Apache Spark

Tutorial: Exploring data in DataFrames

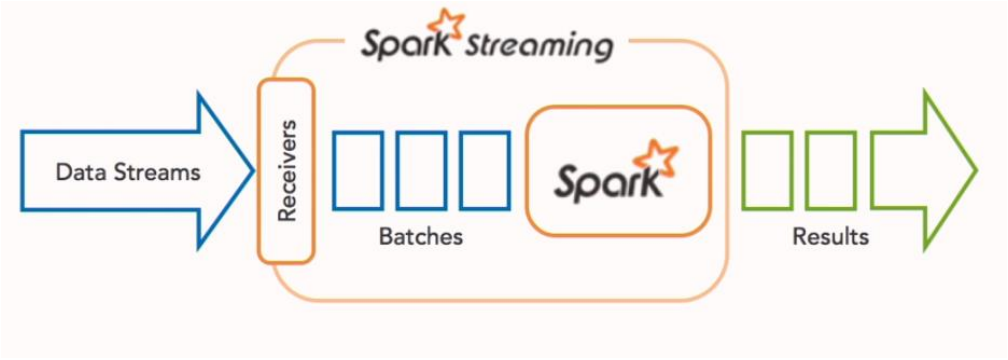
https://www.lynda.com/Apache-Spark-tutorials/Exploring-data-DataFrames/550568/581952-4.html

Background Information

- A fast and general engine for large-scale data processing.
- Advantages
    - Speed: faster than traditional mapreducer jobs on Hadoop. Because it distributes the execution across its cluster.
    - Ease of use: Supports Java, Scala, Python and R
    - Generality: supports SQL, Streaming, Machine Learning, and graphing
    - Platform agnostic: Hadoop, Hive, Cassandra

Spark

- Spark Core
    - Foundational component
    - Task distribution
    - Scheduling
    - Input/Output operations
- Spark SQL
    - DataFrames (same as tables in SQL or Pandas in Python)
- Spark Streaming
    - Streaming analytics: enables our jobs to process new data as it comes in.
    - Micro batches: does the same thing as normal batch operations, but more frequently on smaller datasets.
        - Streams of data would come in and be handled by the receivers.
        - Receivers would generate micro batches.
        - Spark would then process these jobs and then produce the results based on whatever the jobs were defined as.



    - Lambda architecture: takes historical data to start and then as new data comes in you start aggregating data
- MLlib
    - Machine Learning jobs
    - 9x faster than Apache Mahout
    - Contains a lot of common functions
- GraphX
    - Graph processing: identify entities and relationships in our data.
    - Based on RDDS

Databricks

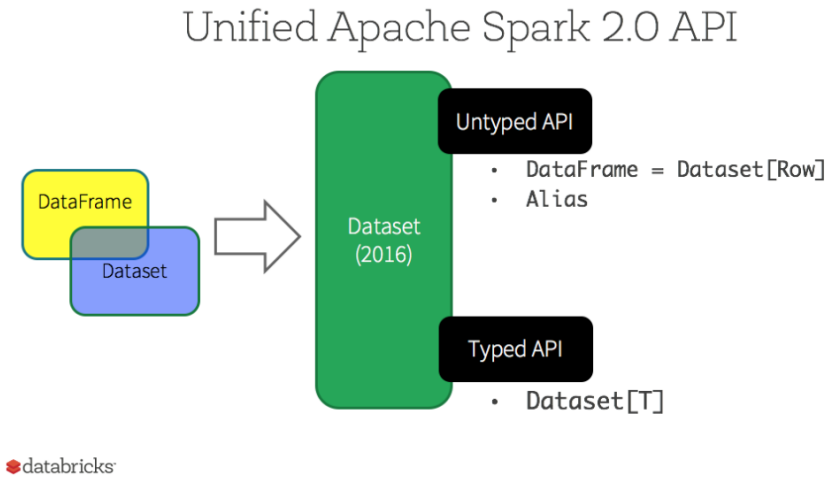
- A cloud based managed platform for running Spark

DataFrame

- Tables of data
- **DataFrame** is a distributed collection of data, which is organized into named columns.

