

YORK UNIVERSITY

EECS 4412 : DATA MINING

WINTER 2020 (PROJECT REPORT)

Sentimental Analysis on Yelp Online Reviews

Student:

Gian Carlo ALIX

ID number:

214760870

April 1, 2020



I. Introduction

Online reviews are often considered two-fold in terms of the purpose they serve. In the consumer-side perspective, people get some inside information from other reviewers regarding the quality of a service of a particular business or the quality particular product. In the producer-side perspective, businesses may take advantage of people's reviews in order to make adjustments and improvements in the services and products that they offer.

In this project, we aim to build a model that is able to classify customer reviews as either positive, negative, or neutral. This a Natural Language Processing (NLP) task known as **sentiment detection** or **sentimental analysis**. We shall compare the different types of classifiers on some training set, and the classifier with the best performance evaluation (using the *accuracy* metric) on this particular dataset will be the model to be used on the given testing set.

II. The Dataset



The **Yelp Open Dataset**, <https://www.yelp.com/dataset> contains millions of customer reviews (may it be from restaurants, auto services, tourist attractions, hotels, etc.), and a random 70,000 were randomly selected from this large dataset. Eighty percent (80%) of this set will be our training set. And the remaining would be the testing set. In other words, the training set consists of 56,000 online reviews while the remaining 14,000 are in the testing set. Note however that the training set includes a sentiment label **positive**, **negative**, or **netural**, while the testing set do not.

Additionally, to make things easier when we compare the performance of our learning classifiers, the training set with 56,000 reviews was further partitioned into the training set and what is known as a **validation set**. The validation set is there for us to easily decide on the type of learning classifier to use on the test set, as there is no easy way to compare the performance evaluations of our classifier. There is no rule of thumb to use for partitioning the 56,000 reviews; however, we can say that because the test set consists of 14,000 reviews then the validation set must have 14,000 as well. Hence, we can subdivide 56,000 reviews into 42,000 reviews in the training set and 14,000 reviews in the validation set.

In terms of the attributes, the training/validation sets consist of the **text**, **ID**, and **class** attributes; meanwhile, the testing set only has the **text** and the **ID** attributes.

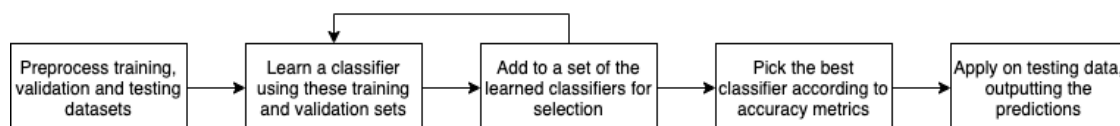







Figure 1: The process of the project, as illustrated above; drawn in <https://draw.io>


III. Data Preprocessing

We can see from Figure 1 the main process of our project and we shall be using this as a guide from now on. This is the beginning of the process and in this stage, we perform **data preprocessing**, where we change the format of the data or convert it into something more usable when we perform the classification tasks later on. There are 6 stages of the data preprocessing and we'll go through each one carefully.

Here is a legend of what the icons symbolize:

-  - A task done in Python; see an accompanying Python script for reference
-  - Credits to M. Porter for the code: <https://tartarus.org/martin/PorterStemmer/>
-  - A task done in Weka; default values are used unless otherwise specified
-  - A task done in Excel; some manual tasks on CSV files done easily in Excel

 **Stage 1:** In this stage, we attempt to do three things. Firstly, we would like to drop the ID attribute of both training and testing sets; the class attribute of the training set should be dropped as well. Thus, only the text attribute is left for both training and testing sets. Secondly, we want to remove any special characters that may be present in each review. We might not be able to remove all but the most common ones such as period, comma, question mark, exclamation point, equal sign, colon, semicolon, dollar sign, pound sign, brackets, parentheses, slashes, plus sign, minus sign, asterisk, ampersand, percent sign, and caret to name a few. And thirdly, for each review then we attempt to remove all stopwords. The stopwords are provided by clicking [here](#), and adding to these are space and numbers from 0 to 9. The entire Python script I wrote can be found in Appendix A of this report, and the outputs should be 2 CSV files (called `train_set_stage1_preprocessed.csv` and `test_set_stage1_preprocessed.csv`) that does the above tasks with a success message printed out that the stage 1 of preprocessing is complete.


 **Stage 2:** In this stage, we apply the Porter Stemming Algorithm (we may use the Python version; see Appendix B), and the script can be seen here:


<https://tartarus.org/martin/PorterStemmer/python.txt>

We have to run the following command in the terminal:

```
python PorterStemmer.py XXXXX
```

where XXXXX here is either one of the output files produced by Stage 1 (however we have to run this twice: one for the training set, and another one for the testing set). Credits again to M. Porter for his word stemming algorithm. We have modified the original version of the algorithm to output the results into a file, instead of displaying it onto a terminal screen. The output after running the script is `stemmed.csv`, but we rename the files to `train_stemmed.csv` and `test_stemmed.csv` to distinguish the two output files.

