Text Classification

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

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

Hoai Phuoc Truong
phuoc.truong2@mail.mcgill.ca
260526454

I. Introduction

The goal of the project is to classify abstracts from English scientific articles into 1 of 4 possible categories: statistics, math, physics and computer science using only the text in the abstracts. There were 88637 entries provided for training/validation, and 15645 entries in the test set. Our approach was to extract the text features from the abstract using Natural Language Toolkit [4] together with Scikit-Learn library [6], and then either use all or a manually selected subset of these features to train several classifiers (linear and nonlinear).

II. Related work

There has been many discussions of successful use of Naïve Bayes for text classification such as multi-variate Bernoulli model by Larkey and Croft (1996) [3], or multinomial model by Mitchell (1997) [5]. In addition, there has also been successful application of more complex models such as Support Vector Machines (SVM) in text classifications (e.g. Joachims (1998) [1]). In this project we aim to explore these approaches (linear and nonlinear) on classification of scientific paper abstracts.

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 `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`. 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 [2] 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`. 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 the above optimizations, we further improved our performance by expanding our training set with semi-supervised learning. After training a model on the labeled training corpus, we predicted the labels for the test set. We chose the labels that the model was most sure of (the ones with the highest probability), and added them to the training set before training again. We repeated this process several times, until there weren't any examples in the test set that the model was sufficiently sure of (this was typically less than a few thousand examples).

We also experimented with ensemble methods using Scikit-learn's `BaggingClassifier` to improve our performance even further. We managed to reach the top of the Kaggle leaderboard by pulling out all the stops and using semi-supervised learning to train an ensemble of 100 bagged Naïve Bayes classifiers. This was our most successful model by far.

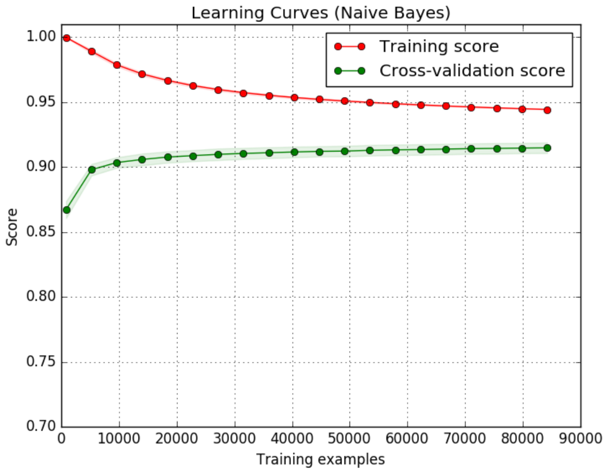

Figure 1: The learning curve for standard Multinomial Naïve Bayes

Table I: Hyper-parameter grid search for Naïve Bayes using 100K features.

α	CV Accuracy
1×10^{-20}	0.88754
1×10^{-18}	0.88907
1×10^{-16}	0.89096
1×10^{-14}	0.89304
1×10^{-12}	0.89608
1×10^{-10}	0.89931
1×10^{-8}	0.90255
1×10^{-6}	0.90684
1×10^{-4}	0.91132
0.01	0.91458
1	0.90077

The hyper-parameter α was chosen with a 10-fold cross-validation grid search, and was consistently chosen to be 0.01.

B. *Nonlinear*

1) Implementation: Random forest was chosen as our non-linear model. Implementation of random forest adhered to the definition introduced in the lectures, with some changes for convenience in programming details. The implementation has the following hyper-parameters:

- Number of trees `k` in the forest.
- Number of features to be randomly selected `m` at each iteration.
- Minimum size of the node `min_node_size`, at which point it will no longer be split.

Regarding optimization, since each tree in the forest can be trained independently from each other, there was a good opportunity for optimizing the training by performing the training in parallel so that all CPU cores are utilized. There were several attempts to improve the training process by exploiting this, but were not successful in the end due to a restriction in Python. For other main stream programming languages such as Java or C/C++, the operating system automatically distributes thread level parallelism work to all available CPU cores. In contrast, Python underlying implementation using C (CPython) has the Global Interpreter Lock which only allows one block of code of the program to be executed at one point in time. This results in thread parallelism being meaningless for CPU intensive tasks (but is still useful for IO bounded tasks). Java implementation of Python (Jython) does not suffer from this, but is the amount of available libraries is very limited since most Python extensions for machine learning packages are written using C/C++ for performance reason. Although a workaround is possible, we chose to utilize our CPU cores using a different approach presented below.

The choice for these hyper-parameter values were done with cross validation on a grid search. To overcome the limitation of being limited to one CPU core mentioned above, the grid search was done in parallel by manually spawning multiple

Table II: Hyper-parameter grid search for random forest model using 55000 features.

No	k	m	min_node_size	validation_accuracy	time (s)
1	5	10	20	0.7568	2223
2	10	10	20	0.8201	4560
3	10	20	20	0.8335	15860
4	10	30	20	0.8358	13865
5	10	40	20	0.8382	10515
6	20	10	20	0.8493	9863
7	20	20	20	0.8120	34767
8	20	30	20	0.8244	44201
9	30	10	20	0.8639	29197
10	40	20	20	0.8733	43002
11	10	10	50	0.8093	3277
12	20	20	50	0.8598	12579
13	20	30	50	0.8575	20011
14	30	20	50	0.8683	30230

processes with different hyper-parameters, and thereby utilizing multiple available CPU cores on the machine.

In addition, once each tree node finalized in our training, we discarded the data held by that node, and only kept the class predictions together with their weights (e.g. 25% math, 25% physics, 35% computer science and 15% statistics). This saves memory since at the end of the training process, data instance was not necessary anymore (unless we wanted to continue training the tree possibly with new data, which was beyond the scope of this project).

During the training process, we noticed that matrix splitting at each iteration was the bottleneck, occupying on average 50% to 60% of the training time. Numpy array implementation would have delivered much better performance, but since the feature matrix was encoded as compress sparse row matrix for memory efficiency (see Naïve Bayes implementation above), slicing performance was greatly reduced.

2) Choosing hyper-parameters: A grid search for hyper-parameters was used during the testing, each with cross validation with 5 folds. Due to the constraints in performance mentioned previously, we were only able to train the implemented forests with 55000 extracted features. Note that the run time presented below might not be 100% accurate due to the use of swap memory during training (since default RAM was insufficient) and interference of other processes (e.g. browser, media player…)

The following observations were made from the table:

- It is noticeable that accuracy was increasing significantly as the number of trees increased. For example, No. 7 and No. 10 (improved 6% by doubling the number of trees). Increasing number of trees in the forest naturally increased overall run time.
- On the other hand, an increase in the number of features used at each iteration did not improve validation accuracy as much (e.g. No.4 and 5, No.12 and 13). Increasing number of features used at each iteration also naturally increased overall run time.
- Lastly, an increase in the minimum size of the node deteriorates the performance, at the cost of run time. For example, No. 2 and No.11

C. Optional

As our optional additional model, we used Scikit-learn's `LinearSVC`. This SVM produced good results, but was very slow (it took several hours to train) and did not perform quite as well as our Naïve Bayes implementation. We introduced bagging and semi-supervised learning in exactly the same way we did with Naïve Bayes, but this only marginally improved the model's performance. Because of the tremendous amount of time it took to train the model, we were unable to do much experimentation with hyper-parameters.

We also experimented with our own implementations of neural networks and K-Nearest Neighbors, but both of these performed poorly; KNN failed to produce more than 70% cross-validated accuracy, and the neural network failed to produce any results at all (with the number of features in our dataset, and without the ability to use sparse matrices, the memory use ballooned to several gigabytes).

V. Testing and Validation

A. Linear

Table III: Classification report on standard Multinomial Naïve Bayes

Class	Precision	Recall	F1-Score
cs	0.89	0.92	0.91
math	0.96	0.94	0.95
physics	0.92	0.92	0.92
stat	0.89	0.87	0.88
Average	0.92	0.92	0.92

Table IV: Validation confusion matrix on standard Multinomial Naïve Bayes

	cs	math	physics	stat
cs	5284	145	113	182
math	161	4820	80	66
physics	179	31	3287	93
stat	322	37	76	2852

Our Naïve Bayes model performed exceedingly well. Like many other teams, our validation accuracy and cross-validated accuracy (computed with Scikit-learn's `cross_val_score` function) was significantly greater than the score on Kaggle. Our best-performing model (an ensemble of 100 Multinomial Naïve Bayes classifiers trained on data extended through semi-supervised learning) had 91.4121% 10-fold cross-validated accuracy, but scored only 0.83188 when submitted to the site. Our accuracy on the set-aside validation data was consistently very similar to the cross-validated accuracy, which leads us to believe that the model should generalize well.

