Malware Classification: Distributed Data Mining with Spark

Luba Gloukhov **Cody Wild** **David Reilly**

Project Website: http://msan-vs-malware.com
Project Github: https://github.com/reillydj/KaggleMicrosoftMalware

Abstract

This paper describes the objectives, methods, challenges, and results of our participation in the Microsoft Malware Classification Kaggle competition. The primary motivation for joining this competition was to gain hands-on experience with machine learning in a distributed setting. To this end, we utilized Amazon's Simple Storage Service (S3) and Elastic Cloud Compute (EC2) in conjunction with Apache Spark and Oxdata H2O to store, access, parse, and model the roughly 400GB of text data provided by Microsoft. Three models have been produced thus far, of which the top-performing was a random forest, achieving a log-loss of 0.079: a 96% improvement over the equal probability benchmark.

Introduction

The amount of data currently being produced every day has reached a truly incredible magnitude, and, as businesses are struggling to leverage this data to their advantage, the ability to gleam impactful insights from this endless tide of information is quickly becoming a much sought-after skill. As such, our team thought the acquisition of firsthand experience with the scalable tools being developed to better access, parse, and manipulate massive amounts of data would be an extremely valuable and worthwhile endeavor. Furthermore, it is now a rarity for anyone in the data science community to conveniently find him or herself with a preprocessed, polished collection of data, on which popular machine learning algorithms can be immediately applied. With this in mind, we felt it would be unreasonable to select anything but an unconventional, convoluted set of data as our proxy for exploring the world of distributed machine learning. For these reasons, and, given our motivations, objectives, and allotted time, we chose the Microsoft Malware Classification Challenge dataset because it perfectly aligned with our size, complexity, and availability criterion.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The Data

The task of grouping variants of malware files into their respective families is a daunting task. One of the primary inhibitants of the successful completion of this task, i.e. the successful classification of malware files into their respective families, is the sheer mass of the amount of data that must be sifted through in order to evaluate and identify potentially maliscious files. For example, Microsoft finds itself having to analyze and classify tens of millions of data points every day. The explosive magnitude of this problem is, in no small part, due to the polymorphisms introduced in order to manipulate, modify, and obfuscate the malware files for the purposes of avoiding detection. Thus, the ability to group together malware files by their polymorphic maliscious components would substantially aid Microsoft in their unceasing endeavor to seize these files before they are able to do their damage. For this reason, Microsoft provided the data science community with a substantial malware dataset in an effort to open-source the task of discovering effective techniques of identifying and classifying malware files.

The dataset contains 9 malware families: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, and Gatak. For each observation we were provided with a bytes file and an asm file. The bytes file is the raw hexadecimal representation of the malware's binary content. A snapshot of one of these bytes files is shown below.

```
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
```

The lengths of these bytes files were found to differ across the classes, which gave us a very early indication that the length of the file may serve as a useful feature during the modeling stage of our project.

The asm file, generated by the IDA disassembler, is a log containing various metadata such as rudimentary function calls, memory allocation, and variable manipulation. A snapshot of one of these asm files is shown below.

```
; ---------------------------------------------------------------------------
                align 10h
                mov     dword ptr [ecx], offset off_42BB08
                jmp     sub_402C51
; ---------------------------------------------------------------------------
                align 10h
                push    esi
                mov     esi, ecx
                mov     dword ptr [esi], offset off_42BB08
                call    sub_402C51
                test    byte ptr [esp+8], 1
                jz      short loc_40104E
                push    esi
                call    ??3@YAXPAX@Z    ; operator delete(void *)
                add     esp, 4

loc_40104E:                             ; CODE XREF: .text:00401043^Xj
                mov     eax, esi
                pop     esi
                retn    4
; ---------------------------------------------------------------------------
                align 10h
                mov     eax, [esp+8]
                mov     cl, [eax]
                mov     edx, [esp+4]
                mov     [edx], cl
                retn
; ---------------------------------------------------------------------------
                align 10h
                mov     eax, [esp+4]
                lea     edx, [eax+1]

loc_401077:                             ; CODE XREF: .text:0040107C^Yj
```

Kaggle split this dataset into a labeled training set and an unlabeled test set. There are imately 11,000 observations in the training set and 11,000 observations in the test set. Each of these observations is roughly 50MB in size, resulting in around 200GB of training data and 200GB of test data.

The Tools

Our team sought a collection of tools that would aid us in the following tasks: storage, computation, and modeling.

