Exploratory Data Analysis with PySpark (Spark series part I)

APRIL 18, 2019

Spark is an incredible tool for working with data at scale (i.e. data too large to fit in a single machine's memory). It has an API catered toward data manipulation and analysis, and even has built in functionality for machine learning pipelines and creating ETLs (extract load transform) for a data driven platform or product. For those of us with experience in Python or SQL, API wrappers exist to make a Spark workflow look, feel and act like a typical Python workflow or SQL query. The goal of this post is to present an overview of some exploratory data analysis methods for machine learning and other applications in PySpark and Spark SQL.

This post is the first part in a series of coming blog posts on the use of Spark and in particular PySpark and Spark SQL for data analysis, feature engineering, and machine learning. So stay tuned!

If you'd prefer to learn with a Jupyter Notebook, you can access all of the code on my GitHub page by clicking here.

The dataset used is downloaded from a weblink, so once you get Spark up and running, this guide should be self contained.

When learning a new API, it's always good to reference the docs here are the Spark MLib docs for the version of Spark used in this guide.

First, we'll setup a Spark context and a Spark SQL context.

```
# Setting up spark
import findspark
findspark.init()
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession
from pyspark.sql import *
from pyspark.sql.functions import *
conf = SparkConf().setMaster("local").setAppName("PySpark_feature_eng")
spark = SparkSession.builder.getOrCreate()
print(spark)
```

```
<pyspark.sql.session.SparkSession object at 0x109ddabe0>
```

```
# Set sqlContext from the Spark context
from pyspark.sql import SQLContext
sqlContext = SQLContext(spark)

# Check Spark Version - it changes fast!
# Be very careful when referencing Stack Overflow Q's/A's from older versions of Spa
print(spark.version)
```

```
2.4.0
```

Read in some data

Here I'll be working through EDA techniques on a dataset of residential homes sold in 2017 in the city of St. Paul, MN. We'll download the data using pandas before converting it in to Spark. I do it this way for ease but at the cost of schema - Spark requires more attention to the type of individual columns and how missing values are handled. It isn't quite as versatile as pandas is in inferring data types from the data itself and literally can't handle having more than one data type in a single column. There is a built in method to attempt to infer a schema for the data types when none is provided, which we'll try out after converting all values in the pandas dataframe to strings.

You can download the dataset by clicking or copying this link.

