Housing_Kaggle

September 6, 2023

1 Predicting Housing Prices - Kaggle

In this notebook, I walk through the process of predicting housing prices in Kaggle's housing data competition. I prepare the data using various transformers. After processing the data, I use multiple algorithms and a bayesian optimizer to find the hyperparameters that minimze the error. Let's get started!

1.1 Libraries and Initial Set-Up

[1]:
```
# General libraries
from math import ceil
import pandas as pd
import numpy as np
import random as ran
import warnings
warnings.filterwarnings('ignore')

# Plots
import matplotlib.pyplot as plt

# Processing
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler
from sklearn.impute import KNNImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_selection import SelectKBest, mutual_info_regression
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import mean_squared_error

# Optimizer
from bayes_opt import BayesianOptimization

# ML Models
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor as RFR
from xgboost import XGBRegressor
from sklearn.neighbors import KNeighborsRegressor
```

```python
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from tensorflow.keras import optimizers
from scikeras.wrappers import KerasRegressor
from tensorflow.keras.callbacks import EarlyStopping
```

Above, I import all the necessary packages. I import general libraries necessary to begin the script. I then import libraries to plot, process the data, score the models, and the necessary algorithms.

Now, I'll do a little set up by pre-setting the seed, the number of crossfolds, importing the training data, finding missing values, and creating a comparison matrix. The comparision matrix will hold the name of the model, it's root mean squared error, and the best hyperparameters from the optimizer. The data frame allows me to compare model performance across each algorithm.

[2]:
```python
# Random draw of seed for random state #
#seed = int(ran.uniform(1, 9999))
''' Got 2095 '''
seed = 2095


# Set mnumber of cross folds #
cv = 5


# Import training data #
train = pd.read_csv('train.csv')


# Split into X and Y #
X = train.drop(['SalePrice', 'Id'], axis = 1)
y = pd.DataFrame(train['SalePrice'])


# Find missing values #
missing = X.isnull().sum().sort_values(ascending = False)


# Make matrix to compare models #
train_compare = pd.DataFrame(columns = ['Model', 'RMSE', 'hypers'])
```

1.2 Initial Alteration of the Data Frame

The data frame has many missing values due to not having the feature that is being measured. For example, the frame has a variable call 'PoolQC' that measures the quality of the swiming pool. If the home does not have a pool, then it it labeled as missing. To avoid missing values in the frame, missing values were altered to fit the variable of interest.

[3]:
```python
# Assign no pool (NP) to PoolQC #
X['PoolQC'].fillna('NP', inplace = True)


# Assign no feature (NF) to MiscFeature #
X['MiscFeature'].fillna('NF', inplace = True)


# Assign no ally (NAL) to Alley #
X['Alley'].fillna('NAL', inplace = True)


# Assign no fence (NF) to Fence #
X['Fence'].fillna('NF', inplace = True)


# Assign no fire place (NFP) to FireplaceQu #
X['FireplaceQu'].fillna('NFP', inplace = True)


# Assign no garage (NG) to GarageType #
X['GarageType'].fillna('NG', inplace = True)


# Fill garage variables with NG if no garage #
garage = ['GarageYrBlt', 'GarageFinish', 'GarageQual', 'GarageCond']
for x in garage:
    X[x].fillna('NG', inplace = True)
del x, garage


# Fix GarageYrBlt since it was mixed type #
X['GarageYrBlt'] = X['GarageYrBlt'].astype(str)


# Fill basement varaibles with no basement (NB) #
basement = ['BsmtExposure', 'BsmtFinType2', 'BsmtQual', 'BsmtCond',␣
 ↪'BsmtFinType1']
for x in basement:
    X[x].fillna('NB', inplace = True)
del x, basement
```

1.3 Plot Target Variable

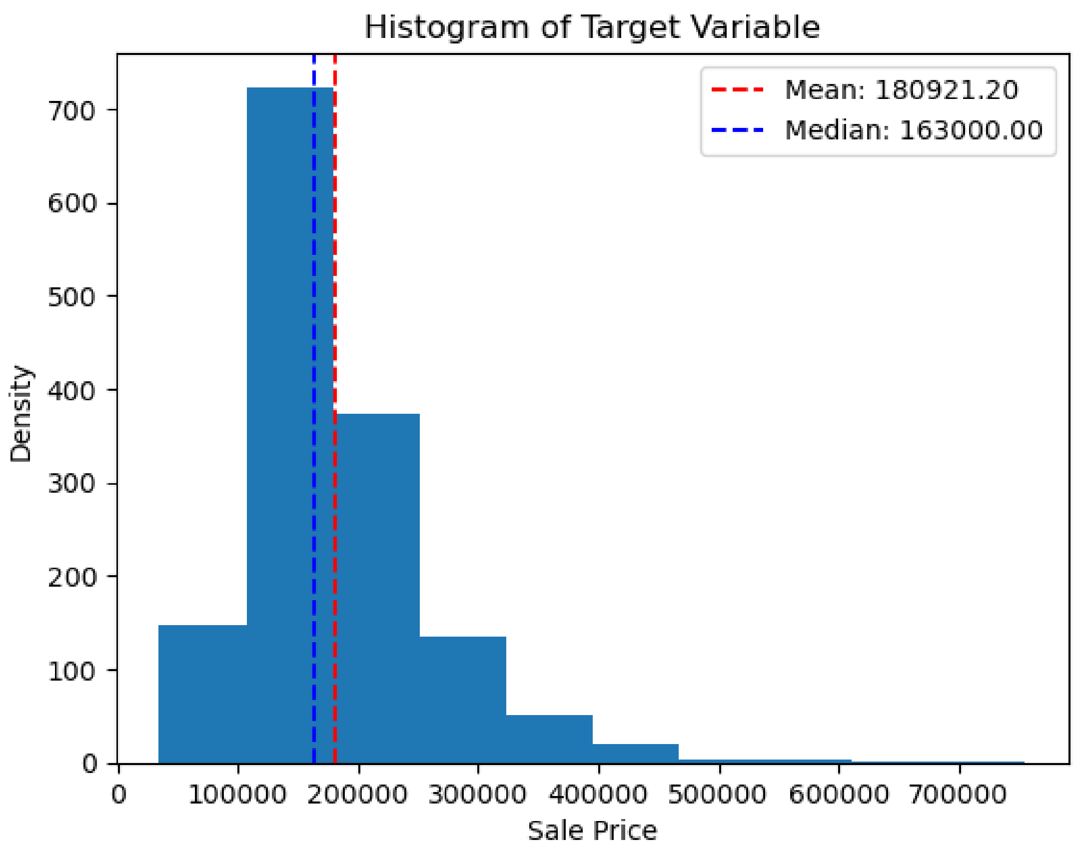
After initial set-up, I'll plot the target variable to check for non-normality.

