Twitter Sentiment Analysis using PySpark

Sanghamitra Muhuri | MS in Business Analytics | University of Illinois at Chicago | smuhur2@uic.edu

Motivation

The objective is to build a Twitter Sentiment Analyzer developed using PySpark which is a combination of Apache Spark and Python. For the sentiment classification of tweets, machine learning models on Logistic Regression using SparkML have been built. We experimented with HashingTF-IDF, CountVectorizer-IDF and N-gram algorithms to determine the best performing model based on testing and validation. Spark's ability to perform well on iterative algorithms makes it ideal for implementing machine learning techniques for the purpose of sentiment analysis.

To train the model, we used "Sentiment140" dataset that contains 1.6 million entries of tweets along with binary sentiment attribute of positive labels and negative labels.

Dataset

Our Dataset Sentiment140 has been downloaded from http://help.sentiment140.com/for-students/.

Data Pre-Processing

- We observe that dataset has tweets greater than 140 characters which are removed. HT ML decoding of the data is performed using Beautiful Soup.
- Followed by removing UTF-8 BOM, @mentions and URL address ('http:','www.')
- For the purpose of language modelling we turned all the words to lowercasing with the u se of string.lower() function.
- Negation handling list had been defined, and special characters, numbers and whitespac es were removed.
- Lastly, Tokenization and joining was done using NLTK Tokenizer. 3959 Null tweets were removed after the data cleaning process.
- We will split the dataset into three parts: training, validation (1%) and test (1%).

Exploratory Data Analysis

- The Python library known as wordcloud was used to generate visualizations of positive and negative words. And the Term Frequency has been calculated using CountVectorizer.
- We have observed that are no null entries and 6 columns: Sentiment: the polarity of the tweet (0 = negative, 2 = neutral, 4 = positive), ID of the tweet, Date of tweet, Query(lyx), User Handle, Text of tweet.
- Further data analysis revealed sentiment column had 50% of the data with negative label, and another 50% with positive label.

- There was also a keen observation that no skewness was shown on the class division.

Models:

1. HashingTF + IDF + Logistic Regression

We implemented Logistic regression with HashingTF and IDF and got 78% accuracy for our prediction on unseen test data.

Term Frequency-Inverse Document Frequency (TF-IDF) is a common text pre-processing step. In Spark ML, TF-IDF is separate into two parts: TF (+hashing) and IDF.

TF: HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. In text processing, a “set of terms” might be a bag of words. The algorithm combines Term Frequency (TF) counts with the hashing trick for dimensionality reduction.

IDF: IDF is an Estimator which fits on a dataset and produces an IDFModel. The IDFModel takes feature vectors (generally created from HashingTF) and scales each column. Intuitively, it down-weights columns which appear frequently in a corpus.

We split each sentence into words using *Tokenizer*. For each sentence (bag of words), we use HashingTF to hash the sentence into a feature vector. We use IDF to rescale the feature vectors; this generally improves performance when using text as features. Our feature vectors could then be passed to a learning algorithm.

Testing

We have cross validated our model with the validation set to tune the parameters. Further, we used our model to predict on the test set and found our accuracy to be ~78.85%

2. CountVectorizer + IDF + Logistic Regression Model

CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

During the fitting process, CountVectorizer will select the top vocabSize words ordered by term frequency across the corpus. An optional parameter “minDF” also affect the fitting process by specifying the minimum number (or fraction if < 1.0) of documents a term must appear in to be included in the vocabulary.

```
##############CountVectorizer + IDF + Logistic Regression#########
from pyspark.ml.feature import CountVectorizer

tokenizer = Tokenizer(inputCol="text", outputCol="words")
cv = CountVectorizer(vocabSize=2**16, inputCol="words", outputCol='cv')
idf = IDF(inputCol='cv', outputCol="features", minDocFreq=5) #minDocFreq: remove sparse terms
label_stringIdx = StringIndexer(inputCol = "target", outputCol = "label")
lr = LogisticRegression(maxIter=100)
pipeline = Pipeline(stages=[tokenizer, cv, idf, label_stringIdx, lr])

pipelineFit = pipeline.fit(train_set)
predictions = pipelineFit.transform(val_set)
accuracy = predictions.filter(predictions.label == predictions.prediction).count() / float(val_set.count())
roc_auc = evaluator.evaluate(predictions)

print ("Accuracy Score: {0:.4f}".format(accuracy))
print ("ROC-AUC: {0:.4f}".format(roc_auc))
```

```
Accuracy Score: 0.7900
ROC-AUC: 0.8613
```

Testing

We used ROC curve for evaluation on our model and found the percentage under the curve to be ~86%.

Also, we used test set to predict the accuracy which turned out to be ~79%.

3. N-gram Implementation

An n-gram is a sequence of n tokens (typically words) for some integer n. The NGram class can be used to transform input features into n-grams.

NGram takes as input a sequence of strings (e.g. the output of a Tokenizer). The parameter n is used to determine the number of terms in each n-gram. The output will consist of a sequence of n-grams where each n-gram is represented by a space-delimited string of n consecutive words. If the input sequence contains fewer than n strings, no output is produced.

We used VectorAsembler in pipeline to combine features from each n-gram. We extracted 16,000 features from unigram, bigram and trigram to get 16,000 final features.

Testing

We used ROC curve for evaluation on our model and found the percentage under the curve to be ~88.64%. Also, we used test set to predict the accuracy which turned out to be ~81.31%.

Conclusion

In conclusion, we found that the best model for sentiment analysis which gives the maximum area under the curve with highest accuracy is the N-gram implementation.