CS 5604 Information Storage and Retrieval
Spring 2016
Project Report

Text Classification

Team Members:

Hossameldin Shahin <hshahin@vt.edu>
Matthew Bock <mattb93@vt.edu>
Michael Cantrell <mcantrell@vt.edu>

Project Advisor:

Prof. Edward A. Fox

5/4/2016
Virginia Tech, Blacksburg

Abstract

In the grand scheme of a large Information Retrieval project, the work of our team was that of performing text classification on both the tweet collections and their associated webpages. In order to accomplish this task, we sought to complete three primary goals. We bagan by performing research to determine the best way to extract information that can be used to represent a given document. Following that, we worked to determine the best method to select features and then construct feature vectors. Our final goal was to use the information gathered previously to build an effective way to classify each document in the tweet and webpage collections. These classifiers were built with consideration of the ontology developed for the IDEAL project. To truly show the effectiveness of our work at accomplishing our intended goals, we also provide an evaluation of our methodologies.

The team assigned to perform this classification work last year researched various methods and tools. Some of these proved to be useful in accomplishing the goals we set forth. Last year's team developed a system that was able to accomplish similar goals to those we set forth with a promising degree of success. Our goal for this year was to improve upon their successes using new technologies such as Apache Spark. Spark provided us with the tools needed to build a well optimized system capable of working with the provided collections of tweets and webpages in a fast and efficient manner. Spark is also very scalable, and based on our results with the small collections we have confidence in the performance of our system on larger collections.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The classification team's goal is to take collections of tweets and webpages and classify them based on their relevance to given classes or topics. Our team fits into the grand scheme of the project by working between the Collection Management team and the Solr team. The Collection Management team was responsible for taking the raw tweet and webpage data and filtering out any obvious spam, vulgarity, or otherwise unreadable and unwanted content. They then wrote the cleaned data into a table in our HBase instance. Once this was done, we attempted to classify each document's relevance to a specific collection. We explored several methods for accomplishing this, starting with the methodology laid out by last year's Classification team [2]. Once we have the data classified, we are be able to pass it along to the Solr team by writing the newly classified data back into the primary HBase table as a column family. The Solr team is then be able to use the column family in the indexing of all the tweet and webpage data. They then provide the indexes for the data to the Front End team, allowing anyone to make use to the system we have created..

We begin by documenting our understanding of some of the pertinent literature, namely the course textbook and the Classification team report from last year; this can be found in Chapter 2, our literature review. Chapter 3 is the primary section of the document and includes our discussion of the project requirements, an outline for our design for the classification portion of the larger project, and finally a breakdown of our progress and suggested future work for the text classification project.

These chapters are then followed by a User Manual in Chapter 4, where we discuss the details in which a user of our methods and programs would have interest. We then include a separate Developer Manual in Chapter 5, which documents and includes details of the code base so that it might be extended and leveraged by other developers.

Following this we include our conclusions about the state of our final work and present our thoughts for possible future work in Chapter 8.

Chapter 2

Literature Review

2.1 Textbook

The textbook [5] introduces the classification problem. That is, given a set of classes, the goal is to determine what subset of those classes a document may belong to. To that end, the textbook describes a number of methodologies for selecting features. These features are then used by one of the classification methods discussed. The primary methods discussed are Naïve Bayes, Vector Space Classification, and the Support Vector Machine.

2.2 Papers

The primary paper that has been used as a reference is the final report of the Classification team from last year [2]. We have read through this report to understand the progress that the Classification team made last year as well as using the paper to gain an understanding of the task and interactions of the systems that we are using to perform our work. At a high level, the paper described a methodology employed by the team in which they were able to apply the Naïve Bayes method to classify sets of data. The team used Apache Mahout machine learning technology to generate Naïve Bayes classifiers to make predictions for new data. The biggest difference from last year's work to this year's work is that we primarily used Apache Spark, and so while wereable to reference their work, we used entirely different technologies and needed to modify our approach accordingly.

Initially, we attempted an approach where we would issue queries to Solr to build a set of training data for a classifier. However, we were informed by the GRAs that this approach would not work because the Solr API was not exposed on the cluster, and so we would not be able to access it. When we started looking into new approaches, we were advised to look at Frequent Pattern Mining (FPM). We looked at a paper by Han et al. [3], which presents a novel frequent pattern tree (FP-tree) structure and FP-tree based mining method called FP-growth. This method allows for mining the complete set of frequent patterns by pattern fragment growth. In this paper they also demonstrate that their new method is an order of magnitude faster than the typical apriori algorithm. The paper goes on to compare the performance against other popular data mining algorithms and discusses the use of the algorithm on some large industrial databases. We make use of the FP-growth algorithm and data structure in our text classification methodologies.

Chapter 3

Requirements, Design, And Implementation

3.1 Requirements

Based upon group discussions in a classroom setting, it was decided what information that the Collection Management team would provide to the other teams that need to work with cleaned tweet data. A full specification of these decisions can be found by viewing the Collection Management team's report; however we will briefly discuss the points that were relevant to us.

Given the cleaned tweet data from Collection Management, our team is able to perform the methodologies we describe later to classify whether a document is relevant or non-relevant to a given class. A detailed layout of the project with our interactions highlighted is provided by Figure 3.1.

We then place our classification information in the HBase datastore to mark the relevance of each document. The Solr team then indexes this class information that we provide, allowing for more robust query results and more useful facets to be created.

3.2 Design

Our design is based primarily off of recommendations from the GRAs assisting the class. We have also taken substantial input from last year's Classification team [2] and Professor Fox.

We designed our solution around pulling training data from and testing on a small collection of around 100,000 tweets. This was originally going to be performed on the small collection that was assigned to our team, `#germanwings`. However, due to some changes and in-class discussion among the other teams, we decided to continue with designing and testing our solution using a cleaned dataset provided by the Collection Management team. This dataset was the `#wdbj7shooting` small collection.

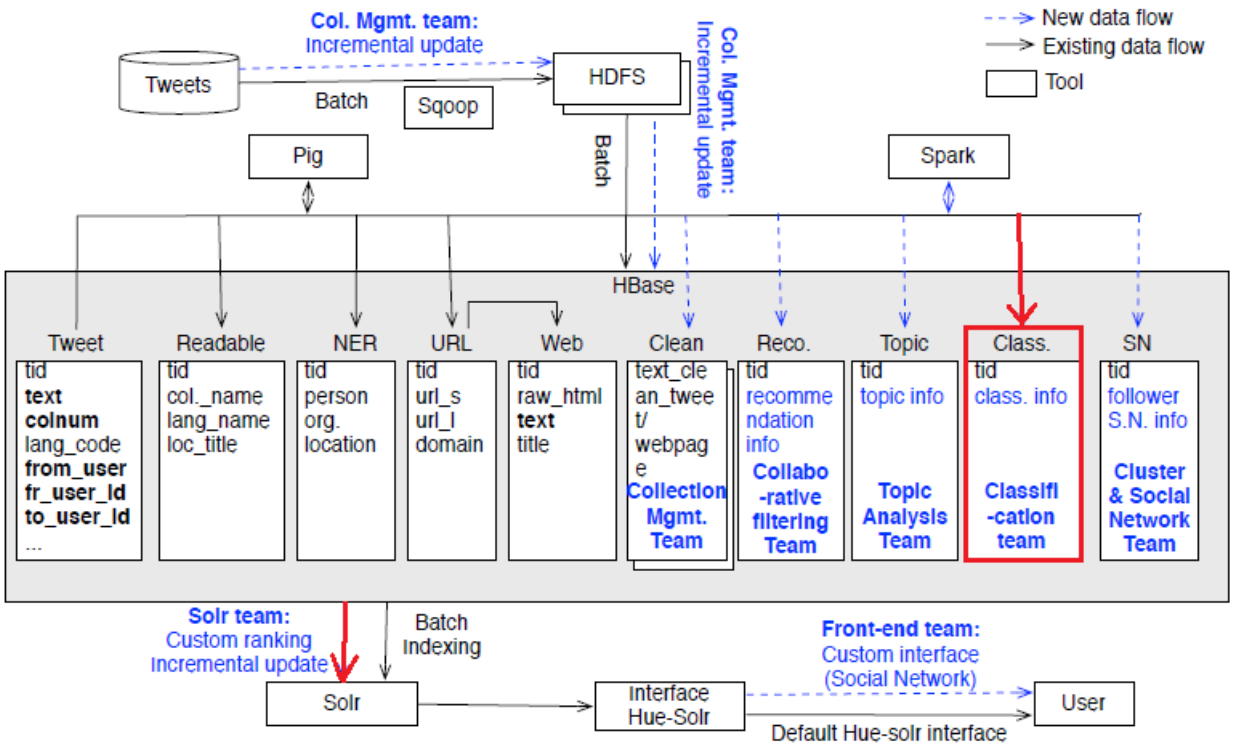


Figure 3.1: Layout of the project teams, as provided by Professor Fox and his GRAs.

3.3 Implementation

Our implementation can be easily laid out in a step by step workflow, with only one step requiring true manual input and evaluation. Figure 3.2 illustrates the general flow of data that we have implemented thus far. Below we will discuss each step in detail.

Our methodology primarily revolves around building a training set. This training set is used for training a machine learning model, a.k.a. a classifier.

3.3.1 Environment Set–Up

Our group decided initially to avoid working directly on the cluster, granting us full administrative rights to a machine, which allowed us to set up and test different configurations and algorithms beyond what might currently be supported by the Hadoop cluster being used for class. During this time we made use of a Virtual Machine generously provided to us by the Virginia Tech IT Security Lab, as one of our group members is a member of that lab.

However, after the necessary upgrades were performed on the Hadoop cluster, we have migrated all of our processing over to the cluster used by the class. This allows us to place our data directly in the HBase datastore instead of dealing with the transfer of data files from our separate machine to the cluster. This migration also allowed for other groups to easily make use of our scripts and for us to write our classification data directly to the cluster.

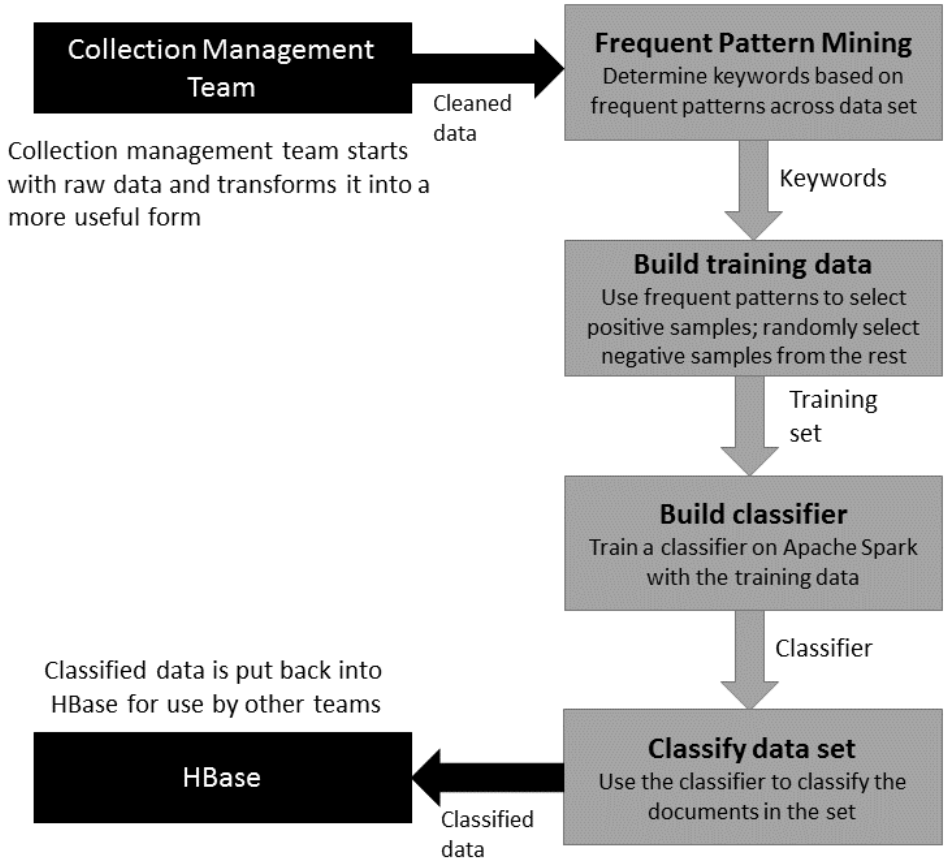


Figure 3.2: High level overview of the flow of data through the classification system.

3.3.2 Building the Training Data

In order to begin the classification process need to prepare a set of training data to use for the machine learning classification algorithm. In order to do this, we assume that we are working with provided data that has been cleaned of profanity and non-printable characters.

For our methodology we then need to take the content of each tweet and webpage as a string and tokenize it. This means that we remove any duplicate words and have the result be a set of the vocabulary in the tweet (or webpage). At this stage we also remove any stop words that are in the vocabulary to make sure our algorithms in the next phase are not skewed by taking stop words into account. During this phase we also strip the # from any hashtags that are present in the tweet.

In our initial design and testing we did not yet have access to cleaned tweets and webpages from the Collection Management team, so we worked primarily to build the methodology that would be used once those resources became available. Therefore, some of the steps mentioned previously, such as the removal of the stop words and the removal of the # from hashtags are unnecessary when working with the data provided by Collection Management in HBase.

The next step in our algorithm involves the use of Frequent Pattern Mining (FPM) to determine the most frequently used patterns of words in the text of a series of tweets or webpages. FPM looks at the set of existing vocabulary for each text and determines which tokens appear together most often within a tweet's

vocabulary.

This is the stage where manual intervention is necessary. We look at a file containing a sorted list of all the frequent patterns; from this we choose a set of terms that appear the most frequently and most accurately represent the collection. In our attempts at training a classifier for our small data collection, we chose to select a frequent pattern of maximum four words. This was essentially an arbitrary choice, but we believe that it strikes a good balance between being too specific and missing some relevant tweets and being too broad and pulling in a number of non-relevant tweets.

We now need to create positive and negative training samples. To do this, we pull a random subset of tweets that contain our frequent pattern and classify those as positive samples. For our negative sample we pull a random subset of tweets that do not contain our frequent pattern. To allow for enough training data, we select approximately 5-10% of the data in the collection as training data for our classification labeling.

3.3.3 Training the Classifier

Then we use the sets of positive and negative samples to train a classifier. We are using a logistic regression classifier in our implementation, primarily due to its ease of implementation in Spark.

FPM allows us to develop a training set by filtering the tweets and webpages down to some subset of those that do and do not contain the selected frequent patterns. We take these subsets as the positive training data and the negative training data.

At this point we feed the selected documents into the classifier; this step uses all the words in each document of the training data, not just the words used for FPM.

To ensure that this labeling is at least mostly correct, we inspect both the positive and negative training sets (or a portion of them) to confirm that each set is composed of either relevant tweets and webpages or non-relevant ones as appropriate. If in this inspection we find that the choice of frequent pattern has generated a poor training set, we choose a different frequent pattern and attempt the process again.

These attempts are relatively easily repeated due to the fact that the running of our FPM algorithm only requires approximately 10 seconds for each collection and the preparation of the training set and prediction for each tweet takes about 1 minute for each collection. The largest time spent is in the manual selection of a frequent pattern for training set generation.

Since we are using Spark CrossValidator, with careful selection of the frequent pattern we only need to perform a single iteration of this procedure for each collection. CrossValidator takes a range for each regularization parameter and number folds, then trains multiple classifiers and select the best and return it. We use the best model selected to perform the prediction.

3.3.4 Predicting the Class

After training the classifier, we apply the classifier to all the tweets across our small data collection. This results in a normalized floating point scoring for each tweet on a scale of non-relevant (0) to relevant (1) to the collection based upon the training data we selected.

3.3.5 Evaluating the Classifier

We can evaluate the accuracy of our model by judging how well it classifies some of the data that could have been in our positive or negative samples. This is the most intuitive evaluation however it requires a great deal of manual effort to perform. We need to pull out a sampling of classified data and look through it manually, marking whether or not the classification was correct.

Another evaluation that we can perform looks at how well each small collection does at being relevant to the topic at hand. If a large number of the documents in the collection are classified as non-relevant, then the collection as a whole has not done a good job capturing the event in question.

Finally, we need to perform an evaluation of how well our method of using FPM to determine the training documents works. This is much like the first evaluation in that it can be very manual. We need to dump the generated training sets out to files and then manually decide whether a given set has all (or primarily) relevant or non-relevant documents, as appropriate.

In order to begin to accomplish these evaluation goals, we built 7 surveys using Google Forms and distributed them to the class. Given the class size, and with each survey holding 100 randomly selected tweets, we planned to be able to have 3 people classify each of 700 tweets. This was to allow for us to compare the relevance labeled by our classmates to the relevance labeled by the classifier.

Due to some limitations in Google Forms, we had to prepare some Google Scripts code to generate a form from a list of questions on a Google Sheets document. We include this code in the submission to VTechWorks.

Evaluation Results

As mentioned previously, we constructed 7 surveys of 100 tweets each and distributed them to the 20 students and professor for the class, totaling 21 people to perform the labeling. This was intended to provide 3 responses per tweet, we then planned to average the answers to provide a similar score to that output by the classifier.

Unfortunately, surveys are often inconsistent, and we feel that our evaluation may be somewhat effected by the inconsistencies encountered:

- Only 675 tweets were labeled

  There were 25 tweets that were never labeled by any of the participants in our surveys, we are unsure of whether this was due to an error with the survey construction itself or if the students simply quit before completing the survey. If the latter, it brings into question the validity of their other labels because they may have just been rushing through.

- Of 21 asked to participate, only 16 responses were received.

  One of the 7 surveys only had one participant respond and label the requested tweets. Another survey had one participant only label 3 tweets. This lack of response obviously effects the averaging to compare with the classifier results.

For our comparison, we weighted each of the three possible labels — Relevant, Maybe Relevant, and Non-Relevant — as 1.0, 0.5, and 0.0 respectively. We then averaged the responses to compare with the output of

our classifier. This was not the most robust method of evaluation for our classifier, but it provided a general idea of the accuracy of our classifier on this small collection.

Figure 3.3 shows the distribution of average answers for the 675 tweets. These were calculated by assigning a number to each answer as mentioned above, then averaging those answers. This average was then divided evenly into thirds to denote Relevant, Maybe Relevant, or Non-Relevant. Figure 3.4 shows the distribution

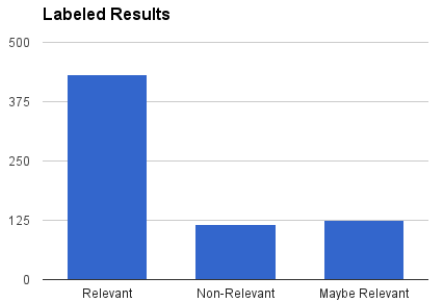


Figure 3.3: Manually Labeled

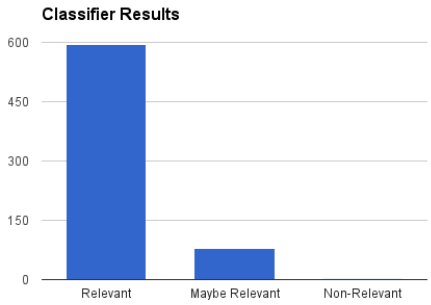


Figure 3.4: Classifier Labeled

that would have been generated by our classifier. An important point here is that the thresholds were set at the same values for the two histograms. The classifier results were evenly divided into thirds to create the figure.

The key point to note is that while with the thresholds as they currently are, our classifier places too many tweets as relevant, the general trend of the classifier is very similar to that of the labeled tweets. We feel that with some simple modification of the threshold values as to what constitutes relevant, non-relevant, and maybe relevant, our classifier could be shown to perform very accurately compared to the manually labeled tweets.

3.3.6 Interfacing with HBase

Manual File Uploading

We are currently able to process the six small data files and determine a probability representing how likely it is that a specific tweet is relevant to the collection. For each of the data sets we have produced a .tsv file with two columns: one containing the tweet ID (which serves as the row key in HBase), and one containing a value between 0.0 and 1.0 which represents the probability that the tweet is relevant to the category.

Within HBase, we have created a column family named "classification." Within our column family, we only have need of one column, named "relevancy." This column is where the probability value for each document in the database will be stored for access by other teams. Figure 3.5 shows an example query for a row, which returns all of the data in that row. The data produced by our system is highlighted.

For this version of the system, we are inserting the data into HBase manually. We accomplish this by using Hadoop's importTsv script as explained in the tutorial provided by the Solr team. The importTsv script is provided by HBase and allows you to take a .tsv file and specify what the columns in the file contain. We accomplish this with the following command:

```
hbase(main):001:0> get 'ideal-cs5604s16','602-584403586253135874'
COLUMN                                                CELL
 classification:relevance                             timestamp=1461087318835, value=0.995566893566
 clean_tweet:clean_text                               timestamp=1460396518198, value=RT madpigs The whol
 clean_tweet:hashtags                                 timestamp=1460396908877, value=#Garissa
 clean_tweet:urls                                     timestamp=1460396722111, value=http://...
 tweet:archivesource                                  timestamp=1460215032285, value=twitter-search
 tweet:cleantext                                      timestamp=1460215032285, value=RT @madpigs_: The w
                                                      A2\xE2\x82\xAC\xC2\xA6
 tweet:colnum                                         timestamp=1460215032285, value=602
 tweet:created_at                                     timestamp=1460215032285, value=Sat Apr 04 17:14:07
 tweet:from_user                                      timestamp=1460215032285, value=_WhiteKidAustin
 tweet:from_user_id                                   timestamp=1460215032285, value=403231760
 tweet:geo_coordinates_0                              timestamp=1460215032285, value=0.0
 tweet:geo_coordinates_1                              timestamp=1460215032285, value=0.0
 tweet:geo_type                                       timestamp=1460215032285, value=
 tweet:iso_language_code                              timestamp=1460215032285, value=en
 tweet:profile_image_url                              timestamp=1460215032285, value=http://abs.twimg.co
 tweet:source                                         timestamp=1460215032285, value=<a href="http://twi
 tweet:text                                           timestamp=1460215032285, value=RT @madpigs_: The w
                                                      A2\xE2\x82\xAC\xC2\xA6
 tweet:time                                           timestamp=1460215032285, value=1428167647
 tweet:to_user_id                                     timestamp=1460215032285, value=
 tweet:tweet_id                                       timestamp=1460215032285, value=584403586253135874
20 row(s) in 0.2210 seconds
```

Figure 3.5: An example row in HBase. Our data is highlighted.

```
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \
  -Dimporttsv.columns=HBASE_ROW_KEY,classification:relevance \
  ideal-cs5604s16 DATA_FILE
```

Here you can see we specify in the "-Dimporttsv.columns=" parameter that the first column in our .tsv file corresponds to the row key and the second corresponds to the classification:relevance column. We also specify that we are writing to the "ideal-cs5604s16" table and give it the file to upload.

Direct Reading and Writing

Using the importTsv script has allowed us to work on our system in our own virtual environment, since we are able to produce output files that can then be manually uploaded to the cluster. This was very useful while we were waiting for the software on the cluster to be updated so that we would have access to necessary libraries. Since the software on the cluster was updated, we moved our implementation onto the cluster; making it so that our classification system directly reads data from, and writes data to, the database.

Directly reading and writing from the database has several advantages. The main advantage and the primary motivation for doing it this way is that it allowed us to automate the process. Our system will be able to work directly with the HBase table, and will not need someone to manually feed in data files as input and handle the files it produces as output. This will greatly simplify the classification process for any future data that gets added to the database.

Directly reading and writing from HBase is done via the pyspark library provided by Spark. The library allows python to establish a connection to HBase which it can use to stream data back and forth. We first created a rudimentary "hello world" version of this process working on our local virtual environment. This simplistic version of the read/write functionality was able to scan the table for rows which fit some filter parameter (in this case, they were in a specific collection). It would then use the data from each row to populate another column in the same row.

Since then, we have improved our reading and writing system to be fully end-to-end. The system can now read information directly from HBase to perform its Frequent Pattern Mining, and write the prediction results directly back into the table.

```
tweets = hbase_rdd.flatMapValues(lambda v: v.split("\n")) \
          .filter(lambda x: x[0].startswith(broadcastCollectionNumber.value) \
            and json.loads(x[1])["columnFamily"] == "collection-management") \
          .map(parse_tweet) \
          .map(lambda x: Row(id=x[0], text=x[1])) \
          .toDF() \
          .cache()
```

Figure 3.6: Python code for formatting data read from HBase.

Figure 3.6 shows a snippet of the code used to read and format data straight from HBase. We use Spark to create a Resilient Distributed Dataset (RDD), then we perform several operations on that dataset. First, we split the data into separate elements at every newline character. This will split the columns within the same column family into individual RDD elements. next, we apply a filter to the data. This filter will get rid of any data which is not desired. In this case we filter out elements which are not in the collection we are currently processing, and elements which are not in the column family that we are interested in. Next, we have two mapping functions. The first applies a regular expression to do some basic cleaning of the tweet text. The second formats the data into a tuple containing first the id of the tweet and second the text of the tweet. Finally, we convert the set of tuples into a dataframe and cache it in memory so that we can perform the frequent pattern mining operations on it.

Writing back to HBase follows a similar procedure. Once we have processed the data and determined the probability to store in HBase, we need to format the data into a tuple formatted as such:

```
(row key, [row key, column family, column name, value])
```

In our case, the row key is the tweet collection number and id, the column family is `classification`, the column name is `relevance`, and the value is the probability. This information can then be written into HBase via the `saveAsNewAPIHadoopDataset` function provided by pyspark.

Chapter 4

User Manual

4.1 Installation Requirements

In this documentation, we make the assumption that you have already installed at least Spark version 1.5.0 and have it configured properly. If you have not performed this step, you can find installation and setup instructions at `http://spark.apache.org/downloads.html`.

The next step will be to clone the Git repository of our team. This can be found hosted on Github at: `https://github.com/hosamshahin/Spring2016_IR_Project.git`

Finally, you will need to extract the `small_data.zip` file that can be found in the `Spring2016_IR_Project/data` directory

4.2 Preparing the Server

In order for some of the Machine Learning libraries in Spark to work, you need to ensure that you have `libgfortran3` installed.

On a Debian based system, you can run the following command:

```
$ sudo apt-get install libgfortran3
```

Next install Anaconda which is a completely free Python distribution. It includes more than 400 of the most popular Python packages for science, math, engineering, and data analysis. For complete instructions on how to download and install Anaconda go to:

`https://www.continuum.io/downloads`

Anaconda is shipped with IPython notebook, a powerful interactive shell for Python.

To set up Spark and IPython to run together and provide a nice web interface, there are a number of things that need to be done.

Begin by creating an IPython profile for PySpark. This can be done using the command:

```
$ ipython profile create pyspark
```

If you are running other IPython profiles, you may want to change the default port. This can be done by modifying ~/.ipython/profile_pyspark/ipython_notebook_config.py.

You can then change the default port within that file.

```
c = get_config()
# Simply find this line and change the port value
c.NotebookApp.port = <your port number>
```

Next the following lines need to be placed in .bashrc.

```
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
export PYTHONPATH=$SPARK_HOME/python/lib/py4j-0.8.2.1-src.zip:$PYTHONPATH
```

After this the .bashrc file should be reloaded by relogging in, or sourcing the .bashrc file.

To run the IPython server you should run the following command from within the cloned git repository, replacing <IP> with the IP address of your server.

```
ipython notebook --profile=pyspark --no-browser --ip <IP>
```

You can then navigate to <IP>:<PORT> in your browser to get access to the IPython notebook. By default the port will be 8888.

4.3 Using the IPython Notebook

Once the IPython notebook has been opened in the browser, open the Tweets_Classification-V2.ipynb IPython notebook. You should see something similar to Figure 4.1.

In order to use the script, you will need to modify the base_dir variable in cell 10. Set this to point at the data directory.

4.3.1 Using the Configuration File

To go further in using the script, you may need to make modifications to the collections_config.json file under data; see Figure 4.3. The provided configuration file is already configured to work with the six small datasets that were assigned to the groups at the start of the semester. However, the frequent patterns (FP) field has not been properly configured. This will need to be done for each collection as you work through the tutorial.

To classify your data, you need to specify the table ID of your collection in the collections_config.json configuration file. You should model your entry off of one of the sample entries in the file, or modify a sample directly. It is also suggested that you update the name field to the name of your collection. At the moment, don't worry about the FP values.

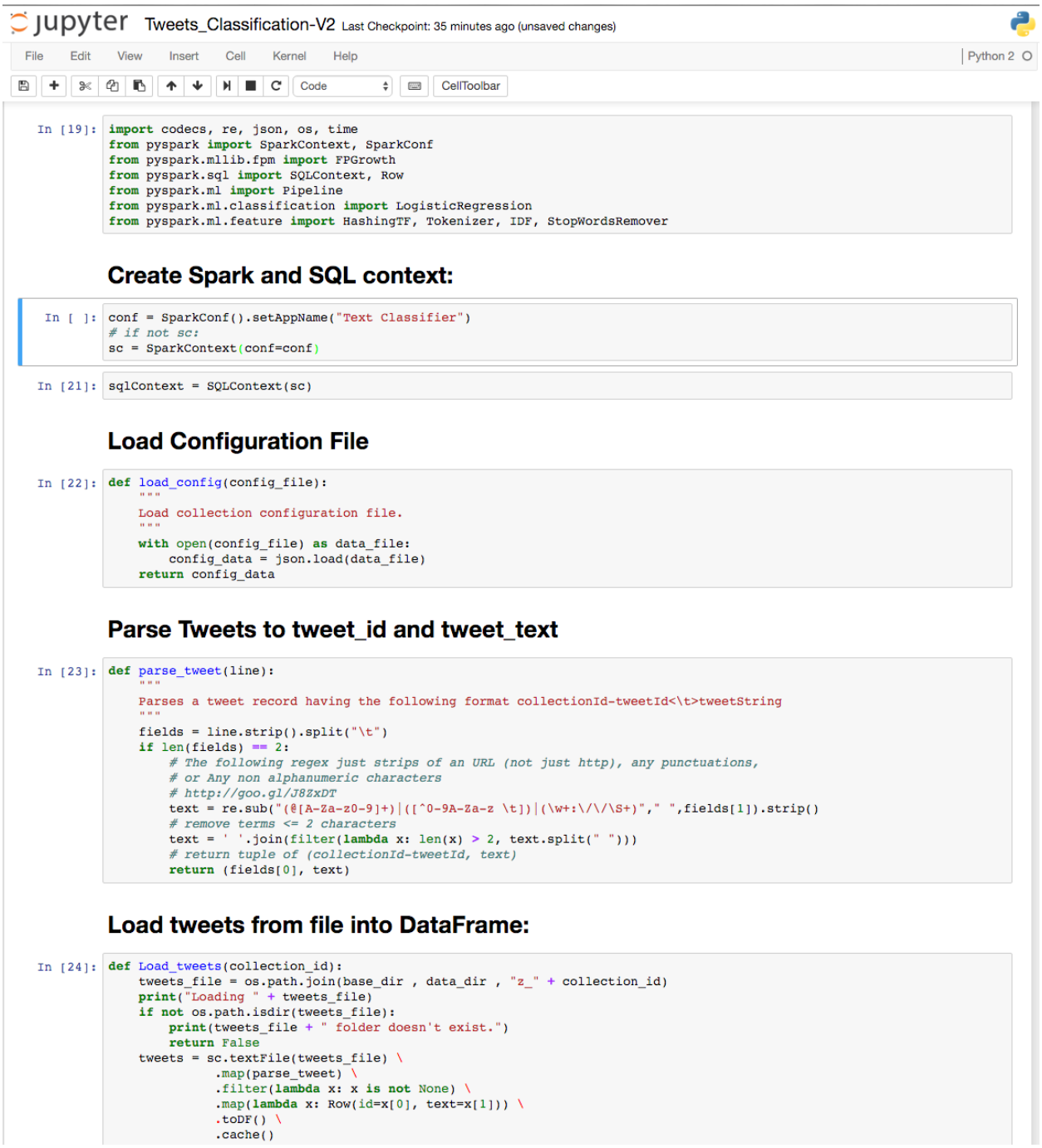


Figure 4.1: IPython notebook

4.3.2 Selecting a Frequent Pattern

After setting the configuration file, begin at the first cell and press Shift + Enter to execute each cell in order. Continue this until you reach "Manually choose frequent patterns and write them in the configuration file."

You will now need to open each of the frequent pattern output files located at:
`data/FPGrowth/«timestamp»_«collectionId»_«collection_name».txt`

```
{
  "collections": [{
    "Id": "602",
    "name": "Germanwings",
    "type": "binary",
    "FP": ["african", "lives"]
  }, {
    "Id": "541",
    "name": "NAACPBombing",
    "type": "binary",
    "FP": ["bombing", "colorado"]
  }, {
    "Id": "668",
    "name": "houstonflood",
    "type": "binary",
    "FP": ["cambodia", "insurance"]
  }, {
    "Id": "700",
    "name": "wdbj7 shooting ",
    "type": "binary",
    "FP": ["unbelievable", "roanoke", "gun"]
  }, {
    "Id": "686",
    "name": "Obamacare",
    "type": "binary",
    "FP": ["insurance", "health"]
  }, {
    "Id": "694",
    "name": "4thofJuly",
    "type": "binary",
    "FP": ["independenceday", "happy"]
  }]
}
```

Figure 4.2: Collections configuration file

You should now inspect the file for the tokens frequently found together. Look for high frequency patterns. Choose a pattern that seems highly relevant to your collection. We suggest a pattern of two to four words to strike a balance between over and under specification. An example of what this file might look like is shown in Figure 4.3. The file is formatted as: the number of times a pattern occurred across the collection, and then the pattern of words themselves. Keep in mind that this is a tokenized and unordered list for the pattern, FPM only looks to see if those words appear within the tweet text, order is not considered.

Take the pattern and copy it as the value of "FP" in the configuration file.

```
5664 african stood lives charliehebdo world
5664 african stood lives charliehebdo germanwings
5664 african stood lives charliehebdo
5664 african stood lives
5664 african stood germanwings
5664 african stood charliehebdo world germanwings
5664 african stood charliehebdo world
5664 african stood charliehebdo germanwings
5664 african stood charliehebdo
5664 african stood
5664 african matter lives germanwings
5664 african matter lives
5664 african matter germanwings
5664 african matter
5664 african lives world germanwings
5664 african lives world
5664 african lives charliehebdo world germanwings
5664 african lives charliehebdo world
5664 african lives charliehebdo germanwings
5664 african lives charliehebdo
5664 african charliehebdo world germanwings
5664 african charliehebdo world
5664 african charliehebdo germanwings
```

Figure 4.3: Frequent Patterns Mining algorithm output file for Germanwings collection

4.3.3 Building a Training Set

After the frequent pattern has been selected and input into the configuration file, continue to step through the script using Shift + Enter in each cell.

This will take you through the process of building a training set by identifying a positive sample set and a negative sample set and placing those into a DataFrame.

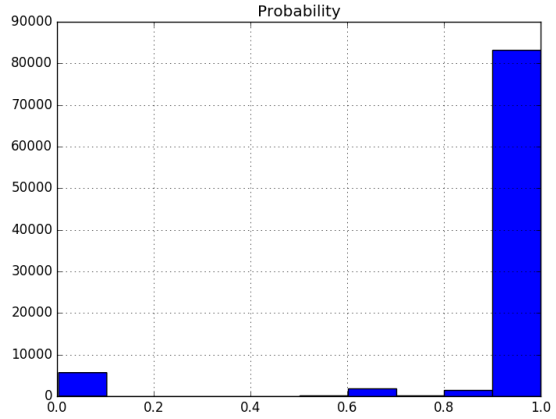
4.3.4 Training a Classifier

Further progression through the script will actually use the training data constructed in the last step to train a logistic regression classifier.

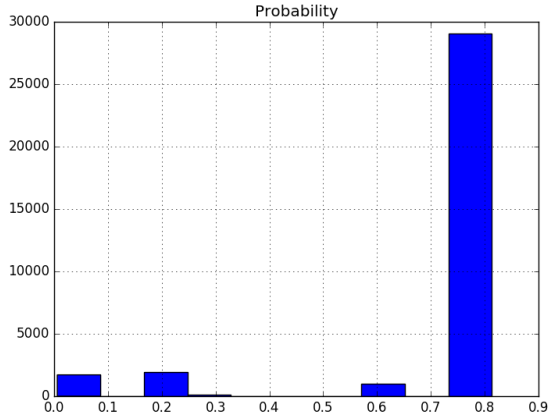
4.3.5 Running the Classifier

Finally, apply the classification model to our test data and provide a floating point prediction between relevance (1.0) or non-relevance (0.0) for each tweet in the test data. This data can be found in the `predictions` directory and has the same name formatting as the frequent pattern output files.

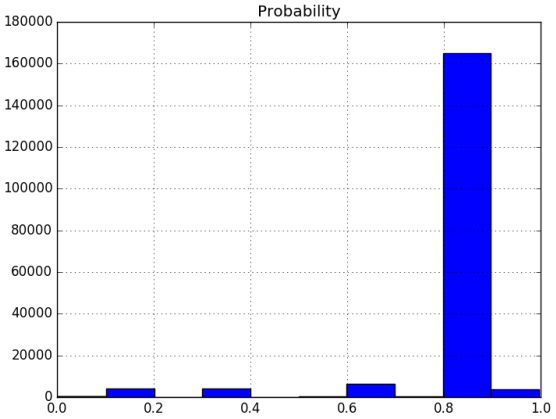
When running this classifier and getting an output that is a probability, it helps to know what the data looks like. This helps us to decide where is a reasonable cut-off point for deciding relevance vs non-relevance. This also allows us to judge how effectively a particular collection captures the intended event. We have created histograms showing the distributions for each of the small collections. These histograms are shown in Figure 4.4



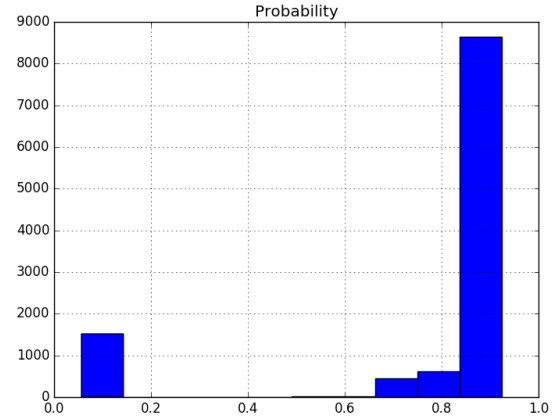
(a) Germanwings



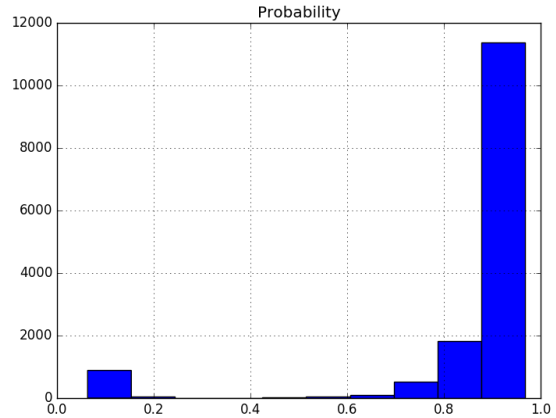
(b) NAACPBombing



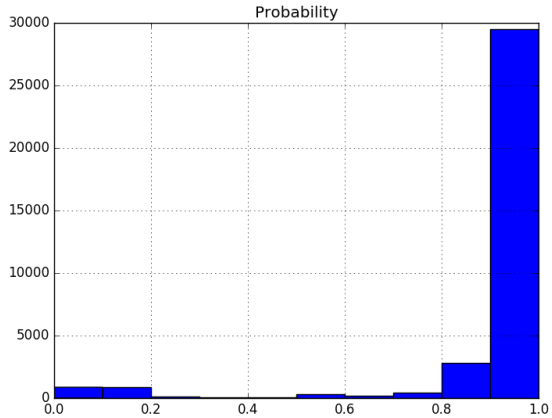
(c) Obamacare



(d) wdbj7shooting



(e) houstonflood



(f) 4thofJuly

Figure 4.4: Histograms for the small collections of tweets

Chapter 5

Developer Manual

5.1 Algorithms

5.1.1 Frequent Pattern Mining

Our methodology for constructing training data is based on a frequent pattern mining algorithm [3]. Spark.mllib version 1.5 provides a parallel implementation of FP-growth, a popular algorithm for mining frequent itemsets.

5.1.2 Logistic Regression

Logistic regression is a popular method to predict a binary response. It is a special case of Generalized Linear models that predicts the probability of the outcome. A logistic regression model is constructed once with training data and can then be applied to documents very efficiently, which makes it scale well. There are several parameters to experiment with which are documented in the spark.mlib documentation. Assuming the frequent patterns are constructed well, it can be applied to any collection. For more background and more details about the implementation, refer to the documentation of the logistic regression in spark.mllib.

5.1.3 Dependencies

Java

You will want the latest version of Java, which at the time of this writing is Java Version 8 Update 73. Download and installation instructions for your environment can be found at:

https://java.com/en/download/

The Hadoop cluster we are working with is currently running Java Version 7 Update 99. This version of Java is capable of performing everything that we require for this work. However, due to security concerns among other things we recommend using the latest stable version of Java if setting up a cluster from scratch.

Python

You will need Python 2.6.6 (our work should be compatible with newer versions of the 2.x.x line, but we have not tested it). Download and installation instructions for your environment can be found at:

https://www.python.org/downloads/

While the cluster that we have implemented our solution on is using Python 2.6.6, we recommend using the latest stable version of the Python 2.x.x line if setting up a cluster from scratch.

5.1.4 Apache Spark

This system is built on Apache Spark 1.5.0, which can be downloaded at:

http://spark.apache.org/downloads.html

Any newer versions of Spark should also be compatible for the foreseeable future. So if setting up a cluster from scratch, we recommend installing the latest stable version of Spark.

Chapter 6

Plan

Our work on this project has happened in two phases. In the first phase, we attempted to create an initial prototype working with a small set of sample data. This phase was accomplished mainly in our own external environment so that we can experiment more freely with the technologies to gain a better understanding of how they work and interact together.

Phase two of the project involved taking the work done in phase one and expanding it out to the other small sets of data provided to us. This phase also includes a definition and implementation of an evaluation approach. Phase one is small enough that we felt a subjective evaluation was sufficient, but as we expanded it became essential to have a more concrete, quantitative measure of effectiveness.

Please see Table 6.1 for a rough weekly breakdown of work accomplished.

Table 6.1: Weekly breakdown of work to be done.

Weeks	End Date	Tasks
Week 1	22 Jan.	Understanding the classification task
Week 2	29 Jan.	• Understanding the classification task • Read about Hadoop streaming using Python
Week 3	5 Feb.	Start online tutorials about Hadoop and Apache Spark. [6][7]
Week 4	12 Feb.	Continue online tutorials about Hadoop and Apache Spark. [4]
Week 5	19 Feb.	**Phase 1 will include only tweets small data set:** • Understanding the classification task • Read about Hadoop streaming using Python
Week 6	26 Feb.	• Prepare training data using FPM • Build classifier using Apache Spark
Week 7	4 March	• Build classifier using Apache Spark • Prepare HBase Schema for data storage
Week 8	11 March	Optimize classifier performance

Weeks	End Date	Tasks
Week 9	18 March	**Phase 2 will include tweets and webpages:** • Run our methodology to classify the tweets on the cluster. • Apply the Frequent Pattern methodology on the cleaned webpages provided by Collection Management team. • Develop HBase interface through which the classifier prediction output will be saved on HBase.
Week 10	25 March	Design an evaluation approach to test and evaluate our methodology.
Week 11	1 April	Discuss with GRAs and Solr team to finalize HBase schema.
Week 12	8 April	Move functioning prototype over to newly updated Hadoop cluster. Train multiple classifiers and pick the best model per collection.
Week 13	15 April	More research on hyper-parameter optimization. Check the feasibility of integrating hyper-parameters optimization library [1] output with Spark.
Week 14	22 April	Search for known approaches to select the most representative data samples for each collection. Then check whether training a classifier using these data samples will enhance the performance or not.
Week 15	29 April	Final evaluations and modifications to the system.

Chapter 7

Conclusion

We are very satisfied with the final state of our project. We have succesfully developed an end-to-end system that can read data from HBase, classify all of the data, and write the results back. Our evaluation has shown that the accuracy of the system could be improved slightly, but is largely in line with the manual classification results we received from the class. Some additional structured evaluation would be able to provide more insight into the differences between our processed data and the manually classified data. Our system is also designed with future work in mind. The design of our system will make it easy to process other small collections, and eventually expand out to larger collections.

Chapter 8

Future Work

There are a few points with our work that need to be accomplished soon to provide the other groups working with this project some useful information. First and foremost, we are in a position to provide the other groups information regarding the precision of the small collections. This will be useful for them to know if they should be working with the entirety of the collections or only a subset that we deem as related to the small collection and the event it is supposed to encapsulate.

Further reaching future work would be combining multiple classifiers, trained on different aspects of the same training set, into a single classifier that may provide a more robust prediction than any of the single classifiers on their own.

Another aspect to look at would be a close collaboration with the work done by the Clustering team. It would be interesting to compare how well the current method of training a classifier based on frequent patterns would compare to training a classifier based on using the words that they used for their clustering within a collection.

Looking even further out, our methodology needs to be evaluated on how it might handle collection growth. These collections will ideally grow incrementally over time so there will need to be a way to apply a classifier incrementally to the new data, but also potentially have this classifier receive feedback that might modify it over time as things such as slang terms and the English language in general change. We do not believe that implementing this is within the scope of our current project, but believe it would be a very interesting extension on the work in the future.

Acknowledgements

We would like to acknowledge and thank the following for assisting and supporting us throughout this project.

- NSF grant IIS - 1319578, III: Small: Integrated Digital Event Archiving and Library (IDEAL)
- Dr. Edward Fox
- Digital Library Research Laboratory Graduate Research Assistants
  - Sunshin Lee
  - Mohamed Magdy Farag
- Other teams in CS 5604
- Virginia Tech IT Security Lab

Bibliography

[1] Dan Bergstra James; Yamins and David D. Cox. “Hyperopt: A Python library for optimizing the hyper-parameters of machine learning algorithms”. In: *Proceedings of the 12th Python in Science Conference*. 2013, pp. 13–20.

[2] Xuewen Cui, Rongrong Tao, and Ruide Zhang. *Classification Team Project for IDEAL in CS5604, Spring 2015*. http://vtechworks.lib.vt.edu/bitstream/handle/10919/52253/ReportClassify.pdf. 2015.

[3] Jiawei Han, Jian Pei, and Yiwen Yin. “Mining frequent patterns without candidate generation”. In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 1–12.

[4] *Learn HBase*. https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/. April 2016.

[5] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval*. Vol. 1. 1. Cambridge University Press Cambridge, 2008.

[6] *Spark Examples for Python*. https://github.com/apache/spark/tree/master/examples/src/main/python. February 2016.

[7] *Tutorial by Solr Team*. https://canvas.vt.edu/courses/21271/files/folder/2016/Tutorials?preview=691471.