 **Stage 3:** For this particular stage of the preprocessing, although we have separate training and testing sets already, we may want to combine the two datasets together, in preparation for the `String2WordVector` in the fourth stage of data preprocessing. We shall see how that works in Stage 4 in Weka. For the meantime, we combine the training and testing sets while being careful not to mess the order of the reviews up. The first 56,000 reviews of the resulting `merged_set.csv` are reviews from the training set and the remaining 14,000 reviews are from the testing set. The order of the reviews should have been maintained by the end of this stage. See Appendix C for the code.

 **Stage 4:** Next, we want to use the Data Preprocessing features available in Weka, particular the `String2WordVector`. Now let us first load up the `merged_set.csv` dataset in Weka and we shall notice that it should only have one attribute, the `text` attribute (since, we dropped all other columns in the previous preprocessing stages). Another thing to observe is that this particular attribute is nominal, and in order to use `String2WordVector` filter, then we must first convert this nominal attribute to string attribute using the `NominalToString` filter with default settings. After this conversion, we can then use `String2WordVector` filter, with the following settings as shown in Figure 2.

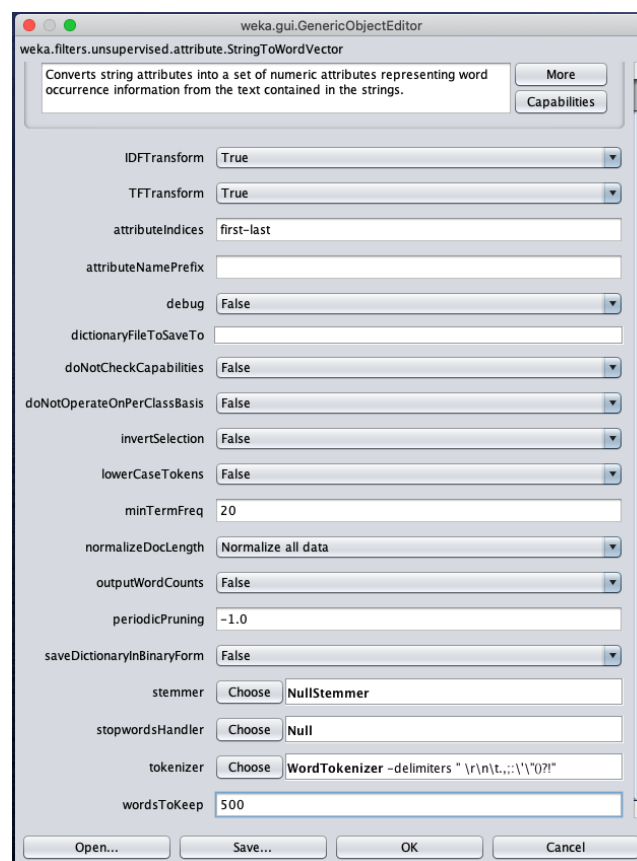




Figure 2: The fourth stage of data preprocessing called `String2WordVector`, using the following settings


Let us cover a bit about the settings that were changed in this filter. The `IDFTransform` and `TFTransform` have been both set to `True` so that the words may be transformed into a vector of numbers, using the $TF-IDF$ (Term Frequency and Inverse Document Frequency) metric. Then we set the `minTermFrequency` to 20, which means that those words in this particular document (reviews in this case) that have appeared less than 20 times are considered infrequent and should be removed. We also set `normalizeDocLength` to `Normalize all data` which should do data normalization. For the stemmer, we did not choose to touch it as we have already finished stemming. For stopwords handler, we also did not change anything as we have already did stop word removal. And then for tokenizer, we keep the same delimiters; as some of them have not yet been used back from Stage 1 Preprocessing. We also choose to keep 500 words, than the default 1000. Learning and training of the model might take time, given the big data set that we have. Hence, we'll use only top 500 of the 1000 words.

We then click "Apply" and after running the filter, we can simply save the file as `s2wv.csv` (for "string2wordvector"). And this file we use for the next stage of preprocessing.

 **Stage 5:** In this stage after converting the string words into numerical vectors according to the $TF-IDF$ metric, then we can now split it back into training and testing sets. This is not so hard to do as the order of the datasets was preserved after Stage 4 of Preprocessing. Furthermore, we can split the training set into two sets of ratio 3:1, where the larger chunk is called the official **training set** and the other being the **validation set**. So in the end of this stage we produce the three official datasets: 42,000 reviews in the training set, 14,000 reviews in the validation set, and 14,000 reviews in the testing set. Also at the end of this stage, we must add back in the `class` attribute in the original training set into this new training set (note that we have to add this additional attribute first before splitting into training and validation sets). Then also add a pseudo-attribute called `class` for the testing set, with randomized labels of **positive**, **negative** and **neutral**. The code can be seen in Appendix D and the output files should be:

- `for_model_training.csv`
- `for_model_validation.csv`
- `for_model_testing.csv`

 **Stage 6a:** Stage 6 is subdivided into two sub-tasks. The files produced in Stage 5 is almost ready for use. However, we'll first convert the CSV files into ARFF files. To do this, we first need to obtain a list of attributes of the datasets. Note that the attributes of all datasets are expected to be the same due to the nature of how we preprocessed the data. We can use the Python script from Appendix E to perform this stage. The output should be a text file, `attributes_list.txt`.

