Project 2: Text Classification
Team Name on Kaggle : INS

Inderjot Kaur Ratol
260661928
inderjot.ratol@mail.mcgill.ca

Nicolas Truong
260430127
thanh-huan.truong@mail.mcgill.ca

Senjuti Kundu
260669284
senjuti.kundu@mail.mcgill.ca

1. Introduction

The goal of this project is to classify conversations, extracted from Reddit threads from 8 different subreddits, into their corresponding subreddit or category. The class labels/categories provided are movies, hockey, nba, nfl, soccer, world news and news. Our task was to choose three classifiers to perform the categorization of conversations. We decided to complete the task with Naive Bayes, Decision Trees and Support Vector Machines.

2. Problem Statement

The raw data was provided as a file containing a unique id per conversation, where these conversations belonged to one of the eight categories mentioned previously. As the task was to perform supervised learning, we were provided labeled conversations where the label referred to the category of a conversation. In order to prepare the data for the learning step, we needed to preprocess the data.

2.1. Preprocessing the data

We followed these steps to preprocess the raw data before using it in the learning step.

- First, we divided the conversations containing multiple sentences into a list of sentences.
- Every sentence was then used to create tokens. Here tokens refer to the individual words as well as non-ASCII characters i.e. punctuation marks, numbers etc.
- We purged all the punctuation marks and numbers from the tokens.
- We further cleaned the data by removing non-informative words like ”is”, ”an”, ”the” etc. which are called *stopwords* in the context of Natural Language Processing.
- We extended the standard NLTK *stopwords* list with some words which were observed to be non-informative. For example, ’com’, ’co’, ’gt’, ’st’, ’th’, ’u’ and ’r.
- Considering all the possible data transformation which might obtain better results, we decided to create *lemmas* for all the words in our vocabulary. Lemmatization is the process of finding the inflectional form of a word [1]. So, we kept both normal tokens as well as their corresponding lemmas for further processing. We preferred lemmatization over stemming because a substantial amount of information is lost while stemming the words and we did not consider this approach suitable for the problem at hand.

2.2. Feature Selection

The selection of features differs based on the type of classifier. Below, we have discussed the feature selection for all the three classifiers used in this project.

2.2.1. Naive Bayes. We chose to compare the results based on the following features:

- **Lemmas vs original words** The rationale behind choosing lemmas in the first place was our assumption that minor differences in words, depending upon the parts-of-speech and getting a single word for both singular and plural, will help in compressing the sparse input matrix and will produce more favorable results. So, we decided to treat those differences by using lemmatization.
  We experimented by using both lemmas and the original words for our classifier Naive Bayes. By comparing the accuracy and F1 score difference between the two versions, we decided to go with the original words because using lemmas always dropped the accuracy by at least 6% and proved that our assumption about the negligence of singular/plurals for this particular data set is not valid.

- **Frequency (count) vs TF-IDF** We used both counts of words in each category and their *Term frequency with Inverse Document Frequency*. TF-IDF penalizes the words which are common amongst all the categories, thereby providing a

1. https://en.wikipedia.org/wiki/Lemmatisation

better data set to be used with Naive Bayes by assuring partial exclusivity between the words present per category. In order to comply with the given data, we calculated TF-IDF frequencies by replacing document in the standard TF-IDF formula with the conversations. Contrary to what we expected, we got lower accuracy with TF-IDF values as compared to word counts. So, we dropped the TF-IDF's and used word counts as a feature.

- **All words vs n- Top words per category** Next, we decided to experiment with the number of words we use per category. We assumed that most related words per category will be having a higher frequency in that category. So we created 8 lists, one per category, containing all the words that have ever appeared in a category. The lists were sorted in the descending order of word frequencies (counts). So we experimented with the number of words we chose to train our classifier on. Starting with top 200, 300 to 2000, we calculated accuracy along with 6-fold cross validation per experiment. Accuracy increased with the increasing number of words used for training. Comparing the top 2000 words against all the words, there was a minor difference in accuracy. There was a trade-off between accuracy and longer processing time and we chose better accuracy with more processing time.

2.2.2. Decision Trees. For the decision tree method, the 150 most common words in each of the different categories were found. These words of each category were merged together into a master list where any duplicates were eliminated. This list of most common words found amongst all 8 subreddits served as the master list to form the binary vectors defining individual conversation and this list consisted of 450 words. For example, if the master list was of the form ['news', 'sports'], and a conversation contained the word 'news' but not 'sports', then its binary vector representation was [1, 0].

Similarly to the Naive Bayes method, lemmatization was attempted on the dataset for the Decision Tree, but also ended in a decrease in accuracy and F1 score. For this reason, original words were kept as is when training the tree.

2.2.3. Support Vector Machines. We experimented with the following features:

- **Single words vs N-grams** Following the general trends prevalent in NLP, we decided to classify the text conversations using bigrams (2-grams) and trigrams (3-grams) instead of single words. Using bigrams resulted in an increase of 2% accuracy as compared to the results produced using single words as features. Using a range of n-grams from 1 to 3-grams further increased the accuracy by 3%.
- **Frequency (count) vs TF-IDF** Similar to Naive Bayes, TF-IDF values reduced the accuracy of the classifier. Hence, we used the counts only.

3. Algorithms and Implementation

3.1. Naive Bayes

Multinomial Naive Bayes is used for the text classification task. As discussed in Section 3.2.1, features used in the implementation are tokens (original words), word frequencies and all the words appearing per category which is used for the final learning phase.

3.1.1. Training Phase. 6-fold cross validation is used to train the classifier. For training purpose, we only calculate the frequency distributions of the words per conversation and add the obtained *bag of words* to the related category's word collection. The end result of the training phase is the frequency distribution of each word per category and prior probabilities.

Prior Probabilities Based on the training phase data, prior probabilities are calculated for each category. The formula used for prior calculation is -

$$\frac{N(C)}{TC}$$

where N(C) is the number of conversations belonging to category C and TC refers to total conversations in the training set.

3.1.2. Testing Phase. We used the *bag-of-words* and frequency distributions obtained in the training phase to test and make predictions. These are the steps we followed in the implementation of Naive Bayes:

- **Word Probabilities** Considering the Naive Bayes assumption of conditional independence of probabilities, the probability of a word with frequency f is calculated as follows:

  $$p(w/C) = \frac{f_w}{TW_C}$$

  where f_w denotes the count of word w occurring in category C and TW_C denotes total number of words in C.

  In Naive Bayes, the maximum likelihood for a particular category is calculated by multiplying the individual word probabilities with the prior probability of the related category.

  Problem- This type of multiplication creates a problem when some unseen word appears in the test data, which results in a word probability 0 and subsequently affects the likelihood of the whole category being predicted as an output. While experimenting with the classifier, we found that there are always some words that never appear in the training set but will appear in test data, thereby jeopardizing the

maximum likelihood estimates. In order to avoid this problem, *Laplace* smoothing is used.
Solution-
Laplace Smoothing- This method gives us the following formula for word probabilities:

$$p(w/C) = \frac{f_w + 1}{TW_C + |V|}$$

As explained in Jackie's Notes, it assumes the possibility of at least one occurrence of an unknown word in the document/conversation. In the denominator, it adds the number of words present in our vocabulary $|V|$, where $|V|$ is the total number of unique words in the whole training data set. So for words with no prior occurrences, f_w will be 0. But they will still have a probability greater than 0, calculated based on the formula above.

3.1.3. Predictions. Calculating the maximum likelihood estimates for a conversation for each category leaves us with 8 probability values at the end. Choosing the maximum value provides us with the likely category that conversation would belong to. All the predictions are then added to a list and are compared against the actual labels for test data set for validation and testing. Table 1 shows the results produced by training the classifier over 80% of the data and testing the predictions on rest of data i.e. 20%. While calculating final accuracy, classifier was not trained using cross-validation and was fed the training data directly so that there is no bias and testing set remains as unseen data for the classifier.

TABLE 1. RESULTS FOR NAIVE BAYES

category	Precision	Recall	F1-Score
hockey	97%	88%	92%
movies	95%	97%	96%
nba	96%	89%	92%
news	84%	72%	77%
nfl	85%	95%	89%
politics	75%	92%	83%
soccer	97%	94%	95%
world news	83%	87%	85%
avg/total	89%	89%	89%

3.2. Decision Trees

Decision trees were the second method attempted to classify the reddit conversations. As mentioned in Section 3.2.2, binary features of whether a certain word was present in the conversation or not were used to form binary vectors. Leaves were created when the information gain resulting from the split returned 0, meaning that the split put all conversations in one group.

3.2.1. Training Phase. A 6-fold cross validation methodology was used to train the tree. A decision tree was first built using the training data, and then validated using the remaining data. Several attempts were made to try to fine-tune the training method. For example, it was possible to modify the number of most common words per category. It was noted that about 150 words per category gave the highest accuracy and anything higher than 150 added long computation times with little benefit.

Another technique to fine-tune the decision tree was Early Stopping, where the tree growth was prevented after a certain depth, inserting a leaf with the most dominant category at that point, instead of continuing building with nodes. This was done to prevent overfitting since leaves are becoming increasingly specialized to fit a very specific category and format at deeper levels, as well as helping with the training time by cutting down on the number of branches and nodes. However, after trying a few different numbers, shallow trees had worse accuracy while deeper trees had very little difference with no Early Stopping. This was ultimately kept to save computation time, however minimal it was.

3.2.2. Predictions. The results after a 6-fold cross-validation were averaged and are shown in Table 2. Given that that the accuracy of 70% is greater than 12.5% which represents the accuracy if it were purely guessing a category, the decision tree seems to be working to a certain extent.

TABLE 2. RESULTS FOR DECISION TREE

Accuracy	Precision	Recall	F1-Score
70%	70%	70%	70%

TABLE 3. RESULTS FOR DECISION TREE PER CATEGORY

category	Precision	Recall	F1-Score
hockey	70%	68%	69%
movies	85%	87%	86%
nba	69%	69%	69%
news	56%	57%	56%
nfl	73%	72%	72%
politics	68%	67%	67%
soccer	73%	73%	73%
world news	69%	70%	69%
avg/total	70%	70%	70%

3.3. Support Vector Machines

We have used Python's scikit-learn's implementation of Support Vector Machines. We have used two different SVM models, depending upon the method of preprocessing.

3.3.1. SVM using Doc2Vec. The preprocessing steps we have undertaken are as follows:

- Tokenization
- Removal of Punctuation

TABLE 4. RESULTS FOR SVM USING DOC2VEC MODEL

Category	Precision	Recall	F1-Score	Support
hockey	0.84	0.86	0.85	4150
movies	0.82	0.85	0.84	4493
nba	0.85	0.84	0.84	3682
news	0.53	0.56	0.54	4254
nfl	0.85	0.84	0.85	3951
politics	0.71	0.65	0.68	3963
soccer	0.87	0.87	0.87	4308
worldnews	0.69	0.69	0.69	4199
avg/total	0.77	0.77	0.77	33000

- Removal of html tags and emojis
- Stopword removal
- Stemming

Feature Extraction As an alternative to basic sparse feature representations like Tf-Idf and Bag of Words, we decided to use neural embeddings instead. Neural embeddings not only reduce the dimensionality of the feature set but also provide a dense semantic representation of the document under consideration. We have used Doc2Vec to convert a given document into a vector and used this vector for downstream classification tasks. Each Reddit conversation is considered as a separate document And therefore has its own vector.

Doc2Vec/Paragraph Vector is an algorithm based on Word2vec which converts a variable length paragraph or document into a fixed length feature vector. The feature vectors so generated preserve the semantic meaning of the documents such that similar documents end up close to each other in the embedding space.

The advantages of this feature extractor over standard information retrieval based feature extractors are:

- These do not give rise to high dimensional sparse feature vectors like Bag of Words and Tf-Idf. Typical vector dimensions in this case are between 300-500. These neural embeddings are not sparse by any means but are distributed.
- The semantic meaning of the paragraphs and documents are preserved in this case.

So we just preprocess the text and generate these document vectors using the doc2vec implementation in gensim. Once we have a trained doc2vec model on our training corpus we can use these document vectors for any classification tasks.

During testing, given the raw conversation text, we preprocess it and then infer a document feature vector using the trained Doc2Vec model.

Once we have this feature vector ready, we just pass it to a SVM classifier (i.e. which has already been trained during training phase).

The results after a 6-fold cross-validation are shown in Table 2.

3.3.2. SVM using CountVectorizer. We used Scikit learn's CountVectorizer to calculate the counts (frequencies) of the words. The CountVectorizer converts a collection of text documents to a matrix of token counts. Apart from the normal preprocessing steps mentioned in Section 2.1, we experimented with n-grams as well. Table 5 shows the results obtained after using CountVectorizer with stopwords removal.

TABLE 5. RESULTS FOR SVM USING COUNTVECTORIZER

Category	Precision	Recall	F1-Score
hockey	0.96	0.95	0.96
movies	0.97	0.98	0.97
nba	0.96	0.95	0.96
news	0.82	0.83	0.83
nfl	0.94	0.95	0.95
politics	0.87	0.86	0.87
soccer	0.97	0.96	0.97
worldnews	0.89	0.90	0.90
avg/total	0.92	0.92	0.92

Removing only the stopwords resulted in an accuracy of 92%. Further, we tried using stemmer, lemmatizer and TF-IDF values. However none of these features increased the accuracy but instead increased the processing time. So we tried using n-grams. We started with bigrams which increased the accuracy from 92% to 96%. Encouraged by seeing a huge difference in the accuracy, we experimented with trigrams and 4-grams as well. We obtained our best results by using n-grams in the range of (1,4). Table 6 shows the results of using SVMs with n-grams.