[4]:

```
# Plot of Y #
plt.hist(y)
mean_value = y.mean()
median_value = y.median()
plt.axvline(mean_value.item(), color='red', linestyle='--', label='Mean')
plt.axvline(median_value.item(), color='blue', linestyle='--', label='Median')
plt.xlabel('Sale Price')
plt.ylabel('Density')
plt.title('Histogram of Target Variable')

# Add the text to the legend
mean_legend = plt.Line2D([], [], color='red', linestyle='--', label=f"Mean:␣
    ↪{mean_value.item():.2f}")
median_legend = plt.Line2D([], [], color='blue', linestyle='--', label=f"Median:
    ↪ {median_value.item():.2f}")
plt.legend(handles=[mean_legend, median_legend])

plt.show()
del mean_value, median_value
```



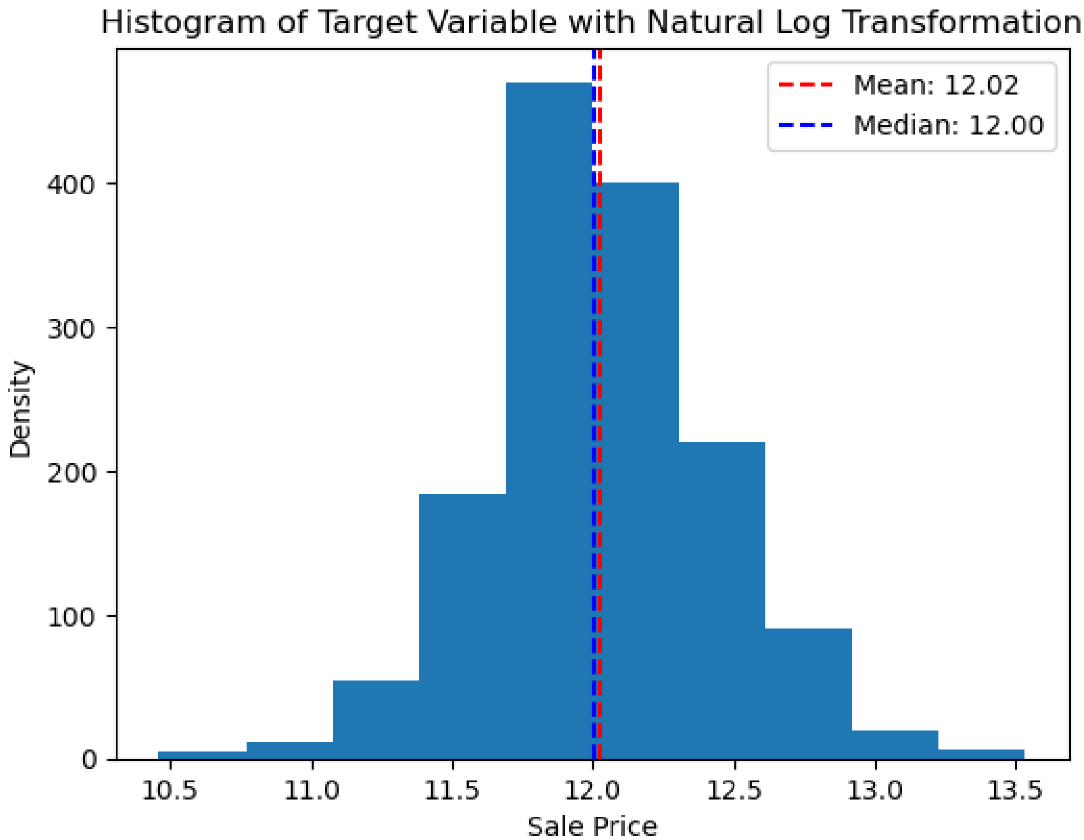
The plot above demonstrates a right skew in the target due to the right tail. In addition, there is distance between the mean and median values of the variable. I'll transform the variable with a natural log function and re-plot the data to ensure normality.

[5]:
```
# Natural log the target variable #
y_log = np.log(y)

plt.hist(y_log)
mean_value = y_log.mean()
median_value = y_log.median()
plt.axvline(mean_value.item(), color='red', linestyle='--', label='Mean')
plt.axvline(median_value.item(), color='blue', linestyle='--', label='Median')
plt.xlabel('Sale Price')
plt.ylabel('Density')
plt.title('Histogram of Target Variable with Natural Log Transformation')

# Add the text to the legend
mean_legend = plt.Line2D([], [], color='red', linestyle='--', label=f"Mean:␣
↪{mean_value.item():.2f}")
median_legend = plt.Line2D([], [], color='blue', linestyle='--', label=f"Median:
↪ {median_value.item():.2f}")
plt.legend(handles=[mean_legend, median_legend])

plt.show()
del mean_legend, mean_value, median_legend, median_value
```



After applying the transformation, the target variable is now noramlly distributed.

1.4 Preprocessing

I'll now process the data to ensure is compatible for ML models. I'll split the data into training and validation data, which will be analyzed at the end. After, I encode my categorical variables and implement a KNN imputer. Last, I make a few variables into integer values.