 **Stage 6b:** In Stage 6a, we have obtained the attributes of the datasets. These are actually the `String2WordVectors` attributes. Remember that there was about 500 of them, hence it is not feasible to type each one and format it in the way the ARFF wants us to. And so the Python script from Appendix E should have helped us. If you would notice, the result is written onto a text file and this contains the ARFF headings. So in this stage, simply

remove the headings of each dataset (by either opening Excel or using a regular notepad), and then replace it with the output of `attributes_list.txt`. Finally, replace each of the filename's extension to `.arff`. The filenames should be:

- `for_model_training.arff`
- `for_model_validation.arff`
- `for_model_testing.arff`

You can always choose to do "Save As" instead in order to keep the CSV files if desired.


IV. Methodology and Classifiers

The long process of data preprocessing has finally ended and we can finally start training some models and learning classifiers. Following from Figure 1's process, we opt to select several various learning classifiers and see how well each one performs (training from the training set and then being evaluated at the validation set). And then later on select the best performing algorithm and then give rationale and justifications as to what makes it the best. This selected best will be used to predict the sentiments of the reviews in the testing set. Note again that the current class labels sitting in the testing set right now are merely place-holders. They are only used for ensuring the correctness of syntax in Weka. Any accuracies or performance evaluation there will be ignored.

Classifier	Type	Accuracy
Bayesian Networks (BayesNet)	Bayes	68.6%
Discriminative Multinomial Naïve Bayes (DMNB)	Bayes	81.9%
J48 Model	Trees	74.5%
Logistic (Generalized Linear Model GLM) Model	Function	82.0%
Multiclass Classifier (Baseline Model: Logistic)	Meta	82.2%
Multiclass Classifier Updatable (Baseline Model: Logistic)	Meta	80.8%
Multiclass Classifier Updatable (Baseline Model: SGD)	Meta	80.8%
Multilayer Perceptrons (Neural Networks or NNs)	Function	-
Multinomial Naïve Bayes (MNB)	Bayes	74.0%
Multitext Naïve Bayes (MTNB)	Bayes	68.7%
Naïve Bayes (NB)	Bayes	62.2%
Random Forests (RF)	Trees	79.6%
RIPPER (JRip Model)	Rules	75.4%
Sparse Generator (SG)	Function	20.5%
Support Vector Machine (SVM)	Function	71.9%

Table 1: This table shows a summary of the list of classifiers I have attempted to use for the Yelp Review Sentimental Analysis, along with their respective accuracy metrics.

V. Discussion

 Table 1 shows a summary of the different classifiers I have tried using Weka on the training set (and using the validation set as the supplied test set). Note that there is no particular order of how I tested the classifiers.

Given that there are many classifiers possible for use as seen in Table 1, it can be shown that some of them just do not work at all. Hence, we may use the process of elimination to deduce which model we will be using on our testing set.

Let us get rid first of all the obvious ones:

1. As this is a supervised task, then any unsupervised machine learning algorithm such as *K*-Means Clustering do not work and should not be used.
2. *K*-Nearest Neighbors (KNNs) is a popular classification algorithm, but is not in the table. Why? This is because its type is **Lazy**. Because it is a lazy learner, then it would have trouble classifying data it has never seen before especially when these newly unseen dataset is huge. To test this out for myself, I tried this classifier on the training set with the validation set as the supplied testing set. This resulted into my Weka crashing. Hence, it is infeasible.
3. Another infeasible classifier is Neural Networks (NN) or MLP (Multi-layer Perceptron). Contrary to the popular belief that neural networks are the most effective in text classification, this does not seem to demonstrate this claim. One factor I presume is due to the fact that we are dealing with a very big dataset. I have run this on Weka and actually timed this. I would like to know how long this training takes. Apparently, at fourteen hours and still running, I decided to stop as this is another infeasible classifier for this particular dataset. Hence I decided to take the loss on this classifier.
4. From Table 1, we might notice the accuracies of some classifier being lower than 70%. They are the lowest ones in the chart; it makes sense to eliminate them. Sparse Generator (a function that produces sparse models) performed the worst with 20.5%, which should go. Bayesian Networks also has to go with 68.6%. Now Naïve Bayes is one of the popular classifiers for sentiment analysis, but apparently did not do so well (with a performance accuracy of only 62.2%). Perhaps due to the very large dataset? And then there's also another version of Naïve Bayes, called MultiText, a supposedly good classifier for text classification. Its performance accuracy is 68.7% and the reason for its performance not to do really well is most likely due to the fact that this algorithm works well with actual text (i.e. ignores numerical values). We note that the `String2WordVector` has already converted the text to numerical values, and hence this should give problems for the Naïve Bayes Multitext Classifier. Due to the popularity and effectiveness of the Naïve Bayes class of classifiers, we shall turn to the other classifiers under this class and see if they perform better.

