Kaggle Competition: House Prices: Regression Analyses

MSDS 422: Module 2 Assignment 1

Requirements

1) Conduct your analysis using a cross-validation design.
2) Conduct EDA and provide appropriate visualizations in the process.
3) Build a minimum of two separate regression models using the training set.
4) Evaluate polynomial, indicator, dichotomous, & piecewise model components.
5) Create at least one feature from the data set.
6) Evaluate the models' assumptions.
7) Evaluate goodness of fit metrics on the training and validation sets.
8) Submit predictions for the unseen test set available on Kaggle.com.
9) Provide your Kaggle user name and a screen snapshot of your Kaggle scores.
10) Discuss what your models tell you in layman's terms.

Data Preparation, Exploration, and Visualization

In this section, I want to use my previous EDA as a baseline and improve my data cleaning so that the linear regression models that I will use in the future can be more accurate. This will involve missing value imputation and creating dummy variables.

In [1]:
```
# import modules
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns
import re
import numpy as np
from scipy import stats
from sklearn import tree
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics

# Figures inline and set visualization style
%matplotlib inline
sns.set()
```

In [2]:
```
# import train and test sets
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
```

In [3]:
```
# store target variable of training data in a safe place
sale_price_train = train.SalePrice

# store ID column separately since it is useless for prediction
train_id = train.Id
test_id = test.Id

train.drop("Id", axis=1, inplace=True)
test.drop("Id", axis=1, inplace=True)

# concatenate training and test sets for EDA
data = pd.concat([train.drop(['SalePrice'], axis=1), test])
```

In [4]:
```
# showing the first few rows of the data
data.head()
```

Out[4]:

	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities
0	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub
1	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub
2	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub
3	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub
4	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub

5 rows × 79 columns

In my last EDA, I placed a heavy emphasis on exploring numerical variables. However, just by looking at the first few rows of the `data` DataFrame, we can see that we really have a lot more than just numerical data. It's important to use these variables as well for our regression models later.

Let's explore beyond what we have already explored in the `eda.ipynb`, which is from Module 1 Assignment 1. I would like to explore the proportion of missing data for each column.

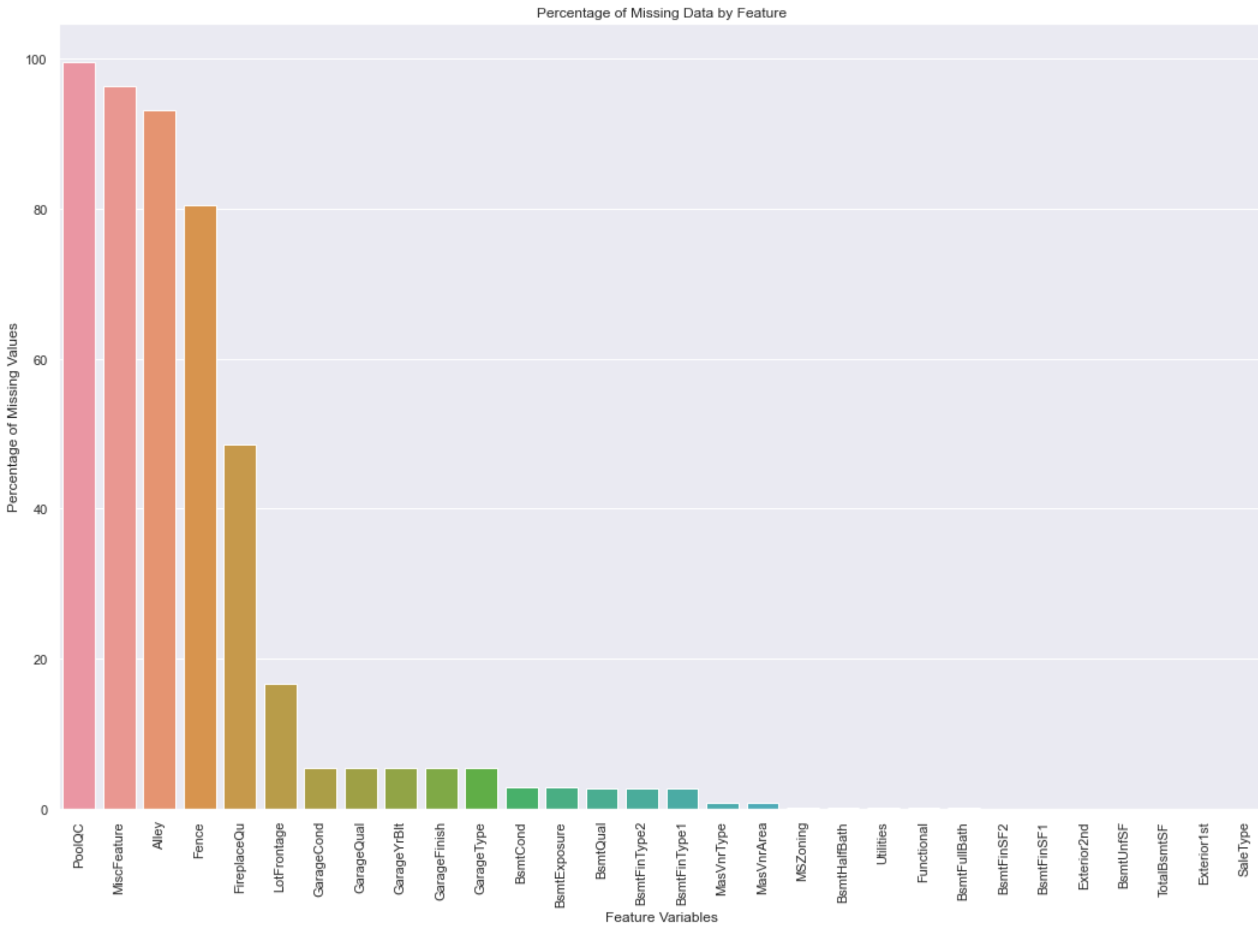
In [5]:
```
data_null = ((data.isnull().sum() / len(data)) * 100).sort_values(ascending=Fals
missing_data = pd.DataFrame({"Percentage of Missing Data": data_null})
missing_data.head()
```

Out[5]:

	Percentage of Missing Data
PoolQC	99.657417
MiscFeature	96.402878
Alley	93.216855
Fence	80.438506
FireplaceQu	48.646797

In [6]:
```
# plotting a bar plot to compare the variables and their proportion of missing d
fig, ax = plt.subplots(figsize=(18, 12))
plt.xticks(rotation="90")
short_missing_data = missing_data.iloc[:30]
```

```
sns.barplot(x=short_missing_data.index, y=short_missing_data["Percentage of Miss
plt.xlabel("Feature Variables")
plt.ylabel("Percentage of Missing Values")
plt.title("Percentage of Missing Data by Feature");
```



Insights: PoolQC, MiscFeature, Alley, and Fence are the variables that have the most amount of missing data. We should investigate the variables with missing data, and figure out what NaN values mean for these variables.

In [7]:
```
missing_data = missing_data.loc[missing_data["Percentage of Missing Data"] > 0]
missing_data.tail()
```

Out[7]:

	Percentage of Missing Data
SaleType	0.034258
Electrical	0.034258
KitchenQual	0.034258
GarageArea	0.034258
GarageCars	0.034258

In [8]:
```
print(f"We have to explore {len(missing_data)} columns and their null value mean
```

```
We have to explore 34 columns and their null value meanings
```

Missing Value Imputation

I will go in order by highest percentage of missing data from the `missing_data` DF, and will refer to the Kaggle data dictionary for this comptetition, found here.

In [9]:
```
missing_data.index
```

Out[9]:
```
Index(['PoolQC', 'MiscFeature', 'Alley', 'Fence', 'FireplaceQu', 'LotFrontage',
       'GarageCond', 'GarageQual', 'GarageYrBlt', 'GarageFinish', 'GarageType',
       'BsmtCond', 'BsmtExposure', 'BsmtQual', 'BsmtFinType2', 'BsmtFinType1',
       'MasVnrType', 'MasVnrArea', 'MSZoning', 'BsmtHalfBath', 'Utilities',
       'Functional', 'BsmtFullBath', 'BsmtFinSF2', 'BsmtFinSF1', 'Exterior2nd',
       'BsmtUnfSF', 'TotalBsmtSF', 'Exterior1st', 'SaleType', 'Electrical',
       'KitchenQual', 'GarageArea', 'GarageCars'],
      dtype='object')
```

- **PoolQC:** This is pool quality. A value of "NA" means there is no pool. Thus, we should impute this properly with the value "No Pool" for clarity.

In [10]:
```
data["PoolQC"].fillna("None", inplace=True)

# checking that "inplace" works for fillna
data[["PoolQC"]].head()
```

Out[10]:

	PoolQC
0	None
1	None
2	None
3	None
4	None

- **MiscFeature**: Data description says this is miscellaneous feature not covered in other categories. A value of "NA" means there are no misc features. We can encode this is "None" instead.

In [11]:
```
data["MiscFeature"].fillna("None", inplace=True)
```

- **Alley:** This is the type of alley access to property. NA means no alley access. We can encode this with "None."

In [12]:
```
data["Alley"].fillna("None", inplace=True)
```

- **Fence:** This is fence quality. NA means no fence.

In [13]:
```
data["Fence"].fillna("None", inplace=True)
```

- **FireplaceQu**: Fireplace quality. NA means no fireplace.

In [14]:
```
data["FireplaceQu"].fillna("None", inplace=True)
```

- **LotFrontage:** Linear feet of street connected to property. This is a numerical value, so we should impute the values with the median value. Because LotFrontage is likely similar for each house in the same neighborhood, we should group by neighborhood to find the proper median value.

In [15]:
```
data["LotFrontage"] = data.groupby("Neighborhood")["LotFrontage"].transform(lamb

# Double checking that this worked
(data[["LotFrontage"]].isnull().sum() / len(data)) * 100
```

Out[15]:
```
LotFrontage    0.0
dtype: float64
```

- **GarageCond, GarageQual, GarageFinish, GarageType**: Any NA values within these columns mean "no garage". We can encode with None.

In [16]:
```
for c in ["GarageCond", "GarageQual", "GarageFinish", "GarageType"]:
    data[c].fillna("None", inplace=True)
```

- **GarageYrBlt, GarageArea, GarageCars**: These are numerical values, so NA values should be replaced with the value of 0 rather than a string like "None." NA values mean there is no garage, so imputing with 0 makes more sense.

In [17]:
```
for c in ["GarageYrBlt", "GarageArea", "GarageCars"]:
    data[c].fillna(0, inplace=True)
```

- **BsmtCond, BsmtExposure, BsmtQual, BsmtFinType1, BsmtFinType2:** These are qualitative variables about the basement, where the NA values mean there is no basement. We can encode these with "None" instead.

In [18]:
```
for c in ["BsmtCond", "BsmtExposure", "BsmtQual", "BsmtFinType1", "BsmtFinType2"
    data[c].fillna("None", inplace=True)
```

- **BsmtHalfBath, BsmtFullBath, BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF:** These are numerical variables about the basement, where the NA values should be translated to 0 since there are none of these features regarding basement.

In [19]:
```
for c in ["BsmtHalfBath", "BsmtFullBath", "BsmtFinSF1", "BsmtFinSF2", "BsmtUnfSF
    data[c].fillna(0, inplace=True)
```

- **MasVnrType, MasVnrArea:** Having an NA value likely means that there is no masonry veneer. Unfortunately, the data dictionary does not describe this. I would be wary of this variable and likely drop it in the future, or ensure that these variables have minimal weight in the linear regression model.

In [20]:
```
# qualitative variable
data["MasVnrType"].fillna("None", inplace=True)

# quantitative variable
data["MasVnrArea"].fillna(0, inplace=True)
```

- **MSZoning:** Identifies the general zoning classification of the sale. We can fill it in with the mode of this variable since it is qualitative. We see that "RL" is the most common value for this feature.

