Machine Learning with PySpark (Spark Series Part 3)

JULY 17, 2019

Welcome to the third installment of the PySpark series. In this post, we will cover a basic introduction to machine learning with PySpark. This includes model selection, performing a train-test split on a date feature, considerations to think about before running a PySpark ML model, working with PySpark's vectors, training regression models, evaluating the models, and saving a loading models. All of these machine learning efforts are applied in order to predict the final sales price of a subset of homes from St. Paul Minnesota.

If this is your first time here, make sure to take a look at Spark Series Part 1 - EDA and Part 2 - Feature Engineering to get better acquainted with the dataset under consideration. I'll assume in this post that you've already followed the instructions in Parts 1 and 2 to get Spark up and running on your machine and have the dataset downloaded and ready for analysis. You can also take a look at the entire Spark Series as a single Jupyter Notebook on my GitHub page.

Without further ado, let's jump in and get coding!

Model and Evaluation Metrics Selection

With all of the feature engineering in the Spark Series Part 2 behind us, it's now time to consider how those engineered features can be incorporated into a machine learning model and how that model can be assessed for its performance.

The first step in training a model is to make an informed decision as to what model(s) is/are the most appropriate for the data at hand and the question being asked of the data. In a business context, it's often important to start simple and fail fast or iterate. After all, modeling is often a much quicker task than acquiring and cleaning the data necessary to fit a model. Furthermore, simpler models like regressions or tree-based approaches are often quite quick to train and require minimal compute resources. Additionally, they provide the added bonus of insights into which features are more sensitive than others in fitting the model. These are some major advantages over jumping in with more complex approaches like deep learning or reinforcement learning. That work can be done once an MVP (minimum viable product) is in production as part of the continuous integration continuous development (CI/CD) process.

In this guide, the running theme has been to predict the sale price of a house. To predict that kind of continuous value (i.e. $$), we'll want to use methods from Spark ml.regression. Standard linear, generalized linear, or isotonic regression models are available, but the dataset we are working with doesn't exactly satisfy the assumptions of those models. Instead, we'll turn to some tree base approaches - Random Forest and Gradient Boosted Tree Regression for this task.

Choosing an evaluation metric is often just as important as choosing a model and should be done at the outset of an ML project. Businesses live and breath by KPIs (key performance indicators), and machine learning models aren't so different. Just as businesses take great care in defining and monitoring KPIs, so too should data scientists/analysts/researchers when evaluating models. Given that we are fitting regression models, the general idea is to determine how close each prediction the model generates is to the actual sales price. Unlike the 'Price Is Right' tv game show, a regression evaluation metric must consider performance when a model both undershoots and overshoots the target. There are a number of great evaluation metrics that capture this, but two are more commonly used than others: RMSE (root mean squared error) and R^2 (R Squared). RMSE represents the sample standard deviation of the differences between the predicted values and the observed values. RMSE is computationally simple, easily understandable, and is commonly used by ML practitioners. R Squared is often used for explanatory purposes to determine how well the selected independent variable(s) explain the variability in the dependent variable(s). In other words, how well the features that we engineered are able to explain the range of variation in sales prices in our dataset. We'll use these two metrics here. You can take a look here if you'd like to learn more about RMSE or R^2.

Train-Test Split on Date

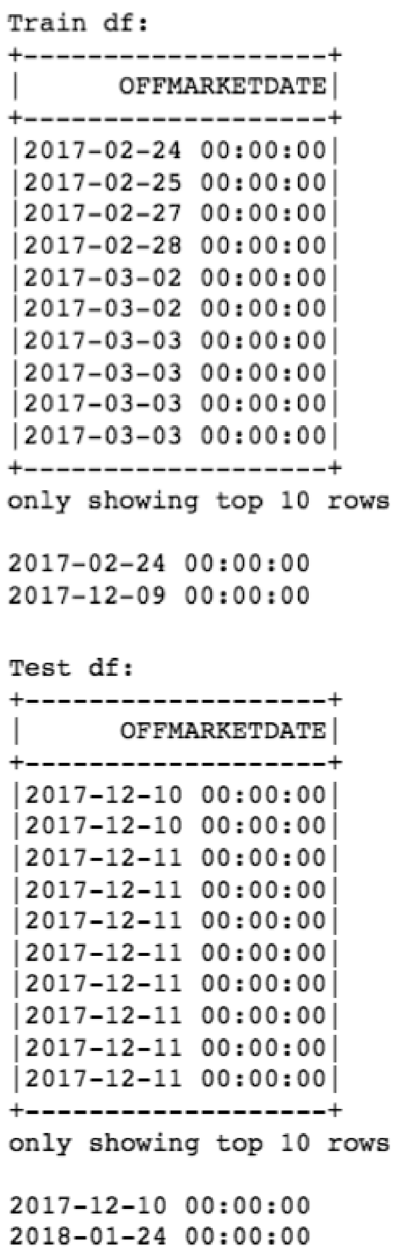
As with any machine learning modeling process, it's necessary to first split the data into training and testing (and perhaps also validation) splits. This is because machine learning models must be generalizable to unseen data. In order to ensure the generalizability of a model, it must be assessed with testing and/or validation data that it hasn't seen before. When working with time series data, it's necessary to split the dataset on the time series dates in order to accomplish this. There are a variety of reasons for this which we'll get into shortly, but for now let's get splitting.

