Exercise on Introduction to PySpark.

1. Get to know the `SparkContext`.

- Call `print()` on `sc` to verify there's a `SparkContext` in your environment.
- `print()` `sc.version` to see what version of Spark is running on your cluster.

2. Import `SparkSession` from `pyspark.sql`.

- Make a new `SparkSession` called `my_spark` using `SparkSession.builder.getOrCreate()`.
- Print `my_spark` to the console to verify it's a `SparkSession`.

3. See what tables are in your cluster by calling `spark.catalog.listTables()` and printing the result!
4. Use the `.sql()` method to get the first 10 rows of the `flights` table and save the result to `flights10`. The variable `query` contains the appropriate SQL query.

   Use the DataFrame method `.show()` to print `flights10`.

5. Run the query using the `.sql()` method. Save the result in `flight_counts`.

- Use the `.toPandas()` method on `flight_counts` to create a `pandas` DataFrame called `pd_counts`.
- Print the `.head()` of `pd_counts` to the console.

6. The code to create a `pandas` DataFrame of random numbers has already been provided and saved under `pd_temp`.

- Create a Spark DataFrame called `spark_temp` by calling the `.createDataFrame()` method with `pd_temp` as the argument.
- Examine the list of tables in your Spark cluster and verify that the new DataFrame is *not* present. Remember you can use `spark.catalog.listTables()` to do so.
- Register `spark_temp` as a temporary table named `"temp"` using the `.createOrReplaceTempView()` method. Remember that the table name is set including it as the only argument!

- Examine the list of tables again!

7. Use the `.read.csv()` method to create a Spark DataFrame called `airports`
    - The first argument is `file_path`
    - Pass the argument `header=True` so that Spark knows to take the column names from the first line of the file.
- Print out this DataFrame by calling `.show()`.

8. Use the `spark.table()` method with the argument `"flights"` to create a DataFrame containing the values of the `flights` table in the `.catalog`. Save it as `flights`.

- Show the head of `flights` using `flights.show()`. The column `air_time` contains the duration of the flight in minutes.
- Update `flights` to include a new column called `duration_hrs`, that contains the duration of each flight in hours.

9. Use the `.filter()` method to find all the flights that flew over 1000 miles two ways:
    - First, pass a SQL **string** to `.filter()` that checks whether the distance is greater than 1000. Save this as `long_flights1`.
    - Then pass a column of boolean values to `.filter()` that checks the same thing. Save this as `long_flights2`.
- Use `.show()` to print heads of both DataFrames and make sure they're actually equal!

10. Select the columns `tailnum`, `origin`, and `dest` from `flights` by passing the column names as strings. Save this as `selected1`.

- Select the columns `origin`, `dest`, and `carrier` using the `df.colName` syntax and then filter the result using both of the filters already defined for you (`filterA` and `filterB`) to only keep flights from SEA to PDX. Save this as `selected2`.

11. Create a table of the average speed of each flight both ways.

- Calculate average speed by dividing the `distance` by the `air_time` (converted to hours). Use the `.alias()` method name this column `"avg_speed"`. Save the output as the variable `avg_speed`.
- Select the columns `"origin", "dest", "tailnum"`, and `avg_speed` (without quotes!). Save this as `speed1`.
- Create the same table using `.selectExpr()` and a string containing a SQL expression. Save this as `speed2`.

12. Find the length of the shortest (in terms of distance) flight that left PDX by first `.filter()`ing and using the `.min()` method. Perform the filtering by referencing the column directly, not passing a SQL string.

- Find the length of the longest (in terms of time) flight that left SEA by `filter()`ing and using the `.max()` method. Perform the filtering by referencing the column directly, not passing a SQL string.

13. Use the `.avg()` method to get the average air time of Delta Airlines flights (where the `carrier` column has the value `"DL"`) that left SEA. The place of departure is stored in the column `origin`. `show()` the result.

- Use the `.sum()` method to get the total number of hours all planes in this dataset spent in the air by creating a column called `duration_hrs` from the column `air_time`. `show()` the result.

14. Create a DataFrame called `by_plane` that is grouped by the column `tailnum`.

- Use the `.count()` method with no arguments to count the number of flights each plane made.
- Create a DataFrame called `by_origin` that is grouped by the column `origin`.
- Find the `.avg()` of the `air_time` column to find average duration of flights from PDX and SEA.

15. Import the submodule `pyspark.sql.functions` as `F`.

- Create a `GroupedData` table called `by_month_dest` that's grouped by both the `month` and `dest` columns. Refer to the two columns by passing both strings as separate arguments.
- Use the `.avg()` method on the `by_month_dest` DataFrame to get the average `dep_delay` in each month for each destination.
- Find the standard deviation of `dep_delay` by using the `.agg()` method with the function `F.stddev()`.

16. Examine the `airports` DataFrame by calling `.show()`. Note which key column will let you join `airports` to the `flights` table.

- Rename the `faa` column in `airports` to `dest` by re-assigning the result of `airports.withColumnRenamed("faa", "dest")` to `airports`.
- Join the `flights` with the `airports` DataFrame on the `dest` column by calling the `.join()` method on `flights`. Save the result as `flights_with_airports`.
    - The first argument should be the other DataFrame, `airports`.
    - The argument `on` should be the key column.
    - The argument `how` should be `"leftouter"`.

- Call `.show()` on `flights_with_airports` to examine the data again. Note the new information that has been added.

17. First, rename the `year` column of `planes` to `plane_year` to avoid duplicate column names.

- Create a new DataFrame called `model_data` by joining the `flights` table with `planes` using the `tailnum` column as the key.

18. Use the method `.withColumn()` to `.cast()` the following columns to type `"integer"`. Access the columns using the `df.col` notation:
    - `model_data.arr_delay`
    - `model_data.air_time`
    - `model_data.month`
    - `model_data.plane_year`

19. Create the column `plane_age` using the `.withColumn()` method and subtracting the year of manufacture (column `plane_year`) from the year (column `year`) of the flight.
20. Use the `.withColumn()` method to create the column `is_late`. This column is equal to `model_data.arr_delay > 0`.
    - Convert this column to an integer column so that you can use it in your model and name it `label` (this is the default name for the response variable in Spark's machine learning routines).
    - Filter out missing values (this has been done for you).
21. Create a `StringIndexer` called `carr_indexer` by calling `StringIndexer()` with `inputCol="carrier"` and `outputCol="carrier_index"`.

    - Create a `OneHotEncoder` called `carr_encoder` by calling `OneHotEncoder()` with `inputCol="carrier_index"` and `outputCol="carrier_fact"`.

22. Create a `StringIndexer` called `dest_indexer` by calling `StringIndexer()` with `inputCol="dest"` and `outputCol="dest_index"`.

    - Create a `OneHotEncoder` called `dest_encoder` by calling `OneHotEncoder()` with `inputCol="dest_index"` and `outputCol="dest_fact"`.

23. Create a `VectorAssembler` by calling `VectorAssembler()` with the `inputCols` names as a list and the `outputCol` name `"features"`.

    - The list of columns should be `["month", "air_time", "carrier_fact", "dest_fact", "plane_age"]`.

24. Import `Pipeline` from `pyspark.ml`.

    - Call the `Pipeline()` constructor with the keyword argument `stages` to create a `Pipeline` called `flights_pipe`.
        - `stages` should be a list holding all the stages you want your data to go through in the pipeline. Here this is just: `[dest_indexer, dest_encoder, carr_indexer, carr_encoder, vec_assembler]`

25. Create the DataFrame `piped_data` by calling the `Pipeline` methods `.fit()` and `.transform()` in a chain. Both of these methods take `model_data` as their only argument.
26. Create the DataFrame `piped_data` by calling the `Pipeline` methods `.fit()` and `.transform()` in a chain. Both of these methods take `model_data` as their only argument.
27. Use the DataFrame method `.randomSplit()` to split `piped_data` into two pieces, `training` with 60% of the data, and `test` with 40% of the data by passing the list `[.6, .4]` to the `.randomSplit()` method.
28. Import the `LogisticRegression` class from `pyspark.ml.classification`.

    Create a `LogisticRegression` called `lr` by calling `LogisticRegression()` with no arguments

29. Import the submodule `pyspark.ml.evaluation` as `evals`.

- Create `evaluator` by calling `evals.BinaryClassificationEvaluator()` with the argument `metricName="areaUnderROC"`.

30. Import the submodule `pyspark.ml.tuning` under the alias `tune`.

- Call the class constructor `ParamGridBuilder()` with no arguments. Save this as `grid`.
- Call the `.addGrid()` method on `grid` with `lr.regParam` as the first argument and `np.arange(0, .1, .01)` as the second argument. This second call is a function from the `numpy` module (imported `as np`) that creates a list of numbers from 0 to .1, incrementing by .01. Overwrite `grid` with the result.
- Update `grid` again by calling the `.addGrid()` method a second time create a grid for `lr.elasticNetParam` that includes only the values `[0, 1]`.
- Call the `.build()` method on `grid` and overwrite it with the output.

31. Create a `CrossValidator` by calling `tune.CrossValidator()` with the arguments:

    - `estimator=lr`
    - `estimatorParamMaps=grid`
    - `evaluator=evaluator`
- Name this object `cv`.

32. Create `best_lr` by calling `lr.fit()` on the `training` data.

- Print `best_lr` to verify that it's an object of the `LogisticRegressionModel` class.

33.

```
# Fit cross validation models
models = cv.fit(training)

# Extract the best model
best_lr = models.bestModel
```

Print `best_lr` to verify that it's an object of the `LogisticRegressionModel` class

34. Use your model to generate predictions by applying `best_lr.transform()` to the `test` data. Save this as `test_results`.

- Call `evaluator.evaluate()` on `test_results` to compute the AUC. Print the output.