Big Data Coursework 2014 – Arthur Jack Russell (acka630)

Part 1 – reading and preparing text files

I decided to split the code for part 1 into two sections, to de-couple a) the creation of TF and IDF files, and b) the creation of hashed vectors. This enables several hash vector sizes to be tested without having to re-create the TF and IDF files each time.

To run cwk_part1a.py from the command-line, use the following template:

```
$ spark-submit --flags <path_to>/cwk_part1a.py <host> <library root> <list of library
child directories> <files per chunk> <max filesize in MiB> <stopwords file> <start
chunk>
```

A working example of this for my current setup is below:

```
$ spark-submit --driver-memory 8G ~/cwk_part1a.py lewes /data/extra/gutenberg/ text-
part 200 0.8 /data/student/acka630/stopwords/stopwords_en_jr.txt 0
```

Allowed values of <host> are “local” and “lewes”. This argument is used to set appropriate Spark Config settings (e.g. 2 cores for Lewes).

The next two arguments allow specification of a set of library directories if required, to tune the size of the library. E.g. to run on all files contained in text-full/1 and text-full/2 but not the rest of text-full, specify:

```
$ spark-submit --driver-memory 8G ~/cwk_part1a.py lewes /data/extra/gutenberg/text-
full 1,2 200 0.8 /data/student/acka630/stopwords/stopwords_en_jr.txt 0
```

The <files per chunk> argument specifies the number of files per to process in each chunk or batch. This allows tuning depending on the memory available – bigger chunks imply less I/O overhead but require more memory to process at once.

The <max filesize in MiB> argument is a filter that allows large files to be skipped. A quick analysis of text-full revealed that just over 40,000 of ~44,000 files were smaller than 0.8MiB – this is the filter I have used for the majority of the analysis.

The <stopwords file> argument specifies the location of a comma-separated stopwords file.

The <start chunk> argument allows part 1a to be restarted from a given chunk. This is useful in the event of an interrupted run. To start a fresh run, specify 0 here.

The key outputs of this code are the following pickle directories:
<current_working_dir>/pickles/TF/TF000…TFNNN (one pickle directory per chunk)
<current_working_dir>/pickles/IDF/

To run cwk_part1b.py from the command-line, use the following template:

```
$ spark-submit --flags <path_to>/cwk_part1b.py <host> <root for TF and IDF files>
<numChunks> <hash bin list> <minimum doc percentage> <destination for TFIDF>
```

A working example of this for my current setup is below:

```
$ spark-submit --driver-memory 8G ~/cwk_part1b.py lewes pickles/ 5 3000,10000 0.1
pickles/TFIDF/
```

Here you specify where to locate the TF and IDF files created in part 1a, the number of TF chunks to process, a list of hash-vector lengths, whether to truncate the vocabulary and where to output the TFIDF pickle files.

In the example, two sets of pickle directories containing TFIDF vectors are created as outputs:
<current_working_dir>/pickles/TFIDF/3000/TFIDF000…TFIDFNNN (one pickle per chunk)
<current_working_dir>/pickles/TFIDF/10000/TFIDF000…TFIDFNNN (one pickle per chunk)

The vocabulary truncation allows removal of words that appear very infrequently. My hypothesis was that words that appear much less often that the document frequency for the least common class being considered are unlikely to help predict any class, but may well contribute noise in aggregate as there are millions of them. This will be discussed further in part four. Specifying 0.1 here will mean that words appearing in fewer that 0.1% i.e. 1 in 1000 documents do not contribute to the hashed TFIDF vectors.

Part 2 – Reading and preparing metadata from XML

To run cwk_part2.py from the command-line, use the following template:

```
$ spark-submit --flags <path_to>/cwk_part2.py <meta path>
```

A working example of this for my current setup is below:

```
$ spark-submit --driver-memory 8G ~/cwk_part2.py /data/extra/gutenberg/meta/
```

The output of this code is the following pickle directory:
<current_working_dir>/pickles/meta/

Part 3 - Training classifiers

To run cwk_part3.py from the command-line, use the following template:

```
$ spark-submit --flags <path_to>/cwk_part3.py <host> <TFIDF parent folder> <meta
folder> <rank list> <numFolds> <output.csv>
```

A working example of this for my current setup is below:

```
$ spark-submit --driver-memory 8G ~/cwk_part3.py lewes pickles/TFIDF/10000/
pickles/meta/ 1,2,3 5 test.csv
```

Here we i) identify the location of the TFIDF vector chunks and meta data to be used for modeling; ii) specify the subjects to build classifiers for; iii) specify the number of folds to use in cross-validation, and iv) specify the destination for results output.

The parameters of the actual models which the script runs are hard-coded in the script itself. I decided on this approach as these are a bit too complicated to specify in the command line. It is also very clear in the script which parts to modify and how to change the parameters.

The output of this code is a CSV file with one line per prediction made by a model, e.g: <current_working_dir>/test.csv

The top 30 subjects by file count in /data/gutenberg/extra/text-full (de-duping by file ID and excluding files over 0.8MiB) are as follows:

Total number of files after de-duping and exclusion of files > 0.8MiB: **39,974**

PS (American Literature)	**5,613**	PN (Literature – General)	394
PR (English Literature)	**4,765**	Love stories	393
PZ (Children's Fiction)	**3,727**	**BS (The Bible)**	**385**
PQ (FR, IT, ES, PT literature)	**2,296**	Detective and mystery stories	382
AP (Periodicals)	1,594	**DS (History – Asia)**	**382**
Short stories	1,405	Historical fiction	354
Fiction	1,185	Western stories	343
Science fiction	**1,162**	**D501 (History – WW1)**	**332**
PT (Germanic literature)	**1,158**	PA (Greek & Latin lang. & lit.)	315
Poetry	518	PH (Uralic and Basque literature)	309
English wit and humor – Periodicals	**512**	PL (E. Asian, African, Oceanic lit.)	289
DA (History – Great Britain)	**507**	BV (Practical theology)	283
Adventure stories	471	BX (Christian Denominations)	261
DC (History – France, And., Mon.)	**410**	Children's stories	252
Conduct of life -- Juvenile fiction	405	**QH (Natural History and Biology)**	**245**

Early results indicated that several of the more general top 10 subjects were hard to classify. The top 10 is also almost exclusively literature and fiction. As a result I decided to discard a few of these from the analysis and focus on subjects further down the chart to give a broader spectrum of topics (e.g. humour, history, religion, science). I retained the top 4 to ensure that I had the biggest classes of literature represented, as well as Science Fiction which can plausibly be expected to have quite a distinct vocabulary.

For cross-validation I split the full corpus into N equal size folds. One was removed at the beginning for final testing, and I conducted cross-validation with the remaining N-1, taking one fold out each time for validation and training on the remaining N-2.

I used N = 4 folds for my analysis to minimize the time required to run the modeling while still giving a range of results and therefore some indication of variability across each model's predictions. Across all models' predictions, I also expected to see the distribution of the range itself. Given that I built thousands of models, this would give some statistical confidence in the variability of results. Performance on the test sample would act as the expected real-world performance, plus or minus the variation observed in cross-validation.

I used all records, with stratification into positive and negative classes per subject and random sampling of 1/N records within each stratum to maintain prior probabilities.

Sampling was done in Python, using numpy.random.sample. I would like to have used a seed so that my results could be replicated, but the numpy method doesn't offer this option.

The code loops over various values of the following regularisation parameters for each method: NB: Lambda; Trees: Maximum Depth, Maximum Bins; LR: Parameter Type (None | L1 | L2), Parameter Value.

I built models using three different vector sizes: 3000, 10000 and 30000. Initial testing indicated that only very weak predictions were possible on smaller vectors (300 and 1000).

I also varied the vocabulary used as described earlier. I used four different vocabulary lengths: the full vocabulary of all terms that appear once or more (6,920,258 terms); a truncation requiring document frequency (DF) >= 0.02%, i.e. the term had to appear in at least 8 out of the 40,000 docs (884,700 terms); a truncation requiring DF >= 0.1% / 40 docs (292,059 terms); and a truncation requiring DF >= 0.5% / 200 docs (92,256 terms).

In total I built 4,616 models on the full corpus of data with different permutations of vector size, vocabulary length and regularisation parameters. I tested each model on the training set that had been used to build it, the validation set and the test set. The full test results of these models including processing times, accuracy, error rates, recall, precision and variability of results are included in the Excel spreadsheet submitted with this report.

A summary of the F1 score for models built using selected regularisation parameters on hash vectors of length 10000, that encapsulates some of the key findings, is presented below.

Set	Validation																
Average of f1score				Subject													
Method	VectorSize	VocabLength	RegParam	PS	PR	PZ	PQ	Science fiction	PT	English wit and humor -- Periodicals	DA	DC	BS	DS	D501	QH	**Grand Total**
LogR		All test average		0.0%	0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
NBay	10000	Full (6.92m)	0.02	48.6%	40.5%	57.2%	50.8%	55.8%	44.3%	47.5%	36.9%	21.9%	59.4%	26.4%	48.8%	56.7%	45.7%
			0.1	48.6%	40.5%	57.0%	50.8%	56.9%	44.1%	47.1%	36.3%	22.1%	57.8%	27.2%	47.3%	57.7%	45.6%
		Trunc (2bp)	0.02	49.8%	42.8%	57.6%	51.0%	65.3%	43.4%	57.4%	38.9%	21.7%	60.6%	34.8%	47.6%	53.7%	48.0%
			0.1	49.8%	42.8%	57.5%	51.0%	65.4%	43.3%	59.5%	37.4%	21.8%	58.6%	33.8%	46.1%	52.1%	47.6%
		Trunc (10bp)	0.02	50.1%	43.5%	56.0%	51.9%	67.0%	43.1%	64.5%	41.5%	22.3%	57.9%	28.6%	48.5%	51.0%	48.2%
			0.1	50.1%	43.5%	56.0%	51.9%	67.0%	42.9%	64.2%	40.2%	22.3%	55.8%	28.7%	46.4%	48.5%	47.5%
		Trunc (50bp)	0.02	49.5%	43.8%	55.1%	49.7%	66.9%	41.8%	64.3%	37.6%	22.4%	55.1%	29.3%	46.8%	48.2%	47.0%
			0.1	49.5%	43.8%	55.0%	49.7%	67.2%	41.8%	64.9%	36.5%	22.4%	52.2%	29.3%	45.6%	46.6%	46.5%
Tree	10000	Full (6.92m)	(4, 20)	18.9%	3.1%	27.8%	48.5%	67.7%	43.8%	92.8%	16.8%	0.0%	58.4%	6.4%	26.1%	45.3%	35.0%
			(5, 10)	36.0%	10.0%	32.4%	41.7%	69.9%	33.2%	90.0%	16.4%	7.2%	53.2%	0.6%	12.3%	34.7%	33.7%
			(6, 5)	37.4%	21.6%	35.2%	44.4%	68.9%	34.3%	84.2%	16.8%	21.1%	56.7%	8.6%	15.2%	36.6%	37.0%
		Trunc (2bp)	(4, 20)	29.9%	1.9%	33.1%	52.7%	70.9%	43.6%	93.9%	17.6%	5.9%	60.7%	8.2%	20.9%	42.0%	37.0%
			(5, 10)	31.5%	16.1%	37.0%	36.7%	71.8%	32.3%	92.7%	14.2%	7.7%	58.9%	9.6%	18.0%	41.5%	36.0%
			(6, 5)	38.5%	23.4%	35.5%	39.3%	70.0%	35.8%	86.8%	16.3%	21.1%	59.8%	6.7%	20.3%	31.8%	36.6%
		Trunc (10bp)	(4, 20)	30.4%	1.3%	32.4%	48.1%	70.5%	42.2%	92.2%	12.9%	0.0%	61.7%	2.7%	39.9%	41.5%	36.6%
			(5, 10)	33.9%	12.5%	40.2%	42.7%	71.7%	35.5%	94.5%	24.7%	16.5%	61.7%	4.2%	25.3%	42.2%	38.9%
			(6, 5)	40.4%	20.6%	42.1%	41.3%	71.6%	37.0%	88.8%	20.2%	18.1%	58.9%	3.9%	23.8%	41.5%	39.1%
		Trunc (50bp)	(4, 20)	26.2%	0.5%	35.8%	50.0%	73.4%	39.7%	92.8%	26.3%	29.3%	59.3%	13.7%	24.9%	39.9%	39.4%
			(5, 10)	34.0%	29.9%	39.6%	49.2%	73.7%	43.4%	93.2%	18.9%	25.5%	60.9%	21.6%	23.4%	42.5%	42.8%
			(6, 5)	39.8%	28.9%	41.5%	43.2%	72.1%	35.6%	85.2%	19.4%	22.7%	60.0%	14.5%	22.2%	40.1%	40.4%

This table indicates that Logistic Regression (LR) performed much worse than the other two methods on F1, i.e. in accurately recalling positive subject matches. There were only a handful of cases where this method yielded even one positive prediction. Naïve Bayes (NB) was a more consistent performer than Decision Trees (Trees), but Trees provided the highest overall performance where they performed well.

The best regularisation value for NB using length 10000 hash vectors was for Science Fiction, using the most truncated vocabulary with a Lambda of 0.1. This achieved an F1 score of 67.2% on the validation set. The “real-world” performance of this model on the holdout test set was 63.5%.

The best regularisation value for Trees using length 10000 hash vectors was for English Wit and Humour - Periodicals, using Max Depth 5 and Max Bins 10 on the vocabulary truncated at DF >= 0.1%. This achieved an F1 score of 94.5% on the validation set. The “real-world” performance of this model on the holdout test set was 94.1%.

The best regularisation value for LR using length 10000 hash vectors was for PT (Germanic Literature), using no regularisation, on the full vocabulary. This achieved an F1 score of 1.2% on the validation set and 0.4% on the holdout test set. Even using a 30000-long hash vector only increased the best performance of LR to 3.0% (3.9% on holdout sample), again with no regularisation and on the DF >= 0.5% truncated vocabulary.

Part 4 – Efficiency, measurements and theoretical discussion

A summary of accuracy measures and running time for different vector sizes is shown in the table below (validation set results for models run using 39,974 documents from text-full):

Method	RegParam	VectorSize	Min Train Time	Min Classify Time	Average accuracy	Average recall	Average precision	Average f1 score
Naïve Bayes	0.02	3000	3.3	1.4	95.3%	38.1%	38.9%	34.1%
		10000	7.9	3.8	94.4%	59.6%	37.9%	44.0%
		30000	18.3	9.1	93.4%	74.7%	37.8%	48.6%
Decision Tree	(3, 10)	3000	7.8	3.2	96.1%	13.1%	19.7%	15.1%
		10000	23.8	6.2	96.1%	21.0%	28.3%	23.1%
		30000	72.3	13.8	96.0%	31.6%	52.7%	36.4%
Logistic Regression	(None, 0.0)	3000	43.6	2.1	95.7%	0.0%	0.0%	0.0%
		10000	108.7	5.9	94.8%	0.0%	0.5%	0.0%
		30000	257.2	14.3	93.3%	0.2%	4.8%	0.3%

Memory usage was harder to measure accurately. I used Spark’s UI to monitor real-time usage of the cache, but beyond that I suspect Spark has invisible methods of memory management including heavy temp-file usage on disk that obscures the full picture. Also, serialization errors (Kryo overflow) became an issue before memory when attempting to build deeper Trees on larger vectors. However some basic facts are the amount of memory required to cache the full set of vectors themselves. The 39,974 x 3000-, 10000- and 30000-long hash vectors required 770MB, 1440MB and 2030MB respectively. The Spark implementations of each model did some additional private caching, which typically doubled to tripled the amount of cache in use per regularisation parameter setting.

The most obvious point from the part 3 results is the poor performance of LR in accurately recalling positive examples of any class. This is likely to be due to the importance of prior probability for LR, and the fact that in most cases the subject classes represented a small fraction of all documents (i.e. the prior probabilities were low). NB does not have this feature – it is shown both positive and negative classes and characterizes them independently of their class probabilities. This means on average it should positively classify items into rare classes in proportion to their occurrence in the population. Similarly, Trees - given enough discriminating degrees of freedom - should also yield roughly the same number of positives (true or false) as there are in the population at large, assuming the attributes it branches on are distributed similarly in the training sample and population as a whole. However LR requires more evidence to go against prior probabilities. It is likely that significant noise present as a result of hash collisions caused a damping of the strongest class signals and therefore exceeding the threshold of positive classification was very rare.

As for training time, for 10,000 long hash-vectors, NB is ~3 times quicker than Trees which is in turn ~5 times quicker than LR. This is qualitatively what I would expect given the relative complexity of each approach. The growth of training time with vector size is interesting: For Trees it grows linearly with vector size – this implies that each possible branching is explored for each hash feature and costs roughly the same to compute. However for NB and LR it grows less than linearly. This implies that all positive data points have a fixed processing cost and that due to the increasing sparsity of the matrices as their length is increased, each feature has fewer non-zero data-points and takes less time to process.

As mentioned in part 3, NB was a more consistent performer than Trees on accuracy metrics. This feels intuitively right. NB uses every signal available - the average predictive power across e.g. 10000 hash buckets and the resultant level of accuracy will tend to remain quite steady between subjects. Trees is a much more volatile method – it relies on the 3-5 strongest signals (hash bins) per subject being sufficiently powerful and universal to accurately classify a document. The results observed such as F1 score of >90% for English wit and humour but several scores in the 20-30% range show that sometimes there are very accurately discriminating signals available, sometimes not.

Application scenarios, extensions and optimisations

This form of classification is likely to be useful in scenarios where reliable class-labels are unavailable, but there is a significant positive benefit in being able to classify items (documents in this case). It could be used by publishers to automatically classify manuscripts that are submitted to them and send them to appropriate departments. Beyond literature, the same approach could be used to classify and potentially rate CVs of job applicants to large companies who are constrained in the human resources they can apply to the problem. In all cases, an obvious extension would be enhancing the features the model can identify either to incorporate domain knowledge or just to filter noise. In the first example, the signal "vocabulary" could be generated in full but then modified with the aid of human input – i.e. the most salient features could be automatically identified but then manually pruned. This may be necessary in the HR example to avoid certain types of unlawful discrimination.

An enhancement to the algorithm overall could be to avoid the necessity to hash by just selecting the top features that comprise the "fingerprint" of each class. Inspection of the TFIDF lists confirms that this is a good proxy for relevance, but in many cases the score and (my perception of) relevance tails off after a few 10s or 100s of terms. I would like to explore the effect on accuracy of limiting the vocabulary to just these top 10-100s of terms per class. This would need to be done in such a way that the vocabulary isn't biased, i.e. independently of holdout test sets. Even in the approach for this coursework, I should technically have taken the test set out before even calculating IDF values as otherwise IDF is potentially biased by inclusion of document frequencies from test set. i.e. the effect of terms that appear more frequently in test set than the training set will be "damped" in the training phase as their frequency in test set will reduce their IDF.