Exercise on Feature Engineering with PySpark

1. Use the `parquet()` file reader to read in `'Real_Estate.parq'` as described in the video exercise.

- Print out the list of columns with `columns`.

2.

- `TAXES`
- `SALESCLOSEPRICE`
- `DAYSONMARKET`
- `LISTPRICE`
- From the listed columns above, identify which one we will use as our dependent variable `$Y$`.
- Using the loaded data set `df`, filter it down to our dependent variable with `select()`. Store this dataframe in the variable `Y_df`.
- Display summary statistics for the dependent variable using `describe()` on `Y_df` and calling `show()` to display it.

3. Create a data validation function `check_load()` with parameters `df` a dataframe, `num_records` as the number of records and `num_columns` the number of columns.

- Using `num_records` create a check to see if the input dataframe `df` has the same amount with `count()`.
- Compare input number of columns the input dataframe has with`num_columns` by using `len()` on `columns`.
- If both of these return `True`, then print `Validation Passed`

4. Using `df` create a list of attribute and datatype tuples with `dtypes` called `actual_dtypes_list`.

- Iterate through `actual_dtypes_list`, checking if the column names exist in the dictionary of expected dtypes `validation_dict`.

- For the keys that exist in the dictionary, check their dtypes and print those that match.

5. Use a `for` loop iterate through the `columns`.

- In each loop cycle, compute the correlation between the current column and `'SALESCLOSEPRICE'` using the `corr()` method.

- Create logic to update the maximum observed correlation and with which column.

- Print out the name of the column that has the maximum correlation with `'SALESCLOSEPRICE'`.

6. Sample 50% of the dataframe `df` with `sample()` making sure to not use replacement and setting the random seed to 42.

- Convert the Spark DataFrame to a `pandas.DataFrame()` with `toPandas()`.
- Plot a distribution plot using `seaborn`'s `distplot()` method.
- Import the `skewness()` function from `pyspark.sql.functions` and compute it on the aggregate of the `'LISTPRICE'` column with the `agg()` method. Remember to `collect()` your result to evaluate the computation.

7. Using the loaded data set `df` filter it down to the columns 'SALESCLOSEPRICE' and 'LIVINGAREA' with `select()`.

- Sample 50% of the dataframe with `sample()` making sure to not use replacement and setting the random seed to 42.
- Convert the Spark DataFrame to a `pandas.DataFrame()` with `toPandas()`.
- Using 'SALESCLOSEPRICE' as your dependent variable and 'LIVINGAREA' as your independent, plot a linear model plot using seaborn `lmplot()`.

8.

- `'STREETNUMBERNUMERIC'`: The postal address number on the home
- `'FIREPLACES'`: Number of Fireplaces in the home
- `'LOTSIZEDIMENSIONS'`: Free text describing the lot shape
- `'LISTTYPE'`: Set list of values of sale type
- `'ACRES'`: Numeric area of lot size

- Read the list of column descriptions above and explore their top 30 values with `show()`, the dataframe is already filtered to the listed columns as `df`
- Create a list of two columns to drop based on their lack of relevance to predicting house prices called `cols_to_drop`. Recall that computers only interpret numbers explicitly and don't understand context.
- Use the `drop()` function to remove the columns in the list `cols_to_drop` from the dataframe `df`.

9. Use `select()` and `show()` to inspect the distinct values in the column `'ASSUMABLEMORTGAGE'` and create the list `yes_values` for all the values containing the string `'Yes'`.

- Use `~df['ASSUMABLEMORTGAGE']`, `isin()`, and `.isNull()` to create a NOT filter to remove records containing corresponding values in the list `yes_values` and to keep records with null values. Store this filter in the variable `text_filter`.
- Use `where()` to apply the `text_filter` to `df`.
- Print out the number of records remaining in `df`.

10. Import `mean()` and `stddev()` from `pyspark.sql.functions`.

- Use `agg()` to calculate the mean and standard deviation for `'log_SalesClosePrice'` with the imported functions.
- Create the upper and lower bounds by taking `mean_val` +/- 3 times `stddev_val`.
- Create a `where()` filter for `'log_SalesClosePrice'` using both `low_bound` and `hi_bound`.

11. Calculate the max and min of `DAYSONMARKET` and put them into variables `max_days` and `min_days`, don't forget to use `collect()` on `agg()`.

- Using `withColumn()` create a new column called 'percentage*scaled*days' based on `DAYSONMARKET`.
- `percentage_scaled_days` should be a column of integers ranging from 0 to 100, use `round()` to get integers.
- Print the `max()` and `min()` for the new column `percentage_scaled_days`.

12. Define a function called `min_max_scaler` that takes parameters `df` a dataframe and `cols_to_scale` the list of columns to scale.

- Use a `for` loop to iterate through each column in the list and minmax scale them.
- Return the dataframe `df` with the new columns added.
- Apply the function `min_max_scaler()` on `df` and the list of columns `cols_to_scale`.

13. Use the aggregate function `skewness()` to verify that `'YEARBUILT'` has negative skew.

- Use the `withColumn()` to create a new column `'Reflect_YearBuilt'` and reflect the values of `'YEARBUILT'`.
- Using `'Reflect_YearBuilt'` column, create another column `'adj_yearbuilt'` by taking 1/log() of the values.

14. Use `select()` to subset the dataframe `df` with the list of columns `columns` and Sample with the provided `sample()` function, and assign this dataframe to the variable `sample_df`.

- Convert the Subset dataframe to a `pandas` dataframe `pandas_df`, and use `pandas` `isnull()` to convert it `DataFrame` into True/False. Store this result in `tf_df`.
- Use seaborn's `heatmap()` to plot `tf_df`.
- Hit "Run Code" to view the plot. Then assign the name of the variable with most missing values to `answer`.

15. Get a count of the missing values in the column `'PDOM'` using `where()`, `isNull()` and `count()`.

- Calculate the mean value of `'PDOM'` using the aggregate function `mean()`.
- Use `fillna()` with the value set to the `'PDOM'` mean value and only apply it to the column `'PDOM'` using the `subset` parameter.

16. Define a function `column_dropper()` that takes the parameters `df` a dataframe and `threshold` a float between 0 and 1.

- Calculate the percentage of values that are missing using `where()`, `isNull()` and `count()`
- Check to see if the percentage of missing is higher than the threshold, if so, drop the column using `drop()`
- Run `column_dropper()` on `df` with the threshold set to .6

17. Convert `walk_df['latitude']` and `walk_df['longitude']` to type double by using `cast('double')` on the column and replacing the column in place `withColumn()`.

- Round the columns in place with `withColumn()` and `round('latitude', 5)` and `round('longitude', 5)`.
- Create the join condition of `walk_df['latitude']` matching `df['latitude']` and `walk_df['longitude']` matching `df['longitude']`.
- Join `df` and `walk_df` together with `join()`, using the condition above and the `left` join type. Save the joined dataframe as `join_df`.

18. Register the Dataframes as SparkSQL tables with `createOrReplaceTempView`, name them the `df` and `walk_df` respectively.

- In the `join_sql` string, set the left table to `df` and the right table to `walk_df`
- Call `spark.sql()` on the `join_sql` string to perform the join.

19. Create a join between `df_orig`, the dataframe before its precision was corrected, and `walk_df` that matches on `longitude` and `latitude` in the respective dataframes.

- Count the number of missing values with `where()` `isNull()` on `df['walkscore']` and `correct_join['walkscore']`. You should notice that there are many missing values because our datatypes and precision do not match.
- Create a join between `df` and `walk_df` that only matches on `longitude`
- Count the number of records with `count()`: `few_keys_df` and `correct_join_df`. You should notice that there are many more values as we have not constrained our matching correctly.

20. Create a new column using `withColumn()` called `LOT_SIZE_SQFT` and convert `ACRES` to square feet by multiplying by `acres_to_sqfeet` the conversion factor.

- Create another new column called `YARD_SIZE` by subtracting `FOUNDATIONSIZE` from `LOT_SIZE_SQFT`.
- Run `corr()` on each of the independent variables `YARD_SIZE`, `FOUNDATIONSIZE`, `LOT_SIZE_SQFT` against the dependent variable `SALESCLOSEPRICE`. Does new feature show a stronger correlation than either of its components?

21. Create a new variable `ASSESSED_TO_LIST` by dividing `ASSESSEDVALUATION` by `LISTPRICE` to help us understand if the having a high or low assessment value impacts our price.

- Create another new variable `TAX_TO_LIST` to help us understand the approximate tax rate by dividing `TAXES` by `LISTPRICE`.
- Lastly create another variable `BED_TO_BATHS` to help us know how crowded our bathrooms might be by dividing `BEDROOMS` by `BATHSTOTAL`.

22. Create a new feature by adding `SQFTBELOWGROUND` and `SQFTABOVEGROUND` and creating a new column `Total_SQFT`

- Using `Total_SQFT`, create yet another feature called `BATHS_PER_1000SQFT` with `BATHSTOTAL`. Be sure to scale `Total_SQFT` to 1000's
- Use `describe()` to inspect the new min, max and mean of our newest feature `BATHS_PER_1000SQFT`. Notice anything strange?
- Create two `jointplots()`s with `Total_SQFT` and `BATHS_PER_1000SQFT` as the x values and `SALESCLOSEPRICE` as the y value to see which has the better R**2 fit. Does this more complicated feature have a stronger relationship with `SALESCLOSEPRICE`?

23. Import `to_date()` and `dayofweek()` functions from `pyspark.sql.functions`

- Use the `to_date()` function to convert `LISTDATE` to a Spark date type, save the converted column in place using `withColumn()`
- Create a new column using `LISTDATE` and `dayofweek()` then save it as `List_Day_of_Week` using `withColumn()`
- Sample half the dataframe and convert it to a pandas dataframe with `toPandas()` and plot the count of the pandas dataframe's `List_Day_of_Week` column by using seaborn `countplot()` where x = `List_Day_of_Week`.

24. Extract the year from `LISTDATE` using `year()` and put it into a new column called `list_year` with `withColumn()`

- Create another new column called `report_year` by subtracting 1 from the `list_year`
- Create a join condition that matches `df['CITY']` with `price_df['City']` and `df['report_year']` with `price_df['Year']`
- Perform a left join between `df` and `price_df`

25. Cast `mort_df['DATE']` to date type with `to_date()`

- Create a window with the `Window()` function and use `orderBy()` to sort by `mort_df[DATE]`
- Create a new column `DATE-1` using `withColumn()` by lagging the `DATE` column with `lag()` and window it using `over(w)`
- Calculate the difference between `DATE` and `DATE-1` using `datediff()` and name it `Days_Between_Report`

26. Import the needed function `when()` from `pyspark.sql.functions`.

- Create a string matching condition using `like()` to look for for the string pattern `Attached Garage` in `df['GARAGEDESCRIPTION']` and use wildcards `%` so it will match anywhere in the field.
- Similarly, create another condition using `like()` to find the string pattern `Detached Garage` in `df['GARAGEDESCRIPTION']` and use wildcards `%` so it will match anywhere in the field.
- Create a new column `has_attached_garage` using `when()` to assign the value 1 if it has an attached garage, zero if detached and use `otherwise()` to assign null with `None` if it is neither.

27. Import the needed functions `split()` and `explode()` from `pyspark.sql.functions`

- Use `split()` to create a new column `garage_list` by splitting `df['GARAGEDESCRIPTION']` on ', ' which is both a comma and a space.
- Create a new record for each value in the `df['garage_list']` using `explode()` and assign it a new column `ex_garage_list`
- Use `distinct()` to get unique values of `ex_garage_list` and `show` the 100 first rows, truncating them at 50 characters to display the values.

28. Pivot the values of `ex_garage_list` by grouping by the record id `NO` with `groupBy()` use the provided code to aggregate `constant_val` to ignore nulls and take the first value.

- Left join `piv_df` to `df` using `NO` as the join condition.
- Create the list of columns, `zfill_cols`, to zero fill by using the `columns` attribute on `piv_df`
- Zero fill the pivoted dataframes columns, `zfill_cols`, by using `fillna()` with a `subset`.

29. Import the feature transformer `Binarizer` from `pyspark` and the `ml.feature` module.

- Create the transformer using `Binarizer()` with the threshold for setting the value to 1 as anything after Friday, 5.0, then set the input column as `List_Day_of_Week` and output column as `Listed_On_Weekend`.
- Apply the binarizer transformation on `df` using `transform()`.
- Verify the transformation worked correctly by selecting the `List_Day_of_Week` and `Listed_On_Weekend` columns with `show()`.

30. Plot a distribution plot of the `pandas` dataframe `sample_df` using `Seaborn` `distplot()`.

- Given it looks like there is a long tail of infrequent values after 5, create the bucket `splits` of 1, 2, 3, 4, 5+
- Create the transformer `buck` by instantiating `Bucketizer()` with the splits for setting the buckets, then set the input column as `BEDROOMS` and output column as `bedrooms`.
- Apply the Bucketizer transformation on `df` using `transform()` and assign the result to `df_bucket`. Then verify the results with `show()`

31. Create a function `train_test_split_date()` that takes in a dataframe, `df`, the date column to use for splitting `split_col` and the number of days to use for the test set, `test_days` and set it to have a default value of 45.

- Find the `min` and `max` dates for `split_col` using `,()`.
- Find the date to split the test and training sets using `max_date` and subtract `test_days` from it by using `timedelta()` which takes a `days` parameter, in this case, pass in `test_days,
- Using `OFFMKTDATE` as the `split_col` find `split_date` and use it to filter the dataframe into two new ones, `train_df` and `test_df`, Where `test_df` is only the last 45 days of the data. Additionally, ensure that the `test_df` only contains homes listed as of the split date by filtering `df['LISTDATE']` less than or equal to the `split_date`.

32. Import the following functions from `pyspark.sql.functions` to use later on: `datediff()`, `to_date()`, `lit()`.

- Convert the date string '2017-12-10' to a pyspark date by first calling the literal function, `lit()` on it and then `to_date()`
- Create `test_df` by filtering `OFFMKTDATE` greater than or equal to the `split_date` and `LISTDATE` less than or equal to the `split_date` using `where()`.
- Replace `DAYSONMARKET` by calculating a new column called `DAYSONMARKET`, the new column should be the difference between `split_date` and `LISTDATE` use `datediff()` to perform the date calculation. Inspect the new column and the original using the code provided.

33. Using the provided `for` loop that iterates through the list of binary columns, calculate the `sum` of the values in the column using the `agg` function. Use `collect()` to run the calculation immediately and save the results to `obs_count`.

- Compare `obs_count` to `obs_threshold`, the `if` statement should be true if `obs_count` is less than or equal to `obs_threshold`.

- Remove columns that have been appended to `cols_to_remove` list by using `drop()`. Recall that the `*` allows the list to be unpacked.
- Print the starting and ending shape of the PySpark dataframes by using `count()` for number of records and `len()` on `df.columns` or `new_df.columns` to find the number of columns.

34. Replace the values in `WALKSCORE` and `BIKESCORE` with -1 using `fillna()` and the `subset` parameter.

- Create a list of `StringIndexer`s by using list comprehension to iterate over each column in `categorical_cols`.
- Apply `fit()` and `transform()` to the pipeline `indexer_pipeline`.
- Drop the `categorical_cols` using `drop()` since they are no longer needed. Inspect the result data types using `dtypes`.

35. Import `GBTRegressor` from `pyspark.ml.regression` which you will notice is the same module as `RandomForestRegressor`.

- Instantiate `GBTRegressor` with `featuresCol` set to the vector column of our features named, `features`, `labelCol` set to our dependent variable, `SALESCLOSEPRICE` and the random `seed` to `42`
- Train the model by calling `fit()` on `gbt` with the imported training data, `train_df`.

36. Import `RegressionEvaluator` from `pyspark.ml.evaluation` so it is available for use later.

- Initialize `RegressionEvaluator` by setting `labelCol` to our actual data, `SALESCLOSEPRICE` and `predictionCol` to our predicted data, `Prediction_Price`
- To calculate our metrics, call `evaluate` on `evaluator` with the prediction values `preds` and create a dictionary with key `evaluator.metricName` and value of `rmse`, do the same for the `r2` metric.

37. Create a `pandas` dataframe using the values of `importances` and name the column `importance` by setting the parameter `columns`.

- Using the imported list of features names, `feature_cols`, create a new `pandas.Series` by wrapping it in the `pd.Series()` function. Set it to the column `fi_df['feature']`.
- Sort the dataframe using `sort_values()`, setting the `by` parameter to our `importance` column and sort it descending by setting `ascending` to `False`. Inspect the results.

38. Import `RandomForestRegressionModel` from `pyspark.ml.regression`.

- Using the model in memory called `model` call the `save()` method on it and name the model `rfr_no_listprice`.
- Reload the saved model file `rfr_no_listprice` by calling `load()` on `RandomForestRegressionModel` and storing it into `loaded_model`.