

# Problem 3 Writeup

## Addressing Lab Questions

First we will address the lab questions, and then we will write up the work done in order to increase model performance.

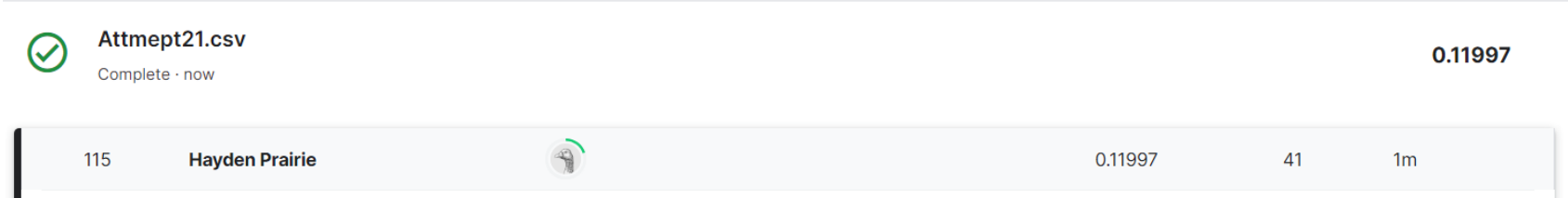
### 1. Part 1

By far one of the best discussions was this one:  
<https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques/discussion/23409>

The main idea was that after rigorous data preprocessing it was essential to remove outliers in order to prevent model over correction. It also went into detail about data leakage and what to avoid when training your model so that it is not over optimistic.

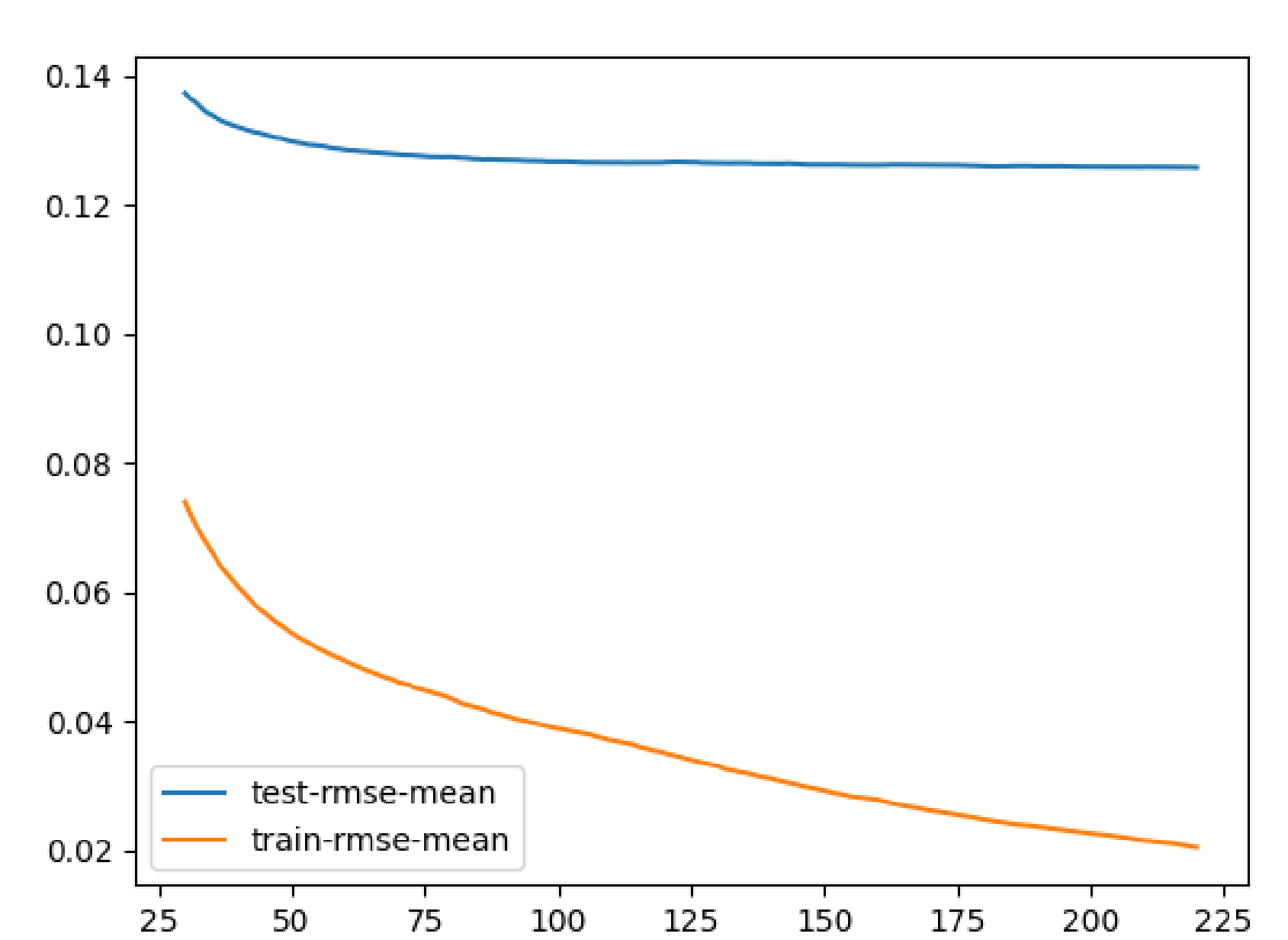
### 2. Part 2

Currently, the best score that we were able to achieve is displayed below.