[6]:
```
# Split into train and validation sets #
X_train, X_val, y_train, y_val = train_test_split(X, y_log, test_size = 0.25,
                                                  random_state = seed)


# Mask of numerical features #
numeric_feats = X_train.select_dtypes(include = ['int64', 'float64']).columns



# Mask categorical features #
cat_feats = X_train.select_dtypes(include = ['object']).columns
```

```
# Encode object variables #
ord_enc = OrdinalEncoder(handle_unknown = 'use_encoded_value', unknown_value=np.
 ↪nan)
X_train[cat_feats] = ord_enc.fit_transform(X_train[cat_feats])


# Transform X_val with encoder #
X_val[cat_feats] = ord_enc.transform(X_val[cat_feats])


# KNN Imputer #
knn_im = KNNImputer(n_neighbors = 10, weights = 'distance')
X_train_imp = pd.DataFrame(knn_im.fit_transform(X_train), columns = X_train.
 ↪columns)


# KNN Imputer for X_val #
X_val_imp = pd.DataFrame(knn_im.transform(X_val), columns = X_val.columns)


# Variables need to be made integer #
X_train_imp[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st']] =␣
 ↪X_train_imp[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st']]\
                                        .apply(lambda x: x.apply(ceil))
X_val_imp[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st']] =␣
 ↪X_val_imp[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st']]\
                                        .apply(lambda x: x.apply(ceil))
```

1.5 Feature Selection

The housing data includes numerous features, and many features are likely to be uncorrelated with the outcome. I limit the number of numeric features with a correlation threshold and the categorical features based on mutual information.

[7]:
```
# Correlation of numeric feats and target #
corr_val = pd.concat([X_train_imp[numeric_feats], y_train], axis =1).
 ↪corr()['SalePrice']


# Mask of features with high correlation to target #
select_num = list(corr_val[(corr_val >= 0.50) | (corr_val <= -0.50)].index)
select_num.remove('SalePrice')


# Get mutual information of categorical features and outcome #
fs = SelectKBest(score_func = mutual_info_regression, k = 'all')
fs.fit(X_train_imp[cat_feats], np.ravel(y_train))
```

```
# Sort the scores and corresponding feature names in descending order
sorted_scores, sorted_features = zip(*sorted(zip(fs.scores_,
                                                 X_train_imp[cat_feats].
 ↪columns),
                                             reverse=True))


# Select the top 20 features
top_20_cat_features = list(sorted_features[:20])


# Combine feature selection list #
selected_feats = select_num + top_20_cat_features


# DF of only selected features #
X_train_imp = X_train_imp[selected_feats]
X_val_imp = X_val_imp[selected_feats]
del corr_val, fs, sorted_features, sorted_scores
```

1.6 Transformation

I now transform my data through categorical and numeric transformers based on the features in the limited data frame. I fit the transformer on the training data and then transform the training and validation data sets.

[8]:
```
########################
#### Transformation ####
########################

# Categorical transformer #
cat_trans = Pipeline(steps = [
    ('encode', OneHotEncoder(sparse_output = False))])


# Numeric tranformer #
num_trans = Pipeline(steps = [
    ('stand', StandardScaler())])


# Construct processor #
processor = ColumnTransformer(
    transformers = [
        ('num', num_trans, select_num),
        ('cat', cat_trans, top_20_cat_features)],
```

```
    remainder = 'passthrough')


# Apply processor to training data #
temp = processor.fit_transform(X_train_imp)


# Get categorical feature names #
enc_cat_features = list(processor.named_transformers_['cat']['encode']\
                        .get_feature_names_out())


# Concat label names #
labels = select_num + enc_cat_features


# Make df of processed data #
X_train = pd.DataFrame(temp, columns = labels)
del temp


# Apply processor to validation data #
temp = processor.transform(X_val_imp)


# Get categorical feature names #
enc_cat_features = list(processor.named_transformers_['cat']['encode']\
                        .get_feature_names_out())


# Concat label names #
labels = select_num + enc_cat_features


# Make df of processed data #
X_val = pd.DataFrame(temp, columns = labels)
del temp, y_log, X_val_imp, X_train_imp
```

2 Start Estimating

After preparing the training and validation datasets, the next few sections outline my strategies for predicting the outcome of interest. I'll use seven different estimation strategies that include Ridge, Lasso, Elastic Net, Support Vector Machine, Random Forest, XGBoost, KNN, and a Neural Net in their regression forms. I use a bayesian optimizer to search for the model with the best predictive power.

2.1 Ridge

[9]:
```
# Define objective for ridge #
def obj_ridge(alpha, fit_intercept, solver):

    """
    Objective function to minimize the error of the
    ridge regression.

    Parameters
    ----------
    alpha : L2 Regularization term.
        Regularizes the coefficients. Values stipulated
        in pbounds.
    fit_intercept : Boolean of fit intercept.
        Indicator of whether or not the model
        fits an intercept.
    solver : Solving method of ridge regression.
        Continuous variable for selecting the best
        solver for the regression.

    Returns
    -------
    error : Mean squared error.
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Fit intercept #
    fit_intercept = bool(round(fit_intercept))

    # Solver #
    if solver <= 1.0:
        solver = 'auto'
    elif solver <= 2.0:
        solver = 'svd'
    elif solver <= 3.0:
        solver = 'cholesky'
    elif solver <= 4.0:
        solver = 'lsqr'
    elif solver <= 5.0:
        solver = 'sparse_cg'
    elif solver <= 6.0:
        solver = 'sag'
    else:
        solver = 'saga'
```

```
    # Instantiate ridge model #
    model = Ridge(alpha=alpha, fit_intercept=fit_intercept, solver=solver,
                  max_iter = 20000, random_state=seed)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, y_train, cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error


# Define search space #
pbounds = {
    'alpha': (0.00000001, 100),
    'fit_intercept': (0, 1),
    'solver': (0, 8),
}

# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_ridge, pbounds=pbounds, random_state=seed,
    verbose = 0)

# Call maximizer #
optimizer.maximize(init_points=50, n_iter=450)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace solver with string #
if best_hypers['solver'] <= 1.0:
    best_hypers['solver']  = 'auto'
elif best_hypers['solver']  <= 2.0:
    best_hypers['solver']  = 'svd'
elif best_hypers['solver']  <= 3.0:
    best_hypers['solver']  = 'cholesky'
elif best_hypers['solver']  <= 4.0:
    best_hypers['solver']  = 'lsqr'
elif best_hypers['solver']  <= 5.0:
    best_hypers['solver']  = 'sparse_cg'
elif best_hypers['solver']  <= 6.0:
    best_hypers['solver']  = 'sag'
```

```python
else:
    best_hypers['solver']  = 'saga'

best_hypers['fit_intercept'] = bool(round(best_hypers['fit_intercept']))


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'Ridge',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)


# Sort by smallest RMSE #
train_compare = train_compare.sort_values('RMSE')
```

2.2 Lasso

[10]:
```python
# Define objective function for lasso #
def obj_lasso(alpha, fit_intercept,
              selection):

    """
    The objective of this function is to minimize the error
    of the lasso function.

    Parameters
    ----------
    alpha : L1 Regularization term.
        Regularizes the coefficients. Values stipulated
        in pbounds.
    fit_intercept : Boolean of fit intercept.
        Indicator of whether or not the model
        fits an intercept.
    selection : Dictates coefficient updates.
        Continuous variable of using either cycle or
        random for coefficient update.

    Returns
    -------
    error : Mean squared error.
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.
    """

    # Fit intercept #
    fit_intercept = bool(round(fit_intercept))
```

```python
    # selection #
    if selection <= 0.5:
        selection = 'cyclic'
    else:
        selection = 'random'

    # Instantiate model #
    model = Lasso(alpha = alpha, fit_intercept = fit_intercept,
                  selection = selection,
                  random_state = seed, max_iter = 20000)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, y_train, cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error


# Define search space #
pbounds = {
    'alpha': (0.0000001, 100),
    'fit_intercept': (0, 1),
    'selection': (0, 1)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_lasso, pbounds=pbounds, random_state=seed,
    verbose = 0)

# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace selection with string #
if best_hypers['selection'] <= 0.5:
    best_hypers['selection'] = 'cyclic'
else:
```

```
    best_hypers['selection'] = 'random'


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'Lasso',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.3 Elastic Net

[11]:
```
# Define objective function for Net #
def obj_net(alpha, l1_ratio, fit_intercept,
            selection):
    """
    The objective of this function is to minimize the error
    of the elastic net model.

    Parameters
    ----------
    alpha : Float
        Constant the multiplies the penalty terms. 0 is equal to OLS.
    l1_ratio : Float
        Ratio of l1 or l2 regularization. 0 is l2. 1 is l1.
    fit_intercept : bool
        Option to fit an intercept.
    selection : String
        Specify how coefficients are updated across iterations.

    Returns
    -------
    error : Float
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Vary fit intercept #
    fit_intercept = bool(round(fit_intercept))

    # Vary selection #
    if selection <= 0.5:
        selection = 'cyclic'
    else:
        selection = 'random'
```

```
    # Instantiate the model #
    model = ElasticNet(alpha = alpha, l1_ratio = l1_ratio,
                       fit_intercept =  fit_intercept,
                       selection = selection, random_state = seed,
                       max_iter = 20000)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, np.ravel(y_train), cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error

# Define search space #
pbounds = {
    'alpha': (0.00001, 100),
    'l1_ratio': (0.001, 0.99),
    'fit_intercept': (0, 1),
    'selection': (0, 1)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_net, pbounds=pbounds, random_state=seed,
    verbose = 0)


# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace selection with string #
if best_hypers['selection'] <= 0.5:
    best_hypers['selection']  = 'cyclic'
else:
    best_hypers['selection']  = 'random'


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
```

```
                                pd.DataFrame({'Model' : 'Elastic_Net',
                                 'RMSE': np.sqrt(best_mse * -1),
                                 'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.4 Support Vector Machine

[12]:
```
# Define objective function for SVM #
def obj_SVR(kernel, degree,
            gamma, C, epsilon,
            shrinking):
    """
    The objective of this function is to minimze the erro
    of the support vector regression.

    Parameters
    ----------
    kernel : Kernel used in solver.
        String inputs that are used in optimizer.
    degree : Degree of polyomial kernel.
        Only used in poly alogrithm.
    gamma : Kernel cofficient.
        Only used in rbf, poly, and sigmoid.
    C : L2 regularizer.
        More regularization at smaller values.
    epsilon : Epplison value in SVR model.
        Specifies penalty in training loss function.
    shrinking : Boolean value.
        Dictates if the model uses shrinking heuristic.

    Returns
    -------
    error : Mean squared error.
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Kernel #
    if kernel <= 1:
        kernel = 'linear'
    elif kernel <= 2:
        kernel = 'poly'
    elif kernel <= 3:
        kernel = 'rbf'
    else:
```

```
        kernel = 'sigmoid'

    # Gamma #
    if gamma <= 0.5:
        gamma = 'scale'
    else:
        gamma = 'auto'

    # Shrinking #
    shrinking = bool(round(shrinking))

    # Instantiate SVR #
    model = SVR(kernel =  kernel, degree = int(degree),
                gamma = gamma, C = C,
                epsilon = epsilon, shrinking = shrinking,
                max_iter = 50000)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, np.ravel(y_train), cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error


# Define search space #
pbounds = {
    'kernel': (0, 4),
    'degree': (1, 10),
    'gamma': (0, 1),
    'C': (0.0001, 100),
    'epsilon': (0.0001, 100),
    'shrinking': (0, 1)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_SVR, pbounds=pbounds, random_state=seed,
    verbose = 0)


# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450)


# Pull best info #
```

```python
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace kernel with string #
if best_hypers['kernel'] <= 1:
    best_hypers['kernel'] = 'linear'
elif best_hypers['kernel'] <= 2:
    best_hypers['kernel'] = 'poly'
elif best_hypers['kernel'] <= 3:
    best_hypers['kernel'] = 'rbf'
else:
    best_hypers['kernel'] = 'sigmoid'


# Replace gamma with string #
if best_hypers['gamma'] <= 0.5:
    best_hypers['gamma'] = 'scale'
else:
    best_hypers['gamma'] = 'auto'


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'SVR',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.5 Random Forest

[13]:
```python
# Define objective function for random forest #
def obj_RF(n_estimators, criterion,
           min_samples_split, min_samples_leaf,
           max_features, bootstrap, min_impurity_decrease):
    """



