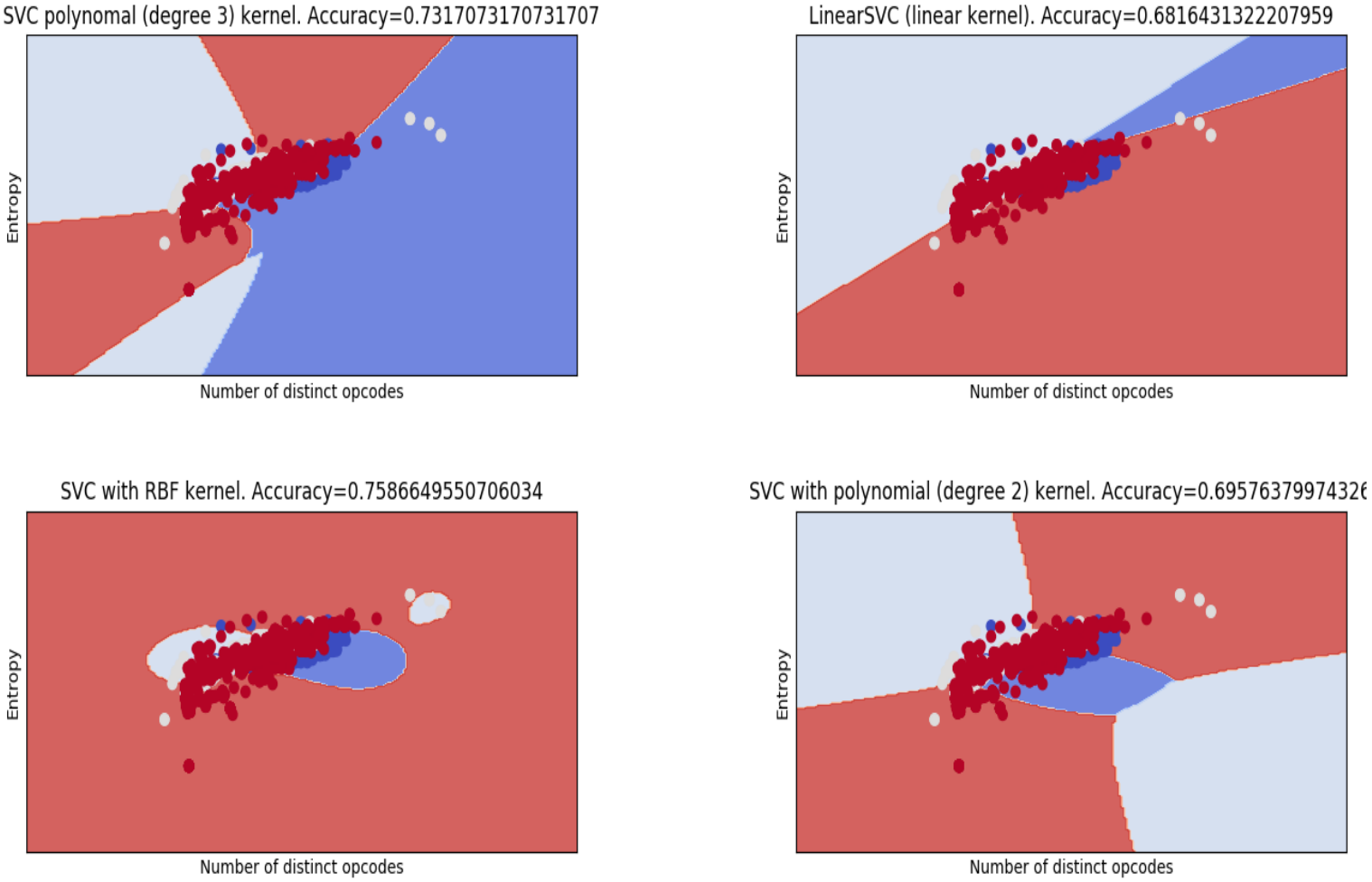

Figure 4: Family clusters using different SVM kernels

10 K-Means Clustering

Based on simply visually inspecting the dataset from our primitive SVM and KNN experiments, we didn't expect that K-Means would be able to differentiate classes based on their centroids. This is because there is *a lot* of overlap between the families when plotted on a 2D plane. Unlike KNN, K-means cannot handle the case of a class being made up of several disjoint clusters, which seems to be the case here. The best achived accuracy using K-Means was found under the following conditions:

Key	**Value**
Clusters	3
Num Features	2
Init	KMeans++
Accuracy	46.66%

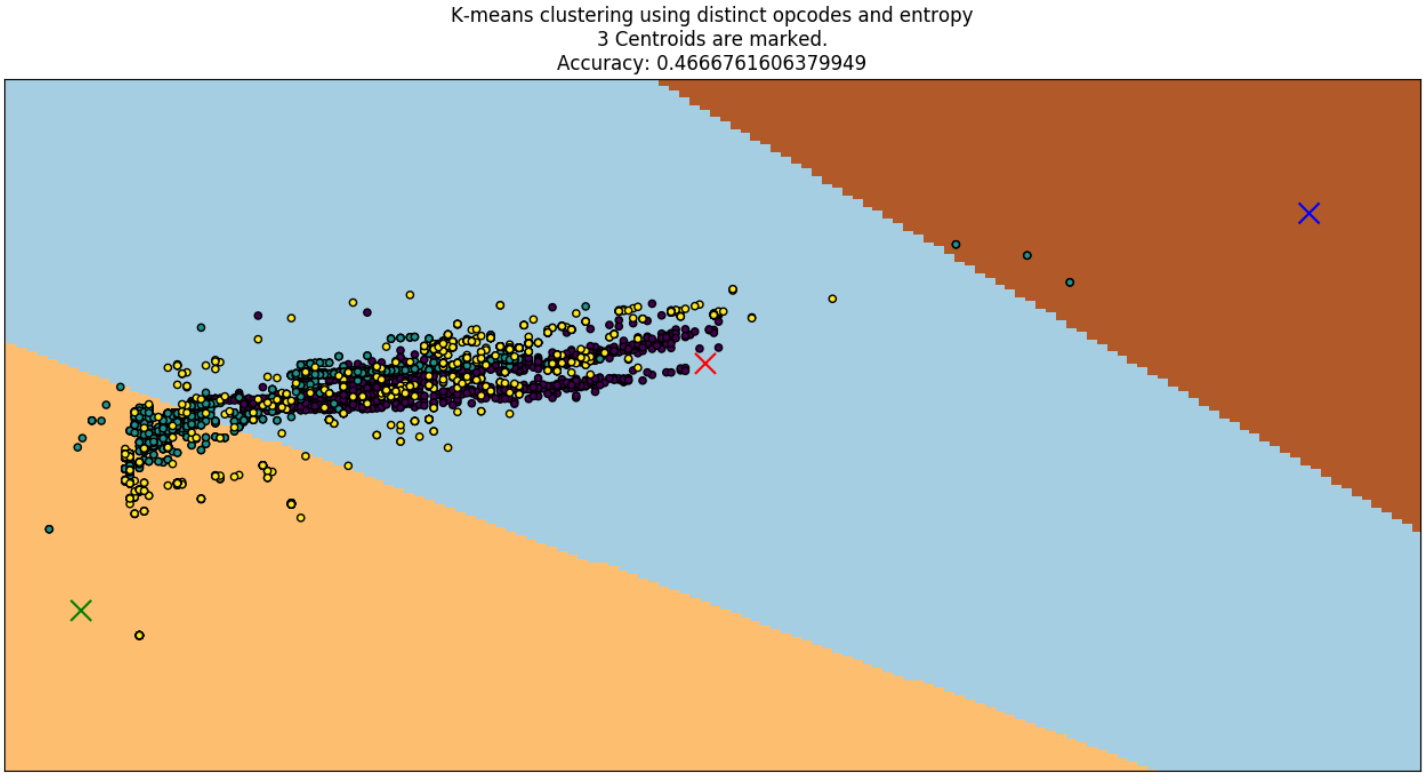

Figure 5: (enhanced) K-Means clustering using the parameters specified.

