Alec Schneider & Thomas McGuire

6/22/2021

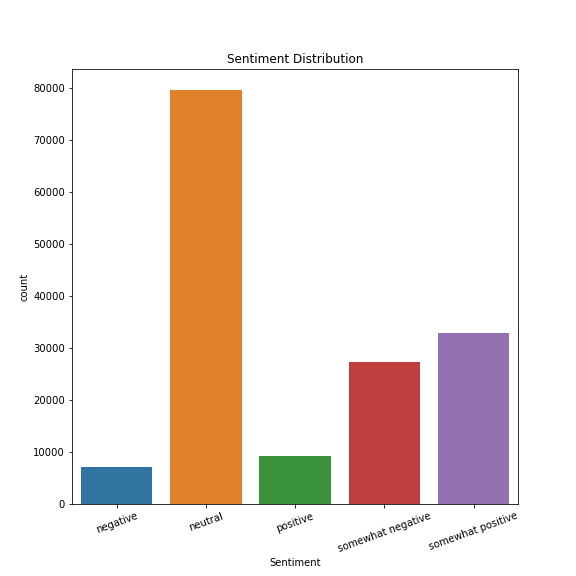
NLP Final Project

**Sentiment Analysis With Movie Reviews**

We were challenged to find a dataset of text documents with the goal of creating features and models that would allow us to analyze sentiments. To take on such a task, we needed to find a dataset that is rich in documents and proper sentiment labeling. Using [Kaggle](https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data), we were able to find the famous Rotten Tomatoes movie review dataset that has been parsed by the Stanford parser.

The movie review dataset includes 8,529 unique sentences, with 156,060 phrases parsed from the sentences as a separate column in the text file. Each phrase has a sentiment label associated with it, and this will be used to score our sentiment classification models.

|  |  |  |
| --- | --- | --- |
| **Sentiment Label** | **Sentiment** | **Count** |
| 0 | Negative | 7,072 |
| 1 | Somewhat Negative | 27,273 |
| 2 | Neutral | 79,582 |
| 3 | Somewhat Positive | 32,927 |
| 4 | Positive | 9,206 |



As you can see in the table and graphic above, this is not a balanced dataset. The number of neutral sentences is far greater than any of the other sentiment labels. Due to the size of the dataset, we decided not to upsample any of the other sentiment phrases. We also decided against downsampling the neutral phrases to preserve valuable data. We are aware of the potential bias that could be represented by our classifiers due to these decisions, but we believe that the bias will not outweigh the performance from using the original training set.

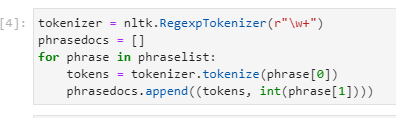
**Text Preprocessing**

Before we can analyze the phrases, there are a few steps required to prepare the text data for the NLTK, Apache Spark, and PyTorch libraries’ various Natural Language Processing (NLP) modules and functions.

Step 1: Tokenize each phrase in the dataset.

*Tokenization is the process of splitting a text string into multiple substrings (also called tokens). For example, The tokenized version of the sentences “Hello there Bob” is: [“Hello”, “there”, “Bob”].*

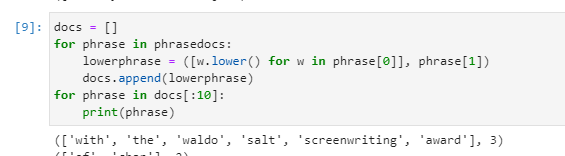
Python examples with NLTK:





Step 2: Ensure all tokens in the dataset are in lowercase to avoid any potential caps lock or typing errors made during the review creation.

Code example:



Step 3 (Comparison Experiments Only): Removal of stopwords and punctuation

*Stopwords are words that are extremely common in the English language such as ‘a’, ‘the’, or ‘an’. Because these words are so common, when we try to create our bag of words, they would dominate the top words no matter how many were selected and offer very little to assist our model in distinguishing between various sentiments. Punctuation is also common in every single sentence and typically does not offer the model any help in figuring what makes a sentence positive against slightly positive*.

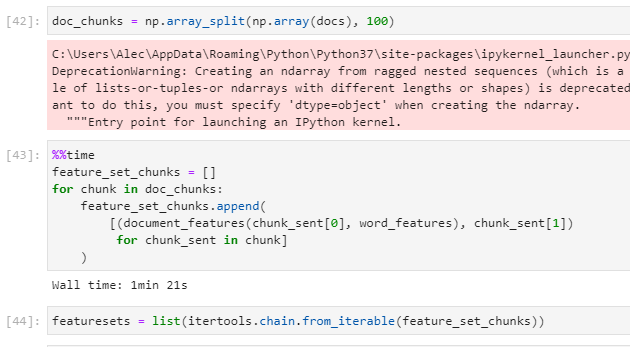
**Feature Engineering**

With the preprocessed text data we can now create features from the tokens to extract additional value. There are endless approaches to feature engineering, and we will cover a few major feature engineering techniques for text data.

Option 1: Bag Of Words (BOW)

*A Bag Of Words is a method of feature extraction with text data. It is the method of representing all words in a dataset of text documents.*

In order to get the most accurate Bag of Words for our future classification, we had to make all the words lowercase (Text Preprocessing Step 2) so that we could discover which were the 1000 most common words in our dataset. We choose 1000 words due to memory issues faced on our computers when using larger vocab sizes. To ensure randomization, we shuffled the documents. So we first tokenized each phrase from the dataset and saw the corresponding sentiment after also ensuring the phrases were all lowercase. We then checked the frequency distribution across all the words. After we had that, we could get the 1000 most common words.

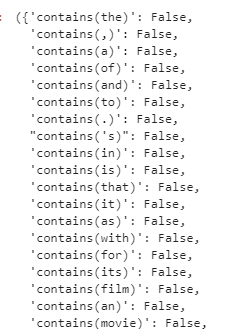


Option 2: Unigram

*A Unigram is an individual word that is checked for in a text document, which can be passed to a language model to calculate probabilities.*

Once we had the 1000 most common words, we could then go through each document and see which documents had which individual words. We checked the first document to see how many of the most common words it contained. We also counted the number of times those individual words appeared in the corpus. Because of the sheer size of the number of documents, we had to split this part into 100 chunks of documents and then recombined those chunks (code example below) before passing the features to our models. After it was recombined, we could run our Naive Bayes with Kfold cross-validation and get our accuracy, recall, precision, and F-measure scores.

Example of the Unigram feature set:

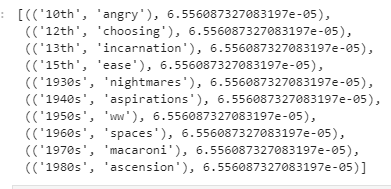


Option 3: Bigrams

*A Bigram is a sequence of two words (e.g. “hi there”) paired together that can be used as features to calculate probabilities in a language model.*

Similar to the unigrams above, we wanted to see how bigrams or two consecutive words would affect our model. We determined what the 1000 most common bigrams were and the number of times each of those appeared in the corpus. This was again after removing stopwords, any word that was not English, and ensuring that all the words were lowercase. We split the corpus into 100 bigram chunks and then ran our Naive Bayes with Kfold cross-validation to determine how accurate our model was.

Bigram example with the raw frequency of occurrence:

****

Option 4: Part-Of-Speech (POS) Tagging

*Part-Of-Speech Tagging is the method of processing a sequence of words and attaching a tag to each word so that we can identify the verbs, nouns, adverbs, etc., in text documents.*

After we had used the bigrams, we wanted to analyze how the Part-of-Speech tagging would affect our model. We counted the number of nouns, verbs, adjectives, and adverbs for each document, then compared that to how often that number of parts of speech occurred across the entire corpus. We again had to split into 100 piece chunks then recombine before performing the Naive Bayes with Kfold cross-validations.

POS Tagging example:



Option 5: Count Vectorizer

*Count Vectorization is a method within* [*Apache Spark*](https://spark.apache.org/docs/latest/ml-clustering.html#latent-dirichlet-allocation-lda) *to convert a collection of text documents to vectors of token counts. This method is very similar to the Bag Of Words method described in option 1*.

The count vectorizer will be used in our preprocessing pipeline that will be used in our Latent Dirichlet Allocation (LDA) clustering experiment. More information on LDA will be provided in the Experiments section that follows.

Example of text before (raw) and after applying the Count Vectorizer (vectors)

+-----+---------------+-------------------------+

|label|raw |vectors |

+-----+---------------+-------------------------+

|0 |[a, b, c] |(3,[0,1,2],[1.0,1.0,1.0])|

|1 |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|

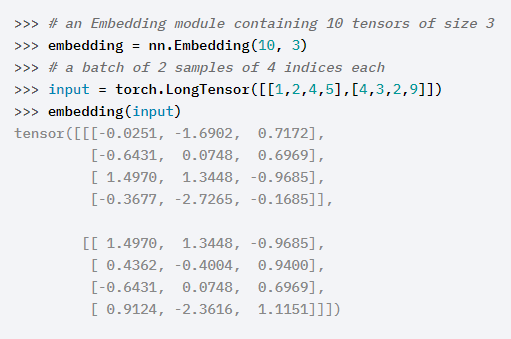
+-----+---------------+-------------------------+

Option 6: Word Embeddings

*A Word Embedding is a learned representation for text where words that have the same meaning have a similar representation. Each word is mapped to a vector and vector values are learned for predefined fixed-sized vocabulary from a corpus of text.*

Our neural networks in part B of the Experiments section will use the Word Embeddings method by leveraging [PyTorch’s embedding layer](https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html#torch.nn.Embedding) class to handle the embedding for us within our text classification neural networks.

Example of using Word Embedding on a PyTorch tensor:



**Experiments**

*See Appendix for notebooks and scripts related to each experiment.*

1. **Baseline Experiments With Comparison:**

All baseline and comparison (excluding LDA) experiments will leverage NLTK’s Naive Bayes text classification algorithm to predict the sentiments of each document. The Naive Bayes algorithm attempts to find the “maximum a posteriori” or most likely class in which input belongs by using the Bayes Rule. The Bayes Rule utilizes the prior probability of a document (d) belonging to a class (c) and updates the probability of this possibility based on the new information passed to it. Since the probabilities are updated for every new input passed to the algorithm, it can be expensive to update. However, Naive Bayes can be extremely powerful in text classification with the bag of words approach as the bag of words is creating the probability of a word occurring in a document.

Baseline - No Filtering

1. Unigram Features using top 1000 most frequent words with 5 Fold Cross Validation Naive Bayes

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 55.8% |
| Precision | 44.8% |
| Recall | 41.1% |
| F-Measure | 41.7% |

1. Top 1000 Bigram Features with 10 Fold Cross-Validation

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 51.0% |
| Precision | 45.6% |
| Recall | 20.0% |
| F-Measure | 13.7% |

1. Part-Of-Speech Tagging with 10 Fold Cross-Validation

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 53.6% |
| Precision | 40.9% |
| Recall | 39.6% |
| F-Measure | 38.7% |

Comparison Features - Removal of Stop Word And Punctuation

1. Unigram Features using top 1000 most common words with 15 Fold Cross-Validation Naive Bayes

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 57.2% |
| Precision | 48.8% |
| Recall | 38.1% |
| F-Measure | 40.8% |

1. Top 500 Bigram Features with 5 Fold Cross-Validation

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 51.1% |
| Precision | 60.7% |
| Recall | 20.3% |
| F-Measure | 14.1% |

1. Part-Of-Speech Tagging with 5 Fold Cross-Validation

|  |  |
| --- | --- |
| **Metric** | **Score** |
| KFold Accuracy | 55.6% |
| Precision | 43.8% |
| Recall | 39.5% |
| F-Measure | 40.6% |

New Approach - Latent Dirichlet Allocation (LDA) using PySpark

Latent Dirichlet Allocation (LDA) is an unsupervised clustering algorithm, which allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similarF. LDA is popular in machine learning for topic discovery because it does not require labels to train, unlike Naive Bayes or our neural networks below. To fit the training data on our LDA clustering model, we constructed a data pipeline in Spark using the following steps.

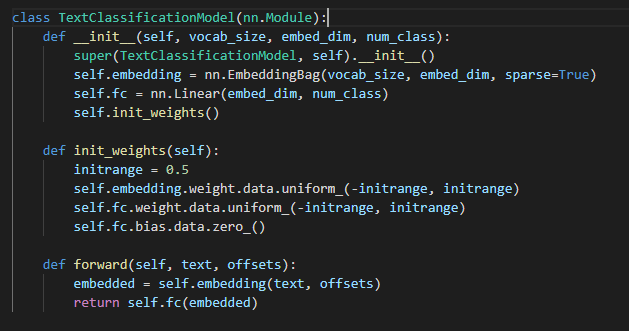
* RegexTokenizer
* Stop Word Remover
* Count Vectorizer
* LDA

From here we were able to compute the probabilities for each cluster to which the algorithm believes the document belongs. After predicting the probabilities, we can assign the max probability as the cluster prediction and get the following scores. Since this is an unsupervised learning algorithm, it is not surprising to see poor results such as this.

|  |  |
| --- | --- |
| **Metric** | **Score** |
| Accuracy | 16.8% |
| Precision | 20.4% |
| Recall | 20.2% |
| F-Measure | 13.8% |

1. **Advanced Sentiment Classification with PyTorch**

For our advanced approach to sentiment classification, we decided to create a simple neural network architecture and use different hyperparameters to tune the model. Our results below represent the three best combinations of hyperparameters for our TextClassificationModel class. To prepare our training dataset for our neural network, we must first tokenize it just as we would for our Naive Bayes classifier before passing it to the Embedding Bag layer. After tokenization, we can pass the vocab size and predetermined embedding size to the neural network. As stated in the Feature Engineering section, Word Embedding maps each word in the vocab to a vector that can be used to represent the similarity between words. After embedding the vocabulary, we must predict a probability for each class in the dataset, and that is where our linear layer comes in. Our linear layer will map the input vectors, using a linear function, into our known 5 classes (num\_classes), which can be used to get a final predicted class such as 0, 1, 2, etc. The code outlining the two layers in the neural net can be seen below. Our neural nets and training data will be passed to the GPU, in order to speed up training execution.

Text Classification Model Architecture:

Neural Network #1

Hyperparameters:

Embedding Size = 64

Epochs = 25

Learning Rate = 1.0

Batch Size = 64

Loss Function: Cross Entropy

Optimizer: Stochastic Gradient Descent

-----------------------------------------------------------

| end of epoch 25 | time: 28.35s | valid accuracy 0.571

-----------------------------------------------------------

Total Training Time: 13.96mins

|  |  |
| --- | --- |
| **Metric** | **Score** |
| Accuracy | 57.1% |
| Precision | 53.9% |
| Recall | 35.8% |
| F-Measure | 38.0% |

Neural Network #2

Hyperparameters:

Embedding Size= 128

Epochs = 10

Learning Rate = 10.0

Batch Size = 24

Loss Function: Cross Entropy

Optimizer: Stochastic Gradient Descent

-----------------------------------------------------------

| end of epoch 10 | time: 50.65s | valid accuracy 0.583

-----------------------------------------------------------

Total Training Time: 8.44mins

|  |  |
| --- | --- |
| **Metric** | **Score** |
| Accuracy | 58.3% |
| Precision | 59.6% |
| Recall | 44.1% |
| F-Measure | 47.7% |

Neural Network #3

Hyperparameters:

Embedding Size = 128

Epochs = 10

Learning Rate = 5.0

Batch Size = 128

Loss Function: Cross Entropy

Optimizer: Stochastic Gradient Descent

-----------------------------------------------------------

| end of epoch 10 | time: 33.48s | valid accuracy 0.585

-----------------------------------------------------------

Total Training Time: 4.87mins

|  |  |
| --- | --- |
| **Metric** | **Score** |
| Accuracy | 58.5% |
| Precision | 58.8% |
| Recall | 42.5% |
| F-Measure | 45.7% |

**Summary**

Biggest Challenges

The data set size and subsequent memory size of feature extractions in the NLTK feature engineering methods led to frequent program failures. As noted in the Feature Engineering section, we combated these memory errors by splitting our corpus of documents into 100 chunks, applied the feature engineering method, then recombined all the features into one list before being able to pass the data to our Naive Bayes classifiers. After resolving feature extraction issues, our Naive Bayes models took upward of 90 minutes to train, and upwards of another 30 minutes to make predictions, which reduces a modeler’s ability to rapidly analyze model performance and build new models in the hopes of improving performance.

It is important to note that we faced no memory or performance issues while using Apache Spark for LDA clustering, or PyTorch. End-to-end feature extraction, model training, and sentiment prediction for these methods could be completed in upwards of a few minutes. Compared to some Naive Bayes classifiers, time savings were up to about a factor of 20. This incredible performance gain can not be overlooked. This is due to the fact that Spark leverages parallel data processing on partitions of the full dataset it is given, and PyTorch utilized the GPU, which is known to increase the performance of matrix operations utilized in neural networks. These methods should be heavily favored when using large text datasets.

We also had trouble getting any decent classification accuracy with the feature extraction methods used in this analysis. We believe that more neural net approaches should be leveraged to enhance classification accuracy, despite their lack of interpretability.