TABLE 6. RESULTS FOR SVM USING COUNTVECTORIZER AND N-GRAMS

Category	Precision	Recall	F1-Score
hockey	0.98	0.98	0.98
movies	0.98	0.99	0.99
nba	0.98	0.97	0.97
news	0.92	0.91	0.92
nfl	0.97	0.97	0.97
politics	0.94	0.93	0.94
soccer	0.98	0.98	0.98
worldnews	0.94	0.96	0.95
avg/total	0.96	0.96	0.96

3.4. Naive Bayes with Scikit-Learn

While experimenting with our Naive Bayes implementation, we decided to compare the performance with Scikit-learn implementation of Naive Bayes and is discussed in the section 4. As Naive Bayes is known for decent predictions in the domain of text classification, we went forward to try and predict with it. We experimented using the following different features with CountVectorizer:

1. Using n-grams in the ranges (1,2),(1,3),(1,4) and (1,5).
2. Using minimum document frequency and maximum document frequency values.

3. Using maximum number of features value.
4. Using smoothing parameter for Multinomial Naive Bayes.

We got our best results with the features- n-grams in the range of (1,4), alpha (smoothing parameter) with value 0.1 and removing the stopwords from the text. It gave us an accuracy of 97.290 % which we used for our final predictions.

4. Testing and Validation

4.1. Naive Bayes

Other than using the 6-fold cross-validation of our classifier, We compared our implementation with Scikit Learn's Naive Bayes implementation. When Scikit Learn's Naive Bayes is run with CountVectorizer using only one additional feature, i.e. *stopwords* and smoothing parameter with value 0.01, it results in an accuracy of 93.4%. Running our implementation of Naive Bayes with the similar features results in an accuracy of 89.99% i.e. approximately 90%. Getting the results as close to Scikit Learn's gave us confidence that we have implemented it right. Major difference in the accuracy of Scikit Learn's accuracy was observed on using n-grams as a feature. The accuracy increased from 92% to 97%. Due to limited time, we could not integrate n-grams in our implementation.

4.2. Decision Trees

Testing and validation was done by training over 80% of the dataset, and then testing on the remaining 20%, to see if the same results would be achieved as in the k-fold cross validation. Table 7 shows the results on the 20%. As they are very similar to the results obtained in Table 2, it is assumed that the model is consistent and working well.

TABLE 7. TESTING RESULTS

Accuracy	Precision	Recall	F1-Score
70%	71%	70%	70%

5. Discussion

5.1. Naive Bayes

Looking at the results presented in Table 1, we observe that "politics" and "news" has the least accuracy. The reason for the lower recall and accuracy can be considered as presence of similar words in both category. Making common judgement, "politics" can be assumed as a subset of "news". So, these two categories are highly interrelated which is why the classifier struggles to classify these topics properly. Using n-grams might have helped us get a better accuracy than 90%.

5.2. Decision Trees

Table 3 shows the average results after 6-fold cross validation for each category. The decision tree seems to be able to predict certain categories better than others. Given that "news", "politics" and "world news" are related topics, this would explain their low numbers in Table 3. The low results for "hockey", "nba", "nba" and "soccer" may be reasoned the same way, as they are all related to sports. The "movies" category did significantly better than the average. As it would seem that it is distinct from the other categories, the decision tree made fewer errors when trying to classify conversations about movies.

A technique that could have been applied to decision trees would have been post-pruning, and removing nodes that were formed with a minimal information gain. This may would have helped with overfitting.

The advantages of using decision trees would be that it is fairly simple to implement. It only relies on building trees, using information gain and entropy to determine the best split on each node. Given that the vectors used are made of binary values, there are only two categories to test per column, 0 and 1. However, finding the split is expensive in the amount of operations it must do: at every node, it must go through every single column, find the information gain and then split the entire remaining data set into 2 sets. This is repeated over all nodes, and the tree in this case is made up of hundreds of nodes and leaves, taking lot of computation time.

6. Statement of Contributions

The work was equally distributed amongst all the team members. All participated equally in deciding the preprocessing of data and features' selection. Nicolas implemented the decision trees algorithm and Inderjot worked on implementing Naive Bayes classifier. SVM with Doc2vec is implemented by Senjuti whereas Inderjot implemented the SVM with countvectorizer. Also, Inderjot implemented Scikit-learn Naive Bayes which we used to make our final predictions on Kaggle. We all contributed to the submissions made on Kaggle and every one wrote their part of the report.

7. Conclusion

Our best results were obtained using Naive Bayes with smoothing. This is not surprising considering that the text features are mostly sparse and a linear kernel does a good job at finding a optimal separating hyperplane between them.

Due to time and space constraints, we were not able to refine our feature selection beyond those demonstrated in this paper. Alternate features we could have considered might be part-of-speech counts or going down a different route altogether and choosing a dimensionality reduction technique over feature selection such as the Latent Dirichlet allocation which is tailored for bag-of-words representations.

Alternate supervised Machine Learning techniques which have been previously shown to yield good results for text data are Bayesian networks, Conditional Random Fields and Neural Networks.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.