    Parameters
    ----------
    n_estimators : Float
        Number of trees to estimate in the forest.
    criterion : String
        How the tree measures quality of the split.
    min_samples_split : Float
        Minimum number of samples required to split a node.
```

```python
    min_samples_leaf : Float
        Minimum number of samples required to be a leaf.
    max_features : String
        Number of features to consider when splitting.
    bootstrap : Boolean
        Whether bootstraps are used when building trees.
    min_impurity_decrease : Float
        Node is split if it decreases the impurity.

    Returns
    -------
    error : Mean squared error.
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Criterion #
    if criterion <= 1.0:
        criterion = 'squared_error'
    elif criterion <= 2.0:
        criterion = 'absolute_error'
    elif criterion <= 3.0:
        criterion = 'friedman_mse'
    else:
        criterion = 'poisson'

    # Max features #
    if max_features <= 0.5:
        max_features = 'sqrt'
    else:
        max_features = 'log2'

    # Bootstrap #
    bootstrap = bool(round(bootstrap))

    # instantiate random forest moel #
    model = RFR(n_estimators = int(n_estimators), criterion = criterion,
                min_samples_split =  min_samples_split,
                min_samples_leaf = min_samples_leaf,
                max_features =  max_features, bootstrap =  bootstrap,
                min_impurity_decrease = min_impurity_decrease,
                n_jobs = -1, random_state = seed)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, np.ravel(y_train), cv=cv,
                            scoring='neg_mean_squared_error').mean()
```

```
    # Return error #
    return error


# Define search space #
pbounds = {
    'n_estimators': (1, 1000),
    'criterion': (0, 4),
    'min_samples_split': (0.01, .70),
    'min_samples_leaf': (0.01, .70),
    'max_features': (0, 1),
    'bootstrap': (0, 1),
    'min_impurity_decrease': (0.001, 0.4)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_RF, pbounds=pbounds, random_state=seed,
    verbose = 0)


# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450,)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace criterion with string #
if best_hypers['criterion'] <= 1.0:
    best_hypers['criterion'] = 'squared_error'
elif best_hypers['criterion'] <= 2.0:
    best_hypers['criterion'] = 'absolute_error'
elif best_hypers['criterion'] <= 3.0:
    best_hypers['criterion'] = 'friedman_mse'
else:
    best_hypers['criterion'] = 'poisson'


# Replace max features with string #
if best_hypers['max_features'] <= 0.5:
    best_hypers['max_features'] = 'sqrt'
else:
```

```python
    best_hypers['max_features'] = 'log2'


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'Random_Forest',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.6 XGBoost

[14]:
```python
# Define objective function for XGBoost regression #
def obj_boost(n_estimators, eta, gamma,
              max_depth, subsample, colsample_bytree,
              reg_lambda, alpha):
    """
    The objective of this function is to minimze the error
    of the XGBoosted random forest regression.

    Parameters
    ----------
    n_estimators : Integer
        Number of trees to estimate.
    eta : Float
        Feature weight shrinkage that prevents overfitting.
    gamma : Float
        Min loss reduction needed to make partition on a leaf node.
    max_depth : Int
        Maximum depth of a tree. Deeper trees increase overfitting.
    subsample : Float
        Subsample of the dataset to use in tree.
    colsample_bytree : Float
        Subsample of columns to use in each tree.
    reg_lambda : Float
        L2 regularization on weights. Higher values make models more␣
↪conservative.
    alpha : Float
        L1 regularization on weights. Higher values make models more␣
↪conservative.

    Returns
    -------
    error : Float
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.
```

```
    """

    # instantiate XGBoost #
    model = XGBRegressor(n_estimators = int(n_estimators), eta = eta,
                         gamma = gamma, max_depth = int(max_depth),
                         subsample = subsample, colsample_bytree =␣
 ↪colsample_bytree,
                         reg_lambda = reg_lambda, alpha = alpha,
                         seed = seed, n_jobs = 1)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, np.ravel(y_train), cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error


# Define the search space #
pbounds = {
    'n_estimators': (1, 2000),
    'eta': (0, 1),
    'gamma': (0, 5),
    'max_depth': (2, 7),
    'subsample': (0.5, 1),
    'colsample_bytree': (0.2, 0.9),
    'reg_lambda': (0.05, 10),
    'alpha': (0.05, 10)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_boost, pbounds=pbounds, random_state=seed,
    verbose = 0)


# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']
```

```
# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'XGBoost_Reg',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.7 KNN

[15]:
```
# Define objective function for K-Nearest Neighbors #
def obj_knn(n_neighbors, weights, algorithm,
            leaf_size, p):
    """
    This objective function minimzes the error for k-nearest
    neighbors regression.

    Parameters
    ----------
    n_neighbors : int
        Number of neighbors to use.
    weights : String
        Weight function used in prediction.
    algorithm : String
        Process used to compute the nearest neighbors.
    leaf_size : int
        Leaf size passed to specific algorithms.
    p : int
        Power parameter for Minkowski metric.

