CSE 587 - Data Intensive Computing

Assignment 3: Predictive Analytics with Spark

Submitted by:

Vishva Nitin Patel (vishvani@buffalo.edu)

Yash Nitin Mantri (yashniti@buffalo.edu)

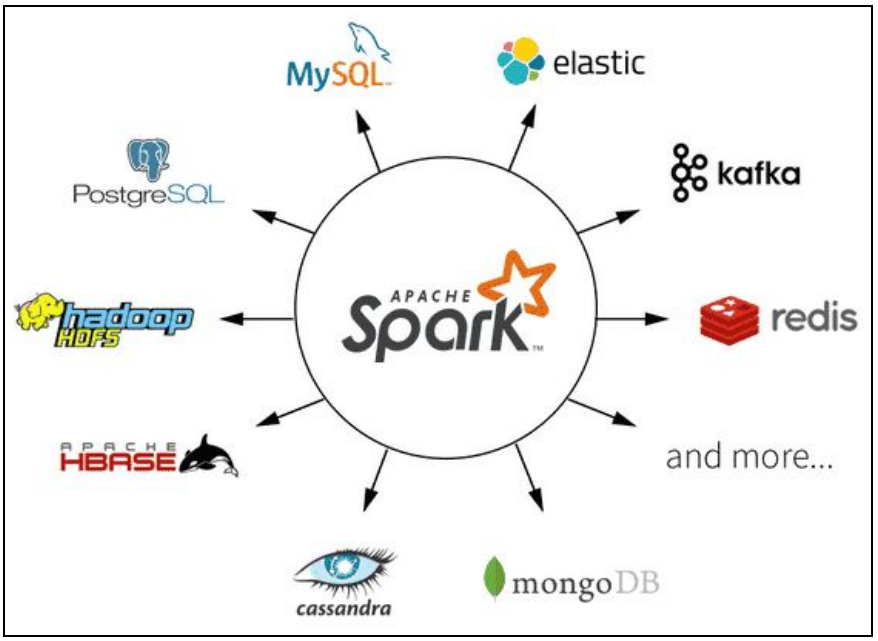
Abhishek Muni (amuni@buffalo.edu)

Guided By:

Deen Dayal Mohan

(Teaching Assistant - CSE Department, UB)

Introduction



The objective of this assignment is to understand the basics of predictive analytics with Apache Spark. The goal is to use Spark Libraries to implement an end to end Predictive Analytics Pipeline and introduce us to the data science competition platform - Kaggle. Starting off, the terminologies involved include:

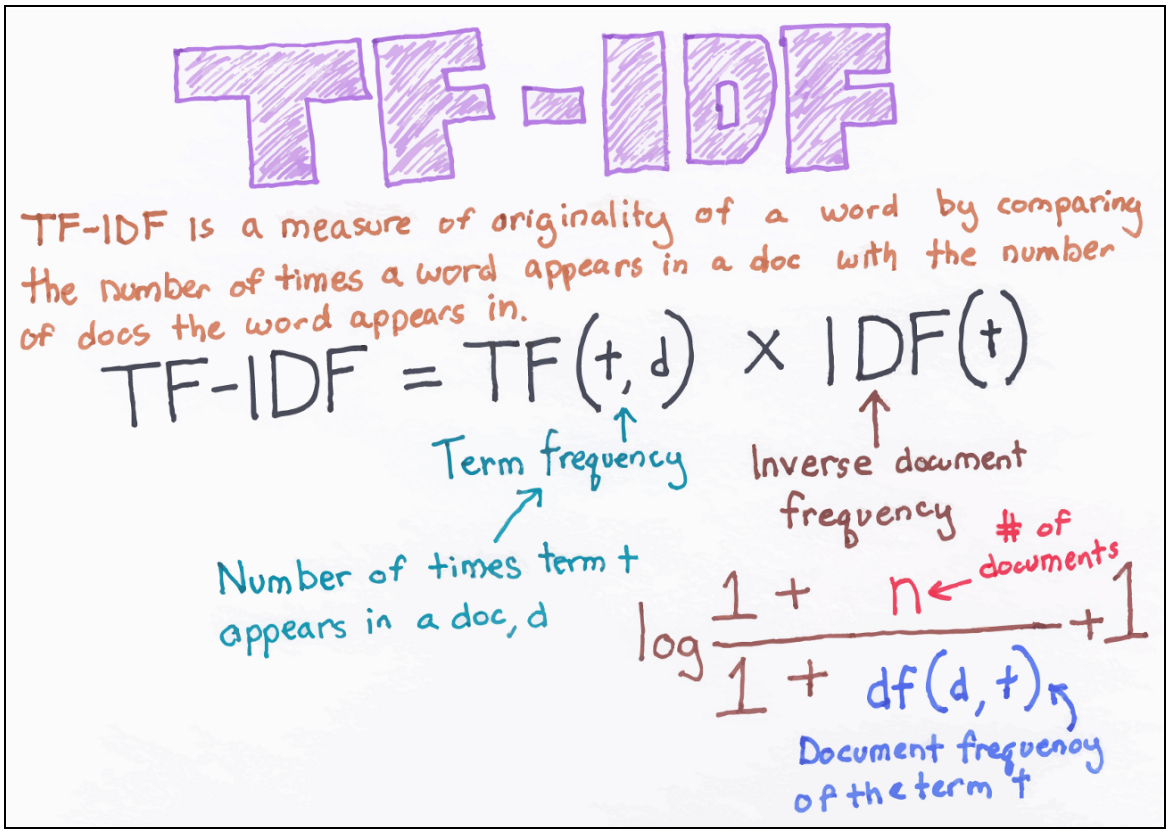
Spark:

Apache Spark is an open-source distributed,general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, **MLlib** for machine learning, GraphX for graph processing, and Spark Streaming. Apache Spark has as its architectural foundation the Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.

Pyspark: It is a Python API for Spark that lets you harness the simplicity of Python and the power of Apache Spark in order to tame Big Data.

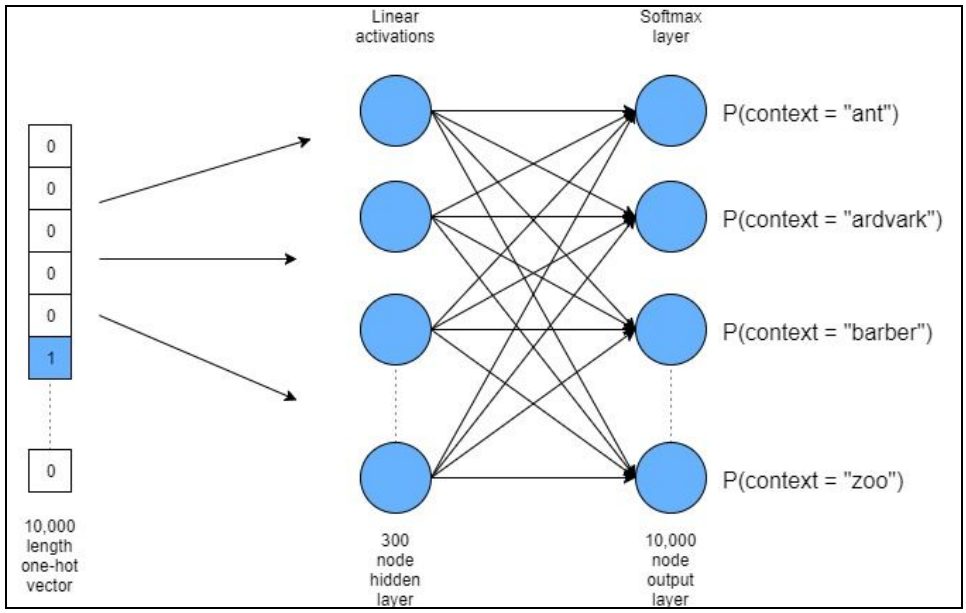
Logistic Regression: Logistic regression is a popular method to predict a categorical response. It is a special case of Generalized Linear models that predicts the probability of the outcomes. In spark.ml logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. Use the family parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant. Multinomial logistic regression can be used for binary classification by setting the family param to "multinomial". It will produce two sets of coefficients and two intercepts.

TF-IDF: Speaking of the NLP aspect of this project, in information retrieval, TF-IDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.



tf–idf is one of the most popular term-weighting schemes today. A survey conducted in 2015 showed that 83% of text-based recommender systems in digital libraries use tf–idf.

Word2vec: Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.



Approach Overview:

#Disclaimer: Our first guess was to go with Random Forest (RF)! We have attached the code for part2 - using RF. with the file name "Not_Working_RandomForest_Part2.ipynb" (Not to be considered for evaluation). RF - Part 2 was implemented to take in one genre's encoded values and predict that genre's presence in test data. This was a single-genre at a time approach. The fault with this approach was that we were unable to predict the same for multi-genre presence. As shown in the figure below, "indie" genre gets the accuracy of 0.90421 on its own, but the issue of inability to predict multi-class labels is elaborated in the figure below:

```
Accuracy for indie (genre9) prediction: 0.9042145593869731
```

Reason why the code is wrong:

The code snipped below doesnot print anything and hence it is proven that our code could not classify on the multi-label scale. All it did was predict one genre at a time.

```
In [14]: for key, value in movie_id_to_predictions.items():
             if sum(value) > 1:
                 print(f"{key}: {value}")

In [ ]:
```

We trained RF. model for each genre individually,but it didn't give good results and consumed high computational time. The reason could be that the base model was not able to handle the imbalance in the dataset which was skewed.

The final approach was based off of the detailed guide given on:
https://www.analyticsvidhya.com/blog/2019/04/predicting-movie-genres-nlp-multi-label-classification/

As for the **final route we took** - we have proceeded with using Logistic regression for all three parts (LogisticRegressionWithLBFGS), with TF-IDF (HashingTF, IDF) in part 2 and Word2Vec under pyspark.ml.feature for part 3.

```
!java -version

openjdk version "1.8.0_252"
OpenJDK Runtime Environment (build 1.8.0_252-8u252-b09-1~18.04-b09)
OpenJDK 64-Bit Server VM (build 25.252-b09, mixed mode)
```

The Java version was modified as shown in the figure.

PART -1: Basic Model

F1 - Score: 0.97170

Theory:

With the implementation of first part we understood the role of the following data-preprocessing functions:

RegexTokenizer - A regex based tokenizer that extracts tokens either by using the provided regex pattern to split the text or repeatedly matching the regex (if gaps is false).

StopWordsRemover - A feature transformer that filters out stop words from input.

CountVectorizer - 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.

```
 id | texts                           | vector
----|---------------------------------|---------------
 0  | Array("a", "b", "c")            | (3,[0,1,2],[1.0,1.0,1.0])
 1  | Array("a", "b", "b", "c", "a")  | (3,[0,1,2],[2.0,2.0,1.0])
```

collectAsMap - Return the key-value pairs in this RDD to the master as a dictionary.

Objective

The goal of this part was to deploy a Predictive Analytics model to predict the genres corresponding to all the movie_ids given. We chose to go forth with the Logistic regression model.

Working of the code:

After the data-preprocessing steps as clarified in the theory section above were performed, we used the inbuilt LogisticRegressionWithLBFGS model whose working is specified below:

Classification model trained using Multinomial Logistic Regression.

Parameters:

weights – Weights computed for every feature.

intercept – Intercept computed for this model. (Only used in Binary Logistic Regression. In Multinomial Logistic Regression, the intercepts will not bea single value, so the intercepts will be part of the weights.)

numFeatures – The dimension of the features.

numClasses – The number of possible outcomes for k classes classification problem in Multinomial Logistic Regression.

Multinomial logistic regression

Multiclass classification is supported via multinomial logistic (softmax) regression. In multinomial logistic regression, the algorithm produces K sets of coefficients, or a matrix of dimension $K \times J$ where K is the number of outcome classes and J is the number of features. If the algorithm is fit with an intercept term then a length K vector of intercepts is available.

> Multinomial coefficients are available as `coefficientMatrix` and intercepts are available as `interceptVector`.

> `coefficients` and `intercept` methods on a logistic regression model trained with multinomial family are not supported. Use `coefficientMatrix` and `interceptVector` instead.

The conditional probabilities of the outcome classes $k \in 1, 2, \ldots, K$ are modeled using the softmax function.

$$P(Y = k|\mathbf{X}, \boldsymbol{\beta}_k, \beta_{0k}) = \frac{e^{\boldsymbol{\beta}_k \cdot \mathbf{X} + \beta_{0k}}}{\sum_{k'=0}^{K-1} e^{\boldsymbol{\beta}_{k'} \cdot \mathbf{X} + \beta_{0k'}}}$$

We minimize the weighted negative log-likelihood, using a multinomial response model, with elastic-net penalty to control for overfitting.

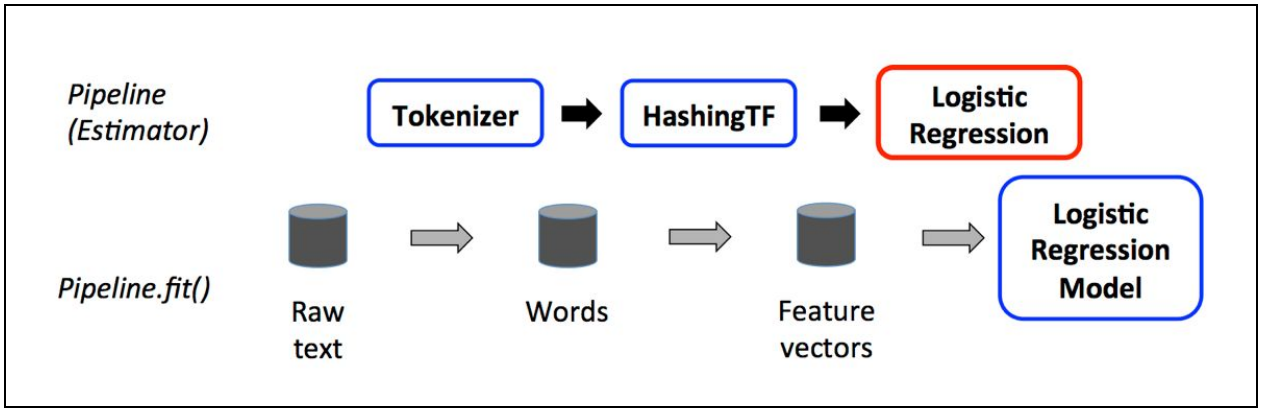
$$\min_{\beta, \beta_0} -\left[\sum_{i=1}^{L} w_i \cdot \log P(Y = y_i|\mathbf{x}_i)\right] + \lambda\left[\frac{1}{2}(1-\alpha)\,||\boldsymbol{\beta}||_2^2 + \alpha||\boldsymbol{\beta}||_1\right]$$

Part-2: Using TF-IDF to improve the model

F1 - Score: 0.97441

Working of Code:

After part-1 worked successfully, we introduced TF-IDF to improvise on the accuracy of the model. The pipeline describing the work-flow with TF-IDF incorporated is shown in the figure below:



The underlying working of TF_IDF can be understood as follows:

TF and IDF are implemented in HashingTF and IDF. HashingTF takes an RDD of the list as the input. Each record could be an iterable of strings or other types.

TF-IDF

Note We recommend using the DataFrame-based API, which is detailed in the ML user guide on TF-IDF.

Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus. Denote a term by t, a document by d, and the corpus by D. Term frequency $TF(t,d)$ is the number of times that term t appears in document d, while document frequency $DF(t,D)$ is the number of documents that contains term t. If we only use term frequency to measure the importance, it is very easy to over-emphasize terms that appear very often but carry little information about the document, e.g., "a", "the", and "of". If a term appears very often across the corpus, it means it doesn't carry special information about a particular document. Inverse document frequency is a numerical measure of how much information a term provides:

$$IDF(t,D) = \log \frac{|D|+1}{DF(t,D)+1},$$

where $|D|$ is the total number of documents in the corpus. Since logarithm is used, if a term appears in all documents, its IDF value becomes 0. Note that a smoothing term is applied to avoid dividing by zero for terms outside the corpus. The TF-IDF measure is simply the product of TF and IDF:

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D).$$

There are several variants on the definition of term frequency and document frequency. In `spark.mllib`, we separate TF and IDF to make them flexible.

Our implementation of term frequency utilizes the hashing trick. A raw feature is mapped into an index (term) by applying a hash function. Then term frequencies are calculated based on the mapped indices. This approach avoids the need to compute a global term-to-index map, which can be expensive for a large corpus, but it suffers from potential hash collisions, where different raw features may become the same term after hashing. To reduce the chance of collision, we can increase the target feature dimension, i.e., the number of buckets of the hash table. The default feature dimension is $2^{20} = 1,048,576$.

```
from pyspark.mllib.feature import HashingTF, IDF

# Load documents (one per line).
documents = sc.textFile("data/mllib/kmeans_data.txt").map(lambda line: line.split(" "))

hashingTF = HashingTF()
tf = hashingTF.transform(documents)

# While applying HashingTF only needs a single pass to the data, applying IDF needs two passes:
# First to compute the IDF vector and second to scale the term frequencies by IDF.
tf.cache()
idf = IDF().fit(tf)
tfidf = idf.transform(tf)

# spark.mllib's IDF implementation provides an option for ignoring terms
# which occur in less than a minimum number of documents.
# In such cases, the IDF for these terms is set to 0.
# This feature can be used by passing the minDocFreq value to the IDF constructor.
idfIgnore = IDF(minDocFreq=2).fit(tf)
tfidfIgnore = idfIgnore.transform(tf)
```

Part-3: Custom Feature Engineering

F1 - Score: 0.99461

Working of Code:

After part-2 worked successfully, we introduced Word2vec to further improvise on the accuracy of the model.

Word2Vec computes distributed vector representation of words. The main advantage of the distributed representations is that similar words are close in the vector space, which makes generalization to novel patterns easier and model estimation more robust. Distributed vector representation is shown to be useful in many natural language processing applications such as named entity recognition, disambiguation, parsing, tagging and machine translation.

The working of pyspark Word2vec can be explained as follows:

Model

In our implementation of Word2Vec, we used skip-gram model. The training objective of skip-gram is to learn word vector representations that are good at predicting its context in the same sentence. Mathematically, given a sequence of training words $w_1, w_2, \ldots, w_T$, the objective of the skip-gram model is to maximize the average log-likelihood

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j=-k}^{j=k}\log p(w_{t+j}|w_t)$$

where k is the size of the training window.

In the skip-gram model, every word w is associated with two vectors u_w and v_w which are vector representations of w as word and context respectively. The probability of correctly predicting word w_i given word w_j is determined by the softmax model, which is

$$p(w_i|w_j) = \frac{\exp(u_{w_i}^{\top} v_{w_j})}{\sum_{l=1}^{V}\exp(u_l^{\top} v_{w_j})}$$

where V is the vocabulary size.

The skip-gram model with softmax is expensive because the cost of computing $\log p(w_i|w_j)$ is proportional to V, which can be easily in order of millions. To speed up training of Word2Vec, we used hierarchical softmax, which reduced the complexity of computing of $\log p(w_i|w_j)$ to $O(\log(V))$

Example

The example below demonstrates how to load a text file, parse it as an RDD of `Seq[String]`, construct a `Word2Vec` instance and then fit a `Word2VecModel` with the input data. Finally, we display the top 40 synonyms of the specified word. To run the example, first download the text8 data and extract it to your preferred directory. Here we assume the extracted file is `text8` and in same directory as you run the spark shell.

Scala | **Python**

Refer to the `Word2Vec` Python docs for more details on the API.

```
from pyspark.mllib.feature import Word2Vec

inp = sc.textFile("data/mllib/sample_lda_data.txt").map(lambda row: row.split(" "))

word2vec = Word2Vec()
model = word2vec.fit(inp)

synonyms = model.findSynonyms('1', 5)

for word, cosine_distance in synonyms:
    print("{}: {}".format(word, cosine_distance))
```

References

https://www.analyticsvidhya.com/blog/2019/04/predicting-movie-genres-nlp-multi-label-classification/

https://spark.apache.org/docs/latest/

https://towardsdatascience.com/tf-term-frequency-idf-inverse-document-frequency-from-scratch-in-python-6c2b61b78558

https://www.edureka.co/blog/pyspark-programming/#Learnpysparkprogramming

https://spark.apache.org/docs/latest/ml-classification-regression.html

https://spark.apache.org/docs/latest/api/java/org/apache/spark/mllib/classification/LogisticRegressionWithLBFGS.html

https://spark.apache.org/docs/1.6.2/api/java/org/apache/spark/ml/feature/RegexTokenizer.html

https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/ml/feature/StopWordsRemover.html

https://spark.apache.org/docs/latest/ml-features#countvectorizer

https://spark.apache.org/docs/latest/ml-classification-regression.html#logistic-regression

https://spark.apache.org/docs/latest/mllib-feature-extraction.html#tf-idf

https://spark.apache.org/docs/2.2.0/mllib-feature-extraction.html#word2vec