IST-664 Final Project

Stack Exchange Channel Classification

Chris Wilson, Jeff Levesque

# Introduction

Stack Exchange is a network of websites dedicated to answering questions from Internet users. Stack Exchange started in 2008 as Stack Overflow, a forum for computer programming questions, as a free alternative to the paid subscription site Experts Exchange[[1]](#footnote-1). In 2009 the site expanded into other sites, modeled after the Stack Overflow template, to cover a diverse set of domains. These domains now include Technology, Culture/Recreation, Life/Arts, Science, Business and Professional[[2]](#footnote-2). In addition to each of these domains, the Stack Exchange network also hosts some language specific sites, such as Russian, Portuguese, Japanese, etcetera. Each of these domains is what is considered a “channel” in the Stack Exchange, thus a question should be directed to the channel which can best answer your domain specific question. The driving motivator for this research is to create a classifier that could automatically determine the best channel to post your question. This will be done using natural language processing fundamentals and machine learning.

# Data Preparation

Stack Exchange provides a historical dump of all the contents from the various channels to Archive.org[[3]](#footnote-3) in XML format. The 75 GB data set consists of 173 folders, one for each channel, and within each channel it is separated by type (i.e Posts, Comments, Users, Badges, etc.). For the purpose of classifying questions into channels we needed to be certain that a question did indeed belong on the channel it was posted to. Thankfully the data in each channel’s Post.xml file revealed the posted question id, and a verified answer id. These id numbers allowed us to create a script that would match the posted question to the verified answer. A script was created to read in the Posts.xml file from each channel and create JSON output consisting of the channel, post id, question, and verified answer (Appendix 1a).

Although the transformation reduced the data to 6 GB, it was determined that trying to classify 173 channels would be too much for the initial research into this subject. For this reason, the top 10 most popular channels were used for this study, these channels in order include: Math, Russian, LaTex, Super User, Server Fault, Code Review, Ask Ubuntu, Portuguese, Unix, and Physics. By utilizing these channels, we were able to reduce the number of classes while still retaining over half of the data (3 GB).

# NLP with NLTK

## Preprocessing and Tokenization

The first step in the NLP process is gathering a corpus and ensuring data quality. The JSON data created in the preparation step must be read into Python and examined for suitability. During this first step it was noted that 9 of the 1.3 million questions could not be deserialized from JSON to Pandas objects due to Unicode characters creating odd escape sequences. This was safely handled by discarding the offending questions, as it only accounted for .0007% of the data. Further investigation into the raw questions also revealed that they were HTML formatted, and thus a function was created to strip the HTML encoding, leaving only text.

Tokenization of the text was accomplished through a function utilizing NLTK word tokenizer and built in Python functions, a sample can be seen in Figure 1.1. This function first splits the sentence into tokens using the NLTK function, then lowercases the tokens and finally removes all tokens that consist of entirely punctuation. The output of this custom function can be seen in Figure 2.1 as applied to the first ten questions of the corpus.



Figure 1.1



Figure 2.1

This function was then used to create a bag of words, a list of all of the tokens contained in the corpus, and the NLTK function FreqDist was applied. Inspection of the top 50 words showed that many were stop-words and bogus characters, however this was ignored for the time being. The number of features to start with was selected on a hunch, at the top 2000 features, to assess the performance of our classifier without over-fitting or creating a computationally burdensome problem.

With the features selected, one can continue to ready the data for use with Naïve Bayes classifier provided by NLTK. This classifier, and I suspect all NLTK classifiers, expect a feature set that consists of a Python list of dictionaries, which contain the feature as a key and either the Boolean nature of the feature for that document, or a count of occurrences. This quickly because troublesome, as a Python dictionaries rely on strings as the key. A string in Python is a minimum of 40 bytes of information, which means that one document feature set is 2000 \* 40 bytes (roughly 80 kilobytes) for keys alone. The full corpus contained 1.3 million documents, which translates into 80 kilobytes \* 1,339,625 (102 GB) of RAM needed to store just the keys all the documents.

This forced the team to sample just 20% of the dataset to reduce memory consumption. Fortunately, Pandas comes with the ability to take a random sample of data without replacement. The distribution of documents, in percent of overall by channel, can be seen for the original data (Figure 3.1) and after sampling (Figure 4.1). The sampling reduced the corpus to roughly 267 thousand questions and retained the original distribution. Once the memory consumption problem had been solved, the team could move on to creating the feature set from the sampled corpus. The function in Figure 5.1 shows the creation of the feature set using a Boolean identifier as a first pass.



Figure 3.1



Figure 4.1



Figure 5.1

## Classification

The Naïve Bayes classifier from the NLTK package was selected for the initial experiment and was run with a traditional train/test split of 80/20%. This was done as a time saving measure, as the large amount of data used was prohibitive for NLTK. A five-fold cross validation was after initial run to . The metrics used to judge the performance of the model are: Precision, Recall and F1 score. These metrics should provide us with information in how our model performed at classifying the different classes.

Initial results were good, with the worst-case macro average of 74% precision, 76% recall, and 75% F1 score (Figure 6.1). The model performed very well in classifying the Math, Russian, Portuguese, and LaTex channels. The model suffered the most in making a distinction between Physics and Math, and all of the computer technology related channels. This is not all that surprising, given that the Russian and Portuguese channels had specific symbols and words that most likely never appear in the other eight channels. What was surprising was the high performance of the LaTex channel, as it is an intersection between Math and computer programming.



Figure 6.1

The confusion matrix in Figure 7.1 clearly shows that Math was predicted most often, as it was the largest fraction of the dataset and thus had the Apriori advantage in the Naïve Bayes method. One can also see the struggle for Physics versus Math, where almost half of the Physics questions were classified as Math. The distribution is less pronounced in the technology channels, where equal misclassification occurred across many different channels. A list of the top 20 most informative features of this model are shown in Figure 8.1, where you can clearly spot the Portuguese and Russian characters, along with the keyword “ubuntu”, and LaTex specific phrases like “\\usepackage”.

The cross validation using five-folds showed that our model did perform consistently across all folds, which signifies a low risk of over-fitting occurred. The results of this can be seen in Figure 9.1.

A screenshot of a cell phone

Description automatically generated

Figure 7.1



Figure 8.1



Figure 9.1

## Classification using Part of Speech

Part of speech tagging is a method in which a word is classified into a specific part of speech (e.g Noun, Pronoun, Adjective) by analyzing the words that are adjacent. The experiment performed is to pair the word and the part of speech predicted to form a feature. This feature is assigned a Boolean if the specific tag and word combination appear in the document. This is very similar to the method used in the first experiment, with the notable exception that a word must also be in the right context to be considered true.

To accomplish tagging a word by the part of speech, a function was created that lowercases the word, then uses NLTK’s pos\_tag function and returns a set of tokens in the format: ‘<POS>\_<word>’ (Figure 10.1). An example of the tokenizer in action is shown in Figure 11.1.



Figure 10.1



Figure 11.1

Like the unigram Boolean experiment, the tokenized words were put into a bag of words list, and the top 2000 features were used to create the feature sets. These feature sets were then used as input to a five-fold cross validated Naïve Bayes model.

The results of this model were extremely poor (Figure 12.1), showing precision and recall between 9-10% and F1 scores at 6%. The confusion matrix in Figure 13.1 shows that almost all questions were classified as Math. This leads the team to believe that the introduction of part of speech tags created too many unique features and led to information loss. This information loss would cause the model to fall back on using purely Apriori probability to classify the document. Given that the Math channel contains three times as many documents as the second largest channel, it makes sense that our algorithm would just use a majority baseline type prediction.



Figure 12.1

A screenshot of a social media post

Description automatically generated

Figure 13.1

# SciKit-Learn with NLTK Tokenizer

The next set of experiments leveraged SK-Learn pipelines with a custom tokenizer function. SK-Learn pipelines are a method of packaging the vectorization and machine learning steps of a model together. The most convenient feature of SK-Learn is the option to use a Pandas dataframe as input into the vectorizers. This reduces memory consumption by converting the Pandas series into a dense NumPy matrix during vectorization. In comparison to the NLTK feature set method, this method uses half of the RAM to generate the document term matrix.

## Tokenization and Vectorization

SK-Learn’s default vectorizer, count vectorizer, was selected for this study for the expansive customization and filtering options it provides. This vectorizer uses a custom tokenizer constructed of NLTK tokenizers and Python functions (Figure 14.1). The function tokenizes the document based on a regular expression, only saving words that have digits or non-whitespace characters (expect underscore) surrounded by word boundaries. The re-assembled string is tokenized again using a Treebank tokenizer to correctly handle contractions and plural words. Lastly, it strips out all tokens containing purely punctuation, and eliminates tokens under three characters in length. In addition to the custom tokenizer, all words were converted to lowercase, English stop words were removed, accents stripped out, and words that did not appear in at least .05% of the documents were discarded (Figure 15.1). The minimum document frequency was selected to reduce the total number of features and minimize over-fitting and computation time.