For the first split, we will define a function that takes as its input the number of test days that we'd like to include in our testing set and returns the date on which we should make the split. This doesn't take into consideration the size of the training and testing datasets, only the number of days to consider, so be forewarned.

```python
from datetime import timedelta

# Here we will write a function to split the dataset into a train and
# test split based on the number of days from the most recent date
# in the dataset

def train_test_split_date(df, split_col, test_days=45):
  """Calculate the date to split test and training sets"""
  # Find how many days our data spans
  max_date = df.agg({split_col: 'max'}).collect()[0][0]
  min_date = df.agg({split_col: 'min'}).collect()[0][0]
  # Subtract an integer number of days from the last date in dataset
  split_date = max_date - timedelta(days=test_days)
  return split_date

# Find the date to use in spitting test and train
split_date = train_test_split_date(df, 'OFFMARKETDATE')

# Create Sequential Test and Training Sets
train_df = df.where(df['OFFMARKETDATE'] < split_date)
test_df = df.where(df['OFFMARKETDATE'] >= split_date)
            .where(df['LISTDATE'] <= split_date)

# Maybe not the most reasonable split size, but you get the idea
print(test_df.count())
print(train_df.count())
```

```
154
4828
```

```python
# Here are the dates from the train/test sorted by recency
print('Train df:')
train_df[['OFFMARKETDATE']].sort(col('OFFMARKETDATE')).show(10)
print(train_df.agg({'OFFMARKETDATE': 'min'}).collect()[0][0])
print(train_df.agg({'OFFMARKETDATE': 'max'}).collect()[0][0])
print('Test df:')
test_df[['OFFMARKETDATE']].sort(col('OFFMARKETDATE')).show(10)
print(test_df.agg({'OFFMARKETDATE': 'min'}).collect()[0][0])
print(test_df.agg({'OFFMARKETDATE': 'max'}).collect()[0][0])
```



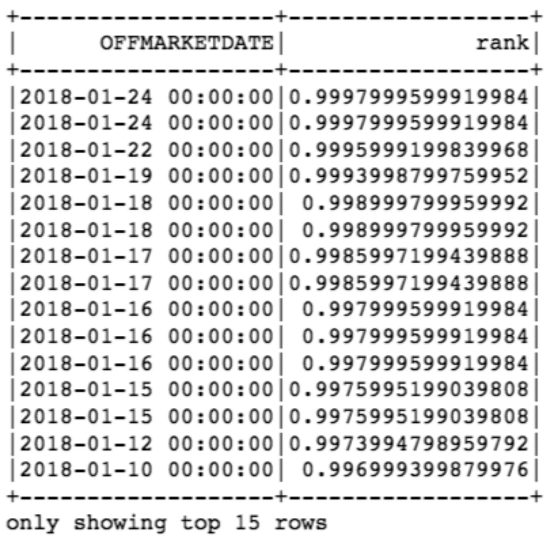
```
Train df:
+-------------------+
|      OFFMARKETDATE|
+-------------------+
|2017-02-24 00:00:00|
|2017-02-25 00:00:00|
|2017-02-27 00:00:00|
|2017-02-28 00:00:00|
|2017-03-02 00:00:00|
|2017-03-02 00:00:00|
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
+-------------------+
only showing top 10 rows

2017-02-24 00:00:00
2017-12-09 00:00:00

Test df:
+-------------------+
|      OFFMARKETDATE|
+-------------------+
|2017-12-10 00:00:00|
|2017-12-10 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
|2017-12-11 00:00:00|
+-------------------+
only showing top 10 rows

2017-12-10 00:00:00
2018-01-24 00:00:00
```

Split on Date by a Percentage of Days

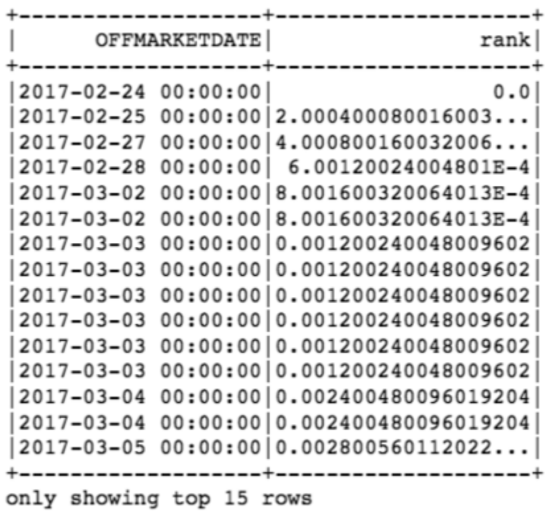
Aside from splitting on a particular number of testing days, it's quite common to create training and testing datasets based on a predetermined fraction of the full dataset. Common training/testing splits in this regard might be 70/30, 80/20, or 90/10. In this case, the testing set would encompass the most recent 30%, 20%, or 10% of observations respectively. In this example below, we will create and 80/20 training/testing split using the percent_rank() and Window functions in PySpark.

```
# First step is to create a column to rank the dates that homes were
# taken off the market
from pyspark.sql.functions import percent_rank
from pyspark.sql import Window

df = df.withColumn("rank", percent_rank().over(Window.partitionBy()
                                           .orderBy("OFFMARKETDATE")))

# Now we can use that rank to split into a train/test
# Here we want the test set to be the most recent days,
# so rank will be greater than 80%
```

```
test_df = df.where("rank >= .8").drop("rank")
train_df = df.where("rank < .8").drop("rank")

# Just to see how the rank approach taken above works
df[['OFFMARKETDATE', 'rank']].sort(col('rank').desc()).show(15)
df[['OFFMARKETDATE', 'rank']].sort(col('rank')).show(15)
```

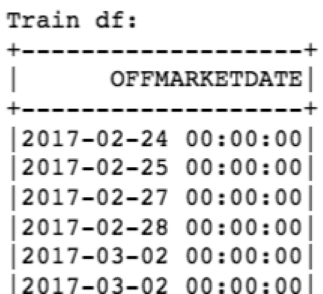


```
+-------------------+------------------+
|      OFFMARKETDATE|              rank|
+-------------------+------------------+
|2018-01-24 00:00:00|0.9997999599919984|
|2018-01-24 00:00:00|0.9997999599919984|
|2018-01-22 00:00:00|0.9995999199839968|
|2018-01-19 00:00:00|0.9993998799759952|
|2018-01-18 00:00:00| 0.998999799959992|
|2018-01-18 00:00:00| 0.998999799959992|
|2018-01-17 00:00:00|0.9985997199439888|
|2018-01-17 00:00:00|0.9985997199439888|
|2018-01-16 00:00:00| 0.997999599919984|
|2018-01-16 00:00:00| 0.997999599919984|
|2018-01-16 00:00:00| 0.997999599919984|
|2018-01-15 00:00:00|0.9975995199039808|
|2018-01-15 00:00:00|0.9975995199039808|
|2018-01-12 00:00:00|0.9973994798959792|
|2018-01-10 00:00:00| 0.996999399879976|
+-------------------+------------------+
only showing top 15 rows
```

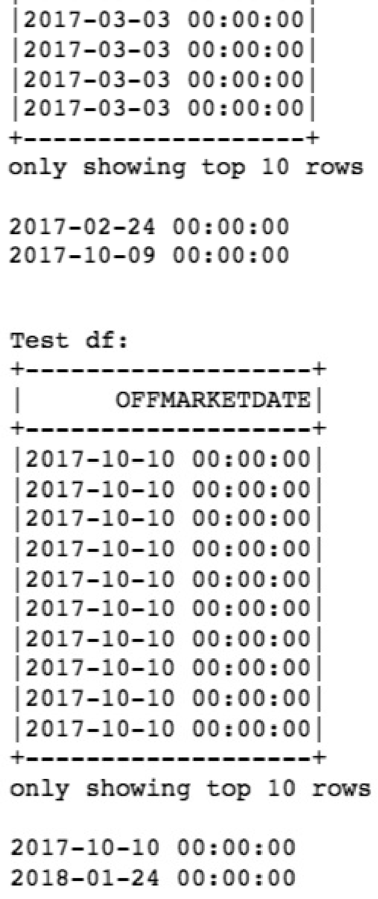


```
+-------------------+--------------------+
|      OFFMARKETDATE|                rank|
+-------------------+--------------------+
|2017-02-24 00:00:00|                 0.0|
|2017-02-25 00:00:00|2.000400080016003...|
|2017-02-27 00:00:00|4.000800160032006...|
|2017-02-28 00:00:00| 6.00120024004801E-4|
|2017-03-02 00:00:00|8.00160032006401 3E-4|
|2017-03-02 00:00:00|8.00160032006401 3E-4|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-03 00:00:00|0.001200240048009602|
|2017-03-04 00:00:00|0.002400480096019204|
|2017-03-04 00:00:00|0.002400480096019204|
|2017-03-05 00:00:00|0.002800560112022...|
+-------------------+--------------------+
only showing top 15 rows
```

```
# Here are the dates from the train/test sorted by recency
print('Train df:')
train_df[['OFFMARKETDATE']].sort(col('OFFMARKETDATE')).show(10)
print(train_df.agg({'OFFMARKETDATE': 'min'}).collect()[0][0])
print(train_df.agg({'OFFMARKETDATE': 'max'}).collect()[0][0])

print('')
print('Test df:')
test_df[['OFFMARKETDATE']].sort(col('OFFMARKETDATE')).show(10)
print(test_df.agg({'OFFMARKETDATE': 'min'}).collect()[0][0])
print(test_df.agg({'OFFMARKETDATE': 'max'}).collect()[0][0])
```



```
Train df:
+-------------------+
|      OFFMARKETDATE|
+-------------------+
|2017-02-24 00:00:00|
|2017-02-25 00:00:00|
|2017-02-27 00:00:00|
|2017-02-28 00:00:00|
|2017-03-02 00:00:00|
|2017-03-02 00:00:00|
```

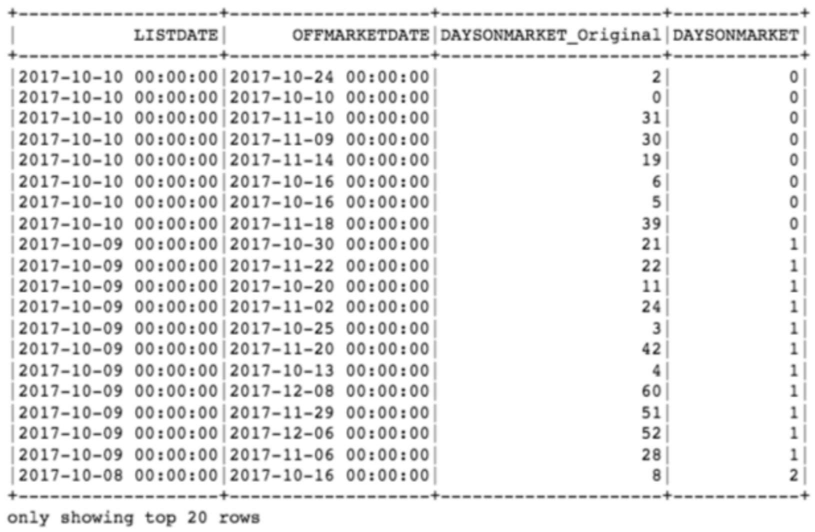

```
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
|2017-03-03 00:00:00|
+-------------------+
only showing top 10 rows

2017-02-24 00:00:00
2017-10-09 00:00:00


Test df:
+-------------------+
|      OFFMARKETDATE|
+-------------------+
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
|2017-10-10 00:00:00|
+-------------------+
only showing top 10 rows

2017-10-10 00:00:00
2018-01-24 00:00:00
```

Adjusting the time feature to reflect known information

Leaking information to your model during training can be dangerous. Data leakage will cause your model to have very optimistic metrics for accuracy but once real data is run through it the results are often very disappointing. In the example of predicting home prices, the number of days a house sits on the market is often an important predictor of its final sales price. Homes that get snatched up quickly are more likely to be priced and sold at or above market rates while homes that linger on the market are likely to see price reductions and go for below market rates once they finally get sold. As a result, it's necessary to ensure that the information about days on market is not 'leaked' into the model, otherwise it will not be as generalizable to unseen data.

In the chunk below, we are going to ensure that DAYSONMARKET only reflects known information at the time of predicting the value. In other words, if the house is still on the market, we don't know how many more days it will stay on the market. We need to adjust our test_df to reflect what information we currently have as of 2017-10-10 - the first date present in the testing dataset.

```
from pyspark.sql.functions import datediff, to_date, lit

# Earliest date in the 80/20 train/test split
split_date = to_date(lit('2017-10-10'))
# Create Sequential Test set - to simulate 'real world' data where the off
# market date is unknown.
# After all, what use is a home price prediction model for homes that
# have already been sold?
test_df = df.where(df['OFFMARKETDATE'] >= split_date) \
            .where(df['LISTDATE'] <= split_date)

# Create a copy of DAYSONMARKET to review later
test_df = test_df.withColumn('DAYSONMARKET_Original', test_df['DAYSONMARKET'])

# Recalculate DAYSONMARKET from what we know on our split date - '2017-10-10'
# I.e. we cannot allow our model to peer into the future and know when
# properties will go off market when testing the model against simulated
# 'real-world' data
test_df = test_df.withColumn('DAYSONMARKET', datediff(split_date, 'LISTDATE'))

# Review the difference
test_df[['LISTDATE', 'OFFMARKETDATE',
         'DAYSONMARKET_Original', 'DAYSONMARKET']]\
        .sort(col('LISTDATE').desc()).show()
```



```
+-------------------+-------------------+---------------------+------------+
|           LISTDATE|      OFFMARKETDATE|DAYSONMARKET_Original|DAYSONMARKET|
+-------------------+-------------------+---------------------+------------+
|2017-10-10 00:00:00|2017-10-24 00:00:00|                    2|           0|
|2017-10-10 00:00:00|2017-10-10 00:00:00|                    0|           0|
|2017-10-10 00:00:00|2017-11-10 00:00:00|                   31|           0|
|2017-10-10 00:00:00|2017-11-09 00:00:00|                   30|           0|
|2017-10-10 00:00:00|2017-11-14 00:00:00|                   19|           0|
|2017-10-10 00:00:00|2017-10-16 00:00:00|                    6|           0|
|2017-10-10 00:00:00|2017-10-16 00:00:00|                    5|           0|
|2017-10-10 00:00:00|2017-11-18 00:00:00|                   39|           0|
|2017-10-09 00:00:00|2017-10-30 00:00:00|                   21|           1|
|2017-10-09 00:00:00|2017-11-22 00:00:00|                   22|           1|
|2017-10-09 00:00:00|2017-10-20 00:00:00|                   11|           1|
|2017-10-09 00:00:00|2017-11-02 00:00:00|                   24|           1|
|2017-10-09 00:00:00|2017-10-25 00:00:00|                    3|           1|
|2017-10-09 00:00:00|2017-11-20 00:00:00|                   42|           1|
|2017-10-09 00:00:00|2017-10-13 00:00:00|                    4|           1|
|2017-10-09 00:00:00|2017-12-08 00:00:00|                   60|           1|
|2017-10-09 00:00:00|2017-11-29 00:00:00|                   51|           1|
|2017-10-09 00:00:00|2017-12-06 00:00:00|                   52|           1|
|2017-10-09 00:00:00|2017-11-06 00:00:00|                   28|           1|
|2017-10-08 00:00:00|2017-10-16 00:00:00|                    8|           2|
+-------------------+-------------------+---------------------+------------+
only showing top 20 rows
```

Final considerations before running a PySpark ML model

After doing a lot of feature engineering it's a good idea to take a step back and look at what you've created. If you've used some automation techniques on your categorical features like exploding or OneHot Encoding, you may find that you now have hundreds of new binary features. While the subject of feature selection is material for a whole other guide, there are some quick steps you can take to reduce the dimensionality of your dataset, making it more amenable for ML applications.

We'll start by dropping columns with less than 30 observations. Why 30? Well, 30 is a common minimum number of observations for statistical significance. Any less than that and the relationships can cause overfitting because of a sheer coincidence!

Think about bringing in external data

External data can often be the difference between a good model and a great model. For home sales price, it might be worthwhile to consider data such as mortgage rates, the walk/bike score of a house, the potential for solar panels or geothermal heating, seasonal indicators such as bank holidays or considerations of the time of the year that a home is listed (e.g. changes from mean/median home value for a given month/quarter/season compared to the entire year), etc. You might consult a local realtor who would more than likely be quite happy to provide quarterly or yearly reports that you could glean data from an incorporate it into your model.

Make sure your data fulfill the chosen models assumptions

This is a no brainer, but bares repeating - if the model you choose to use comes with assumptions, run unit some tests to ensure your data satisfy those assumptions. Often, models assume that the data is normally distributed or aligns with some other distribution. Unit tests can ensure that this is the case.

PySpark ML vectors

Just to add one extra layer of complexity when using Spark, the PySpark machine learning algorithms require all features to be provided in a single column as a vector. Luckily, there is a fit/transform function provided to handle that, but it can also affect whether or not your features are truly ready for modeling. For example, while random forest regression can handle missing values, vectors cannot - so we need to assign missings/NaNs to a value outside the range of the values in the feature such that those observations can still be included but satisfy requirements for vectorization of all of the columns into one.

For our running example, most columns are strings which are not supported by vectors. Since the focus of this guide is on feature engineering and not modeling, per se, those features will simply be dropped.

First, remove features based on missing value threshold

This is for binary (0/1) features.

The code chunk below is commented out because it will simply throw an error - I did not include all of the binary features into the df

```
## If you end up with a bunch of binary features, you can make sure to include only
#  those that have at least 30 positive values (e.g. 1's) with the code below

#obs_threshold = 30
#cols_to_remove = list()

# Inspect first 10 binary columns in list
#for col in binary_cols[0:10]:
  # Count the number of 1 values in the binary column
#  obs_count = df.agg({col: 'sum'}).collect()[0][0]
  # If less than our observation threshold, remove
#  if obs_count < obs_threshold:
#    cols_to_remove.append(col)

# Drop columns and print starting and ending dataframe shapes
#new_df = df.drop(*cols_to_remove)
```

Prepare Data for Vectorization

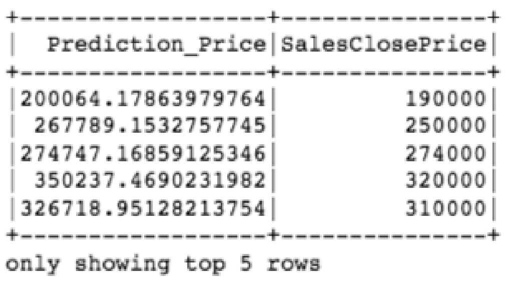
```
# Replace missing values
train_df = train_df.fillna(-1)
test_df = test_df.fillna(-1)

# Have to naively remove all string columns for the vectorizer
string_cols = []
for col in range(1, len(train_df.columns)):
    if train_df.dtypes[col][1] == 'string':
        string_cols.append(train_df.dtypes[col][0])

# The * character unpacks a list to apply it on an
# element-by-element basis
train_df_for_vec = train_df.drop(*string_cols)
train_df_for_vec = train_df_for_vec.drop(*['LISTDATE',
                                           'OFFMARKETDATE',
                                           'MLSID',
                                           'garage_list'])

test_df_for_vec = test_df.drop(*string_cols)
test_df_for_vec = test_df_for_vec.drop(*['LISTDATE',
                                         'OFFMARKETDATE',
                                         'MLSID',
                                         'garage_list'])

# Define columns to be converted to vectors for model
feature_cols = list(train_df_for_vec.columns)

# Remove the dependent variable from the list
feature_cols.remove('SalesClosePrice')
```

Turn all features into a single feature vector

```python
from pyspark.ml.feature import VectorAssembler

# Create the vector assembler transformer
vec = VectorAssembler(inputCols=feature_cols, outputCol='features')

# Apply the vector transformer to data
train_df_vec = vec.transform(train_df_for_vec)
test_df_vec = vec.transform(test_df_for_vec)
```

Train some regression models

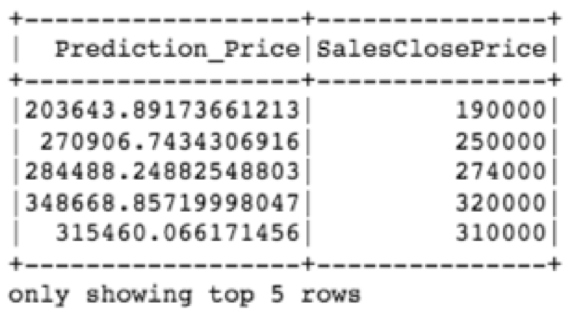
```python
from pyspark.ml.regression import GBTRegressor

# Train a Gradient Boosted Trees (GBT) model.
gbt = GBTRegressor(featuresCol='features',
                           labelCol='SalesClosePrice',
                           predictionCol="Prediction_Price",
                           seed=42
                           )

# Train model.
gbt_model = gbt.fit(train_df_vec)

# Make predictions
gbt_predictions = gbt_model.transform(test_df_vec)

# Assess predictions
gbt_predictions.select('Prediction_Price', 'SalesClosePrice').show(5)
```



```
+------------------+---------------+
|  Prediction_Price|SalesClosePrice|
+------------------+---------------+
|200064.17863979764|         190000|
| 267789.1532757745|         250000|
|274747.16859125346|         274000|
| 350237.4690231982|         320000|
|326718.95128213754|         310000|
+------------------+---------------+
only showing top 5 rows
```

```python
from pyspark.ml.regression import RandomForestRegressor

# Train a Gradient Boosted Trees (GBT) model.
rf = RandomForestRegressor(featuresCol='features',
                           labelCol='SalesClosePrice',
                           predictionCol="Prediction_Price",
                           seed=42
                           )

# Train model.
rf_model = rf.fit(train_df_vec)

# Make predictions
rf_predictions = rf_model.transform(test_df_vec)
```

```
# Assess predictions
rf_predictions.select('Prediction_Price', 'SalesClosePrice').show(5)
```



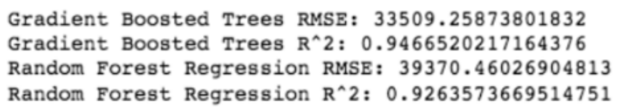
```
+------------------+---------------+
|  Prediction_Price|SalesClosePrice|
+------------------+---------------+
|203643.89173661213|         190000|
| 270906.7434306916|         250000|
|284488.24882548803|         274000|
|348668.85719998047|         320000|
|  315460.066171456|         310000|
+------------------+---------------+
only showing top 5 rows
```

Model Evaluation

Even though the impetus for this guide is feature engineering, I actually incorporated very minimal feature engineering into the final data fed into the regression models. Nevertheless, and perhaps surprisingly, the models both tend to perform quite well right out of the box. Depending on the business context, these models might be sufficient, but certainly the methods for feature engineering described Part 2 of this series would be able to push model performance much higher.

The next step is to calculate model performance using our pre-determined evaluation metrics.

```
from pyspark.ml.evaluation import RegressionEvaluator

# Select columns to compute test error
evaluator = RegressionEvaluator(labelCol='SalesClosePrice',
                                predictionCol='Prediction_Price')
# Dictionary of model predictions to loop over
models = {'Gradient Boosted Trees': gbt_predictions,
          'Random Forest Regression': rf_predictions}
for key, preds in models.items():
  # Create evaluation metrics
  rmse = evaluator.evaluate(preds, {evaluator.metricName: 'rmse'})
  r2 = evaluator.evaluate(preds, {evaluator.metricName: 'r2'})

  # Print Model Metrics
  print(key + ' RMSE: ' + str(rmse))
  print(key + ' R^2: ' + str(r2))
```



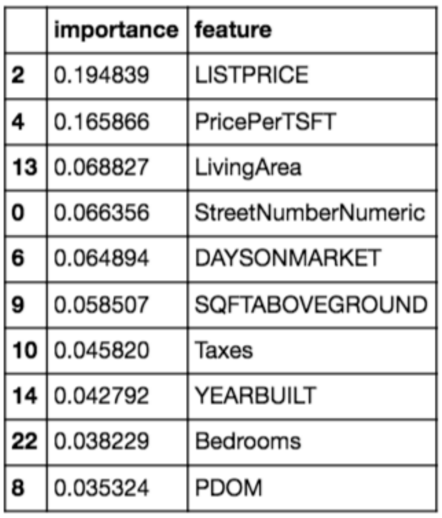
```
Gradient Boosted Trees RMSE: 33509.25873801832
Gradient Boosted Trees R^2: 0.9466520217164376
Random Forest Regression RMSE: 39370.46026904813
Random Forest Regression R^2: 0.9263573669514751
```

The RMSEs might seem high, but remember that these are home prices that are being predicted, with sale prices in the hundreds of thousands of dollars.

Explore Feature Importance

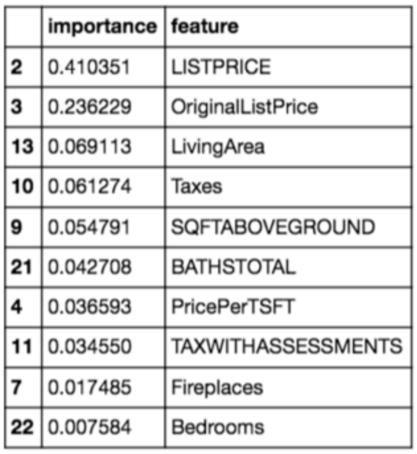
As mentioned previously, one of the upsides of the tree-based regressors we used is that the importance of individual features to the final model can be determined. This is useful for all sorts of reasons such as identifying the value of feature engineering, knowing which features might not be needed in order to speed up training for future models, and being able to report to stakeholders as to how your model is working.

```
# Convert feature importances to a pandas column -
# no need to use Spark with such small dataframes
gbt_fi_df = pd.DataFrame(gbt_model.featureImportances.toArray(),
                         columns=['importance'])
rf_fi_df = pd.DataFrame(rf_model.featureImportances.toArray(),
                        columns=['importance'])

# Convert list of feature names to pandas column
gbt_fi_df['feature'] = pd.Series(feature_cols)

# Sort the data based on feature importance
gbt_fi_df.sort_values(by=['importance'], ascending=False, inplace=True)

# Inspect Results
gbt_fi_df.head(10)
```



	importance	feature
2	0.194839	LISTPRICE
4	0.165866	PricePerTSFT
13	0.068827	LivingArea
0	0.066356	StreetNumberNumeric
6	0.064894	DAYSONMARKET
9	0.058507	SQFTABOVEGROUND
10	0.045820	Taxes
14	0.042792	YEARBUILT
22	0.038229	Bedrooms
8	0.035324	PDOM

```
# Convert list of feature names to pandas column
rf_fi_df['feature'] = pd.Series(feature_cols)

# Sort the data based on feature importance
rf_fi_df.sort_values(by=['importance'], ascending=False, inplace=True)

# Inspect Results
rf_fi_df.head(10)
```



	importance	feature
2	0.410351	LISTPRICE
3	0.236229	OriginalListPrice
13	0.069113	LivingArea
10	0.061274	Taxes
9	0.054791	SQFTABOVEGROUND
21	0.042708	BATHSTOTAL
4	0.036593	PricePerTSFT
11	0.034550	TAXWITHASSESSMENTS
7	0.017485	Fireplaces
22	0.007584	Bedrooms

Save and load models

As a final step, I'll show how to save one of the trained models and load it in to any other application.

```
# Save the models
gbt_model.save('house_price_regression_gbt_model')
rf_model.save('house_price_regression_rf_model')

from pyspark.ml.regression import RandomForestRegressionModel

# Load the model
rf_model_loaded = RandomForestRegressionModel \
                   .load('house_price_regression_rf_model')
```

And there you have it! In this Spark Series, I provided an overview of EDA, feature engineering, and model training using the PySpark API. This should give you a strong foundation onto which you can train other models for all kinds of applications, from recommender systems to classification tasks to unsupervised ML applications like clustering or customer segmentation.

There's a lot that wasn't covered in this series. For example ML parameter tuning, working with streaming data, and Spark SQL to name a few. So get out there and start exploring the Spark universe!

1 LIKES   SHARE

Newer   Older