Subreddit posts Classifier

October 2, 2021

Abstract

In this project, different Machine Learning Classifiers have been implemented to process text from Reddit and classify each post of the data. Two different Naive Bayes algorithms have been implemented from scratch. In addition, several SciKit-Learn based Machine Learning algorithms have also been used. All these implementations have been compared in terms of accuracy and running time by changing the principal hyperparameters of each one. It is concluded that the best algorithm is Logistic Regression for the given data.

1 Introduction

In this report, we describe an end-to-end project to implement and optimize text classifiers. Two different versions of Naive Bayes classifiers are implemented from scratch. The results obtained with these are then compared with different models from SciKit-Learn [5].

The models are used to classify posts from the website Reddit [3] and predict which of eight Subreddits were they posted on. The method proposed includes the classical steps of a text processing project: data acquisition, pre-processing texts, converting a collection of text documents to a matrix of token counts, model training, comparison between models, and data visualization. We aim at getting acquainted with each of these steps while looking for a model that predicts the proposed data with the highest accuracy possible, while maintaining training and test times low.

Firstly, we propose two different models for the Naive Bayes classifier, both exploiting the Bag of Words model (see [2]). The first model, Multinomial Naive Bayes, utilizes the Term Frequency of an attribute, as how many time a word occurs in a document. The second model, Multi-variate Bernoulli Naive Bayes, utilizes the Term Frequency in binarized fashion, as whether a word occurs in a particular document or not. Then both models compute the maximum-likelihood estimate based on the training data to estimate the class-conditional probabilities in the multinomial model.

Secondly, we also propose the computation of predictions through a some of the most popular Machine Learning models implemented in SciKit-Learn: Logistic Regression, and Support Vector Machine and Random Forest. For each of them, we test different hyperparameters combining the use of pipelining and cross validation of the training set. Analysis of accuracy, running times, and different setups are conducted to arrive to an optimal versions of the models suggested in this report.

2 Dataset

The dataset used in this project is a group of texts gathered from posts and comments of eight different Subreddits of the website Reddit. Each text belong to one of eight different classes, corresponding to each of the Subreddits:

rpg : Subreddit for tabletop role playing games

anime : Subreddit for discussing Japanese animation

datascience : Subreddit for discussing matters related to data science, including machine learning

hardware : Computer hardware news and discussions

cars : Discussions and news about cars

gamernews : Discussing video game related news.

gamedev : Subreddit for video game developers

computers : Subreddit for discussing anything about computers.

Each line on the *csv* file provided for training the models includes two fields: text of a post/comment, and the Subreddit it was posted on. The corresponding histogram of occurrences of each target can be seen in figure 1. A second *csv* file is given for testing the models produced in this report, on which no target label is given.

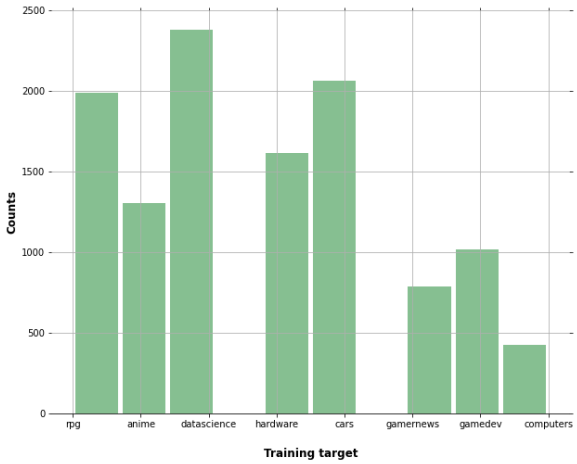

Figure 1: Histogram of training targets of the data.

3 Proposed approach

3.1 Text cleaning

In order to prepare the text for the Machine Learning models to be implemented we proceeded with different techniques of text cleaning:

General cleaning : Certain undesired symbols (such as @,|,(,)) were deleted or replaced (for a thorough list see the files attached in the Appendix). All the text was converted to lower case. Stop words such as "I" and "and", which likely like of any weight on the final decision, were deleted.

Stemming and Lemmatizing : Words with simmilar roots were replaced by the stem, or changed to a common generic word (like conjugated verbs being changed to their ifinitive counterpart).

Using the class CountVectorizer() from SciKit-Learn we transformed the clean text to sparse matrix containing the TF-IDF values of each feature. Different numbers of n-grams were tested. The class Normalizer() from SciKit-Learn was employed afterwards, to normalize the resulting feature vectors by their mean and standard deviation, which has proven to increase accuracy for some of the machine learning algorithms to be used.

3.2 Models implementation

3.2.1 Two models of Naive Bayes Classifiers

In this project, we develop two different versions of the Naive Bayes Classifier (see [1]). The corresponding implementations can be found in Appendix. Here we proceed to describe the main equations on which they are based.

Let M be the number of target classes, N the number of observations in the training set and P the number of words in the corpus obtained from the training set. The data matrix $\mathbf{x_j}$ contain the counts of each attribute (word or n-gram) in each instance of the training set. The prior probability of each class can be calculated with the following expression:

$$\Pr(c_i) = \frac{\#\mathrm{c_i}}{\sum_{i=0}^{\mathrm{M}-1} \#\mathrm{c_i}} \tag{1}$$

For the model of Multinomial Naive Bayes, the class-conditional probabilities of each attributes is calculated according to:

$$\Pr\left(\mathbf{x_j}|c_i\right) = \frac{\sum_{i=0}^{\mathrm{N}-1}\left(\mathbf{x_j}|c_i\right) + \alpha}{\sum_{i=0}^{\mathrm{P}-1}\sum_{j=0}^{\mathrm{N}-1}\left(\mathbf{x_j}|c_i\right) + P \times \alpha} \tag{2}$$

were α is the smoothing parameter with $\alpha < 1$ for Lidstone smoothing, and $\alpha = 1$ for Laplace smoothing (see [2]). The posterior probabilities of $\mathbf{x_k}$ are calculated:

$$\Pr\left(c_i|\mathbf{x_k}\right) = \log\left(\Pr(c_i)\right) + \log\left(\Pr\left(\mathbf{x_k}|c_i\right)\right) \times \mathbf{x_k} \tag{3}$$

and the target class can be found with:

$$\hat{c} = \underset{i}{\operatorname{argmax}} \Pr\left(c_i|\mathbf{x_k}\right) \tag{4}$$

For the model Multi-variate Bernoulli Naive Bayes, the data matrix is first binarized, then the class-conditional probabilities of each attributes is calculated according to

$$\Pr\left(\mathbf{x_j}|c_i\right) = \frac{\sum_{i=0}^{\mathrm{N}-1}\left(\mathbf{x_j}|c_i\right) + \alpha}{P + 2 \times \alpha} \tag{5}$$

and the corresponding posterior probabilities and target classes of $\mathbf{x_k}$ are obtained with

$$\Pr\left(c_i|\mathbf{x_k}\right) = \log\left(\Pr(c_i)\right) + \log\left(\Pr\left(\mathbf{x_k}|c_i\right)\right) \times \mathbf{x_k} + \log\left(1 - \Pr\left(\mathbf{x_k}|c_i\right)\right) \times \left(1 - \mathbf{x_k}\right) \tag{6}$$

and the target class can be found with:

$$\hat{c} = \underset{i}{\operatorname{argmax}} \Pr\left(c_i|\mathbf{x_j}\right) \tag{7}$$

3.2.2 Other models from SciKit-Learn

Besides the Naive Bayes models implemented from scratch, we also tested and optimized other common machine learning models implemented with SciKit-Learn. For each model, a grid search was conducted to tune the corresponding hyperparameters and select the model with the highest accuracy for the given dataset. The models used were:

Logistic Regression: This model uses discriminative learning to estimate the probability of a target class given the value of the features by means of a logistic function. It incorporates intrinsically a regularization method whose penalty (l1 or l2) or regularization strength (C) can are the main hyperparameters to be changed.

Support Vector Machine: This model is based in maximizing the gap between samples of different classes when the examples are represented as points in space. Although originally designed for solving linear problems, it can be extended to nonlinear problems using the kernel trick, implicitly mapping the inputs into high-dimensional feature spaces.

Random Forest: This model is based in ensemble learning for classification. It constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes of the individual trees.

3.3 Parameter tunning

For the case of the Naive Bayes models implemented from scratch, a study of accuracy versus smoothing parameter α was conducted in order to obtain the optimal value for α. Also the running time and accuracy behavior was analysed for different number of features used in CountVectorizer, so that an idea could be obtained of how much one gains in accuracy as the complexity of the data model is increased.

For the three different models of SciKit-Learn mentioned in the previous subsection, a process of hyperparameter tunning was conducted using the method of grid search [4] provided by the library of SciKit-Learn. Each of the three models were pre-pipelined with the classes CountVectorizer(), TfidfTransformer() and Normalizer(). For each pipeline the following parameters were varied with GridSearch in order to find the highest accuracy

CountVectorizer(): The parameter ngram-range, which contains the limits between n-grams to study was varied between the value (1,1), (1,2) and (1,3).

TfidfTransformer(): The parameter use-idf was varied between the values True and False.

Normalizer(): No parameter was varied here.

Machine Learning Model: For the case of Logistic Regression, the parameters varied were the value of C and the penalty of the regularization (l1 or l2). For the case of the Support Vector Machine the parameter tuned was as well the strength of regularization C. Finally, for the random forest, we changed the parameter of max-depth of the trees.

4 Results

The results discussed in this report are only a fraction of what is shown in the attached `mini_project_2_team_2`. We refer the reader to this notebook for the complete discussion.

4.1 Analysis of accuracy, running time and smoothing parameter of the Naive Bayes Classifiers implemented from scratch

4.1.1 Accuracy versus different smoothing parameter

Firstly, for the two versions of the Naive Bayes classifier implemented, both Lidstone smoothing and Laplace smoothing is experimented. In both models, α is spaced evenly on a log scale in $[1 \cdot 10^{-3}, 1]$. The training data set is then tested using 5-fold cross-validation. For faster test times, the maximum number of tokens is set to 5000.

a) b)

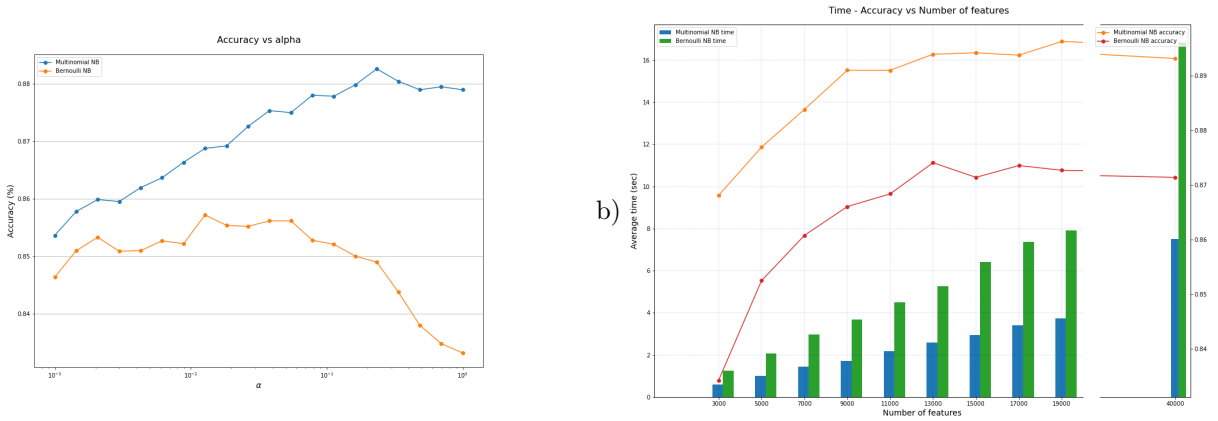

Figure 2: (a) Accuracy versus smoothing parameter. (b) Training time and Accuracy versus number of features.

It can be observed in figure 2(a) that in general, the accuracy of Multinomial NB is higher than Multi-variate NB for any α. While Multinomial NB's accuracy maintain a steady growth as α increases, then peaks at 88.26% at $\alpha = 0.2336$; Multi-variate NB reach its maximum accuracy at 85.72% at $\alpha = 0.0127$ and start to decline afterward.

Secondly, we will investigate the relationship between number of attributes, accuracy and running time. As there are more than 40000 different words in the training dataset, determining a appropriate number of number of attributes can be crucial to find the best model. The data set is again tested using 5-fold cross-validation. The smoothing parameter α is set to 0.1, as it produces relatively good result in both models. It can be observed in figure 2(b) that in general, increasing number of attributes yields higher accuracy, however the training time also grows linearly. At 19000 attributes, the accuracy of Multinomial NB is 89.62%, while Multi-variate NB yields 87.26%. In addition, both training time and accuracy of Multinomial NB is better than Multi-variate NB, therefore we conclude that Multinomial NB is more preferred in this dataset.

Furthermore, increasing number of attributes further than 20000 does not guarantee to improve the accuracy. The result at 40000 attributes is slightly decreasing from peak at 19000 attributes; however the training time is double. As a result, if fast simulation and testing is desired, we strongly recommend users to limit the number of attributes under 20000. On the other hand, users can set the number of attributes to maximize the performance.

Thirdly, we present a brief analysis of the predicting precision and the running times (training + testing times) obtained with two of the SciKit-Learn algorithms (more thorough information regarding these and other SciKit-Learn algorithms used in this project can be found in the Appendix). Figure 3 shows the prediction precision (a) and running time (b) obtained for the Logistic Regression implementation. In this case, the parameters that were changed were the regularization strength C, and the n-grams used in the CountVectorizer function: (1,1) means that only 1-grams were used, while (1,2) means that both 1-grams and 2-grams were used.

a) b)

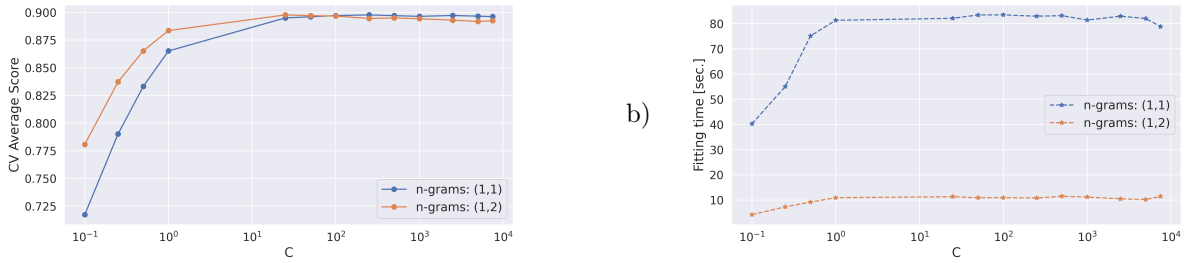

Figure 3: (a) Cross Validation score of the logistic regression algorithm vs hyperparameter Regularization Strength C. (b) Training + Testing time of Logistic Regression algorithm vs hyperparameter Regularization Strength C.

It can be seen that the highest values of precision are attained at $C = 25$ for n-gram = (1,1) and at $C = 200$ for n-gram=(1,2), which also corresponds to the highest accuracy achieved at 90.1% for this algorithm. Also, for small values of C we see that the n-gram=(1,2) results give higher precision, but as C increases the two curves become increasingly close. This is important, since the running time for the cases n-gram=(1,1) is around eight time faster than for the cases n-gram=(1,2) as shown in figure 3(b). Hence, one can obtain the highest accuracy possible with the algorithm while still saving in running time by using the hyperparameter n-gram = (1,1) instead of the much more expensive n-gram=(1,2).

Fourthly, a similar analysis can be observed in figure 4 for the case of the implementation of Support Vector Machine. In this case, also prediction precision (a) and running time (b) have been plotted as functions of the hyperparameters α (regularization strength) and n-gram. In this case, both precision curves are pretty close, peaking at $\alpha \approx 10^5$ for a precision of 90.1%. This demonstrates that the choice of n-gram = (1,1) is much better since it renders similar prediction accuracy at the eight of the running time when compared with n-gram = (1,2).

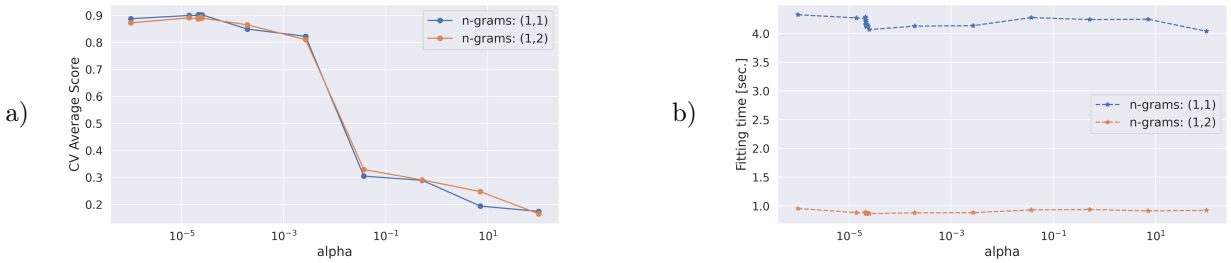

Figure 4: (a) Cross Validation score of the Support Vector Machine algorithm vs hyperparameter Regularization Strength α. (b) Training + Testing time of Support Vector Machine algorithm vs hyperparameter Regularization Strength α.

Finally, it is interesting to analyze the confusion matrix of the results obtained, to understand where the algorithm failed the most. Figure 5 show this confusion matrix for one of the algorithms implemented in the form of a heat map. It can be seen that **gammernews** is often mistaken as **hardware**, and **gammernews** is wrongly recognized the most as the accuracy is only 67.6%. This kind of analysis can help in further improving the feature engineering methods to increase the final accuracy.

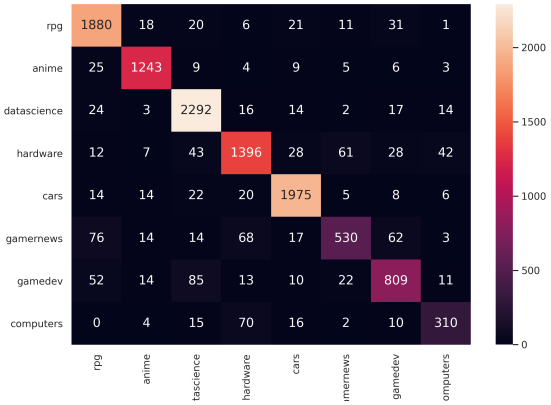

Figure 5: Heat Map of the Confusion Matrix for results obtained with the Support Vector Machine ($\alpha = 10^{-5}$ and n-gram=(1,1).Columns correspond to actual values, rows correspond to predicted values

5 Discussion and Conclusions

In this project, we implemented several machine algorithms to classify a dataset of Subreddits. Text cleaning procedures such as stop word cleaning, lemmatization and stemming were used to improve the final accuracy of the models. For each algorithm implemented (either from scratch in the case of Naive Bayes or from the library SciKit-Learn) several hyperparameters were tested in order to obtain higher precision at smaller running times. Firstly, we observe that the Multinomial NB gave higher accuracy than the Multi-variate NB for the same values of α. Secondly, we also observe how the precision of the predictions with these algorithms increases as the

number of words in the CountVectorizer is increased up to a certain point, on which increasing this number has no further effect in the accuracy. Finally, we could compare several implementations of machine learning models using the SciKit-Learn library. The highest accuracy obtained was of 90.1% for the Logistic Regression classifier, using the parameters n-gram = (1,2) and $C = 25$. We could verify that the running times varied by a factor of 8 as we changed the number of n-grams to be used from n=1 to n=2. In conclusion, the report presents a comprehensive review of the use and testing of a wide range of classical machine learning algorithms in the task of text classification.

References

[1] Andrew McCallum and Kamal Nigam. *A comparison of event models for Naive Bayes text classification.* 1998.

[2] Sebastian Raschka. *Naive Bayes and Text Classification.* URL: `https://sebastianraschka.com/Articles/2014_naive_bayes_1.html`. (accessed: 01.11.2020).

[3] *Reddit.* URL: `https://www.reddit.com/`.

[4] *Scikit-Learn GridSearchCV.* URL: `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`.

[5] *Scikit-Learn Machine Learning in Python.* URL: `https://scikit-learn.org/stable/`.

6 Appendix

Complete notebook can be found in the attached files MiniProject2Team2.ipynb and expanded_model_with_scikit_learn.ipynb.

It follows the code of the two Naive Bayes algorithms implemented from scratch.

```
import numpy as np
import pandas as pd
import time

class MultinomialNB_scratch:
    def __init__(self, alpha=1):
        self.alpha = alpha
    def fit(self, X_train, y_train):
        # m = no. of instances
        # n = no. of features
        m, n = X_train.shape
        self._classes = np.unique(y_train)
        n_classes = len(self._classes)

        # init: Prior & Likelihood
        self._priors = np.zeros(n_classes)
        self._likelihoods = np.zeros((n_classes, n))

        for idx, c in enumerate(self._classes):
            # X_train_c =  matrix containing only instances with class c
            X_train_c = X_train[c == y_train]
            # compute priorprobability of class c as no. of instances of class c
            # over all instances.
            self._priors[idx] = X_train_c.shape[0] / m
            # compute likelihoods with self.alpha for smoothing
            self._likelihoods[idx, :] = ((X_train_c.sum(axis=0)) + self.alpha) / (np.sum(
    X_train_c.sum(axis=0) + self.alpha))
    def predict(self, X_test):
        return [self._predict(x_test) for x_test in X_test]
    def _predict(self, x_test):
        # Calculate posterior for each class
        posteriors = []
        for idx, c in enumerate(self._classes):
            prior_c = np.log(self._priors[idx])
            likelihoods_c = self.calc_likelihood(self._likelihoods[idx,:], x_test)
            posteriors_c = np.sum(likelihoods_c) + prior_c
            posteriors.append(posteriors_c)
        return self._classes[np.argmax(posteriors)]
    def calc_likelihood(self, cls_likeli, x_test):
        return np.log(cls_likeli) * x_test
    def score(self, X_test, y_test):
        y_pred = self.predict(X_test)
        return np.sum(y_pred == y_test)/len(y_test)
```

```
43
44 class BernoulliNB_scratch:
45     def __init__(self, alpha=1, binarize=0.5):
46         self.alpha = alpha
47         self.binarize = binarize
48     def fit(self, X_train, y_train):
49         # m = no. of instances
50         # n = no. of features
51         m, n = X_train.shape
52         self._classes = np.unique(y_train)
53         n_classes = len(self._classes)
54
55         #convert X_train to binary with threshold at
56         X_train = np.where(X_train > self.binarize, 1, 0)
57
58         # init: Prior & Likelihood
59         self._priors = np.zeros(n_classes)
60         self._likelihoods = np.zeros((n_classes, n))
61
62         for idx, c in enumerate(self._classes):
63             # X_train_c =  matrix containing only instances with class c
64             X_train_c = X_train[c == y_train]
65             # compute priorprobability of class c as no. of instances of class c
66             # over all instances.
67             self._priors[idx] = X_train_c.shape[0] / m
68
69             # compute likelihoods with self.alpha for smoothing
70             self._likelihoods[idx, :] = ((X_train_c.sum(axis=0)) + self.alpha) / (X_train_c.
    shape[0] + 2*self.alpha)
71
72     def predict(self, X_test):
73         return [self._predict(x_test) for x_test in X_test]
74     def _predict(self, x_test):
75
76         x_test = np.where(x_test > self.binarize, 1, 0)
77
78         # Calculate posterior for each class
79         posteriors = []
80         for idx, c in enumerate(self._classes):
81             prior_c = np.log(self._priors[idx])
82             likelihoods_c = self.calc_likelihood(self._likelihoods[idx,:], x_test)
83             posteriors_c = np.sum(likelihoods_c) + prior_c
84             posteriors.append(posteriors_c)
85         return self._classes[np.argmax(posteriors)]
86     def calc_likelihood(self, cls_likeli, x_test):
87         return np.log(cls_likeli) * x_test + np.log(1-cls_likeli) * (1-x_test)
88     def score(self, X_test, y_test):
89         y_pred = self.predict(X_test)
90         return np.sum(y_pred == y_test)/len(y_test)
```