PySpark Notes

1. Creating new column or operating on existing column

- .withColumn() method, which takes two arguments. First, a string with the name of your new column, and second the new column itself.

##Partitions

```
# Check the number of partitions in fileRDD
print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD_part from file_path with 5 partitions
fileRDD_part = sc.textFile(file_path, minPartitions = 5)

# Check the number of partitions in fileRDD_part
print("Number of partitions in fileRDD_part is",
fileRDD_part.getNumPartitions())
```

2. Basic RDD Operations
- Transformations (Map, filter, flatmap,union)
- Actions
  (collect(),take(),first() ,count(),countByKey(),reduceByKey(),SortByKey(),reduce(): aggregates elements of rdd, saveAsTextFile(): saves rdd into text file with each partition as a seperate file, coalsce(): saves as a single file)

3. Pair RDDs
   3.1) reduceBykey(): It combines value with the same key.
   3.2) sortBykey() : orders pair rdd by keys.
   3.3) groupByKey(): groups by key
   3.4) join(): joins two Rdds based on their keys

4. DataFrames on Spark

dataFrame Transformations : elect(), filter(), groupBy(), dropDuplicates(), withColumnRenamed()
dataframe actions : printSchema(), head(), show(), count() and describe()

5. Saprk SQL

- Creating a temporary view : df.createOrReplaceTempView()

```
query = """ SELECT ____________"""
spark.sql(query)
```

6. Data Visualization : In spark, we have HandySpark pacakage designed to improve user experience. syntax: df.toHandy()

5. filter()

The .filter() method takes either an expression that would follow the WHERE clause of a SQL expression as a string, or a Spark Column of boolean (True/False) values.

```
long_flights1 = flights.filter("distance > 1000")
```

6. select()

The Spark variant of SQL's SELECT is the .select() method. This method takes multiple arguments - one for each column you want to select. These arguments can either be the column name as a string (one for each column) or a column object (using the df.colName syntax)

```
temp = flights.select(flights.origin, flights.dest, flights.carrier)
```

7. max() and min()

```
# Find the shortest flight from PDX in terms of distance
flights.filter(flights.origin == "PDX").groupBy().min("distance").show()

# Find the longest flight from SEA in terms of air time
flights.filter(flights.origin == "SEA").groupBy().max("air_time").show()
```

8. ML Pipelines

At the core of the pyspark.ml module are the Transformer and Estimator classes. Almost every other class in the module behaves similarly to these two basic classes.

Transformer classes have a .transform() method that takes a DataFrame and returns a new DataFrame; usually the original one with a new column appended. For example, you might use the class Bucketizer to create discrete bins from a continuous feature or the class PCA to reduce the dimensionality of your dataset using principal component analysis.

Estimator classes all implement a .fit() method. These methods also take a DataFrame, but instead of returning another DataFrame they return a model object. This can be something like a StringIndexerModel for including categorical data saved as strings in your models, or a RandomForestModel that uses the random forest algorithm for classification or regression.

9. Renaming columns & joins

```
# Rename year column
planes = planes.withColumnRenamed('year','plane_year')
```

```
# Join the DataFrames
model_data = flights.join(planes, on='tailnum', how="leftouter")
```

10. Casting:
```
#Cast the columns to integers
model_data = model_data.withColumn("arr_delay",
model_data.arr_delay.cast("integer"))
model_data = model_data.withColumn("air_time",
model_data.air_time.cast("integer"))
```

11. Encoding :The first step to encoding your categorical feature is to create a StringIndexer. Members of this class are Estimators that take a DataFrame with a column of strings and map each unique string to a number. Then, the Estimator returns a Transformer that takes a DataFrame, attaches the mapping to it as metadata, and returns a new DataFrame with a numeric column corresponding to the string column.

