CS 181 Practical 2 Writeup

Hikari Sorensen, Matt Leifer, Tomislav Zabcic-Matic

March 10, 2017

1 Overview

The classification of XML documents required multiple steps:

1. Create some feature representation of the XML documents in the training set
2. Do supervised learning on the feature representation

The labels for the training data, corresponding to the malware class of each XML document in the training set, were found using the starter code's `extract_feats` function.

2 Feature Representation

2.1 XML parsing

The XML files were first parsed, using the ElementTree XML API, into lists of "words", where the words were the result of splitting the XML file by the delimiters and " " (space). We also removed the following symbols: " < ", "\ > ", "$\backslash n$", and quotation marks. Each file then comprised a "sentence" of its parsed words, so that there were 3086 sentences in the training set.

For example, the first 50 words of the first XML file in the raw training data are:

```
['processes', 'process', 'applicationtype', 'Win32Application', 'executionstatus', 'OK',
 'filename', 'c:\\1025be1934a50c3355adb359507f2862.EX', 'filename_hash', 'hash_error',
 'filesize', '149270', 'index', '1', 'md5', '1025be1934a50c3355adb359507f2862',
 'parentindex', '0', 'pid', '1952', 'sha1', '0069ea50001a6c699d0222032d45b74b2e7e8be9',
 'startreason', 'AnalysisTarget', 'starttime', '00:01.704', 'terminationreason',
 'NormalTermination', 'terminationtime', '00:10.547', 'username', 'Administrator',
 'thread', 'tid', '1960', 'all_section', 'load_image', 'address', '$400000',
 'end_address', '$415000', 'filename', 'c:\\1025be1934a50c3355adb359507f2862.EX',
 'filename_hash', 'hash_error', 'size', '86016', 'successful' '1', 'load_dll']
```

Moreover, we could use these parsed words to find XML calls and variables that might distinguish one class from another.

In particular, we examined all the XML files we were given to see if there were any distinctive system calls (words) that were distinct to the various kinds of malware. This would be an important justification for using the parsed words as features. We've included a table below that shows the fraction of each class of file and the fraction of files that had a certain call in them. No one call was unique to a particular class, but there were some interesting call distributions across the classes. The table below includes only 4 of hundreds of possible calls and variables we found that occurred in the training set.

	`recv\_socket`	`CicLoaderWndClass`	`AutoRun`	`dump\_line`
Agent	$\frac{30}{114}$	$\frac{24}{114}$	$\frac{20}{114}$	$\frac{27}{114}$
Autorun	$\frac{3}{50}$	$\frac{0}{50}$	$\frac{7}{50}$	$\frac{4}{50}$
FraudLoad	$\frac{12}{37}$	$\frac{20}{37}$	$\frac{12}{37}$	$\frac{13}{37}$
FraudPack	$\frac{22}{32}$	$\frac{7}{32}$	$\frac{2}{32}$	$\frac{22}{32}$
Hupigon	$\frac{8}{41}$	$\frac{3}{41}$	$\frac{14}{41}$	$\frac{8}{41}$
Krap	$\frac{12}{39}$	$\frac{3}{39}$	$\frac{27}{39}$	$\frac{14}{39}$
Lipler	$\frac{33}{50}$	$\frac{50}{50}$	$\frac{0}{50}$	$\frac{33}{50}$
Magania	$\frac{1}{41}$	$\frac{0}{41}$	$\frac{26}{41}$	$\frac{1}{41}$
None	$\frac{53}{1609}$	$\frac{1021}{1609}$	$\frac{34}{1609}$	$\frac{53}{1609}$
Poison	$\frac{0}{21}$	$\frac{5}{21}$	$\frac{0}{21}$	$\frac{1}{21}$
Swizzor	$\frac{454}{542}$	$\frac{3}{542}$	$\frac{0}{542}$	$\frac{528}{542}$
Tdss	$\frac{5}{32}$	$\frac{14}{32}$	$\frac{8}{32}$	$\frac{5}{32}$
VB	$\frac{12}{376}$	$\frac{9}{376}$	$\frac{40}{376}$	$\frac{12}{376}$
Virut	$\frac{4}{59}$	$\frac{11}{59}$	$\frac{5}{59}$	$\frac{5}{59}$
Zbot	$\frac{8}{40}$	$\frac{0}{40}$	$\frac{8}{40}$	$\frac{8}{40}$