Dataset

- A combination of RDD's and DataFrames
- It can be queried like a dataframe
- Type data like an RDD



Actions and Transformations

- Actions
    - Show
    - Count
    - Collect

    - Save
- Transformation (Lazy, similar to SQL functions)
    - Select
    - Distinct
    - groupBy
    - sum
    - orderBy
    - filter
    - limit

Types of Text Data

- Unstructured
    - Schema-never files
    - There is not schema present in that text file
    - Emals, log file, plain text
- Semi-Structured Data
    - Text based
    - Tagged or structed in a loose way
    - Represented by JSON files. Gives shape and organization to data. Dynamic and can change
    - They have a schema only after we readit from our disk
    - Called schema later files
- Structured Data
    - Easiest to work with since it has a schema in mind
    - Lives in databases and some files like comma separated values files

Data aggregation

Is any process in which information is gathered and expressed in a summary form, for purposes such as statistical analysis. A common aggregation purpose is to get more information about particular groups based on specific variables such as age, profession, or income.

DataFrameWriter

- Allows us to save a DataFrame as a table in Spark.
- If we are connected to Hive, it will create it as a managed table in the Hive metastore by default.

Tutorial: Spark for Machine Learning & AI

https://www.lynda.com/Apache-Spark-tutorials/Spark-Machine-Learning-AI/559180-2.html

What is Spark?

- Distributed data processing platform for big data
- Distributed: runs on a cluster of servers
- Processing: performs computations, such as machine learning and data analysis

- Big Data: data sets that are not easily analyzed on a single server
- Use Cases:
    - Real-time monitoring
    - Text analysis
    - Ecommerce pattern analysis
    - Healthcare and genomic analysis

Steps in Machine Learning

1. Preprocessing: collect, reformat and transform data
2. Model Building: apply machine learning algorithms to training data
3. Validation: Assess the quality of models built in step 2

- Preprocessing
    - Extract, transform, and load data to staging area
    - Review data for missing data and invalid values
    - Normalize and scale numeric data
    - Standardize categorical values
- Building Models:
    - Selecting algorithms
    - Executing algorithms to fit data to models
    - Tuning hyperparameters
- Validating Models
    - Applying models to additional test sets
    - Measuring quality of models: accuracy, precision, sensitivity

MLlib Contents

- Algorithms
    - Classification
    - Regression
    - Clustering: group similar items together
    - Topic modeling: example, identify dominant themes in a text
- Workflors
    - Feature transformations
    - Pipelines
    - Evaluations
    - Hyperparameter tuning
- Utilities
    - Distributed math libraries
    - Statistical functions

Preprocessing Types

1- Numeric
2- Text

Numeric

Normalize:

- Maps data values from their original range to the range of 0 to 1
- Avoids problems when some attributes have large ranges and others have small ranges
    - Salaries: large range
    - Years of employment: small range
- As a result, when we have salaries that range from 50,000 to 100,000 then we compare it to attributes such as distance with ranges of 10 to 100 miles, normalizing these attributes will have them on the same range.
- As a result, these features will not adversely impact our model

Standardize:

- Map data values from their original range to -1 to 1
- Mean values of 0
- Normally distributed with standard deviation of 1
- Transforms our data into a bell curve shape formation
- Used when attributes have different scales and ML algorithms assume a normal distribution, such as SVMs. Some algorithms work better when we have unit variance and zero mean.

Partitioning:

- Map data values from continuous values to buckets
- Deciles and percentiles are examples of buckets
- Useful when you want to work with groups of values instead of a continuous range of values

Text

Tokenizing:

- Map text from a single string to a set of tokens or words
- Example : This is a sentence -> [“This”, “is” , “a”, “sentence”]

TF IDF (Term Frequency - Inverse Document Frequency):

- Map text from a single, typically long string, to a vector indicating the frequency of each word in a text relative to a group of texts
- Widely used in text classification
- Infrequently used words are more useful for distinguishing categories of text
- First, we get the term frequency by tokenizing a sentence or a text. We get a list of words. Then we count the number of times the words appear.
- Then we repeat the process for all the documents in our corpus (corpus = collection of documents).
- We use these two sets of counts, the counts of the number of times the word shows up in a single document and how often those words show up across all of the documents that we're working with.

- And we feed those two numbers into the term frequency inverse document frequency calculation.
- Finally we get a TF IDF measure.

Clustering

- Clustering algorithms group data into clusters that allow us to see how large data sets can break down into subgroups
- Example, k-means algorithms
- K-means algorithms work well for small and midsize data sets
- Bisecting k-means algorithms are faster and more suitable for large data sets.

Regression Algorithms

- Linear Regression
- Decision Tree Regression
- Gradient-boosted tree regression

Spark Functions

- `emp_df = spark.read.csv(path, header = True)`
- `emp_df.columns()`
- `emp_df.take(5)`
- `emp_df.count()`
- `sample_df = emp_df.sample(False, 0.1)`
- `emp_managers_df = emp_df.filter("salary >= 100000)`

2.4: Exploring Data in DataFrames

- ```
  df = spark.read.load(path,
                       format='com.databricks.spark.csv',
                       header='true',
                       inferSchema='true')
  ```
- `df.select("Country").show()`
- ```
  display(
        df
          .select("Country")
          .distinct()
          .orderBy("Country")
  )
  ```

Remove

Remove duplicates from the column "Country" and sort by Country alphabetically

- ```
  display(
    df
      .select(df["InvoiceNo"],df["UnitPrice"]*df["Quantity"])
      .groupBy("InvoiceNo")
      .sum()
    )
  ```

  Aggregation

  Display() will show the results

  Select the invoice column, and for each invoice, multiply the unit price and the quantity in that row . To show only one number for each invoice, we will group the total price for that invoice through groupBy() and sum(). Sum() will be responsible to add up all the sales.

- ```
  df.filter(df["InvoiceNo"]==536596).show()
  ```

  Filter
  show the dataframe row entry for invoice no. 536596

- ```
  display(
    df
      .select(df["Country"],
  df["Description"],(df["UnitPrice"]*df["Quantity"]).alias("Total"))
      .groupBy("Country", "Description")
      .sum()
      .filter(df["Country"]=="United Kingdom")
      .sort("sum(Total)", ascending=False)
      .limit(10)
    )
  ```

  Aggregation and Filteration

  Select and show two columns, country and description.

  For each row, multiply the unit prices and totals, create a new column of the resultant as “Total”

  Group the quantities based on the countries and description, but sum them using sum()

  Then filter the data shown by using the double equal signs ==

  In this case == is not assigning something, rather, it ensures the data shows is United Kingdom for country

  Finally sort the data by the total descending order, and limit it to only 10 rows
  ```

Saving Results (Tables)

- r1.write.saveAsTable("product_sales_by_country")

  DataFrames can also be saved as persistent tables using the saveAsTable command

  saveAsTable will materialize the contents of the dataframe and create a pointer to the data in the HiveMetastore

Normalizing Code

- features_df = spark.createDataFrame([
  (1, Vectors.dense([10.0, 10000, 1.0]),),

  (2, Vectors.dense([20.0, 30000, 2.0]),),

  (3, Vectors.dense([30.0, 40000, 3.0]),)]

  , [“id”, “features”])

- feature_scaler = MinMaxScaler(inputCol=“features”, outputCol = “sfeatures”)
  transfer the input column, which is named features and we want that scaled version of that input column to go to a new ouput (new features) column which is called sfeatures, which is short for scaled features.

- smodel = feature_scaler.fit(features_df)
  fit our DF to the model we created above.

- sfeatures_df = smodel.transform(features_df)
  this will apply the transformation and create the scaled data set. It will give us the new scales features.

Standardizing Code

- features_df = spark.createDataFrame([
  (1, Vectors.dense([10.0, 10000, 1.0]),),

  (2, Vectors.dense([20.0, 30000, 2.0]),),

  (3, Vectors.dense([30.0, 40000, 3.0]),)]

  , [“id” , “features”])

- feature_stand_scaler = StandardScaler(inputCol=“features”, outputCol = “sfeatures”, withStd=True, withMean = True)

transfer the input column, which is named features and we want that scaled version of that input column to go to a new ouput (new features) column which is called sfeatures, which is short for scaled features. Because we want the normal curve with mean = 0 and a std. dev = 1, we set the parameters as above.

- stand_smodel = feature_stand_scaler.fit(features_df)
  fit our DF to the model we created above.
- stand_sfeatures_df = stand_smodel.transform(features_df)
  this will apply the transformation and create the scaled data set. It will give us the new scales features.

Bucketizing Code

- splits = [-float("inf"), -10.0, 0.0, 10.0, float("inf")]
  create 4 buckets from -inf to -10, then -10 to 0, then 0 to 10, then 10 to +inf

- b_data = [(-800.0,), (-10.5,), (-1.7,), (0.0,), (8.2,), (90.1,)]
  the parenthesis are used because we want to map each element to a row in the data frame later

- b_df = spark.createDataFrame(b_data, ["features"])
  create a bucket data frame with the data (b_data) called features

- bucketizer = Bucketizer(splits = splits , inputCol="features", outputCol="bfeatures")
  bucket object that maps a column of continuous features to a column of feature buckets.

- bucketed_df = bucketizer.transform(b_df)
  we don't need to fit() when bucketizing because the splits is a list of boundaries that we want for each bucket, so there is no need to fit.

- bucketed_df.show()
  will show how each data is "mapped" in which bucket, -800 and – 10.5 will be in the first bucket , whereas -1.7 will be in the second bucket, and so on.

Tokenizing Code

- sentences_df = spark.createDataFrame([
  (1, "This is an introduction to Spark MLlib"),
  (2, "MLlib includes libraries for classification and regression"),
  (3, "It also contains supporting tools for pipelines")],

["id", "sentence"])
Sentence dataframe

- sent_token = Tokenizer(inputCol = "sentence" , outputCol = "words")
  Create a tokenizer object, which will give us a new feature column called words.
  Definition: A tokenizer that converts the input string to lowercase and then splits it by white spaces.

- sent_tokenized_df = sent_token.transform(sentences_df)
  we don't need a fit function, because we are not fitting data. the tokenizer will take each sentence and split the string into separate words.

- sent_tokenized_df.show()

TF IDF Code

- sentences_df = spark.createDataFrame([
  (1, "This is an introduction to Spark MLlib"),
  (2, "MLlib includes libraries for classification and regression"),
  (3, "It also contains supporting tools for pipelines")],
  ["id", "sentence"])
  Sentence dataframe

- sent_token = Tokenizer(inputCol = "sentence" , output = "words")

- sent_tokenized_df = sent_token.transform(sentences_df)

- hashingTF = HashingTF(inputCol = "words" , outputCol= "rawFeatures", numFeatures = 20)
  Maps a sequence of terms to their term frequencies using the hashing trick. We need to tell it how many features we want to keep track of.

- sent_hfTF_df = hashingTF.transform(sent_tokenized_df)
  transform the data, take in the tokenized sentences and give us a hashmap (dictionary) with frequencies

- sent_hfTF_df.take(1)

  [Row(id=1, sentence='This is an introduction to Spark MLlib',
  words=['this', 'is', 'an', 'introduction', 'to', 'spark', 'mllib'],
  rawFeatures=SparseVector(20, {1: 2.0, 5: 1.0, 6: 1.0, 8: 1.0, 12: 1.0, 13: 1.0}))]

Return a map of each word, represented by an index (e.g. this = 1), and the frequency?

- idf = IDF(inputCol = “rawFeatures”, outputCol= “idf_features”)

  Compute the Inverse Document Frequency (IDF) given a collection of documents.

  This will scale the rawFeature vector values based on how often the words appear in the entire collection of sentences.

- idfModel = idf.fit(sent_hfTF_df)

  to create a new dataframe with IDF features column. So to do that I'm going to specify tfidf_df. this is our dataframe that has both the term frequency and the inverse document frequency transformations applied, and I'm going to create that by using the idfModel.

