Predict Closed Questions on Stack Overflow with Naive Bayes Classification

Implemented in Hadoop MapReduce, Mahout, and Spark

Team members: Min Xu, Yan Long & Xiaoguang Mo

Introduction

The goal of this project is to practice solving real world large-scale data processing problems with machine learning algorithms and popular data processing frameworks and tools.

The problem we choose is to predict which new questions asked on Stack Overflow will be closed.

Millions of programmers use Stack Overflow to get high quality answers to their programming questions every day. With more than six thousand new questions asked on Stack Overflow every weekday, a software solution to predict closed questions is desirable.

About *6%* of all new questions on Stack Overflow end up "closed". Questions can be closed as off topic, not constructive, not a real question, or too localized. More in depth descriptions of each reason can be found in the Stack Overflow FAQ.

Our goal is to build classifiers that predict whether or not a question will be closed given the question as submitted, along with the reason that the question was closed.

We choose **Naive Bayes Classification** algorithm to solve the problem, and implement it using 3 of the most popular data processing frameworks, **Apache Hadoop MapReduce**, **Apache Mahout**, and **Apache Spark**.

Design

Naive Bayes Classifiers are linear classifiers that are known to be simple yet efficient. Its basic idea is based on Bayes' theorem, and "*naive*" means it assumes all the features are mutually independent, which is usually not true in real world.

For text classification, the input to the classifier is a document d, and a fixed set of classes (or categories) $C = \{c_1, c_2, ..., c_j\}$. In our case, $C = \{open, closed\}$, or if we consider the actual reason that a question was closed, $C = \{open, not\text{-}a\text{-}real\text{-}question, not\text{-}constructive, off\text{-}topic, too\text{-}localized\}$. The output of the classifier is simply a predicted class, $c \in C$.

To build a classifier, the input is a set of hand-labeled documents $\{(d_1, c_1), (d_2, c_2), \ldots, (d_m, c_m)\}$, which is called the training dataset given a fixed set of classes $C = \{c_1, c_2, ..., c_j\}$. The output is a learned classifier $\gamma: d \rightarrow c$.

We use the **bag of words** representation to handle text in documents, i.e. the post title and body. The class with maximum probability is

$$C_{max} = \text{argmax}_{c \in C} P(w_1, w_2, ..., w_n \mid c) P(c) = \text{argmax}_{c \in C} P(c) \prod_{w_i \in V} P(w_i|c)$$

Where

$$P(c_j) = \frac{doccount(C=c_j)}{N_{document}}, \quad P(w_i|c_j) = \frac{count(w_i, c_j)}{\sum_{w \in V} count(w, c_j)}$$

If a given class and feature value never occur together in the training data, then the frequency-based probability estimate will be zero. We use **Laplace smoothing** to handle such cases.

$$P(w_i|c_j) = \frac{count(w_i, c_j) + 1}{\sum_{w \in V} count(w, c_j) + |V|}$$

In practice, multiplying lost of probabilities can result in **floating-point underflow**. We can take logarithm of the probabilities to convert multiplications to additions.

$$\log C_j = \text{argmax}_{c \in C} \log P(c) + \prod_{w_i \in V} \log P(w_i|c_j)$$

For the discrete numeric features such as reputation and number of undeleted answers of the post owner, we simply use binning to discretize the feature values.

Implementations

Hadoop Environment

We use the Hadoop environment at the Santa Clara University Engineering Design Center. The cluster runs 64-bit CentOS 6.6 Linux distribution, Java Runtime version 1.7.0, and is configured with the Cloudera CDH-5.2 system which contains Hadoop 2.6.0 with YARN (MapReduce v2), Mahout 0.9, and Spark 1.1.0.

There are 24 worker nodes with quad-core CPUs, with 96 CPU cores in total, and supports up to 192 hardware threads (2 hyperthreads per core). The total RAM is 768GB. Raw HDFS storage is 261TB, and usable HDFS Storage is 80TB (with a replication factor of 3).