10.1 Boosting Results

Whether the score of the file was determined by the average or by the maximum value of the scores of 5 randomly restarted HMMs, the Boosting experiments did not yield a useful classifier. This can be clearly seen by the ROC curve (which was created where N = 2, the number of unique opcodes = 35, and the scores from the HMMs were averaged) where the true positive rate and false positive rate are all over the place. A more intuitive way of understanding why this method failed is to compare the scores of 5 randomly chosen files from the same family the HMMs were trained on and the scores from 5 files of a different family than the HMMs were trained on. for example:

```
5 random score from family X
Score 1: -17102.39110157962
Score 2: -2453.0432816283646
Score 3: -24443.862815912897
Score 4: -771.3044309566772
Score 5: -2648.110295014743
```

```
5 random scores from family Y
Score 1: -22302.30088705916
Score 2: 8179.418459861058
Score 3: 10555.795790797403
Score 4: 7972.734255134028
Score 5: 9193.531925297084
```

It is easy to see that a simple threshold is not sufficient to categorize a file as either malware from the same family the HMMs were trained on or malware from a different family. Different hyperparameters were tested, such as letting N = 3, 4, 5, or letting the number of unique opcodes be 20, 30, and 40. Each one yielded a similar result. It is possible that the HMMs are overfitting, and so performing some sort of cross validation would produce better results, but due to the overwhelming success of the Stacking experiments (next section), this was not pursued.

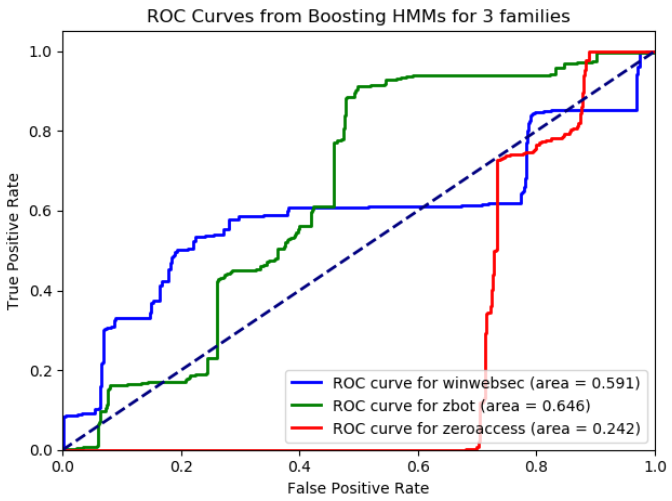

Figure 6: ROC Curve where $N = 2$.

10.2 Stacking Results

Key	Value
N	5
Unique Opcodes	35
Max Iterations	100
SVM Parameters	$c = 2, g = 2^{-16}$
WinWebSec Accuracy	98.9324%(2780/2810)
Zbot Accuracy	98.366%(3311/3366)
ZeroAccess Accuracy	98.4615%(3520/3575)

