CS185C:
Final Project
Malware Classification

Jordan Conragan, Brett Dispoto

May 18, 2020

Contents

Part I
Preprocessing

1 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 II

Experiments

2 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.

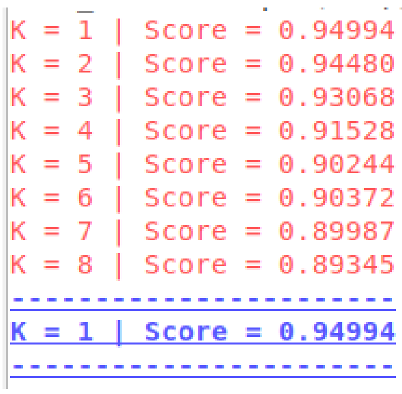
2.1 KNN Procedure

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.

2.2 KNN Results

Below are the results of the above procedure:



```
K = 1 | Score = 0.94994
K = 2 | Score = 0.94480
K = 3 | Score = 0.93068
K = 4 | Score = 0.91528
K = 5 | Score = 0.90244
K = 6 | Score = 0.90372
K = 7 | Score = 0.89987
K = 8 | Score = 0.89345
-----------------------
K = 1 | Score = 0.94994
-----------------------
```

Figure 1: Finding the best value for k.

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

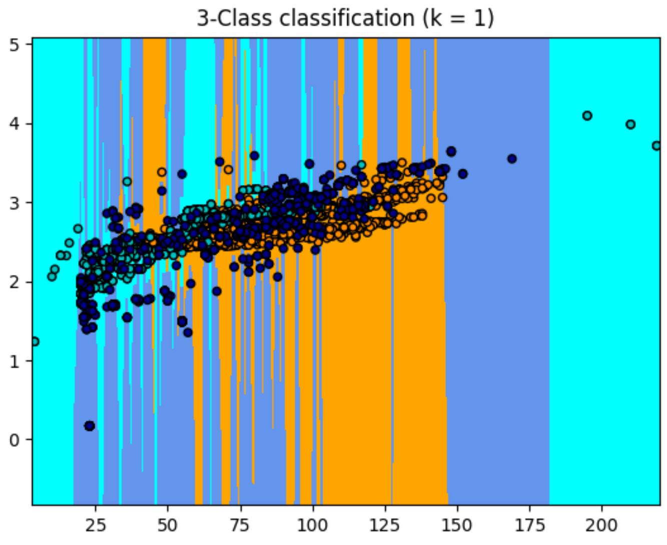


Figure 2: Family Clusters when $K = 1$

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. Below are the clusters when $K = 3$:

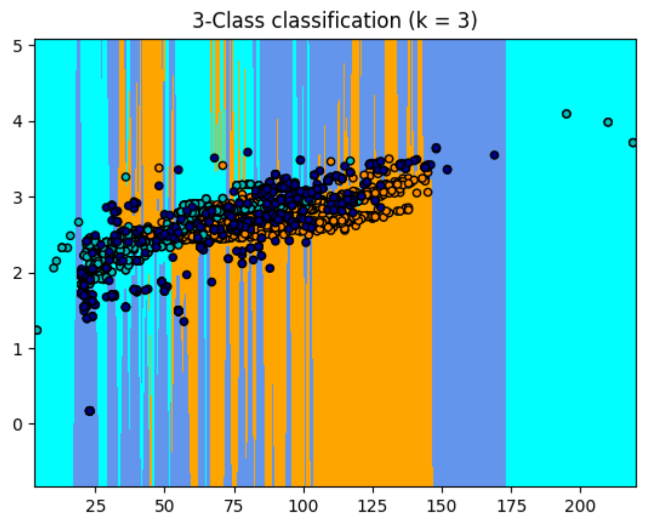


Figure 3: Initial results from Bagging.

3 Primitive SVM

3.1 SVM Procedure

3.2 SVM Results

4 HMM Bagging

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

4.1 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 $k = 1$

4.2 HMM Bagging Results

4.2.1 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. Screenshots of the information described here can be found in figure [4] in the appendix.

On the Result:

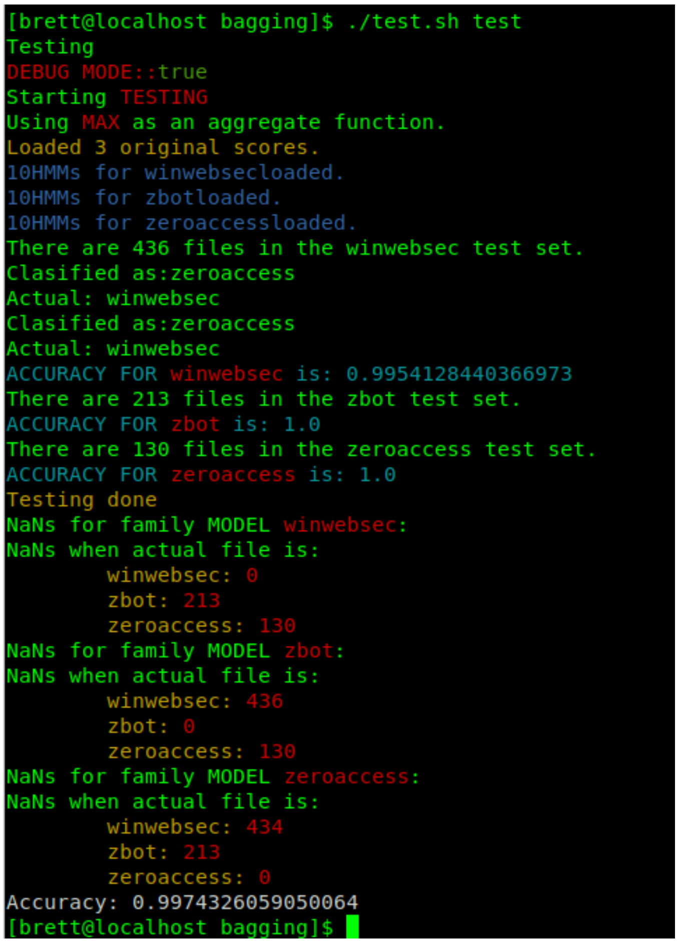
1. Whenever a sample was scored **using an HMM which was NOT trained on the family which the sample belongs to, the score is returned as NaN.** This makes for very good accuracy; however, this behaver cannot be mathematically or otherwise justified.

   - 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.

A Selected Screenshots



```
[brett@localhost bagging]$ ./test.sh test
Testing
DEBUG MODE::true
Starting TESTING
Using MAX as an aggregate function.
Loaded 3 original scores.
10HMMs for winwebsecloaded.
10HMMs for zbotloaded.
10HMMs for zeroaccessloaded.
There are 436 files in the winwebsec test set.
Clasified as:zeroaccess
Actual: winwebsec
Clasified as:zeroaccess
Actual: winwebsec
ACCURACY FOR winwebsec is: 0.9954128440366973
There are 213 files in the zbot test set.
ACCURACY FOR zbot is: 1.0
There are 130 files in the zeroaccess test set.
ACCURACY FOR zeroaccess is: 1.0
Testing done
NaNs for family MODEL winwebsec:
NaNs when actual file is:
        winwebsec: 0
        zbot: 213
        zeroaccess: 130
NaNs for family MODEL zbot:
NaNs when actual file is:
        winwebsec: 436
        zbot: 0
        zeroaccess: 130
NaNs for family MODEL zeroaccess:
NaNs when actual file is:
        winwebsec: 434
        zbot: 213
        zeroaccess: 0
Accuracy: 0.9974326059050064
[brett@localhost bagging]$
```

Figure 4: Initial results from Bagging.