Reference

Data Camp course.

Data Exploration

Check Version

Checking the version of which Spark and Python installed is important as it changes very quickly and drastically. Reading the wrong documentation can cause lots of lost time and unnecessary frustration!

In [1]:
```
import findspark
findspark.init('c:/spark/spark')
```

In [2]:
```
import pyspark # only run after findspark.init()
from pyspark.sql import SparkSession
```

In [3]:
```
spark = SparkSession.builder.getOrCreate()
```

In [4]:
```
# Return spark version
print(spark.version)

# Return python version
import sys
print(sys.version_info)
```

```
2.4.0
sys.version_info(major=3, minor=6, micro=4, releaselevel='final', serial=0)
```

In [5]:
```
df = spark.sql("select 'spark' as hello")
df.show()
```

```
+-----+
|hello|
+-----+
|spark|
+-----+
```

Load in the data

Reading in data is the first step to using PySpark for data science! Let's leverage the new industry standard of parquet files!

```
In [ ]: # I have problem reading the following file
        # I may read with pandas as data frame , and then transformed it into spark data 
        # Read the file into a dataframe
        df = spark.read.parquet("Real_Estate.csv")

        # Print columns in dataframe
        print(df.columns)
```

What are we predicting?

Which of these fields (or columns) is the value we are trying to predict for?

TAXES SALESCLOSEPRICE DAYSONMARKET LISTPRICE

```
In [ ]: # Select our dependent variable
        Y_df = df.select(['SALESCLOSEPRICE'])

        # Display summary statistics
        Y_df.describe().show()
```

+-------+------------------+ |summary| SALESCLOSEPRICE| +-------+------------------+ | count| 5000| | mean| 262804.4668| | stddev|140559.82591998563| | min| 48000| | max| 1700000| +-------+------------------+

Correct! We want to know how much a house will actually sell for. We can see the range of values it has here and the average which will help us in our next steps!

Verifying Data Load

Let's suppose each month you get a new file. You know to expect a certain number of records and columns. In this exercise we will create a function that will validate the file loaded.

```
In [ ]: def check_load(df, num_records, num_columns):
          # Takes a dataframe and compares record and column counts to input
          # Message to return if the critera below aren't met
          message = 'Validation Failed'
          # Check number of records
          if num_records == df.count():
            # Check number of columns
            if num_columns == len(df.columns):
              # Success message
              message = 'Validation Passed'
          return message

        # Print the data validation message
        print(check_load(df, 5000, 74))
```

Verifying DataTypes

In the age of data we have access to more attributes than we ever had before. To handle all of them we will build a lot of automation but at a minimum requires that their datatypes be correct. In this exercise we will validate a dictionary of attributes and their datatypes to see if they are correct. This dictionary is stored in the variable validation_dict and is available in your workspace.

In [ ]:
```
# Create list of actual dtypes to check
actual_dtypes_list = df.dtypes
print(actual_dtypes_list)

# Iterate through the list of actual dtypes tuples
for attribute_tuple in actual_dtypes_list:

  # Check if column name is dictionary of expected dtypes
  col_name = attribute_tuple[0]
  if col_name in validation_dict:

    # Compare attribute names and types
    col_type = attribute_tuple[1]
    if col_type == validation_dict[col_name]:
      print(col_name + ' has expected dtype.')
```

Using Corr()

The old adage 'Correlation does not imply Causation' is a cautionary tale. However, correlation does give us a good nudge to know where to start looking promising features to use in our models. Use this exercise to get a feel for searching through your data for the first time, trying to find patterns.

A list called columns containing column names has been created for you. In this exercise you will compute the correlation between those columns and 'SALESCLOSEPRICE', and find the maximum.

In [ ]:
```
# Name and value of col with max corr
corr_max = 0
corr_max_col = columns[0]

# Loop to check all columns contained in list
for col in columns:
    # Check the correlation of a pair of columns
    corr_val = df.corr('SALESCLOSEPRICE', col)
    # Logic to compare corr_max with current corr_val
    if corr_val > corr_max:
        # Update the column name and corr value
        corr_max = corr_val
        corr_max_col = col

print(corr_max_col)
```

LIVINGAREA

It makes sense that homes with larger living areas would be correlated with more expensive homes!

Using Visualizations: distplot

Understanding the distribution of our dependent variable is very important and can impact the type of model or preprocessing we do. A great way to do this is to plot it, however plotting is not a built in function in PySpark, we will need to take some intermediary steps to make sure it works correctly. In this exercise you will visualize the variable the 'LISTPRICE' variable, and you will gain more insights on its distribution by computing the skewness.

The matplotlib.pyplot and seaborn packages have been imported for you with aliases plt and sns.

In [ ]:
```
# 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.show()

# Import skewness function
from pyspark.sql.functions import skewness

# Compute and print skewness of LISTPRICE
print(df.agg({'LISTPRICE': 'skewness'}).collect())
```

```
[Row(skewness(LISTPRICE)=2.790448093916559)]
```
Awesome, checking the distribution visually is a great way to get an idea of what steps will need to be taken before applying a model. We can see the 'ListPrice' is mostly pushed to the left, which means its skewed. We can use the skewness function to verify this numerically rather than visually.