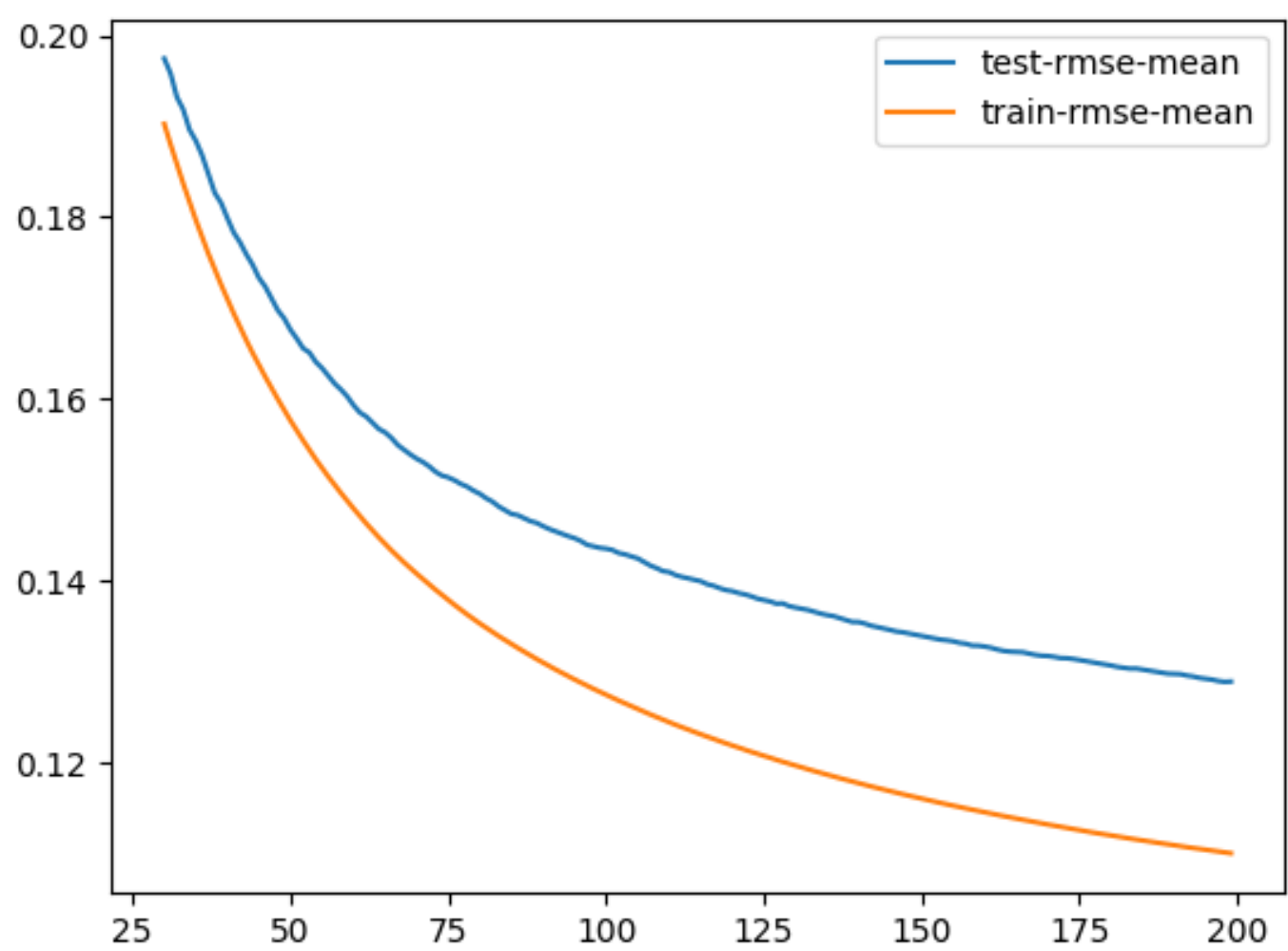


### 3. Part 3

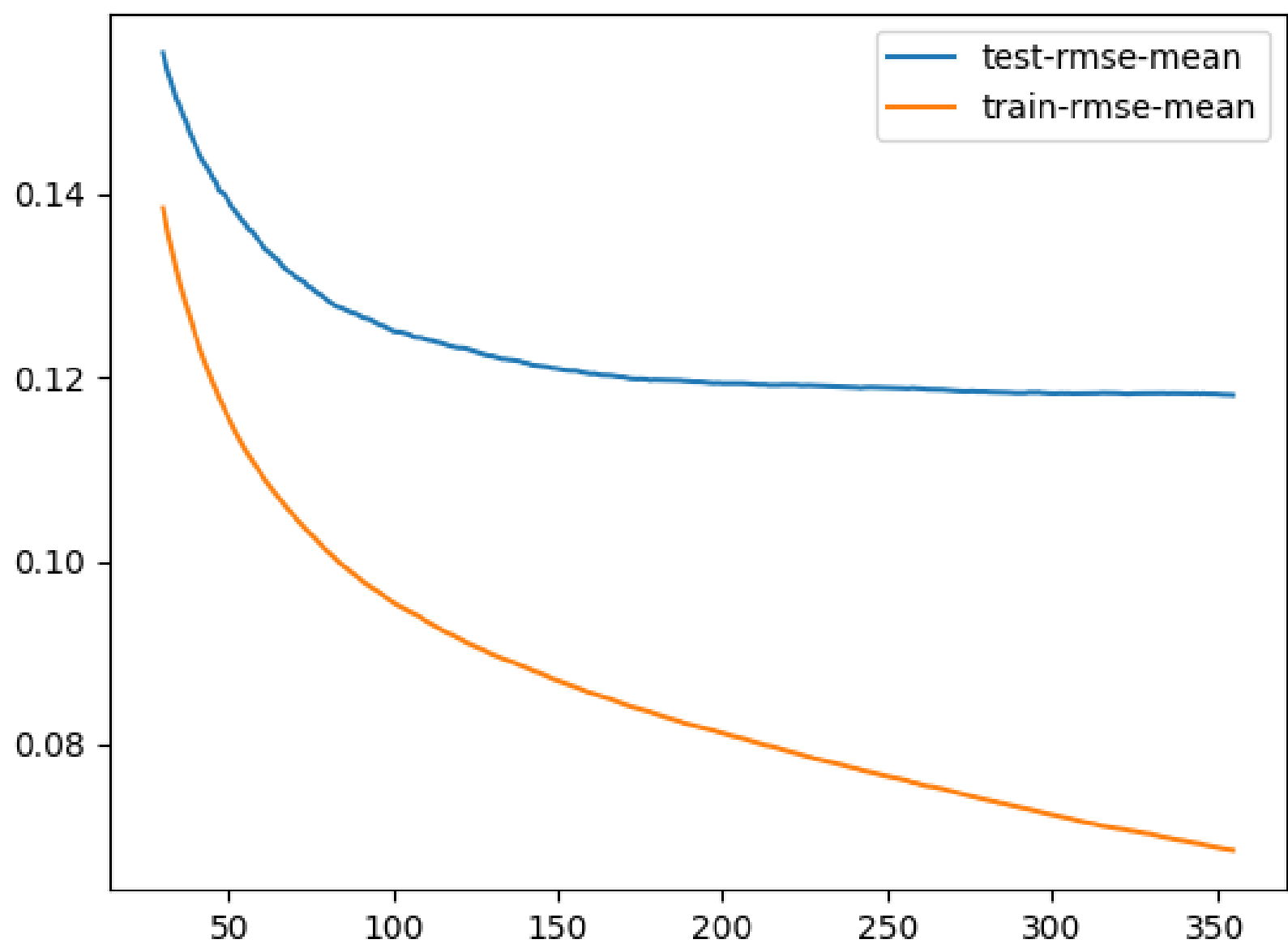
An example of a clearly over trained model is shown below.



An example of a clearly under trained model is shown below.



An example of a well trained model is shown below.



## Step 1: Initial Data Cleaning

The first thing that we can do in order to improve our MSE is to improve the preprocessing of our data. Several things were done wrong with the previous lab which would affect the MSE. There are several ordinal categories which had NaN values indicating that the house simply didn't have the feature and not that it wasn't recorded.

When working with ordinal values we can either one-hot encode its values or we can assign the values in order. Let's first start by assigning increasing values in order. After going through the dataset and reading about each of the features we can create the following code to assign values to assumed ordinal features.

```
bin_map = {np.nan:0,'Po':1,'Fa':2,'TA':3,'Gd':4,'Ex':5}
bin_map_2 = {np.nan: 0,"Unf" : 1, "LwQ": 2, "Rec": 3,"BLQ" : 4, "ALQ" : 5, "GLQ" : 6}
bin_map_3 = {np.nan:0,'No':1,'Mn':2,'Av':3,'Gd':4}
bin_map_4 = {np.nan:0,'MnWw':1,'MnPrv':2,'GdWo':3,'GdPrv':4}
bin_map_5 = {'Fin': 3, 'RFn': 2, 'Unf': 1, np.nan: 0}
bin_map_6 = {'Typ': 8, 'Min1': 7, 'Min2': 6, 'Mod': 5, 'Maj1': 4, 'Maj2': 3, 'Sev': 2, 'Sal': 1}
```

```
all_data['ExterQual'] = all_data['ExterQual'].map(bin_map)
all_data['ExterCond'] = all_data['ExterCond'].map(bin_map)
all_data['BsmtCond'] = all_data['BsmtCond'].map(bin_map)
all_data['BsmtQual'] = all_data['BsmtQual'].map(bin_map)
all_data['HeatingQC'] = all_data['HeatingQC'].map(bin_map)
all_data['KitchenQual'] = all_data['KitchenQual'].map(bin_map)
all_data['FireplaceQu'] = all_data['FireplaceQu'].map(bin_map)
all_data['GarageCond'] = all_data['GarageCond'].map(bin_map)
all_data['GarageQual'] = all_data['GarageQual'].map(bin_map)
all_data['PoolQC'] = all_data['PoolQC'].map(bin_map)
```

```
all_data['BsmtFinType1'] = all_data['BsmtFinType1'].map(bin_map_2)
all_data['BsmtFinType2'] = all_data['BsmtFinType2'].map(bin_map_2)
```

```
all_data['BsmtExposure'] = all_data['BsmtExposure'].map(bin_map_3)
```

```
all_data['Fence'] = all_data['Fence'].map(bin_map_4)
```

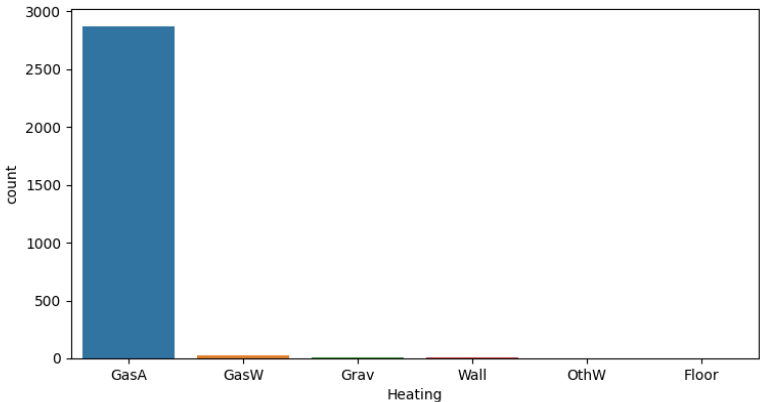
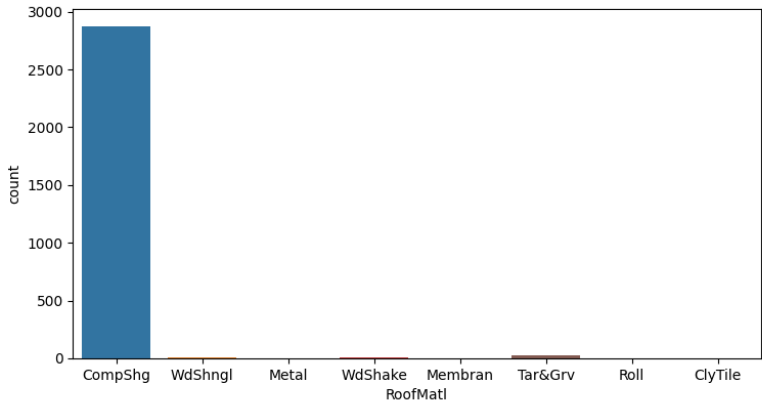
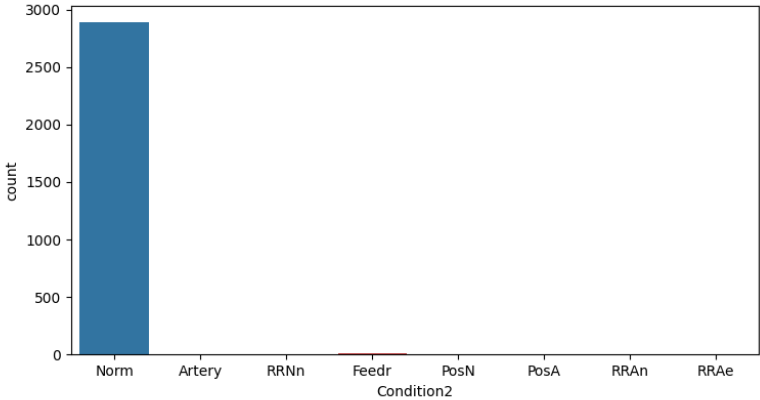
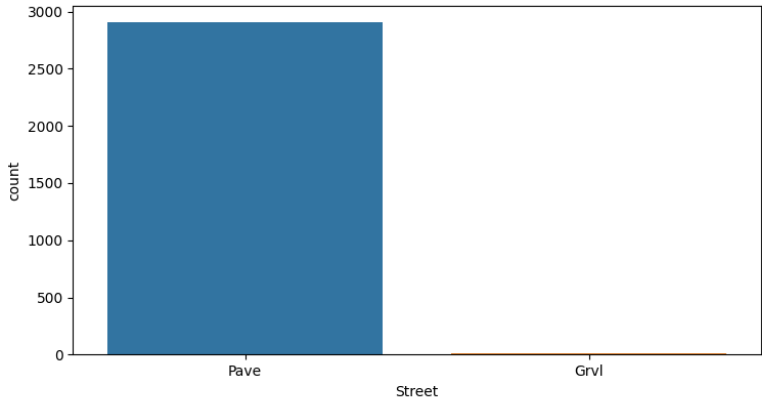
```
all_data['GarageFinish'] = all_data['GarageFinish'].map(bin_map_5)
```

```
all_data['Functional'] = all_data['Functional'].map(bin_map_6)
```

We can recognize that some columns with NaNs should not be filled with the average of those columns and instead they should be filled with 0 or the mode.

```
# Fill certain categorical columns NaN with None and numerical values with 0
col_cat_fillNA = ['Alley', 'GarageType', 'MiscFeature', 'MasVnrType']
for col in col_cat_fillNA:
    all_data[col].fillna('None', inplace=True)
col_num_fillNA = ['MiscFeature', 'MasVnrArea', 'BsmtFullBath', 'BsmtHalfBath',
                  'Functional', 'GarageArea', 'GarageCars', 'TotalBsmtSF',
                  'BsmtUnfSF', 'BsmtFinSF2', 'BsmtFinSF1']
for col in col_num_fillNA:
    all_data[col].fillna(0, inplace=True)
```

Next what we can do is determine some features which we think might not be useful. When plotting the distribution of each of the features we can see that a few of them have very little variance.



As the variance is so low, our model will not really be able to learn much from them, so we should go ahead and remove them from the dataset.

```
# Drop Columns with low variance
all_data = all_data.drop(['Street','Condition2','RoofMatl','Heating'],axis=1)
```

Another thing that we can notice about the dataset is that some features such as MSSubClass and MoSold have numerical values stored in them, however they are actually categorical should instead be assigned as such and then one-hot encoded.

```
# Convert some numerical value into catergorical type
all_data['MSSubClass'] = all_data['MSSubClass'].astype('object')
all_data['MoSold'] = all_data['MoSold'].astype('object')
```

Finally what we can do is fill the remaining values with the mean or mode, based on if they are numerical or categorical. When doing this we could take the average over the entire dataset, however, we can probably get a better estimate/value if we take the average of a house with respect to its neighborhood. Hopefully this improves our model as it will create more accurate estimates of average.

```
# Fill LotFrontage with median value of the neighborhood
all_data['LotFrontage'] = all_data.groupby('Neighborhood')['LotFrontage'].transform(lambda x: x.fillna(x.median()))

# Fill MSZoning with mode value of MSSubClass
all_data['MSZoning'] = all_data.groupby('MSSubClass')['MSZoning'].transform(lambda x: x.fillna(x.mode()[0]))

# Fill other categorical features with mode value of the neighborhood and OverallQual
col_cat_fillNA = ['Electrical', 'Utilities', 'Exterior1st', 'Exterior2nd', 'SaleType', 'KitchenQual']
for col in col_cat_fillNA:
    all_data[col] = all_data.groupby(['Neighborhood', 'OverallQual'])[col].transform(lambda x: x.fillna(x.mode()[0]))
```

We can then ensure that all of the data is numerical by one-hot encoding the remain object types. Note that all of these are nominal.

```
# Get dummies variables for remain data
all_data = pd.get_dummies(all_data)
```

# Step 2: Further Data Preprocessing

## 1. Adding and Adjusting Features

One thing that can be seen with some of the features is that they are often complex and don't have simpler features to express the same idea. For example the total square footage of the house is split into three categories or houses have labels with the sq footage of a deck, but do not have an indicator if they have a deck. We can create extra features to try to supply the model with more data.

```
# Add some features
all_data['TotalSF1'] = all_data['TotalBsmtSF'] + all_data['1stFlrSF'] + all_data['2ndFlrSF']
all_data['HasWoodDeck'] = (all_data['WoodDeckSF'] == 0)
all_data['HasOpenPorch'] = (all_data['OpenPorchSF'] == 0)
all_data['HasEnclosedPorch'] = (all_data['EnclosedPorch'] == 0)
all_data['Has3SsnPorch'] = (all_data['3SsnPorch'] == 0)
all_data['HasScreenPorch'] = (all_data['ScreenPorch'] == 0)
all_data['TotalBathrooms'] = all_data['FullBath'] + all_data['HalfBath'] * 0.5 + all_data['BsmtFullBath'] + all_data['BsmtHalfBath'] * 0.5
all_data['TotalPorchSF'] = all_data['OpenPorchSF'] + all_data['3SsnPorch'] + all_data['EnclosedPorch'] + all_data['ScreenPorch'] +
all_data['WoodDeckSF']
```

Another thing that can be change is the features which use a year. Instead of have just a plot around large numbers such as a range from 1950-2020, we can adjust this so that it is a plot around 0-150.

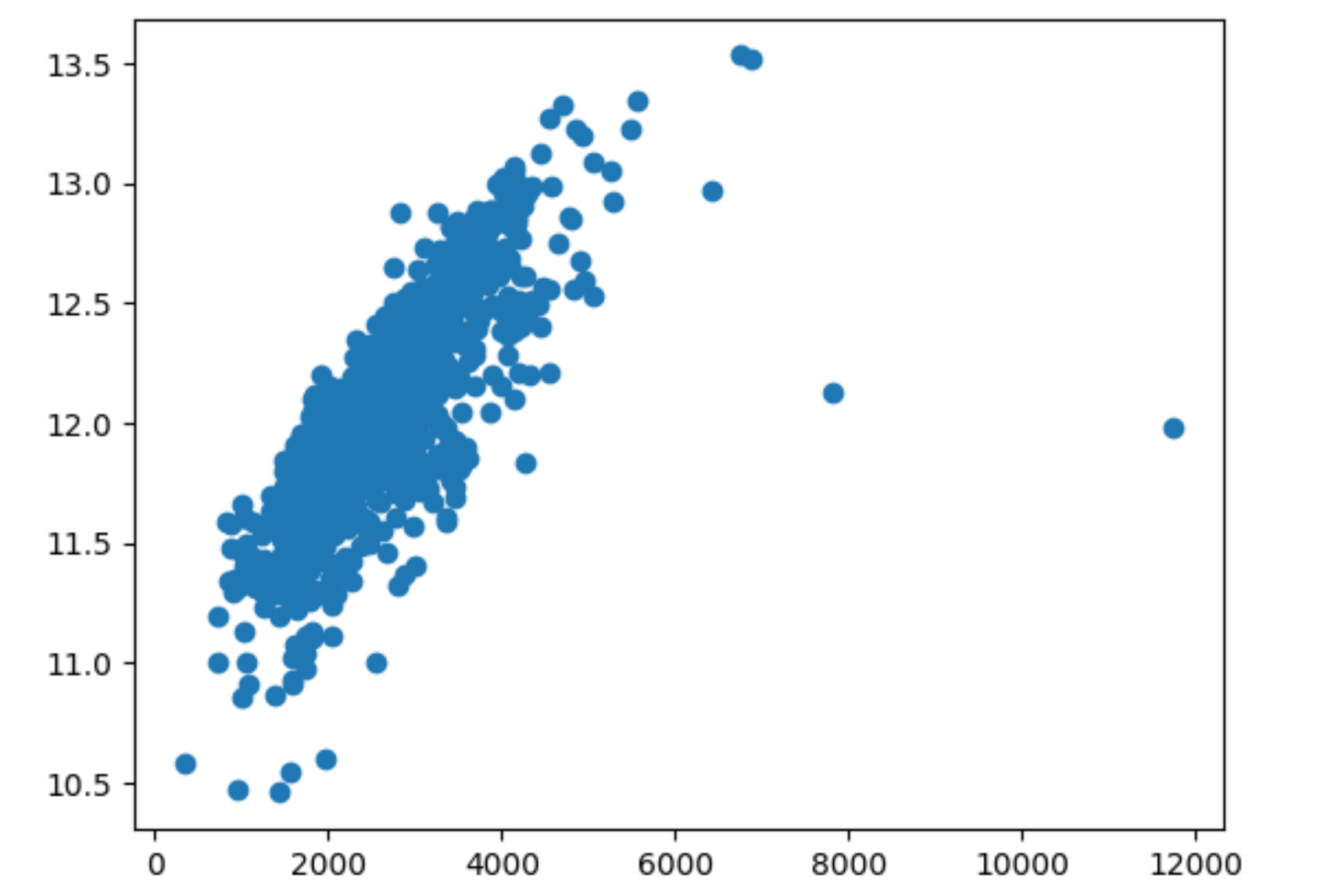
```
# Convert all features related to year to age
for i in ['YearBuilt', 'YearRemodAdd', 'GarageYrBlt', 'YrSold']:
    all_data[i] = all_data[i] - all_data[i].min()
```

Just as we log transformed the labels in the Lab 2, we can also log transform any numeric values which are heavily skewed.