Classifier	Type	Accuracy
Discriminative Multinomial Naïve Bayes (DMNB)	Bayes	81.9%
J48 Model	Trees	74.5%
Logistic (Generalized Linear Model GLM) Model	Function	82.0%
Multiclass Classifier (Baseline Model: Logistic)	Meta	82.2%
Multiclass Classifier Updatable (Baseline Model: Logistic)	Meta	80.8%
Multiclass Classifier Updatable (Baseline Model: SGD)	Meta	80.8%
Multinomial Naïve Bayes (MNB)	Bayes	74.0%
Random Forests (RF)	Trees	79.6%
RIPPER (JRip Model)	Rules	75.4%
Support Vector Machine (SVM)	Function	71.9%

Table 2: This table shows a summary of the remaining possible classifiers for testing after the first round of elimination.

Table 2 then shows a summary of the remaining classifiers that are considered for selection. Now, I'd like to cover a problem that some classifiers face. In some of them above (and also in general), it is possible that some classifiers are not able to predict any data as positive, or perhaps none of the reviews were classified as neutral, or maybe none was predicted as negative. Especially on an imbalanced dataset, if the classifier fails on predicting those data that belong to the minority, then there is an *underfitting* problem. Usually, a misclassification cost is set on imbalanced datasets, however we are bypassing this as (1) hard to set an accurate misclassification cost matrix that would penalize the classifier for classifying a minority-belonging data as one that belongs to the majority class, and (2) some of the classifiers above need not a misclassification cost penalties in order to perform well (the DMNB and logistic models for instance). Usually a meta function in Weka called `CostSensitiveClassifier` is used instead and one baseline function is selected where the misclassification cost is applied onto the prediction model.

In connection to this underfitting problem, we shall see that after applying the above classifiers onto the validation set, then some classifiers are not able to predict any positive review, no negative review, or no neutral. In other words, in a confusion matrix where the actual values are represented by rows and predicted values as columns, then there are some classifiers whose confusion matrix has columns that have all 0s (either positive column, negative, neutral or a combination), which are treated as terrible classifiers. We shall remove them.

Firstly, we can remove the Multiclass Classifier Updatable for both models with baseline of Logistic and Stochastic Gradient Descent (SGD) version. It is interesting to note that both classifiers exhibit a $\sim 80\%$ accuracy. However none of these models were able to classify any neutral reviews in the dataset, and we note that neutral reviews is the minority class in this dataset. So technically, if there was a misclassification cost imposed on these classifiers, then it should have lower accuracy metrics. Hence these classifiers are eliminated from our set. Another interesting remark for these models is that they both produced the same confusion matrix, even though their baseline classifiers are different - one is logistic and the other is

SGD. Here is the common confusion matrix of the models; notice how the neutral column has 0 predictions, implying the models' inability to recognize the minority class.

=== Confusion Matrix ===

a	b	c	<-- classified as
9375	239	0	a = positive
928	1941	0	b = negative
1282	235	0	c = neutral

One possible reason why the Multiclass Classifier Updateable doesn't work well is due to the fact that an updateable classifier means that an error is always being applied during the training process to correct any outputs, which is meant for accuracy to increase. However, this only works for multi-class datasets with 2-class classifiers. Hence, it must be the case that since we have 3 classes (positive, negative and neutral), then this particular model did not work for our dataset. Another Multiclass Classifier is in our list, which is also another meta classifier, but is not an updateable version. We'll see later if it is a good classifier or not.

SVMs, or Support Vector Machines, is another good classifier for text classification. However, it does not seem to be the case for this particular dataset. The essence of SVMs is to find a separating line or hyperplane that would separate datapoints in the search space - which means that since SVMs look for only one best line that separates the data objects, then only 2-class datasets should work. So even if its accuracy in this case is 71.9%, then we eliminate this classifier from our list. It can be shown through the confusion matrix that similar with the Multiclass Updateable Classifiers, SVMs too are unable to classify any neutral-type reviews, as seen in the confusion matrix below:

=== Confusion Matrix ===

a	b	c	<-- classified as
9608	6	0	a = positive
2414	455	0	b = negative
1501	16	0	c = neutral

The "Zero-Column Confusion Matrix" as how I call it is an issue that was faced by such classifiers mentioned (in fact Multitext Naïve Bayes (MTNB) and SparseGenerator Classifiers have this issue as well and was already removed prior due to their low accuracies of less than 70%). Now it doesn't mean that if a particular classifier's confusion matrix's columns (either one of positive, negative, neutral or a combination of such) does not add to zero, then they are good. In fact, we also have to take into consideration how many of the reviews were classified to be positive, negative or neutral by the classifier. We may want to pay attention real closely to the neutral column of the classifier's confusion matrix, since neutral is the minority class in this dataset. Observe that the RIPPER (or JRip) Model has only classified 22 reviews as neutral while the Random Forest (a usually good classifier even in text mining) has only classified 34 neutral reviews, these are very little numbers compared to how many

Classifier	Type	Accuracy
Discriminative Multinomial Naïve Bayes (DMNB)	Bayes	81.9%
J48 Model	Trees	74.5%
Logistic (Generalized Linear Model GLM) Model	Function	82.0%
Multiclass Classifier (Baseline Model: Logistic)	Meta	82.2%
Multinomial Naïve Bayes (MNB)	Bayes	74.0%

Table 3: An updated table showing a summary of the remaining possible classifiers for testing after the second round of elimination.

total actual neutral reviews there are in the validation set (1,517 neutrals). So even if their accuracies are 75.4% and 79.6% respectively and none of the columns of their confusion matrix add to 0, but as the total number of neutrals predicted is very small, this seems to suggest that we should eliminate them in our classifier set and would not be effective prediction models. Below are the confusion matrices:

RIPPER:

=== Confusion Matrix ===

a	b	c	<-- classified as
9163	445	6	a = positive
1486	1380	3	b = negative
1307	197	13	c = neutral

RandomForest:

=== Confusion Matrix ===

a	b	c	<-- classified as
9426	182	6	a = positive
1161	1701	7	b = negative
1312	184	21	c = neutral

We can see from Table 3 an updated table of the remaining classifiers after we have eliminated classifiers according to the confusion matrix. We have five possible classifiers left, and we'll be reducing these into the final top two. So firstly, being in the Bayes family, we can compare (according to accuracy) the Discriminative Multinomial Naïve Bayes Model (DMNB) and the Multinomial Naïve Bayes Model (MNB), and see that DMNB is far more superior than MNB in terms of accuracy, since its accuracy is at 81.9% and MNB is only 74.0%. It should be clear which to pick and which to drop. Furthermore, it should be noted that DMNB is a specialized modified version of MNB with supposedly better accuracy (more on this in a while); hence it should be clear that we can simply drop MNB and rely on the performance of DMNB (at the very end of the day, it's more useful to compare two models that are completely different in nature rather than compare two models of the same type). Next is the Logistic Model (or the Generalized Linear Model or GLM) and the Multiclass Classifier with a Logistic Model as its baseline. Both of them have an accuracy of around $\sim 82\%$, with the metaclassifier having a little bit higher accuracy. We can compare the two since both have the nature of "logistic model". Since the metaclassifier is a classifier that builds upon the baseline classifier, then it makes sense to select the Multiclass Classifier with baseline Logistic instead of the GLM. So we simply drop the GLM as its accuracy is only 82.0% while the Multiclass Classifier has 82.2% accuracy. Finally, we have J48. Apparently it is a good classifier for some datasets, but not this particular dataset. And given that the other two models have more than 80% accuracy, we select those as top two and drop J48 for good.

Before we can compare the top 2 models (i.e. the Discriminative Multinomial Naïve Bayes and the Multiclass Classifier with Logistic Baseline), let us first again summarize our process of elimination when it comes to selecting these top two classifiers:

1. We initially began with about 15 classifiers in our set (that is after we have removed a couple of the most obvious choices).
2. The next criteria to look at was if there was any classifier whose accuracy on the validation to be less than 70%. We remove those and only 10 remained out of the 15.
3. Next is we used the "Zero-Column Confusion Matrix" criteria. This also included all classifiers who has columns in their confusion matrix which is very low (particularly the inability to recognize neutral reviews). This has brought the set down to only 5 classifiers remaining.
4. Select the top two out of the 5 classifiers left. It so happens that two pairs of them belong to the same class of classifiers or at least have the same flavour of model (thus selecting the obvious superior among each pair accuracy-wise). The worst classifier among the five was dropped due to it having the lowest accuracy among the five.

Before we make a final decision with the better classifier between Multiclass Classifier with Logistic Baseline Classifier and the Discriminative Multinomial Naïve Bayes Classifier, we shall give some brief details and descriptions for each classifier, stating some pros and cons and why one should select one and why one should not select it.

Multiclass Classifier with Logistic Baseline

Time taken to test model on supplied test set: 3.49 seconds

=== Summary ===

Correctly Classified Instances	11509	82.2071 %
Incorrectly Classified Instances	2491	17.7929 %
Kappa statistic	0.5895	
Mean absolute error	0.1709	
Root mean squared error	0.2931	
Relative absolute error	53.9382 %	
Root relative squared error	73.6676 %	
Total Number of Instances	14000	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.947	0.343	0.858	0.947	0.900	0.653	0.919	0.958	positive
	0.753	0.066	0.745	0.753	0.749	0.684	0.941	0.829	negative
	0.163	0.020	0.501	0.163	0.247	0.242	0.809	0.350	neutral
Weighted Avg.	0.822	0.251	0.796	0.822	0.798	0.615	0.912	0.866	

=== Confusion Matrix ===

a	b	c	<-- classified as
9102	383	129	a = positive
592	2159	118	b = negative
913	356	248	c = neutral

