Feature Engineering with PySpark (Spark Series Part 2)

In this post, we'll cover the following topics related to feature engineering:

Feature engineering with basic math operations combining two or more features

Feature engineering for Time Components

Feature engineering for Text Data

There are two great packages that help to implement feature engineering and are worth looking into - tsfresh and Featuretools.

Basic Feature Engineering - Combining two or more Columns Using Simple Math Operations

The square footage of a home both above ground and below ground that you can sum into a new, total square footage feature (total_sqft).
Then,another new feature, price_per_sqft by taking total_sqft/list_price.
Combining price_per_sqft with another existing or newly created feature can often make things difficult to interpret, so it's usually best to stop at combinations of three columns.

In [1]:

```
#接續先前工作
```

In [2]:

```
import os
os.environ["ARROW_PRE_0_15_IPC_FORMAT"] = "1"

import numpy as np
import pandas as pd

#eda
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import udf,col, to_timestamp
from pyspark.sql.types import *
import pyspark.sql.functions as F

from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.regression import LinearRegression
import seaborn as sns
```

In [3]:

```
import matplotlib.pyplot as plt
%matplotlib inline

#記得提高圖的解析度
%config InlineBackend.figure_format = 'retina'
```

In [4]:

```
#read data
sparkcontext = SparkSession.builder.master("spark://master:7077").appName("san_p
aul_housing_price").getOrCreate()
HOUSING_DATA = '/user/curtis0982/data/san_paul_housing/san_paul_housing/2017_StP
aul_MN_Real_Estate.csv'
```

In [5]:

```
review_df = spark.read.csv(path =HOUSING_DATA  ,header=True).toPandas()
# Quickly run through the columns and change to strings to avoid the
# presence of different data types within columns - Spark complains if so
for column in review_df.columns:
    if review_df[column].dtype == object:
        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
```

In [6]:

```
# 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)
```

In [7]:

```
# 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')]
```

In [8]:

```
# 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"))
#Three-Quarter Bathroom 指沒有浴缸的浴室
df = df.withColumn("BATHSTHREEQUARTER", df["BATHSTHREEQUARTER"].cast("double"))
df = df.withColumn("BATHQUARTER", df["BATHQUARTER"].cast("bigint"))
#BathsHalf指只有馬桶 洗手台
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"))
#稅上房屋價值
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.')
```

In [ ]:

```

```

In [9]:

```
# Lot size in square feet
#acres英亩
acres_to_sqfeet = 43560
df = df.withColumn('LOT_SIZE_SQFT', df['ACRES'] * acres_to_sqfeet)

# Create new column YARD_SIZE
#yard+lot=foundation
df = df.withColumn('YARD_SIZE', df['LOT_SIZE_SQFT'] - df['FOUNDATIONSIZE'])

# Corr of ACRES vs SALESCLOSEPRICE
print("Corr of ACRES vs SalesClosePrice: " +
      str(df.corr('ACRES', 'SalesClosePrice')))
# Corr of FOUNDATIONSIZE vs SALESCLOSEPRICE
print("Corr of FOUNDATIONSIZE vs SalesClosePrice: " +
      str(df.corr('FOUNDATIONSIZE', 'SalesClosePrice')))
# Corr of YARD_SIZE vs SALESCLOSEPRICE
print("Corr of YARD_SIZE vs SalesClosePrice: " +
      str(df.corr('YARD_SIZE', 'SalesClosePrice')))
```

```
Corr of ACRES vs SalesClosePrice: 0.22060612588935358
Corr of FOUNDATIONSIZE vs SalesClosePrice: 0.6152231695664397
Corr of YARD_SIZE vs SalesClosePrice: 0.20714585430854235
```

Another basic method of feature engineering is to create ratios based on two or more existing features.

These are commonly used in tracking KPIs (or key performance indicators) because ratios can capture many different business models.
For example, the ratio of products purchased to products viewed or number of customers who converted (or purchased a product) per number of unique customer visits to a website on a given day.

Other basic summary metrics that can be engineered include: sums/counts, distributions (mean, median, percentiles), and probabilities or rates.

Below, we'll calculate the ratio of assessed valuation to list price, the amount of taxes to list price, and beds to baths for houses in our dataset.

In [10]:

```
# Create feature Ratios

# ASSESSED_TO_LIST
df = df.withColumn('ASSESSED_TO_LIST', df['AssessedValuation']/df['LISTPRICE'])
df[['AssessedValuation', 'LISTPRICE', 'ASSESSED_TO_LIST']]\
                .sort(col('ASSESSED_TO_LIST').desc()).show(5)

# TAX_TO_LIST
df = df.withColumn('TAX_TO_LIST', df['TAXES']/df['LISTPRICE'])
df[['TAX_TO_LIST', 'TAXES', 'LISTPRICE']].sort(col('TAXES').desc()).show(5)

# BED_TO_BATHS
df = df.withColumn('BED_TO_BATHS', df['BEDROOMS']/df['BATHSTOTAL'])
df[['BED_TO_BATHS', 'BEDROOMS', 'BATHSTOTAL']].sort(col('BED_TO_BATHS').desc()).
show(5)
```

```
+-----------------+---------+-------------------+
|AssessedValuation|LISTPRICE|   ASSESSED_TO_LIST|
+-----------------+---------+-------------------+
|            255.0|        5|               51.0|
|           2861.0|    40000|           0.071525|
|           8588.0|   164900| 0.05208004851425106|
|           9090.0|   189900|0.047867298578199054|
|           6600.0|   150000|              0.044|
+-----------------+---------+-------------------+
only showing top 5 rows

+--------------------+-------+---------+
|         TAX_TO_LIST|  TAXES|LISTPRICE|
+--------------------+-------+---------+
|   8.786029665401863|2547070|   289900|
| 0.02541769451296753|  38124|  1499900|
|0.022931159420289855|  31645|  1380000|
|0.018127058823529413|  30816|  1700000|
|0.019410033444816052|  29018|  1495000|
+--------------------+-------+---------+
only showing top 5 rows

+------------+--------+----------+
|BED_TO_BATHS|BEDROOMS|BATHSTOTAL|
+------------+--------+----------+
|         5.0|       5|         1|
|         5.0|       5|         1|
|         4.0|       4|         1|
|         4.0|       4|         1|
|         4.0|       4|         1|
+------------+--------+----------+
only showing top 5 rows
```

Do you notice anything interesting about the ratios above? First, we can see some pretty major outliers in the first two ratios. A list price of only $5 dramatically skews the ratio for that observation. In the Tax-to-List ratio, we see an outlier at the opposite end of the extreme - a super high tax bill is very much an anomaly there relative to the list price of that home. These should be caught ahead of time in EDA, but a good metric or KPI will always show these kinds of issues right at the forefront and alert the data analyst or scientist that there might be some issues that need addressed.

Let's create a few more basic features and explore how they relate to our running goal of predicting a home's sale price.

In [11]:

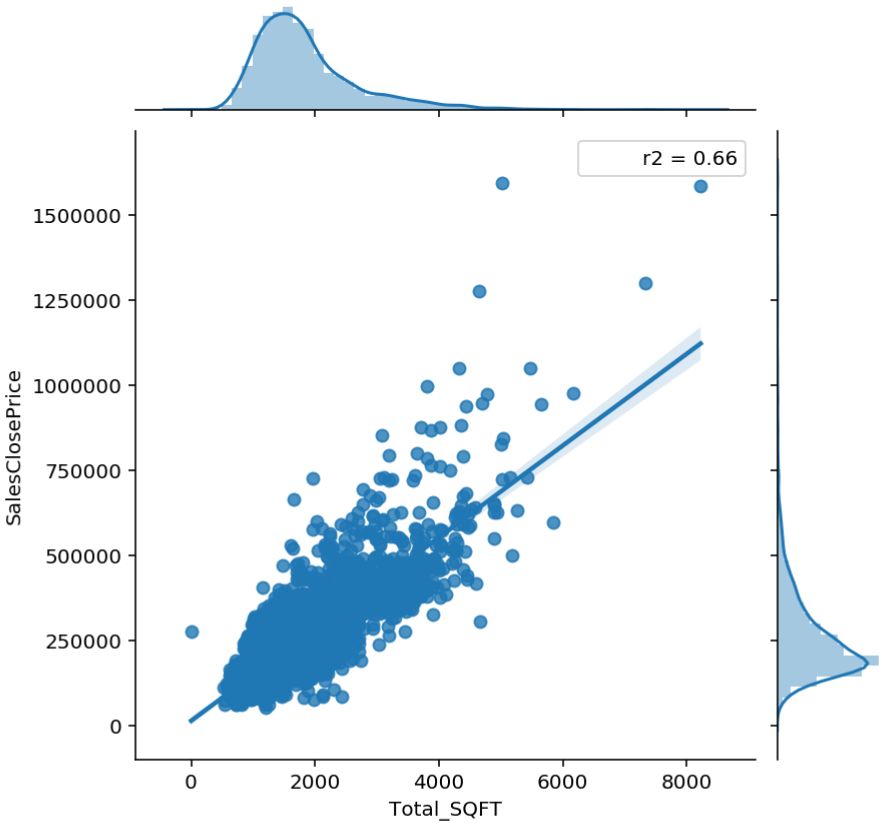
```
# Create new feature by adding two features together
df = df.withColumn('Total_SQFT', df['SQFTBELOWGROUND'] +
                   df['SQFTABOVEGROUND'])

# Create additional new feature using previously created feature
df = df.withColumn('BATHS_PER_1000SQFT', df['BATHSTOTAL'] /
                                        (df['Total_SQFT'] / 1000))
df[['Total_SQFT','BATHSTOTAL','BATHS_PER_1000SQFT']].describe().show()

# Check on outlier - a house that is 1 sqft would be a little tight,
# but at least it has a bathroom
df[['Total_SQFT','BATHSTOTAL','BATHS_PER_1000SQFT']]\
  .sort(col('BATHS_PER_1000SQFT').desc()).show(5)
```

```
+-------+----------------+-----------------+------------------+
|summary|      Total_SQFT|       BATHSTOTAL|BATHS_PER_1000SQFT|
+-------+----------------+-----------------+------------------+
|  count|            5000|             5000|              5000|
|   mean|        1844.528|           2.2008|1.4302617483739883|
| stddev|840.4421899398714|0.9574250323333459| 14.128904102459373|
|    min|               1|                1|0.39123630672926446|
|    max|            8222|                7|            1000.0|
+-------+----------------+-----------------+------------------+

+----------+----------+------------------+
|Total_SQFT|BATHSTOTAL|BATHS_PER_1000SQFT|
+----------+----------+------------------+
|         1|         1|            1000.0|
|       934|         4| 4.282655246252676|
|       600|         2|3.3333333333333335|
|       649|         2| 3.081664098613251|
|      1047|         3| 2.865329512893983|
+----------+----------+------------------+
only showing top 5 rows
```

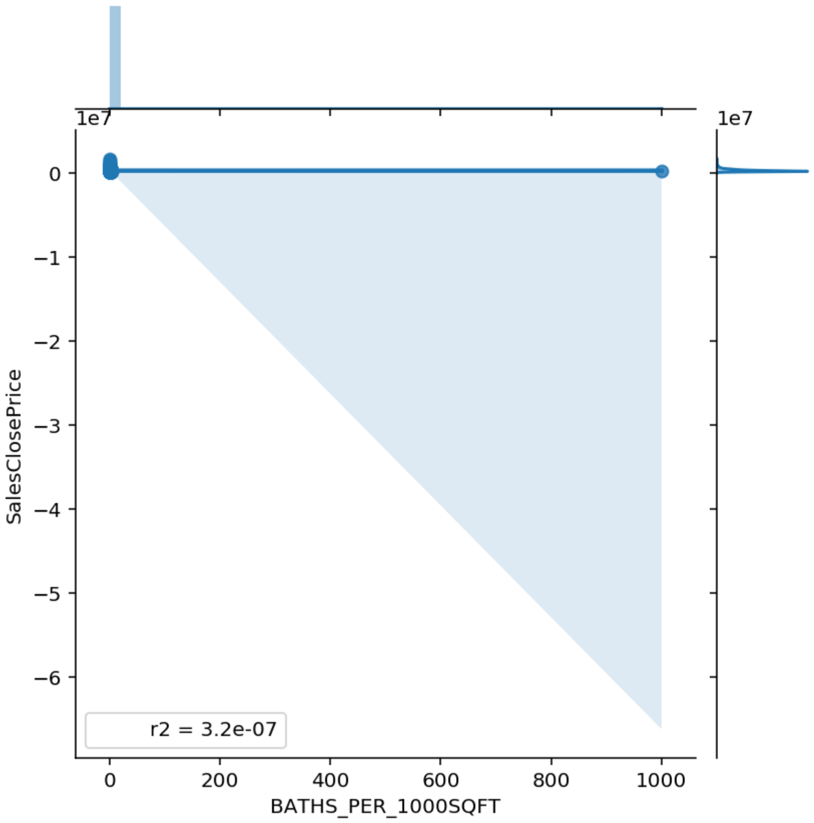
In [12]:

```
# Sample and create pandas dataframe
pandas_df = df.sample(False, 0.5, 0).toPandas()

import scipy.stats as stats
# Define a function to compute r^2
def r2(x, y):
    return stats.pearsonr(x, y)[0] ** 2

# Linear model plots
sns.jointplot(x='Total_SQFT', y='SalesClosePrice', data=pandas_df, kind="reg",
              stat_func=r2)
plt.show()
sns.jointplot(x='BATHS_PER_1000SQFT', y='SalesClosePrice', data=pandas_df,
              kind="reg",stat_func=r2)
plt.show() # Clearly have an outlier here
```

```
/home/curtis0982/anaconda3/lib/python3.7/site-packages/pyarrow/panda
s_compat.py:752: FutureWarning: .labels was deprecated in version 0.
24.0. Use .codes instead.
  labels, = index.labels
/home/curtis0982/anaconda3/lib/python3.7/site-packages/seaborn/axisg
rid.py:1848: UserWarning: JointGrid annotation is deprecated and wil
l be removed in a future release.
  warnings.warn(UserWarning(msg))
```




```
/home/curtis0982/anaconda3/lib/python3.7/site-packages/seaborn/axisg
rid.py:1848: UserWarning: JointGrid annotation is deprecated and wil
l be removed in a future release.
  warnings.warn(UserWarning(msg))
```



The second plot above shows a clear outlier, which can be viewed in the table output above. It shows a house listed with just 1 sq. ft! That's a pretty tiny house, at least it has a bathroom. But in all seriousness, this is another example of where EDA should be used to filter out outliers ahead of time. The point of leaving the outlier in is to show how dramatically results can be skewed if outliers aren't addressed ahead of time.

Feature Engineering for Time Components

For our running example of predicting home prices, seasonality can make a big difference in the final sale price of a home.
There is often less inventory and less demand in the winter months, often leading to reduced prices.
Alternatively, the spring and early summer often sees a big jump ahead of moves scheduled around the end of the school year and summer break.
As a result, having a strong sense of the timing and seasonality of your data is an essential step in feature engineering ahead of training and validating models.

For the examples below, PySpark's week starts on Sunday, with a value of 1 and ends on Saturday, a value of 7.

In [ ]:

```
# Import needed functions
from pyspark.sql.functions import to_date, dayofweek

# Convert to date type
df = df.withColumn('LISTDATE', to_date('LISTDATE'))
# note that this was in timestamp format,
# to_date doesn't work on strings

# Get the day of the week
df = df.withColumn('List_Day_of_Week', dayofweek('LISTDATE'))

# Sample and convert to pandas dataframe
sample_df = df.sample(False, .5, 42).toPandas()

# Plot count plot of of day of week - most houses are listed on Thursdays
# and Fridays in this dataset it seems
sns.countplot(x="List_Day_of_Week", data=sample_df)
plt.show()
```

Individual aspects of a date-time can be very relevant features for training machine learning models. But these models can’t really understand the nuance of a full date. As a result, it’s helpful to pull out the individual aspects of a date, which the model can generalize around. These include such aspects as day of week (as above), day of year, week of year, month of year, and so on.

In [ ]:

```
# Let's pull individual components of the timestamp feature we have as
# separate features

# Import needed functions
from pyspark.sql.functions import year, month, dayofmonth, weekofyear

print(df.withColumn('Year', year('LISTDATE'))[['Year']].show(5))
print(df.withColumn('Month', month('LISTDATE'))[['Month']].show(5))
print(df.withColumn('DayofMonth', dayofmonth('LISTDATE'))[['DayofMonth']]\
        .show(5))
print(df.withColumn('WeekofYear', weekofyear('LISTDATE'))[['WeekofYear']]\
        .show(5))
```

Date Differences

Another important concept in seasonality is date differences and lag. Often, a more nuanced feature such as the difference in dates between two date features can be a more sensitive indicator of a trend that a machine learning model can capture. For our running example, the amount of time a house has been on the market will very much be a sensitive feature in its final sale price. Thus, it's worthwhile to look back in time on the training data to calculate a new feature based on the amount of time a house sat on the market before it sold.

In [ ]:

```
# Import a needed function
from pyspark.sql.functions import datediff

# Assess the accuracy of the days on market field compared to list date and off
 market date
df.withColumn('Days_on_market', datediff('OFFMARKETDATE', 'LISTDATE'))\
            [['Days_on_market', 'DAYSONMARKET']].show(10)
```

Lag and Windows

Another example is using window and lag functions. These allow for the creation of new features that capture the lag in time events, such as when houses are listed on the market or perhaps when reports are submitted or purchases are made.

In [ ]:

```
# Create new feature based on difference b/t list date and off market date
mort_df = df.withColumn('Days_on_market',
                        datediff('OFFMARKETDATE', 'LISTDATE'))

# Create window
w = Window().orderBy(mort_df['LISTDATE'])

# Create lag column
mort_df = mort_df.withColumn('LISTDATE-1', lag('LISTDATE', count=2).over(w))

# Calculate difference between date columns
mort_df = mort_df.withColumn('Days_bt_List_Lag',
                             datediff('LISTDATE-1', 'LISTDATE'))

# Print results
mort_df.select('Days_bt_List_Lag','LISTDATE','LISTDATE-1').distinct().show(15)
```

Feature Engineering for Text Data

Extracting text to new features with when()

Features often contain text, which often comes unstructured. Let's apply structure to text data by creating a new binary feature based on the presence of a keyword or phrase in a text column.

In [ ]:

```
# Take a look at an unstructured text feature
df[['GARAGEDESCRIPTION']].show(10, truncate=100)
```

In [ ]:

```
# Import when function
from pyspark.sql.functions import when

# Create boolean conditions for string matches
# Note the use of the '%' wildcards that act just like they do in SQL
has_attached_garage = df['GARAGEDESCRIPTION'].like('%Attached Garage%')
has_detached_garage = df['GARAGEDESCRIPTION'].like('%Detached Garage%')

# Conditional value assignment
# The otherwise() function catches any missing or non-conforming data
df.withColumn('has_attached_garage',
              (when(has_attached_garage, 1)
              .when(has_detached_garage, 0)
              .otherwise(None)))[['GARAGEDESCRIPTION', 'has_attached_garage']].s
how(10, truncate=100)
```

Splitting, exploding, and pivoting

Did you notice above that the GARAGEDESCRIPTION feature contains other interesting details about the house? We could manually run through each of those details using when() statements, but it is also possible to split string features on a specified character, create a new list-like array, pivot using those values, and then join them back to the original df to add a bunch of new features in one series of steps.

Be careful with the number of distinct elements that you pivot on or else your feature space can explode in a bad way.

In [ ]:

```
# Import needed functions
from pyspark.sql.functions import split, explode

# Note that both split() and explode() take as arguments the
# df['col_name'] and not just 'col_name' as with some other functions

# Convert string to list-like array
# note the comma AND space to split on here
df = df.withColumn('garage_list', split(df['GARAGEDESCRIPTION'], ', '))

# Explode the values into new records
ex_df = df.withColumn('ex_garage_list', explode(df['garage_list']))

# Inspect the values
ex_df[['ex_garage_list']].distinct().show(15, truncate=50)
```

The code chunk above first splits the ‘GARAGEDESCRIPTION’ feature on commas with a trailing space. This creates a list-like array of individual words or phrases that can then be ‘exploded’. This takes individual element in a given array and creates a new row for each. This is somewhat like Pandas melt of dplyr gather but for arrays that are within a Spark column. In other words, where there was previously one long description in the ‘GARAGEDESCRIPTION’ feature, splitting and exploding creates a new row for each observation based on the various elements of the description. If one house has both an attached garage and a separate detached garage, splitting and exploding would result in two rows for that house.

This is helpful, but not exactly ideal from a feature engineering for machine learning perspective. This is where pivoting comes in handy. With pivot, monotonically increasing i.d., lit, and coalesce, the distinct categories in the ex_garage_list column as seen above can each become individual columns with a binary presence/absence for each home in the dataset. This is precisely the kind of transformation needed for feeding the new features into a machine learning model. After all, homes with fancy heated garages are probably more valuable than homes with gravel driveways only.

In [ ]:

```
from pyspark.sql.functions import monotonically_increasing_id, lit, first, coale
sce

# First, give each row in the df a unique id number, 'NO' for 'number'
ex_df = ex_df.select("*").withColumn("NO", monotonically_increasing_id())

# Create a dummy column of constant values - lit() stands for literal and is
# often needed when interfacing pyspark.sql.Column methods with a standard
# Python scalar.
# In other words, lit() is used to allow single values where an entire column
# is expected in a function call. Spark doesn't broadcast easily like
# Pandas in the current version.
ex_df = ex_df.withColumn('constant_val', lit(1))

# Pivot the values into boolean columns
piv_df = ex_df.groupby('NO').pivot('ex_garage_list').agg(coalesce(first('constan
t_val')))

piv_df.columns # lots of new features
```

In [ ]:

```
# Now, to join those new features to our original dataset

# Join the dataframes together and fill null
joined_df = ex_df.join(piv_df, on='NO', how='left')

# Columns to zero fill
zfill_cols = piv_df.columns

# Zero fill the pivoted values
zfilled_df = joined_df.fillna(0, subset=zfill_cols)

 # Just added around 30 new features all from one old text feature!
len(zfilled_df.columns)
```

Binarizing, Bucketing & Encoding

Binarizing

A binarizer can turn a feature of continuous measurements into a binary (0,1) feature based on whether or not feature observations exceed a given threshold. This can be useful in turning a more complex range of information (i.e. continuous values) into more a straightforward presence/absence feature. Examples might include the presence/absence of a fireplace instead of how many fireplaces or the presence of absence of sqft below ground to indicate a finished or partially finished basement as opposed the raw square footage in the basement that was finished.

For this example, I'm curious about houses that were listed on the market on a Thursday, Friday, or Saturday - Those were shown to be the more popular list days of the week (see histogram above). Perhaps this might be relevant details for a feature for various models.

In [ ]:

```
# Import transformer
from pyspark.ml.feature import Binarizer

# Create the transformer - 5 is Thursday based on the Spark timestamp
binarizer = Binarizer(threshold=4.0, inputCol='List_Day_of_Week',
                      outputCol='Listed_On_Thurs_Fri_Sat')

# Change column of interest from bigint to double for the Binarizer
df = df.withColumn('List_Day_of_Week', df['List_Day_of_Week'].cast('double'))

# Apply the transformation to df and verify
binarizer.transform(df)[['LISTDATE','List_Day_of_Week',
                         'Listed_On_Thurs_Fri_Sat']].show(10)
```

Bucketing

Bucketing applies a similar concept to binarizing, but in this case it is for creating discrete groups within a given range based on split values specified by the user. Examples to consider here might be cases where you have one or two long tails in your data and want to simplify the feature-space by bucketing. In the example here, many high-income buyers are less likely to care about the presence of bedrooms beyond 5 (since very few homes have more than 5 bedrooms), so we can keep much of the existing structure in the feature that conveys the number of bedrooms but reduce the feature-space for homes with more than 5 bedrooms to a single bucket.

In [ ]:

```
from pyspark.ml.feature import Bucketizer

# Plot distribution of sample_df
sns.distplot(sample_df['Bedrooms'], axlabel='Bedrooms')
plt.show()

# Create the bucket splits and bucketizer
splits = [0, 1, 2, 3, 4, 5, float('Inf')]
buck = Bucketizer(splits=splits, inputCol='Bedrooms',
                  outputCol='bedrooms_bucket')

# Apply the transformation to df and verify results
buck.transform(df)[['Bedrooms',
                    'bedrooms_bucket']].sort(col('Bedrooms').desc()).show(15)
```

One Hot Encoding

One Hot Encoding is another great way to handle categorial variables. You may notice after running the chunk below that the implementation in PySpark is different than Pandas get_dummies() as it puts everything into a single column of type vector rather than a new column for each value. It's also different from sklearn's OneHotEncoder in that the last categorical value is captured by a vector of all zeros.

In [ ]:

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer

# Map strings to numbers with string indexer
string_indexer = StringIndexer(inputCol='SchoolDistrictNumber',
                               outputCol='School_Index')
indexed_df = string_indexer.fit(df).transform(df)

# Onehot encode indexed values
encoder = OneHotEncoder(inputCol='School_Index', outputCol='School_Vec')
encoder.transform(indexed_df)[['SchoolDistrictNumber', 'School_Index',
                               'School_Vec']].show(truncate=100)
```

And there we have it! I hope you enjoyed this entree into the world of feature engineering with PySpark and Spark SQL. This really only scratches the surface of what’s possible. I highly encourage you to find a data set that interests you and trying these new found skills out on your own unique project!

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```

```