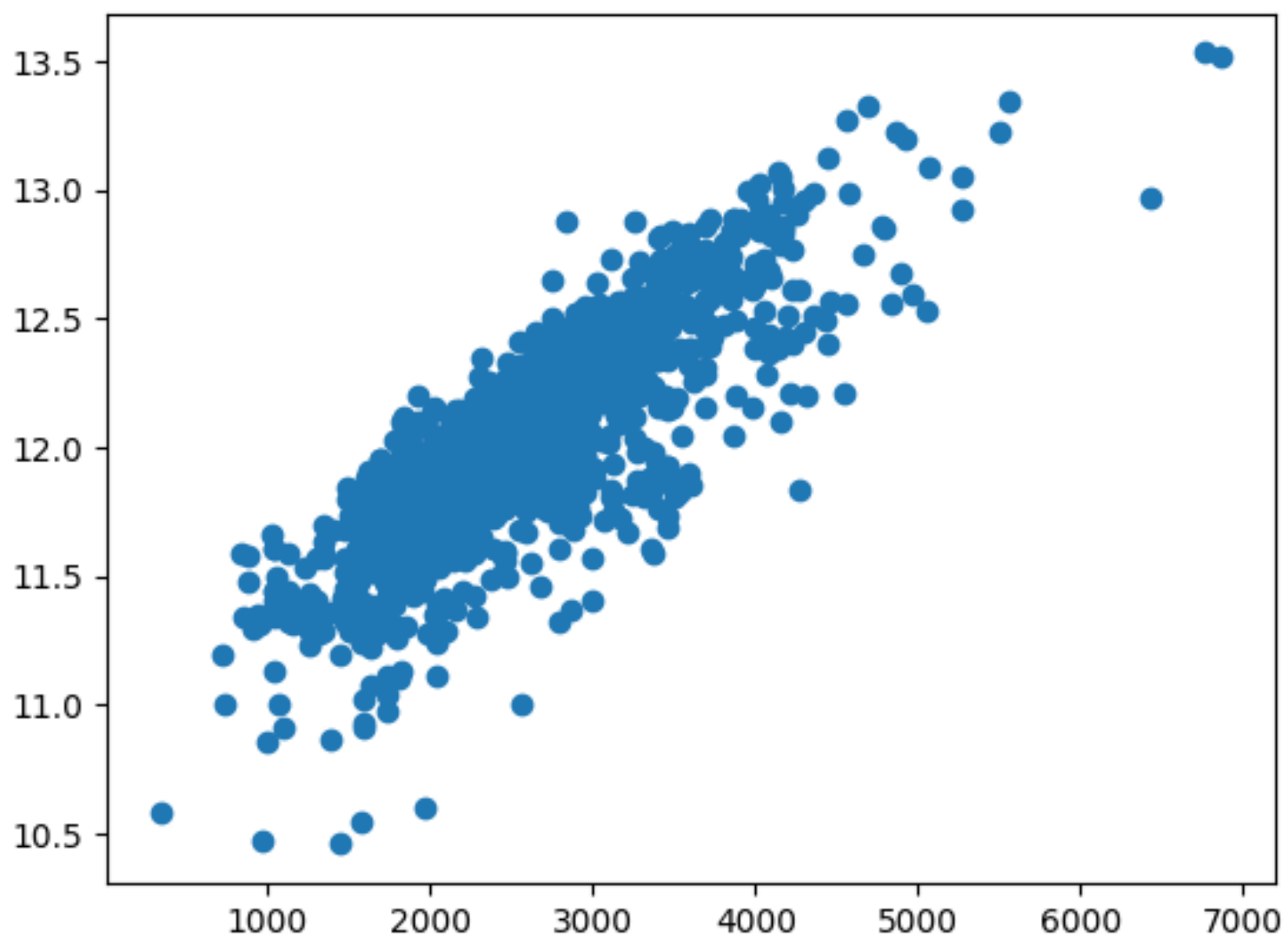
```
# Log transform skewed numeric values
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))
skewed_feats = skewed_feats[skewed_feats > 0.75]
skewed_feats = skewed_feats.index
all_data[skewed_feats] = np.log1p(all_data[skewed_feats])
```

## 2. Removing anomalies

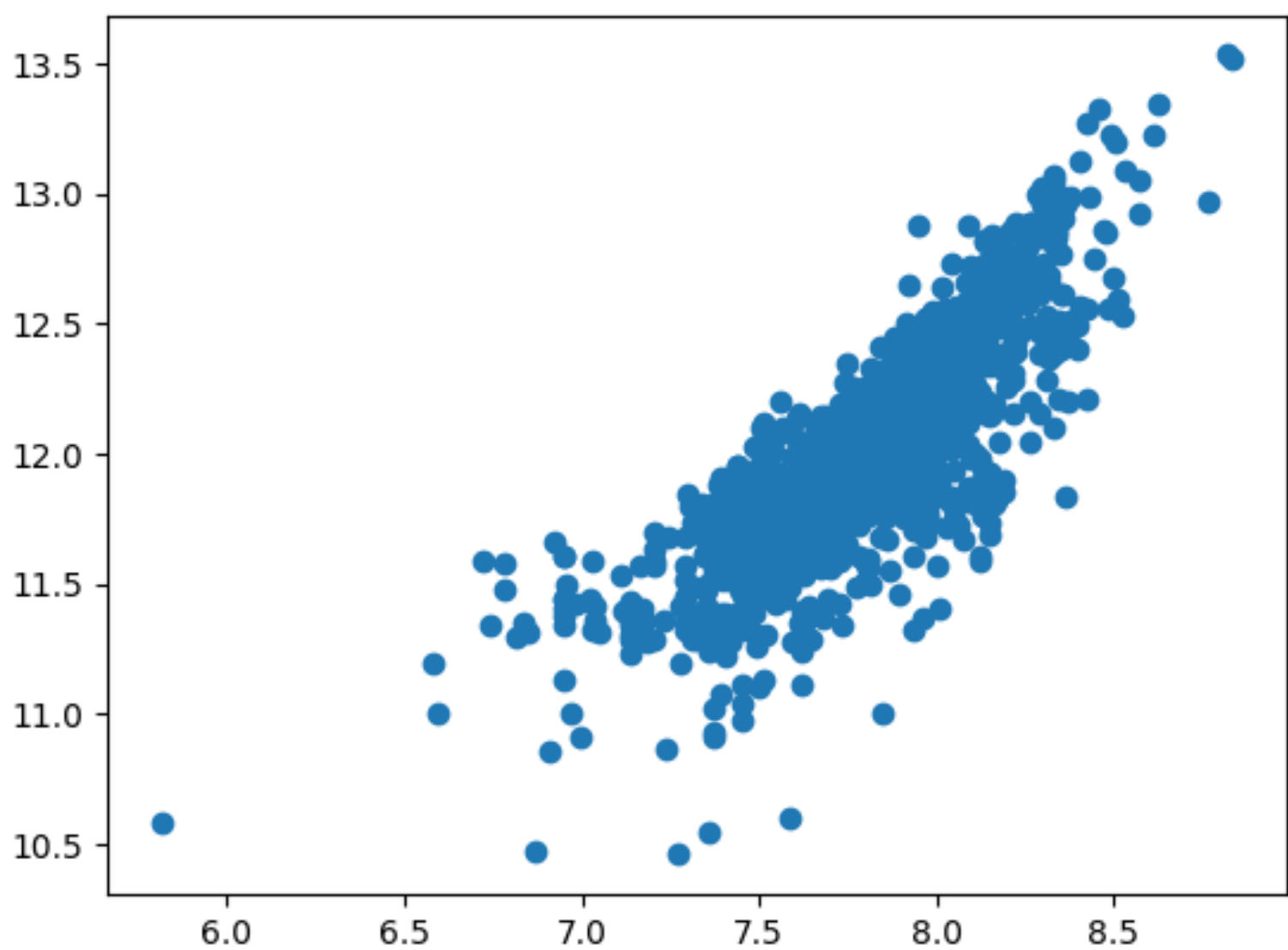
One thing to look out for in the dataset in anomalies in the data, the could heavily skew the data in ways that we don't want. When plotting the sell price (log transform) of a house to the square footage, we can see a fairly linear distribution, however there are some outliers. (Note these outliers are only really clear if the SquareFootage has not been log transformed)



So we can go ahead and remove those two apparent anomalies within the dataset, and we can see a more linear distribution.



Here is the distribution after reapplying the skew.

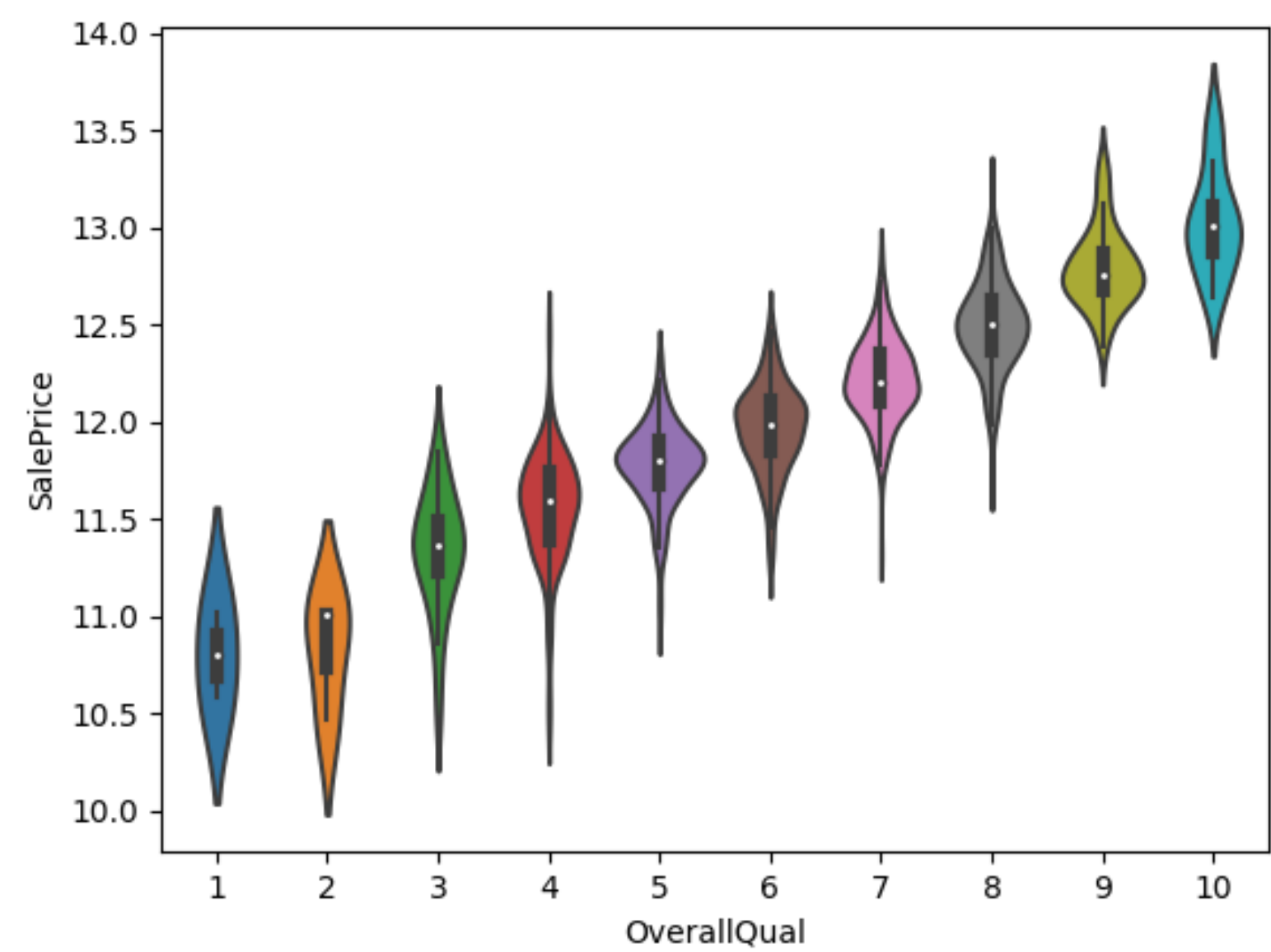



---

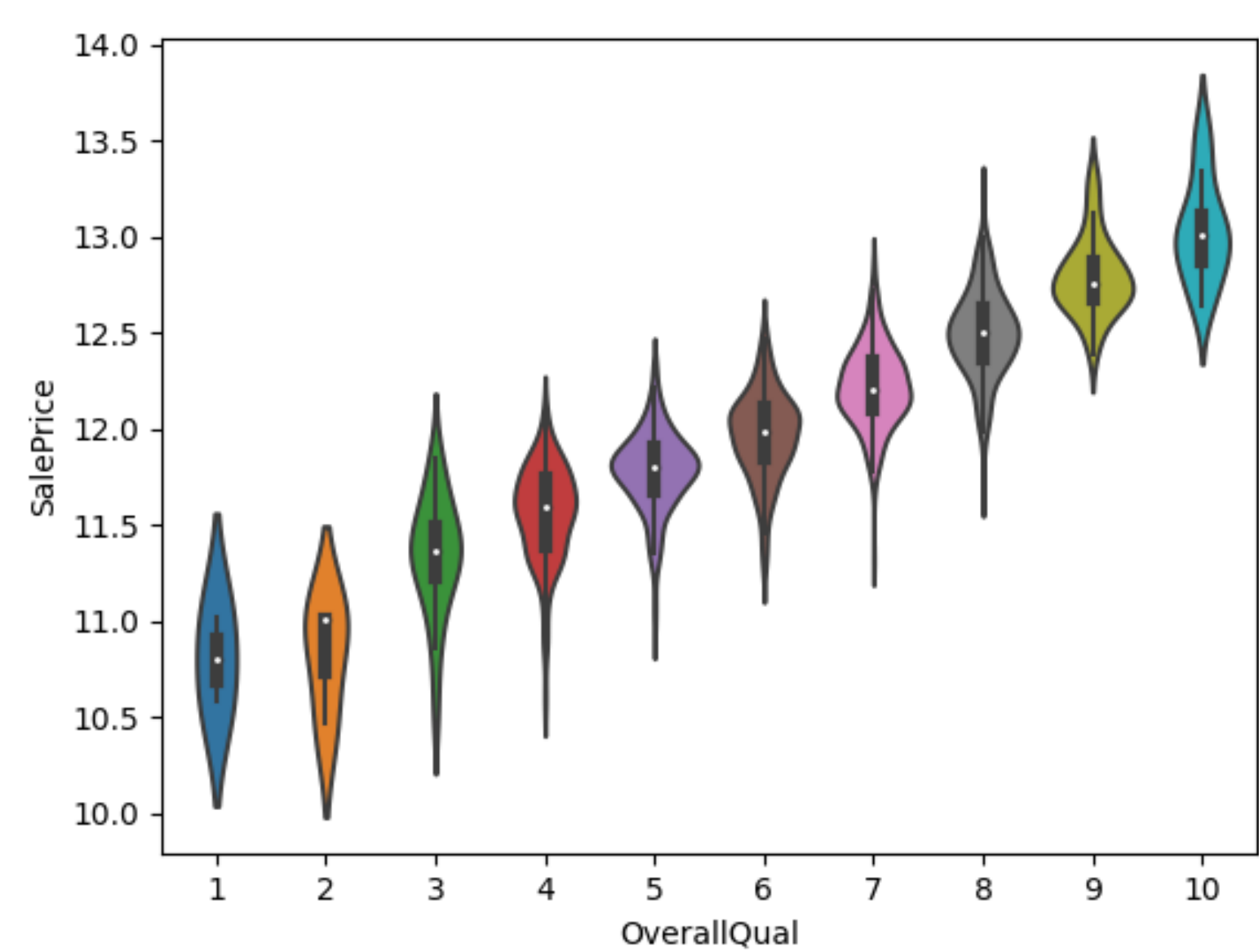
```
# Removing anomalies
loc = X_train.index[X_train['TotalSF1'] < np.log1p(7500)].tolist()
X_train = X_train.iloc[loc]
y = y.iloc[loc]
```

---

We can check for further anomalies in the dataset by looking at the violin plots of the sale price in comparison to the quality of the house, as again this scales fairly linearly. We can see with the violin graph below that OverallQual has fairly similar variance on each notch except when it is equal to 4, which indicates that there are some anomalies within the dataset.



We can go ahead and remove the anomalies from the dataset, and we see that we now get the following violin plot.



### 3. Rescaling Numerical Features

One final thing of data processing which we can preform is to rescale numerical values.

### 4. Feature Selection

Now we have a ton of features and we need to thin the herd. We can use sklearn `SelectFromModel` function in order to help select which features to remove from the dataset. After running the following code snippet, we can see that we are left with 26 features.

```
feature_sel_model = SelectFromModel(Lasso(alpha=0.005,random_state=0))

feature_sel_model.fit(X_train, y)
selected_feat = X_train.columns[(feature_sel_model.get_support())]

print('total features: {}'.format((X_train.shape[1])))
print('selected features: {}'.format(len(selected_feat)))
print('features with coefficients shrank to zero: {}'.format(np.sum(feature_sel_model.estimator_.coef_ == 0)))

print(selected_feat)
```

```
total features: 243
selected features: 26
features with coefficients shrank to zero: 217
Index(['LotArea', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
      'BsmtQual', 'BsmtFinSF1', 'HeatingQC', '1stFlrSF', '2ndFlrSF',
      'GrLivArea', 'KitchenQual', 'Functional', 'FireplaceQu', 'GarageYrBlt',
      'GarageFinish', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
      'EnclosedPorch', 'ScreenPorch', 'MiscVal', 'YrSold', 'Condition1_Norm',
      'TotalSF1', 'TotalBathrooms'],
      dtype='object')
```

(Note after going back and finding the optimal hyperparameters we were left with 107 features after Lasso selection)

We ended up never using feature selection as this often caused the models to underfit to the data.

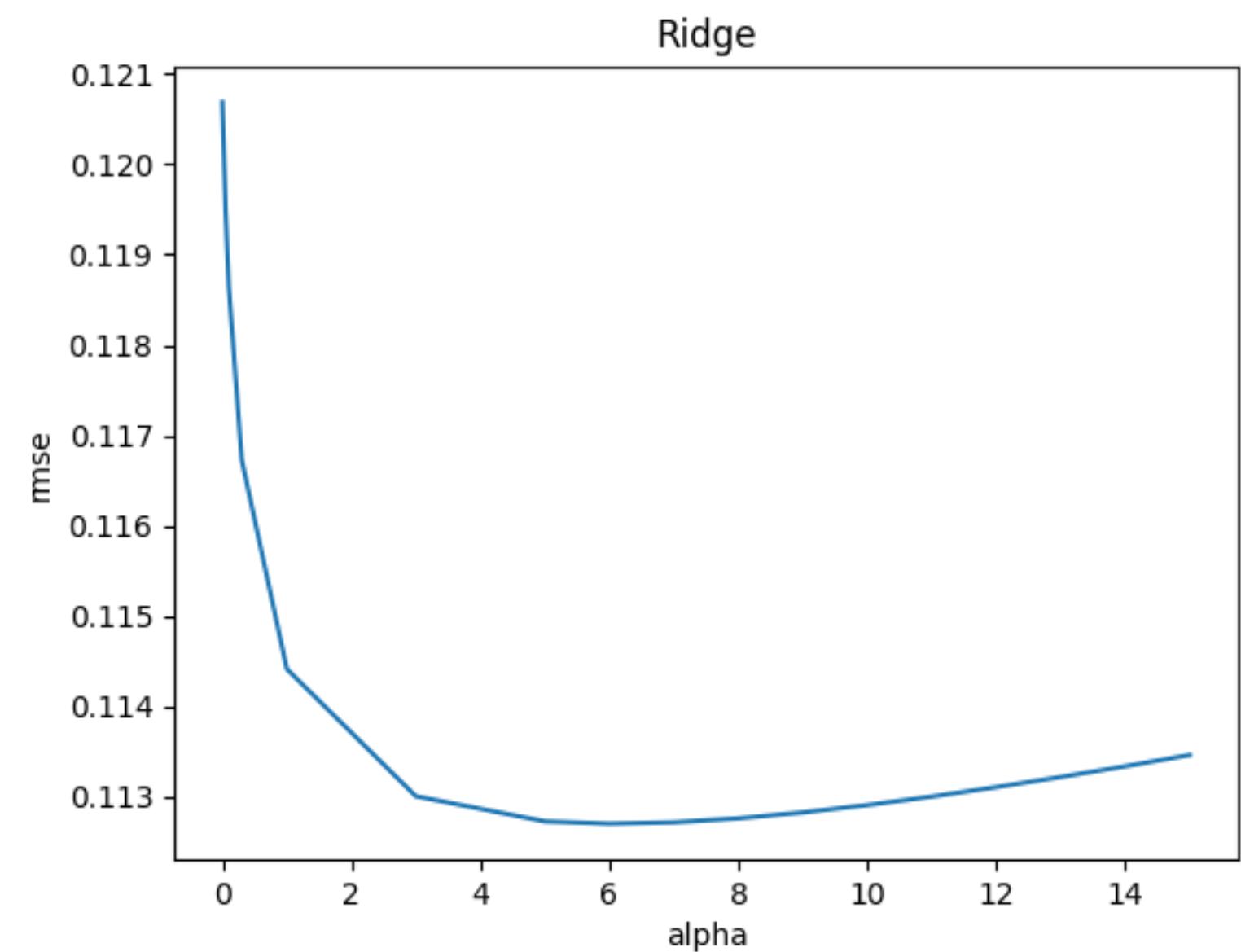
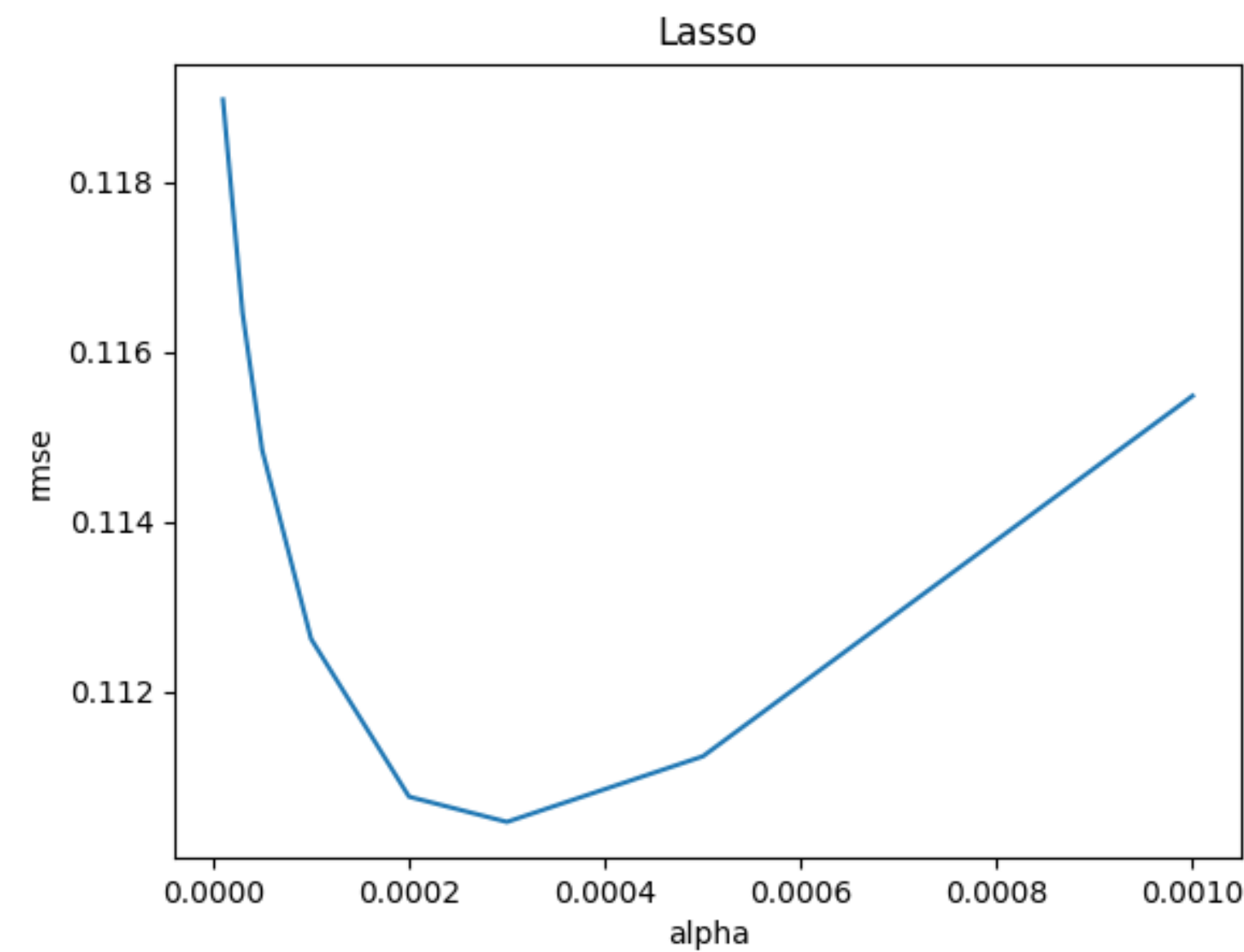


