CSCI 8360 Data Science Practicum
Department of Computer Science
University of Georgia

Project 1: Malware Classification

DUE: Tuesday, February 12 by 11:59:59pm

Out January 22, 2019

1 Overview

Congratulations, you've all done admirably with the standard faire of text analysis. The scoreboard is impressive, and all without using any utility packages that run on Spark.

Your next task is similar, but different: you're performing classification of documents, rather than just word counting. You will probably end up using similar techniques to the ones you developed yourselves in P0. However, this time, you'll be using hexadecimal binaries as documents, and attempting to classify them into one of several possible malware families. **Your goal is to design a large-scale classifier in Apache Spark that maximizes its classification accuracy against a testing dataset.**

For this project, we are using the data from the Microsoft Malware Classification Challenge, which consists of nearly *half a terabyte* of uncompressed data. There are no fewer than 9 classes of malware, and each instance of malware has one, and only one, of the following family categories:

1. Ramnit
2. Lollipop
3. Kelihos_ver3
4. Vundo
5. Simda

6. Tracur
7. Kelihos_ver1
8. Obfuscator.ACY
9. Gatak

The format of the data is somewhat different from before. All the documents are in hexadecimal format, in their own files (one file per document); these files are located here:

`https://storage.googleapis.com/uga-dsp/project1/data/bytes/<file>`

and the Google Storage path:

`gs://uga-dsp/project1/data/bytes/<file>`

Each file looks something as follows:

```
...
00401060 53 8F 48 00 A9 88 40 00 04 4E 00 00 F9 31 4F 00
00401070 1D 99 02 47 D5 4F 00 00 03 05 B5 42 CE 88 65 43
00401080 6F 3D 4D 00 77 73 CD 47 21 A5 F0 48 87 8E 4A 00
00401090 DF 47 00 00 8A E6 41 4D 73 D7 4A 00 0B 45 00 00
...
```

The first hexadecimal token of each line, looking something like `00401060`, is just the line pointer and can be safely ignored (as these will appear throughout many of the files, not unlike articles in natural language). The other hexadecimal pairs, however, are the code of the malware instance itself and should be used to predict the malware's family.

Each binary file is independently identified by its hash, such as `01SuzwMJEIXsK7A8dQbl`. The files are named with their hashes (and have "`.bytes`" file extensions, after the hash; so a full filename would look like `01SuzwMJEIXsK7A8dQbl.bytes`); in a different directory are text files that contain these hashes, one per line, to indicate which files (or documents) are part of which dataset. The dataset definitions are located here:

`https://storage.googleapis.com/uga-dsp/project1/files/<file>`

and the Google Storage path:

`gs://uga-dsp/project1/files/<file>`

Specifically, here are the available files in the `files` directory:

- `X_train_small.txt`, `y_train_small.txt`
- `X_test_small.txt`, `y_test_small.txt`
- `X_train.txt`, `y_train.txt`
- `X_test.txt`

Each `X*` contains a list of hashes, one per line. Each corresponding `y*` file is a list of integers, one per line, indicating the malware family to which the binary file with the corresponding hash belongs.

There are "small" and "large" versions of the data available. The "small" is meant to help in initial algorithm development, as these data are small enough to fit on one machine. Only when you are confident in your core classification strategy should you move to the "large" dataset.

You'll notice there is no `y_test.txt` file. That file is being held on AutoLab, and will be used to evaluate your trained model and score your submission.

If you are so inclined, the raw text files of the programs, which were reverse-compiled from the `bytes` files above, are also available. These are *even larger* than the binary files, but could still potentially be useful for classification; using them would be considered extra. These are available at:

`https://storage.googleapis.com/uga-dsp/project1/data/asm/<file>`

and the Google Storage path:

`gs://uga-dsp/project1/data/asm/<file>`

The hashes are still the same as before for the binary files, but instead of a `.bytes` extension, the filename has a `.asm` extension.