    Returns
    -------
    error : Float
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Variation on weights #
    if weights <= 0.5:
        weights = 'uniform'
    else:
        weights = 'distance'

    # Variation on algorithm #
    if algorithm <= 1.0:
        algorithm = 'auto'
```

```
    elif algorithm <= 2.0:
        algorithm = 'ball_tree'
    elif algorithm <= 3.0:
        algorithm = 'kd_tree'
    else:
        algorithm = 'brute'

    # Variation on p #
    if p <= 1.0:
        p = 1
    elif p <= 1.0 and algorithm != 'brute':
        p = 1
    else:
        p = 2

    # Instantiate model #
    model = KNeighborsRegressor(n_neighbors =  int(n_neighbors), weights =␣
↪weights,
                                algorithm = algorithm, leaf_size =␣
↪int(leaf_size), p = p)

    # Cross validation and mean MSE #
    error = cross_val_score(model, X_train, np.ravel(y_train), cv=cv,
                            scoring='neg_mean_squared_error').mean()

    # Return error #
    return error


# Define search space #
pbounds = {
    'n_neighbors': (2, 10),
    'weights': (0, 1),
    'algorithm': (0, 4),
    'leaf_size': (2, 50),
    'p': (0.001, 2)
}


# Set the optimizer #
optimizer = BayesianOptimization(
    f=obj_knn, pbounds=pbounds, random_state=seed,
    verbose = 0)


# Call maximizer #
optimizer.maximize(init_points = 50, n_iter = 450)
```

```
# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace weights with string #
if best_hypers['weights'] <= 0.5:
    best_hypers['weights'] = 'uniform'
else:
    best_hypers['weights'] = 'distance'


# Replace algorithm with string #
if best_hypers['algorithm'] <= 1.0:
    best_hypers['algorithm'] = 'auto'
elif best_hypers['algorithm'] <= 2.0:
    best_hypers['algorithm'] = 'ball_tree'
elif best_hypers['algorithm'] <= 3.0:
    best_hypers['algorithm'] = 'kd_tree'
else:
    best_hypers['algorithm'] = 'brute'


# Replace p #
if best_hypers['p'] <= 1.0:
    best_hypers['p'] = 1
elif best_hypers['p'] <= 1.0 and best_hypers['algorithm'] != 'brute':
    best_hypers['p'] = 1
else:
    best_hypers['p'] = 2



# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'KNN_Reg',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

2.8 Neural Network

[16]:
```python
# Define objective function for network #
def obj_net(batch_size, epochs, activation, num_nodes,
            num_hidden_layers, learning_rate, rate, optimizer):
    """
    The objective of this function is to minimize the error of the
    neural network

    Parameters
    ----------
    batch_size : Int
        The number of cases to include in each batch.
    epochs : Int
        Number of runs through the data when updating weights.
    activation : String
        Type of activation function for the layer.
    num_nodes : Int
        Number of nodes to include in the hidden layer.
    num_hidden_layers : Int
        Number of hideen layers in the model.
    learning_rate : Float
        How much to change the model with each model update.
    rate : Float
        Dropout rate for each hidden layer to prevent overfitting.
    optimizer : String
        Optimizer to use for the model.

    Returns
    -------
    error : Float
        Cross validation returns root mean error that is later
        convereted into RMSE in the comparison frame.

    """

    # Set Optimizer #
    if optimizer <= 0.33:
        optimizer = optimizers.Adam(learning_rate = learning_rate)

    elif optimizer <= 0.66:
        optimizer = optimizers.Adagrad(learning_rate = learning_rate)

    else:
        optimizer = optimizers.RMSprop(learning_rate = learning_rate)

    # Set activation function #
    if activation <= 0.33:
```

```python
        activation = 'relu'

    elif activation <= 0.66:
       activation = 'sigmoid'

    else:
       activation = 'tanh'

    # Instantiate model
    model = Sequential()

    # Set input layer #
    model.add(Dense(int(num_nodes), activation = activation,
                    input_shape = (X_train.shape[1],)))

    # Set hidden layer with batch normalizer #
    for _ in range(int(num_hidden_layers)):
        model.add(Dense(int(num_nodes), activation = activation))
        model.add(BatchNormalization())
        model.add(Dropout(rate = rate, seed = seed))

    # Add output layer #
    model.add(Dense(1))

    # Set compiler #
    model.compile(optimizer = optimizer,
                  loss = 'mean_squared_error')

    # Set early stopping #
    early_stopping = EarlyStopping(monitor='val_loss',
                                   patience=15,
                                   restore_best_weights=True)

    # Create model #
    reg = KerasRegressor(model = lambda : model,
                         batch_size = int(batch_size),
                         epochs = int(epochs),
                         validation_split = 0.2,
                         callbacks = [early_stopping],
                         random_state = seed,
                         verbose = 0 )

    # Cross validation and mean MSE #
    error = cross_val_score(reg, X_train, np.ravel(y_train), cv=cv,
                        scoring='neg_mean_squared_error').mean()

    # Return error #
```

```
    return error

# Define search space #
pbounds = {
    'batch_size': (50, 1460),
    'epochs': (5, 500),
    'learning_rate': (0.001, 0.15),
    'num_nodes': (5, 80),
    'num_hidden_layers': (2, 20),
    'activation': (0, 1),
    'rate': (0.0, 0.9),
    'optimizer': (0, 1)
}


# Set the optimizer #
optimizer = BayesianOptimization(f=obj_net, pbounds=pbounds,
                                 random_state=seed,
                                 verbose = 0)


# Call the maximizer #
optimizer.maximize(init_points=50, n_iter=450)


# Pull best info #
best_hypers = optimizer.max['params']
best_mse = optimizer.max['target']


# Replace optimizer and learning rate #
if best_hypers['optimizer'] <= 0.33:
    best_hypers['optimizer'] = optimizers.Adam(learning_rate = best_hypers['learning_rate'])

elif best_hypers['optimizer'] <= 0.66:
    best_hypers['optimizer'] = optimizers.Adagrad(learning_rate = best_hypers['learning_rate'])

else:
    best_hypers['optimizer'] = optimizers.RMSprop(learning_rate = best_hypers['learning_rate'])


# Replace activation with string #
if best_hypers['activation'] <= 0.33:
    best_hypers['activation'] = 'relu'
```

```
elif best_hypers['activation'] <= 0.66:
      best_hypers['activation'] = 'sigmoid'

else:
     best_hypers['activation'] = 'tanh'


# Fill comparison matrix #
train_compare = pd.concat([train_compare,
                           pd.DataFrame({'Model' : 'Neural_Net',
                            'RMSE': np.sqrt(best_mse * -1),
                            'hypers': [best_hypers]})], ignore_index = True)

train_compare = train_compare.sort_values('RMSE')
```

```
2023-09-06 16:54:36.009166: W
tensorflow/tsl/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU
frequency: 0 Hz
```

2.9 Best Models

After creating and optimizing the models, the script filled the train_compare matrix with the best models and their associated errors.

[17]:
```
display(train_compare)
```

	Model	RMSE	hypers
7	Neural_Net	0.169579	{'activation': 'relu', 'batch_size': 263.09534…
0	Ridge	0.186247	{'alpha': 6.3440798520386705, 'fit_intercept':…
1	XGBoost_Reg	0.186940	{'alpha': 0.3281333441940331, 'colsample_bytre…
2	SVR	0.190142	{'C': 22.462618867292676, 'degree': 9.02554746…
3	KNN_Reg	0.202695	{'algorithm': 'kd_tree', 'leaf_size': 22.98472…
4	Elastic_Net	0.211990	{'alpha': 1e-05, 'fit_intercept': 0.0, 'l1_rat…
5	Lasso	0.214700	{'alpha': 1e-07, 'fit_intercept': 0.0, 'select…
6	Random_Forest	0.222696	{'bootstrap': 0.24545196171088643, 'criterion'…

2.10 Predict on the Validation Set

After training each model, I now predict on the validation set to see which model does the best job.

[18]:
```
# Ridge #
ridge_dict = train_compare.loc[train_compare['Model'] == 'Ridge', 'hypers'].
 ↪values[0]


mod_ridge = Ridge(alpha = ridge_dict['alpha'],
                  fit_intercept =ridge_dict['fit_intercept'],
```

```
                  solver = ridge_dict['solver'],
                  random_state = seed)

mod_ridge.fit(X_train, y_train)


# Lasso #
lasso_dict = train_compare.loc[train_compare['Model'] == 'Lasso', 'hypers'].
 ↪values[0]

mod_lasso = Lasso(alpha = lasso_dict['alpha'],
                  fit_intercept = bool(lasso_dict['fit_intercept']),
                  selection = lasso_dict['selection'],
                  random_state = seed,
                  max_iter = 20000)

mod_lasso.fit(X_train, y_train)


# Elastic Net #
elastic_dict = train_compare.loc[train_compare['Model'] == 'Elastic_Net',␣
 ↪'hypers'].values[0]

mod_elastic = ElasticNet(alpha = elastic_dict['alpha'],
                         fit_intercept = bool(elastic_dict['fit_intercept']),
                         l1_ratio = elastic_dict['l1_ratio'],
                         selection = elastic_dict['selection'],
                         max_iter = 20000,
                         random_state = seed)

mod_elastic.fit(X_train, y_train)


# SVR #
svr_dict = train_compare.loc[train_compare['Model'] == 'SVR', 'hypers'].
 ↪values[0]

mod_SVR = SVR(C = svr_dict['C'],
              degree = int(svr_dict['degree']),
              epsilon = svr_dict['epsilon'],
              gamma = svr_dict['gamma'],
              kernel = svr_dict['kernel'],
              shrinking = bool(svr_dict['kernel']))

mod_SVR.fit(X_train, y_train)
```

```
# Random Forest #
rf_dict = train_compare.loc[train_compare['Model'] == 'Random_Forest',␣
 ↪'hypers'].values[0]

mod_rf = RFR(bootstrap = bool(rf_dict['bootstrap']),
             criterion = rf_dict['criterion'],
             max_features = rf_dict['max_features'],
             min_impurity_decrease = rf_dict['min_impurity_decrease'],
             min_samples_leaf = rf_dict['min_samples_leaf'],
             min_samples_split = rf_dict['min_samples_split'],
             n_estimators = int(rf_dict['n_estimators']))

mod_rf.fit(X_train, np.ravel(y_train))


# XGBoost Regression #
boost_dict = train_compare.loc[train_compare['Model'] == 'XGBoost_Reg',␣
 ↪'hypers'].values[0]

mod_boost = XGBRegressor(alpha = boost_dict['alpha'],
                         colsample_bytree = boost_dict['colsample_bytree'],
                         eta = boost_dict['eta'],
                         gamma = boost_dict['gamma'],
                         max_depth = int(boost_dict['max_depth']),
                         n_estimators = int(boost_dict['n_estimators']),
                         reg_lambda = boost_dict['reg_lambda'],
                         subsample = boost_dict['subsample'],
                         random_state = seed,
                         n_jobs = 3)

mod_boost.fit(X_train, np.ravel(y_train))


# K-Nearest #
knn_dict = train_compare.loc[train_compare['Model'] == 'KNN_Reg', 'hypers'].
 ↪values[0]

mod_KNN = KNeighborsRegressor(algorithm = knn_dict['algorithm'],
                              leaf_size = int(knn_dict['leaf_size']),
                              n_neighbors = int(knn_dict['n_neighbors']),
                              p = float(knn_dict['p']),
                              weights = knn_dict['weights'])

mod_KNN.fit(X_train, y_train)


# Neutral Net #
```

```
net_dict = train_compare.loc[train_compare['Model'] == 'Neural_Net', 'hypers'].
 ↪values[0]


mod_net = Sequential()

mod_net.add(Dense(int(net_dict['num_nodes']),
                  activation = net_dict['activation'],
                  input_shape = (X_train.shape[1],)))


# Set hidden layer with batch normalizer #
for _ in range(int(net_dict['num_hidden_layers'])):
    mod_net.add(Dense(int(net_dict['num_nodes']), activation =␣
 ↪net_dict['activation']))
    mod_net.add(BatchNormalization())
    mod_net.add(Dropout(rate = net_dict['rate'], seed = seed))


# Add output layer #
mod_net.add(Dense(1))

# Set compiler #
optimizer = optimizers.get(net_dict['optimizer'])
optimizer.build(mod_net.trainable_variables)
mod_net.compile(optimizer = net_dict['optimizer'],
              loss = 'mean_squared_error')

mod_net.fit(X_train, np.ravel(y_train))


# Make list of models #
mod_list = [mod_ridge, mod_lasso, mod_elastic, mod_SVR, mod_rf, mod_boost,␣
 ↪mod_KNN, mod_net]


# Make matrix to compare models #
val_compare = pd.DataFrame(columns = ['Model', 'RMSE'])


# Loop model predictions on validation set #
for x in mod_list:
    pred = x.predict(X_val)
    mse = mean_squared_error(y_val, pred)
    rmse = np.sqrt(mse)
    model_name = type(x).__name__
    val_compare = pd.concat([val_compare,
```

```
                                pd.DataFrame({'Model': [model_name],
                                               'RMSE': [np.exp(rmse)],
                                               'Model_Specs': [x]})],
                               ignore_index = True)


# Sort by RMSE #
val_compare = val_compare.sort_values('RMSE')


# Display Dataframe #
display(val_compare)
```

```
35/35 [==============================] - 0s 896us/step - loss: 19.8544
12/12 [==============================] - 0s 490us/step
```

```
                   Model       RMSE  \
0                  Ridge   1.219422
5           XGBRegressor   1.220066
3                    SVR   1.223673
2             ElasticNet   1.233840
1                  Lasso   1.235832
6    KNeighborsRegressor   1.240242
4  RandomForestRegressor   1.255096
7             Sequential  12.484327

                                         Model_Specs
0  Ridge(alpha=6.3440798520386705, random_state=2…
5  XGBRegressor(alpha=0.3281333441940331, base_sc…
3  SVR(C=22.462618867292676, degree=9, epsilon=0…
2  ElasticNet(alpha=1e-05, fit_intercept=False, l…
1  Lasso(alpha=1e-07, fit_intercept=False, max_it…
6  KNeighborsRegressor(algorithm='kd_tree', leaf_…
4  (DecisionTreeRegressor(criterion='friedman_mse…
7  <keras.engine.sequential.Sequential object at …
```