### Step 3: Building a Better Model

Using cross validation, the following function can be used to determine the MSE of a given model.

```
def rmse_cv(model):  
    rmse=np.sqrt(-cross_val_score(model, X_train, y, scoring="neg_mean_squared_error", cv = 5))  
    return(rmse)
```

From this we can look at both Ridge and Lasso Regression and determine the best alpha for both of these functions. We can see about the best alpha values for both Ridge and Lasso in the plots below.




We can see that with our processed dataset we get the best Ridge with alpha of 6 and the best Lasso with an alpha of 0.0003. Let go ahead and make a submission just on the Lasso prediction as its MSE is less then the ridge. We can use this prediction as a basis for MSE for the rest of models.

Without the new data preprocessing we got the following score using the best alpha for that data.


	<b>first_pred.csv</b> Complete · 4d ago · simple ridge pred	<b>0.13564</b>
---	--	----------------

After the new data preprocessing we can see a significant increase in our score, indicating the impact that the data preprocessing had on our MSE.

	<b>Attmept10.csv</b> Complete · now	<b>0.12245</b>
---	--	----------------

Before we go to stacking we should also run xgboost to see how it fairs. When submitting on the new preprocessing we see a better score.

	<b>xgb_sol.csv</b> Complete · 3d ago	<b>0.13286</b>
---	---	----------------

	<b>xgb_sol_new.csv</b> Complete · now	<b>0.12886</b>
---	--	----------------

I tried running some feature selection to improve the accuracy of the model and got no luck, resulting in the following score.

	<b>xgb_sol_feature_selection.csv</b> Complete · now	<b>0.12949</b>
--	--	----------------

Alright now we can run the same stacking model as before and see if we can get a better score, then we did. Here we will stack the Lasso and Ridge results and ran another Ridge, Lasso, and XGB boost. The model can be seen in the code snippet below and we can compare the results of further data preprocessing.

	<b>problem7_attempt2.6_sol.csv</b> Complete · 3d ago	<b>0.12367</b>
---	---	----------------

	<b>Attmept11.csv</b> Complete · now	<b>0.12081</b>
---	--	----------------

```
model_lasso = Lasso(alpha=0.0003).fit(X_train,y)
model_lasso_data = model_lasso.predict(X_train)
model_lasso_data_test = model_lasso.predict(X_test)
X_train_temp = np.column_stack((X_train,model_lasso_data))
X_test_temp = np.column_stack((X_test,model_lasso_data_test))

model_ridge = Ridge(alpha=6).fit(X_train,y)
model_ridge_data = model_ridge.predict(X_train)
model_ridge_data_test = model_ridge.predict(X_test)
X_train_temp = np.column_stack((X_train_temp,model_ridge_data))
X_test_temp = np.column_stack((X_test_temp,model_ridge_data_test))

def rmse_cv2(model):
    rmse= np.sqrt(-cross_val_score(model, X_train_temp, y, scoring="neg_mean_squared_error", cv = 5))
    return(rmse)

model_ridge = Ridge(alpha=6).fit(X_train_temp,y)
model_lasso = Lasso(alpha=0.0003).fit(X_train_temp,y)

lasso_preds = np.expm1(model_lasso.predict(X_test_temp))
ridge_preds = np.expm1(model_ridge.predict(X_test_temp))

model_xgb = xgb.XGBRegressor(n_estimators=360, max_depth=2, learning_rate=0.03) #the params were tuned using xgb.cv
model_xgb.fit(X_train_temp, y)
xgb_preds = np.expm1(model_xgb.predict(X_test_temp))

print(rmse_cv2(model_xgb).mean())

preds = (0.6)*(lasso_preds+ridge_preds)/2 + xgb_preds*(0.4)

solution = pd.DataFrame({"id":test.Id, "SalePrice":preds})
solution.to_csv("Attempts/Attmept11.csv", index = False)
```

# Step 4: Experimenting

Here we are just going to experiment with a bunch of different things and see what things work and what things don't. The list below will show what things are done along with the improvement that are seen with them or not.