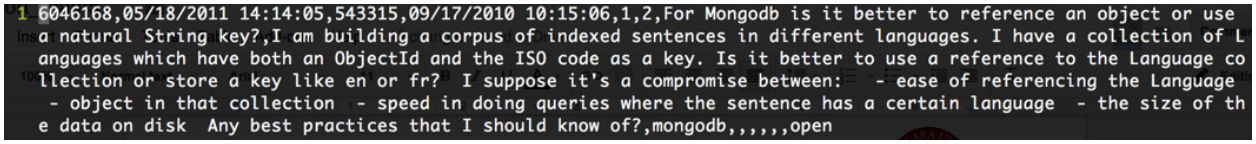
Data Preprocessing

We use the data from Kaggle. There are 15 columns of the raw data.

- Input: 12 columns
  `PostCreationDate, OwnerUserId, OwnerCreationDate, ReputationAtPostCreation, OwnerUndeletedAnswerCountAtPostTime, Title, BodyMarkdown, Tag1, Tag2, Tag3, Tag4, Tag5`
- Output
  `OpenStatus`
- Additional Data: 2 columns
  `PostId, PostClosedDate`

A sample screenshot for one single raw record.

The biggest problem with the raw data is the “BodyMarkDown” column. This column including a lot of symbols . Such as ‘\t’, ’\n’, ‘\r’, ‘,’ etc. Those symbols will conflict with our standard column separator(comma)and standard row separator(‘\n’). In order to make our data looks like a standard CSV format for further use. We took some time to write a data preprocessing program to replace all these in this column. Please refer the code in our code folder parse_line.cpp.
After pre-processing, the data look like this.



The steps for data-processing:

```
cat train_October_9_2012.csv | tr -d '\r' > train-sample-no-enter.csv
g++ parseline_new.cpp -O2
./a.out < train-sample-no-enter.csv > output.txt
```

Implementation in Apache Hadoop MapReduce

Hadoop MapReduce is a general purpose data processing framework and only provides low level API. To build the Naive Bayes Classifier in MapReduce, we implemented everything from scratch, including classification model building, tokenization, stop word removal, Laplace smoothing and feature weighting. The program does not depend on any external libraries, except for the Java standard library and Hadoop. The Java program contains 800+ lines of Java code and Bash scripts.

In the MapReduce implementation, the following features are used for building our classifier: `Title`, `BodyMarkdown`, `Tags (Tag1 ... Tag5)`, `ReputationAtPostCreation`, `OwnerUndeletedAnswerCountAtPostTime`.

Tokenization Considerations

To tokenize text in `BodyMarkdown` and `Title`, we experimented with several different options. The most straightforward way is to only split the text using white spaces without any further processing of the tokens. It ends up with a much larger model (with a much larger vocabulary) due to all the different punctuations and special characters within the tokens. The precision is relatively higher, however the recall is unacceptably low (even less than 5%), which is undesirable.

Stop words in text tend to carry less information and is generally useless. To make it simple, we use a list of 319 English stop words to filter the tokens in `BodyMarkdown` and `Title`. Due to the extra complexity and time limitation, stop words in other languages and segmentation of German and some Asian languages (Chinese, Japanese and Korean) are not considered.

Further normalizations such as converting all letters to lowercase turns out to have negative impact on precision and recall. Stemming and lemmatization needs much more work while are expected to have the same negative impacts, so we did not go further toward that direction.

Using N-grams seems to be a promising improvement. We tried tokenization with ***bigrams***, which ends up with a ***10 times larger vocabulary*** ($52,955,879$ unique bigrams vs $5,317,949$ unique unigrams). To make the resulting predictor fit into memory, we lowered the training set size. Experiment shows a precision drop of 20% compared to unigrams and the recall is less than 5%.

We end up doing tokenization using unigrams, splitting using all kinds of punctuations and removing stop words, which gains a better balance between precision and recall.

