# Predicting Iowa House Prices (Tree Algorithms)

December 20, 2020

### 0.1 Predicting Iowa House Prices with Random Forest and Decision Trees

This project predicted house prices in Ames, Iowa with 79 features (2006-2010). The training set had 1460 observations and the test set had 1459 observations.

I engineered features for tree models, and trained random forest and decision trees in this notebook

**Feature Engineering:**

- Outliers
- Skewness
- Missing Values
- Categorical variables (Label Encoding)

**Models:**

- Random Forest
- Decision Tree
- All with PCA

```
In [46]: #import libraries
         import pandas as pd
         import matplotlib.pyplot as plt
         import seaborn as sns
         import numpy as np
         from scipy.stats import norm, skew
         from scipy import stats

         #import libraries--feature engineering
         from sklearn.preprocessing import LabelEncoder
         from scipy.special import boxcox1p
         from sklearn.preprocessing import StandardScaler
         from sklearn.decomposition import PCA

         #import libraries--modeling
         from sklearn.linear_model import LinearRegression, ElasticNet, Lasso, Ridge
         from sklearn.kernel_ridge import KernelRidge
         from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import KFold, cross_val_score, train_test_split, GridSea
from sklearn.metrics import mean_squared_error
from sklearn.neighbors import KNeighborsClassifier
import os
import warnings
warnings.filterwarnings('ignore')
def ignore_warn(*args, **kwargs):
    pass
warnings.warn = ignore_warn #ignore warnings from sklearn and seaborn

#setup graphs
color = sns.color_palette()
sns.set_style('darkgrid')
%matplotlib inline
pd.set_option('display.float_format', lambda x: '{:.3f}'.format(x)) #Limiting floats
```

In [47]:
```
os.getcwd()
```

Out[47]: '/Users/qingchuanlyu/Documents/Application/Projects/Iowa Housing'

In [48]:
```
#import datasets
train = pd.read_csv('/Users/qingchuanlyu/Documents/Application/Projects/Iowa Housing/
test = pd.read_csv('/Users/qingchuanlyu/Documents/Application/Projects/Iowa Housing/h
train.shape, test.shape
```

Out[48]: ((1460, 81), (1459, 80))

### Feature Engineering

In [49]:
```
#fix outlier: YrSold is earlier than YrBuilt for the observation 1089 in the test dat
test.loc[1089]["YrSold"] = 2009
test.loc[1089]["YrActualAge"] = 0
```

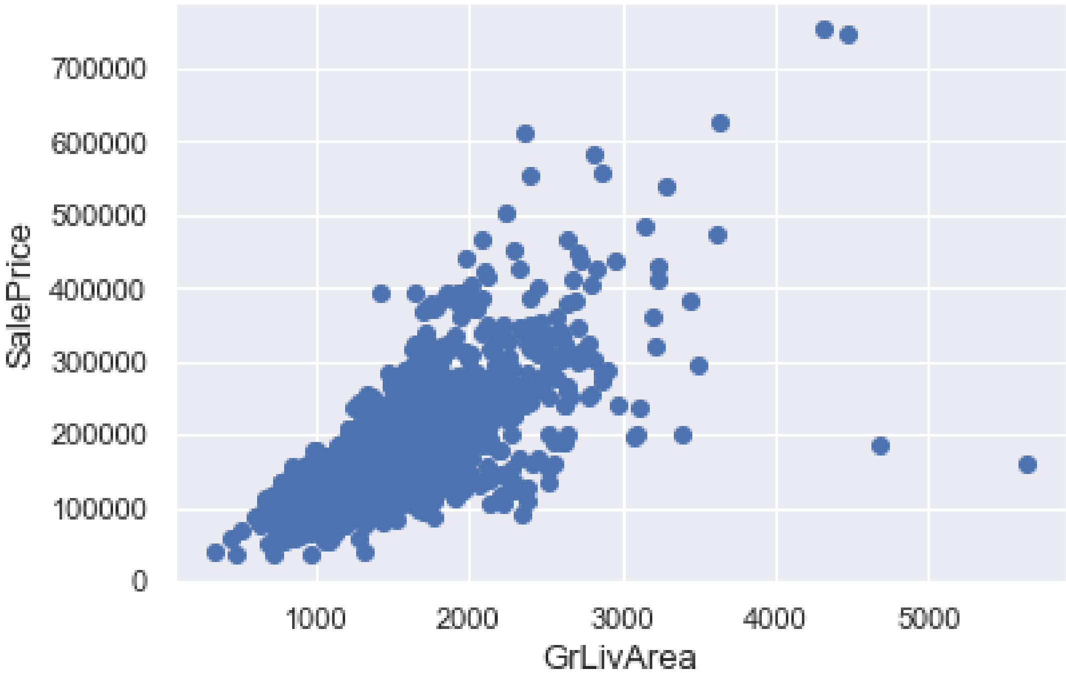
In [50]:
```
#store the 'Id' column then drop it from original datasets--not used in modeling
#axis = 1 indicates col
train_ID = train['Id']
test_ID = test['Id']
train.drop("Id", axis = 1, inplace = True)
test.drop("Id", axis = 1, inplace = True)
```

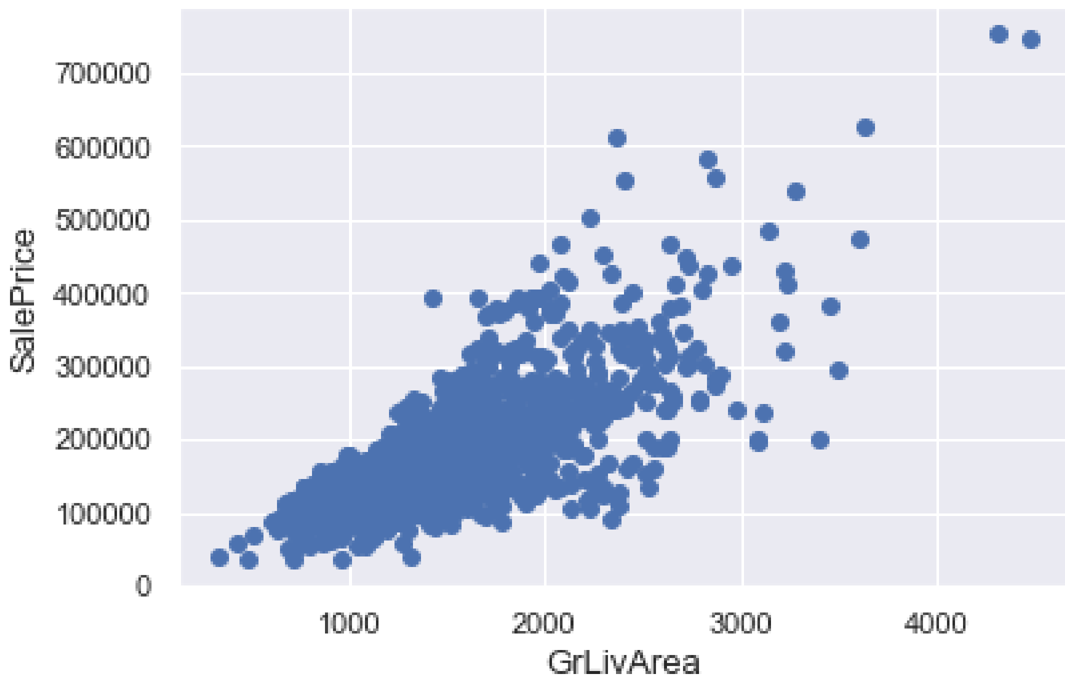
In [51]:
```
###Outliers
#use a scatter plot to observation the relationshiop between living areas and prices
fig, ax = plt.subplots()
ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
```

```
plt.show()

#Delete outliers in the bottom-right corner of the scatter plot
train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].inde

#Check distribution again
fig, ax = plt.subplots()
ax.scatter(train['GrLivArea'], train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```






In [52]:
```
#Target variable
#Check the distribution of target variable: saleprice
sns.distplot(train['SalePrice'] , fit=norm);

#Get the fitted parameters used by the function
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

#Plot the distribution of salesprice
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')
```

```
 mu = 180932.92 and sigma = 79467.79
```

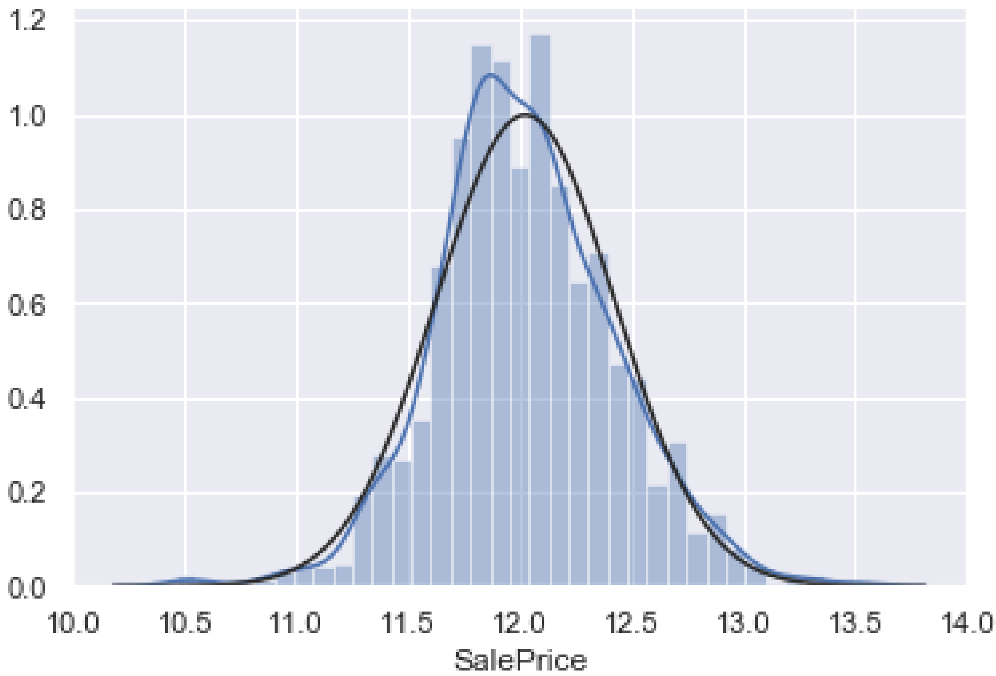
Out[52]:
```
Text(0.5, 1.0, 'SalePrice distribution')
```


In [53]:
```
# Label is a little right-skewed. Use log transformation to make it more normally dis
#use the numpy fuction log1p to apply log(1+x): plus 1 to avoid -inf
train["SalePrice"] = np.log1p(train["SalePrice"])

#Check the new distribution
sns.distplot(train['SalePrice'] , fit=norm);