Using Visualizations: lmplot

Creating linear model plots helps us visualize if variables have relationships with the dependent variable. If they do they are good candidates to include in our analysis. If they don't it doesn't mean that we should throw them out, it means we may have to process or wrangle them before they can be used.

seaborn is available in your workspace with the customary alias sns.

In [ ]:
```
# Select a the relevant columns and sample
sample_df = df.select(['SALESCLOSEPRICE', 'LIVINGAREA']).sample(False, 0.5, 42)

# Convert to pandas dataframe
pandas_df = sample_df.toPandas()

# Linear model plot of pandas_df
sns.lmplot(x='LIVINGAREA', y='SALESCLOSEPRICE', data=pandas_df)
plt.show()
```

now we can see that as LivingArea increases, the price of the home increases at a relatively steady rate.

Dropping a list of columns

Our data set is rich with a lot of features, but not all are valuable. We have many that are going to be hard to wrangle into anything useful. For now, let's remove any columns that aren't immediately useful by dropping them.

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

In [ ]:
```
# Show top 30 records
df.show(30)

# List of columns to remove from dataset
cols_to_drop = ['STREETNUMBERNUMERIC', 'LOTSIZEDIMENSIONS']

# Drop columns in list
df = df.drop(*cols_to_drop)
```

Knowing just the house number doesn't tell us anything about what value the house should be. Likewise the freeform text field is likely too messy to extract useful information from. We can always come back to these after our intial model if we need more information.

Using text filters to remove records

It pays to have to ask your clients lots of questions and take time to understand your variables. You find out that Assumable mortgage is an unusual occurrence in the real estate industry and your client suggests you exclude them. In this exercise we will use isin() which is similar to like() but allows us to pass a list of values to use as a filter rather than a single one.

In [ ]:
```
# Inspect unique values in the column 'ASSUMABLEMORTGAGE'
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
text_filter = ~df['ASSUMABLEMORTGAGE'].isin(yes_values) | df['ASSUMABLEMORTGAGE']
df = df.where(text_filter)

# print count of remaining records
print(df.count())
```

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

Filtering numeric fields conditionally

Again, understanding the context of your data is extremely important. We want to understand what a normal range of houses sell for. Let's make sure we exclude any outlier homes that have sold for significantly more or less than the average. Here we will calculate the mean and standard deviation and use them to filer the near normal field log_SalesClosePrice.

```
In [ ]: from pyspark.sql.functions import mean, stddev

        # Calculate values used for outlier filtering
        mean_val = df.agg({'log_SalesClosePrice': 'mean'}).collect()[0][0]
        stddev_val = df.agg({'log_SalesClosePrice': 'stddev'}).collect()[0][0]

        # Create three standard deviation (μ ± 3σ) lower and upper bounds for data
        low_bound = mean_val - (3 * stddev_val)
        hi_bound = mean_val + (3 * stddev_val)

        # Filter the data to fit between the lower and upper bounds
        df = df.where((df['log_SalesClosePrice'] < hi_bound) & (df['log_SalesClosePrice']
```

now we've set proper constaints on our data. If we were to get new data, or the value for Jumbo Loans changes, we can dynamically refilter it!

Custom Percentage Scaling

In the slides we showed how to scale the data between 0 and 1. Sometimes you may wish to scale things differently for modeling or display purposes.

```
In [ ]: # Define max and min values and collect them
        max_days = df.agg({'DAYSONMARKET': 'max'}).collect()[0][0]
        min_days = df.agg({'DAYSONMARKET': 'min'}).collect()[0][0]

        # Create a new column based off the scaled data
        df = df.withColumn('percentage_scaled_days',
                          round((df['DAYSONMARKET'] - min_days) / (max_days - min_days))

        # Calc max and min for new column
        print(df.agg({'percentage_scaled_days': 'max'}).collect())
        print(df.agg({'percentage_scaled_days': 'min'}).collect())
```

```
[Row(max(percentage_scaled_days)=100.0)]
[Row(min(percentage_scaled_days)=0.0)]
```

Scaling your scalers

In the previous exercise, we minmax scaled a single variable. Suppose you have a LOT of variables to scale, you don't want hundreds of lines to code for each. Let's expand on the previous exercise and make it a function.

In [ ]:
```
ef 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

df = min_max_scaler(df, cols_to_scale)
# Show that our data is now between 0 and 1
df[['DAYSONMARKET', 'scaled_DAYSONMARKET']].show()
```

Creating scalable solutions that can be reused will free up many hours of you and your teams time. Additionally it means that you have fewer things to correct should you need to make changes.

Correcting Right Skew Data

In the slides we showed how you might use log transforms to fix positively skewed data (data whose distribution is mostly to the left). To correct negative skew (data mostly to the right) you need to take an extra step called "reflecting" before you apply the inverse log. To reflect use this formula for each value: (xmax+1)–x.