Without local storage, importing a csv file into Spark can be a little tricky. In these cases, you might be working with data from an AWS S3 bucket or pulling in data from an SQL or Parquet database. For our purposes, after reading in and changing some column data types of the csv file with Pandas we'll create a Spark dataframe using the SQL context.

```
import pandas as pd

# Download data into a pandas dataframe
review_df = pd.read_csv('https://assets.datacamp.com/production/repositories/1704/da

# Quickly run through the columns and change to strings to avoid the
# presence of different data types within columns - Spark complains if so
o_type = review_df['MLSID'].dtypes
for column in review_df.columns:
    if review_df[column].dtype == o_type:
        review_df[column] = review_df[column].astype('str')

review_df['streetaddress'] = review_df['streetaddress'].astype('str')
# review_df.dtypes # check your data types if desired

# Setup a Spark SQL context and read in the pandas dataframe to a Spark df
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
df = sqlContext.createDataFrame(review_df)

# Need to convert some data types (since they were all converted to
# strings in the pandas df)
data_types = \
[('NO', 'bigint'),
 ('MLSID', 'string'),
 ('STREETNUMBERNUMERIC', 'bigint'),
 ('STREETADDRESS', 'string'),
 ('STREETNAME', 'string'),
 ('POSTALCODE', 'bigint'),
 ('STATEORPROVINCE', 'string'),
 ('CITY', 'string'),
 ('SALESCLOSEPRICE', 'bigint'),
 ('LISTDATE', 'timestamp'),
 ('LISTPRICE', 'bigint'),
 ('LISTTYPE', 'string'),
 ('ORIGINALLISTPRICE', 'bigint'),
 ('PRICEPERTSFT', 'double'),
 ('FOUNDATIONSIZE', 'bigint'),
 ('FENCE', 'string'),
 ('MAPLETTER', 'string'),
 ('LOTSIZEDIMENSIONS', 'string'),
 ('SCHOOLDISTRICTNUMBER', 'string'),
 ('DAYSONMARKET', 'bigint'),
 ('OFFMARKETDATE', 'timestamp'), # some columns are time-based values
 ('FIREPLACES', 'bigint'),
 ('ROOMAREA4', 'string'),
 ('ROOMTYPE', 'string'),
 ('ROOF', 'string'),
 ('ROOMFLOOR4', 'string'),
 ('POTENTIALSHORTSALE', 'string'),
 ('POOLDESCRIPTION', 'string'),
 ('PDOM', 'double'),
 ('GARAGEDESCRIPTION', 'string'),
 ('SQFTABOVEGROUND', 'bigint'),
 ('TAXES', 'bigint'),
 ('ROOMFLOOR1', 'string'),
 ('ROOMAREA1', 'string'),
 ('TAXWITHASSESSMENTS', 'double'),
 ('TAXYEAR', 'bigint'),
 ('LIVINGAREA', 'bigint'),
 ('UNITNUMBER', 'string'),
 ('YEARBUILT', 'bigint'),
 ('ZONING', 'string'),
 ('STYLE', 'string'),
 ('ACRES', 'double'),
 ('COOLINGDESCRIPTION', 'string'),
 ('APPLIANCES', 'string'),
 ('BACKONMARKETDATE', 'timestamp'),
 ('ROOMFAMILYCHAR', 'string'),
 ('ROOMAREA3', 'string'),
 ('EXTERIOR', 'string'),
 ('ROOMFLOOR3', 'string'),
 ('ROOMFLOOR2', 'string'),
 ('ROOMAREA2', 'string'),
 ('DININGROOMDESCRIPTION', 'string'),
 ('BASEMENT', 'string'),
 ('BATHSFULL', 'bigint'),
 ('BATHSHALF', 'bigint'),
 ('BATHQUARTER', 'bigint'),
 ('BATHSTHREEQUARTER', 'double'),
 ('CLASS', 'string'),
 ('BATHSTOTAL', 'bigint'),
 ('BATHDESC', 'string'),
 ('ROOMAREA5', 'string'),
 ('ROOMFLOOR5', 'string'),
 ('ROOMAREA6', 'string'),
 ('ROOMFLOOR6', 'string'),
 ('ROOMAREA7', 'string'),
 ('ROOMFLOOR7', 'string'),
 ('ROOMAREA8', 'string'),
 ('ROOMFLOOR8', 'string'),
 ('BEDROOMS', 'bigint'),
 ('SQFTBELOWGROUND', 'bigint'),
 ('ASSUMABLEMORTGAGE', 'string'),
 ('ASSOCIATIONFEE', 'bigint'),
 ('ASSESSMENTPENDING', 'string'),
 ('ASSESSEDVALUATION', 'double'),
 ('latitude', 'double'),
 ('longitude', 'double')]

# Correct all the column types
# .withColumn will be used heavily in this guide - it creates a new Spark
# dataframe column, which can overwrite existing columns of the same name
df = df.withColumn("LISTDATE", to_timestamp("LISTDATE", "MM/dd/yyyy"))
df = df.withColumn("OFFMARKETDATE", to_timestamp("OFFMARKETDATE", "MM/dd/yyy"))
df = df.withColumn("AssessedValuation", df["AssessedValuation"].cast("double"))
df = df.withColumn("AssociationFee", df["AssociationFee"].cast("bigint"))
df = df.withColumn("SQFTBELOWGROUND", df["SQFTBELOWGROUND"].cast("bigint"))
df = df.withColumn("Bedrooms", df["Bedrooms"].cast("bigint"))
df = df.withColumn("BATHSTOTAL", df["BATHSTOTAL"].cast("bigint"))
df = df.withColumn("BATHSTHREEQUARTER", df["BATHSTHREEQUARTER"].cast("double"))
df = df.withColumn("BATHQUARTER", df["BATHQUARTER"].cast("bigint"))
df = df.withColumn("BathsHalf", df["BathsHalf"].cast("bigint"))
df = df.withColumn("BathsFull", df["BathsFull"].cast("bigint"))
df = df.withColumn("backonmarketdate", df["backonmarketdate"].cast("double"))
df = df.withColumn("ACRES", df["ACRES"].cast("double"))
df = df.withColumn("YEARBUILT", df["YEARBUILT"].cast("bigint"))
df = df.withColumn("LivingArea", df["LivingArea"].cast("bigint"))
df = df.withColumn("TAXYEAR", df["TAXYEAR"].cast("bigint"))
df = df.withColumn("TAXWITHASSESSMENTS", df["TAXWITHASSESSMENTS"].cast("double"))
df = df.withColumn("Taxes", df["Taxes"].cast("bigint"))
df = df.withColumn("SQFTABOVEGROUND", df["SQFTABOVEGROUND"].cast("bigint"))
df = df.withColumn("PDOM", df["PDOM"].cast("bigint"))
df = df.withColumn("Fireplaces", df["Fireplaces"].cast("bigint"))
df = df.withColumn("FOUNDATIONSIZE", df["FOUNDATIONSIZE"].cast("bigint"))
df = df.withColumn("PricePerTSFT", df["PricePerTSFT"].cast("double"))
df = df.withColumn("OriginalListPrice", df["OriginalListPrice"].cast("bigint"))
df = df.withColumn("LISTPRICE", df["OriginalListPrice"].cast("bigint"))
df = df.withColumn("SalesClosePrice", df["SalesClosePrice"].cast("bigint"))
df = df.withColumn("PostalCode", df["PostalCode"].cast("bigint"))
#df = df.withColumn("No.", df["No."].cast("bigint"))

# Drop the No. column, Spark is very unhappy with the '.' in this column name -
# this will be rectified later on
df = df.drop('No.')
```

Basic Summary Stats

It's always good to begin EDA with a basic understanding of the structure of the data. That means knowing things like how big the dataset is in terms of number of rows and columns, what the distribution of different variables (or features) look like, and how different features interact with each other. This section shows how to find this out using PySpark.

```
# Shape - you can't just use the shape method for a Spark df,
# it doesn't exist as of February 2019. But you can:
print((df.count(), len(df.columns)))
```

```
(5000, 73)
```

```
# Describe a column
df[['SalesClosePrice']].describe().show() # note double bracket col indexing
                                          # the .show() to force Spark to
                                          # run the job
```

```
+-------+------------------+
|summary|   SalesClosePrice|
|   count|             5000|
|    mean|      262804.4668|
|  stddev|140559.82591998548|
|     min|            48000|
|     max|          1700000|
+--------+------------------+
```

```
# Or multiple columns
df.select(['LISTPRICE', 'Taxes', 'SQFTABOVEGROUND', 'FOUNDATIONSIZE',
           'Fireplaces']).describe().show()
```

```
+-------+-----------------+-----------------+-----------------+------------------+-------------
----+
|summary|        LISTPRICE|            Taxes|  SQFTABOVEGROUND|    FOUNDATIONSIZE|       Firepl
aces|
+-------+-----------------+-----------------+-----------------+------------------+-------------
----+
|  count|             5000|             5000|             5000|              5000|
5000|
|   mean|      275002.6304|        3373.6548|         1489.013|         1016.4118|           0.
5848|
| stddev|409620.68253253604|36040.698280462675|626.4868033517984|339.26040521297097|0.708879017810
7813|
|    min|                5|                0|                1|                 1|
0|
|    max|         21990000|          2547070|             7200|              3792|
8|
+-------+-----------------+-----------------+-----------------+------------------+-------------
----+
```

```
# Covariance
# Covariance is a measure of how two variables change with respect to each
# other. A positive number would mean that there is a tendency that as one
# variable increases, the other increases as well. A negative number would
# mean that as one variable increases, the other variable has a tendency to
# decrease. The sample covariance of two columns of a DataFrame can be
# calculated as follows:

print(df.cov('SalesClosePrice', 'YEARBUILT'))
# or
print(df.stat.cov('SalesClosePrice', 'YEARBUILT'))
```

```
1281910.384063509
1281910.384063509
```

```
# Correlation is perhaps more straightforward to interpret
# Correlation is a normalized measure of covariance that is easier to
# understand, as it provides quantitative measurements of the statistical
# dependence between two random variables.

print(df.corr('SalesClosePrice', 'YEARBUILT'))
print(df.stat.corr('SalesClosePrice', 'YEARBUILT'))
```

```
0.23475142032506827
0.23475142032506827
```

```
# Perform an aggregation function
print("Average sales price: ${}".format(df.agg({'SalesClosePrice': 'mean'})\
      .collect()[0][0]))
print("Standard deviation of sales prices: ${}"\
      .format(df.agg({'SalesClosePrice': 'stddev'}).collect()[0][0]))
print("Max sales price: ${}"\
      .format(df.agg({'SalesClosePrice': 'max'}).collect()[0][0]))
print("Min sale price: ${}".format(df.agg({'SalesClosePrice': 'min'})\
      .collect()[0][0]))
```

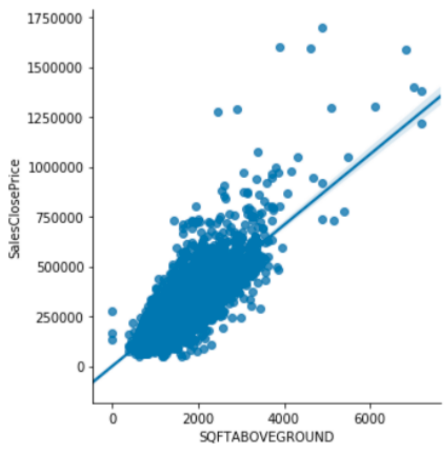
```
Average sales price: $262804.4668
Standard deviation of sales prices: $140559.82591998548
Max sales price: $1700000
Min sale price: $48000
```

Visual inspection through linear models and distribution skew

The distribution of different features, as well as their interactions, are important to understand both in terms of preparing for machine learning applications but also to better understand the structure of a dataset. The saying goes, a picture is worth a thousand words, and that definitely applies to data visualization.

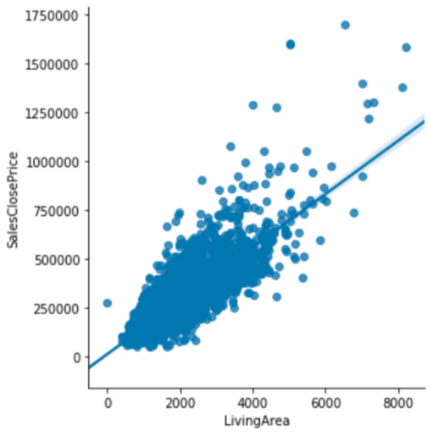
```
# Bring in some plotting libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Sample spark df and plot
# It's a best practice to sample data from your Spark df into pandas so as to
# not asplode your machine's memory. Remember, the idea is to use Spark for
# data that is too large to fit in your machine's memory
s_df = df.select(['SalesClosePrice', 'YEARBUILT'])\
         .sample(withReplacement=False, fraction=0.5, seed=42)

# Create a Pandas df from a subset of columns, those that are sampled
s_df_pandas = s_df.toPandas()

# Or the entire Spark df
df_pandas = df.toPandas()

# A basic seaborn linear model plot
sns.lmplot(y = 'SalesClosePrice', x = 'SQFTABOVEGROUND', data=df_pandas)
```

```
<seaborn.axisgrid.FacetGrid at 0x110f09320>
```

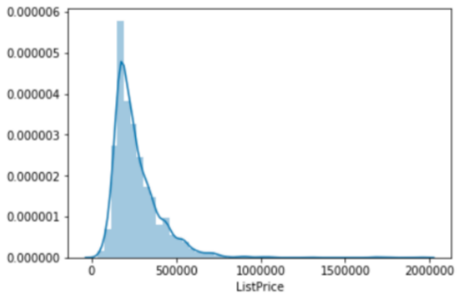


```
sns.lmplot(y = 'SalesClosePrice', x = 'LivingArea', data=df_pandas)
```

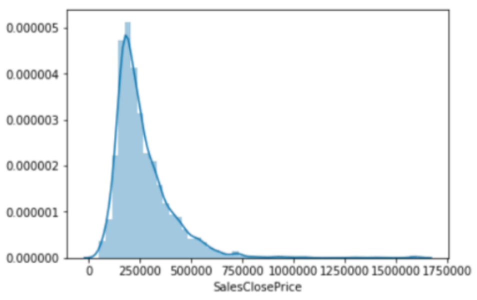
```
<seaborn.axisgrid.FacetGrid at 0x11352bb70>
```



```
# Plot distribution of pandas_df features
# Select a single column and sample and convert to pandas
sample_df = df.select(['LISTPRICE']).sample(False, 0.5, 42)
pandas_df = sample_df.toPandas()

# Plot distribution of pandas_df and display plot
sns.distplot(pandas_df)
plt.xlabel('ListPrice')
plt.show()
```



```
# Plot distribution of pandas_df and display plot
# Select a single column and sample and convert to pandas
sample_df = df.select(['SalesClosePrice']).sample(False, 0.5, 42)
pandas_df = sample_df.toPandas()

# Plot distribution of pandas_df and display plot
sns.distplot(pandas_df)
plt.xlabel('SalesClosePrice')
plt.show()
```



```
print(df.corr('SalesClosePrice', 'LivingArea'))
print(df.corr('SalesClosePrice', 'SQFTABOVEGROUND'))
```

```
0.8162701823275657
0.7901878498502953
```

