Reddit Comments Classification - Kaggle Competition 2019 The Young Engineers

Arnold Kokoroko
Computer Science and Operations Research
Université de Montréal
Montréal, Qc
arnold.kokoroko@umontreal.ca

Ilyas Amaniss
Computer Science and Operations Research
Université de Montréal
Montréal, Qc
ilyas.amaniss@umontreal.ca

1 Introduction

For this Kaggle competition, 70 000 reddit comments each labeled with one of the 20 subreddits were available to train different models and algorithms in order to obtain the best text classifier [1]. Three models were selected by our team, the Young Engineers, to partake in this competition: a Naive Bayes classifier with additional Laplace smoothing, a multilayer perceptron and a support vector machine. This report summarizes the training and optimization process of the three models and the comparison of their performance.

2 Feature Design

In order to make use of the raw data consisting of various sentences made of words, web links, numerical digits, emojis made with punctuation, and more, many preprocessing steps were necessary in order to extract meaningful features from the reddit comments. Prior to training the models, the sentences were first tokenized, then the stop words were removed from the dataset and finally a stemming process was applied to obtain the roots of the words. Using this processed dataset, interesting features were then captured using a bag-of-words representation where the most popular words in the whole text corpus were saved. Here, we describe each of the methods we used, however it is important to note that their efficiency also depended of the learning algorithm used.

2.1 Tokenization

Tokenization is the process of breaking a document down into words, punctuation marks and numerical digits. It is an important step which will ease the next steps of the preprocessing. So, using the Natural Language Toolkit (NLTK) for tokenization available in python, each reddit comment was divided into individual strings. Then, considering that punctuation were not valuable to the text classification task, we made the design choice to remove them from our datasets. Finally, to create more consistency among the different words, all strings were rewritten in lower case. This was to prevent the same word from being represented as two different words because of an uppercase.

2.2 Stop words

Additionally, to extract the most important features we decided to removed stop words and all single characters. Stop words are words which usually do not add important significance to the sentences or are words which are very common in a language. Therefore, using NLTK's list of stop words which contained for instance "a", "an", "the", "it", "am", all stop words were removed, thus also reducing the number of different words to consider during training.

2.3 Stemming

The stemming process allows to further reduce the quantity of words to be considered. Indeed, stemming a word is removing the suffixes or prefixes of this word. For instance, "connections", "connected", "connecting" and "connection" would all be transformed to the word "connect". For this Kaggle competition, it is not important to consider the singular and plural form of the same word as two different instances. The stemming process was used in most learning algorithms we selected except for the multilayer perceptron which surprisingly performed better without using stemming. However, generally having the root of the word seemed to hold meaningful information to classify the reddit comment.
We used the NLTK's PorterStemmer library to reduce the words to their abbreviated form.

2.4 Bag-of-words

Then, we have the backbone of our feature extraction: The bag-of-words. A bag-of-words representation will contain, for each datapoint, the number of times a certain word w_i, chosen to be part of the vocabulary, occurs within the reddit comment.
An important step with using the bag-of-words is constructing its vocabulary. We decided to try two methodologies: The first bag-of-words model would be based on the vocabulary created with the most frequent words in the reddit comments, while the second bag-of-words model would be based on term frequency-inverse document frequency (TF-IDF).

2.4.1 Vocabulary: Highest Frequency

When using the highest frequencies, we decided to add to the vocabulary the most popular words of each label. The idea behind this being that popular words are more likely to hold information than words that are rarely used. To do so, we clustered all the reddit comments into a single text corpus and counted the occurrence of each word. The words were then sorted in decreasing order of occurrence to find the most popular words in the text corpus. The vocabulary was then constructed with a range of 40,000 to 70,000 words.

2.4.2 Vocabulary: Term Frequency-Inverse Document Frequency

TF-IDF is a measure of how concentrated the occurrences of a given word is inside a document in a corpus [2]. Consider the term frequency TF_{ij} as:

$$TF_{ij} = \frac{f_{ij}}{max_k f_{kj}}$$

where f_{ij} is the number of occurrence of the word i in the document j and $max_k f_{kj}$ is the maximum occurrence of any words in this same document [2].
Also, consider the inverse document frequency (IDF) defined as:

$$IDF_i = log_2 \frac{N}{n_i}$$

where N is the total number of documents in the corpus and n_i the number of documents where the word i appears [2]. Then, the TF-IDF score of term i in the document j can be described as:

$$TF_{ij} \times IDF_i$$

The library scikit-learn was used in order to compute the TF-IDF of the words in the text. One of the reasons we used TF-IDF is the implicit normalization of the data by reflecting how relevant a term is in the text corpus. This would be done by offsetting the word's total number of occurrences with the number of documents it appears into [3]. It would also reduce the effect of the disparity of a document's length when considering the frequency of words' occurrence relative to each other. The number of words used using TF-IDF was 40,000 to 50,000.