Feature Weighting

It is noted that some features are of more importance than others. E.g. the title is usually a good summary the post body, and tags categorizes the topic of a post even better. We tried to assign higher weights for `Title` and `Tags`, and gained higher precisions as a result. In particular, assigned a weight 8 to `Tags`, and the precision is improved by nearly *15%*.

We also tried using `(PostCreationDate - OwnerCreationDate)` as a feature, based on the observation that the longer a user was registered on Stack Overflow, the less likely his/her question will be closed, because the user is more likely to have more experience and is more familiar with the rules of asking relevant questions on Stack Overflow. Unfortunately experiment shows this feature does not really help.

Building the Model

Hadoop MapReduce does not support reading CSV files with multirow records. The input data needs to be preprocessed by removing newlines and commas within post body, and removing both escaped and unescaped quotes. This is done by the `CSVPreprocessor`.

The implementation consists of 2 programs, `PostMapReduce` and `PostPredictor`.

`PostMapReduce` is a MapReduce which reads the training data set and builds the Naive Bayes model by calculating all the parameters in the model.

`PostMapper` reads training records, tokenizes the body / title contents, removes punctuations and stop words, and counts number of occurrences of each feature, in open / closed / all posts respectively. Let's look at some examples.

For a word "stack" appeared 3 times in the body or title of a post, let's suppose the post status is "closed", `PostMapper` outputs the following 2 key-value pairs:

Key	Value
stack	3
stack/closed	3

For an "open" post with a tag "machine-learning", `PostMapper` outputs the following 2 key-value pairs:

Key	Value
/TAG/machine-learning	1
/TAG/machine-learning/open	1

For a "closed" post whose owner has a reputation of 120, `PostMapper` outputs the following 2 key-value pairs (we limit the reputation number to [-10, 1000], and use a bin size of 1):

Key	Value
/REPUTATION/120	1
/REPUTATION/120/closed	1

`PostReducer` aggregates and outputs the total values for each key. It also counts the number of unique words in post body and title (i.e. the vocabulary size), and number of unique tags through MapReduce counters.

`PostPredictor` applies the classifier. It reads output of `PostMapReduce` and loads all the key-value pairs into a `HashMap<String, Long>`. For each post it parses, `PostPredictor` goes through the selected features and calculates the probability for each post status ("open" and "closed") respectively, and selects the one with the maximum probability as the predicted post status. `PostPredictor` eventually outputs all the statistics, number of posts that are predicted / actually closed, precision, recall, F-measure, and accuracy.

Implementation in Apache Mahout

In Apache Mahout, we used the following features: `Title, BodyMarkDown, Tags(Tag1 … Tag5)`, and additional information: `PostId`.

Step 1. Convert CSV file into hadoop sequence file (Key-Value pairs).

Sequence file is a list of key-value pairs, and key in our context here is the category+PostId where the Category is label, i.e. open or closed (not a real question, not constructive, off topic, too localized) to which the post question belongs to and PostId is the Post ID and value is the text portion of the question. Both key and value are text.

We wrote a Java program to do this convertion. Refer to the code `CSVtoSeqFile.java`.

In this code, we can choose our experiment as binary classification or multiple classification, and select the features we want. We tried several different combinations of the features.

```
java -jar BigDataNaiveBayes_fat.jar /home/ylong/output.txt /home/ylong/seq-stackoverflow/
```

Step 2. Put the sequence file to hadoop HDFS.

```
hadoop fs -put ~/seq-stackoverflow/chunk-0.txt ./seq-stackoverflow
```

Step 3. Calculate the word frequency in each label(open, close) using tfidf, and convert the sequence file into mahout-SparseVectors.

```
mahout seq2sparse -i ./seq-stackoverflow -o ./stackoverflow-vectors  -wt tfidf
```

This step in mahout is divided into 11 separate jobs. Including
Creating Term Frequency Vectors.
Creating dictionary from `stackoverflow-vectors/tokenized-documents` and saving at `stackoverflow-vectors/wordcount`.
Calculating IDF.
SparseVectorsFromSequenceFiles: Pruning.
There are a lot of jobs doing the temp-directory cleaning.
This is one of the reasons why Mahout is so slow in the platform comparison.

Step 4. Split the data into two parts. Using 60% as training data. 40% as test data.

```
mahout split -i ./stackoverflow-vectors/tfidf-vectors --trainingOutput
./stackoverflow-train-vectors --testOutput ./stackoverflow-test-vectors --randomSelectionPct
40 --overwrite --sequenceFiles -xm sequential
```

Step 5. Train model.

```
mahout trainnb -i ./stackoverflow-train-vectors -el -o ./model -li ./labelindex -ow -c
```

Step 6. Test model.

```
mahout testnb -i ./stackoverflow-test-vectors -m ./model -l ./labelindex -ow -o
./stackoverflow-testing -c
```

The run time of Mahout in our experiment is not that good. Because we are using the default setting. The program itself is not a perfect fit for our particular problem.

Implementation in Apache Spark

Spark has developed NaiveBayes package, and the machine learning data term TF-IDF is also needed to compute the weight of each keyword. Because in real machine learning model, the keyword with highest frequency is possibly not the most important word, like “code”, “new”, “temp”, they are very common in any programming language, which could be included in the post. So the TF-IDF is very important, it could set a low weight to both high frequency words and low frequency words. After the TF-IDF is generated, all training data could apply to TF-IDF to get overall weight of every word in one row. If some word is not included in TF-IDF, additive smoothing “lambda” will be used to handle this situation to avoid abnormal low value of the weight. There will be about 5 jobs in total, RDD will be spited into 30 parts and delivered to 60 slaves.

Results and Evaluation

Results of Apache Hadoop MapReduce

For Naive Bayes Classification, larger training data set is desirable. So we use 99% of the input data as training set and the remaining 1% as test set.

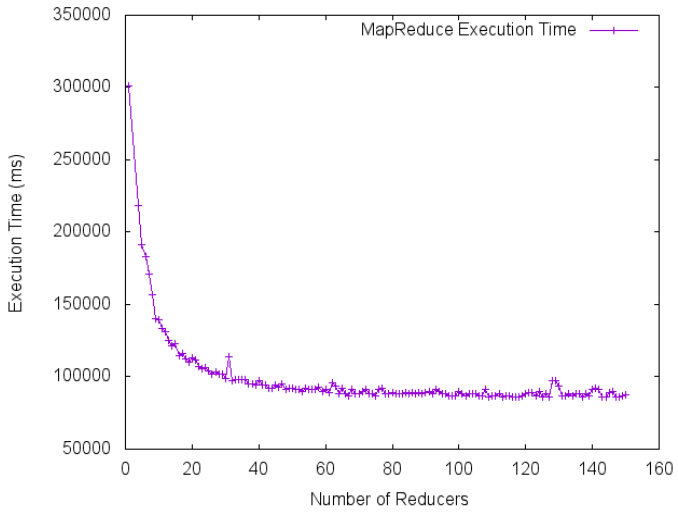
The following table shows some statistics of the training data set, and the classification performance -- Precision, Recall, F-measure, and Accuracy.

Training Dataset Size	3,628,208 posts / 4.0GB
Vocabulary Size	5,317,949 words
Unique Tags	44,061 tags
Precision	39.19%
Recall	24.00%
F-measure (F1)	29.77%
Accuracy	87.86%

Besides classification performance, we would also like to see impact of parallelism on runtime performance. The `PostMapReduce` starts 30 mappers and 96 reducers by default based on the input and output size. It finishes in about 90 seconds.

Hadoop MapReduce does not allow specifying number of mappers directly. The number of mappers is determined by the input size and the split block size (whose default value is the DFS block size). Instead of adjusting the split block size, we manually adjusted the number of reducers, ranging from 1 to 150, to test the impact of number of reducers on execution time.

The result is shown in the figure below. Please note that starting more than 60 reducers has little impact on execution time, because the workload is too small to be further divided and the overhead of starting a reducer dominates the execution time. Better speedup can be achieved by increasing the input size.



Results of Apache Mahout

1.Treat as 5 classes

The original data including 5 categories.

Open, and the rest four categories belong to Close. I tried both(Treat as two classes v.s. Treat as five classes)

```
=======================================================
Summary
-------------------------------------------------------
Correctly Classified Instances          :     593846         78.0874%
Incorrectly Classified Instances        :     166643         21.9126%
Total Classified Instances              :     760489

=======================================================
Confusion Matrix
-------------------------------------------------------
a       b       c       d       e       <--Classified as
```

```
553     1095    446     3697    284      |  6075       a     = not a real question
106     2878    312     1045    26       |  4367       b     = not constructive
112     1043    1230    1259    48       |  3692       c     = off topic
23655   59015   34037   589029  39239    |  744975     d     = open
93      100     80      951     156      |  1380       e     = too localized

=======================================================
Statistics
-------------------------------------------------------
Kappa                                       -0.0157
Accuracy                                    78.0874%
Reliability                                 33.1155%
Reliability (standard deviation)             0.3267
```

2. Treat as two classes

```
=======================================================
Summary
-------------------------------------------------------
Correctly Classified Instances          :     594266         78.2745%
Incorrectly Classified Instances        :     164942         21.7255%
Total Classified Instances              :     759208

=======================================================
Confusion Matrix
-------------------------------------------------------
a       b       <--Classified as
8259    6979     |  15238      a     = close
157963  586007   |  743970     b     = open

=======================================================
Statistics
-------------------------------------------------------
Kappa                                       -0.0169
Accuracy                                    78.2745%
Reliability                                 44.3225%
Reliability (standard deviation)             0.403

15/06/04 20:44:01 INFO driver.MahoutDriver: Program took 32602 ms (Minutes:
0.5433666666666667)
```

Results of Apache Spark

4 GB input file which included 3 million posts will be randomly split into 60% training data and 40% test data, and the overall accuracy is: 79.69045094934553%. It runs much faster than Hadoop and Mahout, just less than 40 seconds for the whole process.

Related Work

The Naive Bayes Classification algorithm used in this project, including bags of words model, feature weighting, Laplace smoothing, using logarithm to avoid underflow, are derived from various research papers, textbooks and wikipedia pages listed in the references below.

The Hadoop MapReduce implementation is completely developed by ourselves and does not depend on any external libraries except the Java standard libraries and Hadoop. It uses the English stop words list from http://www.ranks.nl/stopwords.

The Mahout and Spark implementations make use of Bayes library functions from the Mahout and Spark frameworks respectively.

Conclusion

We implemented Naive Bayes Classifiers to predict closed questions on Stack Overflow using 3 of the most popular data-processing frameworks, Apache Hadoop MapReduce, Mahout, and Spark.

All the 3 frameworks are suitable for developing data intensive applications. MapReduce is a general framework for any large-scale data processing tasks, while both Mahout and Spark provide higher level APIs for commonly used tasks such as fundamental machine-learning and data-mining algorithms.

Regarding **development efficiency**, the MapReduce program has to build everything from scratch using the general-purposed low level API, with ~800 lines of code. While the Mahout and Spark counterparts are much more concise by building on top of higher level APIs.

On the other hand, the MapReduce framework provides better **flexibility** and **scalability**. We are able to do quite a lot manual tuning of the classification model and experiment with new ideas in MapReduce. As a result, the MapReduce implementation gains the best precision, recall and accuracy. Also the MapReduce program is able to build the classification model within 90 seconds using all the 3.6M training records while the other 2 implementations are unable to scale to this level with the limited resources in our experiment environment.

From the experiment results, we can see that the performance of Naive Bayes Classifiers is not ideal even though we made quite a lot manual tunings of the tokenization methods, feature weights, etc. This is partly because of the apparently oversimplified assumptions made by all Naive Bayes classifiers, that the value of a particular feature is independent of the value of any other feature.

Also the problem itself is actually more difficult than those tasks such as classifying spam emails. One can recognize spam emails at a glance but classifying a technical question on

Stack Overflow is not so intuitive. It usually requires technical backgrounds and even in-depth understanding of the domain knowledge. More complex models are needed to better capture the structure and semantics of natural language. This is a good topic for further exploration.

References

[1] Naive Bayes Classifier, http://en.wikipedia.org/wiki/Naive_Bayes_classifier

[2] Raschka, Sebastian. "Naive Bayes and Text Classification I - Introduction and Theory." arXiv preprint arXiv:1410.5329 (2014).

[3] Predict Closed Questions on Stack Overflow, Competition on Kaggle, https://www.kaggle.com/c/predict-closed-questions-on-stack-overflow

[4] Data source, https://www.kaggle.com/c/predict-closed-questions-on-stack-overflow/data

[5] Stack Overflow FAQ on Closed Questions, http://stackoverflow.com/help/closed-questions

[6] Apache Mahout: Scalable machine learning and data mining, http://mahout.apache.org/

[7] Apache Spark: Lightning-Fast Cluster Computing, https://spark.apache.org/

[8] Hadoop Environment at SCU Engineering Design Center, http://wiki.helpme.engr.scu.edu/index.php/Hadoop

[9] Lee, Chang-Hwan, Fernando Gutierrez, and Dejing Dou. "Calculating feature weights in naive bayes with kullback-leibler measure." Data Mining (ICDM), 2011 IEEE 11th International Conference on. IEEE, 2011.

[10] Zaidi, Nayyar A., et al. "Alleviating naive Bayes attribute independence assumption by attribute weighting." The Journal of Machine Learning Research 14.1 (2013): 1947-1988.

[11] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. Vol. 1. Cambridge: Cambridge university press, 2008. Also available at http://www-nlp.stanford.edu/IR-book/

[12] Bag of Words Model, http://en.wikipedia.org/wiki/Bag-of-words_mode

[13] Additive Smoothing, https://en.wikipedia.org/wiki/Additive_smoothing

[14] English Stopwords, http://www.ranks.nl/stopwords

[15] Yang, Ying, and Geoffrey I. Webb. "A comparative study of discretization methods for naive-bayes classifiers." *Proceedings of PKAW*. Vol. 2002. 2002.

[16] Yang, Ying, and Geoffrey I. Webb. "Discretization for naive-Bayes learning: managing discretization bias and variance." *Machine learning* 74.1 (2009): 39-74.

[17] Fürnkranz, Johannes. "A study using n-gram features for text categorization." Austrian Research Institute for Artificial Intelligence 3.1998 (1998): 1-10.

[18] Confusion Matrix, http://en.wikipedia.org/wiki/Confusion_matrix

[19] Precision and Recall, https://en.wikipedia.org/wiki/Precision_and_recall

[20] F1 score, https://en.wikipedia.org/wiki/F1_score

[21] Mahout Bayes example, https://mahout.apache.org/users/classification/twenty-newsgroups.html

[22] Convert txt/CSV into sequence file, https://chimpler.wordpress.com/2013/03/13/using-the-mahout-naive-bayes-classifier-to-automatically-classify-twitter-messages/comment-page-2/#comment-1973

[23] CSV to Mahout-Vector Example, https://gist.github.com/Jossef/e6c8fc0c31f0c2bf036a

[24] Anil, Robin, Ted Dunning, and Ellen Friedman. Mahout in Action. Shelter Island: Manning, 2011.