Mini-Project #2: Text Classification

COMP-551: Applied Machine Learning

Kaggle Team : JulMistYue

Yue Dong and **Mansha Imtiyaz** and **Julyan Keller-Baruch**

{yue.dong2, mansha.imtiyaz, julyan.keller-baruch}@mail.mcgill.ca

260408330, 260712985, 260388157

1 Introduction

For this project, a Kaggle competition was set up to analyze and classify short conversations extracted from the Reddit website into 8 categories: hockey, movies, nba, news, nfl, politics, soccer and worldnews. The goal is to utilize different machine learning algorithms and make predictions of topics for the test set. Our approach was to extract features using Natural Language ToolKit [1] along with Scikit Learn Library [1] and then use all or a subset of these features to train linear and non linear classifiers.

2 Related work

Text classification is a wide area and much work has been done in this field. Classifiers such as Naive Bayes, K-nearest neighbors, Random forests, Logistic regression, etc. are used to solve the text classification problems. Choosing features is as important as finding the best algorithms for classification. One of the most common feature models is the bag of words model, where words are counted and their frequency is calculated. Using Inverse Document Frequency with the bag of words model is said to considerably improve the performance of algorithm [5].

One common characteristic of the bag of words model is the high dimensional feature space. This leads to the problem in learning called *the curse of dimensionality*. We partially tackled this problem by implementing and using Domain Specific Classifier based on paper [3], which improves the speed of similarity search on high-dimensional text dataset as described in section 4.2.1.

[1] http://www.nltk.org/book/ch03.html

3 Problem Representation

3.1 Data Set

There were around 165,000 samples provided for training/validation and 53,218 samples in the test set. The distribution of categories in the dataset is shown in Figure 1. Upon visual inspection of the classes distribution, we assume that the classed are well balanced.

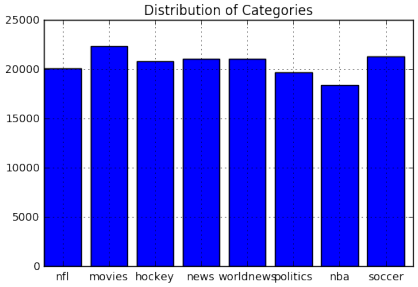

Figure 1: Distribution of Categories in the Training Data

3.2 Text Preprocessing and Cleaning

We first converted the raw text to lowercase and used lemmatization and stemming for better feature extraction [2]. Digits were removed from the raw text as they do not contribute towards the classification. New line characters (\n \r) were removed. Stopwords, punctuations, html formatting encodings (", &, <, and >), domain

[2] We kept it as an option to include or not include lemmatizing/stemming for model development and achieved better performance with lemmatizing/stemming.

extensions (.com, ,org), and speaker/ number tags (<speaker1>) were also removed from the raw text.

3.3 Feature Extraction

Initially, we started with a feature space including 6000 unigrams. We then increased the number of features to around 20,000 to improve the classification accuracy at the cost of longer feature processing time. Based on our experiments, we decided to use the scikit-learn function 'TfidfVectorizer' to transform the data as vectors with TF-IDF values. The TF-IDF values count words or ngrams and return feature vectors. These word counts are then normalized by the frequency of that word in the entire document set. After tuning and iterations, the value of parameters for the vectorizer are chosen as follows:

- **n-gram** : We experimented with unigrams, bigrams and trigrams on several iterations and finalized on a combination of all three which yielded the best accuracy.
- **min-df** : This is the minimum document frequency, each of the word should have to be included as a feature. We decided to use min_df value as **0.01** to filter out the words which occur in only one or two documents.
- **max-df** : We set it to **0.9** so that the words which are too common and occur in almost all of the documents aren't counted as features since they won't have any effect on the classification.
- **smooth-idf** : This was chosen to be True so that the features which weren't seen in the training data were smoothed, i.e idf weight were smoothed by adding one to document frequencies.
- **sub-linear** : To smooth out the vector form, we used sub-linear tf scaling, replacing term frequency by log of term frequency plus 1 (1 + log(tf)).
- **norm** : l_2 norm is chosen for normalization.

After the tf-idf transformation, all the features were saved as sparse matrices to reduce storage requirement and the computation time.

4 Algorithm Selection and Implementation

4.1 Linear Classifier: Naive Bayes

As a baseline classifier, the Naive Bayes algorithm was implemented from scratch in python. The Naive Bayes algorithm is a generative classification model. That is, instead of directly approximate $P(y|x)$, we focus on building a probabilistic model to estimate the likelihood $P(x|y)$. We then use the Bayes rule to compute the posterior and classify new examples according to their most probable class.

The Naive Bayes classifier makes a strong assumption about the conditional distributions among all features: it assumes that all features are conditionally independent. Given a data point x = $(x_1, ..., x_m)$ with m features, $p(x|y)$ could be calculated as:

$$p(x|y) = \prod_{i=1}^{m} p(x_i|y)$$

For our classification task, a list of features was built by generating a frequency distribution of all words across all documents. Since the values in each feature represent the counts, we consider our features as categorical variables. We therefore assume all features follow **multinomial** distributions and used the relevant frequency counting to represent the conditional probability as equation 1. Suppose feature j is categorical with K categories, then

$$p(x_j = k|y) = \frac{N_{yk}}{N_y} \tag{1}$$

where $N_y = \sum_{i=1}^{K} N_{yi}$ is the totaly count of data instances in class y and N_{yk} is the sum of feature k appears in the j-th feature of the training set of class y.

The following are our implementation details. The words were sorted in decreasing order by their number of occurrences across the entire dataset. The features that our algorithm was trained on consisted of a list of the first k elements from the list of features. The search for the optimal k was determined in the cross-validation step. The prior for each document class was then calculated by dividing the number of documents in that class by the total number of documents. The conditional probability of a word w given a particular document class Y was computed

as the number of occurrences of w in Y divided by the sum of the number of words in Y (with duplicates) and the length of our feature set k. Each of the computed conditional probabilities were stored in a dictionary where the keys are the words associated to these probabilities and the values are the probabilities themselves.

We then classified the documents in our test set and assessed the accuracy of our predictions. Before doing so, each test document was stripped of words that had low predictive power in a procedure identical to the one mentioned above. We then removed all words that were not part of our feature list. Using the conditional independence assumption, for each test document, the probability of it being in each particular class y_i was calculated by taking the product of the prior of y_i and the conditional probabilities of each word in that document given y_i. The class for which this probability was highest was outputted as the predicted class for that document. This was repeated for all documents in the test set and predictions were compared with known class values and accuracies were computed by assessing number of false predictions over the entire test set. Details of training on validation set is given in the next section.

4.2 Non-linear Classifier : k-Nearest Neighbors

We implemented the k-Nearest Neighbors classifier with the direct distance search based on L_p distances. There are two hyper-parameters in our algorithm: (1) k – the number of nearest neighbors used for majority votes; (2) p – the parameter for the L_p distance.

After cleaning the data, we used tf-idf to transform the text into the weighted counts as described in the previous section. Since the kNN classifier is slow compared to the linear classifiers, we were only able to use $15,000$ features [3] for training and testing.

Regards to the training/validation split, we decided to choose stratified random splits with 10000/1000 for training and testing, respectively. We didn't use k-fold cross validation because of the limitation of computational powers and the run time of our implementations. The k-NN classifier we implemented involves computing pairwise distances of all points in the dataset. For N samples in D dimensions, this approach scales as $\mathcal{O}[DN^2]$. A possible improvement here is to use K-D Tree or Ball Tree to find nearest neighbors as discussed in Nearest Neighbors of [1].

In terms of setting hyper-parameters, we used the grid search to find the best k and p for classifying the data. The grid search is based on exhaustively considers all parameter combinations of $k \in \{1, 2, 3, 5, 10, 15\}$ and $p \in \{1, 2, 3, 5\}$. Note that when there is a tie from the majority vote of k nearest neighbors, we randomly choose one among the most common classes.

One problem of using the kNN classifier in text categorization tasks is the so-called *the curse of dimensionality*. The curse of dimensionality introduces sparseness of the training data and this sparseness is not uniformly distributed. More points are concentrated on the corners of the feature space which makes the distance search difficult in high dimensions [4].

4.2.1 Domain Specific Classifier

In our task, the kNN classifier searches similar documents as the query document based on some distance metrics. When there are large numbers of documents, the similarity search could become very inefficient.