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
```

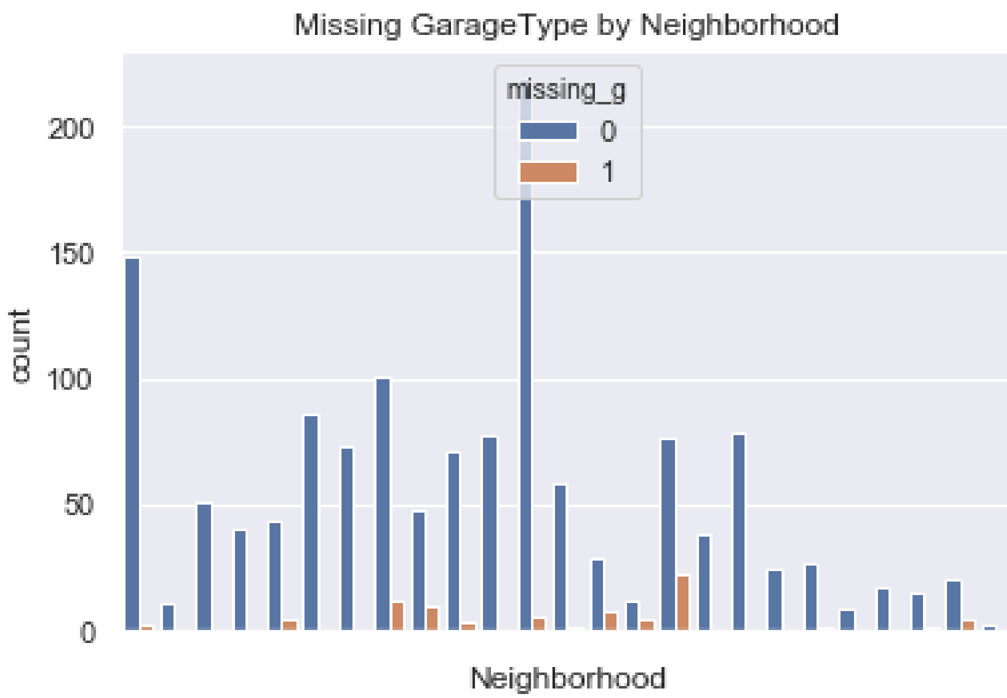
```
 mu = 12.02 and sigma = 0.40
```




In [54]:
```
#predictors in the training set
y_train = train.SalePrice.values
```

In [55]:
```
#check if missing values are Missing at Random, Missing completely at Random, or Not
#Assign label "None" to missing values:
for col in ('MSSubClass', 'MasVnrType', 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'Bsmt
    train[col] = train[col].fillna('None')
#check if GarageType is missing completely at random
train['missing_g'] = np.where(train['GarageType']=="None", 1, 0)
#Check distribution of counts of each garage type with seaborn
ax = sns.countplot(x ='Neighborhood', hue = 'missing_g', data = train)
ax.set(xticklabels=[])
ax.set(title='Missing GarageType by Neighborhood')
#missing values of garage types spread across a few neighborhoods, so keep "None"
```

Out[55]: [Text(0.5, 1.0, 'Missing GarageType by Neighborhood')]

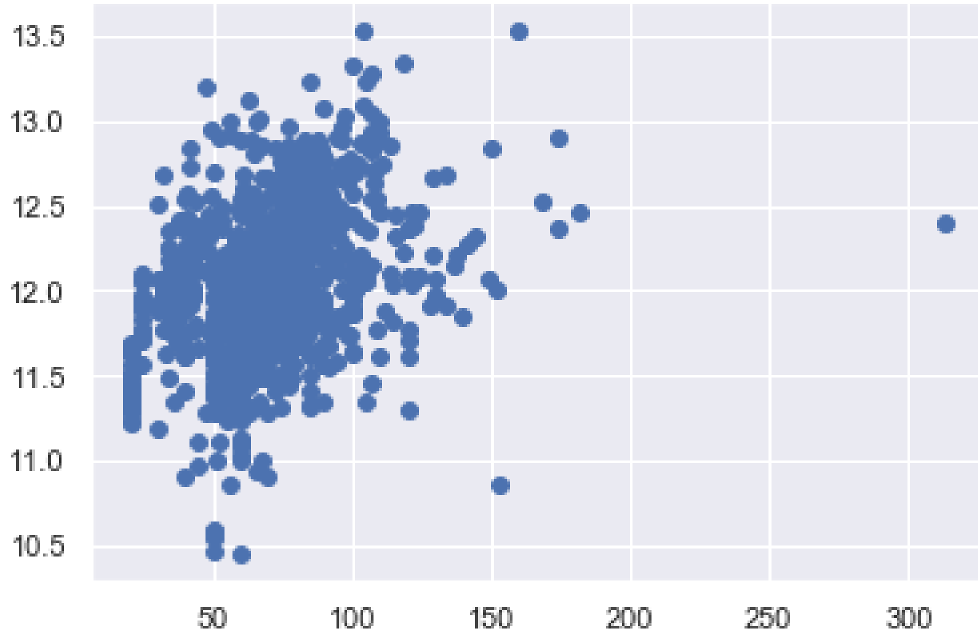



```
In [56]: #check the number of missing values among all variables
         train.isnull().sum()
```

```
Out[56]: MSSubClass          0
         MSZoning            0
         LotFrontage       259
         LotArea             0
         Street              0
                          ...
         YrSold              0
         SaleType            0
         SaleCondition       0
         SalePrice           0
         missing_g           0
         Length: 81, dtype: int64
```

```
In [57]: plt.scatter(train['LotFrontage'], train['SalePrice'])
```

```
Out[57]: <matplotlib.collections.PathCollection at 0x1a19737048>
```




```
In [58]: #saleprice increases with lotfrontage
         #next, check if missing lotfrontage concentrate in larger or samller values
         train.loc[train.LotFrontage.isnull() == True][['SalePrice']].mean()
```

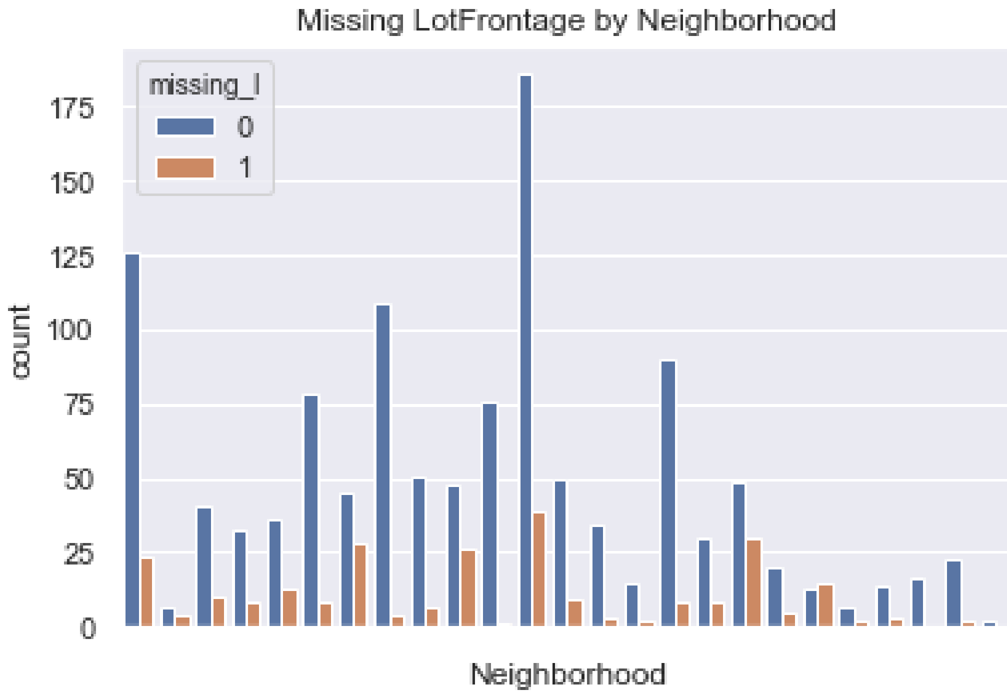
```
Out[58]: SalePrice   12.062
         dtype: float64
```

```
In [59]: train.loc[train.LotFrontage.isnull() == False][['SalePrice']].mean()
```

```
Out[59]: SalePrice   12.016
         dtype: float64
```

```
In [60]: #check if missing LotFrontage concentrate in a few neighborhoods
         train['missing_l'] = np.where(train['LotFrontage'].isnull(), 1, 0)
         ax = sns.countplot(x = 'Neighborhood', hue = 'missing_l', data = train)
         ax.set(xticklabels=[])
         ax.set(title='Missing LotFrontage by Neighborhood')
         #missing LotFrontage spreads out across different neighborhoods
```

```
Out[60]: [Text(0.5, 1.0, 'Missing LotFrontage by Neighborhood')]
```




```
In [61]: #check if mean and median of lotfrontage is different across neighborhoods
         lotfrontage_by_ngh = train.groupby(['Neighborhood']).\
                                   agg(mean_lotfrontage = ('LotFrontage', 'mean'), \
                                       med_lotfrontage=('LotFrontage', 'median')).\
                                   reset_index()
```

```
In [62]: lotfrontage_by_ngh.head(15)
```

Out[62]:

| | Neighborhood | mean_lotfrontage | med_lotfrontage |
|---|---|---|---|
| 0 | Blmngtn | 47.143 | 43.000 |
| 1 | Blueste | 24.000 | 24.000 |
| 2 | BrDale | 21.562 | 21.000 |
| 3 | BrkSide | 57.510 | 52.000 |
| 4 | ClearCr | 83.462 | 80.000 |
| 5 | CollgCr | 71.683 | 70.000 |
| 6 | Crawfor | 71.805 | 74.000 |
| 7 | Edwards | 64.811 | 64.500 |
| 8 | Gilbert | 79.878 | 65.000 |
| 9 | IDOTRR | 62.500 | 60.000 |
| 10 | MeadowV | 27.800 | 21.000 |
| 11 | Mitchel | 70.083 | 73.000 |
| 12 | NAmes | 76.462 | 73.000 |
| 13 | NPkVill | 32.286 | 24.000 |
| 14 | NWAmes | 81.289 | 80.000 |

In [63]:
```
#Group by neighborhood and fill in missing value by the median LotFrontage of all the
#median, mean functions are not affected by missing values. first, obtain the median
nbh_lot = train.groupby(train.Neighborhood)[['LotFrontage']].median()
#med_lot = neigh_lot.groupby("Neighborhood")["LotFrontage"].transform("median")
#all["LotFrontage"] = all["LotFrontage"].fillna(med_lot)
#all.loc[all.Neighborhood.isin(neigh_lot.Neighborhood), ['LotFrontage']] = neigh_lot[
train = train.merge(nbh_lot, on=["Neighborhood"], how='left', suffixes=('','_'))
train['LotFrontage'] = train['LotFrontage'].fillna(train['LotFrontage_']).astype(int)
train = train.drop('LotFrontage_', axis=1)
```

In [64]:
```
#GarageYrBlt, GarageArea and GarageCars : Replacing missing data with 0 (Since No gar
#BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath and BsmtHalfBath : miss
for col in ('MasVnrArea', 'GarageYrBlt', 'GarageArea', 'GarageCars', 'BsmtFinSF1', 'B
    train[col] = train[col].fillna(0)
#Remove "Utilities"-- For this categorical feature all records are "AllPub", except f
train = train.drop(['Utilities'], axis=1)
#Functional : data description says NA means typical
train["Functional"] = train["Functional"].fillna("Typical")
#vars with only one NA value, use mode of this var in the training set to prevent dat
for col in ('KitchenQual', 'Electrical', 'Exterior1st', 'Exterior2nd', 'MSZoning', 'S
    train[col] = train[col].fillna(train[col].mode()[0])
```

In [65]:
```
#Changing OverallCond into a categorical variable
train['OverallCond'] = train['OverallCond'].apply(str)

#Year and month sold are transformed into categorical features.
train['YrSold'] = train['YrSold'].astype(str)
train['MoSold'] = train['MoSold'].astype(str)
```

In [66]:
```
# extract categorical variables
cates = train.select_dtypes(include=['object', 'category']).columns
# process columns, apply LabelEncoder to categorical features
for c in cates:
    lbl = LabelEncoder()
    lbl.fit(list(train[c].values))
    train[c] = lbl.transform(list(train[c].values))
```

In [67]:
```
#add a feature: total areas
train['TotalSF'] = train['TotalBsmtSF'] + train['1stFlrSF'] + train['2ndFlrSF']

###clean Skewed features
#extract numerical features
num_vars = train.dtypes[train.dtypes != "object"].index

# Check the skew of all numerical features
skewed_vars = train[num_vars].apply(lambda x: skew(x.dropna())).sort_values(ascending
skewness = pd.DataFrame({'Skew' :skewed_vars})
skewness.head(15)
```

```
# apply Box Cox Transformation to (highly) skewed features
skewness = skewness[abs(skewness) > 0.8]
skewed_features = skewness.index
lam = 0.25
for f in skewed_features:
    train[f] = boxcox1p(train[f], lam)
```

### Modeling – Random Forest and Decision Trees

```
In [68]: metric = 'neg_mean_squared_error'
         kfold = KFold(n_splits=10, shuffle=True, random_state=1)
```

```
In [69]: #random forest
         rdf = make_pipeline(RobustScaler(), RandomForestRegressor(max_depth=4, n_estimators=1
         param_grid={
                     'max_depth': range(2,4),
                     'n_estimators': (50, 100),
                 }
         search = GridSearchCV(RandomForestRegressor(), param_grid, cv=10, scoring=metric, n_j
         search.fit(train, y_train)
         best_params_forest = search.best_params_
         print(f"{search.best_params_}")
         print(f"{np.sqrt(-search.best_score_):.4}")
```

```
{'max_depth': 3, 'n_estimators': 100}
0.05948
```

```
In [70]: #decision tree
         dt = make_pipeline(RobustScaler(), DecisionTreeRegressor(max_depth=4, min_samples_spl
         param_grid={
                     'max_depth': range(2,4)
                 }
         search = GridSearchCV(DecisionTreeRegressor(), param_grid, cv=10, scoring=metric, n_j
         search.fit(train, y_train)
         best_params_tree = search.best_params_
         print(f"{search.best_params_}")
         print(f"{np.sqrt(-search.best_score_):.4}")
```

```
{'max_depth': 3}
0.085
```

```
In [71]: #PCA
         #standardize dataset -- preparing for PCA: too many features
         scaler = StandardScaler()
         #fit on training set only.
         scaler.fit(train)
         # Apply transform to both the training set and the test set.
```

```
train = scaler.transform(train)

#pca: reducing dimensionality of features; getting rid of collinear features
#choose the minimum number of principal components such that 95% of the variance is r
pca = PCA(.95)
pca.fit(train)
train = pca.transform(train)
```

In [72]:
```
#random forest with PCA
rdf_pca = make_pipeline(RobustScaler(), RandomForestRegressor(max_depth=4, n_estimato
param_grid={
            'max_depth': range(2,4),
            'n_estimators': (50, 100),
        }
search = GridSearchCV(RandomForestRegressor(), param_grid, cv=10, scoring=metric, n_jo
search.fit(train, y_train)
best_params_forest = search.best_params_
print(f"{search.best_params_}")
print(f"{np.sqrt(-search.best_score_):.4}")
```

```
{'max_depth': 3, 'n_estimators': 100}
0.1548
```

In [73]:
```
#decision tree with PCA
dt_pca = make_pipeline(RobustScaler(), DecisionTreeRegressor(max_depth=4, min_samples_
param_grid={
            'max_depth': range(2,4)
        }
search = GridSearchCV(DecisionTreeRegressor(), param_grid, cv=10, scoring=metric, n_jo
search.fit(train, y_train)
best_params_tree = search.best_params_
print(f"{search.best_params_}")
print(f"{np.sqrt(-search.best_score_):.4}")
```

```
{'max_depth': 3}
0.1688
```

Tree Models perform much better without PCA, potentially because PCA dropped important features with small variance, or maybe the linearity of principal components and nonlinearity relationship between features conflict.