3 Algorithms

3.1 Naive Bayes

The first method we used and our performance baseline was the Naïve Bayes classifier. Based on the Bayes' Theorem, a Naïve Bayes classifier will compare the conditional probability of each class based

on the chosen features and will return the class with the highest probability. Given its assumption of independence between the features of the classes, the Naïve Bayes classifier was our fastest method to train. Moreover, Laplace smoothing was used to improve the results.

3.2 Support Vector Machine

Then, we used the support vector machine (SVM) which are designed to handle high-dimensional data by finding a separating hyperplane that maximizes the margin between two or multiple classes [4] [5]. We found the linear SVM to be more efficient than the kernelized SVM when using our preprocessing methods, therefore we chose the linear version. To classify the data, two methods were used. On one hand, we had the *one versus all* method where N different classifiers were built, each specialized in categorizing one of the classes — N being the number of subreddits. On the other hand, we had the one versus one method where $\frac{N(N-1)}{2}$ classifiers were constructed each representing a pair of labels and contributing to the classification vote.

3.3 Multilayer Perceptron

Finally, our third method was the multilayer perceptron (MLP) classifier. The Multilayer perceptron is an artificial neural network composed of an input layer, one or multiple fully connected hidden layers, an output layer, and is trained using backpropagation. In order to limit the capacity of the model and avoid overfitting, we decided to go for a simple network with one hidden layer implemented using Pytorch. The number of neurons is discussed in the results section. The activation function used for the hidden layer was the rectified linear unit (ReLU) and the loss was derived using cross entropy loss. When deriving the loss, we did not have to add a softmax function to the last layer to output the probabilities as it is directly implemented in the `cross_entropy` loss function of Pytorch (see Appendix A). Then, to train the network we used two optimizers: Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam). The learning rate used is also discussed in the results section. Finally, an argmax function was applied to the output to find the predicted classes.

4 Methodology

It is important to note that all the methods described in this section implemented part or all the preprocessing methods described in section 2 such as the bag-of-words feature. They were a crucial part of the success of each models. For more information, please refer to section 2.

4.1 Training and validation sets

In order to fine-tune the hyperparameters of our models or to compare different optimization techniques when training the classifiers, the 70,000 data points were separated in the following way: 80% were allocated for the training set and 20% for the validation set. Our test set was the test data provided for the competition and submitted to Kaggle. Before each training, the data points were randomly shuffled using the `train_test_plit` method from scikit-learn. Once the best parameters were found, a final model was retrained on all the datapoints then submitted to Kaggle.

4.2 Naive Bayes

Our methods to optimize the performance of the Naive Bayes classifier was to find the best combination and number of words in the vocabulary — and to find the best Laplace smoothing value which can be very useful when working with large feature matrices as the chances of having multiple entries equal to zero is high making the the whole conditional probability equal to 0. However, even when using Laplace smoothing, we still had the problem of having to compute very small probabilities. Therefore our solution instead was to use the log probabilities to compute an equivalent result. This greatly improved the computational accuracy and the results. Please refer to the section 5.1 for the specific results of the Naive Bayes classifier.

4.3 Support vector machine

Concerning the support vector machine, the main idea was to find the best hyperparameter for the penalty term C, which controls the tolerance to errors of misclassifying a point when searching for the best margin. Indeed, having a very high value for C is equivalent to training an SVM with hard margin, which is too restrictive for this task, while having a very small value would limit the impact of misclassifying violating points. Thus, using the validation set, multiple linear SVM classifiers with different hyperparameter values for C were trained using both the *one versus one* and *the one versus all* methods to find the most efficient one. The results are summarized in section 5.2.

4.4 Multilayer Perceptron

Among the three models, the MLP had the most hyperparameters to optimize: the number of hidden layers (1,2,3), the number of neurons in each hidden layers (200, 500, 1000), the activation function (sigmoid, ReLU, tanh), the learning rate (10^{-2}, 10^{-3}, 10^{-4}), the optimizer (SGD, Adam), the regularization method (L1, L2) and weight decay, and the mini batch size (32, 64, 128). Due to time and computation restrictions based on the large dataset, we arbitrarily decided to keep one hidden layer and use a mini batch size of at least 64. For the other hyperparameters, we performed a grid search to find the optimal values. A table with the most interesting results can be found in section 5.3.

Furthermore, we realised during training that the model was overfitting very rapidly. Therefore, to increase the regularization we decided to use a dropout layer to prevent the neural networks from overfitting [6]. As such, we added a dropout rate of 0.5 to both the input layer and hidden layer, which greatly improved the results. On top of that, we also used early stopping with the validation set to obtain the best bias-variance trade-off. One interesting side note is that the stemming part of the preprocessing was avoided when using the multilayer perceptron. The model seemed to perform better and extract more meaningful features when working with the full length of each word even though there would be more words to analyse.

5 Results

During the competition, multiple classifiers were tested to asses their performance on the given task. Among all of them, three gave us the best results: The Naive Bayes Classifier, the Support-Vector Machine and the Multilayer Perceptrons. It is also useful to mention that we used the linear version of the Support-Vector Machine. Various classifiers like the logistic regression, the decision trees and random forest, were also attempted and performed decently in the task, but we will stick to the analysis of the 3 aforementioned classifiers.

5.1 Naive Bayes

In the case of the Naive Bayes classifier, the model was built using `MultinomialNB` from scikit-learn. Its validation accuracy can be found in Figure 1. As we can see, the difference between using the words frequency for the vocabulary and TF-IDF is rather important, varying from 1.5 to 2%. In both cases, the optimal number of words (50,000 and 70,000) in the vocabulary was found empirically. Furthermore, the hyperparameter for the Laplace smoothing also had a large impact with an improvement of almost 1% when using a Laplace smoothing of 0.2. This was the second best performing model with a success rate of 57,17% on the validation set.

		Hyperparameters			Accuracy (%)	
Model	Library	Vocabulary	# words	Laplace	Training	Validation
Naïve Bayes	Sklearn	Word Freq	70,000	0.2	-	55.20%
Naïve Bayes	Sklearn	Word Freq	70,000	0.5	-	55.28%
Naïve Bayes	Sklearn	Word Freq	70,000	1	-	54.50%
Naïve Bayes	**Sklearn**	**TF-IDF**	**50,000**	**0.2**	**-**	**57.17%**
Naïve Bayes	Sklearn	TF-IDF	50,000	0.5	-	56.91%
Naïve Bayes	Sklearn	TF-IDF	50,000	1	-	56.40%

Figure 1: Results of Hyperparameters for the Naive Bayes Classifier

5.2 Support Vector Machine

For the SVM, the model was built using `LinearSVC` also from scikit-learn. In this case, the results were greatly improved when using TF-IDF versus the words frequency as we can see in Figure 2. We saw an improvement up to 5.5%. This shows that the normalization of the data with the relevance of each word improves the classifying task. Additionally, the model was slighlty more efficient when using *one versus all* than when using *one versus one*. This was the third best performing model with a success rate of 56.63%.

		Hyperparameters				Accuracy (%)	
Model	Library	Vocabulary	# words	C	MultiClass	Training	Validation
Linear SVM	Sklearn	Word Freq	70,000	0.25	1 vs 1	-	52.10%
Linear SVM	Sklearn	Word Freq	70,000	0.25	1 vs All	-	52.31%
Linear SVM	Sklearn	Word Freq	70,000	1	1 vs 1	-	48.56%
Linear SVM	Sklearn	Word Freq	70,000	1	1 vs All	-	49.13%
Linear SVM	Sklearn	TF-IDF	50,000	0.25	1 vs 1	-	56.25%
Linear SVM	**Sklearn**	**TF-IDF**	**50,000**	**0.25**	**1 vs All**	**-**	**56.63%**
Linear SVM	Sklearn	TF-IDF	50,000	1	1 vs 1	-	54.54%
Linear SVM	Sklearn	TF-IDF	50,000	1	1 vs All	-	54.84%