Beginning with the Multiclass Classifier with Logistic Baseline Classifier, we can say that because we are dealing with a dataset with more than two classes then a multiclass type of classifier would be good. And to further extend this, it's good that its baseline is a logistic model because logistic models are good classifiers that are able to predict (usually just binary classes) but sometimes multi-classes as well. Above, you can see the results after running on the validation set:

Discriminative Multinomial Naïve Bayes Classifier

Here are the results after running on the validation set:

Time taken to test model on supplied test set: 2.96 seconds

=== Summary ===

Correctly Classified Instances	11470	81.9286 %
Incorrectly Classified Instances	2530	18.0714 %
Kappa statistic	0.5837	
Mean absolute error	0.1776	
Root mean squared error	0.2951	
Relative absolute error	56.0563 %	
Root relative squared error	74.179 %	
Total Number of Instances	14000	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.944	0.350	0.855	0.944	0.898	0.644	0.916	0.956	positive
	0.727	0.061	0.754	0.727	0.740	0.675	0.940	0.819	negative
	0.200	0.025	0.492	0.200	0.285	0.265	0.812	0.354	neutral
Weighted Avg.	0.819	0.255	0.795	0.819	0.799	0.609	0.909	0.862	

=== Confusion Matrix ===

a	b	c	<-- classified as
9080	367	167	a = positive
636	2086	147	b = negative
898	315	304	c = neutral

Let us first talk about DMNB a bit. It is a specialized modified version of the MNB (MNB is simply the multinomial version of the common familiar Naïve Bayes Classifier). So basically what happened was NB was specialized into MNB, and MNB specialized into DMNB. Here is a paper by J. Su, J. Sayyad-Shirabad, S. Matwin and J. Huang regarding the DMNB algorithm they have proposed:


<http://www.site.uottawa.ca/~stan/csi5387/DMNB-paper.pdf>

In their results, they have shown that although MNB is simple and beneficial computation-wise, this particular classifier is ineffective compared to other discriminative classifiers. The fact that this classifier maximizes likelihood instead of the conditional likelihood or the accuracy is what makes this classifier perform poorly. The proposed Discriminative Multinomial Naïve Bayes Classifier takes this into consideration and its performance as seen in the results in the paper can compete and outperform state-of-the-art models such as Logistic Classifiers and Support Vector Machines.

Now we may want to mention that the results from the validation class of the DMNB and the MCwLB Classifiers are closely matched (from accuracy, MSE, precision, recall, F-measure, ROC, and any other performance metric). It is hard to simply accept MCwLB and use this model as it only beat DMNB by a few 0.3% in accuracy, which I don't think is the way to go, since it may be possible that it may turn out that selecting DMNB would have been a better predicting model in the testing set. However selecting DMNB seems like I am ignoring the accuracy metric at all. So I decide to look at other factors.

Perhaps we can have a look again at how many positive reviews, negative and neutral were predicted by each classifier. Since the minority class is the neutral class, we will specifically look at this. We can observe that the DMNB is able to classify 304 neutral reviews correctly (total of 618 predictions are neutral) and 248 only for MCwLB (total of 495 predictions are neutral). And we should notice that this is not good for both of these classifiers, as they do not have very good abilities to classify minority-labelled data (correctly). That said, we will take the better among the two of them, which is the DMNB model - because it is able to predict more neutral reviews than the MCwLB classifier). Sure, that the trade off is that we'll be selecting a model that has the lower accuracy, however since the difference is only a measly $\sim 0.3\%$, it can still make sense to give this off, just so we could select the classifier that can predict neutral reviews better. Note again that if a misclassification cost matrix was applied (or the `CostSensitiveCost` Classifier), then it would be possible for MCwLB to have a lower accuracy and do worse than the DMNB classifier, due to the fact that we are penalizing this classifier for not being able to classify more of the minority-type data.

As a result of all our analyses, we can safely say that the Discriminative Multinomial Naïve Bayes Classifier is the model from this set. We select this and apply this to our testing set to obtain the classifier's predictions.

 We get the predictions of the testing set in CSV format. We can use Excel to copy the IDs of the testing data and paste it into a CSV file along with the classifier's predictions and save it as `predictions.csv`.

VI. Conclusions

Classifying Yelp Reviews as either a positive review, a negative or a neutral one can be done through a text classification task known as sentiment analysis. We have seen a number of popular classifiers that are known to be good for text classification; some of which include the Support Vector Machines, the Logistic Classifiers, and Multitext Naïve Bayes. Some of the well-known models such as Neural Networks did not work out in our case due to the massive amount of data in the dataset. The process of elimination was done while comparing which model turned out the best to be used for testing set, and the Discriminative Multinomial Naïve Bayes Classifier turned out to be the best, in terms of accuracy and its ability to predict data that belonged to the minority class.

