Justin Lau, David Yei

Professor Muzny

CS 4120

19 April 2021

Final Project Report

For our final project, we decided to conduct sentiment analysis on video game reviews. We believe sentiment analysis for video games is interesting because user ratings, for instance, on a 1 to 5 scale, can generally determine how good or bad a video game is. However, analyzing text can illustrate specific terms or phrases linked to a video game's rating. For example, the words "beautiful", "visual", and "breathtaking" could be associated with positive reviews, and developers can infer that their game is highly praised for its graphics, and may advertise this aspect of the game more. On the other hand, if words like "clunky", "awkward", and "movement" were associated with negative reviews, developers would assume that their game struggles with its controls, and should be addressed in an update. Without text, we could not gain this specific insight at all. As a result, we wanted to conduct our own sentiment analysis experiment to not only reinforce what we have learned in this class, but also to gain insight into a specific application of NLP that interests us.

We underwent a specific process throughout our notebook: importing, visualizing, cleaning, preprocessing, training, tuning, and then evaluating. We used pandas to store and organize our data, and plotly to visualize some trends in the data. We used pre-built models from the sklearn library, such as Multinomial Naive Bayes, Logistic Regression, Random Forest Classifier (RFC), and KNeighborsClassifier. Furthermore, we used certain tools to aid us in preparing and evaluating the data. For example, we used the train/test split method from the sklearn library to split up our features and labels into respective training and testing sets. In addition, we used the accuracy_score, f1_score, precision_score, and recall_score to obtain the respective accuracy, F-1 score, precision, and recall of our models.

We used two datasets in order to conduct our project. The first dataset was downloaded from Kaggle [1]. This dataset was gathered from the popular platform Steam, where users can buy games and write reviews about them. This data contained five columns: the unique id of the review, the game the review was about, the year the review was posted, the text of the review, and the label of the review (1 → Recommended, 0 → Not recommended). We decided to drop the first three columns because we didn't believe those features would contribute to whether a game would be recommended or not. The second dataset we used was from one of our DS3000 classes, taught by Caglar Yildirim in the Fall 2020 semester. The link to the original dataset is here [2]. The dataset also contains scraped reviews from Steam. For the DS3000 class, the professor dropped the game id and number of users who thought the review was helpful columns from the original dataset, and then saved it as a .csv file, which we used for our project. Thus, two columns remained, the review and its label. Together, we have approximately 36,000 data entries, about 29,000 is used for training, and 8,000 for testing.

We took various preprocessing steps in order to prepare our models for training. We did some feature engineering in order to incorporate non-BOW features into our models. For example, for each review we counted the number of exclamation marks, the number of all capital words (such as “HELLO”), as well as the length of the review in words. Furthermore, we preprocessed the text reviews themselves. We converted all the words to lowercase, removed stop words, and also lemmatized the words. When vectorizing the data, all punctuation was removed. Lastly, we converted any unicode characters to ASCII characters.

We used two vectorizers to transform the reviews into numerical data. The first vectorizer is the CountVectorizer, which counts the frequency of a word in a document. The second vectorizer is the TfidfVectorizer, which uses the tf-idf formula we learned in class. We decided to use only unigrams and bigrams to limit the complexity of our model, and set the minimum document frequency to 2. Both of these vectorizers output a sparse matrix. We then append the numerical non-BOW features onto this sparse matrix and pass it into our models.

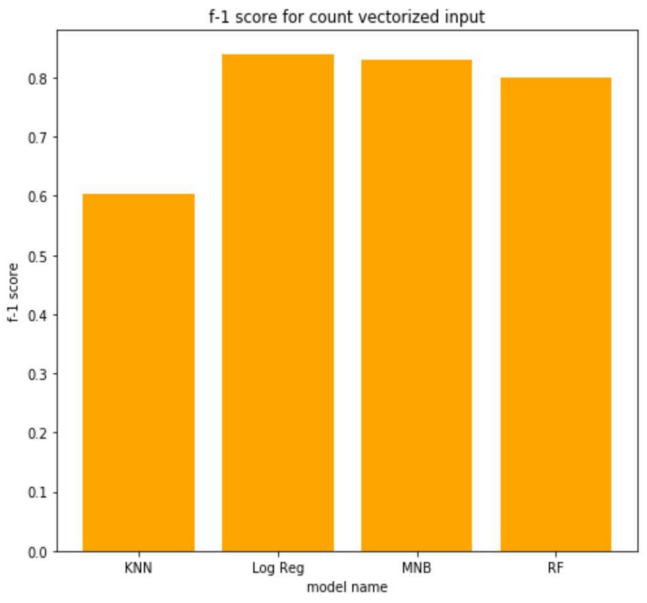
In a state of the art sentiment classification system, we discovered that neural networks are efficient and most suitable for the task [3]. However, these neural networks require mass amounts of data as well as long training times. Instead, in our project, we used four different models from the sklearn library. The first model we used was Multinomial Naive Bayes, which was the implementation we used in homework 4. This model essentially uses Bayes’ theorem to calculate the probability of a class given its features. According to this article on LinkedIn [3], Naive Bayes can perform better than neural network based models, especially in smaller datasets. We used GridSearchCV to optimize the performance of this model by tweaking the alpha parameter, which affects the smoothing (similar to Laplace smoothing as mentioned in class) of the probability distributions [4].

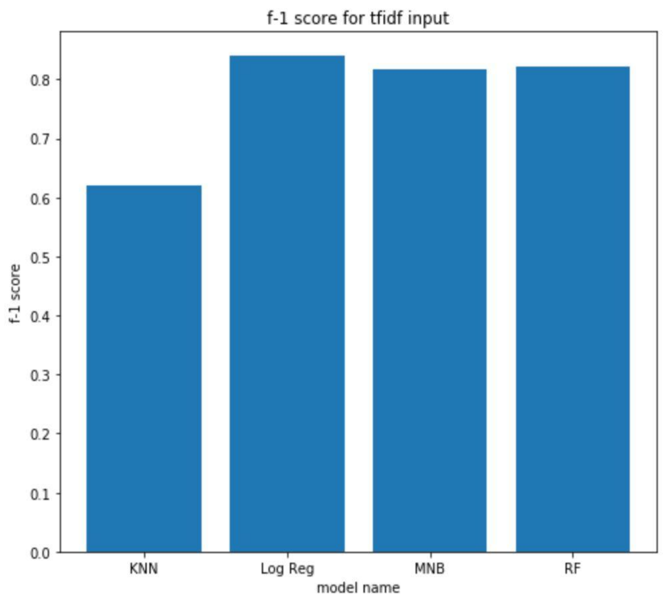
The second model we used was Logistic Regression. This model passes in the features of the review into a sigmoid function, which outputs a value between 0 and 1. If the output is greater than or equal to .5, the model predicts that the review was recommending the game. Otherwise, the model predicts that the review was not recommending the game. We used GridSearchCV to find the best C parameter, which regularizes parameter values in the model and reduces overfitting [5]. We also had to limit the max iterations so that the model would not take too long to run, possibly sacrificing performance.

The third model we used was RFC. This model is basically a collection of decision trees (hence the name forest), but the results from these decision trees are usually averaged to form one final result. The advantage to using multiple trees is that the overfitting of a single decision tree can be reduced if there are other decision trees taken into account. We used random search to tune the number of estimators, whether the model will be bootstrapped, the max_depth of the tree, the minimum samples for a leaf of the tree, and the minimum number of samples required to split a node [6]. Since there were a lot of hyperparameters to try out, we wanted to use a random search to narrow down the optimal hyperparameters, then use a gridsearch to pinpoint these parameters. However, the gridsearch took immensely long for both vectorizers, and so we commented out the code. This is most likely due to the lack of computational power our computers have.

The last model we used was the KNeighbors classifier. This model uses the features to plot a given point on a hyperplane, and picks the labels of the closest N points (hence neighbors). The majority label of these neighbors is the label for that given point. For this model, we used grid search to change how many neighbors the model would query, and also the weight function used for predictions [7].

For each model we used in this project, we calculated the accuracy, F-1 score, precision, and recall of both the training and testing set. We concluded that there is no apparent difference between the performance using a CountVectorizer versus a TfidfVectorizer. For Logistic Regression and MNB, the CountVectorizer performed better. On the other hand, TfidfVectorizer performed better on KNeighborsClassifier and RFC. There are slight differences, but are negligible in the scope of our project. Looking at the results for testing data (these results are most genuine reflection of the model's performance since we are predicting on unseen data), we can see that for both count vectorization and tfidf vectorization, the accuracy is around 80% to 83% across multinomial naive bayes, logistic regression, and random forest classifier. These are fairly trusting performances, predicting most of the unseen reviews correctly, which is what we expected. On the training set, our F-1 scores hover around the mid 90s, which may indicate overfitting (especially in RFC where it is 99).





However, for KNeighbor classifier, the count vectorization F-1 score is 60%, and tfidf vectorization F-1 score is 62%. There is a significant drop in accuracy for KNeighbor classifier compared to the other 3 classifiers, which is something we did not expect. To figure out what caused this deviation, we did some further research with the architecture of these different machine learning models. The default number of nearest neighbors is 5. We retrained the model using n_neighbors=30, and the accuracy only improved to 63%, which is not a very significant upgrade from the previous 60%. Another error that might be causing this inefficient prediction has to do with the scale of the data, since Kneighbors is very sensitive to scale as it calculates distance between data points [8]. In the training data, we created sparse vectors embeddings for the text that are either 0 or 1. However, we also appended other features to the training data, and one of them is the length of the review. The length of the review ranges from 1 to a number in the hundreds, which is on a different scale than the other BOW features that are mostly in single digits, hence it could be a major cause of the failure of Kneighbors classification.

One reason that our performance on the testing set was not higher may be due to sarcasm in the reviews. Consider the review “I LOVE THIS GAME, ESPECIALLY DYING OVER AND OVER AGAIN!”. Our model might pick up on the word “love” and consider it to be a positive review, however a human could easily pick up the sarcasm and tell that it is in fact a negative review. This situation could be remedied by introducing more context (N-grams past 2), however this will also increase the risk of overfitting our data. Another way we could improve performance is to increase the amount of data we use for training and testing. Although our model performed decently well, using more data might lead to a slightly better performance as the models can learn from more patterns in text. Once again, we do need to be careful not to use too much data for training, since overfitting is a risk. Furthermore, the time it would take to train and tune our models would increase severely, which was already an issue with about 29k entries.

In this project the sentiment classification is binary, meaning that one piece of text can either be positive or negative in emotion. However, in the real world, sentiment is always a lot more complicated than a binary representation. Hence, it would be very interesting if we could improve our training data such that instead of having each review labeled with 0 or 1, we label it with a number between 0 and 1, hence having a quantified representation of how negative, neutral, or positive a review is. To make this task still a classification, we could use buckets to represent ranges in sentiment. For example, we can have buckets: 0-0.2, 0.2-0.4, 0.4-0.6, 0.6-0.8, 0.8-1.0. This way, we are quantifying sentiment but at the same time making it a category classification job, so that it is easy to evaluate the performance of the model and to compute the accuracy and f-1 scores. However, this can prove to be too difficult and specific of a task. Sentiment is extremely subjective if we were to be more specific than a 0 or 1. For instance, is there a difference between a sentiment of .75 and .80, and if so, should they be placed in the same class?

A way we could make our classifier more specific, yet still a doable classification task is to add the neutral class into our dataset. An example of a neutral game review would be something like “This game was okay, it could have been better but also could have been worse.” In order to implement this task, our dataset would have to be modified to have labels of 0, 1, and 2. Furthermore, it is hard to determine whether or not a review is neutral, as connotations often carry a slight pull in either sentiment.

There are various experiments and additions we would do in the future. For example, there are parameters in the vectorizers that we did not fiddle around with much. We could try using trigrams along with bigrams and unigrams when vectorizing our data, as well as increasing the minimum document frequency. Furthermore, we could introduce more data into our models, since there are a plethora of reviews to be scraped from Steam. In addition, if we had more computational power, we would try to test out more parameters on our models. Especially for our RFC model, we had to settle with a RandomSearch with limited parameters so that our computers wouldn’t overheat and/or crash. It may be possible for RFCs to perform better than Logistic Regression and MNB if we found the optimal hyperparameters. Lastly, we would also try to uncover which features and words contribute the most to the output. Although it may be difficult to parse a sparse matrix, these insights into specific features would be interesting to look at and possibly visualize. Nonetheless, we are satisfied with the results that we have obtained from this final project.

Works Cited

1. https://www.kaggle.com/piyushagni5/sentiment-analysis-for-steam-reviews?select=train.csv
2. https://zenodo.org/record/1000885#.XdXaH1dKhPY
3. https://www.linkedin.com/pulse/best-ai-algorithms-sentiment-analysis-muktabh-mayank/
4. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
6. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
8. https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn