Exploratory Data Analysis with PySpark (Spark series part I) APRIL 18, 2019 For those of us with experience in Python or SQL, API wrappers exist to make a Spark workflow look, feel and act like a typical Python workflow or SQL query. The goal of this post is to present an overview of some exploratory data analysis methods for machine learning and other applications in PySpark and Spark SQL.

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

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

Read in some data Here I'll be working through EDA techniques on a dataset of residential homes sold in 2017 in the city of St. Paul, MN. You can download the dataset by clicking or copying this link.

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

We'll download the data using pandas before converting it in to Spark. I do it this way for ease but at the cost of schema - Spark requires more attention to the type of individual columns and how missing values are handled.

Spark isn't quite as versatile as pandas is in inferring data types from the data itself and literally can't handle having more than one data type in a single column. There is a built in method to attempt to infer a schema for the data types when none is provided, which we'll try out after converting all values in the pandas dataframe to strings.

In [1]:

```
import os,numpy,pandas
import pyspark.sql.functions as F
from pyspark.sql.types import *
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,udf,to_timestamp
os.environ['ARROW_PRE_0_15_IPC_FORMAT']='1'
os.environ['OBJC_DISABLE_INITIALIZE_FORK_SAFETY'] = 'YES'


from pyspark.ml.feature import VectorAssembler, StandardScaler

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

In [2]:

```
sqlContext = SparkSession.builder.master('spark://master:7077')\
             .appName('san_pau_bj_practice').getOrCreate()
```

In [3]:

```
HOUSING_DATA = '/user/curtis0982/data/san_paul_housing/san_paul_housing/2017_StP
aul_MN_Real_Estate.csv'
```

In [4]:

```
review_df = spark.read.csv(path =HOUSING_DATA  ,header=True).toPandas()
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')
```

In [5]:

```
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
df = sqlContext.createDataFrame(review_df)
```

In [6]:

```
# 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 [7]:

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

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

Basic Summary Stats

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

In [8]:

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

```
(5000, 73)
```

In [9]:

```
# Describe a column
df[['SalesClosePrice']].describe().show()
```

```
+-------+-----------------+
|summary|   SalesClosePrice|
+-------+-----------------+
|  count|             5000|
|   mean|      262804.4668|
| stddev|140559.82591998542|
|    min|            48000|
|    max|          1700000|
+-------+-----------------+
```

In [10]:

```
# Covariance
df.cov('SalesClosePrice','YEARBUILT')
```

Out[10]:

```
1281910.3840634997
```

In [11]:

```
# Or multiple columns
```

In [12]:

```
# Corr
df.corr('SalesClosePrice','YEARBUILT')
```

Out[12]:

```
0.23475142032506952
```

In [60]:

```
# Perform an aggregation function of SalesClosePrice
print("max is :{}".format(df.agg({'SalesClosePrice':'max'}).collect()[0][0]))
print("min is :{}".format(df.agg({'SalesClosePrice':'min'}).collect()[0][0]))
print("std is :{:.2f}".format(df.agg({'SalesClosePrice':'std'}).collect()[0][0
]))
print("variance is :{:.2f}".format(df.agg({'SalesClosePrice':'variance'}).collec
t()[0][0]))
print("mean is :{}".format(df.agg({'SalesClosePrice':'mean'}).collect()[0][0]))
```

```
max is :1700000
min is :48000
std is :140559.83
variance is :19757064662.66
mean is :262804.4668
```

Visual inspection through linear models and distribution skew

In [14]:

```
# Sample spark df and plot
# It's a best practice to sample data from your Spark df into pandas
##.sample參數withReplacement取後放回,我們不要取後放回, 抽樣比率0.5
#A basic seaborn linear model plot 且可看出 離群值
```

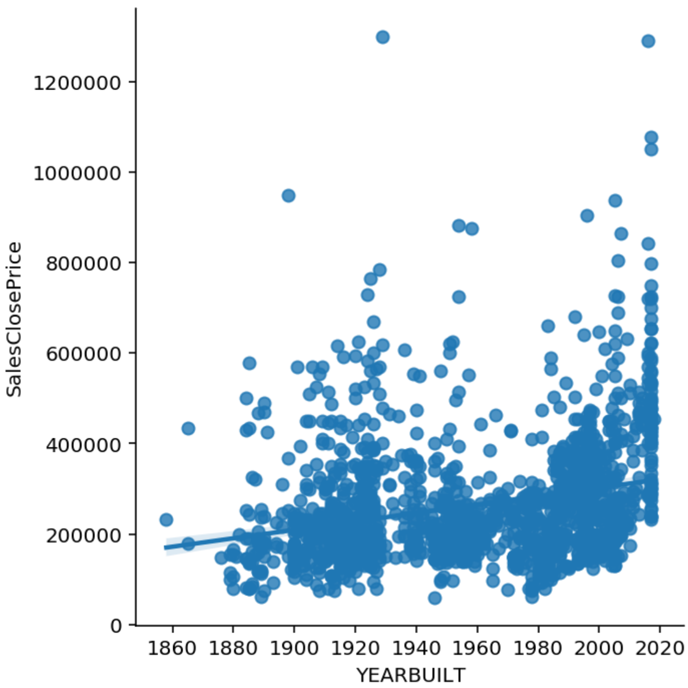
In [15]:

```
pandas_df = df[['SalesClosePrice','YEARBUILT','SQFTABOVEGROUND']].sample(fractio
n = 0.3,\
            withReplacement=False).toPandas()
sns.lmplot(x='YEARBUILT',y='SalesClosePrice',data=pandas_df)
```

```
/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
```

Out[15]:

```
<seaborn.axisgrid.FacetGrid at 0x7fc03f1fa0d0>
```

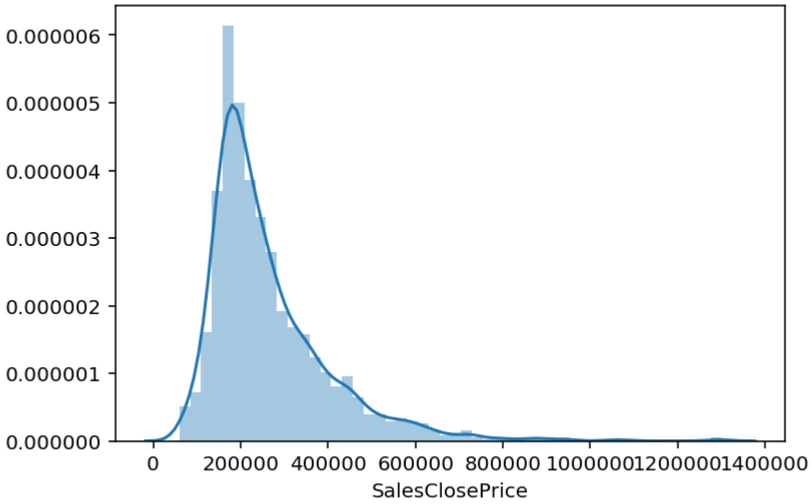


In [16]:

```
# Plot distribution of pandas_df features
sns.distplot(pandas_df['SalesClosePrice'])

#可以看到是一個右偏分配
```

Out[16]:

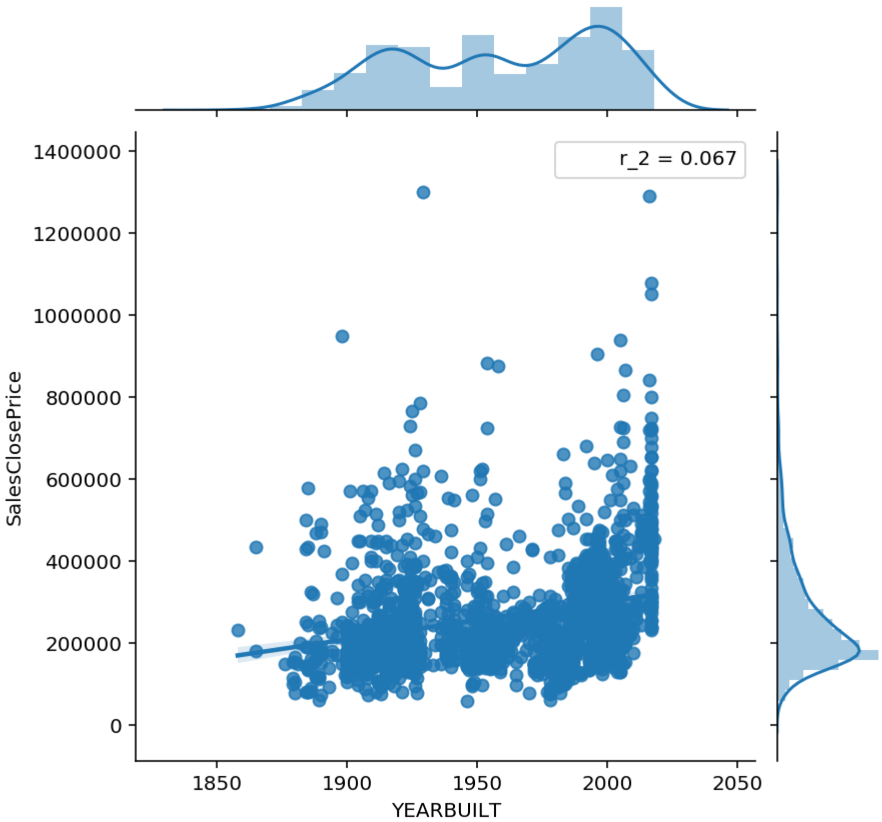
<matplotlib.axes._subplots.AxesSubplot at 0x7fc0394ba510>



In [17]:

```
# Plot distribution of pandas_df and display plot
# 不同圖要兩次plt.show()
from scipy.stats import pearsonr
def r_2(x,y):
    return pearsonr(x,y)[0]**2
sns.jointplot(x='YEARBUILT', y='SalesClosePrice', data=pandas_df, stat_func=r_2,\
              kind='reg')
plt.show()
sns.jointplot(x='SQFTABOVEGROUND', y='SalesClosePrice', data=pandas_df, stat_func=r_2,\
              kind='reg')
plt.show()
```

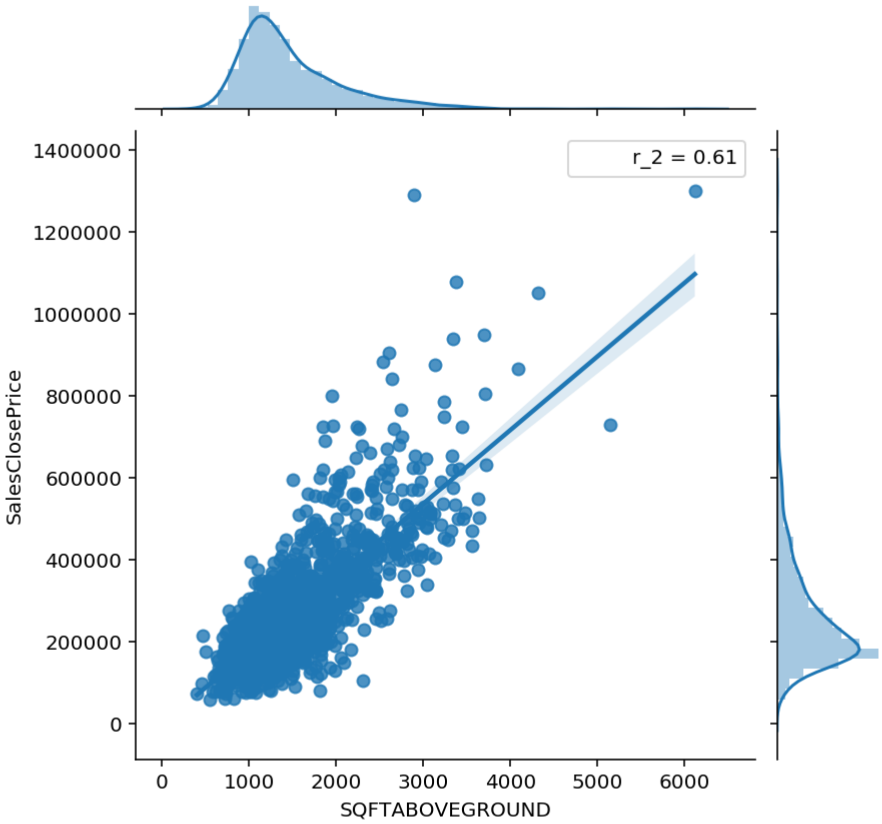
```
/home/curtis0982/anaconda3/lib/python3.7/site-packages/seaborn/axisg
rid.py:1848: UserWarning: JointGrid annotation is deprecated and wil
l be removed in a future release.
  warnings.warn(UserWarning(msg))
```




```
/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))
```

Out[17]:

```
<seaborn.axisgrid.JointGrid at 0x7fc0381c1e90>
```



corr 'SalesClosePrice', 'LivingArea' 'SalesClosePrice', 'SQFTABOVEGROUND'

Filtering based on values or text

Often times, it’s necessary to reduce the scope of an analysis.
1.Filtering out outliers.
2.You might also need to filter out specific categories for features that are categorical in nature.

In [18]:

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

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

In [19]:

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

```
+---------------+
|SalesClosePrice|
+---------------+
|         143000|
|         190000|
|         225000|
|         265000|
|         249900|
|         255000|
|         248000|
|         245000|
|         254990|
|         250000|
+---------------+
only showing top 10 rows
```

In [20]:

```
# implementation of WHERE, while .filter() is provided for the Scala familiar
```

In [21]:

```
# What if you really don't like metal roofs and have a huge budget?
# There are some missing values here, we'll show how to deal with those below
##df['ROOF']!='metal'不要寫成~df['ROOF']=='metal
##兩個條件分別都要用括弧包
df.where((df['ROOF']!='metal') & (df['SalesClosePrice'] >1000000))[['SalesCloseP
rice','ROOF']].show(10)
```

```
+---------------+--------------------+
|SalesClosePrice|                ROOF|
+---------------+--------------------+
|        1277023|                None|
|        1050000|                None|
|        1290000|    Asphalt Shingles|
|        1295000|Asphalt Shingles,...|
|        1215000|    Asphalt Shingles|
|        1380000|                None|
|        1300000|    Asphalt Shingles|
|        1400000|Age Over 8 Years,...|
|        1595000|Asphalt Shingles,...|
|        1600000|              Shakes|
+---------------+--------------------+
only showing top 10 rows
```

In [22]:

```
# More value filtering, this time with aggregation functions
#刪除LISTPRICE >< 3 std
mean_p = df.agg({'LISTPRICE':'mean'}).collect()[0][0]
std_p = df.agg({'LISTPRICE':'std'}).collect()[0][0]
df.where((df['LISTPRICE'] > (mean_p - 3*std_p)) & (df['LISTPRICE'] < (mean_p + 3
*std_p)))[['LISTPRICE','SalesClosePrice']].show(10)
```

```
+---------+---------------+
|LISTPRICE|SalesClosePrice|
+---------+---------------+
|   139900|         143000|
|   210000|         190000|
|   225000|         225000|
|   230000|         265000|
|   239900|         249900|
|   239900|         255000|
|   265000|         248000|
|   273417|         245000|
|   273152|         254990|
|   273482|         250000|
+---------+---------------+
only showing top 10 rows
```

In [63]:

```
# Filter based on text
print(df.select(['ASSUMABLEMORTGAGE']).distinct().show())

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

# 排除值為list的資料
# (good use of that ~ here)
text_filter = ~df['ASSUMABLEMORTGAGE'].isin(yes_values) | \
               df['ASSUMABLEMORTGAGE'].isNull()

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

```
+-------------------+
|  ASSUMABLEMORTGAGE|
+-------------------+
|  Yes w/ Qualifying|
|               None|
| Information Coming|
|Yes w/No Qualifying|
|      Not Assumable|
+-------------------+

None
```

Out[63]:

```
4976
```

In [80]:

```
df.where(~df['ASSUMABLEMORTGAGE'].isNull())[['ASSUMABLEMORTGAGE']].show(10)
```

```
+-----------------+
|ASSUMABLEMORTGAGE|
+-----------------+
|             None|
|             None|
|    Not Assumable|
|             None|
|             None|
|    Not Assumable|
|             None|
|             None|
|             None|
|             None|
+-----------------+
only showing top 10 rows
```

In [62]:

```
# Inspect unique values in the column 'ASSUMABLEMORTGAGE'
# List of possible values containing 'yes'
print('count:',df.agg(F.countDistinct('ASSUMABLEMORTGAGE')).collect()[0][0])
print('count:',df[['ASSUMABLEMORTGAGE']].distinct().show(20))
print('count:',df[['ASSUMABLEMORTGAGE']].distinct().show(20))
# Filter the text values out of df but keep null values
# (good use of that ~ here)
```

```
count: 5
+-------------------+
|  ASSUMABLEMORTGAGE|
+-------------------+
|  Yes w/ Qualifying|
|               None|
| Information Coming|
|Yes w/No Qualifying|
|      Not Assumable|
+-------------------+

count: None
+-------------------+
|  ASSUMABLEMORTGAGE|
+-------------------+
|  Yes w/ Qualifying|
|               None|
| Information Coming|
|Yes w/No Qualifying|
|      Not Assumable|
+-------------------+

count: None
```

In [24]:

```
# Text filtering - where the cooling description is NOT (via the ~)
# central air
```

Dropping NA values or dropping columns outright

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

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

In [25]:

```
# Remove rows with any NA values - naive approach
```

In [26]:

```
# Remove a record if it has NA values in three columns
```

In [27]:

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

Transforming or Adjusting Data

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

In [ ]:

```

```

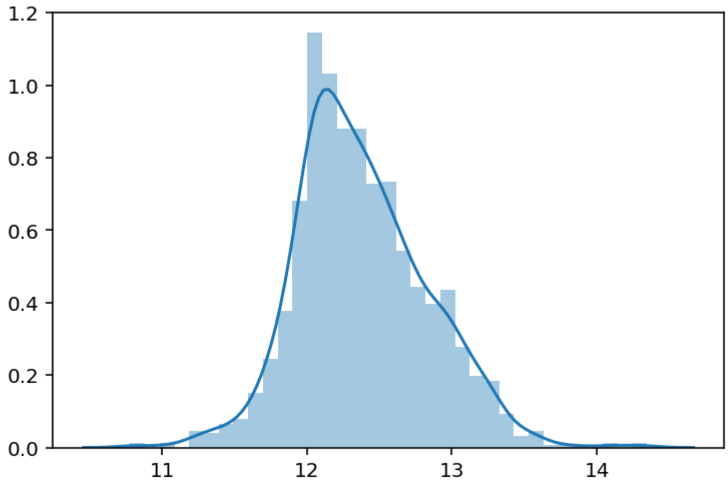
In [28]:

```
#log
df = df.withColumn("SalesClosePrice_log", F.log1p(df["SalesClosePrice"]))
sns.distplot(df[['SalesClosePrice_log']].sample(fraction=0.3,withReplacement=False).toPandas())
#可以看到其實不符合常態分配 考慮使用box cox transform
#更不符合常態分配 刪除
df.drop('SalesClosePrice_log')
```

```
/home/curtis0982/anaconda3/lib/python3.7/site-packages/pyarrow/pandas_compat.py:752: FutureWarning: .labels was deprecated in version 0.24.0. Use .codes instead.
  labels, = index.labels
```

Out[28]:

```
DataFrame[MLSID: string, StreetNumberNumeric: string, 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, Fireplaces: bigint, RoomArea4: string, roomtype: string, ROOF: string, RoomFloor4: string, PotentialShortSale: string, PoolDescription: string, PDOM: bigint, 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: double, 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]
```



In [29]:

```
#檢查skewness
print('count:',df.agg(F.skewness('SalesClosePrice')).collect()[0][0])

#from scipy.stats import norm, skew
```

```
count: 2.623630865263645
```

In [70]:

```
#scipy不能call 用自己寫的
#這個不能用
from scipy.special import boxcox1p
def boxcox1p_sp(x):
    return boxcox1p(x,0)
```

In [71]:

```
def f(s,lambda_box):
    #print(type(s))
    #print(type(alpha))
    if lambda_box == 0:
        return log(s)
    elif lambda_box != 0:
        return ((s+1) ** lambda_box - 1) / lambda_box
#dataset.withColumn(out_col, udf(f, FloatType())(in_col))
```

In [72]:

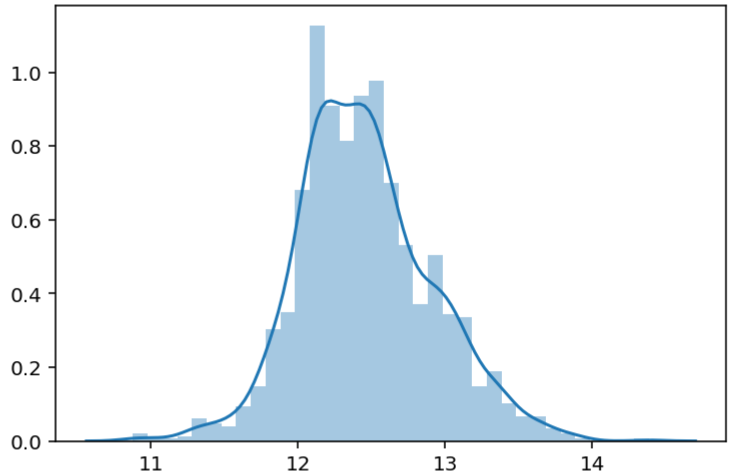
```
#這裡要用lit lit轉成col形式
#使用1.udf參數 2.df.select(col(RECNO),lit(2))

#udf用法
#1.定義函數f
#2.udf(f, 回傳資料型態FloatType())(參數)
#3.沒有spark內部api可以達到功能時才使用

df = df.withColumn("SalesClosePrice_box", udf(f, FloatType())(df['SalesClosePric
e'],F.lit(0.001)))
df[['SalesClosePrice_box']].show(10)
sns.distplot(df[['SalesClosePrice_box']].sample(fraction=0.3,withReplacement=Fal
se).toPandas())
#sns.distplot(df[['SalesClosePrice_box']].toPandas())
```

```
+-------------------+
|SalesClosePrice_box|
+-------------------+
|          11.941342|
|          12.228954|
|          12.400112|
|         12.5657835|
|          12.506379|
|         12.5268345|
|         12.4986515|
|          12.486329|
|          12.526794|
|          12.506784|
+-------------------+
only showing top 10 rows
```

```
/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
```

Out[72]:

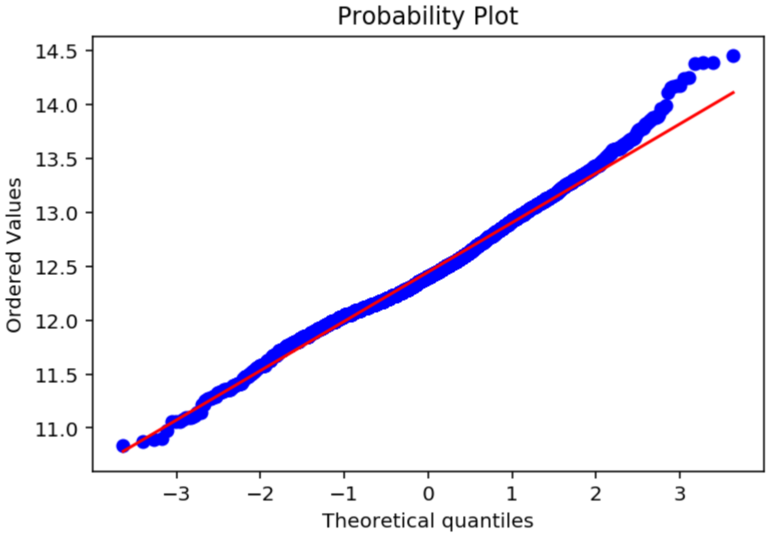
<matplotlib.axes._subplots.AxesSubplot at 0x7fc032cdc910>



In [73]:

```
import scipy.stats as stats
fig = plt.figure()
res = stats.probplot(df[['SalesClosePrice_box']].toPandas()['SalesClosePrice_box'], plot=plt)
plt.show()
```

```
/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
```



Missing Values

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

Imputation

In [66]:

```
# Replacing with the mean value for that column PDOM
PDOM_mean = df.agg({'PDOM':'mean'}).collect()[0][0]
df.fillna(PDOM_mean,subset=['PDOM'])[['PDOM']].show(10)
```

```
+----+
|PDOM|
+----+
|  10|
|   4|
|  28|
|  19|
|  21|
|  17|
|  19|
|   1|
|   5|
|  11|
+----+
only showing top 10 rows
```

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

In [85]:

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

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

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

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

# Drop columns that are more than 60% missing
##column數變成68
len(column_dropper(df_no_dates, 0.60).columns)
```

Out[85]:

```
70
```

In [74]:

```
# Remove a record if it has NA values in three columns
df.dropna(thresh=5).count() # we don't have any missing values aside from one
                            # column, which is nice
                            #thresh=5表示有值得欄位少於5欄就會刪除
```

Out[74]:

```
5000
```

In [84]:

```
print("PostalCode is not null:",df.where(~df['PostalCode'].isNull()).count())
print("PostalCode is null:",df.where(df['PostalCode'].isNull()).count())
```

```
PostalCode is not null: 5000
PostalCode is null: 0
```

In [75]:

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

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

Joining data in Spark with PySpark or with SparkSQL

LivingArea_sqm = df.withColumn('LivingArea_sqm', df['LivingArea'] / 10.764)\ [['streetaddress', 'LivingArea_sqm']]

In [86]:

```
# Joining with PySpark

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

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

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

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

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

```
+--------------------+----------+------------------+---------------+
|       streetaddress|LivingArea|    LivingArea_sqm|SalesClosePrice|
+--------------------+----------+------------------+---------------+
|      1107 Jenks Ave|      1088|101.07766629505761|         172500|
|1181 Edgcumbe Rd,...|       720| 66.88963210702342|          62000|
|      1485 Blair Ave|      1932| 179.4871794871795|         241350|
|       1679 Lark Ave|      1610| 149.5726495726496|         180000|
|  2014 Worcester Ave|      1638|152.17391304347828|         295000|
|     2338 Bourne Ave|      2352|218.50613154960982|         480000|
|2371 Mailand Ct E...|      1381|128.29803047194352|         151000|
|2439 Springside Dr E|      3142|291.89892233370495|         440000|
|   26 10th St W, 410|      1073| 99.68413229282795|         156000|
|2645 New Century ...|      1598|148.45782237086587|         215000|
+--------------------+----------+------------------+---------------+
only showing top 10 rows
```

In [99]:

```
df.registerTempTable("df")
LivingArea_sqm.registerTempTable("LivingArea_sqm")
joint_df = sqlContext.sql(
            """select df.streetaddress, df.LivingArea,df.SalesClosePrice,
                      LivingArea_sqm.LivingArea_sqm
            from df
            left join LivingArea_sqm ON LivingArea_sqm.streetaddress=df.streetadd
ress""")
joint_df.show(10)
```

```
+--------------------+----------+--------------+------------------+
|       streetaddress|LivingArea|SalesClosePrice|    LivingArea_sqm|
+--------------------+----------+--------------+------------------+
|      1107 Jenks Ave|      1088|        172500|101.07766629505761|
|1181 Edgcumbe Rd,...|       720|         62000| 66.88963210702342|
|      1485 Blair Ave|      1932|        241350| 179.4871794871795|
|       1679 Lark Ave|      1610|        180000| 149.5726495726496|
|  2014 Worcester Ave|      1638|        295000|152.17391304347828|
|     2338 Bourne Ave|      2352|        480000|218.50613154960982|
|2371 Mailand Ct E...|      1381|        151000|128.29803047194352|
|2439 Springside Dr E|      3142|        440000|291.89892233370495|
|   26 10th St W, 410|      1073|        156000| 99.68413229282795|
|2645 New Century ...|      1598|        215000|148.45782237086587|
+--------------------+----------+--------------+------------------+
only showing top 10 rows
```

In [98]:

```
df.registerTempTable("df")
LivingArea_sqm.registerTempTable("LivingArea_sqm")
joint_df = sqlContext.sql(
            """select df.`streetaddress` from df""")
joint_df.show(10)
```

```
+--------------------+
|       streetaddress|
+--------------------+
|11511 Stillwater ...|
|     11200 31st St N|
|8583 Stillwater B...|
|      9350 31st St N|
|   2915 Inwood Ave N|
|   3604 Layton Ave N|
|9957 5th Street Ln N|
|9934 5th Street Ln N|
|9926 5th Street Ln N|
|9928 5th Street Ln N|
+--------------------+
only showing top 10 rows
```

Joining data with SparkSQL

by registertemptable

In [91]:

```
# Joining data with SparkSQL
# Register dataframes as tables
df.createOrReplaceTempView('df')
#registerTempTable也可以
#df.registerTempTable('df')
LivingArea_sqm.createOrReplaceTempView('LivingArea_sqm')

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

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

joined_df.show(10)
```

```
+--------------------+----------+------------------+
|       streetaddress|LivingArea|    LivingArea_sqm|
+--------------------+----------+------------------+
|      1107 Jenks Ave|      1088|101.07766629505761|
|1181 Edgcumbe Rd,...|       720| 66.88963210702342|
|      1485 Blair Ave|      1932| 179.4871794871795|
|       1679 Lark Ave|      1610| 149.5726495726496|
|  2014 Worcester Ave|      1638|152.17391304347828|
|     2338 Bourne Ave|      2352|218.50613154960982|
|2371 Mailand Ct E...|      1381|128.29803047194352|
|2439 Springside Dr E|      3142|291.89892233370495|
|   26 10th St W, 410|      1073| 99.68413229282795|
|2645 New Century ...|      1598|148.45782237086587|
+--------------------+----------+------------------+
only showing top 10 rows
```

In [39]:

```
# example data
df_pd = pandas.DataFrame(
    data={'integers': [1, 2, 3],
     'floats': [1.0, 0.5, 2.7],
     'integer_arrays': [[1, 2], [3, 4, 5], [6, 7, 8, 9]]}
)
df_2 = spark.createDataFrame(df_pd)
df_2.printSchema()
```

```
root
 |-- integers: long (nullable = true)
 |-- floats: double (nullable = true)
 |-- integer_arrays: array (nullable = true)
 |    |-- element: long (containsNull = true)
```

In [40]:

```
df_2.show()
def square(x):
    return x**2
```

```
+--------+------+-------------+
|integers|floats|integer_arrays|
+--------+------+-------------+
|       1|   1.0|       [1, 2]|
|       2|   0.5|    [3, 4, 5]|
|       3|   2.7| [6, 7, 8, 9]|
+--------+------+-------------+
```

In [41]:

```
df_2.columns
```

Out[41]:

```
['integers', 'floats', 'integer_arrays']
```

In [42]:

```
#不work
square_udf_int = udf(lambda x: boxcox1p(x,0), FloatType())
```

In [43]:

```
def f(s):
    #print(type(s))
    #print(type(alpha))
    #if alpha == 0:
    #    return log(s)
    #elif alpha > 0:
    return (s * 0.15 - 1) / 0.15
#dataset.withColumn(out_col, udf(f, FloatType())(in_col))
```

In [55]:

```
(
    df_2.select('integers',
          'floats',
          udf(f, FloatType())('integers'))
          #udf(f(df_2['integers']), FloatType()))  #not working
          #udf(boxcox1p, FloatType())(df_2['integers'],F.lit(1))) #not working
    .show()
)
```

```
+--------+------+-----------+
|integers|floats|f(integers)|
+--------+------+-----------+
|       1|   1.0| -5.6666665|
|       2|   0.5| -4.6666665|
|       3|   2.7| -3.6666667|
+--------+------+-----------+
```

In [ ]:

```

```

In [ ]:

```

```