We implemented a classifier called Domain Specific Classifier (DSC) based on the paper [3] to tackle the problem of similarity search. Compared to the kNN classifier, DSC is very fast in similarity search because the training data are represented by only a few points with domain specific words.

In DSC, we look for words appear more times in one category than the sum of all other categories [4]. Then we represent all the documents in each category by a single vector with 0s and 1s indicating the set of domain specific words of this categories [5]. In the classification stage, we classifier the query document as in class j if there are more j-th domain specific words appearing in the query document. This

[3] selected by setting max_feature $= 15,000$ in tf-idf transformation

[4] $f_j(t) > \alpha \sum_{j' \neq j} f_{j'}(t)$ where $f_j(t)$ is the average count of word t appears in all documents in category j and α is a threshold parameter.

[5] If a training dataset has n documents with M distinct words, the training matrix for the kNN classifier is $n \times M$. Suppose there are k classes, the training matrix in DSC is $k \times M$ with $k << n$.

is done by take the argmax of the inner product of the query document to each domain specific vector v_j.

In our experiment, we used 78,832 features and 10-folds cross-validation for training and validating because DSC is very fast. The only hyper-parameter is α which decides how many domain specific words will be in each category. As α increases from 0, the number of domain-specific words for each class label decreases. The optimal choice of α is decided by grid search from 0.1 to 3 with a step size 0.3 combine with cross-validation.

4.3 Scikit-Learn Classifiers

For our optional additional model, we experimented with several classifiers, including Naive Bayes and kNN classifier from the Scikit-learn library, to compare the results with our own implemented algorithm. The results of all the classifiers are shown in section 5.3.

Out of these, the best results were obtained by using Stochastic Gradient Descent Classifier with *modified huber* loss. We did an exhaustive grid search for most of the classifiers for selecting hyperparameters which yield the best accuracy. For Logistic, hyperparameters chosen after optimization were 'lbfgs'[6] for solver, 20,000 features, 'l2' as penalty and C (inverse regularization factor) as 10. For SGD classifier, we tried various loss functions, different values of alpha and n_iter selected as 20. Table 4.3 shows how accuracy was increased with varying hyperparameters for SGD classifier. SVM gave good results for small number of training samples but as we increased the number of samples, the computation time increased and mostly resulted in *Memory Error* or *Process Killed*. We assume that SVM would have performed comparatively better than other models, had there been no memory limitations.

Table 1: ***Hyper-parameter Grid Search for SGD Classifier***

No.	Loss	Penalty	alpha	n_iter	Accuracy
1	hinge	l1	1e-04	10	77.5
2	hinge	l1	1e-05	10	90.9
3	hinge	l2	1e-04	10	92.2
4	hinge	l2	1e-05	10	95.8
4	hinge	l2	1e-05	10	95.8
5	hinge	l2	1e-05	20	96.0
6	modified_huber	l2	1e-05	20	96.0
6	modified_huber	l2	1e-06	20	96.3

[6]'lbfgs' is an optimization method in the family of quasi-Newton methods for parameter estimation.

We experimented with Adaboost and Bagging classifer as well, but the accuracy wasn't improved.

5 Testing and Validation

5.1 Linear Classifier

To verify that the accuracy obtained was attributable to our feature selection and not simply an artifact of our data partitioning, we used *3-fold cross-validation*. The data was split into 3 equal parts of size 55,000 and trained using two parts and tested on the third, for all three combinations of training/testing partitions. We experimented with varying features and ngrams to get to the best accuracy as shown in Table 2.

Table 2: ***Accuracy on Validation set by varying N-grams and Number of Features and Kaggle Submission Score***

N-Gram	No. Features	CV. Acc.	Kaggle Acc.
1-1	500	67.92%	-
1-1	1000	72.56%	-
1-1	5000	80.25%	-
1-1	10,000	86.39%	86.411%
1-2	10,000	89.62 %	89.831%
1-2	15,000	90.05%	90.958%
1-3	15,000	91.24%	91.262%
1-3	20,000	91.86 %	91.909%

Table 2 shows that using combination of unigrams, bigrams and trigrams improved the accuracy. After 20,000, the no. of features had less or no effect on improving the accuracy. The confusion matrix and classification report for our best performing Naive Bayes is given in the Appendix section.

5.2 Non - Linear Classifiers

Because we are dealing with multi-class classification, ROC or AUC is not applicable as the evaluation metrics. We mainly use accuracy as the performance indicator, as well as confusion matrix, precision, recall and F1 score.

The result of the kNN classifier on the validation set is summarized as table 3. Based on table 3, we decide to use $k = 15$ and $p = 2$ for the prediction on our final test dataset. The kaggle test dataset ac-

curacy based on 15,000 feutures and 50,000 training data is 85.216%.

Table 3: ***Accuracy of the kNN classifier based on different k and p values. We used 15k features (tf-idf) and $10000/1000$ for train/validation splits.***

acc.	k=1	k=2	k=3	k=5	k=10	k=15
p=1	0.43	0.34	0.38	0.40	0.41	0.40
p=2	0.76	0.73	0.73	0.78	0.78	**0.82**
p=3	0.49	0.44	0.41	0.39	0.39	0.39
p=5	0.45	0.39	0.38	0.38	0.35	0.34

The result of DSC on the validation set is summarized in table 4. Based on table 4, we decide to use $\alpha = 1$ for the prediction on our final dataset. The kaggle test dataset accuracy is 86.749%.

Table 4: ***Accuracy of Domain Specific classifier based on different α values. Around 78k features were used for training and testing with 10-fold cross-validation.***

acc.	α=0.1	=0.4	=0.7	=1	=1.3
	0.733	0.856	0.864	0.863	0.860
acc.	α=1.6	=1.9	=2.2	=2.5	=2.8
	0.859	0.856	0.848	0.842	0.835

It's interesting to see that changing the distance metrics in kNN classifier from Euclidean to other L_p distances actually hurts the performance. On the other hand, increasing the number of neighbors for prediction improve the accuracy. It's intuitively making sense that more neighbors will leads to better predictions.

The codes were running on a 2.7 GHz Intel Core i5 machine with 8GB memory. Compared to the running speed of kNN classifier (around 2 hours on each 10000/1000 train/validation set), DSC only need averagely 100.8 seconds to run with 10-fold cross validation. Although DSC represents the training data in a lossy compression, DSC still outperforms the kNN classifier in our validation set and give comparable results in the kaggle test set.

5.3 Scikit-Learn

We experimented with several different classifiers, by training on 70% of the training data and doing validation on the remaining 30%. Table 5.3 shows the accuracy, precision, recall and f-Score and best submission score on Kaggle for various models of Scikit Learn Library. For the classifier with poor performance on validation set, predictions were not generated for the test input file.

Table 5: ***Accuracy, Precision, Recall, F-Score on Validation Set and Best Submission Score on Kaggle for various classifiers of Scikit-Learn Library***

Classifier	Accuracy	Precision	Recall	F-Score	Kaggle
Decision Trees	0.6231	l1	1e-04	10	-
Bernoulli Naive Bayes	0.9084	0.91	0.91	0.91	0.91864
Multinomial Naive Bayes	0.9001	0.90	0.90	0.90	0.90875
kNN Classifier	hinge	l2	1e-05	10	0.86898
Logistic Regression	0.9194	0.92	0.92	0.92	0.92149
SGD Classifier	0.9659	0.97	0.97	0.97	0.97133

The validation results for Adaboost and Bagging classifier were 62% and 92% respectively, which were not an improvement on our best score. The confusion matrix for the best performing classifier is shown in table 5.3.

Table 6: ***Confusion Matrix for SGD CLassifier***

	NFL	Movies	Hockey	News	Worldnews	Politics	NBA	Soccer
NFL	4323	8	15	6	18	3	30	5
Movies	3	4737	4	14	1	1	3	7
Hockey	31	9	3792	6	26	2	13	3
News	4	29	4	4043	18	171	4	180
Worldnews	10	10	27	9	4216	1	11	0
Politics	0	5	2	186	1	3848	4	55
NBA	22	10	7	4	14	1	4503	19
Soccer	0	11	0	94	1	50	16	4345

Figure 2 shows the classification report for the best performing classifier.

```
Clasification report:
             precision    recall  f1-score   support

        nfl       0.98      0.98      0.98      4408
     movies       0.98      0.99      0.99      4770
     hockey       0.98      0.98      0.98      3882
       news       0.93      0.91      0.92      4453
  worldnews       0.98      0.98      0.98      4289
   politics       0.94      0.94      0.94      4101
        nba       0.98      0.98      0.98      4580
     soccer       0.94      0.96      0.95      4517

avg / total       0.97      0.97      0.97     35000
```

Figure 2: Classification Report for Best Performing Classifier

6 Discussion

We had started uploading submissions on Kaggle from the third day since the competition was released. For our baseline classifier, we had chosen Naive Bayes and obtained a classification accuracy of 86% on our first submission. With each submission and iteration, we improved our results and eventually became one of the top 10 teams with around 97% classification accuracy.

As shown in section 5, non-linear classifiers, such as kNN classifiers are not ideal for text classification. Compared to the linear models, the accuracy for the predictions wasn't improved. The biggest bottleneck of non-linear classifiers is the speed of learning. Moreover, since the data is sparse in high-dimensional spaces, using a non-linear classifier doesn't improve the results.
We also discovered that the error is mostly caused by *news* class being predicted as *soccer* or *politics* and vice versa from Table 5.3(*confusion matrix for our best classifier*). This could be due to the noise in data or documents with ambiguous features which could be interpreted as both. We assume that the accuracy could have been improved by adding some external features such as name of players for the sports category and alike.

7 Statement of Contributions

We worked as a team for defining the problem. Data analysis and Feature Processing was done by Mansha. Julyan implemented Naive Bayes. Yue worked on kNN and Domain Specific classifier. Experimenting with scikit learn classifiers (part3) was done by Mansha. Everyone wrote about their findings and details of their particular section.
We hereby state that all the work presented in this report is that of the authors.

8 References

1. F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In *Journal of Machine Learning 12* (2011). pp2825-2830

2. Vincent Spruyt. "The Curse of Dimensionality in classification". Online blog.

3. Duan, Hubert Haoyang, Vladimir G. Pestov, and Varun Singla. "Text Categorization via Similarity Search." International Conference on Similarity Search and Applications. Springer Berlin Heidelberg, 2013.

4. Edo Liberty. "Nearest Neighbor Search and the Curse of Dimensionality". Lecture notes.

5. Tsai, Chih-Fong. "Bag-of-words representation in image annotation: A review." ISRN Artificial Intelligence 2012 (2012).

Appendix

The complete pre-processed datasets can be found at `https://drive.google.com/open?id=0B1y42dcrN1bLZGktbDBTN0QxTGc`

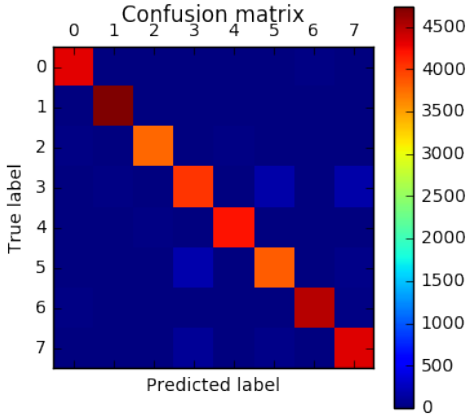

Figure 3: Confusion Matrix Report for Naive Bayes

Clasification report:

	precision	recall	f1-score	support
nfl	0.95	0.95	0.95	2051
movies	0.95	0.97	0.96	2281
hockey	0.97	0.94	0.95	1773
news	0.75	0.75	0.75	2076
worldnews	0.96	0.95	0.95	2045
politics	0.80	0.82	0.81	1907
nba	0.96	0.96	0.96	2216
soccer	0.86	0.86	0.86	2151
avg / total	0.90	0.90	0.90	16500

Figure 4: Classification Report for Naive Bayes

overall accuracy 0.863545454545

	precision	recall	f1-score	support
hockey	0.95	0.90	0.92	1386
movies	0.93	0.95	0.94	1475
nba	0.90	0.95	0.92	1254
news	0.79	0.59	0.67	1397
nfl	0.92	0.94	0.93	1339
politics	0.68	0.86	0.76	1345
soccer	0.95	0.91	0.93	1422
worldnews	0.82	0.82	0.82	1382
avg / total	0.87	0.86	0.86	11000

Figure 5: Classification Report for Domain Specific Classifier