In [ ]:
```
from pyspark.sql.functions import log

# 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 = df.withColumn('Reflect_YearBuilt', (max_year + 1) - df['YEARBUILT'])

# Create a new column based reflected data
df = df.withColumn('adj_yearbuilt', 1 / log(df['Reflect_YearBuilt']))
```

Adjusting variables is a complex task. What you've seen here are only a few of the ways that you might try to make your data fit a normal distribution.

Visualizing Missing Data

Being able to plot missing values is a great way to quickly understand how much of your data is missing. It can also help highlight when variables are missing in a pattern something that will need to be handled with care lest your model be biased.

Which variable has the most missing values? Run all lines of code except the last one to determine the answer. Once you're confident, and fill out the value and hit "Submit Answer".

In [ ]:
```
# 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=30, fontsize=10)
plt.yticks(rotation=0, fontsize=10)
plt.show()

# Set the answer to the column with the most missing data
answer = 'BACKONMARKETDATE'
```

Quickly visuals like can help you quickly elimination variables that provide no value to your analysis.

Imputing Missing Data

Missing data happens. If we make the assumption that our data is missing completely at random, we are making the assumption that what data we do have, is a good representation of the population. If we have a few values we could remove them or we could use the mean or median as a replacement. In this exercise, we will look at 'PDOM': Days on Market at Current Price.

In [ ]:
```
# 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'])
```

Missing value replacement is easy, however its ramifications can be huge. Make sure to spend time considering the appropriate ways to handle missing data in your problems.

Calculate Missing Percents

Automation is the future of data science. Learning to automate some of your data preparation pays dividends. In this exercise, we will automate dropping columns if they are missing data beyond a specific threshold.

In [ ]:
```python
def column_dropper(df, threshold):
  # 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 = df.where(df[col].isNull()).count()
    missing_percent = missing / total_records
    # Drop column if percent of missing is more than threshold
    if missing_percent > threshold:
      df = df.drop(col)
  return df

# Drop columns that are more than 60% missing
df = column_dropper(df, .6)
```

We just assessed all of our variables for completenes in a few seconds. Additionally, this function is totally reusable for our next analysis.

A Dangerous Join

In this exercise, we will be joining on Latitude and Longitude to bring in another dataset that measures how walk-friendly a neighborhood is. We'll need to be careful to make sure our joining columns are the same data type and ensure we are joining on the same precision (number of digits after the decimal) or our join won't work!

Below you will find that df['latitude'] and df['longitude'] are at a higher precision than walk_df['longitude'] and walk_df['latitude'] we'll need to round them to the same precision so the join will work correctly.

In [ ]:
```python
# Cast data types
walk_df = walk_df.withColumn('longitude', walk_df['longitude'].cast('double'))
walk_df = walk_df.withColumn('latitude', walk_df['latitude'].cast('double'))

# Round percision
df = df.withColumn('longitude', round(df['longitude'], 5))
df = df.withColumn('latitude', round(df['latitude'], 5))

# Create join condition
condition = [(df['longitude'] == walk_df['longitude']), (df['latitude'] == walk_d

# Join the dataframes together
join_df = df.join(walk_df, on=condition, how='left')
# Count non-null records from new field
print(join_df.where(~join_df['walkscore'].isNull()).count())
```

```
4849
```

Great work, taking steps make sure that your join keys are in the same format and precision is important if you hope to get the most out of the new data set!

Spark SQL Join

Sometimes it is much easier to write complex joins in SQL. In this exercise, we will start with the join keys already in the same format and precision but will use SparkSQL to do the joining.

In [ ]:
```
# Register dataframes as tables
df.createOrReplaceTempView("df")
walk_df.createOrReplaceTempView("walk_df")

# SQL to join dataframes
join_sql =  """
            SELECT
                *
            FROM df
            LEFT JOIN walk_df
            ON df.longitude = walk_df.longitude
            AND df.latitude = walk_df.latitude
            """
# Perform sql join
joined_df = spark.sql(join_sql)
```

Awesome, having multiple ways to do the same thing allows you to choose what works for your problem!

Checking for Bad Joins

Joins can go bad silently if we are careful, meaning they will not error out but instead return mangled data. Let's take a look at a couple ways that joining incorrectly can change your data set for the worse.

In [ ]:
```
# Join on mismatched keys precision
wrong_prec_cond = [df_orig['longitude'] == walk_df['longitude'], df_orig['latitud
wrong_prec_df = df_orig.join(walk_df, on=wrong_prec_cond, how='left')

# Compare bad join to the correct one
print(wrong_prec_df.where(wrong_prec_df['walkscore'].isNull()).count())
print(correct_join_df.where(correct_join_df['walkscore'].isNull()).count())

# Create a join on too few keys
few_keys_cond = [df['longitude'] == walk_df['longitude']]
few_keys_df = df.join(walk_df, on=few_keys_cond, how='left')

# Compare bad join to the correct one
print("Record Count of the Too Few Keys Join Example: " + str(few_keys_df.count()
print("Record Count of the Correct Join Example: " + str(correct_join_df.count())
```

4999 151 Record Count of the Too Few Keys Join Example: 6152 Record Count of the Correct Join Example: 5000

As you can see, thinking critically about how you join your data is essential to making sure you don't mangle it!

