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

Project 1: Scalable Document Classification

DUE: Thursday, February 1 by 11:59:59pm

Out January 16, 2018

1 Overview

Welcome to the first project of the semester! Hopefully you're all geared up, thanks to Project 0, for some natural language processing. **Your goal is to design a large-scale document classifier in Apache Spark that maximizes its classification accuracy against a testing dataset.**

For this project, we are using the Reuters Corpus, which is a set of news stories split into a hierarchy of categories. There are multiple class labels per document, but for the sake of simplicity we'll ignore all but the labels ending in `CAT`:

1. CCAT: Corporate / Industrial
2. ECAT: Economics
3. GCAT: Government / Social
4. MCAT: Markets

There are some documents with more than one `CAT` label. Treat those documents as if you observed the same document once for each `CAT` label. For example, if a document has both the labels `CCAT` and `MCAT`, you will essentially duplicate that document and give one of them *only* the label `CCAT`, and the other *only* the label `MCAT`.

The format of the data itself is one document per line of the file, and a corresponding file with the ground-truth document labels. In effect, you have files `X.txt` and `y.txt` which

contain the same number of lines–in the former are the documents, and in the latter are the document class (label) listings on corresponding line numbers.

Specifically, here are the available files:

- `X_train_vsmall.txt`, `y_train_vsmall.txt`
- `X_test_vsmall.txt`, `y_test_vsmall.txt`
- `X_train_small.txt`, `y_train_small.txt`
- `X_test_small.txt`, `y_test_small.txt`
- `X_train_large.txt`, `y_train_large.txt`
- `X_test_large.txt`

The data have already been split into `train`ing and `test`ing subsets so you don't need to worry about implementing distributed cross-validation. Train on the X-training set (you should get very high training accuracy), then test with the correspondingly-sized X-testing set.

There are "very small," "small," and "large" versions of the data available. The "very small" and "small" are 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_large.txt` file. That file is being held on AutoLab, and will be used to evaluate your trained model and score your submission.

You can view the data on Google Cloud Storage through the urls:

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

...though be careful with this method for the larger text files. Also, you don't want to use those URLs from *within* the Spark programs; instead, you'll access the data from within Spark using the `gs://` protocol:

`gs://uga-dsp/project1/train/<file>` or `gs://uga-dsp/project1/test/<file>`

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!

- **Features**: Term frequencies are nice but they suffer from intrinsic selection bias, especially common words like articles and prepositions that contribute little semantic meaning. Using TF-IDF or hashing frequencies may improve classification accuracy.
- **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

I would strongly encourage everyone to read up on how **branching and merging** works in git. You can read some documentation here. Creating branches and merging them with master will help minimize code conflicts between teammates.

Everyone must use Spark. What language API you use is entirely up to you and your team. However, you are not allowed to use any packages that build *on top of* Spark. Furthermore, you cannot use algorithms implemented in Spark MLlib or ML–even the featurizing utilities are forbidden for this assignment. However, you are welcome to use local packages (like NumPy arrays or Breeze vectors). Additionally, you can use Spark

DataFrames if you prefer those to RDDs. Bottom line: if in doubt, ask me.

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.

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.

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 Google Cloud Platform

Starting with this assignment, we're using Google Cloud Platform (GCP) for hosting large datasets, and to give you elastic computing resources to deploy and test your algorithms.

First, you'll all be getting Slack messages with a code and instructions to redeem the code to receive $50 in GCP credits.

Second, while you can install the GCP SDK command line utilities (`gcloud` and `gsutil`), you can get along with the web interface. Nevertheless, the install instructions are here: https://cloud.google.com/sdk/docs/

Third, you can access your GCP Console here: https://console.cloud.google.com/ The product listing (click the drop-down in the upper-left corner) can be a bit overwhelming, so I would recommend clicking the "pin" button for **Compute Engine**, **Storage**, and **Dataproc**.

- Compute Engine is the general-purpose interface to the VMs you spin up. You have full control here over what VMs you spin up and how many, but specifically for Spark clusters, I would recommend using the Dataproc interface instead.
- Storage will show files you have saved in Google's distributed filesystem. It's a lot like Amazon's S3, if you've ever used that before. Everything starts with a `s3://` protocol prefix, followed by a bucket name (which you create), and then an absolute path just like any other filesystem. You can store whatever you'd like here, and

you'll definitely use this interface to upload your Spark driver programs/jars so they can be run.

- Dataproc is basically a re-jiggered user interface to Compute Engine, with distributed computing in mind: it offers pre-built Spark and Hadoop images (make sure you use Spark 2.2). It makes it super easy; I would highly recommend going through the Dataproc Quickstart: https://cloud.google.com/dataproc/docs/quickstarts.

Take care to explore the Dataproc interface! For instance, there's a web-based front-end to SSH directly into the VMs you spin up, so you don't even need to configure your console or Terminal application for SSH access; you can just click a button on the Dataproc page.

An important note with Dataproc: by default, the image used to replicate a Spark cluster uses a pretty limited Python environment. There are ways to customize the configuration of the image that is spun up with Dataproc clusters, but it does take a little bit of additional effort. To that end, I'll be working on a startup script that installs Python 3 and will notify everyone when it's ready.

5 Hints and Reminders

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 words to solve the Naïve Bayes equation. In addition to the term frequency of a word 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.

Be careful about text encoding; you may see a lot of "`&quot;`" constructs in the documents. Make sure you can parse these out so they don't become tokens in their own right. Your final classification accuracy will thank you for it.

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.

You only need to correctly predict the `CAT` label, and if there are multiple for a single document, you only need to get one of them for the answer to be correct.

Last year's top scorer obtained 96.842799% accuracy on the test set. They used some very clever additions to the "vanilla" algorithm (suffice to say, they earned an **A** on the project). It's pretty easy to get up around 90%, but those next 4-6 points are *really hard to get*. Can anyone beat last year's benchmark?!

6 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_large.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:

```
MCAT
CCAT
CCAT
```

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 Feb 1, so give yourself plenty of time!

DIRE WARNING: If you do not adhere to this output format and the autograder crashes as a result, your entire team will be forthwith banished from Athenshire! Seriously though, please adhere to this output format.

7 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**. 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). 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. Finally, 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.

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

8 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?).
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!