One word that was particularly interesting was the `CicLoaderWndClass` which occurred relatively rarely in most classes except for the Lipler malware files and the files that were not malware. `CicLoaderWndClass`

was in all of the Lipler files we had and 63% of the non-malware files. Because these calls by themselves did not seem to provide enough information to classify the files - this was essentially the precursor to a Bag of Words model - we decided to use a Word2Vec model that would help capture some of the semantic information contained in the ordering of these calls. This makes intuitive sense because the context in which a system call might appear in one of these files might give some indication as to what that file is doing.

2.2 Word2Vec

The 3086 sentences were then used to train a Word2Vec model that learned the words in the training sentences as its vocabulary. The Word2Vec parameters used were as follows:

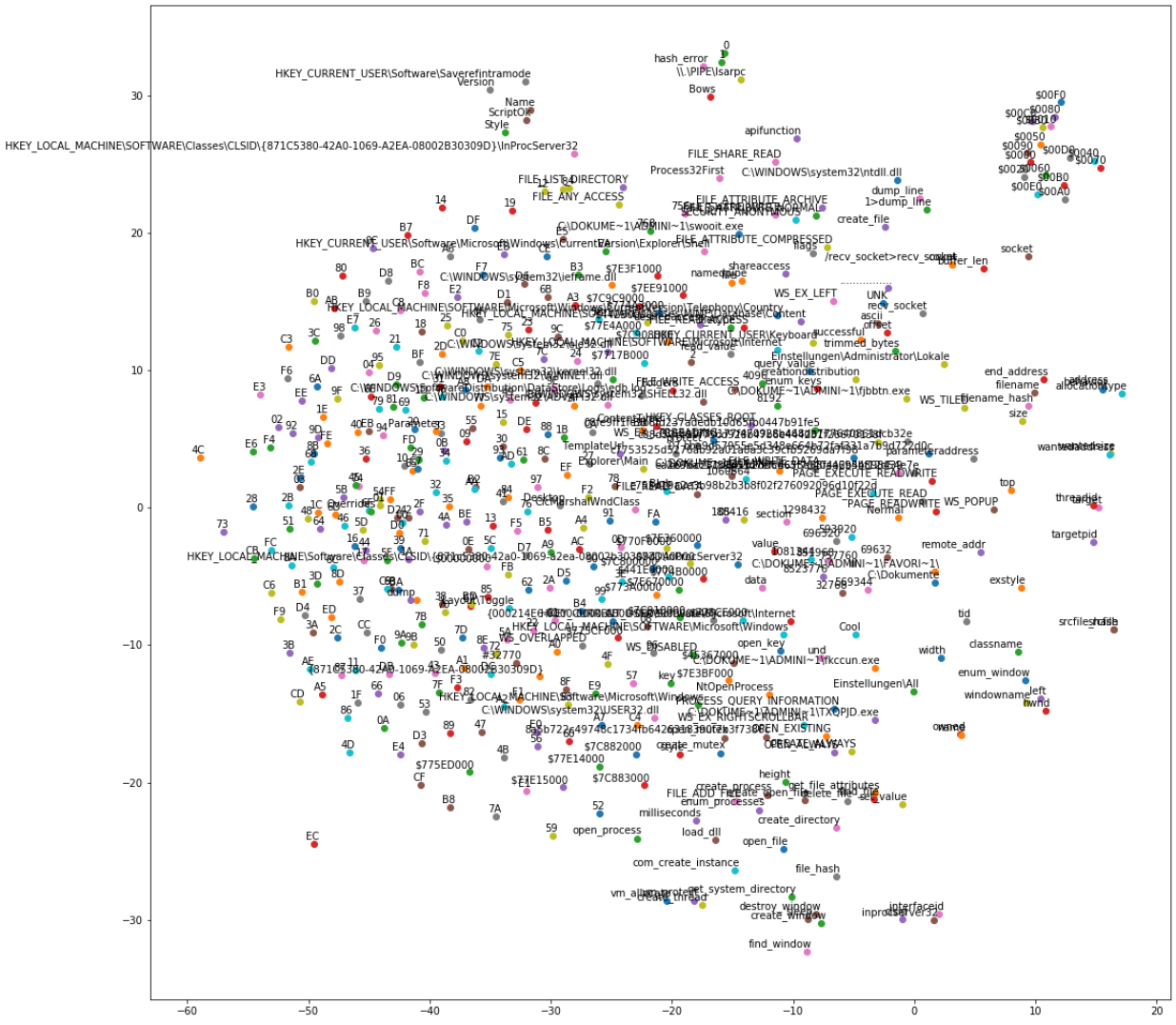
```
num_features = 500
min_word_count = 1
context = 15
downsampling = 1e-3
```