The second step is to encode this numeric column as a one-hot vector using a OneHotEncoder. This works exactly the same way as the StringIndexer by creating an Estimator and then a Transformer. The end result is a column that encodes your categorical feature as a vector that's suitable for machine learning routines!

```
# Create a StringIndexer
carr_indexer = StringIndexer(inputCol='carrier',outputCol='carrier_index')

# Create a OneHotEncoder
carr_encoder =
OneHotEncoder(inputCol='carrier_index',outputCol='carrier_fact')
```

12. **Assemble a vector**

The last step in the Pipeline is to combine all of the columns containing our features into a single column. This has to be done before modeling can take place because every Spark modeling routine expects the data to be in this form. You can do this by storing each of the values from a column as an entry in a vector. Then, from the model's point of view, every observation is a vector that contains all of the information about it and a label that tells the modeler what value that observation corresponds to. Because of this, the pyspark.ml.feature submodule contains a class called VectorAssembler. This Transformer takes all of the columns you specify and combines them into a new vector column.

```
# Make a VectorAssembler
vec_assembler = VectorAssembler(inputCols=["month", "air_time",
"carrier_fact", "dest_fact", "plane_age"], outputCol="features")
```

13. **Create the pipeline**
You're finally ready to create a Pipeline!
Pipeline is a class in the pyspark.ml module that combines all the Estimators and Transformers that you've already created. This lets you reuse the same modeling process over and over again by wrapping it up in one simple object. Neat, right?

```
# Import Pipeline
from pyspark.ml import Pipeline

# Make the pipeline
flights_pipe = Pipeline(stages=[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler])
```

14. Fit & Transform

```
# Fit and transform the data
piped_data = flights_pipe.fit(model_data).transform(model_data)
```

15. Random split

```
# Split the data into training and test sets
training, test = piped_data.randomSplit([.6,.4])
```

16. Logistic Regression

```
# Import LogisticRegression
from pyspark.ml.classification import LogisticRegression

# Create a LogisticRegression Estimator
lr = LogisticRegression()
```

17. Evaluator
The first thing you need when doing cross validation for model selection is a way to compare different models. Luckily, the pyspark.ml.evaluation submodule has classes for evaluating different kinds of models. Your model is a binary classification model, so you'll be using the BinaryClassificationEvaluator from the pyspark.ml.evaluation module.
This evaluator calculates the area under the ROC. This is a metric that combines the two kinds of errors a binary classifier can make (false positives and false negatives) into a simple number. You'll learn more about this towards the end of the chapter!

```
# Import the evaluation submodule
import pyspark.ml.evaluation as evals

# Create a BinaryClassificationEvaluator
```

```
evaluator =
evals.BinaryClassificationEvaluator(metricName='areaUnderROC')
```

18.Make a grid

Next, you need to create a grid of values to search over when looking for the optimal hyperparameters. The submodule pyspark.ml.tuning includes a class called ParamGridBuilder that does just that (maybe you're starting to notice a pattern here; PySpark has a submodule for just about everything!).

You'll need to use the .addGrid() and .build() methods to create a grid that you can use for cross validation.

```
# Import the tuning submodule
import pyspark.ml.tuning as tune

# Create the parameter grid
grid = tune.ParamGridBuilder()

# Add the hyperparameter
grid = grid.addGrid(lr.regParam, np.arange(0, .1, .01))
grid = grid.addGrid(lr.elasticNetParam, [0,1])

# Build the grid
grid = grid.build()
```

19. **Make the validator**

The submodule pyspark.ml.tuning also has a class called CrossValidator for performing cross validation. This Estimator takes the modeler you want to fit, the grid of hyperparameters you created, and the evaluator you want to use to compare your models.

```
# Create the CrossValidator
cv = tune.CrossValidator(estimator=lr,
            estimatorParamMaps=grid,
            evaluator=evaluator
            )
```

20. **it the model(s)**

```
# Call lr.fit()
best_lr = lr.fit(training)

# Print best_lr
print(best_lr)
```

21. **Evaluate the model**

```
# Use the model to predict the test set
test_results = best_lr.transform(test)
```

```
# Evaluate the predictions
print(evaluator.evaluate(test_results))
```

22. **Loading the list in pyspark**

```
spark_data = sc.parallelize(numb)
```