Storage: S3

When the time came to choose the set of tools our team would use to manage this large unconventional dataset, it was immediately obvious that the first tool we would need was one capable of storing a large amount of unconventional data. Given the structure of the malware files, Amazon's Simple Storage Service (S3) seemed to be a cheap repository for storing our data. Furthermore, its ease of use in conjunction with Amazon's Elastic Cloud Compute (EC2) and widespread use in industry made it an obvious choice.

S3 allowed us to create and access secure, durable, and scalable buckets, in which we stored the training data, testing data, and any intermediary data generated during the parsing and preprocessing stages of our project. Access permissions could be easily allocated to all of our team's members, smoothing the process by which each individual member retrieved the data.

Information regarding S3 can be found at the following link: http://aws.amazon.com/s3/

Computation: EC2, Spark

Having chosen S3 as our data repository, the choice of cluster computation tools was almost made for us. Not only can EC2 seamlessly access S3, but it also provided us with a wide range of choices for computational resources. Whether a task called for a 20 node cluster or a single memory-optimized machine, EC2 was able to provide the necessary computational power.

Information regarding EC2 can be found at the following link: http://aws.amazon.com/ec2/

The choice of EC2 was also influenced by its seamless integration with Spark. Spark is a fast, in memory engine for large-scale data processing that runs on top of a Hadoop Distributed File System and can easily access data that has been stored in S3.

Information regarding Spark can be found at the following link: https://spark.apache.org/

Modeling: H2O, Sklearn

There are several tools available for modeling in a distributed framework. Despite the fact that our early datasets of engineered features didn't require distributed computing, we invested time into these methods with aspirations of working with larger datasets down the road. H2O which allows for in-memory analytics on EC2 clusters with 'pre-baked' full-featured algorithms. Specifically, random forest and deep learning, two algorithms supported by H2O, were of high interest for the team for this tackling problem. Unfortunately, our ambition to use H2O on larger datasets was sidetracked by import difficulties explained later.

Information regarding H2O can be found at the following link: http://docs.0xdata.com/deployment/hadoop.html

Parsing and Preprocessing

Most of our thinking and work around parsing and feature engineering in this project was guided by two primary

constraining realities that had to be balanced. The first is the baseline reality of our input dataset; in particular, the fact that its size came less from an overabundance of individual observations (each training and test set only consisted of ≈11,000) and more from the fact that each observation contained roughly 50MB of information to be distilled into features.

The second reality we contended with comes from the fact that our test set can only be scored by Kaggle, and Kaggle only calculates scores based on a log-loss, calculated as a function of class probabilities. This posed a significant problem, as MLLib - Spark's indigenous distributed ML library - hasn't yet been configured to output class probabilties. As a result, in order to run scorable models, we needed to either find an alternate distributed framework - which we explored with H20 - or else reduce our data to a feature size where models could be trained locally using sklearn.

Spark Infrastructure

To achieve this goal of condensing our data into usable form, Spark was heavily used to conduct MapReduce operations that extracted frequencies, word samples, and implement vectorized aggregations. Spark is a fairly new software system, built open source by Databricks, that has built a highly usable set of libraries, particularly in Python, for accessing MapReduce methodologies. It does this by building a flexible and programmatic layer on top of Hadoop and S3/EC2.

Spark provides a script that can be used to launch, start, stop, and log into a Spark cluster. When a cluster is created using this script - after having exported a set of AWS access keys - Spark is automatically installed, and a default Spark Context is created that links the master node to all of the slave nodes, the number of which can be selected at the user's discretion. Once a cluster is created, it - master and slave alike - can be accessed via SSH/SFTP like any other remote compute system. To run a program remotely using PySpark, Spark's Python interface, one logs into the master node of the cluster, using the direct DNS address, and runs a Python file using the spark-submit utility.

We discovered it to be good practice to, before submitting a script, modify a few of Spark's default config options, which can be done by opening spark/conf/spark-defaults.conf in a vim editor. We set spark.eventLog.enabled to 'true', which allows the Spark UI to give detailed debugging information after a Spark script has terminated, and spark.storage.memoryFraction was set to a value in the range 0.05-0.1, as it increases the amount of RAM available for calculation; a necessity when each observation was as large as ours were.

Once you have an environment ready to accept a Spark script to run in a distributed setting, the next hurdle is crafting your solution in a way that fit the map-reduce framework that is fundamental to Spark. An example of such code is shown below.

```
sc.wholeTextFiles(inFile, 100)\
            .flatMap(textSectionParse)\
            .reduceByKey(lambda a, b: a+b)\
            .filter(lambda x: x[1] > 3 )\
            .map(filenameMap)\
            .reduceByKey(lambda a, b: a + b)\
            .saveAsTextFile(outFile)
```

The 'sc' object here is a Spark Context, the Python object responsible for transmitting information across the cluster. The first function, wholeTextFiles, was our input workhorse, because it was structured to read in a full directory of files, using each whole file as a single observation. Secondly, we call a flatMap. This is a method that takes in a tuple of (fileID, fileString) and returns a sequence: in this case, a sequence of words that appear in the _text section of the .asm file, along with the number 1. As a result, each file, previously one observation, would be turned into several hundred observations in the Spark RDD, each of the form ((fileID, token), 1).

This structure was implemented at this step because it allowed the next step, a straightforward reduce by key, to calculate frequencies of each fileid, word occurance by simply summing the second element of the tuple in a reduceByKey framework, since the (fileID, word) tuple serves as a key. Next, in order to reduce cardinality, tokens were filtered to only return those appearing more than three times in the file. Another map - not a flatMap this time - was run, this time. This conducts a simple transposition, changing the structure of each tuple to (fileID, (token, count)). This allows a second reduceByKey to be conducted, which this time simply 'adds' the tuples together into a list of tuples, (token, count) for each fileID. This list is then written to a text file.

Feature Engineering

Our team was tasked with extracting meaningful features from these files in order to conduct 9-way classification. The following features were engineered:

- Hyperfeatures: Very high level features about a file in question. For example: number of lines in the file, number of tokens in the file, number of subroutines in the .asm file, and length of each section (_text, _idata, _rdata, data) within the .asm file.
- Bytes unigrams: the first approach was to take the straightforward frequencies of each of the possible hex duos (i.e. F8) within each of the files. Since these were highly granular tokens, this parse has the benefit of low cardinality, and served as a modeling benchmark due to its small size (≈50MB for the entire training set) and resultant easy of local manipulation. This feature fueled both the Random Forest and Deep Learning H20 models
- Bytes bigrams: Given the utility of .bytes bigrams, we proceeded to try to incorporate .bytes bigrams (i.e. F8 C4)

as frequency features. This resulted in an output with unsustainably high cardinality (initially 13 GB of training set data, even after parsing). To remedy this, all bigrams with less than 100 occurances were removed, leaving a somewhat more managable test set size of 3 GB.

- .asm significance vectors: These significance vectors, which are not identical to but analogous to TFIDF vectors, were calculated by aggregating tokens in a highly simplified parse of the .text section (where most condensed and linguistically intact assembly code occured) by class, and calculating the significance of each token to each class, normalized by both the number of occurances of that token overall and the number of tokens in that class overall.
- .asm semantic vectors: One fascinating aspect of this problem is the fact that the assembly code in the .asm files exists in a space between raw numbers and NLP. Despite being highly regularized and pattern-enforced, code is still a language in that it allows some degree of flexibility in expression, even within those enforced syntactic bounds. With this as a perspective, Google's word2vec python implementation in the gensim package was used to construct a semantic vector model of a 10% sample of tokens. The output of this means that words that are similar in meaning are closely located within this 100-dimensional vector space. This allows us to numerically aggregate distinctions in semantics that might distinguish between the operations of different viruses.

  Specifically, two Word2Vec models were trained separately on sentences extracted from two different sections of the assembly code: .text and .idata.These models were created separately to reflect the fact that there exist distinct linguistic signatures in each of these sections, and if a model were trained on both together, it would be made incoherent, as if a model were trained on German and French and tried to force them into one semantic framework.
  .text section:

```
mov     ecx, offset unk_57B044
shl     edx, 3
cmp     esi, 35h
```

  .idata section:

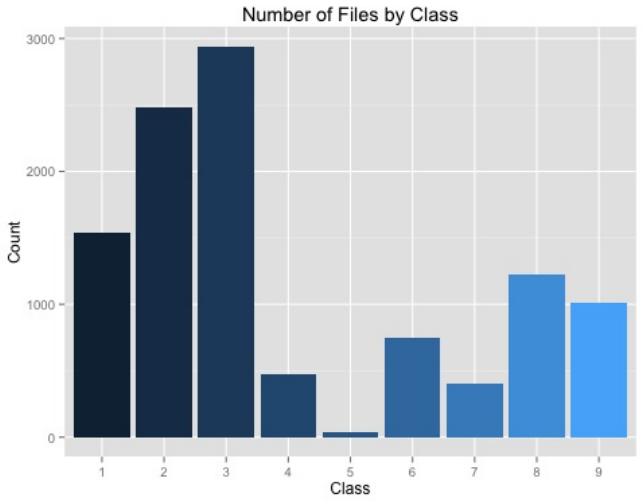
```
BOOL __stdcall SetConsoleCursorInfo(HANDLE
hConsoleOutput,const CONSOLE_CURSOR_INFO
*lpConsoleCursorInfo)
```

  In order to train a model, regular expressions parsing had to be used to programmatically identify the start and end of semantic blocks that could be understood as analogous to sentences, based on the textual structures identified by examining a few of the files.

  Once models were trained - at which point a token could be mapped to a 100-long vector - a Spark parser went through the full data and transformed documents to a list of sentence-vectors, where each sentence-vector was the mean of Word2Vec token vectors within it. In our initial model, we then further aggregated this list of sentence-vectors by taking its mean and variance. If given more time, we would like to return to this point in the process and, instead of aggregating these sentence vectors into a single mean, perhaps aggregate them into ten sequential means to take account of sequential logic within the file.

Modeling

Two predictive models were trained using H2O running on EC2. Once H2O is launched on EC2 and loaded in R, the capabilities of H2O are available in a familiar R syntax. Many of the common R functions such as head(), class(), colnames() and summary() have been built out to handle H2O data objects. The close integration with R makes it easy to visually explore aggregations of the data such as the number of files by class in the training corpus shown below.



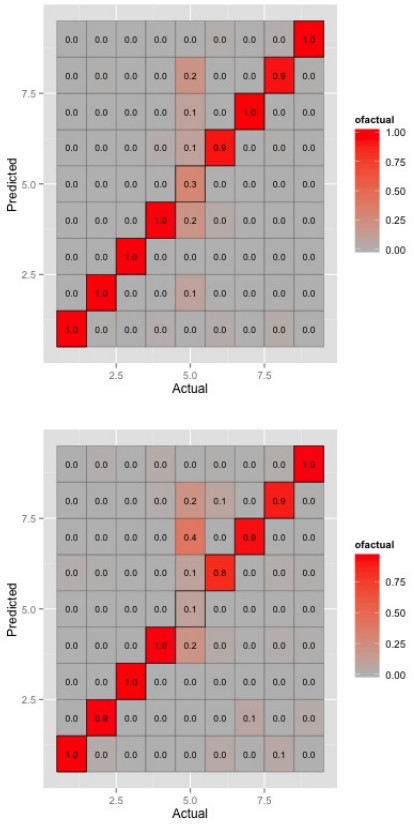
The class distribution is unbalanced, with few training files labeled as class 5 - Simda. In an effort to account for this, a search for optimum resampling proportions was done. Specifically, 9 proportions were pseudo-randomly generated 1500 times. For each set of proportions, stratified five fold cross-validation was performed. The four training folds were resampled according to the set of proportions while the fifth test fold was left with its original distribution of classes. The log-loss achieved by fitting a random forest on the resampled folds was calculated and stored in a dictionary along with the proportions that achieved it. This allowed for the total set of dictionaries to be sorted according to log-loss.

In order to reduce the amount of time required to perform this resampling method, a large EC2 instance with 8 cores was launched. The joblib package was then used to distribute jobs to each of these 8 cores in parallel.

However, the set of proportions that resulted in the lowest log-loss was the original class distributions.

In training the models, the training dataset was split into 80% train and 20% test subsets. Each model was trained on the 80% subset and tested on the other 20%.

Confusion matrices on the 20% testing subset for Random Forests (left) and Deep Learning (right) are shown below (numbers indicate portion of actual correctly predicted). We can see that both models do well in correctly predicting the class for all classes except 5 - Simda.



In the end, it wasn't the look of a confusion matrix but a log-loss calculation that would determine our ranking. The following formula is used to compute log-loss given a vector of labels, allowing us to determine the log-loss of our 20% labeled predictions.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{M} y_{ij}\log(p_{ij})$$

The Random Forest and Deep Learning models trained on H2O lead to a 0.0988557 and 0.1809695 log-loss, respectively.

In summary, H2O's integration with R made exploratory analysis intuitive and relatively pain free. However, subsequent attempts to analyze larger datasets split up across multiple files were unsuccessful. H2O is extremely sensitive to column headers (special characters, duplicate names). H2O has no support for merging of data aside from the H2O built-out cbind() and rbind(). As such, to combine multiple files, it is required that all of the files have all of the same columns in the same order, a surprisingly challenging undertaking when dealing with 21 files of ≈10MB containing thousands of columns. There is also no support for sorting of data. Adding JSON or similar dictionary-key/value support would allow for efficient storage and analysis of files that may be too sparse for columnar representation. H2O is great at fast, scalable machine learning in an environment many data scientists find comfort in. However, being finicky about data inputs and lacking vital data munging capabilities, H2O is far from a standalone tool.

With H20 incapable, due to the aforementioned lack of merging ability, to input large files for analysis, we ended up compromising by finding ways to aggregate our features to sizes (in this case, $\approx 200MB$ each for train and test) that could fit into a local sklearn model. Our final model, a baseline Random Forest trained in sklearn which achieved .079 test set log loss, utilized the following features:

- Bytes unigram frequencies $\rightarrow$ 256 features representing frequencies of each .bytes unigram.
- Hyperfeatures $\rightarrow$ 7 features representing .bytes length, .asm length, section lengths for 4 sections, and numSubroutines
- .asm significance vectors $\rightarrow$ 18 features representing the mean sentence significance vector and the variance across the sentence significance vectors
- _text semantic vectors $\rightarrow$ 200 features representing the mean sentence _text semantic vector and the variance across the sentence _text semantic vectors
- _idata semantic vectors $\rightarrow$ 200 features representing the mean sentence _idata semantic vector and the variance across the sentence _idata semantic vectors

Challenges

The many challenges we faced over the course of this project proved to be some of the most important exercises of the entire endeavor.

First and foremost, 'Big Data' feature-wise is an entirely different situation than 'Big Data' observation-wise. Many distributed systems have effective ways of processing independent rows in parallel, but since many machine learning algorithms require all columns to be present for any calculation to take place, parallelization of independent column process is a harder problem, and one that, at least in our survey of the available toolkits, not one that has been solved well yet.

Second, Amazon Web Services can be a tricky system to navigate effectively, both because of problems inherent to the product - i.e. proper setup of fresh linux machines - but also because of surprisingly clunky syntax and idiosyncratic required structure.

Third, when dealing with Spark, the price you pay for using bleeding edge technology, especially of the open source

variety, is that you're on the front lines, both when it comes to catching outright bugs and finding odd and unpredicted quirks in the way the software was designed. Since we - like many others - rely heavily on the vibrant ecosystem and historical record of questions that have built up on Stack Overflow and the like when gaining knowledge of more established systems like base Python, learning how to solve problems in this lower-resource-depth ecosystem was occasionally a challenge.

Finally, a tangent from above, but an important (and costly) lesson learned: when using ./spark-ec2 to administer EC2 clusters, running ./spark-ec2 stop [clustername] turns out to not actually stop the instances from Amazon's point of view. The flip side of this is that, when restarting a cluster, simply restarting the instances doesn't cut it: for nodes in the cluster to identify each other, they need to be reinitiated with a Spark script. So, to avoid losing money, we found that the proper workflow was the start an instance from the command line, stop them through AWS, and then when the cluster was needed again, start it from the command line again.

Conclusion

The amount of knowledge we acquired over the course of this project was truly astounding and is a testament to the steadfast legitimacy of 'learning by doing.' From unzipping a file into S3 to fitting a distributed deep learning model, each and every experience fostered the procurement of a new, invaluable skill. The path to where we currently stand was far from smooth, but every road bump along the way only served to cement indispensable knowledge that can be further applied to a wide range of environments.

Our team is incredibly proud of what we've been able to accomplish over the course of three weeks, but we are under no illusion that there is no room for improvement. For this reason, we have a list of next steps we would like to make in the very near future including, but not limited to, finding a better method of dealing with underrepresented classes, normalizing our features, incorporating a recurrent structure to capture the logical flow of a malware file, and being able to run our models on a more granular parse.

This project has sparked our interest in distributed machine learning, and we hope to continually improve the quality of the features we engineer, the accuracy of the models we build, and our mastery of the tools we use.