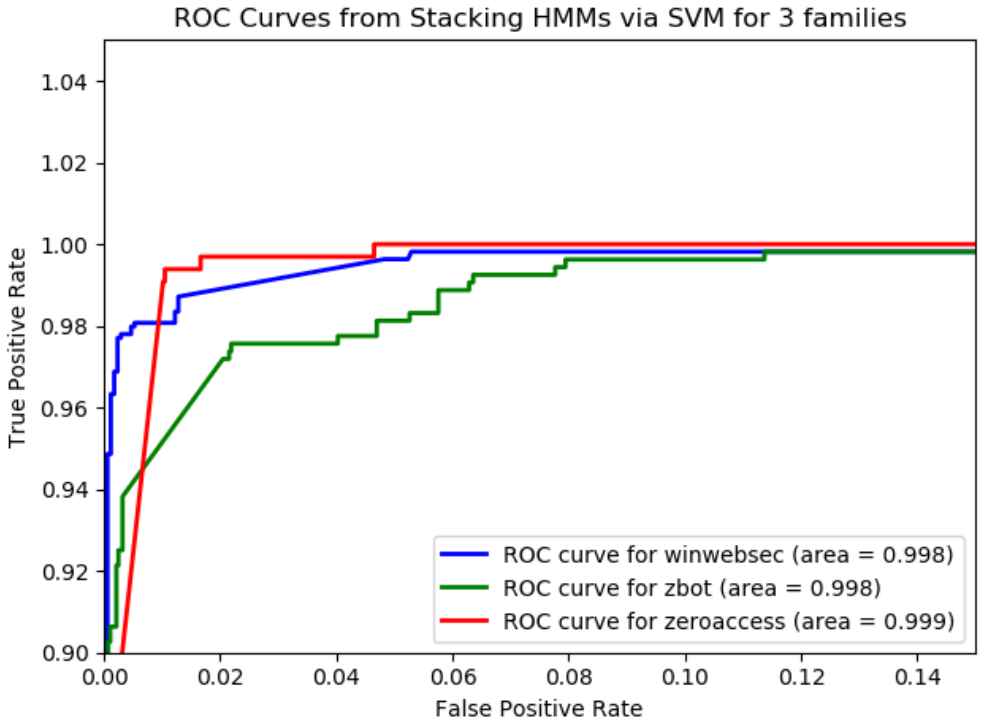

Figure 7: ROC Curve where $N = 2$.

Logistics Each family was split by 50%, 25%, and 25%. 50% of the family trains the HMMs. 25% trains the SVM, and the final 25% tests the SVM. It took well over an hour and a half to train all 15 HMMs used for these experiments, and though multithreading was implemented, our workstation was running out of memory when training with large observation sequences.

Hyperparameter Tuning

Number of Opcodes The number of unique opcodes for the best found classifier was 35. Originally tests were perfromed with 32 unique opcodes and got good results from that. Then 30 and 40 unique opcodes were tried, getting slightly worse results. Finally 35 opcodes was tested, which yielded slightly better results. The difference between 32 and 35 opcodes could just have been due to a lucky random restart, but it does seem like 35 opcodes is reliably better than 30 and 40 (the exact scores I got have unfortunately been lost). Finally, I tried 20 opcodes, getting significantly worse results.

Number of HMMs Stacking was tried with scores from 3, 5 and 6 different HMMs. 5 and 6 HMMs were comparable in accuracy and differences were likely caused by the random restarts. However, since it took 20% longer to train 6 HMMs, not many experiments were performed with them, so it is possible that there was some further way to optimize them beyond what I could do with 5 HMMs.

HMM Tuning For the HMMs themselves, N = 2, 3, 4, and 5 were tried. N = 4 and 5 were comparable in accuracy, though N= 5 seems to have been slightly better (once again, possibly just due to randomness). N = 2 and 3 were outclassed by 4 and 5 by at least 2-3%. N = 5 takes longer to train and uses more memory, but there is a reason I chose it over N = 4, and it has to due with the hyperparameters for the SVM described below.

SVM Tuning Finally, we get to the SVM. The library used was Libsvm version 3.24. There are a couple of SVMs that Libsvm supports, so we chose the default C-SVC. For the SVM kernel, the Radial Basis Function (RBF) kernel was used, which also happens to be the default kernel. Libsvm offers other kernels, such as a linear, polynomial, and sigmoid kernel. The linear kernel yielded horrible results (54% - 60%, little better than guessing) and every other available kernel would cause an *MAX_ITERATIONS_REACHED* error from LibSVM.

