Text Classification

COMP 551: Applied Machine Learning
Kaggle Team: Chris-HP

Christopher Glasz
christopher.glasz@mail.mcgill.ca
260720944

Truong Hoai Phuoc
phuoc.truong2@mail.mcgill.ca
123456789

I. Introduction

Blah blah blah

II. Related work

III. Problem Representation

We chose to represent the text data as vectors of TF-IDF scores. We initially implemented our own code to calculate these feature vectors, but eventually turned to the Scikit-learn [3] `CountVectorizer` and `TfidfTransformer` functions to produce them for us, to speed up the calculation time.

At the outset, we had planned on using the 2,000 most common unigrams and the 200 most common bigrams as features, but this proved to be ineffective. We improved our classification accuracy (at the cost of feature preprocessing time) by increasing to 90,000 unigram and 10,000 bigram TF-IDF scores, for a total of 100K features. Along the way we tried 22K and 55K features (both with a 10/1 unigram-bigram ratio), and found that the performance of our models just continued to increase as we added more features, so we settled on a nice even 100K, as that's just about how many unique terms exist in the training corpus.

Our performance was further improved by lemmatizing the input tokens using NLTK's `WordNetLemmatizer` [2]. This prevented terms carrying the same information (e.g. "authenticate" and "authenticated") from being treated as two separate features. By tagging words by their part of speech before passing them to the lemmatizer, we were able to further improve our features (by reducing terms like "is", "are", and "am" to their common meaning of "be"). This process greatly increased the computation time, but was well worth it, as we only needed to calculate the feature vectors once.

Another decision we made over the course of the project was from where to extract the features. Early on, we were very strict about not using the text in the test corpus to select features, as we did not want to inadvertently influence the training of the model with information from set-aside data. However, as we increased the number of features, the difference between the set of most common words in the training corpus and in the combined training and testing corpora became negligible. Eventually we decided to produce 4 datasets:

1) *X_trn*: 80% of the training examples, with features drawn only from that corpus
2) *X_val*: The remaining 20% of the training examples, with features drawn only from that corpus
3) *X_all*: All examples in the training corpus, with features drawn from the words in both corpora
4) *X_tst*: All examples in the test corpus, with features drawn from words in both corpora

X_trn was used to find optimal hyper-parameters for our models, and used to train before prediction on the validation set. *X_val* was our validation set, and was only used to calculate our validation accuracy. *X_all* was used for final training before predicting labels on the test set, and was also the set over which we performed cross-validation. *X_tst* was, as one would expect, the dataset used for predicting labels on the test set.

All of these sets are saved as SciPy [1] compressed sparse row matrices to save space. They can be read by using the `load_sparse_csr` function in `utils.py`.

IV. Algorithm Selection and Implementation

A. Linear

We decided to implement Naïve Bayes for our linear classifier. We suspected that assuming a multinomial distribution would produce the best results (as that is the distribution of a unigram bag-of-words model), but we implemented both Multinomial and Bernoulli Naïve Bayes, since they are very similar in terms of implementation. The multinomial version did, as expected, perform marginally better, and is the model we used to produce our final predictions.

Our Naïve Bayes implementation went through an extensive evolutionary process, and the final product is extremely efficient. The first several iterations of the code used iterative methods to compute feature and class probabilities, and training took several minutes, sometimes as long as an hour. However, once we made the move to using sparse matrices, the computation time was cut down significantly. This also allowed us to expand our feature set from 22,000 to 100,000 features without worrying about running out of memory.

Our code was further optimized by vectorizing nearly all of the calculations. Early iterations of our implementation had us manually splitting the dataset by class and calculating probabilities individually for each feature and each class. This

is an incredibly inefficient way to work. We optimized by converting our output vector of labels to a one-hot vector encoding using Scikit-learn's `LabelBinarizer` [3]. This allowed us to split the set using a simple matrix multiplication, and calculating feature and class probabilities was boiled down to a couple summations down the correct axes. Coupled with the use of sparse matrices, this reduced our training computation time from the order of hours to seconds.

In addition to these optimizations, we used additional techniques to boost our accuracy.

B. Nonlinear

C. Optional

V. Testing and Validation

A. Linear

B. Nonlinear

C. Optional

VI. Discussion

VII. Statement of Contributions

A. Christopher Glasz

I implemented the code for feature selection, as well as the linear model (Naïve Bayes) and the optional method (SVM). I also produced the code for semi-supervised learning, and the sections of the report related to this work.

B. Truong Hoai Phuoc

References

[1] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2016-10-14]. 2001–. URL: http://www.scipy.org/.

[2] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*. ETMTNLP '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 63–70. DOI: 10.3115/1118108.1118117. URL: http://dx.doi.org/10.3115/1118108.1118117.

[3] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

VIII. Appendix