- One-Hot-Encoding all Ordinal values and see if we get a performance gain

## 1. One - Hot Encoding Ordinal Values

When one hot encoding the ordinal values we can simple make the following change to the code in order to one-hot encode. Labelling the column type as object will cause the `get_dummies` function to one hot encode the columns that we want.

```
all_data['ExterQual'] = all_data['ExterQual'].map(bin_map).astype('object')
all_data['ExterCond'] = all_data['ExterCond'].map(bin_map).astype('object')
all_data['BsmtCond'] = all_data['BsmtCond'].map(bin_map).astype('object')
all_data['BsmtQual'] = all_data['BsmtQual'].map(bin_map).astype('object')
all_data['HeatingQC'] = all_data['HeatingQC'].map(bin_map).astype('object')
all_data['KitchenQual'] = all_data['KitchenQual'].map(bin_map).astype('object')
all_data['FireplaceQu'] = all_data['FireplaceQu'].map(bin_map).astype('object')
all_data['GarageCond'] = all_data['GarageCond'].map(bin_map).astype('object')
all_data['GarageQual'] = all_data['GarageQual'].map(bin_map).astype('object')
all_data['PoolQC'] = all_data['PoolQC'].map(bin_map).astype('object')

all_data['BsmtFinType1'] = all_data['BsmtFinType1'].map(bin_map_2).astype('object')
all_data['BsmtFinType2'] = all_data['BsmtFinType2'].map(bin_map_2).astype('object')

all_data['BsmtExposure'] = all_data['BsmtExposure'].map(bin_map_3).astype('object')

all_data['Fence'] = all_data['Fence'].map(bin_map_4).astype('object')

all_data['GarageFinish'] = all_data['GarageFinish'].map(bin_map_5).astype('object')

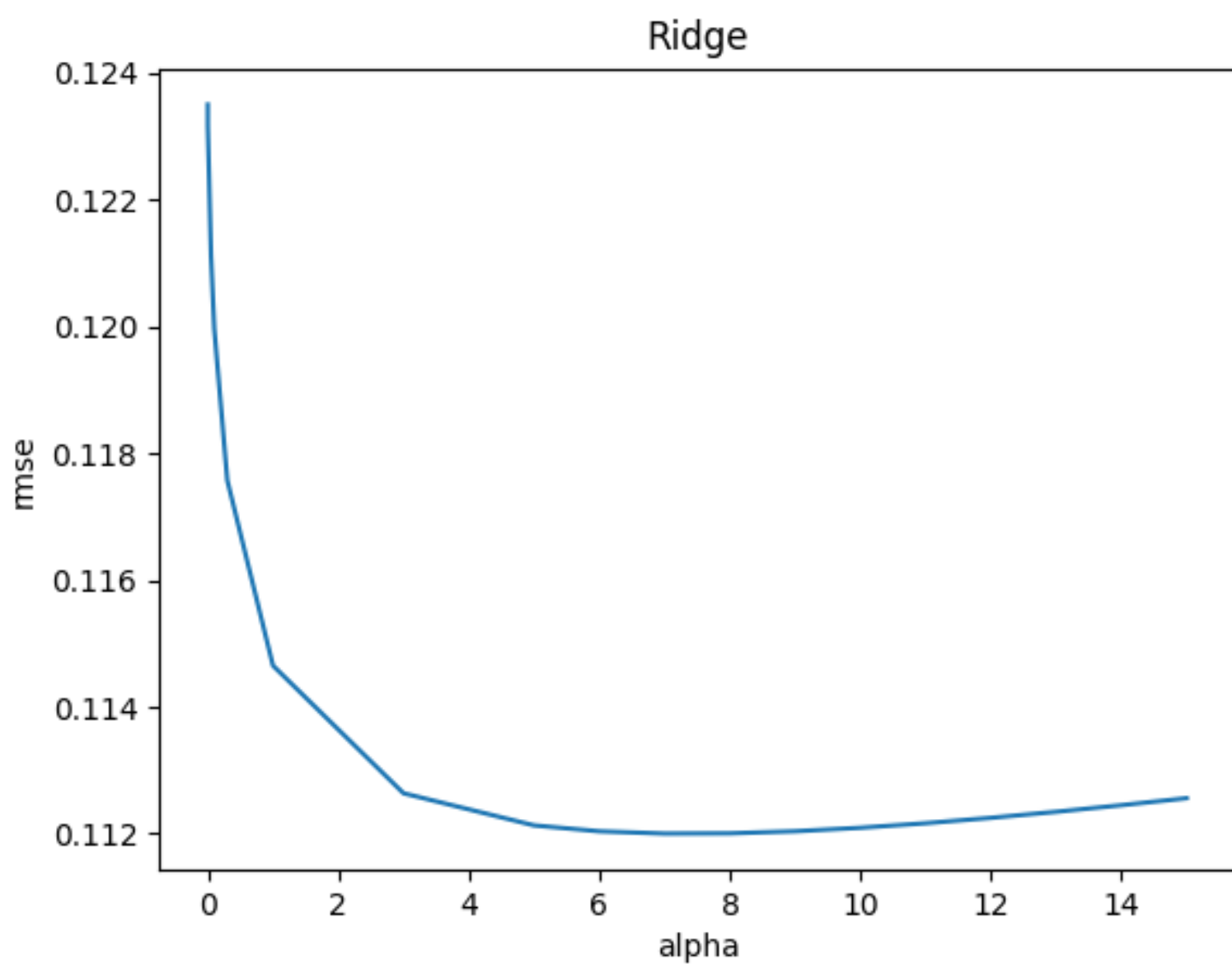
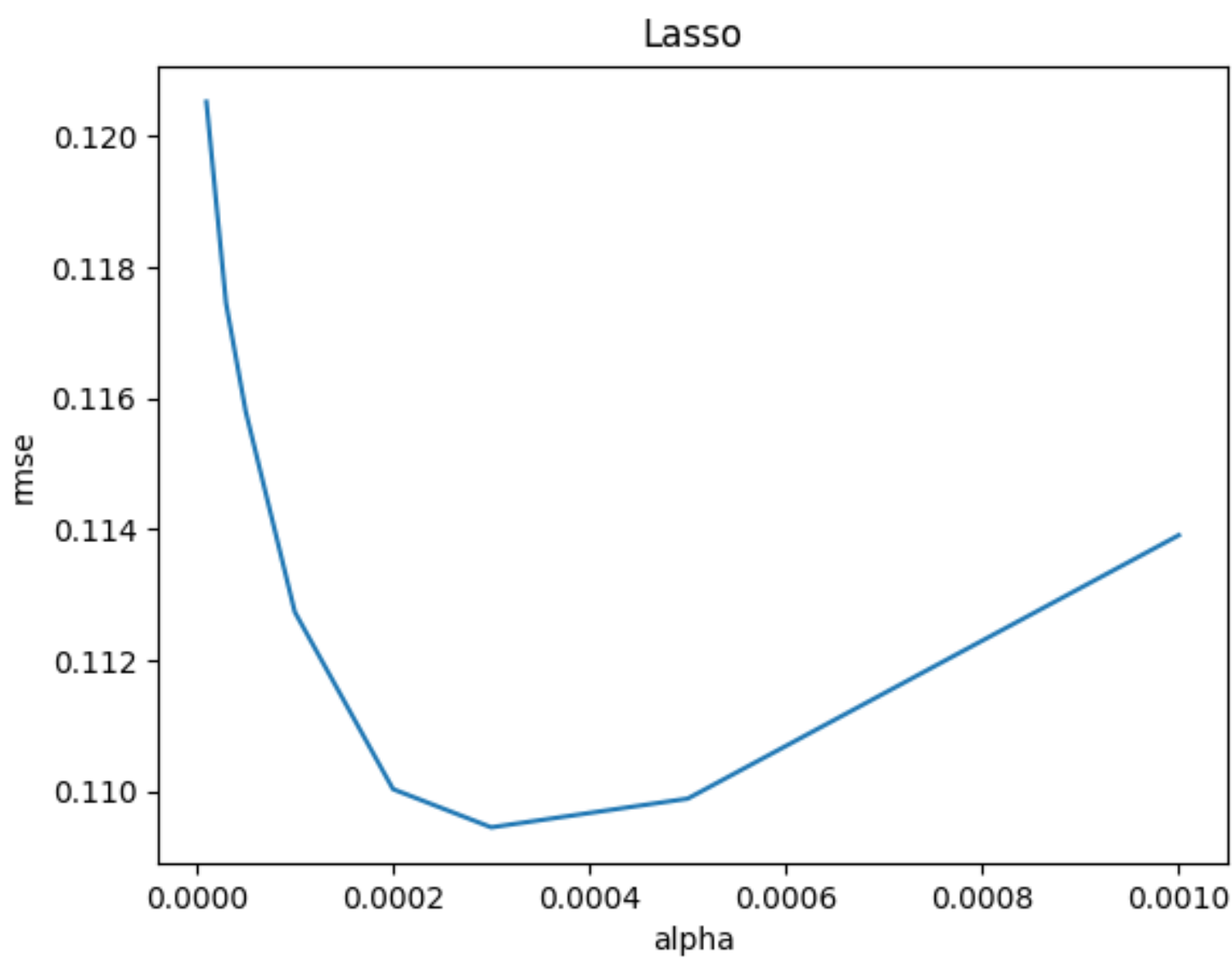
all_data['Functional'] = all_data['Functional'].map(bin_map_6).astype('object')
```

Doing this we do not actually see a performance gain, however we should also check to see if we need to adjust the alphas for Ridge and Lasso regression as this would also heavily affect the MSE of the model.

	<b>Attmept13.csv</b> Complete · now	<b>0.12125</b>
	<b>Attmept12.csv</b> Complete · 1h ago	<b>0.12076</b>

Note Attempt 12 is the best we could get before one-hot encoding the ordinal values. As you can see we don't really take that much of a performance hit if we do the following.

Lets plot the alpha for both Lasso and Ridge and see if it needs to be changed in order to get better performance.



While lasso did not change, ridge change from an alpha of 6 to an alpha of 7, so we will go ahead and update this and see if we can get a better score.