Library Specific Tuning Per the suggestion of the creators of Libsvm, there are two hyperparameters, c (budget) and gamma (RBF kernel parameter), that I should tweak. If appropate values are chosen for these hyperparmeters,the testing accuracy can be drastically increased, but if the wrong values are chosen, then accuracies tend to then range from 60-80%, which is obviously suboptimal. We found through experimentation (choosing values c = 2^2 through 2^-15) that letting c = 2 is pretty much optimal for every family, for any number of HMMs with any N, and for any gamma value for the SVM. It is possible that we could have optimized these hyperparameters more by searching through smaller itervals such as $c = 1.9, 1.8, 1.7, ...2.1, 2.2, ...$ but at that point, we would basically just be optimizing the SVM for the test set of the family, and while that would definitely be good to eek out the highest score in a competition, it probably wouldn't make the model better overall. Finally, we get to gamma. Choosing a good value for gamma is just as important as choosing a good value for c. We found experimentally that good values range between 2^{-10} and 2^{-20}, although choosing the right one between those limits could mean a difference in accuracy of 5% or more. The proper gamma value seemed to change with N, the number of files that are being used in the training set for the HMM/SVM, and random chance from the randomly restarted HMMs. When N = 4, it was difficult to find a single gamma value that would give a consistent result for all 3 families at once. Each family would be 1-5% less accurate than the last (although this did allow me to reach a 99.3007% accuracy on ZeroAccess). However, when N = 5, I found that I could find a gamma value where all families would be about equally accurate (in the 98.5% range).

10.3 Bagging Results

The best found bagging results are given in the below table:

Key	Value
Ensembler Aggregate Function	MAX
HMM's per ensembler	10
N (HMM parameter – # states)	2
Random initialization seed (HMM)	0
Test set size	10% of training set
Test Accuracy	0.9974326059050064

As is shown in the table above, the first bagging experiments went well on the surface.

On the Result:

1. Whenever a sample was scored **using an HMM which was NOT trained on the family which the sample belongs to, the score is returned as NaN.** This makes for very good accuracy; however, this behaver cannot be mathematically or otherwise justified.

   - For example, if a `winwebsec` sample was scored using an HMM which was trained on samples form the `zbot` family, then all the `zbot` HMMs will give `winwebsec` samples a score of NaN. *This was somewhat unexpected, because our HidddenMarkovModel implementation takes special caution (using logrythims) to avoid underflow. Further, we know our HMM is valid because of extensive testing using Mark Stamp's paper "A Revealing Introduction to Hidden Markov Models".* Our intuition tells us that this is okay, because the samples are still being given a valid score when the correct family's HMM's is used.
   - To attempt to remedy this, we tried changing training methodology:
     (a) **Increase number of HMMs** — Currently, durring training, when an HMM is trained on an observation sequence, that observation sequence is much, much larger than the actual samples which are being scored. To remedy this, we can split our "ensemblers" into more "bags" such that they're made up of more HMMs. In turn, each HMM will be trained on a closer amount of data it will be tested on. Despite our rationale.... results from splitting the ensemblers into 30 or 100 HMMs each rather than 10 yielded the *exact same results.*.
     (b) Originally, the plan was to add an SVM on top of the bagging described here. However, because samples of the "wrong" family are always given a score of `NaN`, this would be completely useless.

Part V
Discussion and Future Work

Discussion In review, the following methods were tested to classify malware samples into families:

1. KNN (94.99% accuracy, reduced feature set, overfitting)
2. Pure SVM (75.8% accuracy, reduced feature set)
3. K-Means (46.6% accuracy, reduced feature set)
4. Boosting HMMs (high false positive rate, unreliable model)
5. Bagging HMMs (NaN errors, unexplainable, high accuracy)
6. **Best Model:** Stacking with "boosted" HMMs (extensive hyperparameter tuning, highest accuracy, quck compuation)

Future Work Although the results from the experiments performed here are promising, there is always more work to do. Here is a list of suggested next steps for anyone interested in continuing this work:

1. Work with a larger dataset to better determine method and model accuracy
2. Chose gamma value for SVM based on automated testing instead of a fixed value
3. Investigate n-fold validation when training HMMs
4. Investigate n-fold validation when training/testing SVMs
5. Investigate using K-NN as a metaclassifier for HMM scores
6. Speed up the training and lessen the memory-usage of HMMs
7. Develop Stacking framework further to increase accuracy with samples from malware families not used in the training set
8. Extract more features for further clustering experiments in higher dimensions
9. Investigate the mesterious `NaN` error described in the "Bagging Results" section