- tfidf_df = idfModel.transform(sent_hfTF_df)

K-means Clustering Code

- cluster_df = spark.read.csv(“C:\Users\Ahmed\Downloads\Ex_Files_Spark_ML_AI\Exercise Files\Ch03\03_02\clustering_dataset.csv” , header=True, interSchema=True)
  create a df from the csv file, with the header option true and also infer the schema of the CSV file.

- cluster_df
  verify that it has 3 columns as is in csv file

- cluster_df.show(75)
  will show the 3 columns. Each 25 rows are clustered into a range of numbers

- vectorAssembler = VectorAssembler(inputCols = [“col1”, “col2”, “col3”], outputCol = “features”)
  A feature transformer that merges multiple columns into a vector column.

- vcluster_df = vectorAssembler.transform(cluster_df)
  vcluster = vectorized cluster data frame
  the transformer will give us a new feature column = “features” which is needed because
  the k means algorithm will work with the new column

- kmeans = KMeans().setK(3)
  create an object kmeans , which will have a cluster of 3

- kmeans = kmeans.setSeed(1)
  this will determine where the kmeans algorithm starts
  will give us consistency during testing

- kmodel = kmeans.fit(vcluster_df)
  create a model called kmodel which will be built using KMeans and the data to fit to this model is vcluster_df , because it contains the feature vector

- centers = kmodel.clusterCenters()
  explore the centers of the model, for each cluster. It will give 3 centers for each cluster. It will give for each 25 rows, for each 3 clusters in this exercise.

Naïve Bayes and Preprocessing Iris Data Code

- iris_df = spark.read.csv(“"../data/clustering_dataset.csv"”, inferSchema=True)

- iris_df.take(1)
  [Row(_c0=5.1, _c1=3.5, _c2=1.4, _c3=0.2, _c4='Iris-setosa')]

- iris_df = iris_df.select(col(“_c0”).alias(“sepal_length”),
  col(“_c1”).alias(“sepal_width”),
  col(“_c2”).alias(“petal_length”),
  col(“_c3”).alias(“petal_width”),
  col(“_c4”).alias(“species”)
)
  Change the number of each column to more meaningful names

- vectorAssembler = VectorAssembler(inputCols=[“sepal_length”, “sepal_width”, “petal_length”, “petal_width”], outputCol = “features”)
  merges the columns into a single feature column called “features”

- viris_df = vectorAssembler.transform(iris_df)
  transforms the features into the feature column as specified in the previous code

- viris_df.take(1)

- indexer = StringIndexer(inputCol = “species”, outputCol= “label”)

create an indexer using the transformation StringIndexer

input column is species, because that's the string

And, our output column, will be called label

- iviris_df = indexer.fit(viris_df).transform(viris_df)

  create a data frame that captures this indexed value

- iviris_df.take(1)
  view the new indexed and vectorized data frame created

- splits = iviris_df.randomSplit([0.6, 0.4],1)
  split the data frame into one set with 60% of the data, and the other with 40% of the data. “1” for the seed

- train_df = splits[0]

- test_df = splits[1]

- nb = NaiveBayes(modelType = “multinomial”)
  create a Naïve Bayes model, and instead of binary model, since we have more than 2 labels, we will choose multinomial labels

- nbmodel = nb.fit(train_df)
  fit the data to the model

- predictions_df = nbmodel.transform(test_df)
  once we built the model and fit it with our training data, we can use the model to make predictions. To do so, we can transform the test data on the nbmodel we created.

- predictions_df.take(1)
  take a look at the dataframe now, there will be some columns added with a final column called “label” , in this case the label is 0.0 which means the model predicts that the test example passed belongs to the first iris species

- evaluator = MulticlassClassificationEvaluator(labelCol= “label” , predictionCol = “prediction” , metricName= “accuracy”)
  Evaluator for Multiclass Classification, which expects two input columns: prediction and label.
  MetricName is accuracy because we trying to measure the accuracy between the actual labels and predictions.

- nbaccuracy = evaluator.evaluate(predictions_df)

will take in the prediction column from the predictions dataframe and evaluate the accuracy versus the label column

- nbaccuracy
  will show the accuracy, which is in this case, is ~58%

Multi-layer perceptron Code

- layers = [4,5,5,3]
  the first layer is 4, because we have 4 inputs. The last layer is 3, because we have three types of labels (3 types of iris to classify test data).
  We have two "hidden" layers or middle layers, each with 5 neurons.

- mlp = MultilayerPerceptronClassifier(layers = layers, seed=1)

  mlp will be our instance of the multi-layer perceptron classifier

  the layers mlp will have are determined by the layers list we created

  the seed is set to 1, since multi-layer perceptron uses a random number generator

- mlp_model = mlp.fit(train_df)
  we will have an 'mlp' model that will fit the training data to itself

- mlp_predictions = mlp_model.transform(test_df)
  to make predictions, we will use the model we built with the training data, and transform it with the test data to make predictions.

- mlp_evaluator = MulticlassClassificationEvaluator(metricName= "accuracy")
  create an evaluator to evaluate multiclasses, the metric name is accuracy, since we want to measure the accuracy of the predictions.

- mlp_accuracy = mlp_evaluator.evaluate(mlp_predictions)
  mlp accuracy will hold the accuracy for our model. We call the mlp_evaluator object to evaluate our mlp_predictions.

- mlp_accuracy
  ~93%