where `num_features` sets the word vector dimensionailty; `min_word_count` sets the minimum number of occurrences of words in the training data needed for Word2Vec to include it in its model (this is usually set above 1 to exclude very rare words); `context` sets how far the model should "look" to the left and right of each word to make a prediction of what the word itself should be; and `downsampling` sets the downsampling rate for frequent words, so that the training is quicker.

We actually tried several different values for these parameters, but the parameters shown above gave optimal models. In particular, `min_word_count = 1` worked the best, as would be suggested by intuition - we want to consider all words, even the rare ones that may only occur in the rare classes.

Word2Vec outputs an array of size 1121376×500 of feature vectors for each word in the vocabulary, where 1121376 is the number of distinct words in the vocabulary (comprising all distinct words in all training files, since `min_word_count` was set to 1), and 500 is the number of features for each word, set as the parameter `num_features`.

A 2-D visualization of the Word2Vec vectors of the 500 most common words in the XML documents is shown below.



2.3 KMeans

Now that we have a vector representation of each word in the XML vocabulary, we need to represent each file itself (represented a a sentence) in a way that can be used for classification. What is important about Word2Vec over a simple Bag-of-Words approach is that Word2Vec preserves some information about the semantic similarity between words. In particular, it clusters words that are semantically similar, as can be seen in the plot above.

Thus, we can use the clustered form of the Word2Vec model to find cluster centers that represent groups of semantically similar words. The cluster centers can then be labelled with distinct integers, and then for every word in each XML sentence, we assign it the ID of the cluster to which it belongs.

We found the cluster centers by running KMeans on the Word2Vec word vectors, with $K = \texttt{len(vocab)}/500 \approx 2240$. That is, each cluster had on average 500 words in it. The model would likely have been much better with K significantly larger - ideally, each cluster might have < 10 words on average. However, on such a large vocabulary, KMeans is slow, and so we needed to use much smaller K in order to run the model in a reasonable time.

This in some ways forms a "Bag-of-Cluster-Centers" representation of the XML files. This is useful because it compresses the XML sentences to vectors having values only up to the number K of clusters we choose, in a way that still preserves information about the words' "meanings", but without assigning one of 1121376 labels to each different word. Moreover, there are many words that are particular to individual files, but that are not necessarily predictive on their own. Thus, the clusters reduce the noise in the data by grouping together different words that all serve a similar function from file to file.

Note, however, that while this can be very useful for compression and for smoothing over noise, it may also smooth over rare words that by themselves might identify a file as being in a separate class. Especially with such skewed classes as was found in the training set, this can be a big issue. However, as a first run, we assumed that this would not be too problematic.

The sentence embeddings that were found by assigning XML words to cluster IDs were then embedded in a $3086 \times m$ matrix of 0s, where $m = \max\{\texttt{len(sentence)}\}$. This pads the shorter-length sentences with 0s so that the vector representations of the XML files are all the same size (so that they can be used for classification).

3 Learning

In our classification, we used a Random Forest classifier on the sentence embeddings, as well as XGBoost. Our ideal was to use a Recurrent Neural Network like LSTM, or a Convolutional Neural Network from the Keras ML library, and we intended Random Forest and XGBoost to serve as baselines. However, after much fiddling with Keras, we were not able to configure a working neural network.

Fortunately, however, the models we did manage to train did reasonably well in prediction. The table below shows the performance of various models with various parameters.

"No." refers to the order in which the predictions were submitted to Kaggle; "Model" refers to the type of classifier used; $\texttt{voc}/K$ refers to {the size of vocabulary} / {number of clusters} = average number of words per cluster used in KMeans (larger $\texttt{voc}/K \implies$ fewer clusters); `min_ct` refers to minimum number of occurrences of a word required in Word2Vec for the word to be included in the model; `n_feats` refers to the dimension of the word vectors; `context` refers to the size of the window of neighbors in which a word is considered in Word2Vec; `n_est` refers to the number of trees used in the random forest classifier; `max_d` refers to the maximum depth of the tree in the XGBoost model; η refers to the learning rate of XGBoost; `rounds` refers to the number of rounds for boosting for XGBoost; "Public" and "Private" refer to the public and private Kaggle scores, respectively.

No.	Model	min_ct	n_feats	context	voc/K	n_est	max_d	η	rounds	Public	Private
1	RF	15	800	40	1000	200	-	-	-	0.82263	0.81195
2	RF	10	800	30	1000	200	-	-	-	0.82316	0.81579
3	RF	10	2000	20	500	200	-	-	-	0.83000	0.81305
4	RF	10	5000	15	150	200	-	-	-	0.82947	0.81689
5	XGB	10	5000	15	150	-	10	0.5	30	0.81263	0.806487
6	RF	1	500	10	500	1000	-	-	-	0.82842	0.82292
7	XGB	1	500	10	500	-	15	0.3	30	0.80842	0.80592
8	XGB	10	500	10	500	-	40	0.2	25	0.81474	0.79660

Note here that No. 3 had the greatest public score on Kaggle, but No. 6 had the greatest private score. Moreover, if we had used No. 6 for the final Kaggle competition submission, we would have had a 2nd place score!

#	Δrank	Team Name	Score	Entries	Last Submission UTC (Best – Last Submission)
1	—	millions	0.83388	21	Thu, 09 Mar 2017 04:43:04 (-24.9h)
-		**Snorlax**	**0.82292**	-	**Fri, 10 Mar 2017 18:38:33** Post-Deadline
Post-Deadline Entry If you would have submitted this entry during the competition, you would have been around here on the leaderboard.					
2	↑41	DontUseARandomForest	0.82127	7	Wed, 08 Mar 2017 23:15:07 (-40.9h)
3	↑42	B3-D1	0.82072	17	Thu, 09 Mar 2017 04:54:09 (-38h)
4	↑12	quadsquad	0.81963	15	Thu, 09 Mar 2017 03:51:11 (-0.5h)
5	↓2	!?	0.81798	22	Thu, 09 Mar 2017 04:40:15
6	↑16	TwoHotEncoders	0.81634	6	Mon, 06 Mar 2017 16:00:56 (-0.9h)

However, we overfit to the public test data, and Kaggle entered into the competition a model that had a high score on the public test data, but that didn't generalize as well to the rest of the (private) test data.

4 Going Forward

As mentioned earlier, we would ideally have run an LSTM on the word embeddings. This is because tree-based methods like Random Forest and XGBoost lose the ordering information of the words in a sentence. That is, they only consider the existence of words in the sentence, disregarding the order in which they occur. For XML documents, the ordering of commands and variable assignments matters with regard to the entire file structure.

Broadly speaking, a Recurrent Neural Network (RNN) preserves order information by learning on training data sequentially, rather than considering a whole matrix of features simultaneously as a representation of the training data. A Long Short Term Memory network (LSTM) is a type of RNN that is particularly good at "remembering" information it encountered early in the training data, even when considering other training data that occurs much later. In short, this gives LSTMs the ability to, in a sense, consider the important structure of entire XML sentence as well as the individual calls that are made within it.

A Convolutional Neural Network (CNN) might also have performed quite well, since it also parses each data point "in order", with some assumption of "smoothness" or similarity of structure within any small interval. While LSTM is generally considered somewhat superior to CNNs in natural language processing (since text is naturally sequential), CNNs have also shown reasonably good performance in NLP problems. Moreover, the latter are also often easier to train.