Filtering based on values or text

Often times, it's necessary to reduce the scope of an analysis. Filtering out outliers (e.g. observations that are more than two or three deviations from the mean) of a feature can go a long way to building a model that is appropriately generalizable to unseen data. You might also need to filter out specific categories for features that are categorical in nature.

```
# Filter by values - expensive homes
df.where(df['SalesClosePrice'] > 1000000)[['SalesClosePrice']].show(10)
```

```
+---------------+
|SalesClosePrice|
+---------------+
|        1277023|
|        1050000|
|        1290000|
|        1295000|
|        1215000|
|        1380000|
|        1300000|
|        1400000|
|        1595000|
|        1600000|
+---------------+
only showing top 10 rows
```

```
# Filter by values - cheap homes
df.where(df['SalesClosePrice'] < 100000)[['SalesClosePrice']].show(10)
```

```
+---------------+
|SalesClosePrice|
+---------------+
|          90000|
|          94900|
|          79900|
|          78000|
|          99000|
|          80000|
|          48000|
|          49900|
|          80004|
|          50800|
+---------------+
only showing top 10 rows
```

```
# Porque no los dos? (Why not both?)
# Note that I'm using .filter() here - there is no difference b/t .where()
# and .filter() - .where() is provided for those who are used to the SQL
# implementation of WHERE, while .filter() is provided for the Scala familiar
df.filter((df['SalesClosePrice'] < 100000) |
          (df['SalesClosePrice'] > 1000000))[['SalesClosePrice']].describe()\
          .show()
```

```
+-------+-----------------+
|summary|  SalesClosePrice|
+-------+-----------------+
|  count|              128|
|   mean|   220585.4921875|
| stddev|401209.0720967758|
|    min|            48000|
|    max|          1700000|
+-------+-----------------+
```

```
# What if you really don't like metal roofs and have a huge budget?
# There are some missing values here, we'll show how to deal with those below
df.where((df['ROOF'] != 'Metal') &
          (df['SalesClosePrice'] > 1000000))[['SalesClosePrice', 'ROOF']]\
         .show(truncate=50)
```

```
+---------------+--------------------------------------------+
|SalesClosePrice|                                        ROOF|
+---------------+--------------------------------------------+
|        1277023|                                         nan|
|        1050000|                                         nan|
|        1290000|                            Asphalt Shingles|
|        1295000|Asphalt Shingles, Pitched, Age Over 8 Years|
|        1215000|                            Asphalt Shingles|
|        1380000|                                         nan|
|        1300000|                            Asphalt Shingles|
|        1400000|                  Age Over 8 Years, Tile, Metal|
|        1595000|        Asphalt Shingles, Age 8 Years or Less|
|        1600000|                                      Shakes|
|        1700000|                                         nan|
|        1585000|                                       Slate|
|        1050216|        Asphalt Shingles, Age 8 Years or Less|
|        1077200|                                         nan|
+---------------+--------------------------------------------+
```

```
# More value filtering, this time with aggregation functions
std_val = df.agg({'SalesClosePrice': 'stddev'}).collect()[0][0]
mean_val = df.agg({'SalesClosePrice': 'mean'}).collect()[0][0]

# Values for dropping outliers that are 3 standard deviations from the mean
hi_bound = mean_val + (3 * std_val)
low_bound = mean_val - (3 * std_val)

df.where((df['LISTPRICE'] < hi_bound) & (df['LISTPRICE'] > low_bound))\
         .count() # Here we dropped 91 observations
```

```
4909
```

```
# Filter based on text

# Inspect unique values in the column 'ASSUMABLEMORTGAGE'
print(df.select(['ASSUMABLEMORTGAGE']).distinct().show())

# List of possible values containing 'yes'
yes_values = ['Yes w/ Qualifying', 'Yes w/No Qualifying']

# Filter the text values out of df but keep null values
# (good use of that ~ here)
text_filter = ~df['ASSUMABLEMORTGAGE'].isin(yes_values) | \
               df['ASSUMABLEMORTGAGE'].isNull()

df.where(text_filter).count()
```

```
+-------------------+
|  ASSUMABLEMORTGAGE|
+-------------------+
|  Yes w/ Qualifying|
| Information Coming|
|                nan|
|Yes w/No Qualifying|
|      Not Assumable|
+-------------------+

None
```

```
4976
```

```
# Text filtering - where the cooling description is NOT (via the ~)
# central air
df.where(~df['CoolingDescription'].like('Central'))[['CoolingDescription']]\
  .show(5)
```

```
+------------------+
|CoolingDescription|
+------------------+
|            Window|
|              None|
|              None|
|        Geothermal |
|              None|
+------------------+
only showing top 5 rows
```

Dropping NA values or dropping columns outright

Missing values are a fact of life in data analytics and data science. Data collection schema often fall short and as a result, it is quite often that certain values will not be present in a dataset. It's important to assess is these observations are missing at random or missing not at random. Domain expertise and discussions with stakeholders go a long way in helping understand the nature of missing values.

See the Missing Values chunks below for a more sophisticated approach to dropping NA or NaN values.

```
# Remove rows with any NA values - naive approach
df.dropna().count() #one column is all missing and that drops the whole df
```

```
0
```

```
# Remove a record if it has NA values in three columns
df.dropna(thresh=3).count() # we don't have any missing values aside from one
                            # column, which is nice
```

```
5000
```

```
# Make a list of columns to drop
cols_to_drop = ['No.', 'UNITNUMBER', 'CLASS']

# Drop the columns
df = df.drop(*cols_to_drop) # the star (*) tells the function to unpack the
                            # list and drop them one-by-one
```

Transforming or Adjusting Data

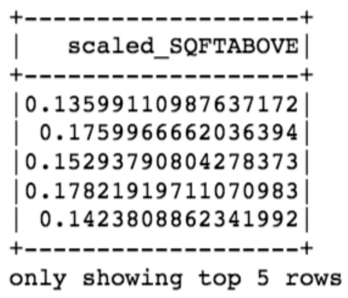
Data transformation is often an essential step in preparing a dataset for modeling. For example, most deep learning models and other statistical models in the Spark-ML library perform significantly better on datasets where individual features have been range normalized between 0 and 1. This often helps reduce computation time as well. In general, it's a good strategy to understand the scale that each feature is on and if there are potentially scalar differences between features. Transforming or adjusting data are key steps to continue gaining intuition around a dataset as well as in preparing the dataset for training a machine learning model.

MinMax Scaling, Standardizing (z-score transformation), Log Scaling

Data does not give up its secrets easily, it must be tortured to confess

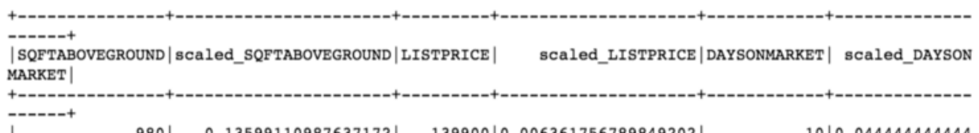
MinMax Scaling

```
# define the minimum and maximum values for the feature of interest
# collect forces the agg func to run
min_val = df.agg({'SQFTABOVEGROUND':'min'}).collect()[0][0]

# [0][0] is necessary to return the value
max_val = df.agg({'SQFTABOVEGROUND':'max'}).collect()[0][0]

# Now use the values to create a new column from the scaled data
# the withColumn() method creates a new column based on a transformation
# of another column
df.withColumn('scaled_SQFTABOVE',
              (df['SQFTABOVEGROUND'] - min_val)/(max_val - min_val))\
              [['scaled_SQFTABOVE']].show(5)
```



```
+-------------------+
|   scaled_SQFTABOVE|
+-------------------+
|0.13599110987637172|
| 0.1759966662036394|
|0.15293790804278373|
|0.17821919711070983|
| 0.1423808862341992|
+-------------------+
only showing top 5 rows
```

```
# How about we make a custom function to scale columns of our choice
def min_max_scaler(df, cols_to_scale):
  # Takes a dataframe and list of columns to minmax scale. Returns a dataframe.
  for col in cols_to_scale:
    # Define min and max values and collect them
    max_days = df.agg({col: 'max'}).collect()[0][0]
    min_days = df.agg({col: 'min'}).collect()[0][0]
    new_column_name = 'scaled_' + col
    # Create a new column based off the scaled data
    df = df.withColumn(new_column_name,
                      (df[col] - min_days) / (max_days - min_days))
  return df

cols_to_scale = ['SQFTABOVEGROUND', 'LISTPRICE', 'DAYSONMARKET']

min_max_scaler(df, cols_to_scale)[['SQFTABOVEGROUND','scaled_SQFTABOVEGROUND',
                                   'LISTPRICE', 'scaled_LISTPRICE',
                                   'DAYSONMARKET', 'scaled_DAYSONMARKET']]\
                                     .show(5)
```



```
+---------------+----------------------+---------+--------------------+------------+-------------
------+
|SQFTABOVEGROUND|scaled_SQFTABOVEGROUND|LISTPRICE|    scaled_LISTPRICE|DAYSONMARKET| scaled_DAYSON
MARKET|
+---------------+----------------------+---------+--------------------+------------+-------------
------+
|            980|   0.13599110987637172|   139900|0.006361756789849202|          10|0.04444444444
```

```
444446|
|          1268|   0.1759966662036394|   210000|0.009549570156791759|          4|0.017777777777
777778|
|          1102|  0.15293790804278373|   225000| 0.01023169855200058|         28| 0.12444444444
444444|
|          1284|  0.17821919711070983|   230000|0.010459074683736854|         19| 0.08444444444
444445|
|          1026|   0.1423808862341992|   239900|0.010909279424574677|         21| 0.09333333333
333334|
+--------------+---------------------+---------+--------------------+-----------+-------------
------+
only showing top 5 rows
```

Standardizing

```
# Similar process to MinMax, but the focus here is on the distribution of the
# data as opposed to its range of values define the mean and std values for
# the feature of interest
mean_val = df.agg({'SQFTABOVEGROUND':'mean'}).collect()[0][0]
std_val = df.agg({'SQFTABOVEGROUND':'stddev'}).collect()[0][0]

# Create new column with standardized data
df_stand = df.withColumn('standardized_SQFTABOVE',
                         (df['SQFTABOVEGROUND'] - mean_val)/std_val)\
                         [['standardized_SQFTABOVE']]

# Check the mean to be close to 0
print(df_stand.agg({'standardized_SQFTABOVE': 'mean'}).collect()[0][0])

# And the stddev to be close to 1
# I left out the [0][0] here to show what you'd see otherwise
print(df_stand.agg({'standardized_SQFTABOVE': 'stddev'}).collect())
```

```
9.094947017729283e-17
[Row(stddev(standardized_SQFTABOVE)=0.9999999999999997)]
```

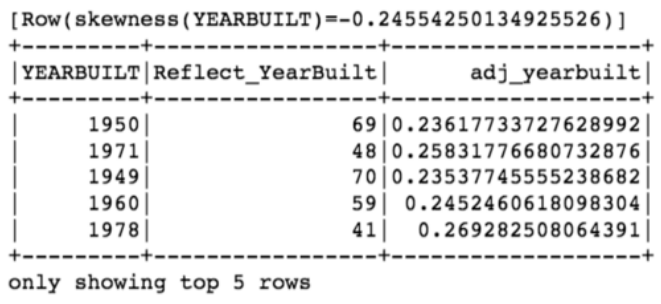
Log Transformation

Log transformations are often a critical step in working with data on different scales or when working with time series data to remove exponential variance. This is necessary when it looks like the variance in time series data is increasing with the level of series and can aid in removing seasonality effects.

```
from pyspark.sql.functions import log

# First check skewness
print(df.agg({'LISTPRICE': 'skewness'}).collect()[0][0])

df.withColumn('logged_ListPrice', log(df['LISTPRICE']))[['logged_ListPrice']]\
  .show(4)
```

```
41.75409786139753
+------------------+
|  logged_ListPrice|
+------------------+
|11.848683160653573|
|12.254862809699606|
|12.323855681186558|
|12.345834587905333|
+------------------+
only showing top 4 rows
```

```
# Another approach to wrangle a skewed feature into something that's more
# normally distributed

# Compute the skewness
print(df.agg({'YEARBUILT': 'skewness'}).collect())

# Calculate the max year
max_year = df.agg({'YEARBUILT': 'max'}).collect()[0][0]

# Create a new column of reflected data
df_reflect = df.withColumn('Reflect_YearBuilt',
                           (max_year + 1) - df['YEARBUILT'])

# Create a new column based reflected data
df_reflect.withColumn('adj_yearbuilt',
                      1 / log(df_reflect['Reflect_YearBuilt']))\
                      [['YEARBUILT', 'Reflect_YearBuilt',
                        'adj_yearbuilt']].show(5)
```



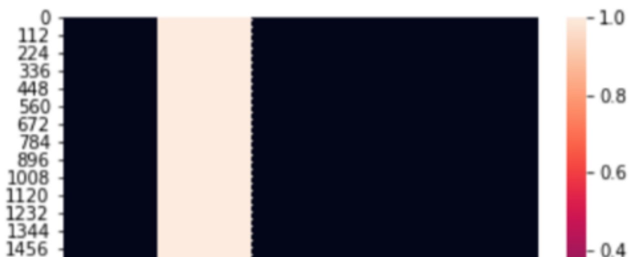
```
[Row(skewness(YEARBUILT)=-0.24554250134925526)]
+---------+-----------------+-------------------+
|YEARBUILT|Reflect_YearBuilt|      adj_yearbuilt|
+---------+-----------------+-------------------+
|     1950|               69|0.23617733727628992|
|     1971|               48|0.25831776680732876|
|     1949|               70|0.23537745555238682|
|     1960|               59| 0.2452460618098304|
|     1978|               41|  0.269282508064391|
+---------+-----------------+-------------------+
only showing top 5 rows
```

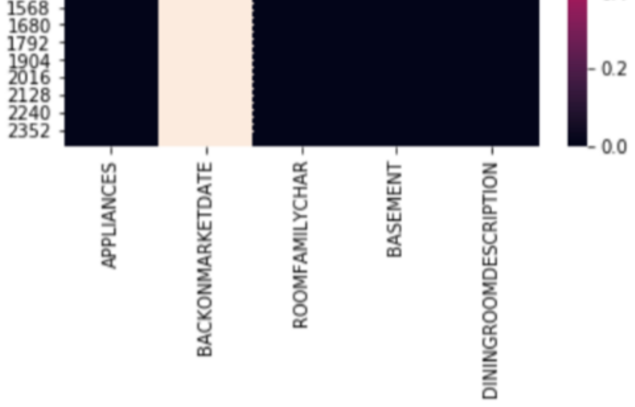
Missing Values

Assessing 'missingness'

A heatmap from seaborn is a good way to determine the extent of missing data in a dataset.

```
columns = ['APPLIANCES','BACKONMARKETDATE','ROOMFAMILYCHAR', 'BASEMENT',
           'DININGROOMDESCRIPTION']
# Sample the dataframe and convert to Pandas
sample_df = df.select(columns).sample(False, 0.5, 42)
pandas_df = sample_df.toPandas()

# Convert all values to T/F
tf_df = pandas_df.isnull()

# Plot it
sns.heatmap(data=tf_df)
plt.xticks(rotation=90, fontsize=10)
plt.yticks(rotation=0, fontsize=10)
plt.show()
```

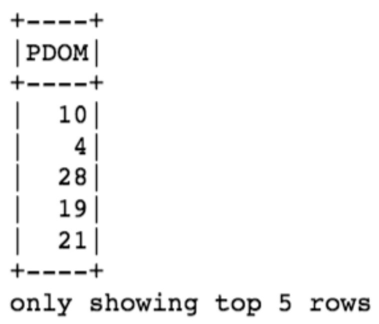




Imputation

Imputation is the process of replacing missing values with another value or values. There are tons of options for imputation from the simple to the complex. Below, I show how to replace missing values with the mean of the feature. Can you think of an application where imputing the mean is likely an optimal solution? How about an application where imputing with the mean of a feature is probably not the best idea?

```
# Count missing rows
missing = df.where(df['PDOM'].isNull()).count()

# Calculate the mean value
col_mean = df.agg({'PDOM': 'mean'}).collect()[0][0]

# Replacing with the mean value for that column
df.fillna(col_mean, subset=['PDOM'])[['PDOM']].show(5)
```



```
+----+
|PDOM|
+----+
|  10|
|   4|
|  28|
|  19|
|  21|
+----+
only showing top 5 rows
```

Dropping columns by a threshold of percent missing (null) or percent NaN

If too many observations are missing in a particular feature, it may be necessary to drop it entirely. Here, I define a function to drop a column, or feature, outright if it does not conform to a threshold for observations present.

```
# Define a function to drop columns if they meet a threshold of missingness or
# NaNness
# Note: this won't work on timestamp or date type columns

# Drop the timestamp columns
df_no_dates = df.select([c for c in df.columns if c not in {'LISTDATE',
                         'OFFMARKETDATE'}])

# Could also do:
#df.drop('LISTDATE', 'OFFMARKETDATE')

def column_dropper(df, threshold = 0.60):
  # Takes a dataframe and threshold for missing values. Returns a dataframe.
  total_records = df.count()
  for col in df.columns:
    # Calculate the percentage of missing values
    missing_null = df.where(df[col].isNull()).count()
    missing_nan = df.where(isnan(df[col])).count()
    missing_percent_null = missing_null / total_records
    missing_percent_nan = missing_nan / total_records
    # Drop column if percent of missing is more than threshold
    if (missing_percent_null > threshold) | (missing_percent_nan > threshold):
      df = df.drop(col)
  return df

# Drop columns that are more than 60% missing
len(column_dropper(df_no_dates, 0.60).columns)
```

```
68
```

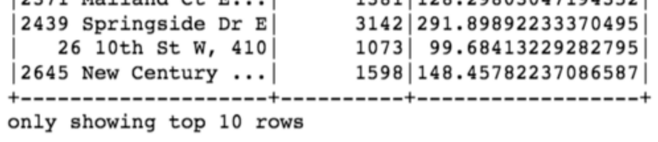
Joining data in Spark with PySpark or with SparkSQL

A critical component of any analyst or data scientist's toolkit is knowing how to join data together. Why? Well because databases in most businesses are often quite complex, quite messy, and quite numerous. It is therefore necessary to know how to grab features of importance from one table in a database and bring in relevant features from other tables.

Spark offers multiple distinct APIs to handle data joins. The PySpark implementation provides a pythonic means of joining data and is reminiscent of pandas. While SparkSQL allows the analyst or data scientist to use SQL queries.

```
# Joining with PySpark

# Convert sqft to sqm (square meters) and select the street address and sqm
# living area size for a new df
LivingArea_sqm = df.withColumn('LivingArea_sqm', df['LivingArea'] / 10.764)\
                 [['streetaddress', 'LivingArea_sqm']]

# Create join condition - here we are joining on the same column
# ('streetaddress')
# But in instances where the join is on columns of different names,
# you need to create a join condition to join on, such as:
### condition = [df['SalesClosePrice'] == LivingArea_sqm['SalesClosePrice']]

# Join the dataframes together
join_df = df.join(LivingArea_sqm, on=['streetaddress'], how='left')

# Count non-null records from new field
join_df.select(['streetaddress', 'LivingArea', 'LivingArea_sqm',
                'SalesClosePrice']).show(10)

# Certainly, the size of the living area in a house is likely going to be a
# major predictor of its sale price
```

```
+--------------------+----------+------------------+
|       streetaddress|LivingArea|    LivingArea_sqm|
+--------------------+----------+------------------+
|      1107 Jenks Ave|      1088|101.07766629505761|
|1181 Edgcumbe Rd,...|       720| 66.88963210702342|
|      1485 Blair Ave|      1932| 179.4871794871795|
|       1679 Lark Ave|      1610| 149.5726495726496|
|  2014 Worcester Ave|      1638|152.17391304347828|
|     2338 Bourne Ave|      2352|218.50613154960982|
|2371 Mailand Ct E...|      1381|128.29803047194352|
```



```
|2439 Springside Dr E|      3142|291.89892233370495|
|   26 10th St W, 410|      1073| 99.68413229282795|
|2645 New Century ...|      1598|148.45782237086587|
+--------------------+----------+------------------+
only showing top 10 rows
```

Joining data with SparkSQL

A word of caution: it's important to be VERY careful so as not to duplicate columns when using a SQL join. Spark doesn't work as intuitively as one might think in this area. Notice the aliasing in the SELECT statement below - if a * was used, the joined_df table will end up with two 'streetaddress' columns and Spark isn't able to distinguish between them because they have the same name (even though they don't really, but that's a different story). This behavior was fixed in the above join via the use of [] or "" around the column name being joined on, but it persists in the SQL join methodology below unless you specifically call for only one of the columns being joined on in the SELECT statement.

```
# Joining data with SparkSQL

# Register dataframes as tables
df.createOrReplaceTempView('df')
LivingArea_sqm.createOrReplaceTempView('LivingArea_sqm')

# SQL to join dataframes
join_sql =  """
            SELECT df.streetaddress, df.LivingArea, LivingArea_sqm
            FROM df
            LEFT JOIN LivingArea_sqm
            ON df.streetaddress = LivingArea_sqm.streetaddress
            """

# Perform sql join
joined_df = spark.sql(join_sql)

joined_df.show(10)
```

I hope you enjoyed this initial dive into the PySpark and Spark SQL APIs. By now, you have the skills necessary to jump in with some exploratory data analysis using the power of Spark.

Future posts in the Spark series will cover feature engineering for machine learning applications, training and validating machine learning models, and how to setup and run Spark on an AWS cluster, so please stay tuned!

Note: This guide was adapted from and expanded upon the course "Feature Engineering with Pyspark" by John Hogue, Lead Data Scientist at General Mills. Huge thanks to John for putting the course together.

7 LIKES   SHARE

Newer   Older