References

- [1] S. Symeonidis. "5 Things You Need to Know about Sentiment Analysis and Classification." Internet: <https://www.kdnuggets.com/2018/03/5-things-sentiment-analysis-classification.html>, March 2018 [Mar. 22, 2020].
- [2] J. Su, J. Sayyad-Shirabad, S. Matwin and J. Huang. *Discriminative Multinomial Naive Bayes for Text Classification*, Ottawa: University of Ottawa, n.d. [Online]. Available: <http://www.site.uottawa.ca/~stan/csi5387/DMNB-paper.pdf> [Mar. 22, 2020].
- [3] M. Porter. "The Porter Stemming Algorithm." Internet: <https://tartarus.org/martin/PorterStemmer/>, Jan. 2006 [Mar. 21, 2020].
- [4] "Class MultiClassClassifierUpdateable". Internet: <https://weka.sourceforge.io/doc.dev/weka/classifiers/meta/MultiClassClassifierUpdateable.html>, n.d. [Mar. 31, 2020].
- [5] "Class NaiveBayesMultinomial". Internet: <https://javadoc.scijava.org/Weka/weka/classifiers/bayes/NaiveBayesMultinomial.html>, n.d. [Mar. 31, 2020].

Appendix

Appendix A: Stage 1 Data Preprocessing

This takes in the original data from `train3.csv` and `test3.csv`, and then outputs CSV files without special characters and stopwords. The class and IDs have also been removed. This can be run either on the **Jupyter Notebook** in the *Anaconda Navigator*, or using a terminal by typing in the command:

```
python stage1_preprocessing.py
```

which should produce two output CSV files and print a success message for doing so.

```
=====
# EECS 4412 - Data Mining
# Stage 1 Preprocessing: stage1_preprocessing.py
# Author: Gian Carlo Alix

# import necessary libraries
import pandas as pd
import numpy as np
import re

def merge_tokens(lst):
    # use this to merge a list of tokens into a full line sentence
    line = ""
    for token in lst:
        line = line + token + " "
    return line

def remove_stopwords(tokens, stop_list):
    # returns a list without stopwords
    new_tokens = []
    for i in range(len(tokens)):
        if tokens[i].lower() not in stop_list:
            new_tokens.append(tokens[i].lower())
    return new_tokens

def preprocess(sentence, stop_list):
    # split using regex
    regex = "[/\\$()\\[\\]!?!%&*:.=\\^,\\+\\s\\t\\n#]"
    tokens = re.split(regex, sentence)

    # remove stopwords
    new_tokens = remove_stopwords(tokens, stop_list)
```

```
ln = merge_tokens(new_tokens)
return ln

def stage1(df,sw,dataset,drop_cols):
    # drop certain columns
    df = df.drop(drop_cols, axis=1)

    # preprocess here
    for i in range(len(df)):
        df['text'][i] = preprocess(df['text'][i], sw)

    # output to a file
    filename = dataset + "_set_stage1_preprocessed.csv"
    df.to_csv(filename,index=False,header=False)
    print("Stage 1 Preprocessing Complete for " + dataset + "_set completed!")

def main():
    # import the datasets
    df_train = pd.read_csv("data/train3.csv")
    df_test = pd.read_csv("data/test3.csv")
    sw = pd.read_csv("stopwords.csv")

    # to eliminate numbers and empty strings,
    # we treat these as stopwords to remove.
    # create a file called more_stopwords.csv
    list_of_more_stopwords = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '']
    sw_additional = pd.DataFrame({'stopwords':list_of_more_stopwords})
    sw_additional.to_csv("more_stopwords.csv",index=False)

    # merge the stopwords into a single list
    sw = sw.append(sw_additional)

    # stage1 preprocessing
    stage1(df_train, list(sw['stopwords']), "train",['class','ID'])
    stage1(df_test, list(sw['stopwords']), "test",['ID'])

main()
```

=====

Appendix B: Modification of the PorterStemmer

The Porter Stemmer Algorithm was used to take the stemmed words of each of the words in each Yelp Review. One may run the Python script using the command below in the terminal and the output (the stemmed words) are displayed on the screen. However, we can modify the code in order for it to output onto a file instead. We can call the output `stemmed.csv`. We do it once for the training set and another for the validation set. And then simply rename each file as `train_stemmed.csv` and `test_stemmed.csv`.

```
python PorterStemmer.py XXXXX
```

Note that we replace XXXXX here with either `train_set_stage1_preprocessed.csv` or `test_set_stage1_preprocessed.csv`. The text in red below is the added code in the `PorterStemmer.py` script.

```
=====
if __name__ == '__main__':
    p = PorterStemmer()
    output_file = open("stemmed.csv", "w")
    if len(sys.argv) > 1:
        for f in sys.argv[1:]:
            infile = open(f, 'r')
            while 1:
                output = ''
                word = ''
                line = infile.readline()
                if line == '':
                    break
                for c in line:
                    if c.isalpha():
                        word += c.lower()
                    else:
                        if word:
                            output += p.stem(word, 0, len(word)-1)
                            word = ''
                        output += c.lower()
                output_file.write(output)
            infile.close()
    output_file.close()
    print("Success! The text has been stemmed!")
=====
```