B. Nonlinear

Our best validation accuracy achieved was `0.8733` with 40 trees, 10 features and 20 minimum node size. To compare our implementation with the reference implementation of scikit-learn, we also did several runs using the library on 55000 and 100000 generated features, since the run time is much better compared to our implementation). The run time was

Table V: Hyper-parameter grid search for reference random forest model from scikit-learn

No	k	min_node_size	val_acc	Note
1	500	1	0.8803	55k features
2	1000	20	0.8774	55k features
3	1000	1	0.8806	55k features
4	1000	1	0.8869	100k features
5	1000	1	0.8808	100k features, cv=10

approximately 4 to 5 times better due to optimization of matrix slicing process. That said, we were not able to use the reference library to train for beyond 1000 trees due to memory limitation. The following table describes the grid search using scikit-learn library. Value of `m` (number of features used at each iteration) was chosen to be square root of the number of features (by default from the library).

It is noticeable that using 100000 features improved the validation accuracy. Despite being able to train with the number of trees and number of features significantly larger, the reference library did not deliver a significant improvement in terms of validation accuracy. Reference library best validation accuracy was `0.8869` while our implementation best validation accuracy was `0.8733`. Despite this, the performance of random forest was worse compared to that of Naïve Bayes.

C. Optional

Just as with other models, our SVM did well on Kaggle, but much better on the validation set. Interestingly, although the SVM did not do as well as Naïve Bayes on Kaggle, it far outperformed the linear model in cross-validated accuracy, with the SVM achieving 96.9328% 10-fold CV accuracy, compared to Naïve Bayes' 91.4121%.

VI. Discussion

A. Linear

We are very happy with the results of the Naïve Bayes classifier. Not only were we able to achieve the top position on the Kaggle leaderboard, but we feel our code is very efficient - when comparing to other implementations (such as Scikit-learn's `MultinomialNB`), our model produced identical results, and was often marginally faster (as a result of the vectorized math and support for sparse matrices).

The weakest part of the learning pipeline was probably the feature selection. Although lemmatization improved the information density significantly, we feel that there is probably a more effective representation of the data. If given more time, we may have experimented with word embeddings, but we are not sure how that would have worked with the models we chose - a neural network may have been able to handle that kind of representation, but it makes little sense for a classifier like Naïve Bayes.

B. Nonlinear

Although performing much better than the baseline classifier, random forest was not as effective as Naïveve Bayes in this classification problem. Our best validation accuracy using our own implementation was `0.8733`, which translated to `0.7664` accuracy on the test set. Our best validation accuracy using reference library was `0.8869`, which translated to `0.7833` accuracy on the test set.

Although our implementation of random forest model was accurate (performing nearly as well as the reference library), many run time optimizations could have been applied. Among these possible optimizations, process-based parallelism was most unfortunately not implemented. As a result, the implemented model took a very long time to train compared to the reference library. This also limited our grid search size to relatively small because of time constraint. Fortunately, we were able to mitigate this and utilized more CPU cores by manually training the model with different parameters in different processes.

Despite not being able to optimize the implementation significantly, we believe that the bottleneck to random forest accuracy was either at feature selection stage or the model choice itself given that the reference library did not improve validation accuracy as much as expected despite being trained with much larger number of trees and features (100000 features) compared to our implemented model. Exploring other alternatives (such as Support Vector Machines with appropriate kernel) may have yielded a better accuracy for our nonlinear model.

C. Optional

Working with SVMs on this project proved to be frustrating - the training was extremely slow when compared to Naïve Bayes, and the results were only marginally better, worse when we used additional methods to improve Naïve Bayes, like semi-supervised learning and ensemble methods.

VII. Statement of Contributions

It should be noted that we were originally a group of three, but the third member had to withdraw from the course. Fortunately, most of her contributions were well-separated from ours (she was working on a KNN classifier), so removing her work from the final submission was simple. We hereby state that all the work presented in this report is that of the authors.

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. Hoai Phuoc Truong

Hoai Phuoc Truong was responsible for the introduction and the related work section of the report. He was also responsible for the implementation of nonlinear model and writing up nonlinear section of the report.

References

[1] Thorsten Joachims. “Text categorization with Support Vector Machines: Learning with many relevant features”. In: *Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings*. Ed. by Claire Nédellec and Céline Rouveirol. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 137–142. ISBN: 978-3-540-69781-7. DOI: 10.1007/BFb0026683. URL: http://dx.doi.org/10.1007/BFb0026683.

[2] 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/.

[3] Leah S. Larkey and W. Bruce Croft. “Combining Classifiers in Text Categorization”. In: *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’96. Zurich, Switzerland: ACM, 1996, pp. 289–297. ISBN: 0-89791-792-8. DOI: 10.1145/243199.243276. URL: http://doi.acm.org/10.1145/243199.243276.

[4] 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.

[5] Tom M. Mitchell. *Machine Learning*. WCB McGraw-Hill, 1997.

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