In [21]:
```
data["MSZoning"].mode()
```

Out[21]:
```
0    RL
dtype: object
```

In [22]:
```
data["MSZoning"].fillna(data["MSZoning"].mode()[0], inplace=True)
```

- **Utilities:** Type of utilities available. Data dictionary does not specify what NA values mean. For this column, we see below from the code that only 2 values are null values, and that the rest of the data (except one) is AllPub, meaning this column may not be useful in prediction. We could fill in the missing values with "AllPub" since it is the mode, but should be wary of this column and possibly drop it in the future.

In [23]:
```
data[["Utilities"]].isna().sum()
```

Out[23]:
```
Utilities    2
dtype: int64
```

In [24]:
```
data[["Utilities"]].value_counts()
```

Out[24]:
```
Utilities
AllPub       2916
NoSeWa          1
dtype: int64
```

In [25]:
```
data["Utilities"].fillna(data["Utilities"].mode()[0], inplace=True)
```

- **Functional:** According to data description, this is home functionality, and we should assume typical unless deductions are warranted. Thus, let's fill the null values with "Typ." We only have 2 NA values, but should not drop this column later since this feature takes many different values.

In [26]:
```
data[["Functional"]].isna().sum()
```

Out[26]:
```
Functional    2
dtype: int64
```

In [27]:
```
data[["Functional"]].value_counts()
```

Out[27]:
```
Functional
Typ           2717
Min2            70
Min1            65
Mod             35
Maj1            19
Maj2             9
Sev              2
dtype: int64
```

In [28]:
```
data["Functional"].fillna("Typ", inplace=True)
```

- **Exterior1st, Exterior2nd:** Exterior covering on house (2nd: if more than one material). This takes on many values, so let's fill it in with the mode. Note that we only have 1 missing value for each of these columns.

In [29]:
```
data[["Exterior1st"]].isna().sum()
```

Out[29]:
```
Exterior1st    1
dtype: int64
```

In [30]:
```
data[["Exterior1st"]].value_counts()
```

Out[30]:
```
Exterior1st
VinylSd        1025
MetalSd         450
HdBoard         442
Wd Sdng         411
Plywood         221
CemntBd         126
BrkFace          87
WdShing          56
AsbShng          44
Stucco           43
BrkComm           6
Stone             2
CBlock            2
AsphShn           2
ImStucc           1
dtype: int64
```

In [31]:
```
data[["Exterior2nd"]].isna().sum()
```

Out[31]:
```
Exterior2nd    1
dtype: int64
```

In [32]:
```
data[["Exterior2nd"]].value_counts()
```

Out[32]:
```
Exterior2nd
VinylSd        1014
MetalSd         447
HdBoard         406
Wd Sdng         391
Plywood         270
CmentBd         126
Wd Shng          81
Stucco           47
BrkFace          47
AsbShng          38
Brk Cmn          22
ImStucc          15
Stone             6
AsphShn           4
CBlock            3
Other             1
dtype: int64
```

In [33]:
```
data["Exterior1st"].fillna(data["Exterior1st"].mode()[0], inplace=True)
data["Exterior2nd"].fillna(data["Exterior2nd"].mode()[0], inplace=True)
```

- **SaleType:** Type of sale. We only have one missing value. Let's fill it in with the mode, which is "WD", meaning Warranty Deed - Conventional.

In [34]:
```
data[["SaleType"]].isna().sum()
```

Out[34]:
```
SaleType    1
dtype: int64
```

In [35]:
```
data[["SaleType"]].value_counts()
```

Out[35]:
```
SaleType
WD          2525
New          239
COD           87
ConLD         26
CWD           12
ConLI          9
ConLw          8
Oth            7
Con            5
dtype: int64
```

In [36]:
```
data["SaleType"].fillna(data["SaleType"].mode()[0], inplace=True)
```

- **Electrical:** Electrical system. Only one null value. We can fill it in with the mode, which is Sbrkr.

In [37]:
```
data[["Electrical"]].isna().sum()
```

Out[37]:
```
Electrical    1
dtype: int64
```

In [38]:
```
data[["Electrical"]].value_counts()
```

Out[38]:
```
Electrical
SBrkr         2671
FuseA          188
FuseF           50
FuseP            8
Mix              1
dtype: int64
```

In [39]:
```
data["Electrical"].fillna(data["Electrical"].mode()[0], inplace=True)
```

- **KitchenQual:** Kitchen quality. Only one null value. Can fill in with the mode, which is "Typical/Average".

In [40]:
```
data[["KitchenQual"]].isna().sum()
```

Out[40]:
```
KitchenQual    1
dtype: int64
```

In [41]:
```
data[["KitchenQual"]].value_counts()
```

Out[41]:
```
KitchenQual
TA             1492
Gd             1151
Ex              205
Fa               70
dtype: int64
```

In [42]:
```
data["KitchenQual"].fillna(data["KitchenQual"].mode()[0], inplace=True)
```

Finally, let's double check that we have no missing data.

In [43]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2919 entries, 0 to 1458
Data columns (total 79 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   MSSubClass     2919 non-null   int64  
 1   MSZoning       2919 non-null   object 
 2   LotFrontage    2919 non-null   float64
 3   LotArea        2919 non-null   int64  
 4   Street         2919 non-null   object 
 5   Alley          2919 non-null   object 
 6   LotShape       2919 non-null   object 
 7   LandContour    2919 non-null   object 
 8   Utilities      2919 non-null   object 
 9   LotConfig      2919 non-null   object 
 10  LandSlope      2919 non-null   object 
 11  Neighborhood   2919 non-null   object 
 12  Condition1     2919 non-null   object 
 13  Condition2     2919 non-null   object 
 14  BldgType       2919 non-null   object 
 15  HouseStyle     2919 non-null   object 
 16  OverallQual    2919 non-null   int64  
 17  OverallCond    2919 non-null   int64  
 18  YearBuilt      2919 non-null   int64  
 19  YearRemodAdd   2919 non-null   int64  
 20  RoofStyle      2919 non-null   object 
 21  RoofMatl       2919 non-null   object 
 22  Exterior1st    2919 non-null   object 
 23  Exterior2nd    2919 non-null   object 
 24  MasVnrType     2919 non-null   object 
 25  MasVnrArea     2919 non-null   float64
 26  ExterQual      2919 non-null   object 
 27  ExterCond      2919 non-null   object 
 28  Foundation     2919 non-null   object 
 29  BsmtQual       2919 non-null   object 
 30  BsmtCond       2919 non-null   object 
 31  BsmtExposure   2919 non-null   object 
 32  BsmtFinType1   2919 non-null   object 
 33  BsmtFinSF1     2919 non-null   float64
 34  BsmtFinType2   2919 non-null   object 
 35  BsmtFinSF2     2919 non-null   float64
 36  BsmtUnfSF      2919 non-null   float64
 37  TotalBsmtSF    2919 non-null   float64
 38  Heating        2919 non-null   object 
 39  HeatingQC      2919 non-null   object 
 40  CentralAir     2919 non-null   object 
 41  Electrical     2919 non-null   object 
 42  1stFlrSF       2919 non-null   int64  
 43  2ndFlrSF       2919 non-null   int64  
 44  LowQualFinSF   2919 non-null   int64  
 45  GrLivArea      2919 non-null   int64  
 46  BsmtFullBath   2919 non-null   float64
 47  BsmtHalfBath   2919 non-null   float64
 48  FullBath       2919 non-null   int64  
 49  HalfBath       2919 non-null   int64  
 50  BedroomAbvGr   2919 non-null   int64  
 51  KitchenAbvGr   2919 non-null   int64  
 52  KitchenQual    2919 non-null   object 
 53  TotRmsAbvGrd   2919 non-null   int64  
 54  Functional     2919 non-null   object 
```

```
 55  Fireplaces     2919 non-null   int64
 56  FireplaceQu    2919 non-null   object
 57  GarageType     2919 non-null   object
 58  GarageYrBlt    2919 non-null   float64
 59  GarageFinish   2919 non-null   object
 60  GarageCars     2919 non-null   float64
 61  GarageArea     2919 non-null   float64
 62  GarageQual     2919 non-null   object
 63  GarageCond     2919 non-null   object
 64  PavedDrive     2919 non-null   object
 65  WoodDeckSF     2919 non-null   int64
 66  OpenPorchSF    2919 non-null   int64
 67  EnclosedPorch  2919 non-null   int64
 68  3SsnPorch      2919 non-null   int64
 69  ScreenPorch    2919 non-null   int64
 70  PoolArea       2919 non-null   int64
 71  PoolQC         2919 non-null   object
 72  Fence          2919 non-null   object
 73  MiscFeature    2919 non-null   object
 74  MiscVal        2919 non-null   int64
 75  MoSold         2919 non-null   int64
 76  YrSold         2919 non-null   int64
 77  SaleType       2919 non-null   object
 78  SaleCondition  2919 non-null   object
dtypes: float64(11), int64(25), object(43)
memory usage: 1.8+ MB
```

Success!

Something else that concerned me from the last EDA notebook is that the distribution of `SalePrice` is positively skewed. We discussed in class that we should apply some transformation to this variable because linear models assume that the target variable is also linear.

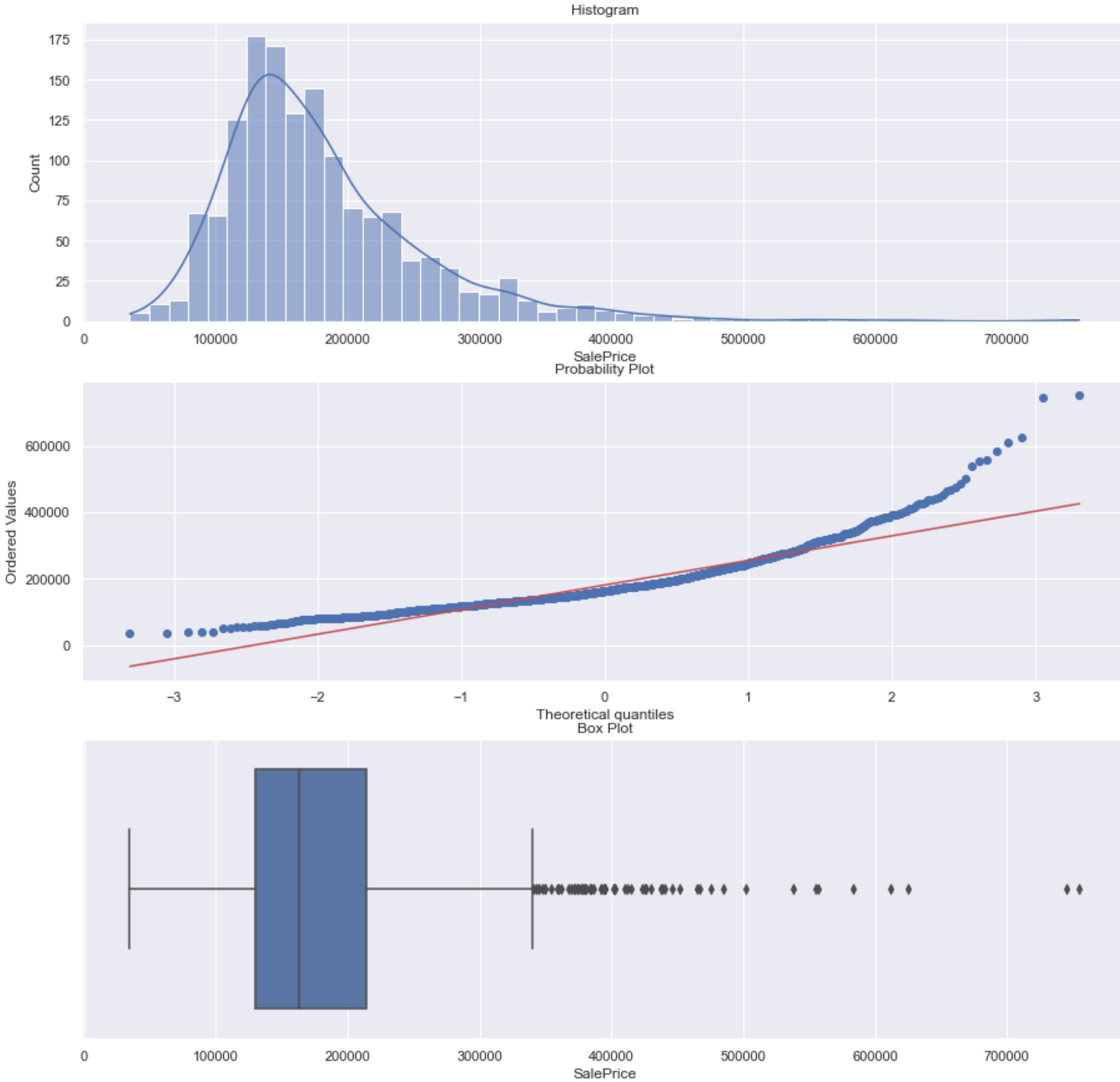
In [44]:

```
# stealing this from a YouTube video (will link below)
def plotting_3_chart(df, feature):
    fig, axes = plt.subplots(3, 1, figsize=(15,15))

    # customizing histogram grid
    # set title
    axes[0].set_title("Histogram")
    # plot histogram
    sns.histplot(x=feature, data=df, kde=True, ax=axes[0])

    # customizing qq plot
    # set title
    axes[1].set_title("QQ Plot")
    # plot qq plot
    stats.probplot(x=df.loc[:, feature], plot=axes[1])

    # customizing box plot
    # set title
    axes[2].set_title("Box Plot")
    # plot box plot
    sns.boxplot(x=feature, data=df, ax=axes[2])
```

In [45]:

```
plotting_3_chart(train, "SalePrice")

# skewness and kurtosis
print(f"Skewness: {train['SalePrice'].skew()}")
print(f"Kurtosis: {train['SalePrice'].kurt()}")
```

```
Skewness: 1.8828757597682129
Kurtosis: 6.536281860064529
```

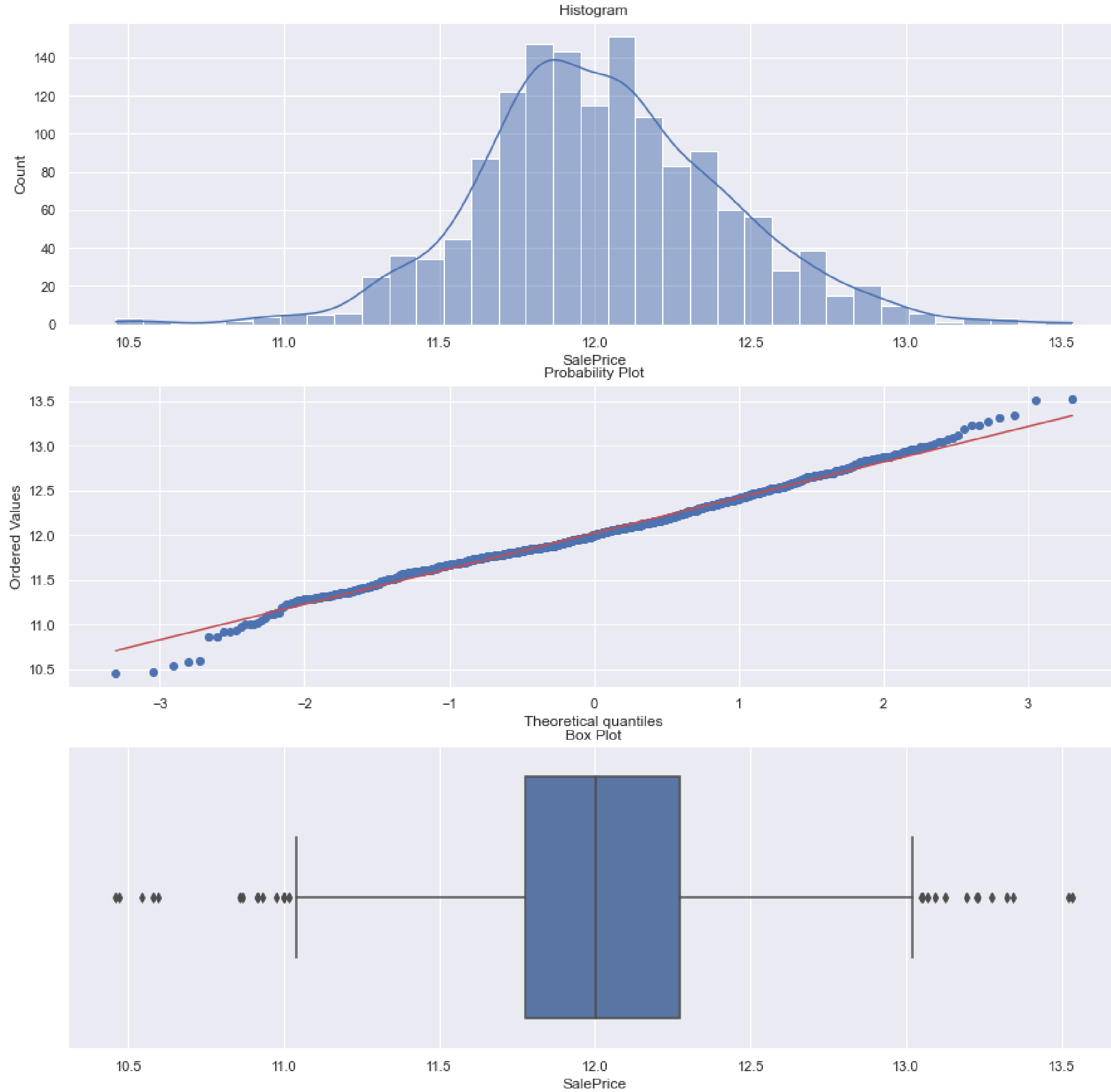


The high value of skewness confirms that `SalePrice` is positively skewed. The high value of kurtosis indicates that the data is very peaked around the mean/median. A more normal distribution would have a kurtosis value of about 3, but our kurtosis value is doubled. Not good!

We can also see that the probability plot is curved, suggesting nonlinearity. A log transformation will help us in this situation to create a more normal distribution.

In [46]:
```python
# store log of target variable of training data in a safe place
sale_price_train_log = np.log1p(train['SalePrice'])

train["SalePrice"] = sale_price_train_log

# plot
plotting_3_chart(train, "SalePrice")

# skewness and kurtosis
print(f"Skewness: {train['SalePrice'].skew()}")
print(f"Kurtosis: {train['SalePrice'].kurt()}")
```

```
Skewness: 0.12134661989685329
Kurtosis: 0.809519155707878
```



Insights: We see that `SalePriceLog` has become more normalized. The lower values of skewness and kurtosis indicate this, as well as the probability plot. From the box plot, we still see some outliers. We may or may not need to drop these outliers, but for now, I don't think it's a good idea because we have multiple outliers that can indicate some uniqueness for those homes, which need to receive a proper and accurate prediction in case the test set also includes outliers such as these.

Feature Engineering

Based on `data.info()` from the previous cell of code that was run, some features should not be integers, such as "MSSubClass". Let's fix that by turning variables into a string.

In [47]:
```
int_vars_to_string = ["MSSubClass", "OverallQual", "OverallCond", "YearBuilt", "
                      "YrSold", "MoSold"]
for c in int_vars_to_string:
    data[c] = data[c].astype(str)
```

Let's combine some features to remove features that may be collinear, such as a value of `TotalSQFT` that is just the SQFT of basement, 1st floor, and 2nd floor combined.

In [48]:
```
# total sqft column
data["TotalSF"] = data["TotalBsmtSF"] + data["1stFlrSF"] + data["2ndFlrSF"]

# concatenating string Year values to create uniqueness and later bin
# got this idea from a YouTube video, but I find it useless
# data["YrBuiltRemod"] = data["YearBuilt"] + data["YearRemodAdd"]

# total number of bathrooms
data["TotalBathrooms"] = data["FullBath"] + data["HalfBath"] + data["BsmtFullBat
```

Before using `pd.get_dummies()`, we should create dummy variables of our own for certain features, such as whether or not a house has a pool, garage, 2nd floor, etc. like I did in the EDA notebook.

In [49]:
```
data["Has2ndFlr"] = data[["2ndFlrSF"]].apply(lambda x: x > 0)
data["HasBsmt"] = data[["TotalBsmtSF"]].apply(lambda x: x > 0)
data["HasPool"] = data[["PoolArea"]].apply(lambda x: x > 0)
data["HasGarage"] = data[["GarageArea"]].apply(lambda x: x > 0)
data["HasFireplace"] = data[["Fireplaces"]].apply(lambda x: x > 0)
```

Now, let's use `pd.get_dummies()` for categorical columns.

In [50]:
```
data = pd.get_dummies(data, drop_first=True).reset_index(drop=True)
data.head()
```

Out[50]:

	LotFrontage	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	1stFlrS
0	65.0	8450	196.0	706.0	0.0	150.0	856.0	85
1	80.0	9600	0.0	978.0	0.0	284.0	1262.0	126
2	68.0	11250	162.0	486.0	0.0	434.0	920.0	92
3	60.0	9550	0.0	216.0	0.0	540.0	756.0	96
4	84.0	14260	350.0	655.0	0.0	490.0	1145.0	114

5 rows × 585 columns

We can see that we created way more columns than we were originally given, and this can be solved through Principal Component Analysis. Although this is beyond the scope of this module, I will still choose to implement it since I am a big fan of this technique.

Notice I didn't remove any columns yet for which I created my own dummy variables, such as `HasGarage`. I will let PCA tell me which features to keep for our final `X` matrix.

Principal Component Analysis

We can use `scikit-learn` to perform PCA. The first step is to scale the data using `StandardScaler` because it is a requirement for PCA. Before this step, we should resplit our data into train and test data.

In [51]:
```
X_train = data[:len(train)]
```

```
X_test = data[len(train):]
y = train[["SalePrice"]]
```

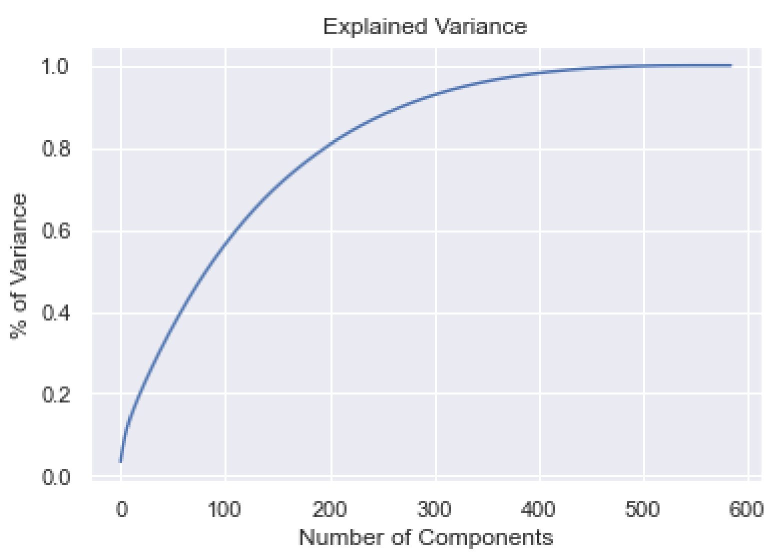
In [52]:
```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_train
```

Out[52]:
```
array([[-0.23082236, -0.20714171,  0.51410389, ..., -0.11785113,
         0.4676514 , -0.30599503],
       [ 0.4380509 , -0.09188637, -0.57075013, ..., -0.11785113,
         0.4676514 , -0.30599503],
       [-0.09704771,  0.07347998,  0.32591493, ..., -0.11785113,
         0.4676514 , -0.30599503],
       ...,
       [-0.18623081, -0.14781027, -0.57075013, ..., -0.11785113,
         0.4676514 , -0.30599503],
       [-0.09704771, -0.08016039, -0.57075013, ..., -0.11785113,
         0.4676514 , -0.30599503],
       [ 0.21509315, -0.05811155, -0.57075013, ..., -0.11785113,
         0.4676514 , -0.30599503]])
```

Below, we are initializing the PCA tool from `scikit-learn`. The "explained_variance" variable is to be used to create a Pareto chart. We will choose the number of features at the elbow that explains the majority of the variance for our final `X`.

In [53]:
```
pca = PCA(n_components=len(data.columns.tolist()))
X_train = pca.fit_transform(X_train)
explained_variance = pca.explained_variance_ratio_
```

In [54]:
```
plt.figure()
plt.plot(np.cumsum(explained_variance))
plt.xlabel("Number of Components")
plt.ylabel("% of Variance")
plt.title("Explained Variance")
plt.show()
```



Insights: We can see that with about 300 features, about 90% of the data is explained. Amazing! We can probably even do fewer since the goal is to achieve at least 80% of the variance explained.

Now we need to figure out which of these columns are the ones contributing the most to

SalePrice .

In [55]:
```
# the number of features I think we should include in our regression
# this is about 85% of the data
# alternatively, we could try 300
pca_num = 250

pca_df = pd.concat([pd.DataFrame(data={"Features": data.columns.tolist()}),
                    pd.DataFrame(data={"Variance Squared": explained_variance**2}
                    axis=1).sort_values(by="Variance Squared", ascending=False)[0
pca_df
```

Out[55]:

	Features	Variance Squared
0	LotFrontage	0.001192
1	LotArea	0.000329
2	MasVnrArea	0.000245
3	BsmtFinSF1	0.000203
4	BsmtFinSF2	0.000165
...	...	...
245	YearBuilt_2002	0.000002
246	YearBuilt_2003	0.000002
247	YearBuilt_2004	0.000002
248	YearBuilt_2005	0.000001
249	YearBuilt_2006	0.000001

250 rows × 2 columns

In [56]:
```
# How much of the variance we are explaining with pca_df
np.sqrt(pca_df["Variance Squared"]).sum()
```

Out[56]: 0.8773580373918672

Let's make X now only contain the features that are in pca_df .

In [57]:
```
pca_features = pca_df["Features"].tolist()
X_new = data.loc[:, pca_features]
X_new.head()
```

Out[57]:

	LotFrontage	LotArea	MasVnrArea	BsmtFinSF1	BsmtFinSF2	BsmtUnfSF	TotalBsmtSF	1stFlrS
0	65.0	8450	196.0	706.0	0.0	150.0	856.0	85
1	80.0	9600	0.0	978.0	0.0	284.0	1262.0	126
2	68.0	11250	162.0	486.0	0.0	434.0	920.0	92
3	60.0	9550	0.0	216.0	0.0	540.0	756.0	96
4	84.0	14260	350.0	655.0	0.0	490.0	1145.0	114

5 rows × 250 columns

We can resplit the data with the proper features that we can now model with.

In [58]:
```
# our edited train set
X = X_new[:len(train)]

# our edited test set
test2 = X_new[len(train):]
```

First ML Model: Ridge Regression

The first thing we should do before building our regression model is splitting our `X` dataframe. Notice that `X` is just our training set that has been cleaned and feature engineered. We want to split our training set into a train and validation set so that we are able to test our model on that validation set before submitting to Kaggle.

Ridge Regression

Let's use Ridge Regression first. The reason we want to do this is because it is better to model data that may suffer from collinearity. We saw that some variables may very well depend on one another (such as "TotalSF").

To build a strong Ridge Regression model, we can use `GridSearchCV` to determine what is the best alpha value for ridge.

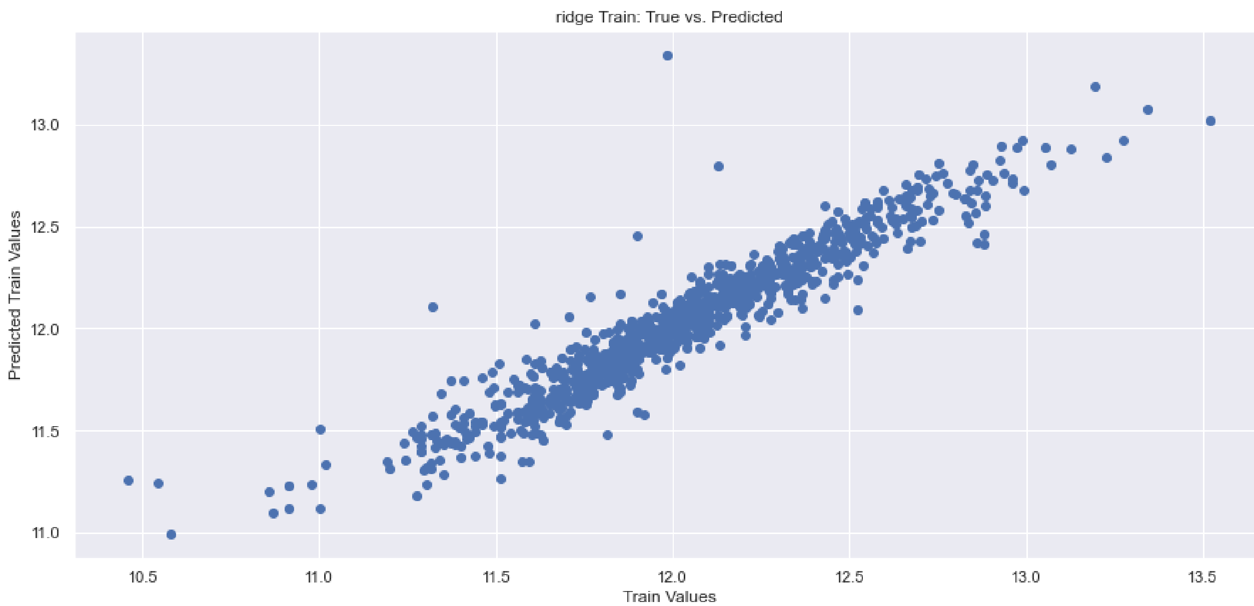
In [59]:
```
def create_model(X, y, test_set, model, model_type, param_grid):
    """
    This creates the best model given a basic model, such as Ridge, Lasso, or El
    parameter grid, making reproducibility easy.

    Inputs:
        X: our feature matrix
        y: our predictor vector, the log of "SalePrice"
        test_set: our test matrix that we wish to predict on, containing the sam
        model: an sklearn model, set with random_state=42
        model_type: str, one of ["ridge", "lasso", "lasso-unselected-feats", "el
    """
    ## SETUP
    # Only allowing 30% of the data go into testing
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ran

    print("Train shapes:")
    print(f"X_train: {X_train.shape}")
    print(f"y_train: {y_train.shape}")

    print("\nTest shapes:")
    print(f"X_test: {X_test.shape}")
    print(f"y_test: {y_test.shape}")

    # Instantiate the GridSearchCV object: model_cv
    model_cv = GridSearchCV(model, param_grid=param_grid, cv=5)

    # Fit it to the data
    model_cv.fit(X_train, y_train)

    # Print the tuned parameter and score
    print(f"\nTuned {model_type} Regression Parameters: {model_cv.best_params_}"
```

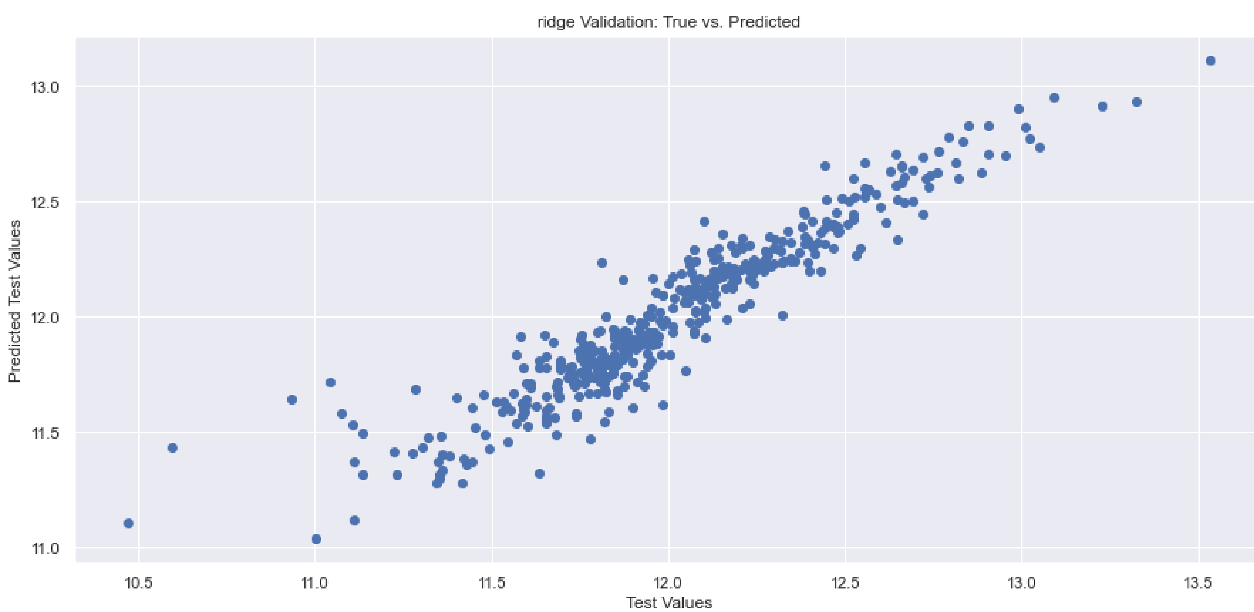
```
    print("Best score is {}".format(model_cv.best_score_))

    ## EVALUATION
    y_train_pred = model_cv.predict(X_train)
    y_test_pred = model_cv.predict(X_test)

    fig, axes = plt.subplots(2, 1, figsize=(15,15))
    # visualizing how well we predicted on training
    axes[0].scatter(y_train, y_train_pred)
    axes[0].set_title(f"{model_type} Train: True vs. Predicted")
    axes[0].set_xlabel("Train Values")
    axes[0].set_ylabel("Predicted Train Values")

    axes[1].scatter(y_test, y_test_pred)
    axes[1].set_title(f"{model_type} Validation: True vs. Predicted")
    axes[1].set_xlabel("Test Values")
    axes[1].set_ylabel("Predicted Test Values")

    plt.show();

    # printing metrics information
    print(f"\n{model_type} Mean Absolute Error: {metrics.mean_absolute_error(y_t
    print(f"{model_type} Mean Squared Error: {metrics.mean_squared_error(y_test,
    print(f"{model_type} Root Mean Squared Error: {np.sqrt(metrics.mean_squared_
    print((f"** {model_type} Root Mean Squared Logarithmic \
    Error **: {np.sqrt(metrics.mean_squared_log_error(y_test, y_test_pred))}"))

    ## TEST PREDICTIONS
    # creating our test submissions and a csv file to submit to Kaggle
    # First, we have to fit to our entire training data, and then predict on our
    # We also need to change the SalePrice value back to its original, non-logar
    test_submit = 0
    if model_type == "ridge":
        model_cv.fit(X, y)
        test_pred = model_cv.predict(test_set)
        test_pred = np.expm1(test_pred)
        test_pred2 = []
        for i in np.arange(len(test_pred)):
            test_pred2.append(test_pred[i][0])

        assert len(test_pred2) == len(test_set)
        test_submit = pd.DataFrame(data={"Id": test_id.tolist(), "SalePrice": te
        print()
        print(test_submit.head())
    else:
        model_cv.fit(X, y)
        test_pred = model_cv.predict(test_set)
        test_pred = np.expm1(test_pred)

        assert len(test_pred) == len(test)
        test_submit = pd.DataFrame(data={"Id": test_id.tolist(), "SalePrice": te
        print()
        print(test_submit.head())

    test_submit.to_csv(f"{model_type}-regression.csv", index=False)
```

In [60]:
```
# Setup the hyperparameter grid
alph = np.arange(0.5, 21.5, 0.5)
fit_intercept = np.array([True, False])
normalize = np.array([True, False])
```

```
param_grid = {'alpha': alph, "fit_intercept": fit_intercept, "normalize": normal

create_model(X, y, test2, model=Ridge(random_state=42), model_type="ridge", para
```

```
Train shapes:
X_train: (1022, 250)
y_train: (1022, 1)

Test shapes:
X_test: (438, 250)
y_test: (438, 1)

Tuned ridge Regression Parameters: {'alpha': 0.5, 'fit_intercept': True, 'normal
ize': True}
Best score is 0.8150276235262528
```






```
ridge Mean Absolute Error: 0.0958241890848923
ridge Mean Squared Error: 0.01980210060140901
ridge Root Mean Squared Error: 0.14071993675882963
** ridge Root Mean Squared Logarithmic     Error **: 0.011012510823137187

     Id      SalePrice
0  1461   111755.612754
1  1462   150376.645546
```

```
2  1463  176656.390217
3  1464  194156.554063
4  1465  202925.747502
```

Notice that Kaggle looks at the Root Mean Squared Logarithmic Error, but I also wanted to print out other metrics.

We can see that we did pretty well when predicting the training set, except for a few outliers. We still did very well on the test set as well. We see that really low values are not doing the best at predictions.

Seems like we have a very low error! Great, let's use this for predictions and submit to Kaggle.

What score does this give us on Kaggle? Remember, the smaller the better!

- **0.15071**

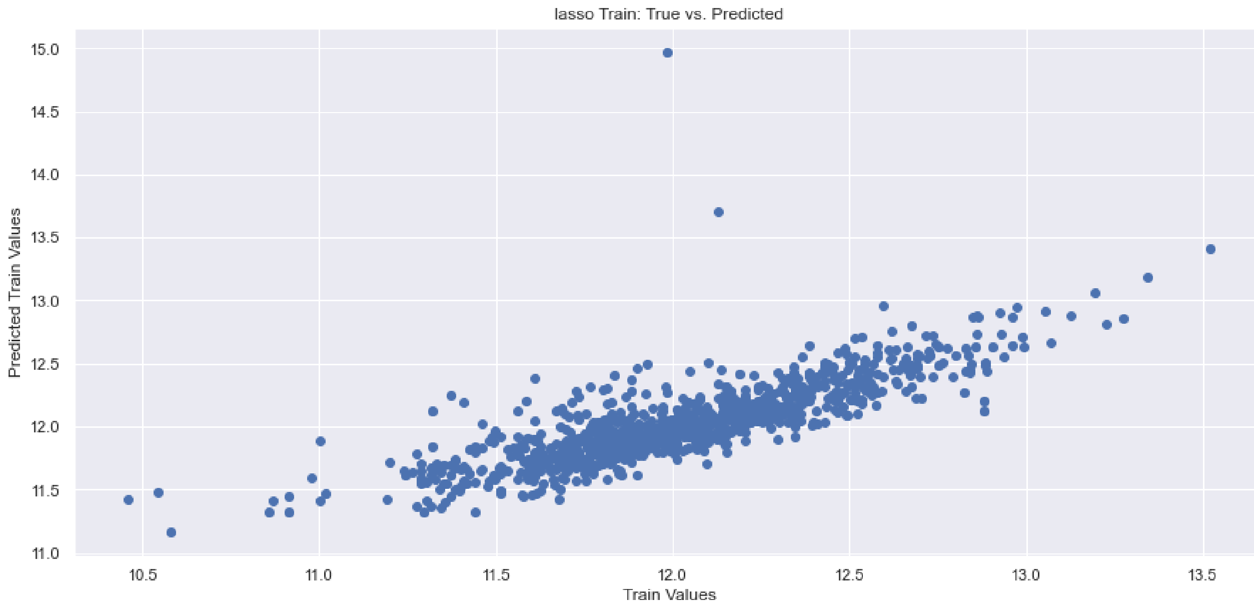
Second ML Model: Lasso Regression

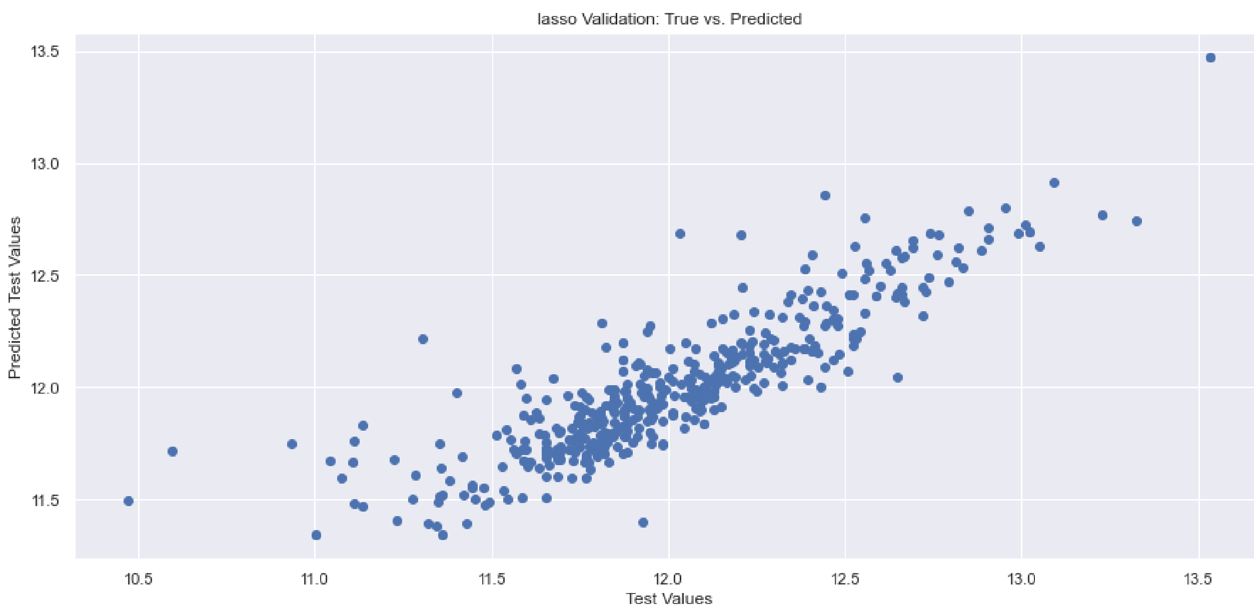
Lasso regression makes some coefficients exactly 0. This may or may not be useful for this problem, but I wanted to try it out to see how it compares to Ridge. This automatically does feature selection, so I will try this with all features and with selected features from Ridge.

Lasso Regression: Selected Features

In [61]:
```python
# had to increase max_iter from the default 1000 to 15000 because it would not c
create_model(X, y, test2, model=Lasso(random_state=42, max_iter=15000), model_ty
```

```
Train shapes:
X_train: (1022, 250)
y_train: (1022, 1)

Test shapes:
X_test: (438, 250)
y_test: (438, 1)

Tuned lasso Regression Parameters: {'alpha': 1.5, 'fit_intercept': True, 'normal
ize': False}
Best score is 0.6055893302155446
```






```
lasso Mean Absolute Error: 0.14840523757181062
lasso Mean Squared Error: 0.043479491417027165
lasso Root Mean Squared Error: 0.20851736478535107
** lasso Root Mean Squared Logarithmic     Error **: 0.016223135267555495

     Id      SalePrice
0  1461  156078.654239
1  1462  164235.100737
2  1463  180323.280586
3  1464  183161.804126
4  1465  162183.214831
```

Insights: Clearly, Lasso is not doing very well. The score is much lower than Ridge and the Root Mean Squared Logarithmic Error is also higher than Ridge. It seems like Lasso performed well on the test set, but not very well on the training set. I'll submit this to Kaggle just to compare.

What score does this give us on Kaggle? Remember, the smaller the better!

- **0.22698**
  We clearly did worse! Lasso is not the way to go in this situation.

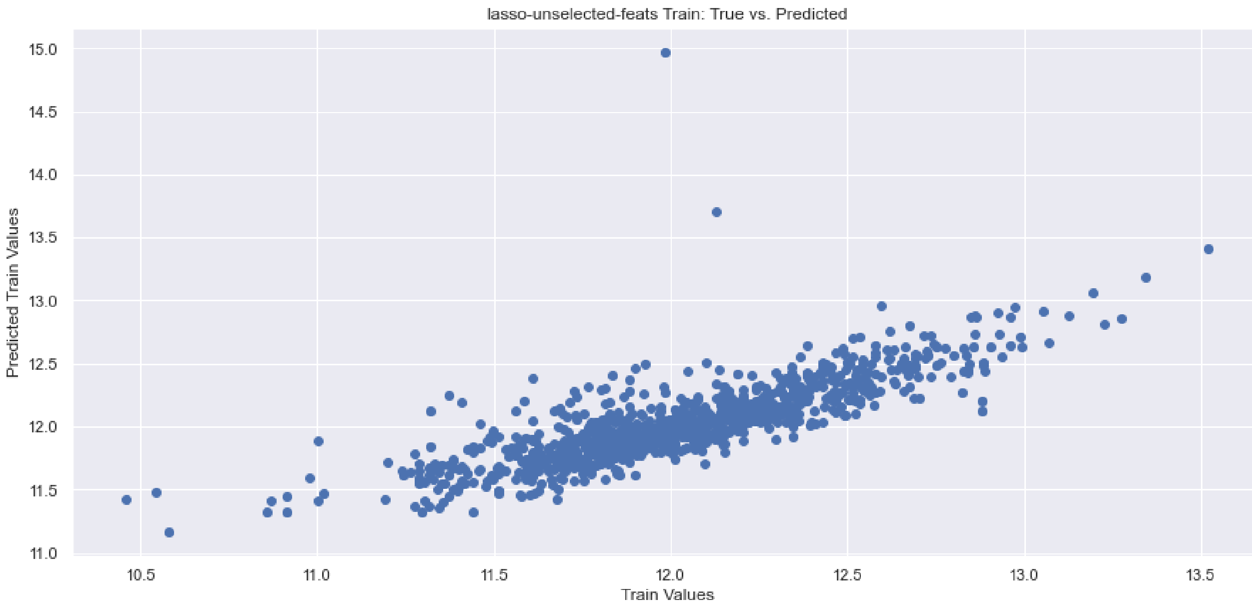
Lasso Regression: Unselected Features

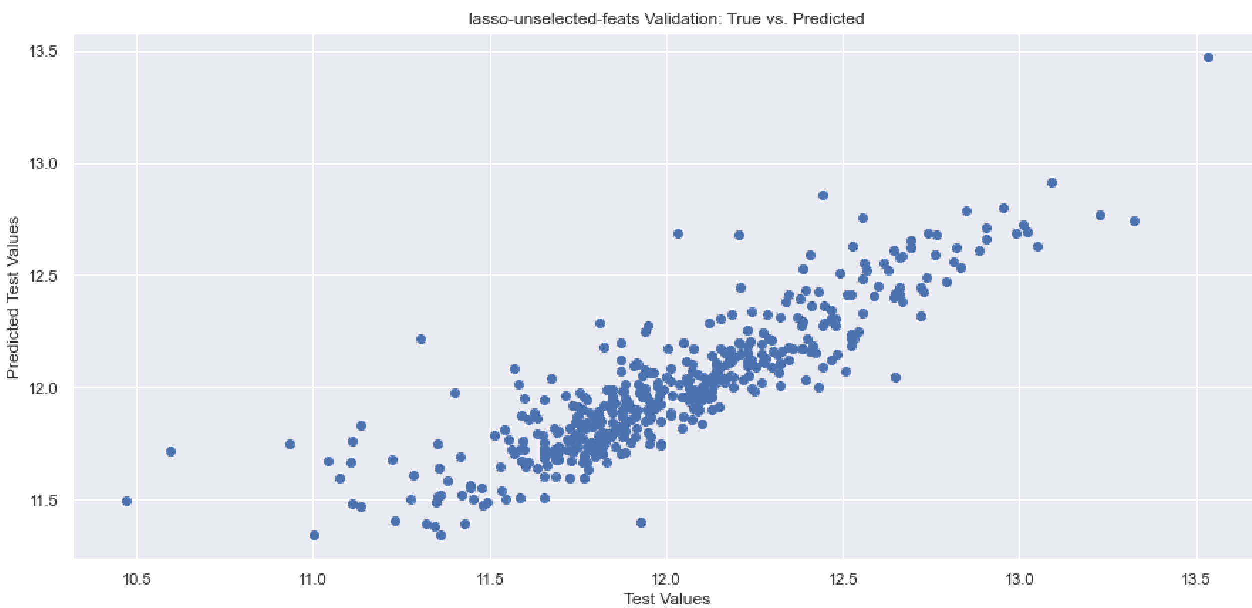
Now, we will give the model all of our features, since Lasso will give weights of 0 to features that are less important.

In [62]:

```
# train and test set with all features
X_lasso = data[:len(train)]
test_lasso = data[len(train):]

# had to increase max_iter from the default 1000 to 15000 because it would not c
create_model(X_lasso, y, test_lasso, model=Lasso(random_state=42, max_iter=15000
             model_type="lasso-unselected-feats", param_grid=param_grid)
```

```
Train shapes:
X_train: (1022, 585)
y_train: (1022, 1)

Test shapes:
X_test: (438, 585)
y_test: (438, 1)

Tuned lasso-unselected-feats Regression Parameters: {'alpha': 1.5, 'fit_intercep
t': True, 'normalize': False}
Best score is 0.6055893302155446
```






```
lasso-unselected-feats Mean Absolute Error: 0.14840523757181062
lasso-unselected-feats Mean Squared Error: 0.043479491417027165
lasso-unselected-feats Root Mean Squared Error: 0.20851736478535107
** lasso-unselected-feats Root Mean Squared Logarithmic     Error **: 0.01622313
5267555495

     Id      SalePrice
0  1461  156078.654239
1  1462  164235.100737
2  1463  180323.280586
3  1464  183161.804126
4  1465  162183.214831
```

Insights: This Lasso Regression model performed just as poorly as the previous Lasso model that had the selected features matrix. Lasso is not the best model for this data set.

What score does this give us on Kaggle? Remember, the smaller the better!

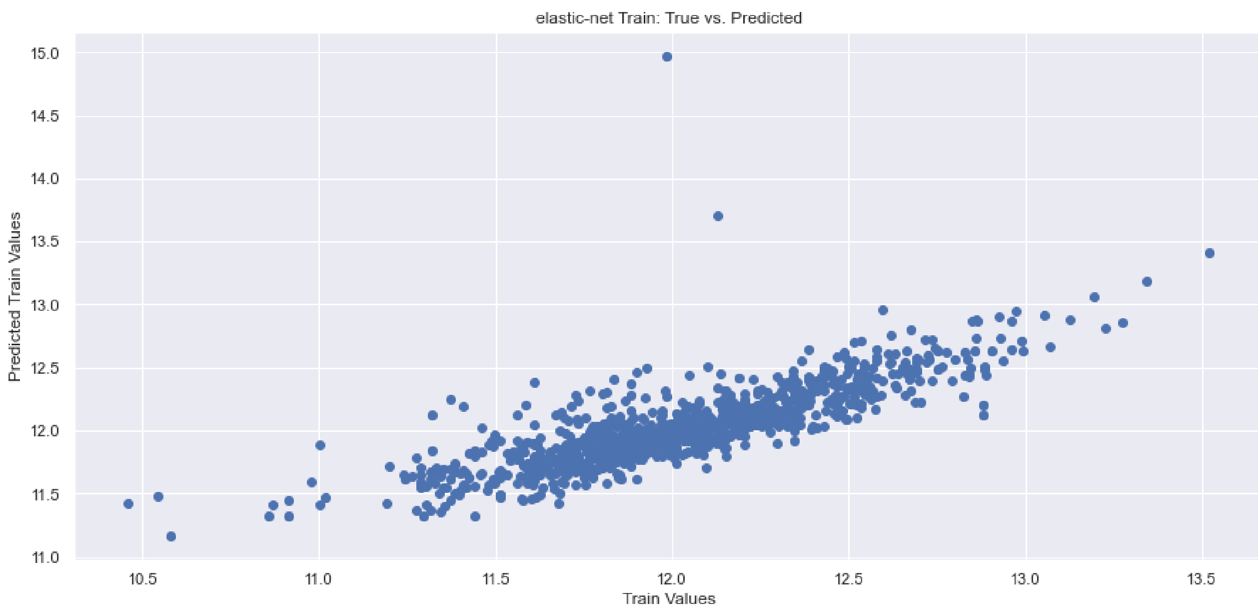
- **0.22698**
  This is the same as the other Lasso model (with selected features). It's best to not use Lasso.

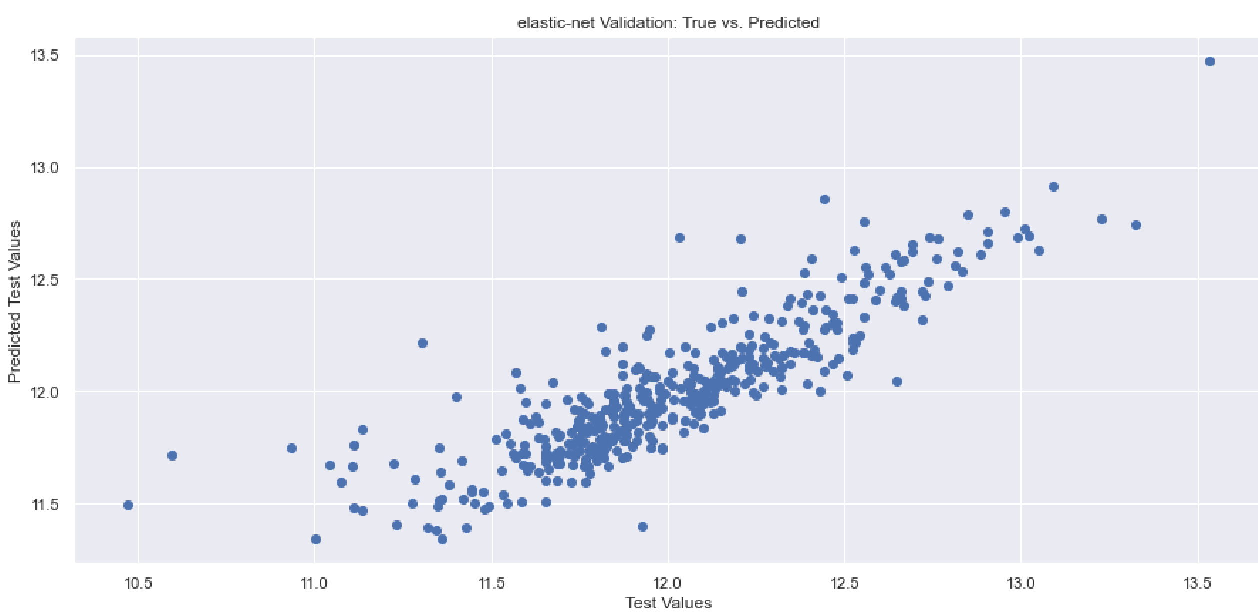
Third ML Model: Elastic Net Regression

Elastic Net is a combination of Lasso and Ridge, so maybe this could potentially make our model more accurate. Let's give it a try. First, we will ensure that we are using `X` with selected features from PCA.

In [63]:
```
# had to increase max_iter from the default 1000 to 20000 because it would not c
create_model(X, y, test2, model=ElasticNet(random_state=42, max_iter=25000), mod
             param_grid=param_grid)
```

```
Train shapes:
X_train: (1022, 250)
y_train: (1022, 1)

Test shapes:
X_test: (438, 250)
y_test: (438, 1)

Tuned elastic-net Regression Parameters: {'alpha': 3.0, 'fit_intercept': True,
'normalize': False}
Best score is 0.6055896785398815
```





```
elastic-net Mean Absolute Error: 0.14840539374477094
elastic-net Mean Squared Error: 0.04347943791693427
elastic-net Root Mean Squared Error: 0.208517236498411
** elastic-net Root Mean Squared Logarithmic     Error **: 0.016223128470221268

     Id      SalePrice
0  1461  157358.570289
1  1462  163691.752849
2  1463  180758.206656
3  1464  184185.042352
4  1465  164028.614354
```

What score does this give us on Kaggle? Remember, the smaller the better!

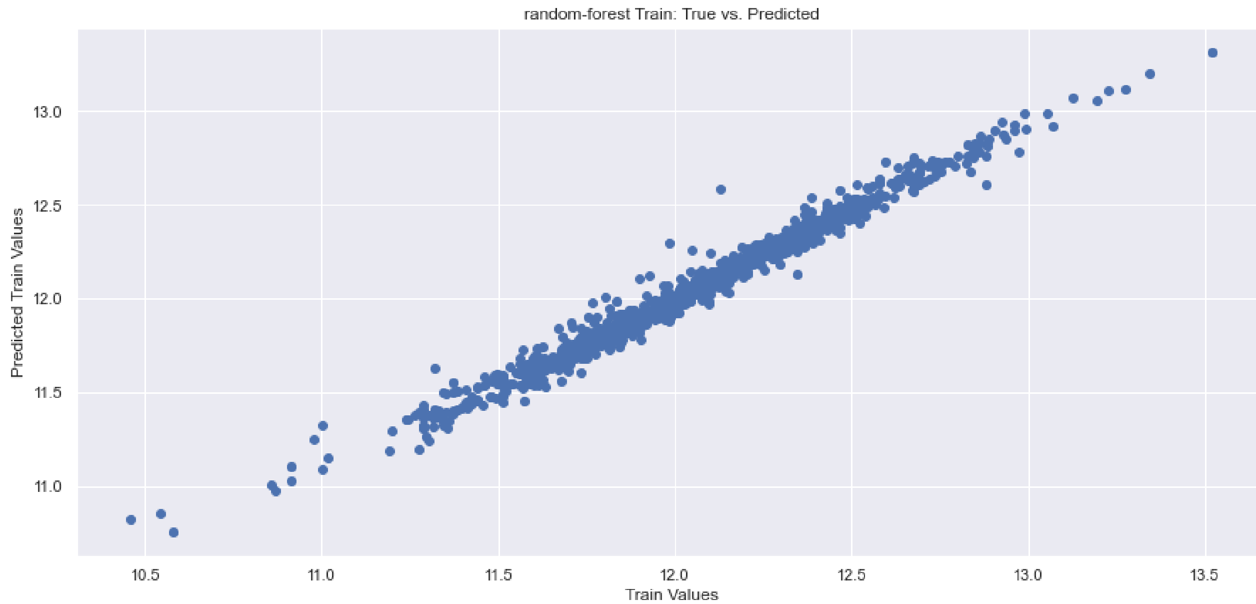
- **0.22668**
  ElasticNet did better than Lasso, but still not good compared to Ridge.

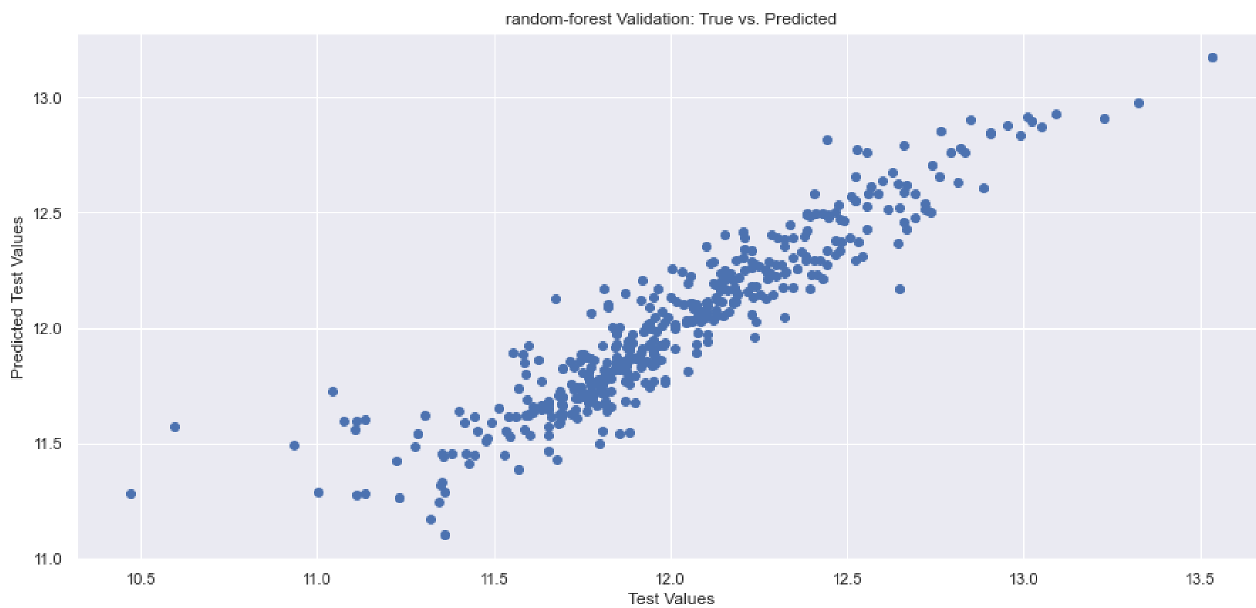
Fourth ML Model: Random Forest Regression

I want to try a random forest regression because it not only predicts future values very well, but it also tends to be extremely accurate. Like the other models we have created, we will find the best parameters through GridSearchCV.

In [64]:
```
# Setup the hyperparameter grid for random forest regression
n_est = np.arange(10, 150)
#max_dep = np.arange(1, 100)
# "max_depth": max_dep}
param_grid = {'n_estimators': n_est}

# create the model
create_model(X, np.array(y["SalePrice"].tolist()), test2, model=RandomForestRegr
             model_type="random-forest", param_grid=param_grid)
```

```
Train shapes:
X_train: (1022, 250)
y_train: (1022,)

Test shapes:
X_test: (438, 250)
y_test: (438,)

Tuned random-forest Regression Parameters: {'n_estimators': 95}
Best score is 0.8306122576272077
```






```
random-forest Mean Absolute Error: 0.10624430804832863
random-forest Mean Squared Error: 0.023784709621477707
random-forest Root Mean Squared Error: 0.15422292184198078
** random-forest Root Mean Squared Logarithmic     Error **: 0.012093135771170 36
```

```
     Id      SalePrice
0   1461   122129.171871
1   1462   158688.561608
2   1463   182553.161686
3   1464   180482.728116
4   1465   194609.254689
```

Insights: We see that Random Forest is predicting the outliers in a better way based on the scatter plot, but our Root Mean Squared Logarithmic Error is somehow higher. Maybe because it is not predicting low test values well.

What score does this give us on Kaggle? Remember, the smaller the better!

- **0.16559**
  Although this is much better than Lasso and ElasticNet, this is still not better than Ridge regression. Maybe we could select more features, which I will try below.

Random Forest Take 2: More Features

Previously, I selected 250 features for our feature matrix X because these many features explained roughly 85% of the data. Maybe we should instead select 300 or 350 to achieve a higher accuracy. I must caution that this may cause overfitting, but we'll see what happens!

In [65]:
```
def create_feature_matrix(pca_num):
    """
    Creates our feature matrix for our regression models based on how many featu

    Inputs:
        pca_num: int, how many features we want in our feature matrix

    Outputs:
        X: our feature matrix for training our models
        test2: a second version of our test set that only contains the same feat
    """
    # creating a df sorted by variance explained squared to select our top pca_n
    pca_df = pd.concat([pd.DataFrame(data={"Features": data.columns.tolist()}),
                        pd.DataFrame(data={"Variance Squared": explained_variance**2}
                        axis=1).sort_values(by="Variance Squared", ascending=False)[0

    print()
    print(pca_df.head())

    # How much of the variance we are explaining with pca_df
    var_squared = np.sqrt(pca_df["Variance Squared"]).sum()
    print(f"\nVariance explained: {var_squared}")

    # creating the feature matrix based on top pca_num features that explain the
    pca_features = pca_df["Features"].tolist()
    X_new = data.loc[:, pca_features]

    ## splitting our edited train and test sets
    # our edited train set
    X = X_new[:len(train)]

    # our edited test set
    test2 = X_new[len(train):]

    return (X, test2)
```

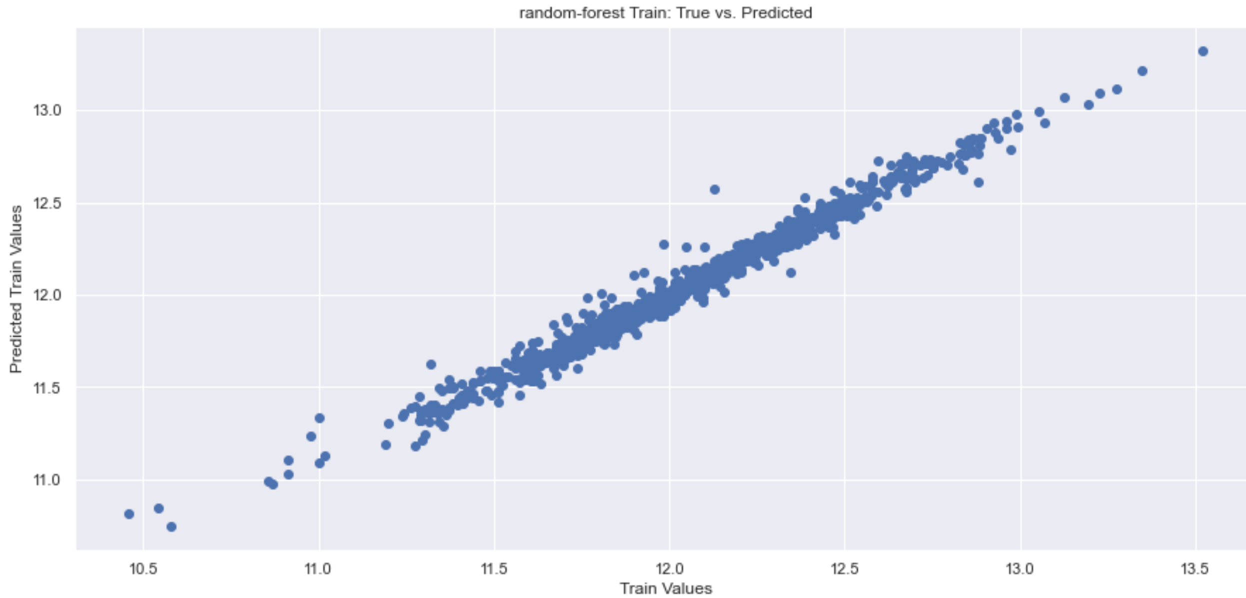
In [66]:
```
X, test2 = create_feature_matrix(300)
```

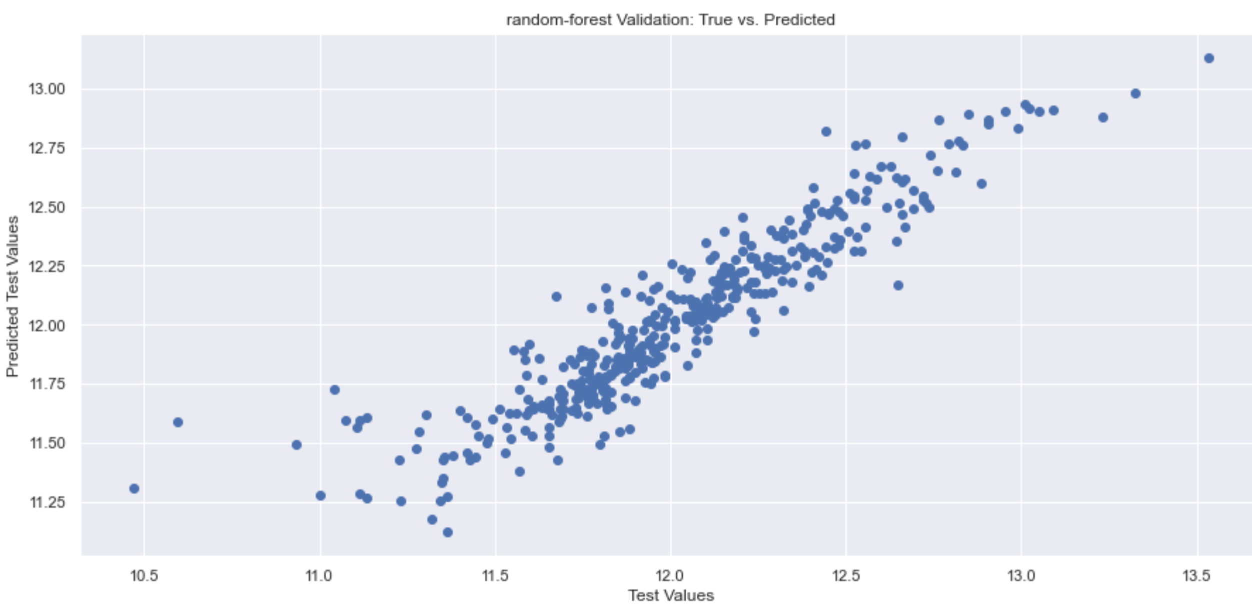
```
      Features  Variance Squared
0  LotFrontage          0.001192
1      LotArea          0.000329
2   MasVnrArea          0.000245
3   BsmtFinSF1          0.000203
4   BsmtFinSF2          0.000165

Variance explained: 0.9272772826385483
```

In [67]:
```
# create the model with same param_grid from the last random forest regressor
create_model(X, np.array(y["SalePrice"].tolist()), test2, model=RandomForestRegr
             model_type="random-forest", param_grid=param_grid)
```

```
Train shapes:
X_train: (1022, 300)
y_train: (1022,)
```

```
Test shapes:
X_test: (438, 300)
y_test: (438,)

Tuned random-forest Regression Parameters: {'n_estimators': 95}
Best score is 0.8294253950501675
```






```
random-forest Mean Absolute Error: 0.10589763937294087
random-forest Mean Squared Error: 0.02386181908599778
random-forest Root Mean Squared Error: 0.1544727130790347
** random-forest Root Mean Squared Logarithmic     Error **: 0.012111766000076112

     Id      SalePrice
0  1461  121846.113439
1  1462  154977.000336
2  1463  182610.484772
3  1464  180887.139245
4  1465  196109.486597
```

Insights: The Root Mean Squared Logarithmic Error increased. The training data looks great in terms of prediction but the test is not doing so well.

What score does this give us on Kaggle? Remember, the smaller the better!

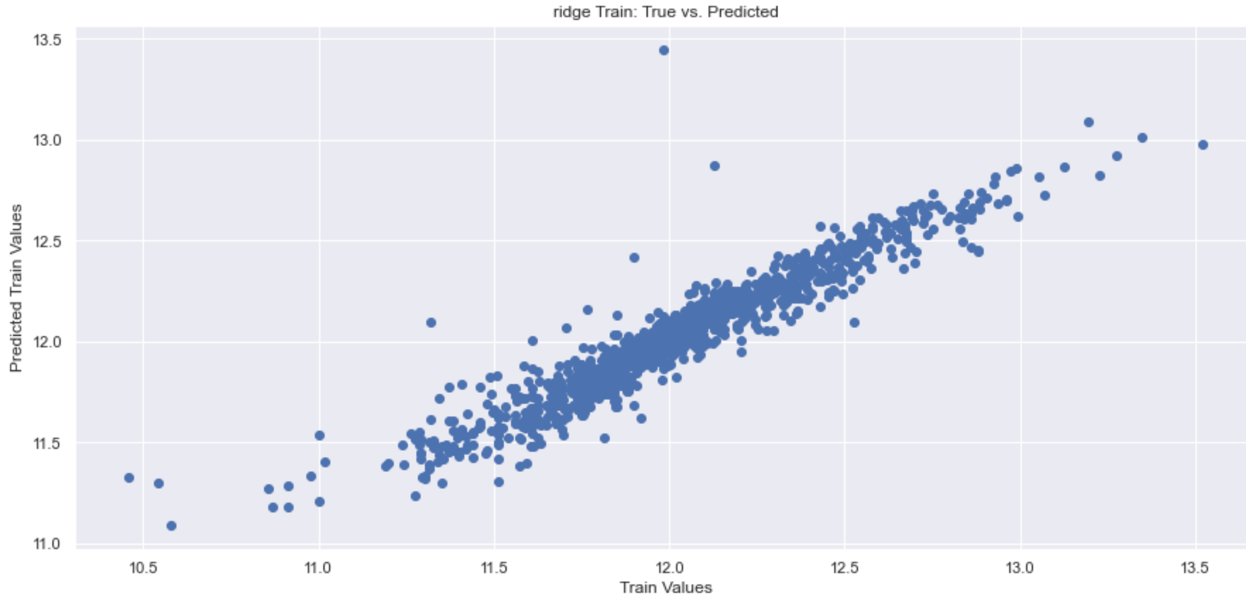
- **0.16515**
  This has definitely improved! However, it is still not as good as Ridge Regression. Let's now try Ridge Regression with the same feature matrix that we just created.

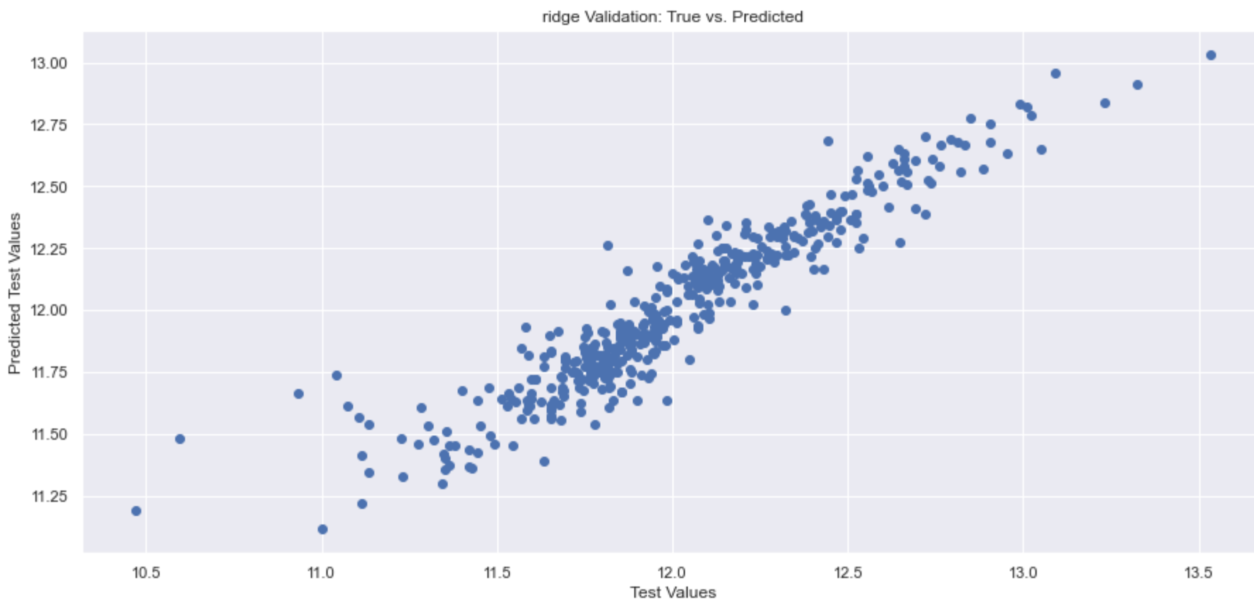
Last Model: Back to Ridge Regression

Random forests seem to not be doing too well, even as we added features. Our best model was Ridge Regression, so we should try to create a new model with the same 300 features we just created for our previous random forest model.

In [68]:
```
# Setup the hyperparameter grid
alph = np.arange(0.5, 21.5, 0.5)
fit_intercept = np.array([True, False])
normalize = np.array([True, False])
param_grid = {'alpha': alph, "fit_intercept": fit_intercept, "normalize": normal

create_model(X, y, test2, model=Ridge(random_state=42), model_type="ridge", para
```

```
Train shapes:
X_train: (1022, 300)
y_train: (1022, 1)

Test shapes:
X_test: (438, 300)
y_test: (438, 1)

Tuned ridge Regression Parameters: {'alpha': 1.0, 'fit_intercept': True, 'normal
ize': True}
Best score is 0.8127842513715638
```






```
ridge Mean Absolute Error: 0.0996154264373381
ridge Mean Squared Error: 0.021768765263439502
ridge Root Mean Squared Error: 0.14754241852240155
** ridge Root Mean Squared Logarithmic     Error **: 0.011515540095241068

     Id      SalePrice
0  1461  110967.736126
1  1462  146273.152224
2  1463  176917.475158
3  1464  193920.442633
4  1465  196654.746933
```

What score does this give us on Kaggle? Remember, the smaller the better!

- **0.14961**
  Wow! Ridge regression is definitely doing better than Random Forests, even though the scatterplot doesn't look very well fit. Maybe we are overfitting with random forests. Let's try Ridge again with 350 features now.

In [69]:
```
X, test2 = create_feature_matrix(350)
```

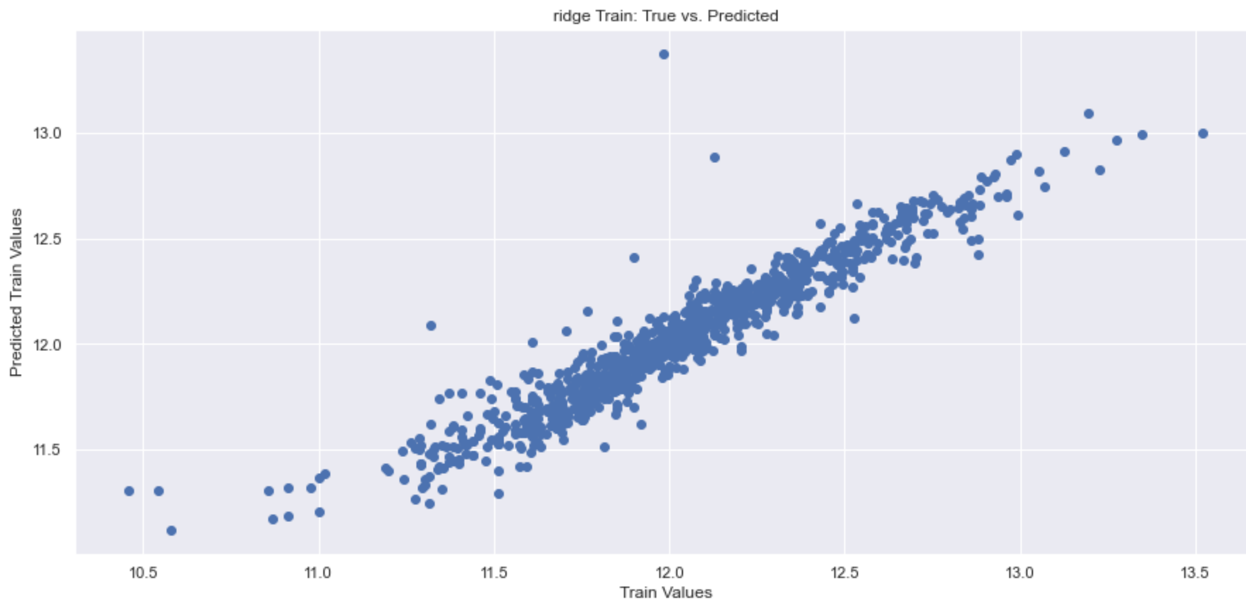
```
     Features  Variance Squared
```

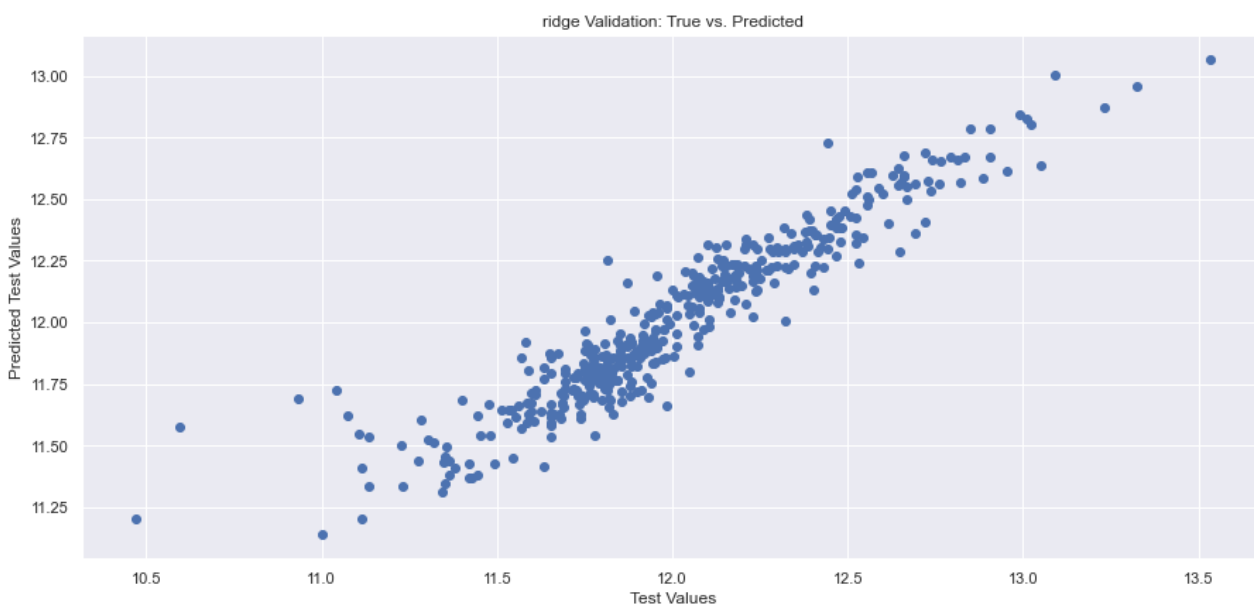
```
0  LotFrontage          0.001192
1      LotArea          0.000329
2   MasVnrArea          0.000245
3   BsmtFinSF1          0.000203
4   BsmtFinSF2          0.000165

Variance explained: 0.9603803169778391
```

In [70]:

```
# create the model with same param_grid from the last ridge regression
create_model(X, y, test2, model=Ridge(random_state=42), model_type="ridge", para
```

```
Train shapes:
X_train: (1022, 350)
y_train: (1022, 1)

Test shapes:
X_test: (438, 350)
y_test: (438, 1)

Tuned ridge Regression Parameters: {'alpha': 1.0, 'fit_intercept': True, 'normal
ize': True}
Best score is 0.8128536482529546
```






```
ridge Mean Absolute Error: 0.09803024819522446
```

```
ridge Mean Squared Error: 0.021548237583250616
ridge Root Mean Squared Error: 0.14679317962102537
** ridge Root Mean Squared Logarithmic     Error **: 0.011481309572392588

     Id      SalePrice
0  1461  111247.286426
1  1462  139403.626565
2  1463  176104.798453
3  1464  193945.897842
4  1465  190041.494195
```

Insights: The scores are slowly improving because I see them decreasing as we add more features.

What score does this give us on Kaggle? Remember, the smaller the better!

- **0.14962**
  This is surprisingly not an improvement from the last time.

Conclusion

Our best model so far is Ridge regression trained on the top 300 features that explain the most variance in the data. This resulted in our best score of 0.14961. In the future, I would like to improve these models with greater feature selection, or trying out completely new models that I haven't learned about yet or thought to use.

In [ ]: