Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the coursework instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct. Marks are provisional and subject to change in response to moderation, assessment board decisions and any ongoing investigations of suspected academic misconduct.

Parts 1, 2

Datasets and Machine used

I processed the files from text-part. I could create .TF files efficiently for any number of files, but for the .IDF calculation my program crashed after about 100 TF files on lewes, and about 50 on my home machine, I could never diagnose why (please refer to my email of 18/11) – memory problems I think - so that limited what I could do. I rewrote the IDF part of the program about 3 different ways but got more or less the same result.

I started working on lewes, but had speed issues, and found it a difficult environment not being able to use an IDE to edit python files directly. Dragging files to and from lewes for small edits slowed me down, and nano is not that productive. So I set up a windows PC and moved the job there, but after the hardware on that failed I moved everything to a 3 year old MacBook Pro. Interestingly run times on that machine were only about +50% of the runtimes on lewes. I could comfortably process 50 tfs in the IDF pass on the Mac so I left it at that, but it affected the validity of the stats I could collect for Part 3, as there really weren't many files for training/testing/validation.

You'll see in my code that I detected the machine name I was running on and set the data_path variable accordingly, rather than passing it as an argument.

Pre-processing - Overview

After an initial investigation of the txt files, I could see there were some things that needed to be dealt with, and that it would be good to deal with these as a pre-processing phase. There were a load of duplicate files, there were files with missing Ids, some files which were physically enormous and files which added nothing to the information, such as the human genome files. There were also txt files which did not have any meta data.

Total txt files	70454
Total rdf files	47196
Exclude : rdf files not in utf-8	-490
Exclude : txt with duplicate ids	-27760
Exclude : txt with no meta data	-698
Exclude : txt with missing Id	-1146
Exclude : txt which are very big (includes eg genome)	-328

Pre-Processing #1

I made Part 2 of the project a pre-processing step so I could identify IDs without any subject information. I made a pickled RDD of the subject list per file as per the instructions, but I didn't use it for my work in Part 1. Instead I made a .CSV file with ID, Subject list and created an RDD of this during my runs for Part 1, using the id of the TF being processed to filter the content and give me the Subjects for that ID. I got this idea after reading the O'Reilly Spark book where there is a use case of filtering a large log file by error statements. As Spark is a tool for text processing I figured I'd use text files to drive the analysis. It made checking quite easy.

Command line -> spark-submit CW-Part2.py
Creates -> XML.csv, MetaData.RDD

Pre-processing #2

Then I ran a file called preprocess-01.py which iterated over the files from text-full, identifying those with duplicate ids, no Id or no meta data (from argument rdf_file_name = XML.csv for my runs). This run creates file_analysis.lst which is a driver file for my main run.

Command line -> spark-submit preprocess-01.py EXCLUDES rdf_file_name

Creates -> file_analysis.lst, shown here

```
DUPLICATE,ID=22224,22224.txt,
GOOD, ID=22226,22226.txt,8,
GOOD, ID=22228,22228-8.txt,9,
DUPLICATE,ID=22228,22228-0.txt,
DUPLICATE,ID=22228,22228.txt,
GOOD, ID=22225,22225.txt,0,
GOOD, ID=22251,22251.txt,1,
NO_META,ID=22255,22255.txt,
NO_META,ID=22255,22255-8.txt,
GOOD, ID=22254,22254-8.txt,2,
DUPLICATE,ID=22254,22254.txt,
GOOD, ID=22252,22252-8.txt,3,
DUPLICATE,ID=22252,22252.txt,
GOOD, ID=22258,22258.txt,4,
DUPLICATE,ID=22258,22258-8.txt,
```

Pre-processing #3

preprocess-01.py can also be used to identify physically big files which should be excluded. I made a listing using "ls –lhR /data/extra/Gutenberg/text-full > txt_file_name" and used preprocess-01.py to look through this list for files bigger than a certain size specified in the run. I used a small python script to create the file "return_Exclude.py" from this .lst file and used it in my main run.

Command line -> spark-submit preprocess-01.py BIG txt_file_name
Creates -> exclude_big.lst

Main Run

Main file : CW-Part1.py <- see header comments

TF : Command line -> spark-submit CW-Part1.py TF number_to_process
IDF : Command line -> spark-submit CW-Part1.py IDF number_to_process
TF.IDF : Command line -> spark-submit CW-Part1.py TF.IDF

OR Whole thing : Command line -> spark-submit CW-Part1.py ALL number_to_process

Part 3

To run :

Naïve Bayes : Command line -> spark-submit CW-Part1.py 'NAIVE-BAYES'
Decision Trees : Command line -> spark-submit CW-Part1.py 'DECISION-TREES'
Logistic Regression : Command line -> spark-submit CW-Part1.py 'LOGISTIC-REGRESSION'

See Run_Stats.xls for listing of the results from my runs. I think there were too few files to give very robust data. There was some discussion about whether the calculation for Naïve Bayes should include the log or not, so I ran it both ways and using the log for normalisation definitely gave better accuracy, and run times took longer. (See XLS for details)

Naïve Bayes gave consistently good levels of accuracy for training and testing, there was a wider spread of results for both logistic regression and decision trees, both of which took longer (see 4a below for details)

See 4d and 4e for discussion of Decision Trees and Logistic Regression.

Part 4.

4a)

Classifier	Computing Time	Memory	Disk Usage	Accuracy
Naïve Bayes	Least	Less, no iteration	Same	Better on smaller sample [1]
Decision Trees	Medium	Higher, iterative	Same	Better for data with interdependencies
Logistic Regression	Highest	Highest, very iterative	Same	Better on larger samples [1]

Naïve Bayes, Decision Trees and Logistic Regression are all members of the family of Machine Learning algorithms termed Classifiers, used with text input to map a feature space to a label space. Classifiers produce a Boolean result, for instance a patient has or does not have an illness based on their symptoms (which are features in this case).

Naïve Bayes : Advantages of Naïve Bayes is that it is simple to understand, the model is based on a normalised count of word frequency in the training documents. Naïve Bayes is based on the assumption that there are no interdependencies between terms, so training Naïve Bayes is straightforward and faster because there is less to learn, information depth is unitary. This can be see even in the super-small sample set that I used in the coursework. The training times for NB are by far the lowest of the three methods. For the same hash size, on average, Decision Trees took +40% of the Naïve Bayes running time, and Logistic Regression took around 4 times longer than Naïve Bayes.

Although it applied to all 3 classifiers, the IDF calculation took significant memory resources though even with a relatively small vocabulary size, and limited the amount of training documents I could process. Anecdotally this also seemed to be true for many of my classmates.

Accuracy of Naïve Bayes is affected by the assumption of no interdependencies. For my small sample set Naïve Bayes gave around the same accuracy as Decision Trees, and slightly lower accuracy than Logistic regression.

Decision Trees : This model handles feature interaction very well, and is easy to understand conceptually. Outliers will naturally be identified by this model. Training time will be higher the more feature interaction there is, and it can take several iterations (and therefore machine resource such as memory and processing time) to determine the optimal maximum depth of the tree. To model Decision Trees well a reasonable amount of training data that expresses the most common interdependencies is needed. Decision Trees are more prone to over fitting than Naïve Bayes as they can generate complex hypotheses.

Accuracy of Decision Trees (for a representative data set) is typically higher than Naïve Bayes because interdependencies are part of the modelled information.

Logistic Regression : Similarly to Naïve Bayes, for a set of words (features) in a document Logistic Regression assumes no interdependencies. Logistic Regression is a discriminative classifier whereas Naïve Bayes is generative. Generative models use Bayes rule to calculate class probability given the inputs, whereas discriminative models learn directly from the inputs to generate the class. Ng and Jordan [1] compare classification for 15 datasets over 1000 random test/train splits and their findings are that, generally, logistic regression models have a higher error rate on smaller sample sets, but as the sample size increases the error rate generally falls to below that of Naïve Bayes. They found Naïve Bayes is typically more accurate over a shorter period of processing time, or a smaller sample set, whereas Logistic Regression gives better results over longer times and/or with higher sample.

4b)

Background : For an online retailing system to be as effective as possible it needs to personalise the content for each user, displaying the products the user is most likely to be interested in. The Holy Grail of such systems is to recommend products users didn't explicitly search for but are likely to buy. This not only generates increased sales, but encourages the user to both return to the site in the future and recommend it to others, generating traffic from users with a high propensity to purchase.

Speed of response at an acceptable level of accuracy for the users search will be the main measures of success criteria for the system. Users will abandon a site which is not responsive, or one which does not generate interesting matches after a few searches. Acceptable parameters for these measures need to be established by trial and error, and the best machine learning algorithm(s) for the system will be chosen accordingly. A user will have more patience with a specialist site (rare pens for instance) than one where there is a lot of choice (e.g. fashionable clothing).

Data Domain :

– Users : There will be two classes of user - anonymous and known. Anonymous users are not registered on the site and there is no history of their preferences or interests at the beginning of their visit. Known users have some recorded history of using the site, and the site can relate that data to the current session for the current user. Known users will have previous preferences that can be represented as text features and this can be used to generate matches during a session.

– Stocked Items : Each item for sale will have an associated set of features. Depending on the nature of the site (domain specific e.g. ski clothing, vs general e.g. household consumables) this might well generate high-dimensional sparse data, speed of searching will be improved with tf.idf hash vectors where possible.

– Searching behaviour : For each usage instance, search term data for each search iteration needs to be collected, along with results returned and the behaviour of the user, giving search

terms/result/behaviour data. Iterations should be related to each other so a chain of behaviour steps can be analysed as decision tree of searches as well as at a unary search/result/behaviour level. This all forms data of the class(text, text, text …) format which can be used to improve the machine learning.

– Feedback ratings : Typically recommender systems allow users to rate the accuracy with which the system has responded to their search. This data, along with actual sales can, again, form data of class(text, text, text …) format.

Use of TF.IDF vectors : Great for retail domains where features of stocked items are well represented by words, books for instance. TF.IDF can be used to identify terms relevant to the users search, filtering out noise. Less useful for items which have search features that have to be subjectively coded or tagged, e.g. films. For these products there will be little noise in the search terms and a utility matrix will be a more useful data structure for searching.

TF.IDF vectors will be easy to produce for products already represented by words. Tagging is a manual task at some point which requires work to be done.

Use of TF.IDF vectors enables large amounts of high-dimensional data to be represented in a hash vector, so online retailers are able to offer a much larger number of products than bricks and mortar stores (described as the Long Tail) with the same ease of searching. TF.IDF vectors allow for cosine distance between items to be determined, allowing similar products to be returned for a search, giving a higher likelihood of a sale (e.g. The Touching the Void/Into Thin Air example given in the course).

If detailed product reviews are left by users of the site, these can be treated as text data and TF.IDF vectors can be created to allow the reviews to be data mined by the site user and the site manager. This is an advantage for online stores who are able to collect and make use of reviews/and or recommendations very easily.

Decision Trees : For even a small site there are a lot of possible interdependencies of behaviour/feature selection/search iterations which make Decision Trees a questionable choice for use in real time. Bouza et al [2] investigate a system which builds a decision tree based on feature terms for search results using the user rating as a feedback mechanism. They report a low precision using this system. Decision Trees would be more useful in analysing behaviour post event to develop ranking of likelihood to buy.

Naïve Bayes : This is a useful model to consider for the online session as it is quick. Adding new products should be a relatively easy task, and potential sales volume could be increased by presenting other products with a similar feature set identified using cosine distance of the TF.IDF vectors.

Logistic Regression : Logistic Regression can extend the Success/Failure binary classification of Naïve Bayes to allow recommendations to be ranked according to likelihood of purchase (based on previous usage data) so extending the system to allow a list of recommendations to be presented, ranked by likelihood of relevance. Depending on the product domain of the site a recommender system might work across product types (e.g. if a user purchases shoes the site might want to recommend appropriate polishes), this is different from the strict text classification that Naïve Bayes deals with, and can be better addressed with a Logistic Regression model.

4c)

Text processing algorithms have many areas of application. Much of the knowledge in the world is recorded in the form of text in one way or another, especially with the pervasiveness of the internet.

As systems which take speech and render it to text in (near) real time improve, this also extends the applications of text classification and information retrieval algorithms considerably. If speech can be considered as simply a way of inputting text into a machine learning algorithm this generates many

possibilities, both in terms of powerful applications for the user and data collection for improving the machine learning algorithms driving them.

As the internet of things grows and produces data held in document databases this creates fuel for text processing applications. A use-case given in the O'Reilly Learning Spark textbook [4] is analysis of a system log file for errors, using text processing. Combining this example with Sparks' ability to process streaming data and extending the idea, systems could be built to monitor all sorts of sensor information and report on classes of conditions in near real time.

Geosensor information which collects a user's location and search query and retrieves information from a text store tagged location opens up a world of possibilities. The potential volume of data is enormous and needs big data techniques to make applications run fast enough to be useful in everyday life..

In their 2012 presentation of Spark RDDs Stoica et al [5] give use-cases of applications that have been built on text processing –

Traffic Modeling – a research study around the San Francisco area based on GPS data from cell phones and taxis GPS systems, allowing traffic flows to be monitored and managed.

Human Genome sequencing (Scalable Nucleotide Alignment Program), genomic data is held as text and so can be searched and classified.

Twitter Spam Classification : This is important as Twitter is a valuable tool of crowd sourcing information, and text from twitter, especially when human-classified with hashtags, can be analysed to quickly respond to natural disasters such as the Haiti earthquake [6] or used to attempt to predict Social attitudes [7]

The example given in the course of establishing authorship can be used to improve accountability in journalism and publishing, not only in traditional newspapers but in social media and online blogs.

The medical field has a lot of application areas for text processing. For instance cancer researchers searching the web for new papers or articles relevant to their specific research. Or for establishing the best treatment protocol for a patient based on their detailed history. New studies suggest that men and women react differently to certain drugs for instance, and treatment protocols are becoming increasingly customised on an individual's unique set of features.

The city of Chicago [8] mined text data collated from various sources such as incoming crime reports to look for relationships between crime and stores with liquor licenses, allowing them to manage their response capacity.

References

[1]. Ng, A and Jordan, M., On Discriminative vs. Generative classifiers: A comparison of logistic regression and Naïve Bayes. University of California, Berkeley (2001)

[2]. Bouza, A., Reif, G., Bernstein, A., and Gall, H., Semtree: ontology-based decision tree algorithm for recommender systems. In International Semantic Web Conference, (2008)

[3] Zhang, T., and Iyengar. V., Recommender Systems Using Linear Classifiers. Journal of Machine Learning Research 2 (2002)

[4] Karau, H., Kowinski, A. and Zaharia, M., Learning Spark. O'Reilly, (2014)

[5] Zaharia, M., Chowdhury, M., Das, T., Ankur, D., Ma, J., McCauley, M., Franklin, M., Shenker, S. and Stoica, I., Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. University of California, Berkeley (2012)

[6] http://www.pbs.org/newshour/rundown/haiti-quake-propels-twitter-community-mapping-efforts/

[7] Asur. S., and Huberman. B., Predicting the Future With Social Media. HP Social Computing Lab. IEEE, (2010)

[8] http://www.mongodb.com/post/35281515963/chicago-cuts-crime-with-mongodb

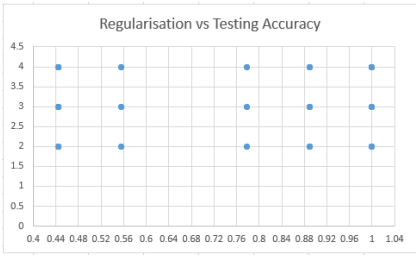
4d)

Although I used what I consider to be a good programming techniques for the following 2 questions, I don't think I managed to gather any useful data. I suspect I have misunderstood something in the setup of the models (although I based them of the labs fairly closely) or the number of files is too small to produce meaningful statistics. I present the findings here purely to demonstrate I have attempted the task and that I understand these figures are not meaningful.

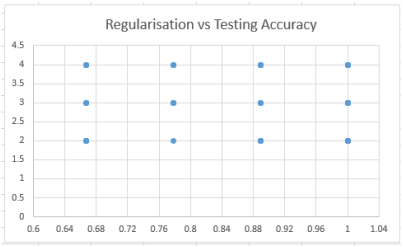
I used pythons string.format() construction to create .CSV files of data from the runs at 3b. I used a variety of hash table sizes and maxDepth settings. I put my data into Excel to analyse the results.

Subject	HashSize	maxDepth	Train File	Test File c	Validate Fil	Time	Train accu
English	10000	4	36	9	5	2.168558	1
English	10000	3	36	9	5	2.172814	1
English	6000	3	36	9	5	2.196884	1
English	8000	3	36	9	5	2.196991	1
English	6000	4	36	9	5	2.197689	1
English	8000	4	36	9	5	2.205704	1
English	8000	2	36	9	5	2.551686	1
English wit and humor -- Pe	6000	4	36	9	5	2.578906	1
AP	8000	2	36	9	5	2.617311	1
English	6000	2	36	9	5	2.629782	1
PZ	8000	4	36	9	5	2.631233	1
English wit and humor -- Pe	6000	2	36	9	5	2.639199	1
English wit and humor -- Pe	8000	2	36	9	5	2.642713	1
AP	10000	2	36	9	5	2.644757	1
PZ	10000	2	36	9	5	2.663438	1
PQ	8000	4	36	9	5	2.670003	1
AP	8000	3	36	9	5	2.674293	1
PQ	10000	3	36	9	5	2.681581	1
AP	6000	2	36	9	5	2.684608	1
PQ	8000	3	36	9	5	2.704173	1
PZ	6000	2	36	9	5	2.70707	1

Testing accuracy vs Regularisation on 9 testing files gave the following results



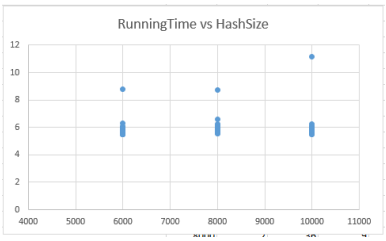
Logistic Regression



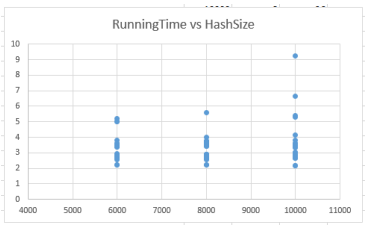
Decision Trees

4e)

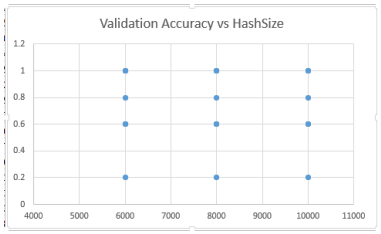
Running Time and Validation Accuracy vs HashSize –



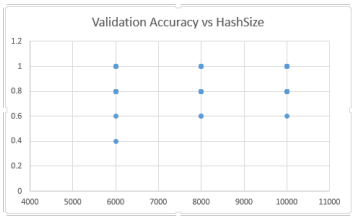
Logistic Regression



Decision Trees



Logistic Regression



Decision Trees

Logistic regression consistently took around twice the time to run as decision trees, and gave marginally less accuracy. The size of the hash vectors didn't seem to make much difference to run times or accuracy.

Hash sizes of 10000 took maybe 0.1-0.2 seconds less to run than 8000 overall, but I really think you'd need far larger sample sizes to measure any effect.