Feature Engineering

In this chapter learn how to create new features for your machine learning model to learn from. We'll look at generating them by combining fields, extracting values from messy columns or encoding them for better results.

Differences

Let's explore generating features using existing ones. In this example you will create a new feature, and then see if the new feature is correlated with our outcome variable.

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

# Create new column YARD_SIZE
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', 'SALESCLOSEPRIC
# Corr of FOUNDATIONSIZE vs SALESCLOSEPRICE
print("Corr of FOUNDATIONSIZE vs SALESCLOSEPRICE: " + str(df.corr('FOUNDATIONSIZE
# Corr of YARD_SIZE vs SALESCLOSEPRICE
print("Corr of YARD_SIZE vs SALESCLOSEPRICE: " + str(df.corr('YARD_SIZE', 'SALESC
```

Corr of ACRES vs SALESCLOSEPRICE: 0.22060612588935327 Corr of FOUNDATIONSIZE vs SALESCLOSEPRICE: 0.6152231695664401 Corr of YARD_SIZE vs SALESCLOSEPRICE: 0.20714585430854268

Not all generated features are worthwhile, many are not but its still worth doing! Most likely this is because there isn't a lot of variation in lot sizes in the neighborhoods we are looking at to create a strong feature. In addition if we look at our data, some of the homes have 0 ACRES if we really wanted to handle this correctly we could have to set the minimum YARD_SIZE to 0.

Ratios

Ratios are all around us. Whether it's miles per gallon or click through rate, they are everywhere. In this exercise, we'll create some ratios by dividing out pairs of columns.

In [ ]:
```
# ASSESSED_TO_LIST
df = df.withColumn('ASSESSED_TO_LIST', df['ASSESSEDVALUATION'] / df['LISTPRICE'])
df[['ASSESSED_TO_LIST', 'ASSESSEDVALUATION', 'LISTPRICE']].show(5)
# TAX_TO_LIST
df = df.withColumn('TAX_TO_LIST', df['TAXES'] / df['LISTPRICE'])
df[['TAX_TO_LIST', 'TAXES', 'LISTPRICE']].show(5)
# BED_TO_BATHS
df = df.withColumn('BED_TO_BATHS', df['BEDROOMS'] / df['BATHSTOTAL'])
df[['BED_TO_BATHS', 'BEDROOMS', 'BATHSTOTAL']].show(5)
```

```
+------------------+--------+----------+
|      BED_TO_BATHS|BEDROOMS|BATHSTOTAL|
+------------------+--------+----------+
|               1.5|       3|         2|
|1.3333333333333333|       4|         3|
|               2.0|       2|         1|
|               1.0|       2|         2|
|               1.5|       3|         2|
+------------------+--------+----------+
only showing top 5 rows
```

Well done, we've created some great ratios to use in our model that people looking at homes might be considering! Often times rather than just hoping that features will be important and trying them all brute force its more worthwhile to talk to someone that knows the context to get ideas!

Deeper Features

In previous exercises we showed how combining two features together can create good additional features for a predictive model. In this exercise, you will generate 'deeper' features by combining the effects of three variables into one. Then you will check to see if deeper and more complicated features always make for better predictors.

In [ ]:
```
# 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'] / 1
df[['BATHS_PER_1000SQFT']].describe().show()

# Pandas dataframe
pandas_df = df.sample(False, 0.5, 0).toPandas()

# Linear model plots
sns.jointplot(x='Total_SQFT', y='SALESCLOSEPRICE', data=pandas_df, kind="reg", st
plt.show()
sns.jointplot(x='BATHS_PER_1000SQFT', y='SALESCLOSEPRICE', data=pandas_df, kind="
plt.show()
```

```
+-------+------------------+
|summary| BATHS_PER_1000SQFT|
+-------+------------------+
|  count|              5000|
|   mean| 1.4302617483739894|
| stddev|  14.12890410245937|
|    min|0.39123630672926446|
|    max|            1000.0|
+-------+------------------+
```

Using the describe() function you could have seen there was a max of 1000 bathrooms per 1000sqft, which is almost for sure an issue with our data since no sane person would need a bathroom for square foot! If you really wanted to use this feature you'd have to filter that outlier out or overwrite it to NULL with when(). After plotting the jointplots()s you should have seen that the less complicated feature Total_SQFT had a much better R**2 of .67 vs BATHS_PER_1000SQFT's .02'. Often simplier is better!

Time Components

Being able to work with time components for building features is important but you can also use them to explore and understand your data further. In this exercise, you'll be looking to see if there is a pattern to which day of the week a house lists on. Please keep in mind that PySpark's week starts on Sunday, with a value of 1 and ends on Saturday, a value of 7.

