Predictive Analytics with Spark

CSE 587: Data Intensive Computing - Assignment 3

May 16, 2020

Kaggle Team Name: Abhishek_Pooja

Abhishek Mishra
UBID: amishra6
PN# 50071253

Pooja Ravi
UBID: pravi
PN# 50291368

Contents

Outline

The assignment focuses on using Spark Libraries to implement an end to end Predictive Analytics Pipeline which implements a movie genre prediction model. The dataset contains information about movies with attributes such as movie id, movie name, plot and genre. The goal is to predict all the genres of a particular movie based on the plot summaries.
There are 20 distinct genre categories and about 31k rows of data for training, 7k rows for testing. Each movie can have multiple genres.
The high level view of the predictive analysis end to end pipeline is as shown below.

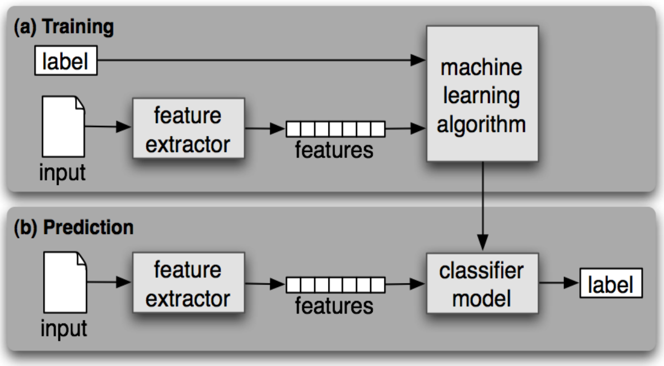

Figure 1: High level view of the end to end pipeline

Loading data

Data was loaded into Pandas dataframe and then a schema was created with which the data was loaded into the PySpark dataframe. The data loaded on the PySpark dataframe was used throughout in the assignment.

One Hot Encoding

The training phase implements One hot encoding on the column *'genre'*. This column is an array of strings where each element represents a genre. This was converted to an array of integers where each position of the array represents a distinct category of genre. A new column was created as a placeholder for each category of genre which held the value '1' to indicate the presence of the genre or the value '0' to indicate the absence of the genre.

Data Pre processing

Since there are about 70k distinct keywords in the *train.csv*, this is extremely intensive to fit on a machine learning model. The data was preprocessed using Pipeline which consisted of stages on StopWordsRemover, RegexTokenizer package in *pyspark.ml.feature*. This data was added as another column into the PySpark dataframe under the column "filtered". This intermediate preprocessed data was further lemmatized using Stemmer and yielded around 27k features. But after taking a close look at the stemmed words, it was noticed that a lot of words didn't make sense and seemed out of context. So the intermediate preprocessed data was used as input for the various implementations of feature extractor.

Feature Engineering

The intermediate preprocessed data obtained from the Data Preprocessing stage was used as input for the various implementations of feature extractor.

Part 1 used Term Document matrix as the feature extractor.
Part 2 used TF-IDF as the feature extractor.
Part 3 used TF-IDF+Word2Vec as the feature extractor.

Building Model

A Binary classifier (Logistic Regression) model was trained for each category of genre. So, 20 binary classifiers were built and these 20 results were aggregated to form the final predictions.

Each binary classifier was trained on its own genre category. For example, for "Drama" binary classifier a new data consisting of movie id, plot, features and Drama was created and the model was trained on this data. The predictions from this model was written into a new column called "P drama". The predicted value of '1' indicates that the movie falls into the category of "Drama" and the value of '0' indicates that the movie doesn't fall into the category of "drama".

The same was followed for each of the 20 categories of genre. These predicted columns were appended into the dataframe as a new column. Once all the 20 predictions are completed, the predicted columns are concatenated into a space separated string and written to a new column, "predictions" in the dataframe. Thus the final output has "movie id" and "predictions" in the dataframe.

The final dataframe consisting of "movie id" and "predictions" was written to a single csv file using dataframe.repartition(1).write.csv(filename, header= True) command. Without the repartition(1) command, write.csv writes to multiple csv files.

Part 1: Term document matrix

Pre processing

Based on the pre processing stages discussed in the Outline section, the input data is preprocessed to remove punctuations, special characters and stopwords. The preprocessed data is written into a new column,"filtered" in the dataframe.

Feature Engineering

A Term Document matrix is created using column "filtered" in the dataframe using **CountVectorizer** from ***pyspark.ml.feature***. This yielded about 34k features which were written into a new column, "features" in the dataframe. The vocabulary size was set to 100000 and the minimum number of documents to be found in was set to 5.

Building Model

The column "features" which was extracted from the Feature Extractor which used Term Document matrix was used by each of the 20 models to predict the presence or absence of the genre. The results were appended into a single column,"predictions" as a string.

Results

This method reported a macro F1 score of 0.97829 after uploading the predicted labels of the test data on the Kaggle website.

Part 2: TF-IDF

Pre processing

Based on the pre processing stages discussed in the Outline section, the input data is preprocessed to remove punctuations, special characters and stopwords. The preprocessed data is written into a new column,"filtered" in the dataframe.

Feature Engineering

A TF-IDF matrix is created using column "filtered" in the dataframe using **HashingTF, IDF** from ***pyspark.ml.feature***. Term frequency–inverse document frequency(TF-IDF), is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. The number of features was set to 100000 and the minimum number of documents to be found in was set to 5. This yielded features which were written into a new column, "features" in the dataframe.

Building Model

The column "features" which was extracted from the Feature Extractor which used TF-IDF was used by each of the 20 models to predict the presence or absence of the genre. The results were appended into a single column,"predictions" as a string.

Results

This method reported a macro F1 score of 0.97968 after uploading the predicted labels of the test data on the Kaggle website. It was noticed that Feature Engineering using TF-IDF performed better than Term Document matrix.

Part 3: Custom Feature Engineering

Pre processing

Similar to the pre processing stages in parts 1 and 2, and as discussed in the Outline section, the input data is preprocessed to remove punctuations, special characters and stopwords. The preprocessed data is written into a new column,"filtered" in the dataframe.

Feature Engineering

For this part as well, a TF-IDF matrix is created using column "filtered" in the dataframe using **HashingTF, IDF** from ***pyspark.ml.feature***, and a Word2Vec matrix using **Word2Vec** from ***pyspark.ml.feature*** as well. The Word2Vec model transforms each document into a vector using the average of all words in the document. The number of features for TF-IDF was set to 50000 and vector size for Word2Vec was set to 50. So for our case, for instance, if one sentence has five words, then Word2Vec will convert each word into a feature vector of size 50. The yielded features from TF-IDF was written into column "idf_features" and Word2Vec output features were written into "w2v_features" column. **VectorAssembler** from ***pyspark.ml.feature*** was used to combine these features and the output was written to "features" column.

Building Model

The column "features" which was extracted from the Feature Extractor which used VectorAssembler to combine features obtained from TF-IDF and Word2Vec, was used by each of the 20 models to predict the presence or absence of the genre. The results were appended into a single column, "predictions" as a string.

Results

This method reported a macro F1 score of 0.99187 after uploading the predicted labels of the test data on the Kaggle website. It was noticed that Feature Engineering using TF-IDF+Word2Vec performed better than TF-IDF alone. Our score can be seen on the leaderboard next to our team name: **Abhishek_Pooja**.