Appendix C: Dataset Merger

We can use this Python script to merge the training and testing sets after they have been stemmed using the Porter Stemmer Algorithm. This is used in preparation for the next preprocessing the stage in Weka. This can be run either on the **Jupyter Notebook** in the *Anaconda Navigator*, or using a terminal by typing in the command:

```
python data_merger.py
```

which should produce a single merged CSV file of the two datasets and then display a success message.

```
=====
# EECS 4412 - Data Mining
# Stage 3 Preprocessing: data_merge.py
# Author: Gian Carlo Alix

# import necessary libraries
import pandas as pd

# temporarily merge the training and testing sets
# do not mess up the order; the first 56K rows are train_sets
# while the remaining ones are test_sets
train_stemmed = pd.read_csv("train_stemmed.csv")
test_stemmed = pd.read_csv("test_stemmed.csv")

merged_dataset = train_stemmed.append(test_stemmed)
merged_dataset.to_csv("merged_set.csv", index=False)
print("Datasets merged!")
=====
```

Appendix D: Dataset Splitter

We can use this Python script to split a dataset into training, validation and testing sets. We also ensure that the datasets have the `class` attribute aside from the `text` attribute. For the training and validation sets, this should be available from the original dataset (not so hard to do as order was maintained on the training set before we split it into training and validation sets). For the testing set, note that since the original dataset did not have a class attribute, we have to do a randomize a class label for each review (note that this is only a pseudo-column for the syntactical correctness in Weka's classifiers). Note however that this can only be done after the reviews have undergone **String2WordVector** stage through Weka. We can run this code either on the **Jupyter Notebook** in the *Anaconda Navigator*, or using a terminal by typing in the command:

```
python data_split.py
```

and the outputs are the following files:

- for_model_training.csv
- for_model_validation.csv
- for_model_testing.csv

```
=====
# EECS 4412 - Data Mining
# Stage 5 Preprocessing: data_split.py
# Author: Gian Carlo Alix

# import necessary libraries
import pandas as pd

# split into the training and testing sets
# order was maintained, so the first 56K is the training
# and the remaining 14K is the testing
s2wv = pd.read_csv("s2wv.csv")
training_dataset = s2wv.head(56000)
testing_dataset = s2wv.tail(14000)

# grab the original training dataset to get the class attribute
df = pd.read_csv("data/train3.csv")
training_dataset = pd.concat([training_dataset, df['class']], axis=1)

# define a new column in the testing set called 'class'
# randomize an int(0,1,2) and assign to each review
# then replace (0,1,2) with (positive, negative, neutral)
# note that this only a pseudo-column for the correctness of syntax in Weka
testing_dataset['class'] = np.random.randint(0,3,size=(len(testing_dataset),1))
testing_dataset['class'] = testing_dataset['class'].replace(0, 'negative')
testing_dataset['class'] = testing_dataset['class'].replace(1, 'neutral')
testing_dataset['class'] = testing_dataset['class'].replace(2, 'positive')

# further split the training set into 75% training and 25% validation
validation_dataset = training_dataset.sample(frac=0.25)
training_dataset = training_dataset.drop(validation_dataset.index)

# output the CSV files
training_dataset.to_csv("for_model_training.csv",index=False)
validation_dataset.to_csv("for_model_validation.csv",index=False)
testing_dataset.to_csv("for_model_testing.csv",index=False)
=====
```

Appendix E: String2WordVector Attributes

To finalize our datasets, we need to convert them into ARFF format. Currently in CSV, opening the file (either one of the three: training, validation and testing) in a regular Notepad (instead of Excel), we should see the first line as the attributes of the dataset and the lines afterward are the corresponding values of the datasets (separated by commas). We should note that because we have now converted the datasets into vector from string, then the elements are no longer words of a review but are now numbers. Since ARFF needs the special heading at the top of the comma-separated data, we can ask Python to generate this for us due to the fact that there are about 500 attributes in the dataset (from our **String2WordVector** setting earlier). We can run this code either on the **Jupyter Notebook** in the *Anaconda Navigator*, or using a terminal by typing in the command:

```
python data_attributes.py
```

and this script should output a text file called `attribute_list.txt` with the ARFF headings. We can just use Excel or a regular Notepad to remove the headings of the three datasets `for_model_training.csv`, `for_model_validation.csv` and `for_model_testing.csv` and replace them with the result of `attribute_list.txt`. Finally, make sure to change the extension to `.arff`.

```
=====
# EECS 4412 - Data Mining
# Stage 6a Preprocessing: data_attributes.py
# Author: Gian Carlo Alix

# import necessary libraries
import pandas as pd

# Get the list of attributes
train_set = pd.read_csv("for_model_training.csv")
train_col = list(trainset.columns)

# construct the arff-formatted headings
output_file = open("attribute_list.txt", "w")
output_file.write("@relation\n\n")
for attribute in train_col:
    output = "@attribute " + attribute + " numeric\n"
    output_file.write(output)
output_file.write("@attribute class {positive, negative, neutral}\n\n")
output_file.write("@data")

output_file.close()
=====
```