Figure 2: Results of Hyperparameters for the Linear SVM

5.3 Multilayer Perceptron

Lastly, the MLP classifier was built using the various functions from Pytorch and scikit-learn to handle the data. After analyzing the results from the Naive Bayes, we decided to focus on using the TF-IDF for the vocabulary with 50,000 features. A table of the most interesting results after performing a grid search of the hyperparameters can be found in Figure 3. As we can see, the Adam optimizer was generally more efficient than the SGD optimizer when using a small learning rate to avoid overshooting the local minimum when doing gradient descent. Also, a key factor for the increase in efficiency was the use of dropout. Even though the training accuracy dropped significantly when using dropout, the overall results were better. This is because dropout "drops" units during the training phase and forces the neural network to learn more robust features and avoids overfitting [6]. Another way to increase regularization was to use early stopping. As we can see in Figure 4, the validation accuracy usually starts to decrease after 20 epochs and the model begins to overfit. Therefore, the number of epochs we decided to train the model during testing was 17.

Other tests not featured in the table are the activation functions: ReLU, Sigmoid and Tanh. After running our tests, the ReLU activation was clearly the most efficient between the three activation functions for this task. Moreover, we realised that using only 1 hidden layer, the number of neurons varying between 400 and 600 did not have a clear impact on the results and having 1000 neurons significantly increased the training time. Therefore we decided to keep this number to 500.

The MLP was our best performing model with a validation accuracy of 59.83% and a final best private score on Kaggle of 59,695% using the hyperparameters featured in Figure 3.

		Hyperparameters								Accuracy (%)	
Model	Library	Vocabulary	# words	# Neurons	Activation	Learning	Optimizer	Mini batch	Dropout	Training	Validation
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-03	SGD	100	0	73.21%	57.12%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-03	Adam	100	0	79.15%	55.45%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-04	SGD	100	0	62.21%	49.78%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-04	Adam	100	0	75.61%	58.84%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-03	SGD	100	0.5	65.34%	58.43%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-03	Adam	100	0.5	67.17%	57.04%
MLP	Pytorch	TF-IDF	50,000	500	ReLU	1.00E-04	SGD	100	0.5	52.63%	54.99%
MLP	**Pytorch**	**TF-IDF**	**50,000**	**500**	**ReLU**	**1.00E-04**	**Adam**	**100**	**0.5**	**62.14%**	**59.83%**

Figure 3: Results of Hyperparameters for the Multilayer Perceptron Classifier

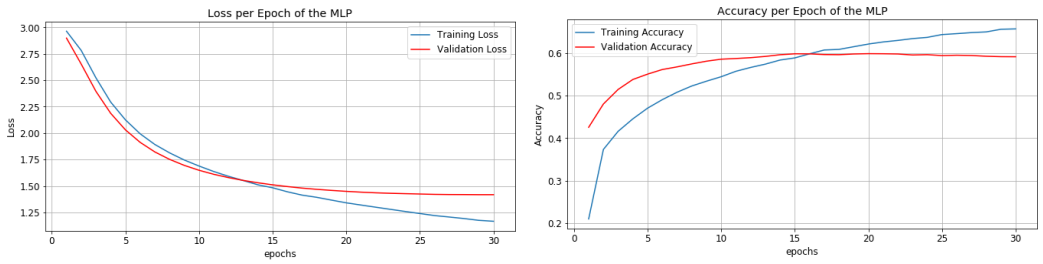

Figure 4: Results of the Loss per Epoch (left) and the Accuracy per Epoch (right)

6 Discussion

6.1 Pros

Most of the effort and time spent on this project was put towards the preprocessing of the given data. This choice of focusing on the preprocessing was found very useful since after training and testing different models, we understood that the the performance of the models was mainly based on how the dataset was processed and less about using fancy models. Also, by starting with a simple model such as the Naive Bayes, we were able to have a functional, even though not perfect, classifier early in the competition. This allowed us to have at least one full iteration of designing, developing and testing a classifier where we learned a lot about the dataset and the task in hand. Indeed, this made the development of the two other models, the SVM and MLP, more efficient and straightforward. Finally, by limiting the number of hyperparameters to fine-tune for each of the models, we were able to get a better understanding of how each of those hyperparameters impacted the classifier being trained.

6.2 Cons

However, by only focusing on a limited selection of hyperparameters, we may have missed some hyperparameters which may have improved the overall results of the classifiers. However, due to the given time frame, it was a necessary decision to take.
Also, working with the entire dataset slowed the training and testing process for each model and thus limited us in our development efficiency.

6.3 Further improvements

Other ideas which could be interesting to explore are the use of Global Vectors for Word Representation (GloVe) or bagging. GloVe is an unsupervised learning algorithm which represents words as vectors [7]. This allows interesting operations such as determining how different or how similar one word is to another by calculating the distance from each other. This could be of use for this text classification competition.
Another interesting idea would be to use bagging by generating different datasets using bootstraping and training different models with them. Then, the final classifier would consist of a majority vote of the models. This is the main concept behind ensemble methods.

7 Acknowledgement

We certify that this report is our own work, based on our personal study and research, and that we have acknowledged all material and sources used.

References

[1] Kaggle, "Classification de commentaires reddit." https://www.kaggle.com/c/ift3395-ift6390-reddit-comments. Accessed: 2019-11-14.

[2] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[3] Wikipedia, "Tf–idf." https://en.wikipedia.org/wiki/Tf\T1\textendashidf. Accessed: 2019-11-14.

[4] B. Ghaddar and J. Naoum-Sawaya, "High dimensional data classification and feature selection using support vector machines," *European Journal of Operational Research*, vol. 265, 09 2017.

[5] C.-F. Lin and S.-D. Wang, "Fuzzy support vector machines," *IEEE Transactions on Neural Networks*, vol. 13, pp. 464–471, 03 2002.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[7] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

Appendix A. Cross-Entropy Loss

```
def cross_entropy(input, target, weight=None, size_average=None, ignore_index=-100,
                  reduce=None, reduction='mean'):
    # type: (Tensor, Tensor, Optional[Tensor], Optional[bool], int, Optional[bool], str) -> Tensor
    r"""This criterion combines `log_softmax` and `nll_loss` in a single
    function.

    See :class:`~torch.nn.CrossEntropyLoss` for details.

    Args:
        input (Tensor) : :math:`(N, C)` where `C = number of classes` or :math:`(N, C, H, W)`
            in case of 2D Loss, or :math:`(N, C, d_1, d_2, ..., d_K)` where :math:`K \geq 1`
            in the case of K-dimensional loss.
        target (Tensor) : :math:`(N)` where each value is :math:`0 \leq \text{targets}[i] \leq C-1`,
            or :math:`(N, d_1, d_2, ..., d_K)` where :math:`K \geq 1` for
            K-dimensional loss.
        weight (Tensor, optional): a manual rescaling weight given to each
            class. If given, has to be a Tensor of size `C`
        size_average (bool, optional): Deprecated (see :attr:`reduction`). By default,
            the losses are averaged over each loss element in the batch. Note that for
            some losses, there multiple elements per sample. If the field :attr:`size_average`
            is set to ``False``, the losses are instead summed for each minibatch. Ignored
            when reduce is ``False``. Default: ``True``
        ignore_index (int, optional): Specifies a target value that is ignored
            and does not contribute to the input gradient. When :attr:`size_average` is
            ``True``, the loss is averaged over non-ignored targets. Default: -100
        reduce (bool, optional): Deprecated (see :attr:`reduction`). By default, the
            losses are averaged or summed over observations for each minibatch depending
            on :attr:`size_average`. When :attr:`reduce` is ``False``, returns a loss per
            batch element instead and ignores :attr:`size_average`. Default: ``True``
        reduction (string, optional): Specifies the reduction to apply to the output:
            ```'none``` | ```'mean``` | ```'sum```. ```'none```: no reduction will be applied,
            ```'mean```: the sum of the output will be divided by the number of
            elements in the output, ```'sum```: the output will be summed. Note: :attr:`size_average`
            and :attr:`reduce` are in the process of being deprecated, and in the meantime,
            specifying either of those two args will override :attr:`reduction`. Default: ```'mean```

    Examples::

        >>> input = torch.randn(3, 5, requires_grad=True)
        >>> target = torch.randint(5, (3,), dtype=torch.int64)
        >>> loss = F.cross_entropy(input, target)
        >>> loss.backward()
    """
    if size_average is not None or reduce is not None:
        reduction = _Reduction.legacy_get_string(size_average, reduce)
    return nll_loss(log_softmax(input, 1), target, weight, None, ignore_index, None, reduction)
```