✔ Attmept14.csv  
Complete · now

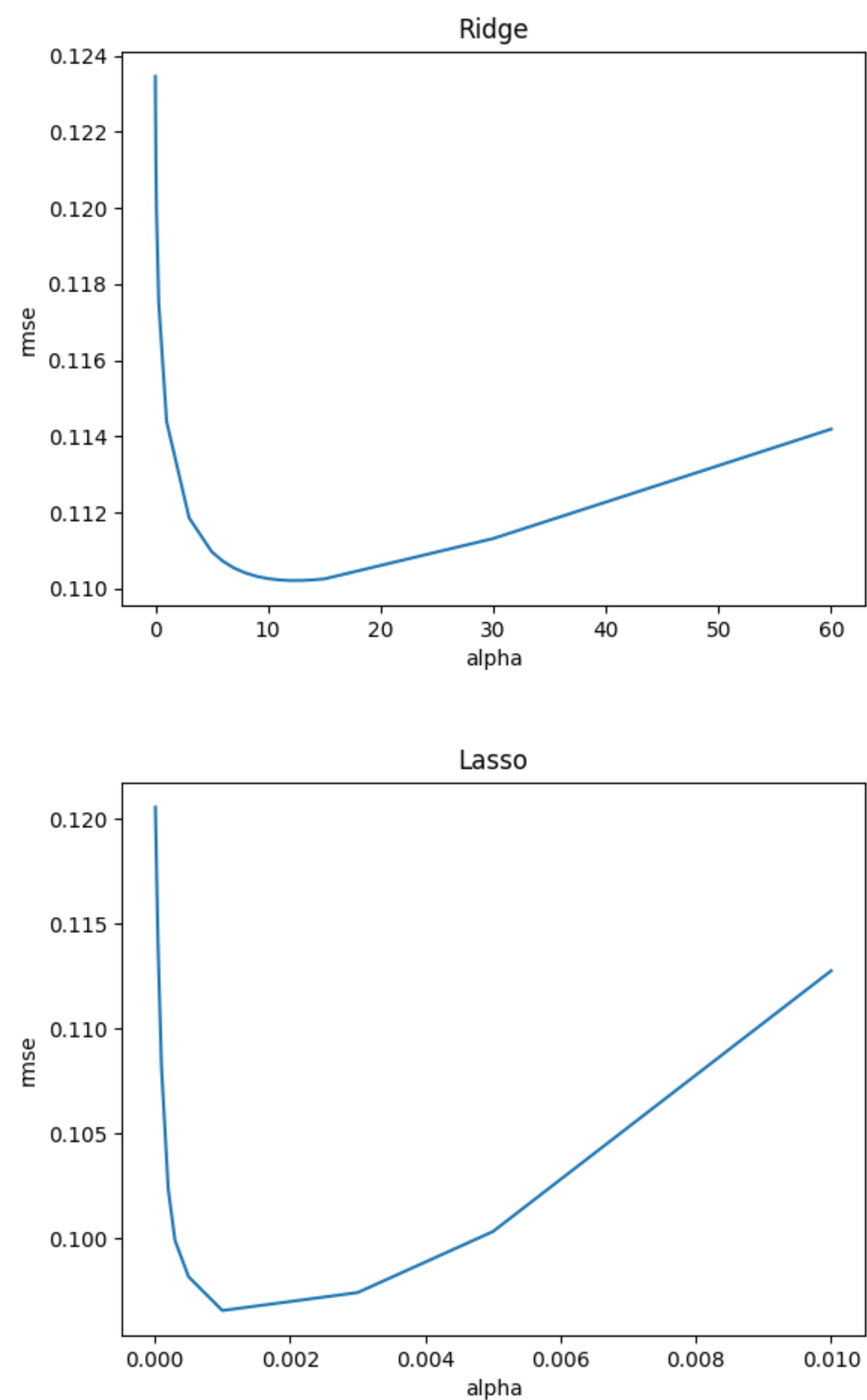
0.12109

While this does give us a slight increase in performance it doesn't beat our best model, so it seems like one-hot encoding all of the ordinal values don't help our model. However it could be that other parameters need to be better tuned, as when just running ridge and lasso we get a better 5-fold cross validation.


```
[0.12052714804553673, 0.11745119834714779, 0.11580274383052218, 0.11273974297074059, 0.11003519662287886, 0.10945230400034453, 0.10989018471229195, 0.11391070800442446]
Best Lasso: 0.109452
[0.12350406274908665, 0.1231636397907094, 0.12112674012793408, 0.11996977239102817, 0.11757461530934288, 0.11465152703167736, 0.1126375308979793, 0.11213090303852609, 0.11203785700686664, 0.11200181571726216, 0.11200560865478035, 0.11203849650882118, 0.1120933147641272, 0.11216504624156357, 0.1122500434375483, 0.11234557501647276, 0.11244954642606393, 0.11256031986661788]
Best Ridge: 0.112002
(460) hprairie@DESKTOP-CAT2S0B:~/4601/Labs/Lab3$ /home/hprairie/miniconda3/envs/4601/bin/python /home/hprairie/4601/Labs/Lab3/Problem3.py
(460) hprairie@DESKTOP-CAT2S0B:~/4601/Labs/Lab3$ /home/hprairie/miniconda3/envs/4601/bin/python /home/hprairie/4601/Labs/Lab3/Problem3.py
[0.1189710996803209, 0.11645787951386974, 0.11484290309836435, 0.11261518362911889, 0.11075553969375355, 0.11045816776955916, 0.11123137905361305, 0.1154793435881369]
Best Lasso: 0.110458
[0.1206914916077804, 0.1206797635321832, 0.11949344819184463, 0.11864569007958439, 0.11672988834358019, 0.11441113820217623, 0.11300142216599321, 0.11272335568491362, 0.11269768169215819, 0.11271264883842336, 0.11275643961545287, 0.11282162298845329, 0.11290317537078962, 0.11299751719832744, 0.11310199623418607, 0.11321458819779066, 0.11333371215499066, 0.11345811075929783]
Best Ridge: 0.112698
```

The first run is with one-hot encoding and the second is without it.

One thing to also look at is changing the second alpha in the stacking layer. When doing that we get the following results.



From this we can now see that the best alphas are 12 for ridge regression and 0.001 for Lasso regression. When after stacking Ridge and Lasso and then using a lasso regression we get the following results, which are the best ones yet.



Attmept15.csv

Complete · 1m ago

0.12026

## 2. Further Outlier Detection

It seems incredibly difficult to determine outliers in the dataset, especially when just looking 2D scatterplot data. Reading from discussion posts on this dataset, people have often gotten the best performance boost from removing outlier, so we are going to try a few things to find a few of these outliers. It is probably the easiest to find outliers while look at the numeric categories, so we will start with those.

We can take all categories which are numerical and use the statsmodel outlier test, which will calculate the Bonferroni p value of each data point with respect to the sale price. We can then say that outliers are any p value which are under 0.05. All of this can be summed up in the code snippet below.

```
bad_points = []
for col in numeric:
    regression = smapi.ols("data ~ x", data=dict(data=y, x=X_train[col])).fit()
    # print(regression.summary())
    testing = regression.outlier_test()
    # print('Bad data points (bonf(p) < 0.05):')
    bad_points.extend(testing[testing[testing.columns[2]] < 0.05].index.tolist())
    ## Figure #
    # figure = smgraph.regressionplots.plot_fit(regression, 1)
    ## Add line #
    # line = smgraph.regressionplots.abline_plot(model_results=regression, ax=figure.axes[0])

bad_points = list(set(bad_points))
X_train = X_train.drop(bad_points)
y = y.drop(bad_points)
```

After running the outlier detection, we found around 11 data points which heavily affecting the performance of the model. After removing those features we can look and see the effect that it has on the second layer Lasso and Ridge regression models.

Using 5 fold CV we can check to performance of before dropping the data points and after. The first test run through is with the data points dropped and the second is without.

```
[0.11282971734920544, 0.10925980441723412, 0.10629524822947896, 0.10148643229566476, 0.0964500176268607, 0.09425783915744132, 0.09238045061359268, 0.09103815733289015, 0.09225853026506294, 0.09553302996103988, 0.10929895806232866]
Best Lasso: 0.091038
[0.11653470796096921, 0.1159000989565957, 0.11361805370143854, 0.11252002107929149, 0.11036618915508267, 0.10774228201833554, 0.10550066261746835, 0.10466584925307947, 0.1044285680960827, 0.10426144187704955, 0.10414571104687431, 0.1040600914819058, 0.10402308349507697, 0.10400155573197807, 0.10399993869752426, 0.10401473648782578, 0.10404321639534875, 0.10408320357242658, 0.10534300630105013, 0.10694018628910476, 0.10850222314843863]
Best Ridge: 0.104000
(4601) hprairie@DESKTOP-CAT2S0B:~/4601/Labs/Lab3$ /home/hprairie/miniconda3/envs/4601/bin/python /home/hprairie/4601/Labs/Lab3/Problem3.py
[0.12054687021812323, 0.11711990580133362, 0.11389320363474167, 0.10820531310710635, 0.10235293095379938, 0.09991048946421915, 0.09818009144317796, 0.09657034954090417, 0.09742493278389638, 0.10033219541303691, 0.11275519818653619]
Best Lasso: 0.096570
[0.12345999929363827, 0.12311987634892277, 0.12108814878632465, 0.1199288468551728, 0.11750134148942593, 0.1143841666282481, 0.1118541761717411, 0.11096757071256798, 0.1107160525229665, 0.11053627258749084, 0.11040820483659657, 0.11031908632910829, 0.11026026605176882, 0.11022559909882305, 0.1102105579829277, 0.11021170779332926, 0.1102263795455856, 0.11025245792035465, 0.11131362992033127, 0.1127512924512875, 0.11418942259555513]
Best Ridge: 0.110211
```

We can see that there was a significant increase in the accuracy of the model after removing the data points. When submitting out new attempt to kaggle we get the following score.

One thing that we can do now is make sure that when we are calculating the mean and modes to input into NaN we don't include the outliers in these calculations, as they could skew the model and affect the performance. So with a little code tweaking we can replace the outlier detections before the filling of NaNs.

However when we go to submit our new trained model on these outliers we get a worse score. This tells us that we were being way to liberal with the number of data points that we were removing, and implying that we need to be more conservative. We can adjust the threshold which the Bonferroni p value needs to be below in order for the data point to be removed. Another thing that we can do is also just correct the data point the the average of the column neglecting it, allowing for us to retain most of the information in the row and help training.

## 3. Hyperparameter tuning for XGBoost

So the biggest problem which could be affecting the model is the XGBoost. We want to tune the XGBoost model so that it gets a good MSE as we could use it as a partial estimate with the Lasso regression. Luckily what we can do is run XGBoost's built in GridSearchCV to try to find the best hyperparameters for the model. We can see this implemented in the following code snippet.

```
xgb_params = {
    'n_estimators': [300,500,700,1000,1500,2000,3000],
    'learning_rate': [0.01, 0.1, 0.2,0.3,0.4,0.6,0.8],
    'max_depth': [2,3, 4, 5],
    'min_child_weight': [1, 2, 3],
}

# Xgboost grid search
xgb_grid = GridSearchCV(xgb.XGBRegressor(), xgb_params, cv=10,verbose=2)
xgb_grid.fit(X_train, y)
print("XGBoost Best Parameters:", xgb_grid.best_params_)
```

While a grid search this large is often not the best choice, I had time to kill and decided to run it. After running for around 4 hours on a AMD 3970X (all 32 cores maxing out the entire time) we got the following output. We found that the best n\_estimators was , the best learning\_rate was , the best max\_depth was, and the best min\_child\_weight was . We can then implement as a model and check in performance on kaggle.

```
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.4s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.5s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.1s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.1s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.1s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=2000; total time= 2.3s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.3s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.2s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.5s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.4s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.5s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 5.3s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.7s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.5s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 4.0s
[CV] END learning_rate=0.8, max_depth=5, min_child_weight=3, n_estimators=3000; total time= 3.8s
XGBoost Best Parameters: {'learning_rate': 0.1, 'max_depth': 2, 'min_child_weight': 3, 'n_estimators': 500}
```

<div><div>✓</div><div>Attmept16.csv</div><div>Complete · 43s ago</div></div>	0.12689
--	---------

We unfortunately did not get that great of a score form XGBoost. However we can take the average of the XGBoost prediction with other predictions and see if we can get a better score.

<div><div>✓</div><div>Attmept19.csv</div><div>Complete · now</div></div>	0.11998
--	---------

<div><div>✓</div><div>Attmept21.csv</div><div>Complete · now</div></div>	0.11997
--	---------

4. Ensure that there is no data leakage

While we never got to this, it would be important to go back in our work and ensure that there is no data leakage from stacking the train and test data in order to improve the convenience of data cleaning. In the future they can remain together, but need to be split when working with NaNs and filling with average.