Figure 14.1



Figure 15.1

## Random Forest Classification

Random Forest classification algorithms are an ensemble model constructed from a “forest” of decision trees. The model creates many decision trees by sub-setting the features and returning the tree that performs the best overall. Decision tree and Random forest methods, although computationally expensive, are very popular due to the ability to handle factorial and numerical data at the same time. These methods also show competitive accuracy when compared to other machine learning algorithms, such as Naïve Bayes and SVM.

### Boolean Unigram

The random forest model performed slightly better than the Naïve Bayes model, with a cross validation mean accuracy of 81.2%. The worst-case macro averages for this model are precision of 79%, recall of 74%, and F1 score at 76% (Figure 16.1). This model improved precision scores over the Naïve Bayes model, however it suffers a bit in the recall. This disparity was small enough that it increased the overall F1 by 1%. Overall these models are comparable in predictive power and both suffer from the same misclassification errors of the technology channels. The most interesting thing about this model is the misclassification of code review as math, which might point to some inaccuracies in the decision tree.



Figure 16.1

A screenshot of a social media post

Description automatically generated

Figure 17.1

The top 20 features for prediction are like the Naïve Bayes model, with the exception that order is slightly different, and Russian characters appear more frequently (Figure 18.1).



Figure 18.1

### Word Frequency Unigram

To determine if word frequency affected the model the team ran the same pipeline again, apart from enabling the word frequency. The flexibility of SK-Learn pipelines really stood out in these comparison models, where a simple configuration change is all that is needed to enable word frequency. By comparison, to enable these features in the pure NLTK method would require the user to create a new function that counted the words as they appeared in the document. The count vectorizer used can be seen in Figure 19.1, when compared to Figure 15.1 it should be noted that the only change was to flip the binary flag to “False”.



Figure 19.1

The word frequency model performed almost identically to the Boolean model, with a reduction in accuracy of .01%. Precision, Recall and F1 were also similar (Figure 20.1), leading the team to believe that word frequency is of no importance to the random forest model in this case.



Figure 20.1

A screenshot of a social media post

Description automatically generated

Figure 21.1

### Boolean Bigram

Considering the near identical performance of the Unigram models, the team wanted to investigate how Bigrams affected the outcome of the model. Once again, enabling this feature was very straight forward by using a simple configuration to the vectorizer in the pipeline. The code in Figure 22.1 show that two changes were made to the original vectorizer, the n\_gram directive was added to limit the tokens to 2 words, and the minimum document frequency was removed. By using bigrams, the feature matrix would become sparser. To retain important features the vectorizer was not restricted to using only bigrams that appeared in a minimum percentage of the corpus.



Figure 22.1

The performance of the Bigram model was considerably lower than the Unigram models, with a mean accuracy of 73.5% in cross validation. The precision, recall, and F1 scores were also much lower with a worst-case macro average of 74%, 60% and 64% respectively. The per-class breakdown in Figure 23.1 shows that strong performers, like Russian, Portuguese, and LaTex took heavy losses in both precision and recall. This compounds with the high rate of misclassification present in the technology channels to lower overall model performance.



Figure 23.1

A screenshot of a social media post

Description automatically generated

Figure 24.1

By looking at the top 20 features for the bigram model (Figure 25.1), one can see that word combinations created features that were too specific to accurately predict channels. Many of the top features are very specific to a certain context, or programming language, such as: ‘begin document’, ‘real numbers’, and ‘magnetic field’. Bigrams also caused non-English language channels to have important features that are common language pairings, such as ‘как это’ which means ‘like this’.



Figure 25.1

# Conclusion

In this study it was shown that both Naïve Bayes and Random forest methods were accurate at classifying the appropriate channel to post a user question to. It was also demonstrated that one can combine the fine-grained control of tokenization provided by NLTK with the convenience and computational reductions of SciKit-Learn. The team also discovered that word frequency has no effect on the classification of questions, and that bigrams create a feature matrix that is too sparse and generic to perform well. This work can be used in conjunction with a digital assistant, or question answer system, to allow people to type a question and retrieve answers from the channel predicted most likely to have the answer.

# Appendix



Appendix 1a

1. https://en.wikipedia.org/wiki/Stack\_Exchange [↑](#footnote-ref-1)
2. https://stackexchange.com/sites# [↑](#footnote-ref-2)
3. https://archive.org/details/stackexchange [↑](#footnote-ref-3)