Feature Analysis

Using our unigrams, bigrams, and POS-tagging across both of our experiments, we can see that each is useful and powerful in its own right. Before using Pytorch, our filtered and non-filtered experiments were performed within a few percentage points but the filtering of the punctuation consistently performed better when using Naive Bayes. Bigrams typically performed worse than unigrams and POS tagging by a significant margin. This is most likely due to the massive variety in how people can speak especially when reviewing movies, making it much more difficult for two words that were not stopwords to appear near each other. POS tagging and unigrams each only rely on one word at a time.

Because of power constraints already mentioned, we were forced to limit to 1000 bigrams and unigrams each, which made it more difficult for the model to appropriately use those features in the remaining documents. Finding one word in almost 160000 documents is challenging, specifically when it is not a common word such as stopwords, and finding two words next to each other is much more difficult. That being said, I am impressed by our models and their measures of success. By analyzing Parts-of-Speech taggings allow us to see if there is a pattern in which sentences or phrases with specific POS lead to a specific sentiment.

Overall Results

On average, the text classification neural networks performed much better across all metrics when compared to the time and resource-heavy Naive Bayes models. The best classifier we trained was the neural network that utilized a 128-word embedding size, 10 epochs, a learning rate of 10, and a batch size of 24 documents. We believe this classifier is better than the neural network with better accuracy, due to the fact that its performance in the precision, f-measure and recall metrics should outweigh the 20bps difference in classification accuracy. It is interesting that the three best neural nets placed within the top four of all classification models despite having a very simple feature extraction process (tokenization paired with word embedding). From there we observed that unigram features performed best for the Naive Bayes classification tasks. Unigrams most likely performed the best because unigrams are much more frequent than bigram pairs and POS phrases. LDA clustering performed the worst by far since it was not trained using the sentiment labels given in the dataset.

**Results Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Accuracy** | **Precision** | **F-Measure** | **Recall** |
| torch\_embed128\_epochs10\_lr5\_batch128 | 58.5% | 58.8% | 45.7% | 42.5% |
| **torch\_embed128\_epochs10\_lr10\_batch24** | **58.3%** | **59.6%** | **47.7%** | **44.1%** |
| filtered\_unigram\_15fold | 57.2% | 48.8% | 40.8% | 38.1% |
| torch\_embed64\_epochs25\_lr1\_batch64 | 57.1% | 53.9% | 38.0% | 35.8% |
| no\_filter\_unigram\_5fold | 55.8% | 44.8% | 41.7% | 41.1% |
| filtered\_pos\_5fold | 55.6% | 43.8% | 40.6% | 39.5% |
| no\_filter\_pos\_10fold | 53.6% | 40.9% | 38.7% | 39.6% |
| filtered\_bigram\_5fold | 51.1% | 60.7% | 14.1% | 20.3% |
| no\_filter\_bigram\_10fold | 51.0% | 45.6% | 13.7% | 20.0% |
| filtered\_LDA | 16.8% | 20.4% | 13.8% | 20.2% |

**Appendix**

Notebooks and Scripts:

1. Baseline Experiments With Comparison:
   1. Baseline - No Filtering
      1. Nltk\_sentiment\_classification.ipynb
      2. Bigrams.ipynb
      3. POS\_tags.ipynb
   2. Latent Dirichlet Allocation (LDA) using PySpark
      1. sparkLDA\_sentiment\_analysis.ipynb
   3. Comparison - Filtering Punctuation and Stop Words
      1. nltk\_sent\_class\_filter.ipynb
2. Advanced Sentiment Classification with PyTorch:
   1. simple\_sentiment\_classification.py
   2. sentiment\_utils.py
   3. Torch\_model\_scoring.ipynb

References:

1. *Sentiment Analysis on Movie Reviews*. Kaggle. (n.d.). https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data.
2. *Clustering*. Clustering - Spark 3.1.2 Documentation. (n.d.). https://spark.apache.org/docs/latest/ml-clustering.html#latent-dirichlet-allocation-lda.
3. *Embedding*. Embedding - PyTorch 1.9.0 documentation. (n.d.). https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html#torch.nn.Embedding.
4. *nltk.tokenize package*. nltk.tokenize package - NLTK 3.6.2 documentation. (n.d.). https://www.nltk.org/api/nltk.tokenize.html.
5. *nltk.classify.naivebayes*. nltk.classify.naivebayes - NLTK 3.6 documentation. (n.d.). https://www.nltk.org/\_modules/nltk/classify/naivebayes.html.
6. Wikimedia Foundation. (2021, June 3). *Latent Dirichlet allocation*. Wikipedia. https://en.wikipedia.org/wiki/Latent\_Dirichlet\_allocation.