2 Theory Guidelines

If you have no idea where to begin with designing a large-scale algorithm for document classification, this section is meant to provide a baseline to get you started. It should be noted, though, that to achieve grades in the **A**-range, you'll need to go above and beyond what is described here.

2.1 Naïve Bayes

Naïve Bayes is a probabilistic classifier based on Bayes' Theorem of conditional probabilities. Let's say we're working with data X, which is $n \times d$ (n instances, each with d

dimensions), each of which belongs to one of K possible classes. Given some instance $\vec{x} \in X$, where $\vec{x} := \{x_1, x_2, ..., x_d\}$, we want to classify it into one of K classes $y_k \in Y$.

$$P(Y = y_k | X = \vec{x}) = \frac{P(Y = y_k) P(X = \vec{x} | Y = y_k)}{P(X = \vec{x})}$$

However, solving this equation in practice is often infeasible, especially when the dimensionality of the data $\vec{x}$ is extremely high (i.e., d is arbitrarily large). This is where the "naïve" part of Naïve Bayes comes into play: we assume that, conditioned on the class y, that the features of $\vec{x}$ are independent of each other.

$$P(Y = y_k | X = \vec{x}) \propto P(Y = y_k) \prod_{i=1}^{d} P(x_i | Y = y_k)$$

We can compute this probability for each label y, and to perform the classification, return the label with the highest probability as our prediction.

$$\hat{y} = \text{argmax}_{k \in 1...K} P(Y = y_k) \prod_{i=1}^{d} P(x_i | Y = y_k)$$

For document classification, you can consider a document as a single instance $\vec{x}$ with d features, where d is the number of words in your vocabulary (the number of *unique* words across *all* documents). Conditional independence in Naïve Bayes imposes the bag-of-words model, where word ordering does not matter, only word frequency.

Word probabilities, then, can be computed with simple term frequencies, where the probability of a particular word conditioned on a class $P(x_i | y_k)$ is the number of times that word appears in documents belonging to class y_k divided by the total number of times the word appears in *all* documents.

Once you have trained your model–essentially, counted all the words in the training set under the different class conditionals–you can then test the accuracy of your model against a testing set.

One final piece of advice in this starter approach: you will need to be able to handle the case where a word shows up in testing that you never observed in training. Without any additional tweaks, your counter for this word would be 0, which then multiplied with the rest of the conditional probabilities of your document would result in an overall probability of 0 for all categories y_k. Not good!

The easiest way to deal with this is impose *pseudocounts*, where an extra count of 1 is added to all words in both training and testing, eliminating any 0 probabilities while still keeping probabilities of infrequent words small.

2.2 Beyond Naïve Bayes

A small list of "extras" you can undertake to boost your grade into the **A**-range. This list is NOT exhaustive, merely a few suggestions. I encourage you to come up with your own improvements! Given the extremely limited vocabulary in the binaries (only 16×16 possible "words", if you consider each two hexademical characters as tokens) and the lack of stopwords, merely applying your approach from Project 0 won't be enough this time.

- **Features**: At a bare minimum, you will need to expand the feature space beyond the base 256 vocabulary terms; bigrams or even higher-order terms would be a good direction. Furthermore, you'll need to enhance the representation of the features beyond simple word counts; TF-IDF would be a solid starting point. You could also investigate dimensionality reduction strategies or embeddings to further refine the feature space. And if you're really ambitious, you could try to find a way to incorporate *both* the binary *and* the text formats of the program files and construct a joint feature set.
- **Smoothing**: Pseudocounting prevents the worst-case scenario of a 0-probability label if you observe a word in testing that you did not observe during training. However, you could potentially implement more sophisticated additive smoothing that scales better with the overall size of the vocabulary.
- **Regularization**: From a different angle, regularization may help focus the decision boundary on the words that make the most difference in classification while simultaneously obviating the need for smoothing.
- **Stopwords**: There are plenty of examples of words that occur frequently but are semantically useless for classification. Removing these prior to training the model can help; unfortunately, there's no universally-agreed-upon stopwords list, so you'd need to come up with your own.
- **Algorithms**: Naïve Bayes is definitely the go-to for document classification, but other classifiers can also be used. Random forests (which are embarrassingly parallel!) and logistic regression are two examples of potential alternative classification strategies that have reasonable training routines.

3 Engineering Guidelines

Everyone must use Spark. What language API you use is entirely up to you and your team.

You can use ANY of Spark's built-in APIs, and any packages that run on Spark. This includes its built-in TF-IDF featurizers, its Singular Value Decomposition, or even its logistic regression. Packages that run on top of Spark can include things like XGBoost. You can also use regular libraries like `nltk` or `scikit-learn`, though obviously you can't use the learning models in either one (plus they really wouldn't scale to this kind of data, anyway). The restriction here is that you can't *not* use Spark.

Create a GitHub repository for you and your team to work on the project. You can make it private, and you can create a GitHub team within the DSP-UGA org, but the development must take place under the GitHub organization as I will be using it and the activity you record on it as part of the grading process. Name the repository with the scheme of `team-name-p1` to help differentiate from other teams and future projects. **Seriously, please name it correctly**.

Exercise good software design principles. This means: adopt a consistent coding style, document your code, use the GitHub ticketing system to identify milestones and flag bugs, and provide informative and frequent `git commit` comments. Also, be sure you include an informative `README`, an accurate `CONTRIBUTORS`, a `LICENSE`, and are organizing your code in a clean and logical way.

Use Google Cloud Platform (GCP). IMPORTANT: If you have never used GCP before, *go register now* and you'll receive \$300 in credits. Otherwise, everyone will receive a code they can redeem for \$50 in GCP credits. This is a virtual computing platform where you can allocate on-demand compute resources configured to your liking. In particular, pay attention to two specific products of GCP: *Storage* and *Dataproc.* Storage is basically a distributed hard disk (with the data paths given in Part 1 above), and Dataproc allows you to allocate and set up entire Spark clusters without installing them from scratch.
Shut down your GCP clusters when you are finished testing. GCP clusters incur costs for every hour they are turned on, even if they are not actively running any jobs. That said, it's also expensive to spin them back up if you're running a lot of jobs, so I would encourage you to turn them on, run a bunch of tests, and only when you're out of tests to run, you shut them down until you can get more in the pipeline.

4 Hints and Reminders

These datasets are *huge*; almost half a terabyte in total. In Project 0 you could kinda fudge your program using operations that weren't terribly scalable. That was by design, since you were probably still getting the hang of Spark. Here, though, those strategies will no longer work: you and your team need to determine a design that *scales.* Ideally some of the Project 0 codebase will be a good starting point, but if it's not, *you need to get to work immediately.*

Compute log-probabilities so you don't run into underflow errors. When you choose the conditional independence of Naïve Bayes, you often end up multiplying lots of tiny probabilities together, which has the potential to underflow. To prevent this, compute the log of these probabilities and, instead of multiplying, add them (as per properties of logarithms). Since the log is a monotonic transformation, then if $P(x) < P(y)$, it holds that $\log P(x) < \log P(y)$.

Consider all the variables you need to compute when counting tokens to solve the Naïve Bayes equation. In addition to the term frequency of a term under a specific label, you will also need a prior probability term for just the specific label, in addition to a marginal probability term of the word by itself. See if you can make clever use of Spark's functional primitives to tally these counts efficiently.

If you run into problems, work with your teammates. Open up GitHub tickets, discuss the problems, find workarounds. If you still run into problems, query the Slack channel. I'll be regularly checking Slack and interjecting where I can. Whatever you do, please don't turn into Lone Riders and try to solve the whole thing yourself.

Last year's top scorer obtained 99.0445%, just shy of the all-time accuracy of 99.311295% accuracy on the test set. Furthermore, the *lowest* competitive scores have been in the 93-95%! Suffice to say, it's going to be pretty easy to get a high score again, but the winner will probably be a matter of a few hundredths of a percentage point and will require every ounce of algorithmic creativity you can muster. Can anyone beat last year's benchmark, or nab a new all-time best?!

5 Submissions

All submissions will go to **AutoLab**, our brand-new autograder and leaderboard system.

One teammate should be designated for submitting on behalf of the team; they should submit to Autolab *using the team name* so I can easily match submissions with teams. If two people from the same team submit, I will pick one at random and grade that submission, so be sure you're communicating clearly.

When submitting, you will submit one file: **a text file containing the predictions of your trained classifier on the `X_test.txt` dataset, one prediction per line for each document**. For example, if you had three documents in the test set to classify, your output might look like:

You can make submissions on behalf of your team to the **Project 1** assignment that is open. When you do, you will submit one file: **a text file containing the predictions of your trained classifier on the `X_test.txt` dataset, one prediction per line for each document**. For example, if you had six documents in the test set to classify, your output might look like:

```
1
7
7
```

```
3
5
3
```

Your classification accuracy will be tabulated and compared against the true labels (hidden on the server) and should appear on the leaderboard.

If you think you can do better than what shows up–particularly to beat out everyone else–make another submission! There's no penalty for additional submissions, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on February 12, so give yourself plenty of time!

6 Grading

Given that this is a practicum, and that you're all talented graduate students, the grading for the projects is a little different, operating on a sliding scale with some stationary points that require extra effort to move beyond.

Everyone's starting grade is a **B** (85%). If you follow the guidelines in this handout, implementing the basic algorithm as described, and everything functions properly, and your documentation is reasonably ok, and there were no major issues with your team, then you also finish with a **B**. If you're happy with that, great! If not, read on:

Getting an **A** requires an investment of additional work, both in terms of functionality and algorithm performance, as well as on the software engineering side. You must demonstrate the extra customizations you made to your code and how those tweaks resulted in an improved score on the final testing set (getting the top score, for instance, would certainly qualify–though I feel I should point out that this sort of thing is relative, in that if you get a new top score that is only 0.000001% better than the previous one, it won't translate to an entire additional letter grade. #SorryNotSorry). If your pipeline is multi-staged, clearly showing that a lot of thought went into the theoretical design–perhaps even using a current method (that is *properly* and *prominently* cited)–it will help boost your grade.

Furthermore, you must show exemplary software engineering techniques: your code style should be clean, logical, and well-documented, and you should provide a clear README with instructions to new users on installing and deploying your document classifier. Your `git commit` comments and GitHub tickets should provide a "paper trail" of documentation to show the route your team took in developing the software. Unit testing, continuous integration tools, consistent and thorough use of branches and merges, and adherence to coding style standards (e.g. Python's PEP8) will also help push your grade above the default.

It should go without saying, but violations of standard coding practices or errors in the theory will drop your grade; you don't get a **B** for simply turning in an empty assignment.

This combination of **algorithmic novelty** and **engineering best-practices** will earn you and your team that well-deserved **A**.

7 Summary

In summary, these are the things I will look over when determining your team grade:

1. **Code**: Organization, structure, style, and clarity. Strategy and theory will also be a big component (what did you implement? was it implemented correctly? does the theory follow logically from the problem you're trying to solve?).
2. **Documentation**: This includes comments in the code, but also in the repository itself (GitHub wiki, `README` instructions). How easy would it be for someone to get up and running with your code, or to submit a bugfix?
3. **Accuracy**: Testing accuracy as submitted on AutoLab, and how far you are from the very top.
4. **Team**: Like the code, I'm looking for structure and organization, as well as a "paper trail" to get a feel for your team's working dynamics. How were project efforts divided among teammates? What do the git logs reveal about who was contributing to the project? How does this align with the `CONTRIBUTORS` content? Were code reviews conducted properly?
5. **Extras**: These include things like continuous integration tools, project websites, theoretical novelty (within the constraints of the project), unit tests with good coverage, among others. Use your imagination!