2.11 Import and Prepare Test Data

I now prepare the test dataset with the transformers created earlier.

[19]:
```
# Import test data #
X_test = pd.read_csv('test.csv')
ids = X_test['Id'].values


# Drop ID #
X_test = X_test.drop("Id", axis = 1)
```

```
# Assign no pool (NP) to PoolQC #
X_test['PoolQC'].fillna('NP', inplace = True)


# Assign no feature (NF) to MiscFeature #
X_test['MiscFeature'].fillna('NF', inplace = True)


# Assign no ally (NAL) to Alley #
X_test['Alley'].fillna('NAL', inplace = True)


# Assign no fence (NF) to Fence #
X_test['Fence'].fillna('NF', inplace = True)


# Assign no fire place (NFP) to FireplaceQu #
X_test['FireplaceQu'].fillna('NFP', inplace = True)


# Assign no garage (NG) to GarageType #
X_test['GarageType'].fillna('NG', inplace = True)


# Fill garage variables with NG if no garage #
garage = ['GarageYrBlt', 'GarageFinish', 'GarageQual', 'GarageCond']
for x in garage:
    X_test[x].fillna('NG', inplace = True)
del x, garage


# Fix GarageYrBlt since it was mixed type #
X_test['GarageYrBlt'] =X_test['GarageYrBlt'].astype(str)


# Fill basement varaibles with no basement (NB) #
basement = ['BsmtExposure', 'BsmtFinType2', 'BsmtQual', 'BsmtCond',␣
 ↪'BsmtFinType1']
for x in basement:
    X_test[x].fillna('NB', inplace = True)
del x, basement


# Transform X_test with encoder #
X_test[cat_feats] = ord_enc.transform(X_test[cat_feats])
```

```python
# KNN Imputer for X_val #
X_test = pd.DataFrame(knn_im.transform(X_test), columns = X_test.columns)


# Variables need to be made integer #
X_test[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st',␣
↪'Exterior2nd', 'KitchenQual', 'MSZoning', 'SaleType']] =\
    X_test[['Electrical', 'MasVnrType', 'GarageYrBlt', 'Exterior1st',␣
↪'Exterior2nd', 'KitchenQual', 'MSZoning', 'SaleType']]\
                                        .apply(lambda x: x.apply(ceil))


# Get pre-selected features #
X_test = X_test[selected_feats]


# Apply processor to validation data #
temp = processor.transform(X_test)


# Get categorical feature names #
enc_cat_features = list(processor.named_transformers_['cat']['encode']\
                        .get_feature_names_out())


# Concat label names #
labels = select_num + enc_cat_features


# Make df of processed data #
X_test = pd.DataFrame(temp, columns = labels)
```

2.12 Predict on Test Set

To finish up the problem, I predict using the test features and the best model from the validation testing stage. I then store these predictions in a CSV file to submit to Kaggle.

[20]:
```python
# Pull best model from val_compare #
best_model = val_compare.iloc[0]['Model_Specs']


# Predict on test set #
predict_test = best_model.predict(X_test)


# Exponentiate to make into real dollars #
```

```
predict_test = np.exp(predict_test)


# Create new DataFrame with the IDs and the predicted sale prices #
predictions_with_id = pd.DataFrame({'Id': ids.ravel(), 'SalePrice':␣
↪predict_test.ravel()})


# Save the predictions to a CSV file with the original IDs
predictions_with_id.to_csv('predicted_sale_prices.csv', index=False)
```

[ ]: