San Francisco Crime Classification using sparkML

- S SACHIN KUMAR (PES1PG21CS032)
- M MUKESH KUMAR (PES1PG21CS022)

Overview

This simulates a real world scenario where we can handle an enormous amount of data for predictive modelling. The data source is a stream and your application faces the constraint of only being able to handle batches of a stream at any given point in time.

Goals

Analyse large data streams and process them for machine learning tasks using Spark Streaming and Spark ML Lib to draw insights and deploy models for predictive tasks

Scope

We are interested in building a system that could classify crime discription into different categories. We create a system that could automatically assign a described crime to category which could help law enforcements to assign right officers to crime or could automatically assign officers to crime based on the classification.

We are using dataset from Kaggle on San Francisco Crime. Our responsibilty is to train a model based on 39 pre-defined categories, test the model accuracy and deploy it into production. Given a new crime description, the system should assign it to one of 39 categories.

To solve this problem, we will use a variety of feature extraction techniques along with different supervised machine learning algorithms in Pyspark.

Software/Languages to be used:

- Python
- Hadoop
- Spark

Libraries Used:

- Pyspark
- Numpy

Analysis

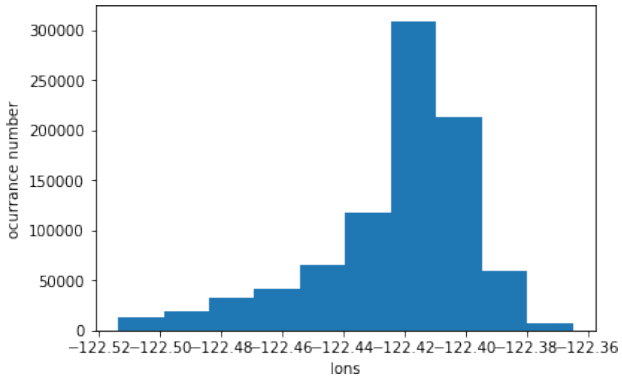
Data Exploration
Data fields
1. Dates - timestamp of the crime incident
2. Category - category of the crime incident (only in train.csv). This is the target variable that is going to be predicted.
3. Descript - detailed description of the crime incident (only in train.csv)

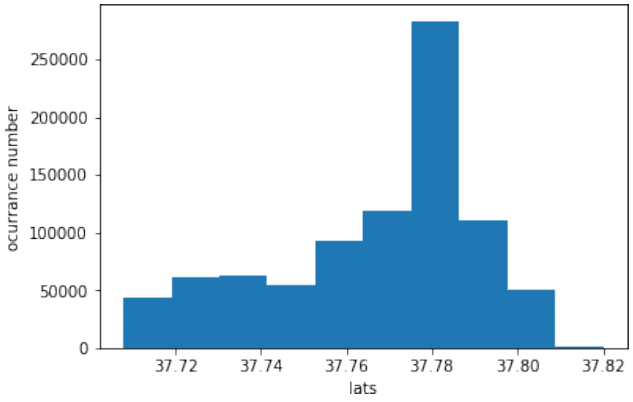
4. DayOfWeek - the day of the week
5. PdDistrict - name of the Police Department District
6. Resolution - how the crime incident was resolved (only in train.csv)
7. Address - the approximate street address of the crime incident
8. X - Longitude
9. Y – Latitude

Obserivation:

longitudes are between [-122.52, -120.5], each value different slightly from others
latitudes are between [37.708, 90]
here as shown there exist some bad outlier values -i.e. close to 90-, the reasons that this is bad that:
- First, San Fransisco latitudes are between [37.707, 37.83269] ,reference: google maps
- Second, as shown in the statistics that the most values are close to 37.7 Also, in longitudes, San Francisco longitudes are between [-122.517652, - 122.3275], from google maps

Plotting the crimes occurrences according to longitudes and latitudes:





Methodology:

1.Set up Spark and load Libraries
2.Data Extraction
3.Partition the dataset int train and test.
4. Define a structure to build a pipeline

The Process of cleaning the dataset involves:

Define tokenization function using RegexTokenizer: RegexTokenizer allows more advanced tokenization based on regular expression (regex) matching. By default, the parameter “pattern” (regex, default: “\s+”) is used as delimiters to split the input text. Alternatively, users can set parameter “gaps” to false indicating the regex “pattern” denotes “tokens” rather than splitting gaps, and find all matching occurrences as the tokenization result.

Define stop remover function using StopWordsRemover: StopWordsRemover takes as input a sequence of strings (e.g. the output of a Tokenizer) and drops all the stop words from the input sequences. The list of stopwords is specified by the stopWords parameter.

Define bag of words function for Descript variable using CountVectorizer: 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 affects 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.

Define function to Encode the values of category variable using StringIndexer: StringIndexer encodes a string column of labels to a column of label indices. The indices are in (0, numLabels), ordered by label frequencies, so the most frequent label gets index 0. In our case, the label colum(Category) will be encoded to label indices, from 0 to 38; the most frequent label (LARCENY/THEFT) will be indexed as 0.

Define a pipeline to call these functions: ML Pipelines provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines.

5. Build a multi-classification

The stages involve to perform multi-classification include:

1. Model training and evaluation
   Build baseline model

   -Logistic regression using CountVectorizer features
   We will build a model to make predictions and score on the test sets using logistics regression using the dataset we transformed using count vectors. And we will see the top 10 predictions from the highest probability from our model, accuracy and other metrics to evaluate our model.

2. Build secondary models
   - Naive Bayes
   Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. The spark.ml implementation currently supports both multinomial naive Bayes and Bernoulli naive Bayes.

   - Logistic regression and Naive Bayes using TF -IDF Features

   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:

$$IDF(t,D)=log(|D|+1DF/(t,D)+1\ ),$$

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 T 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 MLlib, we separate TF and IDF to make them flexible.

3. Apply Logistic Regression Using Word2Vec features

Word2Vec is an Estimator which takes sequences of words representing documents and trains a Word2VecModel. The model maps each word to a unique fixed-size vector. The Word2VecModel transforms each document into a vector using the average of all words in the document; this vector can then be used as features for prediction, document similarity calculations, etc.

Results:

	Logistic Regression	Naive Bayes
Count Vectoriser	97.2%	99.3%
TF-IDF	97.2%	99.5%
Word2Vec	90.7%	

Conclusion:

As you can see, TF-IDF proves to be best vectoriser for this dataset, while Naive Bayes proves to be better algorithm for text analysis than Logistic regression.