```
In [ ]:
```

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

# Convert to date type
df = df.withColumn('LISTDATE', to_date('LISTDATE'))

# 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, 0.5, 42).toPandas()

# Plot count plot of of day of week
ax = sns.countplot(x="List_Day_of_Week", data=sample_df)
plt.show()
```

Fantastic, using these time components and some visualization techniques from earlier we can see its pretty unlikely to list a home on the weekend (Values 1 and 7).

Joining On Time Components

Often times you will use date components to join in other sets of information. However, in this example, we need to use data that would have been available to those considering buying a house. This means we will need to use the previous year's reporting data for our analysis.

In [ ]:
```python
from pyspark.sql.functions import year

# Create year column
df = df.withColumn('list_year', year('LISTDATE'))

# Adjust year to match
df = df.withColumn('report_year', (df['list_year'] - 1))

# Create join condition
condition = [df['CITY'] == price_df['City'], df['report_year'] == price_df['Year'

# Join the dataframes together
df = df.join(price_df, on=condition, how='left')
# Inspect that new columns are available
df[['MedianHomeValue']].show()
```

```
+---------------+
|MedianHomeValue|
+---------------+
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
|         401000|
+---------------+
only showing top 20 rows
```

You can see how easy it is to join data that is reported out at different intervals to use in your data. You also can see how easy it is to use data that would not have been available at the time of someone buying a home; a form of data leakage.

Date Math

In this example, we'll look at verifying the frequency of our data. The Mortgage dataset is supposed to have weekly data but let's make sure by lagging the report date and then taking the difference of the dates.

Recall that to create a lagged feature we will need to create a window(). window() allows you to return a value for each record based off some calculation against a group of records, in this case, the previous period's mortgage rate.

In [ ]:
```python
from pyspark.sql.functions import lag, datediff, to_date
from pyspark.sql.window import Window

# Cast data type
mort_df = mort_df.withColumn('DATE', to_date('DATE'))

# Create window
w = Window().orderBy(mort_df['DATE'])
# Create lag column
mort_df = mort_df.withColumn('DATE-1', lag('DATE', count=1).over(w))

# Calculate difference between date columns
mort_df = mort_df.withColumn('Days_Between_Report', datediff('DATE', 'DATE-1'))
# Print results
mort_df.select('Days_Between_Report').distinct().show()
```

```
+-------------------+
|Days_Between_Report|
+-------------------+
|               null|
|                  7|
|                  6|
|                  8|
+-------------------+
```

we can use this to verify that our mortgage rate data set is consistently reported weekly.

Extracting Text to New Features

Garages are an important consideration for houses in Minnesota where most people own a car and the snow is annoying to clear off a car parked outside. The type of garage is also important, can you get to your car without braving the cold or not? Let's look at creating a feature has_attached_garage that captures whether the garage is attached to the house or not.

In [ ]:
```python
# Import needed functions
from pyspark.sql.functions import when

# Create boolean conditions for string matches
has_attached_garage = df['GARAGEDESCRIPTION'].like('%Attached Garage%')
has_detached_garage = df['GARAGEDESCRIPTION'].like('%Detached Garage%')

# Conditional value assignment
df = df.withColumn('has_attached_garage', (when(has_attached_garage, 1)
                                          .when(has_detached_garage, 0)
                                          .otherwise(None)))

# Inspect results
df[['GARAGEDESCRIPTION', 'has_attached_garage']].show(truncate=100)
```

, by extracting important string values out and condiontionally assigning values we've created an interesting feature to use!

Splitting & Exploding

Being able to take a compound field like GARAGEDESCRIPTION and massaging it into something useful is an involved process. It's helpful to understand early what value you might gain out of expanding it. In this example, we will convert our string to a list-like array, explode it and then inspect the unique values.

In [ ]:
```python
# Import needed functions
from pyspark.sql.functions import split, explode

# Convert string to list-like array
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(100, truncate=50)
```

```
+----------------------------+
|              ex_garage_list|
+----------------------------+
|             Attached Garage|
|      On-Street Parking Only|
|                        None|
| More Parking Onsite for Fee|
|          Garage Door Opener|
|    No Int Access to Dwelling|
|          Driveway - Gravel|
|        Valet Parking for Fee|
|             Uncovered/Open|
|              Heated Garage|
|         Underground Garage|
|                      Other|
|                 Unassigned|
|More Parking Offsite for Fee|
|    Driveway - Other Surface|
|       Contract Pkg Required|
|                     Carport|
|                     Secured|
|             Detached Garage|
|         Driveway - Asphalt|
|                 Units Vary|
|                   Assigned|
|                  Tuckunder|
|                    Covered|
|            Insulated Garage|
|         Driveway - Concrete|
|                     Tandem|
|           Driveway - Shared|
+----------------------------+
```

, looking at the values, it looks like there is a decent amount of values here but not hundreds. If you have too many, when you pivot them it can make your dataset a mess.

Pivot & Join

Being able to explode and pivot a compound field is great, but you are left with a dataframe of only those pivoted values. To really be valuable you'll need to rejoin it to the original dataset! After joining the datasets we will have a lot of NULL values for the newly created columns since we know the context of how they were created we can safely fill them in with zero as either the new has an attribute or it doesn't.

In [ ]:
```python
from pyspark.sql.functions import coalesce, first

# Pivot
piv_df = ex_df.groupBy('NO').pivot('ex_garage_list').agg(coalesce(first('constant

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

# Columns to zero fill
zfill_cols = piv_df.columns

# Zero fill the pivoted values
df = df.fillna(0, subset=zfill_cols)
```

Fantastic, you now have a bunch of boolean columns created from the single compound field. Hopefully some of these will be valuable in our model!

Binarizing Day of Week

In a previous video, we saw that it was very unlikely for a home to list on the weekend. Let's create a new field that says if the house is listed for sale on a weekday or not. In this example there is a field called List_Day_of_Week that has Monday is labeled 1.0 and Sunday is 7.0. Let's convert this to a binary field with weekday being 0 and weekend being 1. We can use the pyspark feature transformer Binarizer to do this.

In [ ]:
```
# Import transformer
from pyspark.ml.feature import Binarizer

# Create the transformer
binarizer = Binarizer(threshold=5.0, inputCol='List_Day_of_Week', outputCol='List

# Apply the transformation to df
df = binarizer.transform(df)

# Verify transformation
df[['List_Day_of_Week', 'Listed_On_Weekend']].show()
```

```
+----------------+-----------------+
|List_Day_of_Week|Listed_On_Weekend|
+----------------+-----------------+
|             6.0|              1.0|
|             1.0|              0.0|
|             1.0|              0.0|
|             5.0|              0.0|
|             2.0|              0.0|
|             1.0|              0.0|
|             4.0|              0.0|
|             7.0|              1.0|
|             4.0|              0.0|
|             6.0|              1.0|
|             5.0|              0.0|
|             4.0|              0.0|
|             7.0|              1.0|
|             1.0|              0.0|
|             4.0|              0.0|
|             7.0|              1.0|
|             7.0|              1.0|
|             5.0|              0.0|
|             6.0|              1.0|
|             5.0|              0.0|
+----------------+-----------------+
only showing top 20 rows
```

transforming features with binarize is helpful in creating more powerful features, in both explainability of your model and performance.

Bucketing

If you are a homeowner its very important if a house has 1, 2, 3 or 4 bedrooms. But like bathrooms, once you hit a certain point you don't really care whether the house has 7 or 8. This example we'll look at how to figure out where are some good value points to bucket.

In [ ]:
```python
from pyspark.ml.feature import Bucketizer

# Plot distribution of sample_df
sns.distplot(sample_df, 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')

# Apply the transformation to df
df = buck.transform(df)

# Display results
df[['BEDROOMS', 'bedrooms']].show()
```

```
+--------+--------+
|BEDROOMS|bedrooms|
+--------+--------+
|     3.0|     3.0|
|     4.0|     4.0|
|     2.0|     2.0|
|     2.0|     2.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     2.0|     2.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
|     3.0|     3.0|
+--------+--------+
only showing top 20 rows
```

Being able to inspect a distribution plot is important if you are considering bucketing feature values together. Here we saw that after 5 bathrooms it was exceedingly rare, so we could combine either effects together in a 5+ value.

One Hot Encoding

In the United States where you live determines which schools your kids can attend. Therefore it's understandable that many people care deeply about which school districts their future home will be in. While the school districts are numbered in SCHOOLDISTRICTNUMBER they are really categorical. Meaning that summing or averaging these values has no apparent meaning. Therefore in this example we will convert SCHOOLDISTRICTNUMBER from a categorial variable into a numeric vector to use in our machine learning model later.

In [ ]:
```
from pyspark.ml.feature import OneHotEncoder, StringIndexer

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

# Onehot encode indexed values
encoder = OneHotEncoder(inputCol='School_Index', outputCol='School_Vec')
encoded_df = encoder.transform(indexed_df)

# Inspect the transformation steps
encoded_df[['SCHOOLDISTRICTNUMBER', 'School_Index', 'School_Vec']].show(truncate=
```

```
+----------------------------+------------+-------------+
|        SCHOOLDISTRICTNUMBER|School_Index|   School_Vec|
+----------------------------+------------+-------------+
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|622 - North St Paul-Maplewood|         1.0|(7,[1],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|622 - North St Paul-Maplewood|         1.0|(7,[1],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
|             834 - Stillwater|         3.0|(7,[3],[1.0])|
+----------------------------+-----------+------------+
only showing top 20 rows
```

Well done! One Hot Encoding is a great way to handle categorial variables. You may have noticed 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.

Building a Model

In this chapter we'll learn how to choose which type of model we want. Then we will learn how to apply our data to the model and evaluate it. Lastly, we'll learn how to interpret the results and save the model for later!

Creating Time Splits

In the video, we learned why splitting data randomly can be dangerous for time series as data from the future can cause overfitting in our model. Often with time series, you acquire new data as it is made available and you will want to retrain your model using the newest data. In the video, we showed how to do a percentage split for test and training sets but suppose you wish to train on all available data except for the last 45days which you want to use for a test set.

In this exercise, we will create a function to find the split date for using the last 45 days of data for testing and the rest for training. Please note that timedelta() has already been imported for you from the standard python library datetime.

In [ ]:
```
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, 'OFFMKTDATE')

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

Creating functions like this take more time upfront but if you intend to use the model over and over again its worth spending more time to do thing properly.

Adjusting Time Features

We have mentioned throughout this course some of the dangers of leaking information to your model during training. 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 this exercise, we are going to ensure that DAYSONMARKET only reflects what information we have at the time of predicting the value. I.e., 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 2017-12-10.

NOTE: This example will use the lit() function. This function is used to allow single values where an entire column is expected in a function call.

In [ ]:
```python
from pyspark.sql.functions import datediff, to_date, lit

split_date = to_date(lit('2017-12-10'))
# Create Sequential Test set
test_df = df.where(df['OFFMKTDATE'] >= split_date).where(df['LISTDATE'] <= split_

# 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
test_df = test_df.withColumn('DAYSONMARKET', datediff(split_date, 'LISTDATE'))

# Review the difference
test_df[['LISTDATE', 'OFFMKTDATE', 'DAYSONMARKET_Original', 'DAYSONMARKET']].show
```

+-------------------+-------------------+---------------------+------------+ | LISTDATE| OFFMKTDATE|DAYSONMARKET_Original|DAYSONMARKET| +-------------------+-------------------+---------------------+------------+ |2017-10-06 00:00:00|2018-01-24 00:00:00| 110| 65| |2017-09-18 00:00:00|2017-12-12 00:00:00| 82| 83| |2017-11-07 00:00:00|2017-12-12 00:00:00| 35| 33| |2017-10-30 00:00:00|2017-12-11 00:00:00| 42| 41| |2017-07-14 00:00:00|2017-12-19 00:00:00| 158| 149| |2017-10-25 00:00:00|2017-12-20 00:00:00| 45| 46| |2017-12-07 00:00:00|2017-12-23 00:00:00| 16| 3| |2017-11-22 00:00:00|2017-12-16 00:00:00| 24| 18| |2017-10-27 00:00:00|2017-12-13 00:00:00| 47| 44| |2017-09-29 00:00:00|2017-12-12 00:00:00| 12| 72| |2017-11-28 00:00:00|2017-12-11 00:00:00| 13| 12| |2017-09-09 00:00:00|2018-01-17 00:00:00| 119| 92| |2017-11-18 00:00:00|2017-12-15 00:00:00| 26| 22| |2017-12-07 00:00:00|2017-12-18 00:00:00| 11| 3| |2017-11-25 00:00:00|2018-01-02 00:00:00| 38| 15| |2017-11-09 00:00:00|2018-01-03 00:00:00| 55| 31| |2017-10-18 00:00:00|2017-12-26 00:00:00| 69| 53| |2017-10-03 00:00:00|2017-12-15 00:00:00| 40| 68| |2017-10-16 00:00:00|2017-12-15 00:00:00| 60| 55| |2017-11-18 00:00:00|2017-12-28 00:00:00| 40| 22| +-------------------+-------------------+---------------------+------------+ only showing top 20 rows Thinking critically about what information would be available at the time of prediction is crucial in having accurate model metrics and saves a lot of embarassment down the road if decisions are being made based off your results!

Feature Engineering For Random Forests

Considering what steps you'll need to take to preprocess your data before running a machine learning algorithm is important or you could get invalid results. Which of the following preprocessing techniques are needed for Random Forest Regression?

Answer the question 50 XP Possible Answers Perform value replacement for missing values and encode categorical text features to numeric. press 1 Scale all features between 0 and 1 with a min max scaler. press 2 Ensure all variables are standard normal distributed, mean 0 and standard deviation of 1. press 3 None of the above. press 4

Answer 1. Missing values are handled by Random Forests internally where they partition on missing values. As long as you replace them with something outside of the range of normal values, they will be handled correctly. Likewise, categorical features only need to be mapped to numbers, they are fine to stay all in one column by using a StringIndexer as we saw in chapter 3. OneHot encoding which converts each possible value to its own boolean feature is not needed.

Dropping Columns with Low Observations

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 course but there are some quick steps you can take to reduce the dimensionality of your data set.

In this exercise, we are going to remove columns that have less than 30 observations. 30 is a common minimum number of observations for statistical significance. Any less than that and the relationships cause overfitting because of a sheer coincidence!

NOTE: The data is available in the dataframe, df.

In [ ]:
```
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)

print('Rows: ' + str(df.count()) + ' Columns: ' + str(len(df.columns)))
print('Rows: ' + str(new_df.count()) + ' Columns: ' + str(len(new_df.columns)))
```

```
Rows: 5000 Columns: 253
Rows: 5000 Columns: 250
```

Removing low observation features is helpful in many ways. It can improve processing speed of model training, prevent overfitting by coincidence and help interpretability by reducing the number of things to consider.

Naively Handling Missing and Categorical Values

Random Forest Regression is robust enough to allow us to ignore many of the more time consuming and tedious data preparation steps. While some implementations of Random Forest handle missing and categorical values automatically, PySpark's does not. The math remains the same however so we can get away with some naive value replacements.

For missing values since our data is strictly positive, we will assign -1. The random forest will split on this value and handle it differently than the rest of the values in the same feature.

For categorical values, we can just map the text values to numbers and again the random forest will appropriately handle them by splitting on them. In this example, we will dust off pipelines from Introduction to PySpark to write our code more concisely. Please note that the exercise will start by displaying the dtypes of the columns in the dataframe, compare them to the results at the end of this exercise.

NOTE: Pipeline and StringIndexer are already imported for you. The list categorical_cols is also available.

In [ ]:
```python
# Replace missing values
df = df.fillna(-1, subset=['WALKSCORE', 'BIKESCORE'])

# Create list of StringIndexers using list comprehension
indexers = [StringIndexer(inputCol=col, outputCol=col+"_IDX")\
            .setHandleInvalid("keep") for col in categorical_cols]
# Create pipeline of indexers
indexer_pipeline = Pipeline(stages=indexers)
# Fit and Transform the pipeline to the original data
df_indexed = indexer_pipeline.fit(df).transform(df)

# Clean up redundant columns
df_indexed = df_indexed.drop(*categorical_cols)
# Inspect data transformations
print(df_indexed.dtypes)
```

```
[('CITY', 'string'), ('LISTTYPE', 'string'), ('SCHOOLDISTRICTNUMBER', 'string'),
('POTENTIALSHORTSALE', 'string'), ('STYLE', 'string'), ('ASSUMABLEMORTGAGE', 'string'),
('ASSESSMENTPENDING', 'string'), ('WALKSCORE', 'double'), ('BIKESCORE', 'double')]
```

Building a Regression Model

One of the great things about PySpark ML module is that most algorithms can be tried and tested without changing much code. Random Forest Regression is a fairly simple ensemble model, using bagging to fit. Another tree based ensemble model is Gradient Boosted Trees which uses a different approach called boosting to fit. In this exercise let's train a GBTRegressor.

In [ ]:
```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.
model = gbt.fit(train_df)
```

Fantastic, as you can see switching from RandomForestRegressor to GBTRegressor was very easy to do. In practice you should try multiple algorithms and evaluate which one fits your data best.

Try use different regression model to do this problem. Include the kernel ridge approach.

Evaluating & Comparing Algorithms

Now that we've created a new model with GBTRegressor its time to compare it against our baseline of RandomForestRegressor. To do this we will compare the predictions of both models to the actual data and calculate RMSE and R^2.

In [ ]:
```python
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':
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))
```

Random Forest Regression RMSE: 22898.84041072095 Random Forest Regression R^2: 0.9666594402208077 Gradient Boosted Trees RMSE: 74380.63652512032 Gradient Boosted Trees R^2: 0.6482244200795505 Be careful in discarding algorithms just because its first pass was not great. Even though Gradient Boosted Trees performed much worse it has many hyper parameters, with proper tuning it would have comparable or better results!

Understanding Metrics

Recall that R^2 and RMSE are both metrics to evaluate the performance of regression models. Both provide a different way to interpret the fit of our model. Which of the following statements is FALSE regarding R^2 or RMSE?

Answer the question 50 XP Possible Answers RMSE is comparable across predictions looking at the same dependent variable. press 1 R^2 is valued between 0 and 100 press 2 R^2 is comparable across predictions regardless of dependent variable. press 3 RMSE is a measure of unexplained variance in the dependent variable. press 4

Answer 2. This is false, R^2 is valued between 0 and 1, where 0 is no better than chance and 1 is a perfect prediction.

Interpreting Results

It is almost always important to know which features are influencing your prediction the most. Perhaps its counterintuitive and that's an insight? Perhaps a hand full of features account for most of the accuracy of your model and you don't need to perform time acquiring or massaging other features.

In this example we will be looking at a model that has been trained without any LISTPRICE information. With that gone, what influences the price the most?

NOTE: The array of feature importances, importances has already been created for you from model.featureImportances.toArray()

In [ ]:
```python
# Convert feature importances to a pandas column
fi_df = pd.DataFrame(importances, columns=['importance'])

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

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

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

We can see that now the features that are the most important are things like the area of the house and taxes both of which are highly correlated with the price of the home.

Saving & Loading Models

Often times you may find yourself going back to a previous model to see what assumptions or settings were used when diagnosing where your prediction errors were coming from. Perhaps there was something wrong with the data? Maybe you need to incorporate a new feature to capture an unusual event that occurred?

In this example, you will practice saving and loading a model.

In [ ]:

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

# Save model
model.save('rfr_no_listprice')

# Load model
loaded_model = RandomForestRegressionModel.load('rfr_no_listprice')
```