main

December 7, 2025

[89]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import learning_curve
df_train = pd.read_csv('train.csv',index_col='id')
df_test = pd.read_csv('test.csv',index_col='id')
```

1 EXPLORING THE DATA

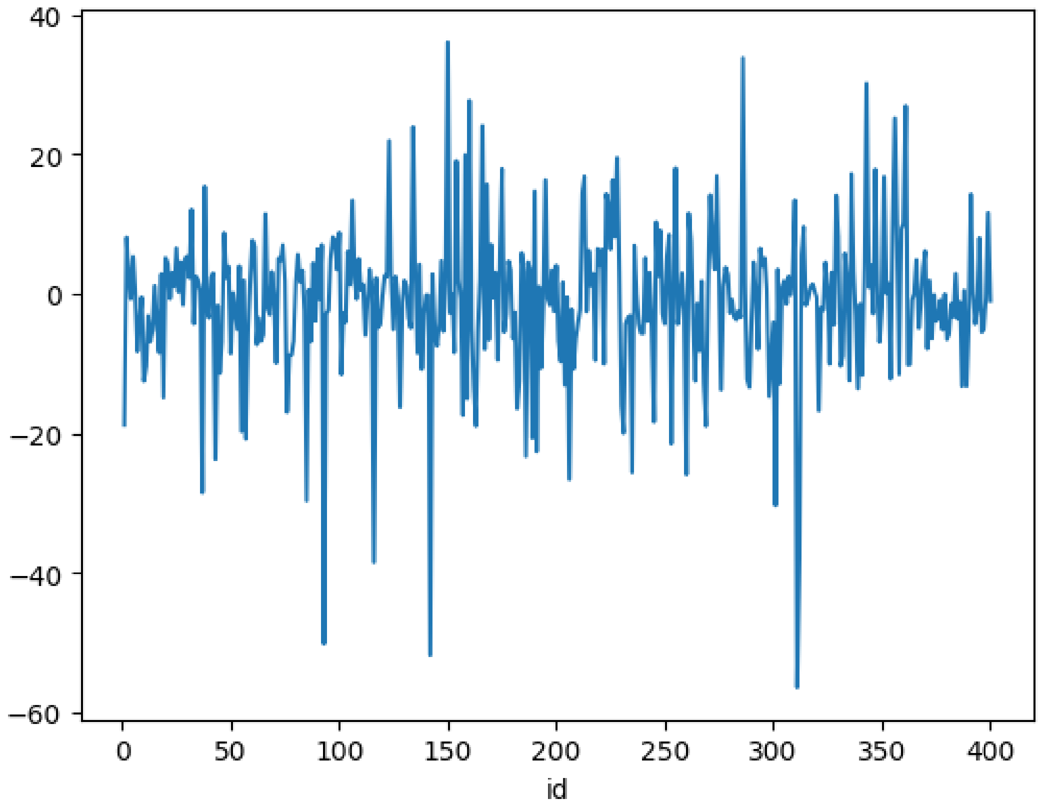
[90]:
```python
df_train.head()
```

[90]:
```
          f1        f2        f3        f4        f5        f6        f7  \
id
1   0.813120 -0.317922  0.027886  2.966993  1.792626  1.639461  1.038417
2   1.187370  0.574752  0.480094  1.003127  1.743455  1.653923  0.289389
3   0.713207 -0.061996  0.423746  0.111901  1.763365  1.651579  1.026891
4   0.779316  0.687488  0.388627  1.343889  1.743500  1.641686 -0.670589
5   0.674119 -0.286842  0.386524  2.416830  1.787492  1.636464  1.782221

          f8        f9       f10  …       f12       f13       f14       f15  \
id                                …
1  -1.265553  2.737895  3.723351  …  2.074784 -0.549459  0.588724  0.670610
2   1.758154  1.447072  0.014709  …  0.904899  0.936627  0.692230  0.631463
3   1.283781  1.116797  0.291573  …  1.581493  1.151487  0.213017  0.766878
4   0.135970  1.553194  2.046264  …  1.995739  0.650913  0.925546  0.657044
5   0.722959  2.030776  0.550102  …  0.485689 -1.000375  0.970572  0.900548

         f16       f17       f18       f19       f20     target
id
1   0.102525  1.963463 -0.349565 -0.294764 -0.206112 -18.759727
2   1.022266  2.211432 -0.318560  0.696366  0.693999   8.082223
3   1.003003  1.824628  1.248738       NaN  0.600792   2.961315
4   0.558577  1.915205  2.763816  1.665433  0.397841  -0.710178
5   0.638401  1.893995 -2.186096  1.838846  1.070571   5.328069
```

```
[5 rows x 21 columns]
```

[91]:
```python
df_train['target'].plot()
```

[91]:
```
<Axes: xlabel='id'>
```



Notes:

Need something that can deal with outliers as the above graph shows

Additionally we will need a imputer to deal with outliers

[92]:
```python
df_train.isna().sum() #!only minor  74 total but only 400 examples so cant just␣
↪get rid of them, imputer?
```

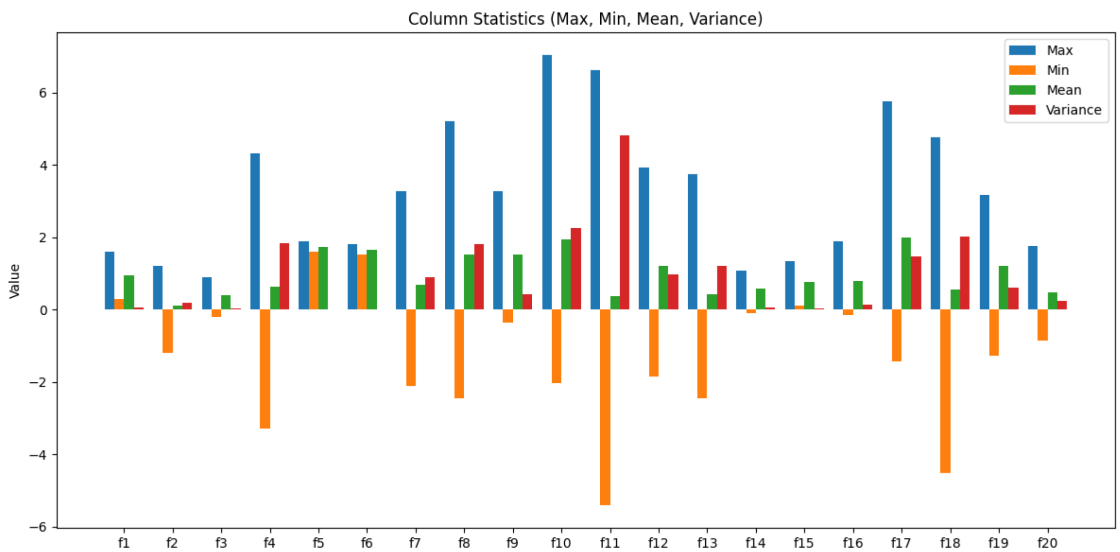
[92]:
```
f1         5
f2         2
f3         3
f4         3
f5         7
f6         2
f7         4
```

```
f8        3
f9        3
f10       5
f11       3
f12       3
f13       6
f14       5
f15       0
f16       5
f17       4
f18       2
f19       3
f20       6
target    0
dtype: int64
```

[93]:
```python
# THE FOLLOWING CODE WAS GENERATED WITH THE HELP OF CHATGPT, THE PROMPTS ARE␣
↪BELOW AND THIS WAS DONE SO THAT I CAN QUICKLY SEE THIS INFORMATION
# AS I WANTED TO SEE HOW THE DATA BEHAVED AND WHETHER I WOULD NEED TO SCALE THE␣
↪DATA

#prompt:
'''
write me some python code that takes in a pandas dataframe and for each column␣
↪in the dataframe,
makes a graph with the max, min, average, and variance for that column

can you make it so it displays all this info on one graph
'''
#Creates one grouped bar chart displaying max, min, mean, and variance for each␣
↪numeric column in the DataFrame.
def plot_all_column_statistics(df):
    numeric_cols = df.select_dtypes(include='number').columns
    stats = { # Collect statistics
        "Max": [],
        "Min": [],
        "Mean": [],
        "Variance": []}
    for col in numeric_cols:
        series = df[col].dropna()
        stats["Max"].append(series.max())
        stats["Min"].append(series.min())
        stats["Mean"].append(series.mean())
        stats["Variance"].append(series.var())
    # Plotting
    x = np.arange(len(numeric_cols))  # column positions
    width = 0.2  # bar width
```

```
    plt.figure(figsize=(12, 6))
    plt.bar(x - 1.5*width, stats["Max"], width, label="Max")
    plt.bar(x - 0.5*width, stats["Min"], width, label="Min")
    plt.bar(x + 0.5*width, stats["Mean"], width, label="Mean")
    plt.bar(x + 1.5*width, stats["Variance"], width, label="Variance")

    plt.xticks(x, numeric_cols)
    plt.ylabel("Value")
    plt.title("Column Statistics (Max, Min, Mean, Variance)")
    plt.legend()
    plt.tight_layout()
    plt.show()
```

[94]:
```
df_testing = df_train.drop('target',axis=1)
plot_all_column_statistics(df_testing)
```



Notes:

The above graph again shows that the data seems to be mostly on the same scale but with some of these extreme max and mins, scaling the data would be helpful

[95]:
```
X_train = df_train.drop('target',axis=1) #spliting the data
y_train = df_train['target']
```

2 DATA PREPROCESSOR

This will be a preprocessor used through all of the models.

As such its not going to do anything fancy, just impute and scale the data, for more fancy stuff like using PCA, thatll be defined at that specifc model.

Both of these preprocessors are needed for reasons that are shown above

[96]:
```
cols =␣
↪['f1','f2','f3','f4','f5','f6','f7','f8','f9','f10','f11','f12','f13','f14','f15','f16','f1

from sklearn.preprocessing import StandardScaler,PolynomialFeatures
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
from sklearn.compose import make_column_selector as selector

feature_processor=Pipeline(steps = [
    ('imputer',SimpleImputer(strategy='median')),
    ('scaler',StandardScaler())
    ])

processor = ColumnTransformer(transformers =␣
↪[('processor',feature_processor,cols)],remainder='passthrough' )
processor
```

[96]:
```
ColumnTransformer(remainder='passthrough',
                  transformers=[('processor',
                                 Pipeline(steps=[('imputer',
SimpleImputer(strategy='median')),
                                                 ('scaler', StandardScaler())]),
                                 ['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7',
                                  'f8', 'f9', 'f10', 'f11', 'f12', 'f13', 'f14',
                                  'f15', 'f16', 'f17', 'f18', 'f19', 'f20'])])
```

3 SOME FUNCTION DEFS FOR CONVIENCE

[97]:
```
from sklearn.metrics import mean_squared_error
#======
#This funciton splits the data into training and validation sets and plots the␣
↪performance of the mode
#======
def make_train_vs_test_curves(model,X_train,y_train):
    train_size_abs, train_scores, test_scores = learning_curve(
        model, X_train, y_train, train_sizes=[0.3, 0.6, 0.9]
    )
    plt.plot(train_scores.flatten(),color='red')
    plt.plot(test_scores.flatten(),color='blue')
#=====
```

```
#This function shows the orginal target data on top of the predicted data based␣
↪on the trained model, while also outputing the square root of the MSE score.
#=====
def training_data_fit_display(model,X_train,y_train):
    x_train_pred = model.predict(X_train)
    print("Here is the MSE score:")
    print(np.sqrt(mean_squared_error(y_train,x_train_pred)))
    plt.plot(x_train_pred)
    plt.plot(y_train,color='red')

time_taken_to_train = []
time_taken_to_predict = []
```

4 General method of training these models:

Because the training set is, relativly small, I decide to not explictly seperate out a testing set from the avaible training data, instead, my general method was to us GridSearchCV to search for hyperparamters and only after finding these params, performing train test spliting to train and then validate the model on the testing set, doing this several times. (this was achieved with the make_train_vs_test_curves function defined above).

Additionally beacuse I wanted to avoid overfitting, I began without using polynomial features for each model, only adding them after if I decided they had performed well enough to continue onwards.

My process for choosing the best models was based on the Kaggle submission score. I knew that overfitting could be a problem so all models i submited to kaggle at least once (even if they didnt have the best scores from my metrics, this was because something could have had a good train MSE score, but being overfit it does bad on the testing data, were as something without the best scores could still have done well in the testing data). Then based on those scores i either dropped the model, or tried to continue and refine it. (This can be seen with the ridge and lasso regression models, which performed ok with my metrics, but did the best in kaggle, so i made more versions/iterations of them till they got better)

Finally to avoid confusion I trained models 1-8 in that order, after seeing there performance on kaggle and on my metrics, i then choose Rideg and Lasso to do further iteraions on with models the the .2 versions and etc

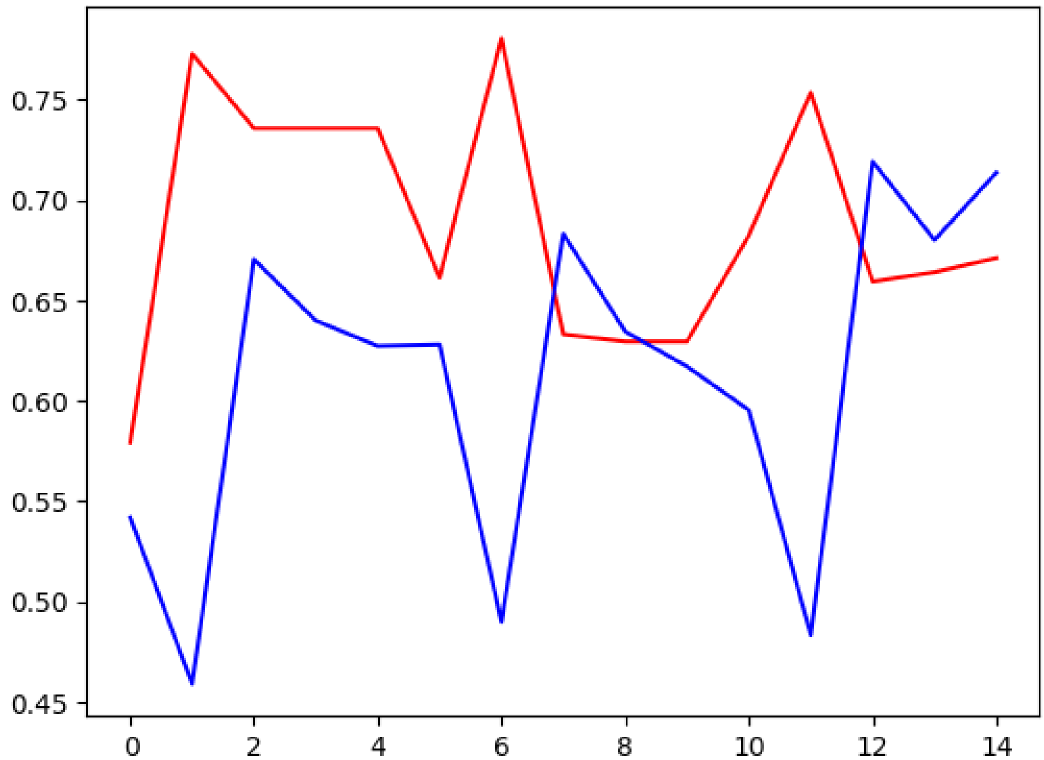
5 MODEL 1: REGRESSION

No regularization and no polynomial features, so nothing to grid search

[98]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error



test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
```

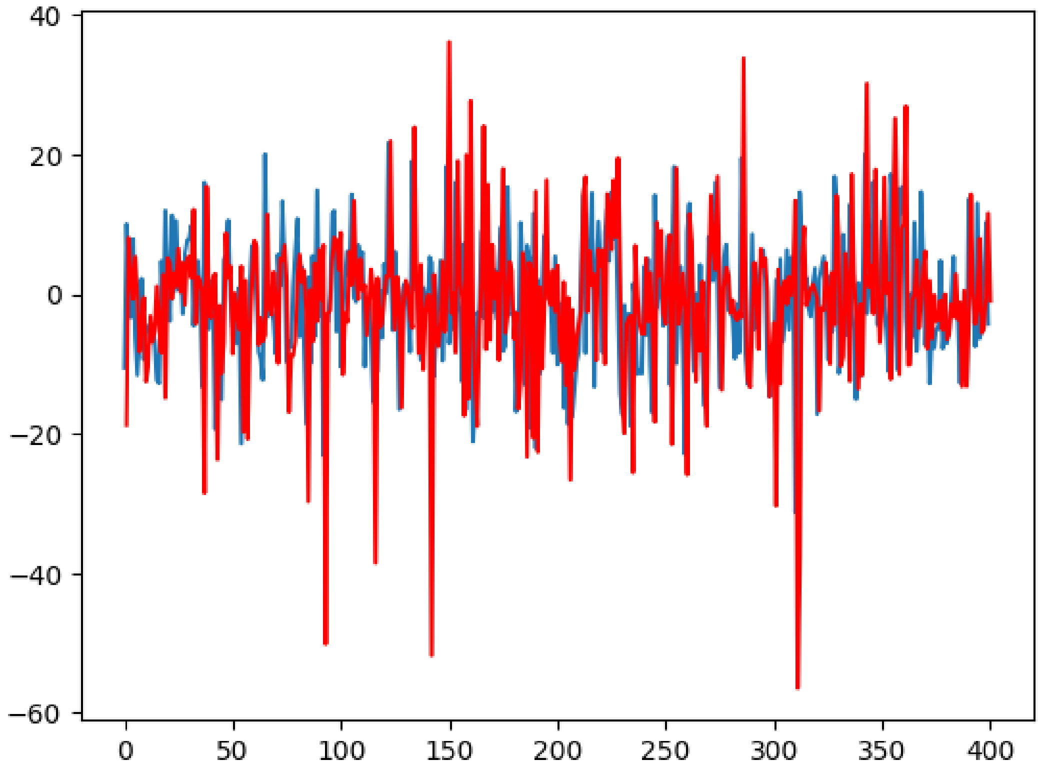
```
    ('model',LinearRegression())
])

import time

start = time.perf_counter()
test_pipeline.fit(X_train,y_train.to_numpy())
end = time.perf_counter()
time_taken_to_train.append(end-start)

start = time.perf_counter()
test_pipeline.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)
```

[99]:
```
make_train_vs_test_curves(test_pipeline,X_train,y_train)
```



[100]:
```
training_data_fit_display(test_pipeline,X_train,y_train)
```

```
Here is the MSE score:
6.213200464581835
```



Summary:

The model performs ok, the MSE score isnt the best, and more importantly the testing score in the validation doesnt perform the best

6 MODEL 2: SVM

Just SVM, nothing on top of it

[101]:
```
from sklearn.svm import SVR

test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('model',SVR())
])

params_dict = {
    "model__C":np.linspace(1,10,30),
    "model__epsilon":np.linspace(.01,1,20) #the two SVM metrics
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
```

```
                    cv = 10,
                    scoring = 'neg_mean_squared_error',
                    verbose = 1,
                    n_jobs = -1
                    )

grid.fit(X_train,y_train)
```

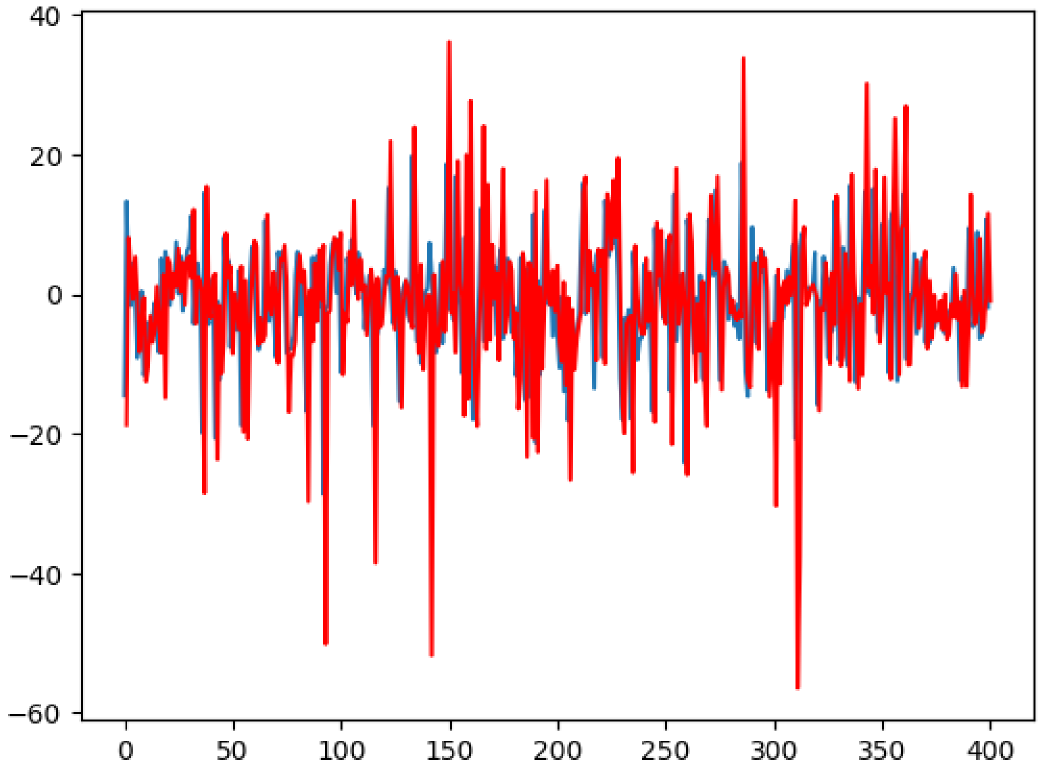
```
Fitting 10 folds for each of 600 candidates, totalling 6000 fits
```

[101]:
```
GridSearchCV(cv=10,
             estimator=Pipeline(steps=[('processor',
                                        Pipeline(steps=[('imputer',
SimpleImputer(strategy='median')),
                                                        ('scaler',
                                                         StandardScaler())])),
                                       ('model', SVR())]),
             n_jobs=-1,
             param_grid={'model__C': array([ 1.        ,  1.31034483,
1.62068966,  1.93103448,  2.24137931,
        2.55172414,  2.86206897,  3.17241379,  3.48275862,  3.79310345,
        4.10344828,  4.4137931 ,  4.72413793,  5.034482…
        7.20689655,  7.51724138,  7.82758621,  8.13793103,  8.44827586,
        8.75862069,  9.06896552,  9.37931034,  9.68965517, 10.        ]),
                         'model__epsilon': array([0.01      , 0.06210526,
0.11421053, 0.16631579, 0.21842105,
       0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737,
       0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368,
       0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.        ])},
             scoring='neg_mean_squared_error', verbose=1)
```

[102]:
```
grid.best_params_
start = time.perf_counter()
grid.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)
```

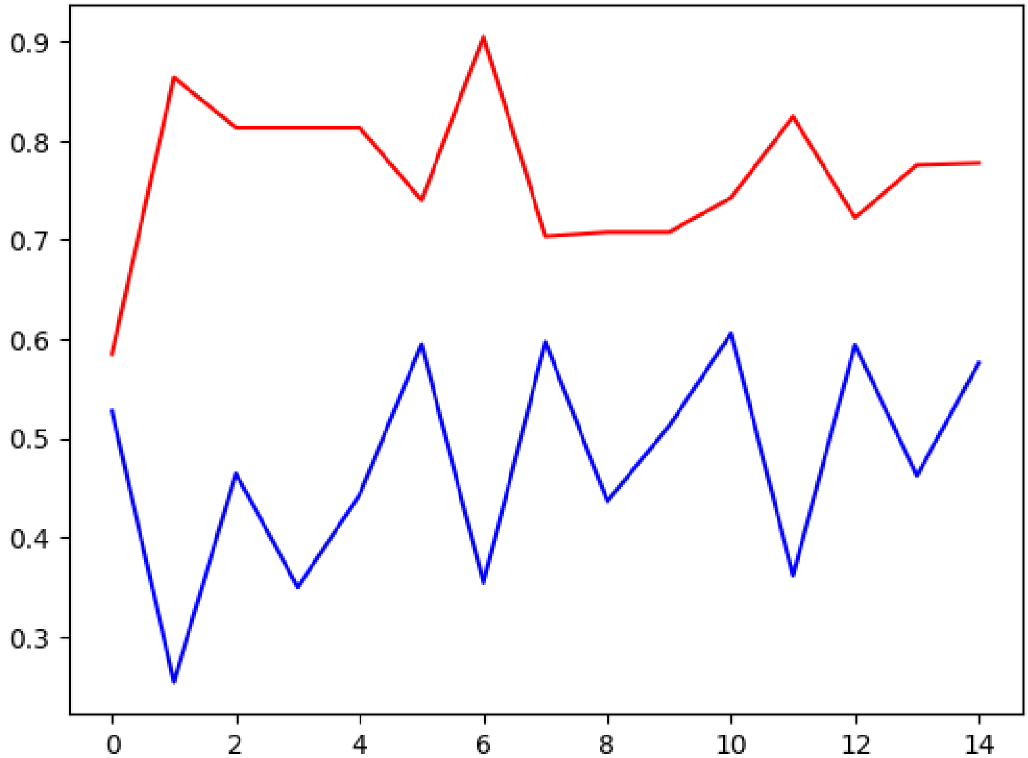
[103]:
```
training_data_fit_display(grid,X_train,y_train)
```

```
Here is the MSE score:
5.252533495200473
```



[104]:

```
best_svm = grid.best_estimator_
make_train_vs_test_curves(best_svm,X_train,y_train)

start = time.perf_counter()
best_svm.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)
```



Summary:

This model performs quite well, with a consistantly higher testing score over training in validation, along with a good MSE.

However, when submited to Kaggle the model didnt perform as well as I hoped. I believe that it was either overfitting or a lack of model complexity.

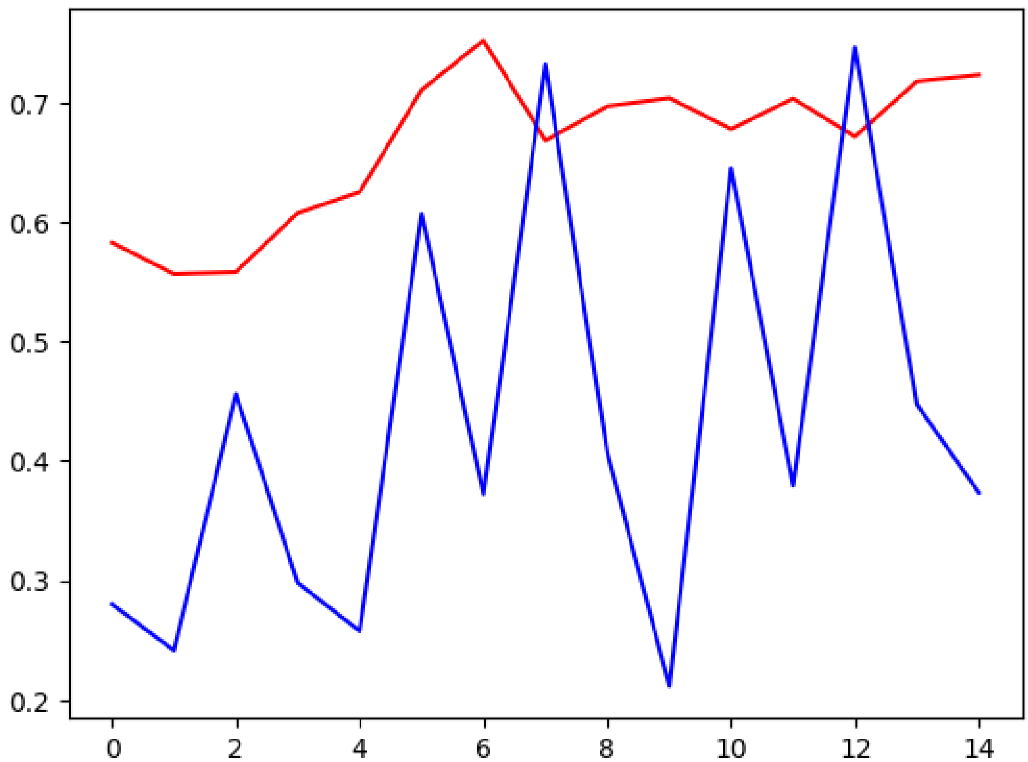
7 MODEL 3: RANDOM FOREST REGRESSION

[105]:
```
from sklearn.ensemble import RandomForestRegressor
```

[106]:
```
test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('model',RandomForestRegressor())
])
```

[107]:
```
from sklearn.model_selection import RandomizedSearchCV

param_dic = {
    'model__n_estimators': range(20,250),
    'model__max_depth': range(2,10),
```

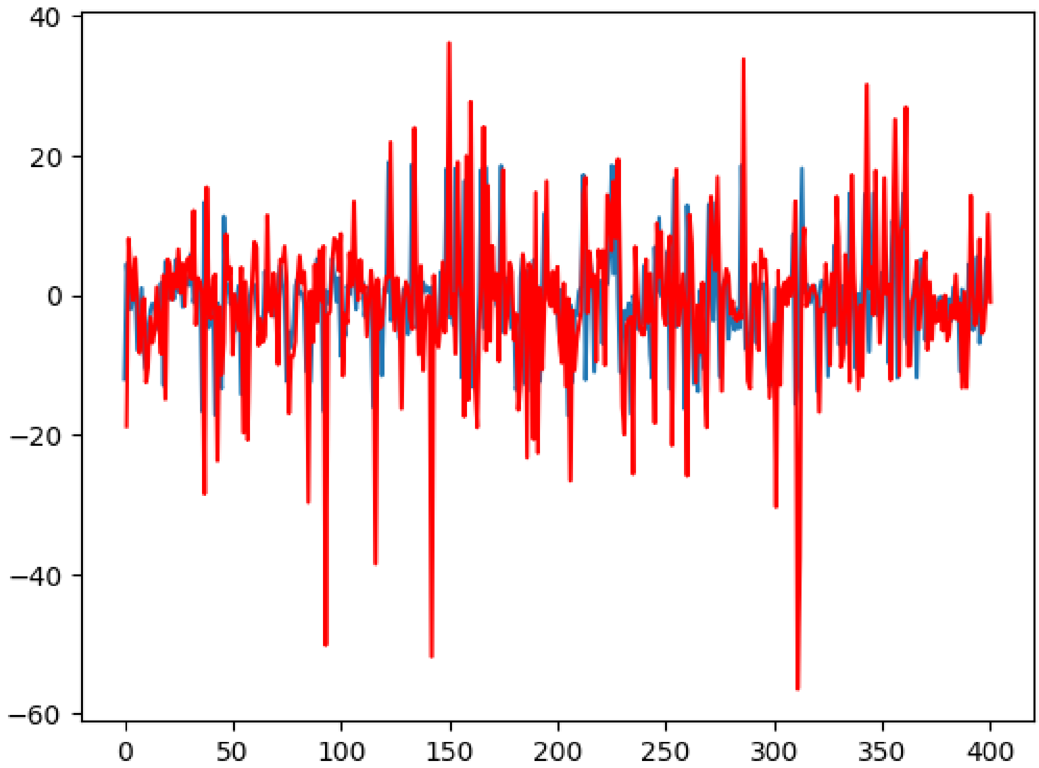
```
    'model__min_samples_split': range(10,250), #the metrics used for random␣
↪forest reg
    'model__min_samples_leaf': range(5,250)
}
grid = RandomizedSearchCV(test_pipeline,
                          param_dic,
                          n_iter=50,
                          scoring='neg_mean_squared_error',
                          n_jobs=-1,
                          cv=5
)
grid.fit(X_train,y_train)
best_random_forest_reg = grid.best_estimator_
make_train_vs_test_curves(best_random_forest_reg,X_train,y_train)
```



[108]:
```
training_data_fit_display(best_random_forest_reg,X_train,y_train)
start = time.perf_counter()
best_random_forest_reg.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)
```

```
start = time.perf_counter()
best_random_forest_reg.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)
```

```
Here is the MSE score:
6.06620035143386
```



Summary:

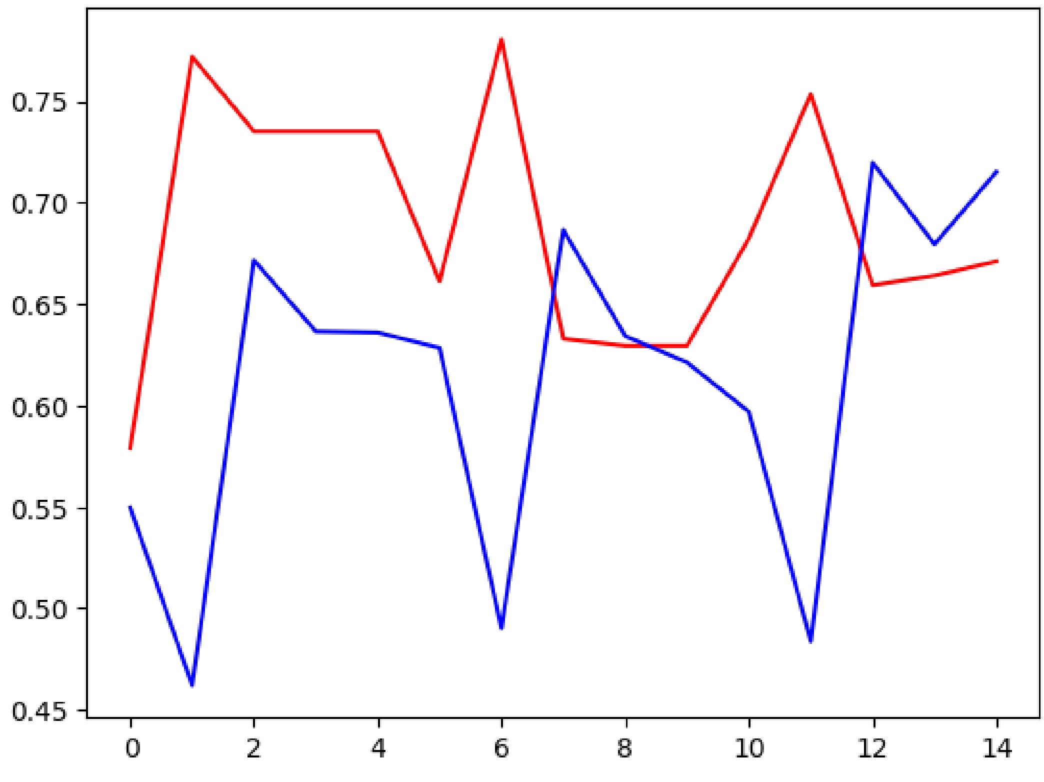
This model didnt do quite well, the MSE score was bad, the train test validation returned poor results and most importantly submiting to kaggle returned absymal results, so im going to drop pursing this model.

8 MODEL 4: RIDGE REGRESSION WITH POLYNOMIAL FEATURES

[109]:
```
from sklearn.linear_model import Ridge

test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('model',Ridge())
```

```
])

params_dict = {
    "model__alpha":np.linspace(0,1,50),
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )
grid.fit(X_train,y_train)
best_ridge = grid.best_estimator_
make_train_vs_test_curves(best_ridge,X_train,y_train)
```

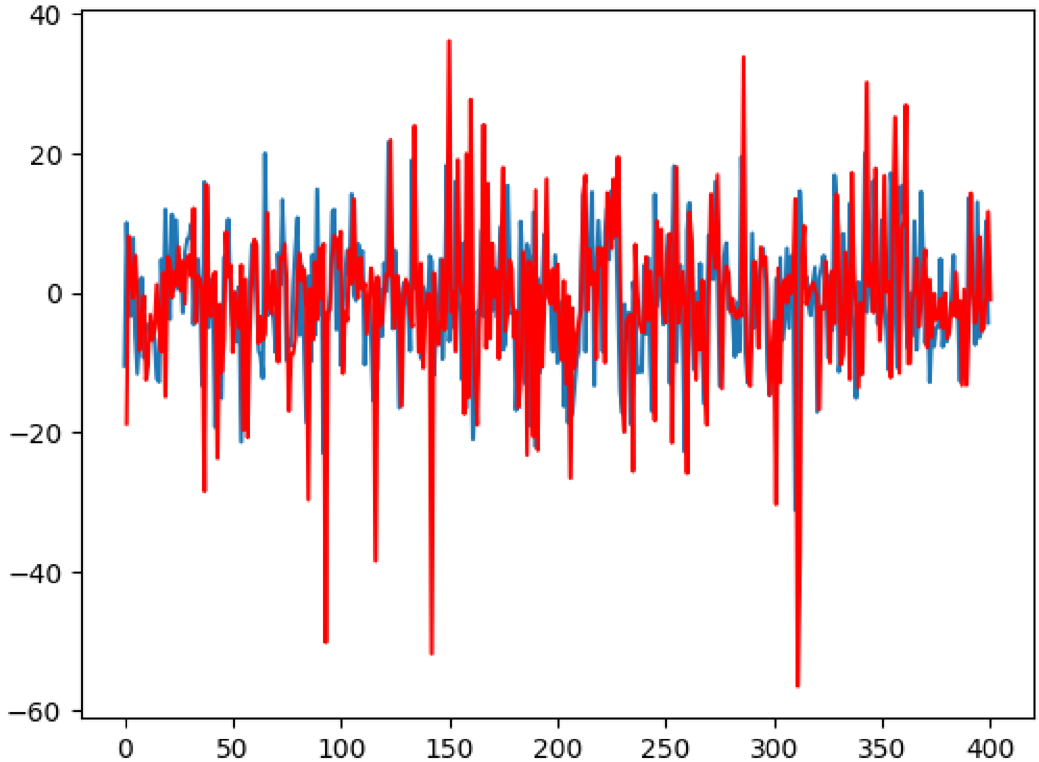
```
Fitting 10 folds for each of 50 candidates, totalling 500 fits
```



[110]:
```
training_data_fit_display(best_ridge,X_train,y_train)
start = time.perf_counter()
best_ridge.predict(X_train)
end = time.perf_counter()
```

```
time_taken_to_predict.append(end-start)


start = time.perf_counter()
best_ridge.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)
```

```
Here is the MSE score:
6.213283783436817
```



Summary:

This model, while intially not seeming to perform as well as SVM in MSE, and train test validation, it seems this model was much better at generalizing, because submitting it to kaggle produced quite good results, as such, a second round of training was performed in a attempt to better nail done the ideal params

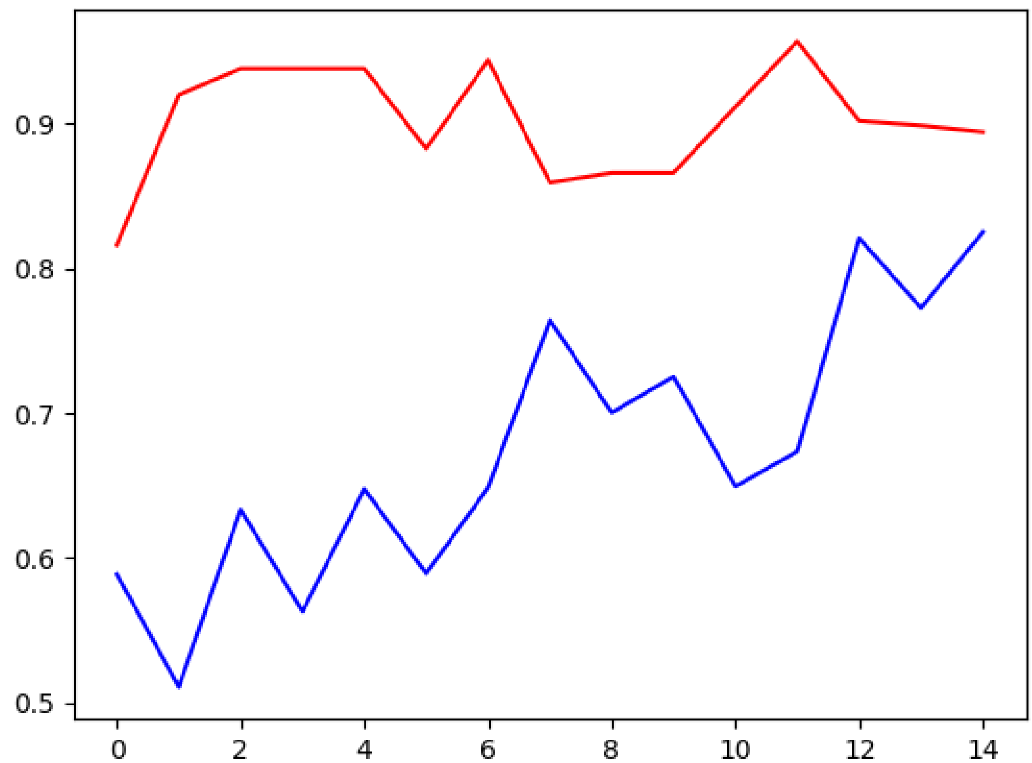
9 MODEL 4.2 MORE TRAINING

Same setup as above, but more time was taken to train, with a wider range of values. I also manuelly narrowed the range of params over repeated iterations based on what the grid search would converge to so that i could get even better values

[111]:

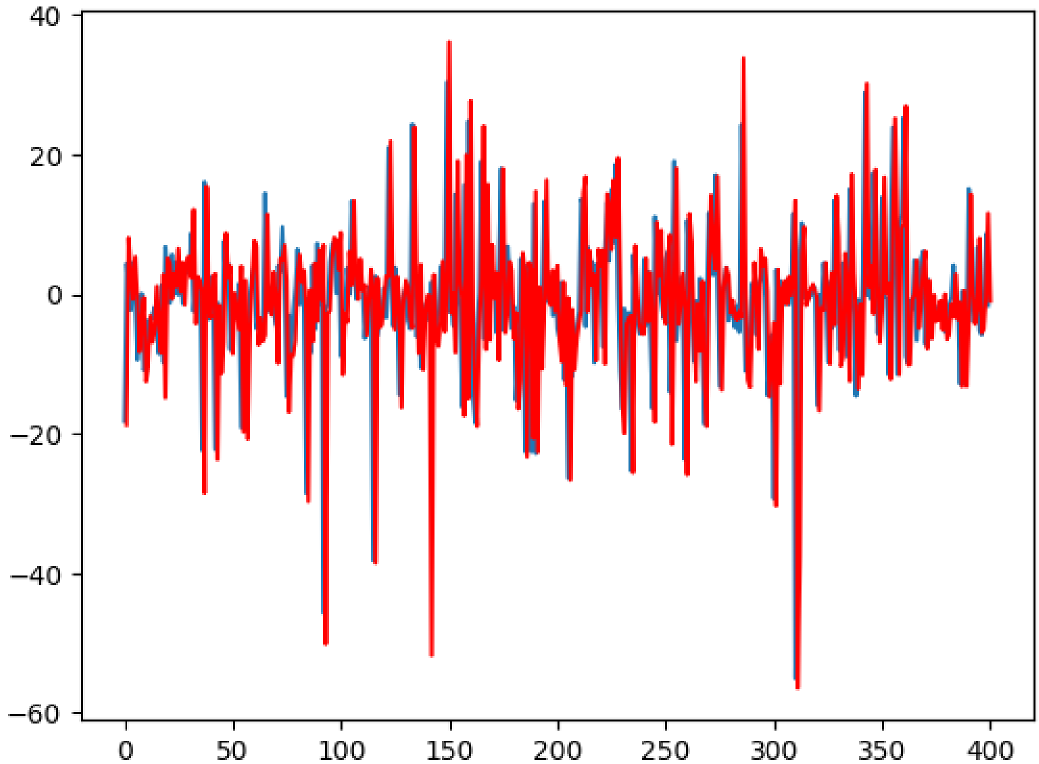
```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA
test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('poly',PolynomialFeatures()),
    ('model',Ridge())
])

params_dict = {
    "model__alpha":np.linspace(1100,1200,500),  #within this range is what i␣
↪eventually settled on as being ideal
    "poly__degree":[3]
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )

grid.fit(X_train,y_train)
best_ridge_2 = grid.best_estimator_

start = time.perf_counter()
best_ridge_2.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)


start = time.perf_counter()
grid.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)


make_train_vs_test_curves(best_ridge_2,X_train,y_train)
```

```
Fitting 10 folds for each of 500 candidates, totalling 5000 fits
```



[112]:
```
training_data_fit_display(best_ridge_2,X_train,y_train)
```

```
Here is the MSE score:
3.151414116911473
```




```
[113]: grid.best_params_ #these are the params that eventualy was settled on
```

```
[113]: {'model__alpha': np.float64(1154.308617234469), 'poly__degree': 3}
```

Summary:

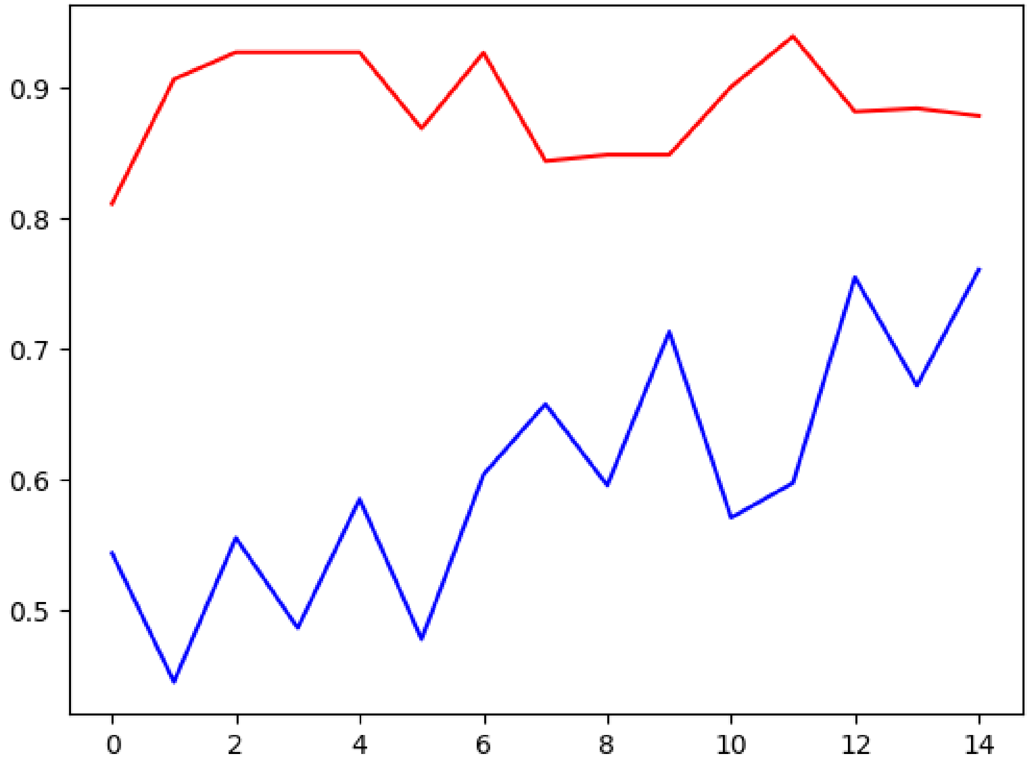
The performance here was great, not only were my metrics improved (mainly the train test validation) but the kaggle performance improved as well, so in a attempt to gain better performance, because i seem to have made the params as ideal as i can, im going to try using PCA.

10 MODEL 4.3 SAME AS ABOVE BUT THIS TIME WITH PCA

```
[114]: from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA
test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('pca',PCA()),
    ('poly',PolynomialFeatures()),
    ('model',Ridge())
])
```

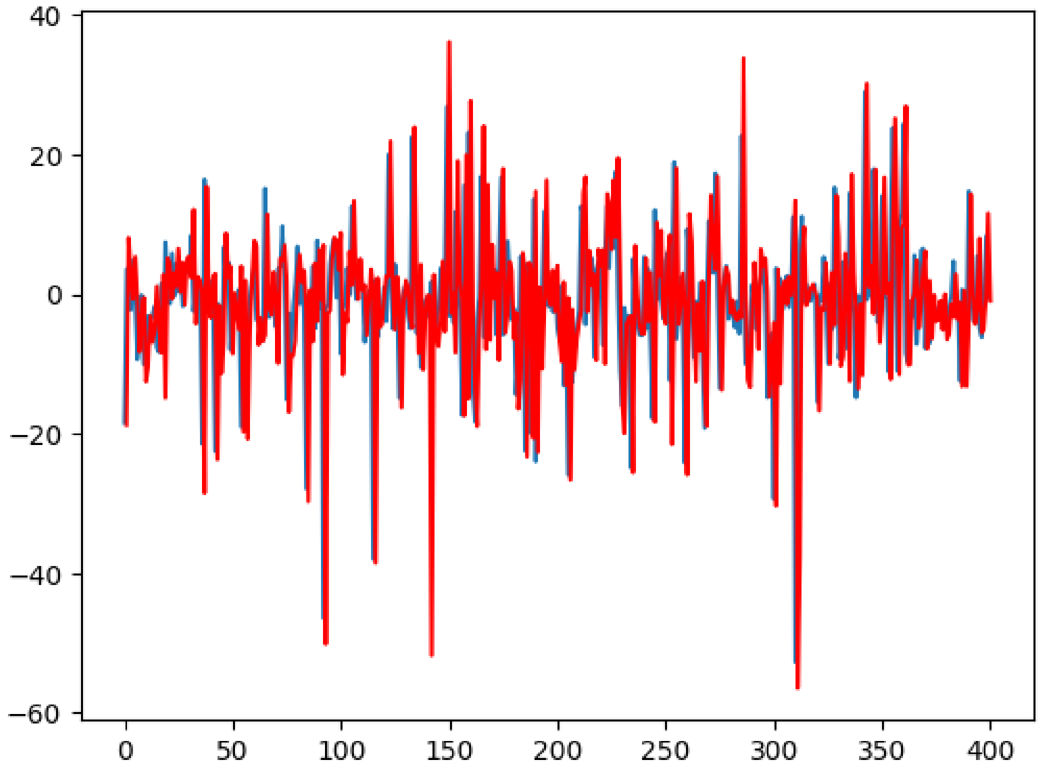
```
params_dict = {
    "model__alpha":np.linspace(1300,1400,500),
    "poly__degree":[3],
    "pca__n_components":[17] #these are only after several training goes which␣
 ↪seemed to have the grid search converging towards these values
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )
grid.fit(X_train,y_train)
best_ridge_3 = grid.best_estimator_

start = time.perf_counter()
best_ridge_3.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)


start = time.perf_counter()
grid.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)


make_train_vs_test_curves(best_ridge_3,X_train,y_train)
```

```
Fitting 10 folds for each of 500 candidates, totalling 5000 fits
```



[115]:
```
training_data_fit_display(best_ridge_3,X_train,y_train)
```

```
Here is the MSE score:
3.472270916747489
```



[116]:
```
grid.best_params_ #best values
```

[116]:
```
{'model__alpha': np.float64(1306.8136272545091),
 'pca__n_components': 17,
 'poly__degree': 3}
```

Summary:

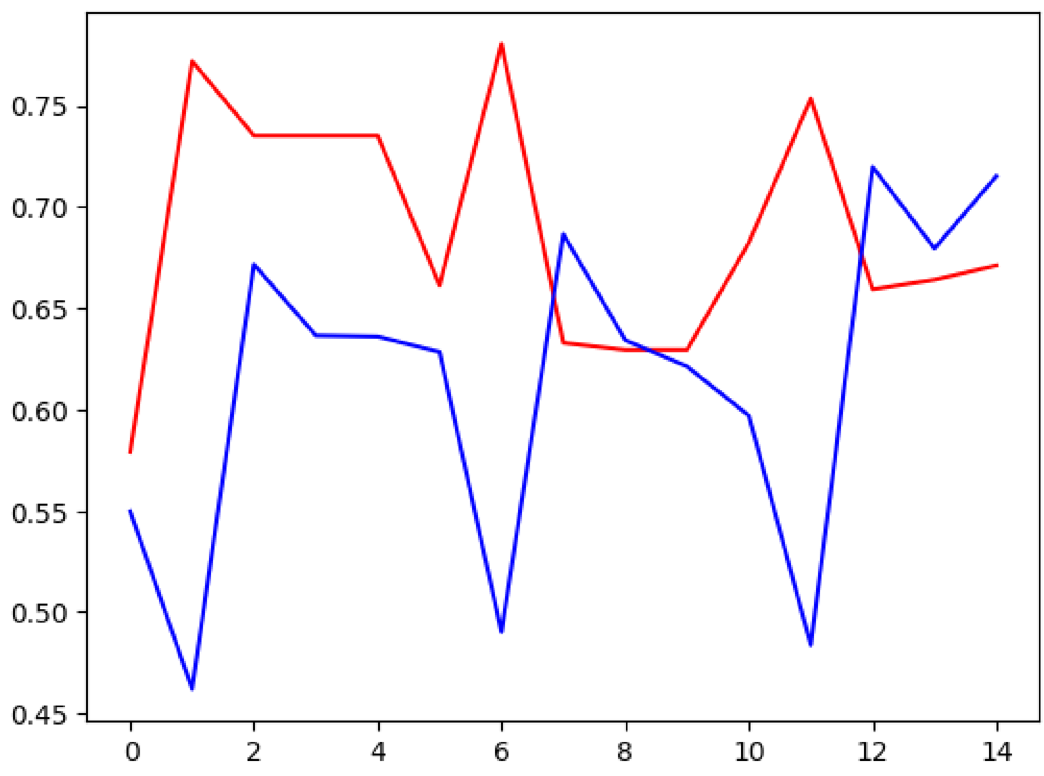
The performance of this model was NOT improved by PCA, it seems that all the dimensions of the data contribute, because even the reduction of 3 dims hurt quite badly the performance of the model, as such im going to not use PCA at all from here on out.

11 MODEL 5 DECISION TREE REGRESSION

[117]:
```
from sklearn.tree import DecisionTreeRegressor

test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('model',DecisionTreeRegressor())
])

params_dict={
```

```
    'model__max_depth':range(1,5),
    'model__min_samples_split':range(5,50,2),
    'model__min_samples_leaf':range(5,50,2),
    'model__max_leaf_nodes':[None,3,5,10],
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )
grid.fit(X_train,y_train)
best_tree = grid.best_estimator_
make_train_vs_test_curves(best_ridge,X_train,y_train)
```

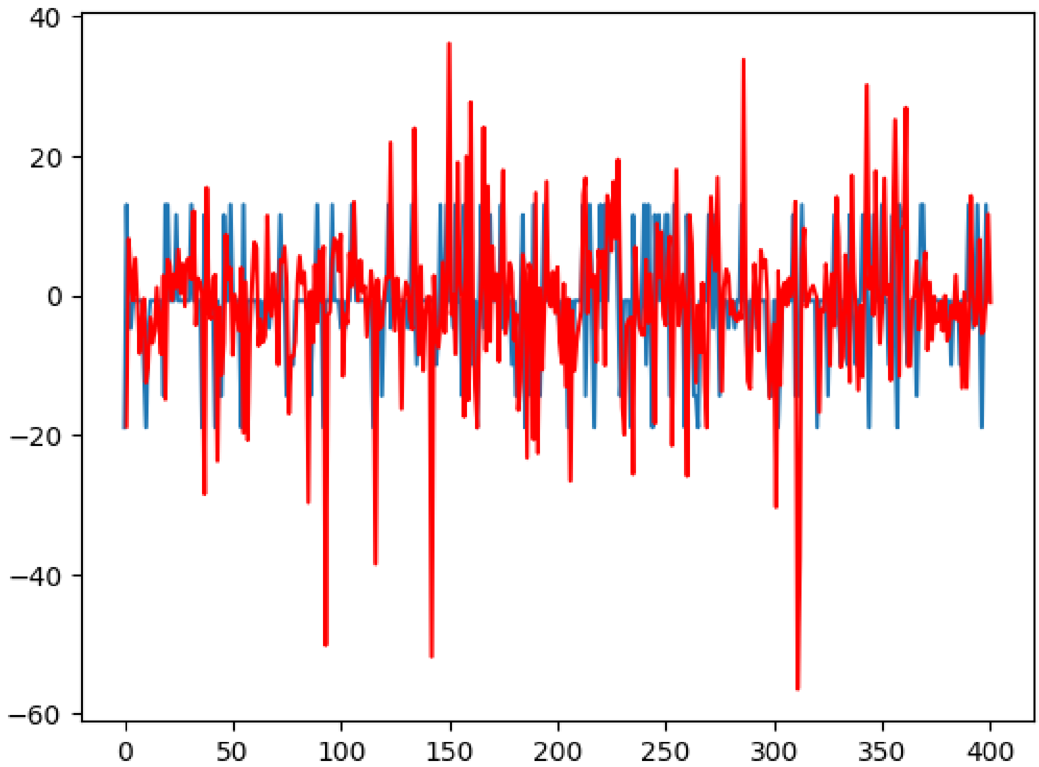
```
Fitting 10 folds for each of 8464 candidates, totalling 84640 fits
```



[118]:
```
training_data_fit_display(best_tree,X_train,y_train)
start = time.perf_counter()
best_tree.predict(X_train)
end = time.perf_counter()
```

```
time_taken_to_predict.append(end-start)

start = time.perf_counter()
best_tree.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)
```

```
Here is the MSE score:
7.4110216300609
```



Summary:

This model performed at the end of the day quite bad, compared to other models. This makes sense though, if a random forest performed poorly, then a regular decision tree is also more then likely to perform pooerly as well.

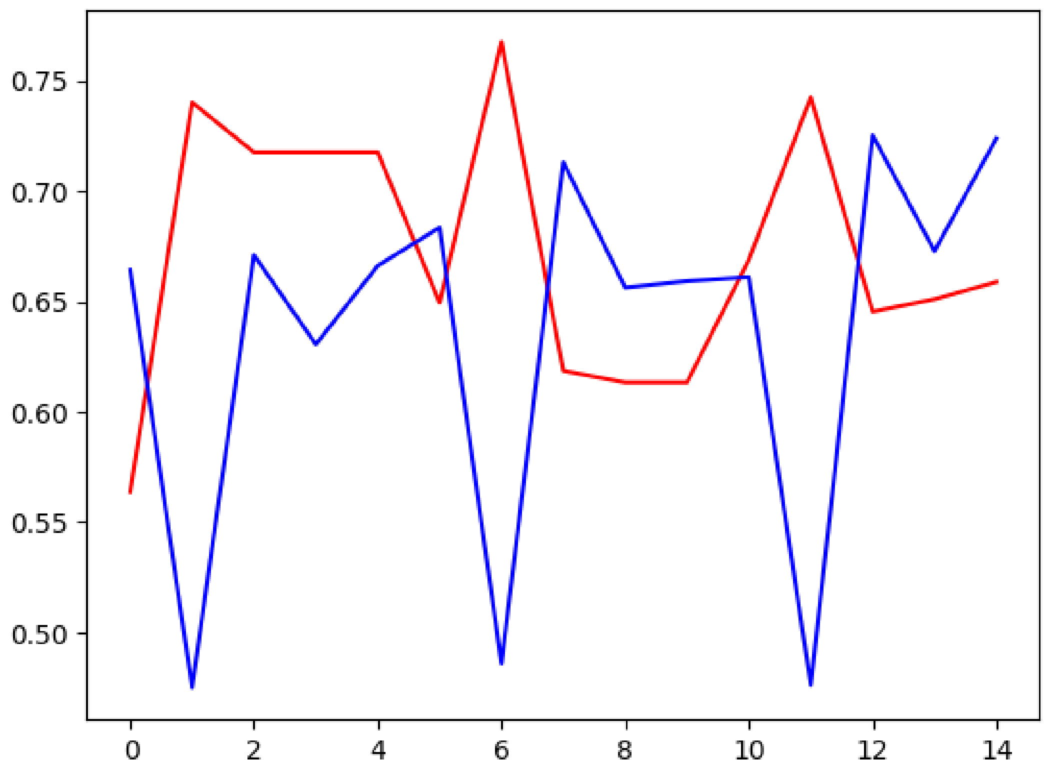
12 MODEL 6 LASSO

Without polynomial features

[119]:
```
from sklearn.linear_model import Lasso

test_pipeline = Pipeline(steps=[
```

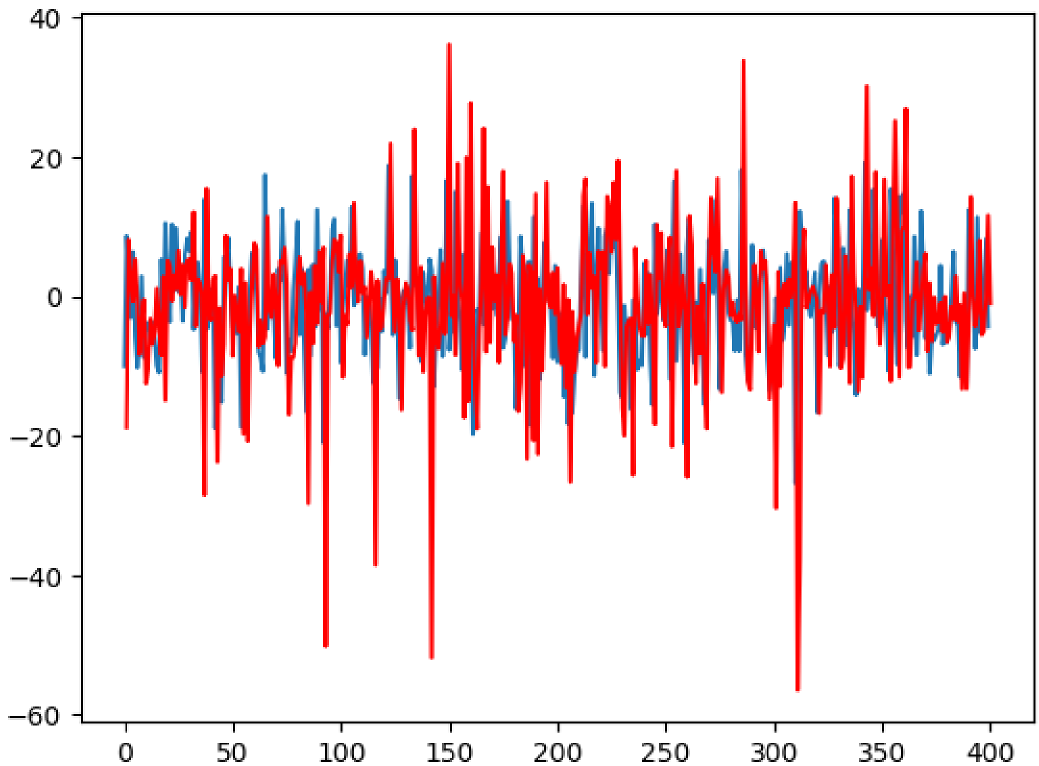
```
    ('processor',feature_processor),
    ('model',Lasso())
])

params_dict = {
    "model__alpha":np.linspace(0.001,1,75),
    "model__fit_intercept":[False,True]
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                     verbose = 1,
                    n_jobs = -1
                    )
grid.fit(X_train,y_train)
best_lasso = grid.best_estimator_
make_train_vs_test_curves(best_lasso,X_train,y_train)
```

```
Fitting 10 folds for each of 150 candidates, totalling 1500 fits
```



[120]:
```
training_data_fit_display(best_lasso,X_train,y_train)
start = time.perf_counter()
best_lasso.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)


start = time.perf_counter()
best_lasso.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)
```

```
Here is the MSE score:
6.330678391528587
```



Summary:

This model performed surprising well. While the metrics i tested it on (seen above) didnt show AMAZING performance, the kaggle score was the best that i had gotten up to this point, so i continued to follow done this model type.

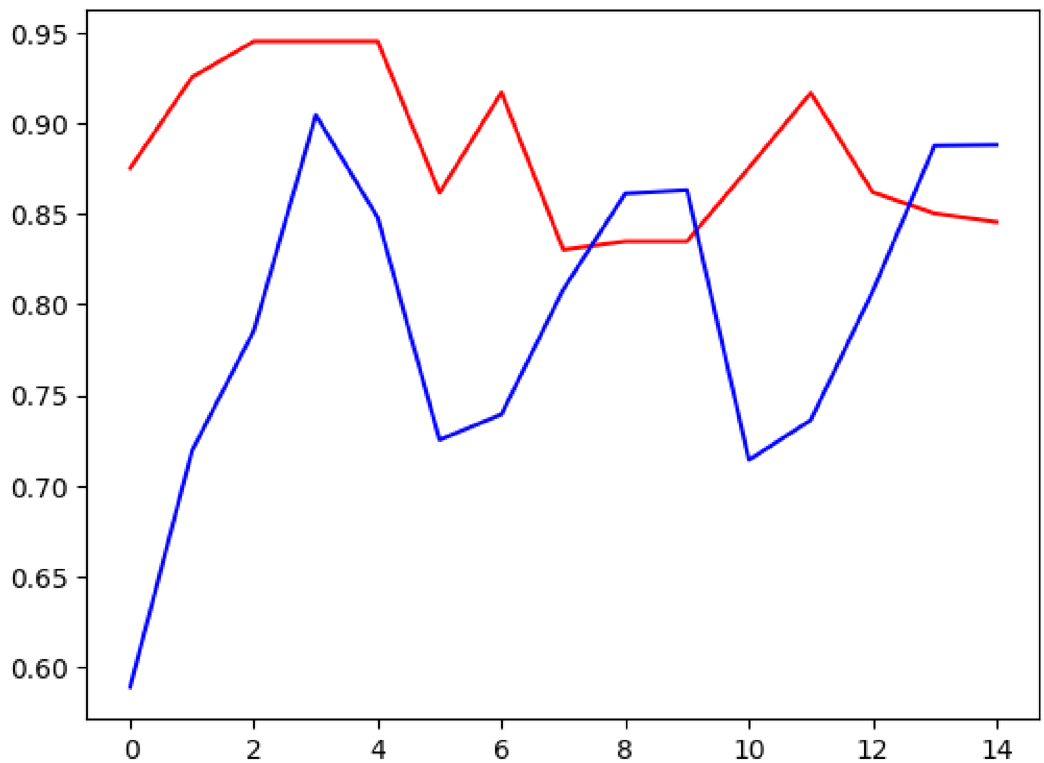
13 MODEL 6.2 MORE TRAINING ON THE LASSO

This time with polynomial features

As I did on Model 4.2 (ridge reg with polynomial features) i manuelly narrowed the range of params that i used based on what the model was converging towards to get better results

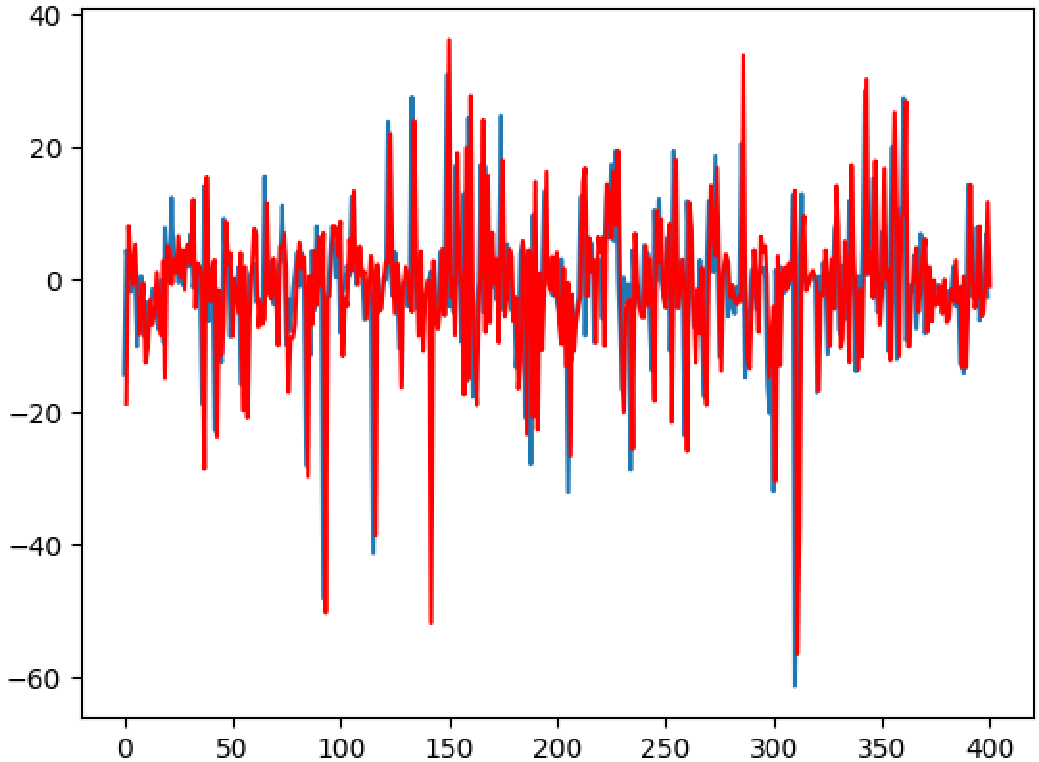
[121]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA

test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('poly',PolynomialFeatures()),
    ('model',Lasso())
])

params_dict = {
    "model__alpha":np.linspace(.25,.75,100),
    "poly__degree":[3,4],
    "model__max_iter":[5000]
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )

grid.fit(X_train,y_train)
best_lasso_2 = grid.best_estimator_


start = time.perf_counter()
best_lasso_2.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)


start = time.perf_counter()
grid.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)



make_train_vs_test_curves(best_lasso_2,X_train,y_train)
```

```
Fitting 10 folds for each of 200 candidates, totalling 2000 fits
```



[122]:
```
training_data_fit_display(best_lasso_2,X_train,y_train)
```

```
Here is the MSE score:
4.044953021119381
```



[123]:
```
grid.best_params_ #final best params
```

[123]:
```
{'model__alpha': np.float64(0.5025252525252526),
 'model__max_iter': 5000,
 'poly__degree': 3}
```

Summary:

This was the best model that I could find, performing miles better then any other model in Kaggle along with the train test validation scores, as such this is my final best Model.

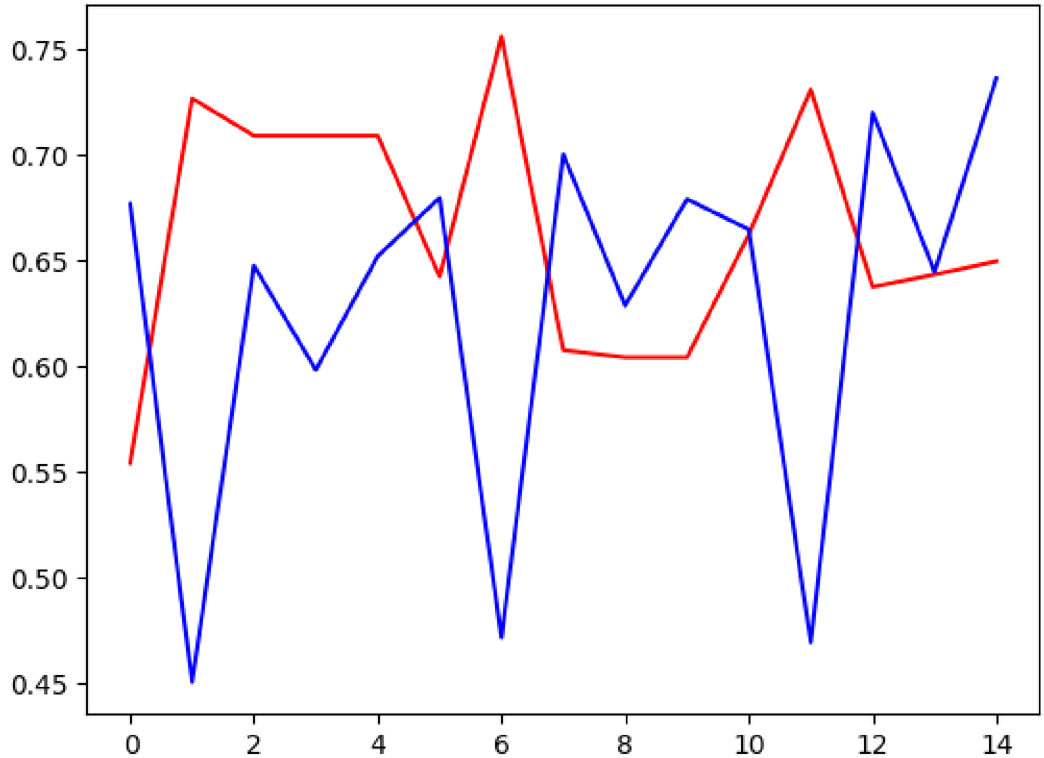
14 MODEL 7 ELASTIC NET

Without polynomial features

[124]:
```
from sklearn.linear_model import ElasticNet
test_pipeline = Pipeline(steps=[
    ('processor',feature_processor),
    ('model',ElasticNet())
])

params_dict = {
    "model__alpha":np.linspace(.02,1,75),
```

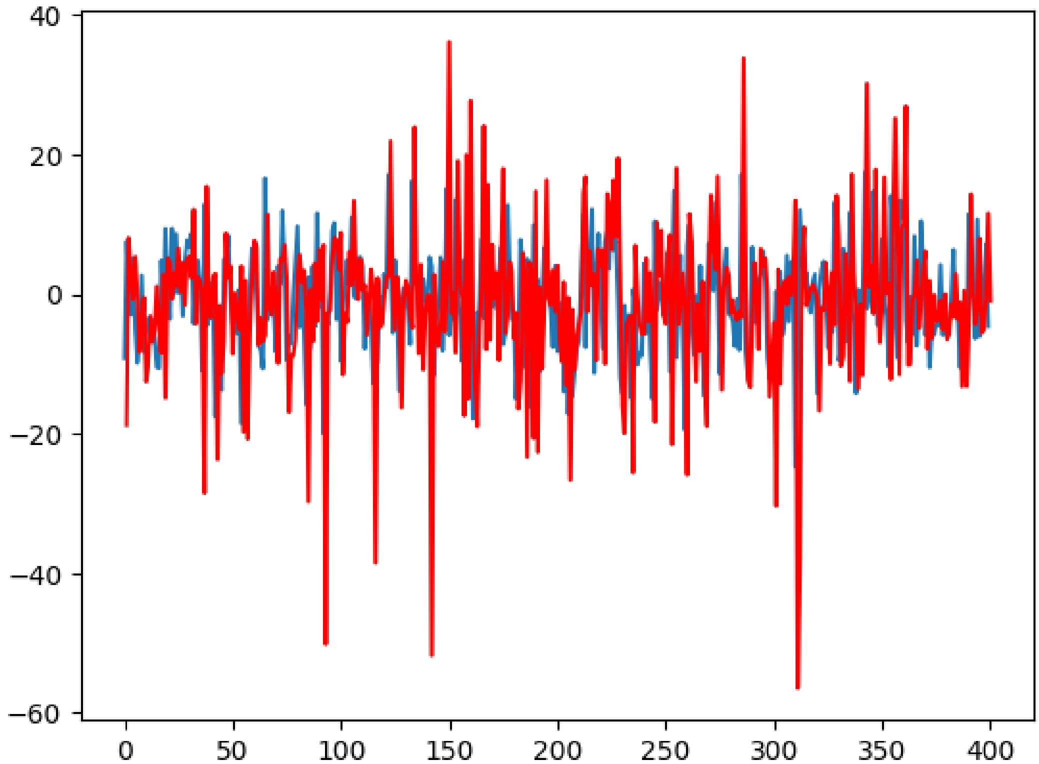
```
    "model__l1_ratio":np.linspace(.02,1,75),
    "model__fit_intercept":[False,True]
}
grid = GridSearchCV(test_pipeline,
                    params_dict,
                    cv = 10,
                    scoring = 'neg_mean_absolute_error',
                    verbose = 1,
                    n_jobs = -1
                    )

grid.fit(X_train,y_train)
best_elastic = grid.best_estimator_

start = time.perf_counter()
best_elastic.fit(X_train,y_train)
end = time.perf_counter()
time_taken_to_train.append(end-start)


start = time.perf_counter()
grid.predict(X_train)
end = time.perf_counter()
time_taken_to_predict.append(end-start)


make_train_vs_test_curves(best_elastic,X_train,y_train)
```

```
Fitting 10 folds for each of 11250 candidates, totalling 112500 fits
```



[125]:
```
training_data_fit_display(best_elastic,X_train,y_train)
```

```
Here is the MSE score:
6.411331395949554
```



Summary:

After the performance of the ridge and lasso i figured testing both combined would be ideal, however it didnt pruduce the best results, being out performed but both ridge and lasso regresssion individually, so i abandoned this route for following ridge and lasso by themselves

15 MODEL 8 NN

This one i manuelly narrowed the range of how many layers to use based on kaggle performance. I didnt have high hopes for this model type, and it didnt perform saddly.

[126]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from numpy.random import seed
import tensorflow
tensorflow.random.set_seed(42)

y_train_np = feature_processor.fit_transform(pd.DataFrame(y_train))
X_train_np = feature_processor.fit_transform(X_train)
```

[127]:
```python
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(10, activation='relu', input_shape=(20,)),
    layers.Dense(1) # Output layer for regression
])
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
history = model.fit(X_train_np, y_train_np, epochs=100, batch_size=32,
 ↪validation_split=0.2)
```

```
Epoch 1/100

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/keras/src/layers/core/dense.py:92: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

10/10            0s 10ms/step -
loss: 1.4544 - mae: 0.9397 - val_loss: 1.5314 - val_mae: 0.9449
Epoch 2/100
10/10            0s 3ms/step - loss:
1.3135 - mae: 0.8829 - val_loss: 1.3798 - val_mae: 0.8986
Epoch 3/100
10/10            0s 3ms/step - loss:
1.1979 - mae: 0.8329 - val_loss: 1.2512 - val_mae: 0.8558
Epoch 4/100
10/10            0s 3ms/step - loss:
1.1007 - mae: 0.7901 - val_loss: 1.1422 - val_mae: 0.8169
Epoch 5/100
10/10            0s 3ms/step - loss:
1.0196 - mae: 0.7532 - val_loss: 1.0501 - val_mae: 0.7826
Epoch 6/100
10/10            0s 3ms/step - loss:
0.9517 - mae: 0.7215 - val_loss: 0.9724 - val_mae: 0.7522
Epoch 7/100
10/10            0s 7ms/step - loss:
0.8944 - mae: 0.6944 - val_loss: 0.9069 - val_mae: 0.7245
Epoch 8/100
10/10            0s 3ms/step - loss:
0.8451 - mae: 0.6703 - val_loss: 0.8500 - val_mae: 0.6989
Epoch 9/100
10/10            0s 3ms/step - loss:
0.8022 - mae: 0.6486 - val_loss: 0.7990 - val_mae: 0.6753
Epoch 10/100
10/10            0s 3ms/step - loss:
0.7643 - mae: 0.6285 - val_loss: 0.7526 - val_mae: 0.6536
Epoch 11/100
```

```
10/10            0s 3ms/step - loss:
0.7305 - mae: 0.6099 - val_loss: 0.7130 - val_mae: 0.6334
Epoch 12/100
10/10            0s 3ms/step - loss:
0.6991 - mae: 0.5927 - val_loss: 0.6778 - val_mae: 0.6153
Epoch 13/100
10/10            0s 3ms/step - loss:
0.6711 - mae: 0.5775 - val_loss: 0.6466 - val_mae: 0.5985
Epoch 14/100
10/10            0s 3ms/step - loss:
0.6460 - mae: 0.5642 - val_loss: 0.6186 - val_mae: 0.5832
Epoch 15/100
10/10            0s 3ms/step - loss:
0.6235 - mae: 0.5524 - val_loss: 0.5936 - val_mae: 0.5693
Epoch 16/100
10/10            0s 3ms/step - loss:
0.6030 - mae: 0.5418 - val_loss: 0.5710 - val_mae: 0.5576
Epoch 17/100
10/10            0s 3ms/step - loss:
0.5844 - mae: 0.5318 - val_loss: 0.5501 - val_mae: 0.5477
Epoch 18/100
10/10            0s 3ms/step - loss:
0.5673 - mae: 0.5228 - val_loss: 0.5310 - val_mae: 0.5381
Epoch 19/100
10/10            0s 3ms/step - loss:
0.5512 - mae: 0.5144 - val_loss: 0.5135 - val_mae: 0.5290
Epoch 20/100
10/10            0s 3ms/step - loss:
0.5364 - mae: 0.5064 - val_loss: 0.4975 - val_mae: 0.5210
Epoch 21/100
10/10            0s 3ms/step - loss:
0.5226 - mae: 0.4989 - val_loss: 0.4831 - val_mae: 0.5133
Epoch 22/100
10/10            0s 3ms/step - loss:
0.5099 - mae: 0.4917 - val_loss: 0.4698 - val_mae: 0.5063
Epoch 23/100
10/10            0s 3ms/step - loss:
0.4979 - mae: 0.4848 - val_loss: 0.4577 - val_mae: 0.5002
Epoch 24/100
10/10            0s 3ms/step - loss:
0.4866 - mae: 0.4784 - val_loss: 0.4466 - val_mae: 0.4945
Epoch 25/100
10/10            0s 3ms/step - loss:
0.4760 - mae: 0.4724 - val_loss: 0.4366 - val_mae: 0.4889
Epoch 26/100
10/10            0s 3ms/step - loss:
0.4661 - mae: 0.4670 - val_loss: 0.4274 - val_mae: 0.4840
Epoch 27/100
```

```
10/10            0s 3ms/step - loss:
0.4566 - mae: 0.4620 - val_loss: 0.4189 - val_mae: 0.4791
Epoch 28/100
10/10            0s 3ms/step - loss:
0.4477 - mae: 0.4573 - val_loss: 0.4109 - val_mae: 0.4742
Epoch 29/100
10/10            0s 3ms/step - loss:
0.4393 - mae: 0.4530 - val_loss: 0.4036 - val_mae: 0.4694
Epoch 30/100
10/10            0s 3ms/step - loss:
0.4313 - mae: 0.4487 - val_loss: 0.3968 - val_mae: 0.4649
Epoch 31/100
10/10            0s 3ms/step - loss:
0.4236 - mae: 0.4446 - val_loss: 0.3905 - val_mae: 0.4610
Epoch 32/100
10/10            0s 3ms/step - loss:
0.4164 - mae: 0.4406 - val_loss: 0.3843 - val_mae: 0.4570
Epoch 33/100
10/10            0s 3ms/step - loss:
0.4094 - mae: 0.4366 - val_loss: 0.3786 - val_mae: 0.4532
Epoch 34/100
10/10            0s 3ms/step - loss:
0.4027 - mae: 0.4327 - val_loss: 0.3733 - val_mae: 0.4498
Epoch 35/100
10/10            0s 3ms/step - loss:
0.3962 - mae: 0.4290 - val_loss: 0.3682 - val_mae: 0.4464
Epoch 36/100
10/10            0s 3ms/step - loss:
0.3899 - mae: 0.4257 - val_loss: 0.3637 - val_mae: 0.4433
Epoch 37/100
10/10            0s 3ms/step - loss:
0.3839 - mae: 0.4225 - val_loss: 0.3594 - val_mae: 0.4402
Epoch 38/100
10/10            0s 3ms/step - loss:
0.3781 - mae: 0.4193 - val_loss: 0.3556 - val_mae: 0.4372
Epoch 39/100
10/10            0s 3ms/step - loss:
0.3726 - mae: 0.4164 - val_loss: 0.3521 - val_mae: 0.4345
Epoch 40/100
10/10            0s 3ms/step - loss:
0.3674 - mae: 0.4136 - val_loss: 0.3490 - val_mae: 0.4320
Epoch 41/100
10/10            0s 3ms/step - loss:
0.3624 - mae: 0.4109 - val_loss: 0.3462 - val_mae: 0.4295
Epoch 42/100
10/10            0s 3ms/step - loss:
0.3577 - mae: 0.4085 - val_loss: 0.3436 - val_mae: 0.4272
Epoch 43/100
```

```
10/10              0s 3ms/step - loss:
0.3531 - mae: 0.4063 - val_loss: 0.3413 - val_mae: 0.4250
Epoch 44/100
10/10              0s 3ms/step - loss:
0.3488 - mae: 0.4041 - val_loss: 0.3390 - val_mae: 0.4229
Epoch 45/100
10/10              0s 3ms/step - loss:
0.3446 - mae: 0.4019 - val_loss: 0.3372 - val_mae: 0.4209
Epoch 46/100
10/10              0s 3ms/step - loss:
0.3404 - mae: 0.3997 - val_loss: 0.3356 - val_mae: 0.4191
Epoch 47/100
10/10              0s 3ms/step - loss:
0.3364 - mae: 0.3976 - val_loss: 0.3340 - val_mae: 0.4175
Epoch 48/100
10/10              0s 3ms/step - loss:
0.3326 - mae: 0.3956 - val_loss: 0.3324 - val_mae: 0.4162
Epoch 49/100
10/10              0s 3ms/step - loss:
0.3288 - mae: 0.3937 - val_loss: 0.3308 - val_mae: 0.4149
Epoch 50/100
10/10              0s 3ms/step - loss:
0.3253 - mae: 0.3918 - val_loss: 0.3291 - val_mae: 0.4134
Epoch 51/100
10/10              0s 3ms/step - loss:
0.3218 - mae: 0.3900 - val_loss: 0.3276 - val_mae: 0.4119
Epoch 52/100
10/10              0s 3ms/step - loss:
0.3185 - mae: 0.3883 - val_loss: 0.3263 - val_mae: 0.4110
Epoch 53/100
10/10              0s 3ms/step - loss:
0.3154 - mae: 0.3867 - val_loss: 0.3249 - val_mae: 0.4100
Epoch 54/100
10/10              0s 3ms/step - loss:
0.3124 - mae: 0.3851 - val_loss: 0.3237 - val_mae: 0.4090
Epoch 55/100
10/10              0s 3ms/step - loss:
0.3095 - mae: 0.3835 - val_loss: 0.3224 - val_mae: 0.4083
Epoch 56/100
10/10              0s 3ms/step - loss:
0.3068 - mae: 0.3820 - val_loss: 0.3215 - val_mae: 0.4077
Epoch 57/100
10/10              0s 3ms/step - loss:
0.3042 - mae: 0.3806 - val_loss: 0.3205 - val_mae: 0.4070
Epoch 58/100
10/10              0s 3ms/step - loss:
0.3017 - mae: 0.3790 - val_loss: 0.3195 - val_mae: 0.4062
Epoch 59/100
```

```
10/10            0s 3ms/step - loss:
0.2992 - mae: 0.3775 - val_loss: 0.3184 - val_mae: 0.4055
Epoch 60/100
10/10            0s 3ms/step - loss:
0.2969 - mae: 0.3760 - val_loss: 0.3175 - val_mae: 0.4049
Epoch 61/100
10/10            0s 3ms/step - loss:
0.2946 - mae: 0.3745 - val_loss: 0.3165 - val_mae: 0.4045
Epoch 62/100
10/10            0s 3ms/step - loss:
0.2924 - mae: 0.3730 - val_loss: 0.3158 - val_mae: 0.4044
Epoch 63/100
10/10            0s 3ms/step - loss:
0.2902 - mae: 0.3717 - val_loss: 0.3151 - val_mae: 0.4042
Epoch 64/100
10/10            0s 3ms/step - loss:
0.2882 - mae: 0.3704 - val_loss: 0.3145 - val_mae: 0.4041
Epoch 65/100
10/10            0s 3ms/step - loss:
0.2861 - mae: 0.3691 - val_loss: 0.3139 - val_mae: 0.4039
Epoch 66/100
10/10            0s 3ms/step - loss:
0.2842 - mae: 0.3678 - val_loss: 0.3137 - val_mae: 0.4039
Epoch 67/100
10/10            0s 3ms/step - loss:
0.2823 - mae: 0.3666 - val_loss: 0.3136 - val_mae: 0.4042
Epoch 68/100
10/10            0s 3ms/step - loss:
0.2804 - mae: 0.3654 - val_loss: 0.3134 - val_mae: 0.4043
Epoch 69/100
10/10            0s 3ms/step - loss:
0.2786 - mae: 0.3640 - val_loss: 0.3130 - val_mae: 0.4043
Epoch 70/100
10/10            0s 3ms/step - loss:
0.2769 - mae: 0.3627 - val_loss: 0.3127 - val_mae: 0.4044
Epoch 71/100
10/10            0s 3ms/step - loss:
0.2753 - mae: 0.3615 - val_loss: 0.3124 - val_mae: 0.4044
Epoch 72/100
10/10            0s 3ms/step - loss:
0.2736 - mae: 0.3604 - val_loss: 0.3119 - val_mae: 0.4042
Epoch 73/100
10/10            0s 3ms/step - loss:
0.2721 - mae: 0.3591 - val_loss: 0.3114 - val_mae: 0.4040
Epoch 74/100
10/10            0s 3ms/step - loss:
0.2706 - mae: 0.3581 - val_loss: 0.3111 - val_mae: 0.4040
Epoch 75/100
```

```
10/10            0s 3ms/step - loss:
0.2692 - mae: 0.3571 - val_loss: 0.3107 - val_mae: 0.4038
Epoch 76/100
10/10            0s 3ms/step - loss:
0.2677 - mae: 0.3560 - val_loss: 0.3104 - val_mae: 0.4037
Epoch 77/100
10/10            0s 3ms/step - loss:
0.2663 - mae: 0.3549 - val_loss: 0.3099 - val_mae: 0.4033
Epoch 78/100
10/10            0s 3ms/step - loss:
0.2649 - mae: 0.3538 - val_loss: 0.3097 - val_mae: 0.4033
Epoch 79/100
10/10            0s 3ms/step - loss:
0.2636 - mae: 0.3529 - val_loss: 0.3096 - val_mae: 0.4034
Epoch 80/100
10/10            0s 6ms/step - loss:
0.2622 - mae: 0.3519 - val_loss: 0.3093 - val_mae: 0.4033
Epoch 81/100
10/10            0s 3ms/step - loss:
0.2609 - mae: 0.3509 - val_loss: 0.3092 - val_mae: 0.4032
Epoch 82/100
10/10            0s 3ms/step - loss:
0.2597 - mae: 0.3499 - val_loss: 0.3089 - val_mae: 0.4031
Epoch 83/100
10/10            0s 3ms/step - loss:
0.2585 - mae: 0.3489 - val_loss: 0.3086 - val_mae: 0.4029
Epoch 84/100
10/10            0s 3ms/step - loss:
0.2573 - mae: 0.3480 - val_loss: 0.3087 - val_mae: 0.4029
Epoch 85/100
10/10            0s 3ms/step - loss:
0.2562 - mae: 0.3471 - val_loss: 0.3085 - val_mae: 0.4027
Epoch 86/100
10/10            0s 3ms/step - loss:
0.2551 - mae: 0.3461 - val_loss: 0.3084 - val_mae: 0.4025
Epoch 87/100
10/10            0s 3ms/step - loss:
0.2540 - mae: 0.3453 - val_loss: 0.3082 - val_mae: 0.4023
Epoch 88/100
10/10            0s 3ms/step - loss:
0.2529 - mae: 0.3444 - val_loss: 0.3081 - val_mae: 0.4022
Epoch 89/100
10/10            0s 3ms/step - loss:
0.2518 - mae: 0.3436 - val_loss: 0.3083 - val_mae: 0.4022
Epoch 90/100
10/10            0s 3ms/step - loss:
0.2507 - mae: 0.3428 - val_loss: 0.3081 - val_mae: 0.4021
Epoch 91/100
```

```
10/10               0s 3ms/step - loss:
0.2497 - mae: 0.3419 - val_loss: 0.3082 - val_mae: 0.4022
Epoch 92/100
10/10               0s 3ms/step - loss:
0.2487 - mae: 0.3411 - val_loss: 0.3086 - val_mae: 0.4025
Epoch 93/100
10/10               0s 3ms/step - loss:
0.2477 - mae: 0.3404 - val_loss: 0.3087 - val_mae: 0.4027
Epoch 94/100
10/10               0s 3ms/step - loss:
0.2467 - mae: 0.3397 - val_loss: 0.3087 - val_mae: 0.4027
Epoch 95/100
10/10               0s 3ms/step - loss:
0.2458 - mae: 0.3391 - val_loss: 0.3090 - val_mae: 0.4030
Epoch 96/100
10/10               0s 3ms/step - loss:
0.2449 - mae: 0.3385 - val_loss: 0.3092 - val_mae: 0.4032
Epoch 97/100
10/10               0s 3ms/step - loss:
0.2440 - mae: 0.3378 - val_loss: 0.3093 - val_mae: 0.4034
Epoch 98/100
10/10               0s 3ms/step - loss:
0.2431 - mae: 0.3372 - val_loss: 0.3096 - val_mae: 0.4037
Epoch 99/100
10/10               0s 3ms/step - loss:
0.2422 - mae: 0.3367 - val_loss: 0.3098 - val_mae: 0.4040
Epoch 100/100
10/10               0s 3ms/step - loss:
0.2414 - mae: 0.3361 - val_loss: 0.3099 - val_mae: 0.4041
```

[128]:
```
model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 10)	210
dense_7 (Dense)	(None, 1)	11

Total params: 665 (2.60 KB)

Trainable params: 221 (884.00 B)

```
Non-trainable params: 0 (0.00 B)
```

```
Optimizer params: 444 (1.74 KB)
```

[129]:
```
feature_processor=Pipeline(steps = [
    ('imputer',SimpleImputer(strategy='median')), #!MAY CHANGE THIS TO MEAN I␣
↪THINK
    ('scaler',StandardScaler())
    ])
df_testing_np =feature_processor.fit_transform(df_test)
df_testing_np.shape
```

[129]:
```
(800, 20)
```

[130]:
```
nn_pred = model.predict(df_testing_np)
```

```
25/25            0s 482us/step
```

Summary:

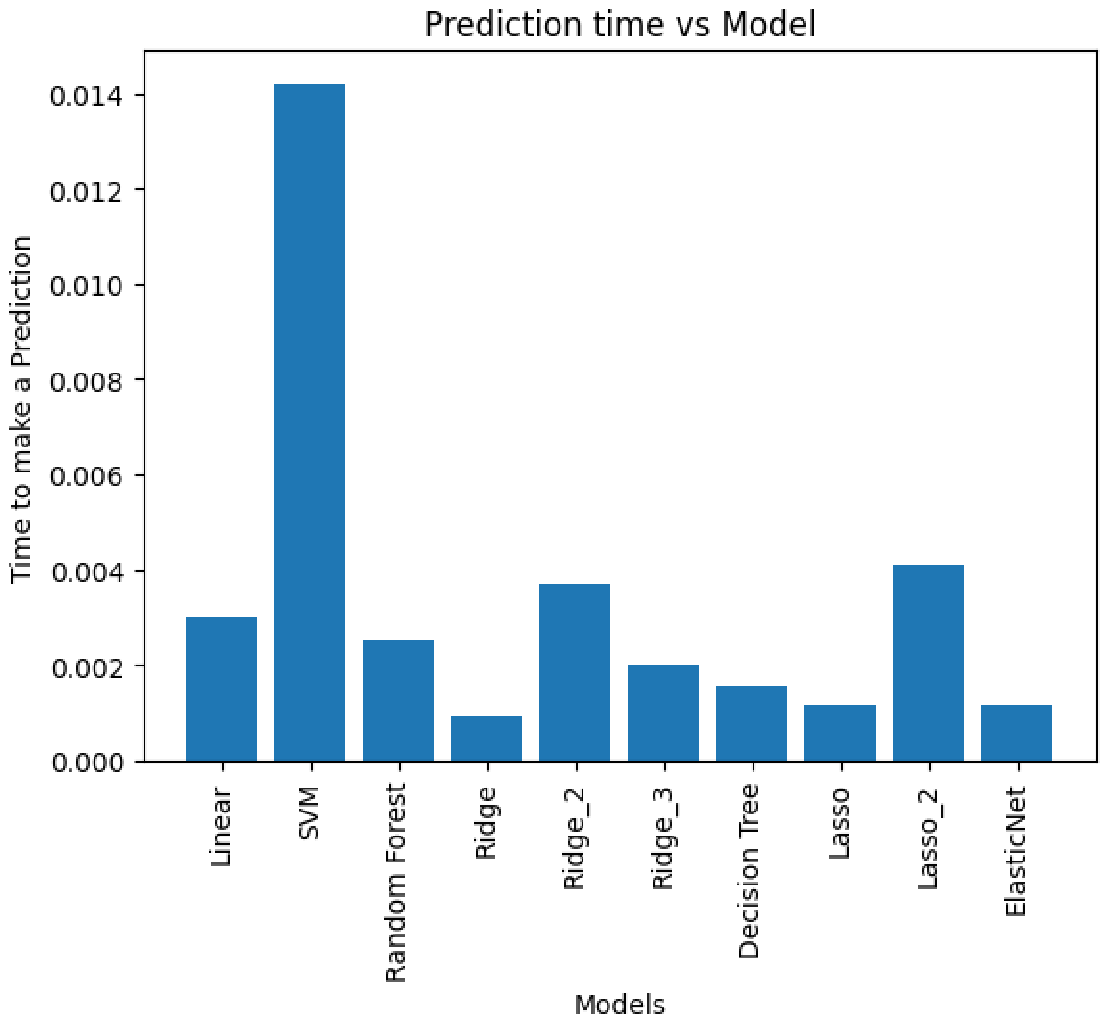
Submission to kaggle showed that this model performed the WORST. Quite disappointing but oh well, and that was after manuelly decreasing the number of layers it still performed the worst in kaggle. Oh well, im going to drop this model then.

Also i know i didnt display any of the model performance metrics, but thats because i didnt want to adapt my code to tensorflow code, and because it performed so bad in kaggle that its not even worth it.

16 QUICK INTERLUDE: PREDICTION AND TRAINING TIME

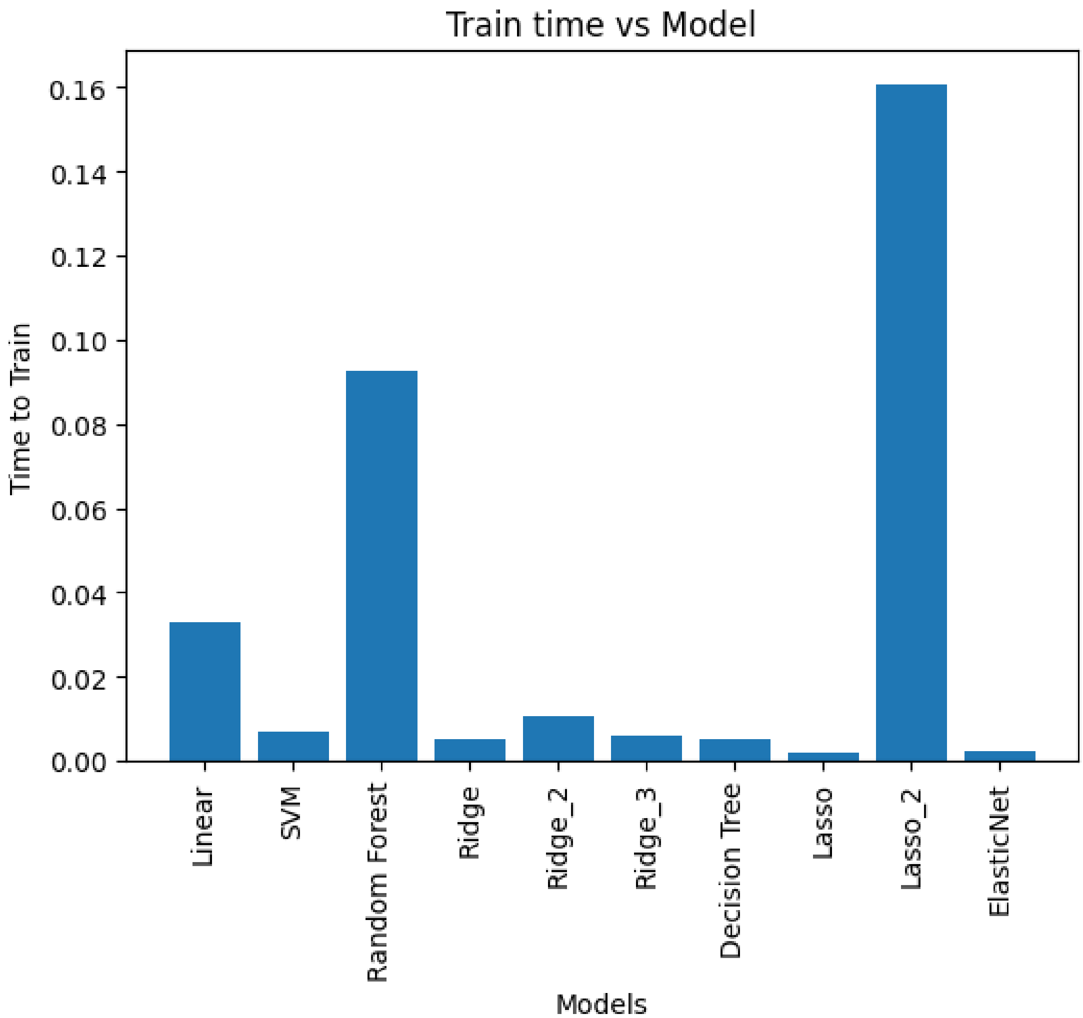
[131]:
```
model_names = ['Linear','SVM','Random␣
↪Forest','Ridge','Ridge_2','Ridge_3','Decision␣
↪Tree','Lasso','Lasso_2','ElasticNet']
plt.bar(model_names,time_taken_to_predict)
plt.xticks(rotation='vertical')
plt.xlabel('Models')
plt.ylabel('Time to make a Prediction')
plt.title('Prediction time vs Model')
```

[131]:
```
Text(0.5, 1.0, 'Prediction time vs Model')
```



[132]:
```
model_names = ['Linear','SVM','Random␣
 ↪Forest','Ridge','Ridge_2','Ridge_3','Decision␣
 ↪Tree','Lasso','Lasso_2','ElasticNet']
plt.bar(model_names,time_taken_to_train)
plt.xticks(rotation='vertical')
plt.xlabel('Models')
plt.ylabel('Time to Train')
plt.title('Train time vs Model')
```

[132]:
```
Text(0.5, 1.0, 'Train time vs Model')
```



Notes:

Note that what I timed per model was explictly the training time, NOT the time taken to perform grid search.

Additional It seems that in general the more complex models take both longer to train and longer to make predicts for.

17 MODEL SUBMISSIONS

[133]:
```
#========
# This function takes in the mprediction and what the file should be called and
 ↪outputs it to a txt for manuelly submission to kaggle
#========
def make_submission_txt(pred,filename):
    index_arr = np.arange(401, 401+pred.size).reshape(-1,1)
```

```
    index_arr = index_arr.astype(int).flatten() #makes index array
    final_array = np.stack((index_arr,pred),axis=1) #stacks them
    df = pd.DataFrame(final_array,columns=['id','target']) #adds name
    df['id'] = df['id'].astype(int) #ouputs
    df.to_csv(filename+'.csv', index=False)
    print("The file hase been created")
```

SVM

[134]:
```
best_svm.fit(X_train,y_train)
best_svm_pred = best_svm.predict(df_test)
make_submission_txt(best_svm_pred,"svm1")
```

```
The file hase been created
```

best__random__forest__reg

[135]:
```
best_random_forest_reg.fit(X_train,y_train)
best_random_forest_reg_pred = best_random_forest_reg.predict(df_test)
make_submission_txt(best_random_forest_reg_pred,"rfr1")
```

```
The file hase been created
```

best__ridge

[136]:
```
best_ridge.fit(X_train,y_train)
best_ridge_pred = best_ridge.predict(df_test)
make_submission_txt(best_ridge_pred,"r1")
#NOTES: so far this one has performed the best when submitted
```

```
The file hase been created
```

best__tree

[137]:
```
best_tree.fit(X_train,y_train)
best_tree_pred = best_tree.predict(df_test)
make_submission_txt(best_tree_pred,"t1")
```

```
The file hase been created
```

best__lasso

[138]:
```
best_lasso.fit(X_train,y_train)
best_lasso_pred = best_lasso.predict(df_test)
make_submission_txt(best_lasso_pred,"lasso1")
```

```
The file hase been created
```

best__elastic

[139]:
```
best_elastic.fit(X_train,y_train)
best_elastic_pred = best_elastic.predict(df_test)
make_submission_txt(best_elastic_pred,"elastic1")
```

```
The file hase been created
```

best__ridge__2

[140]:
```
best_ridge_2.fit(X_train,y_train)
best_ridge_2_pred = best_ridge_2.predict(df_test)
make_submission_txt(best_ridge_2_pred,"r11")
```

```
The file hase been created
```

best__ridge__3

[141]:
```
best_ridge_3.fit(X_train,y_train)
best_ridge_3_pred = best_ridge_3.predict(df_test)
make_submission_txt(best_ridge_3_pred,"r12")
```

```
The file hase been created
```

best__lasso__2

[142]:
```
best_lasso_2.fit(X_train,y_train)
best_lasso_2_pred = best_lasso_2.predict(df_test)
make_submission_txt(best_lasso_2_pred,"l12")
```

```
The file hase been created
```

NN

[143]:
```
make_submission_txt(nn_pred.flatten(),"nn5")
```

```
The file hase been created
```

18 THE BEST MODELS:

A) Lasso with polynomial features: 18.58

B) Ridge with polynomial features: 32.55

C) Ridge with polynomial features and PCA: 47.92

D) Ridge (By itself): 55.22

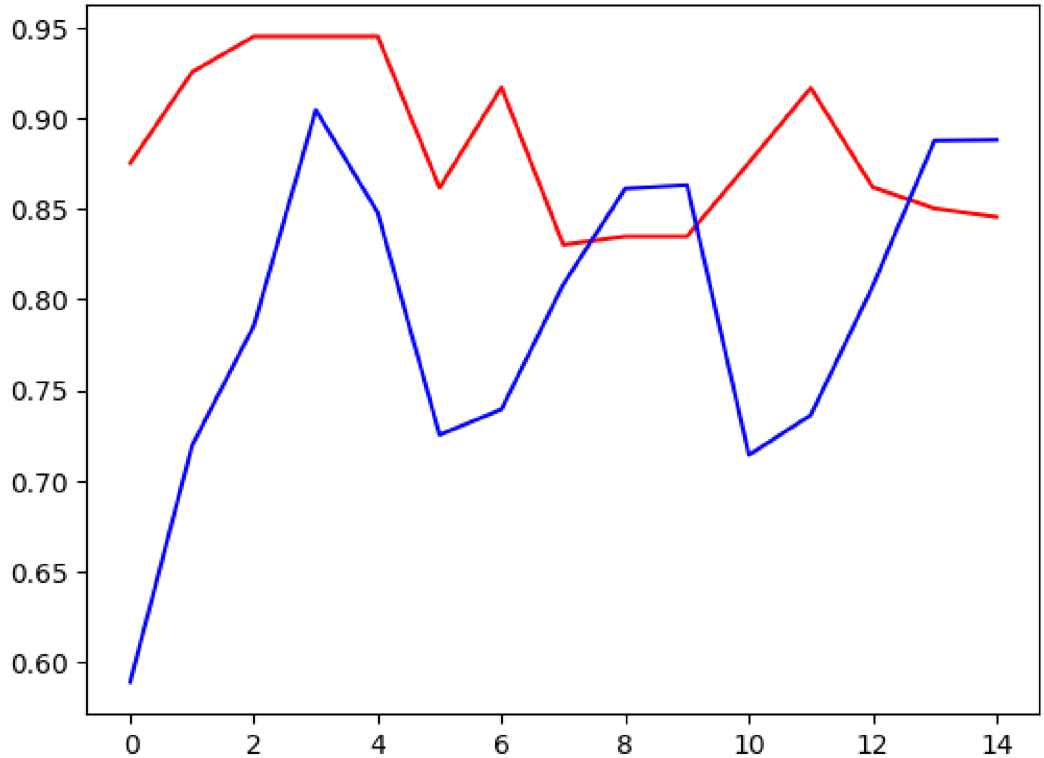
E) Lasso (By itself): 56.93

Ranked by performance in the Kaggle submissions with the MSE score displayed (gotten from kaggles system)

Models A and B and C were able to generalize much better then the other 3, with model A, being able toe generalize the best, miles ahead of model B

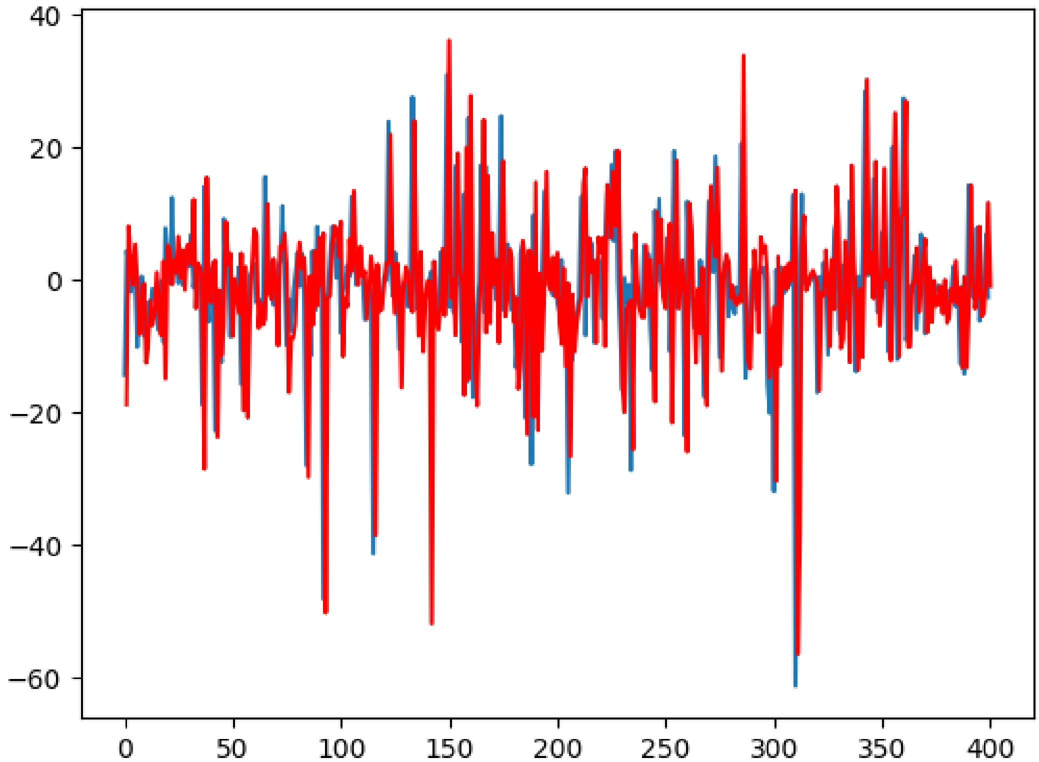
And finally im going to display again how the best model performed on the metrics i tested on.

[144]:
```
best_lasso_2.fit(X_train,y_train)
make_train_vs_test_curves(best_lasso_2,X_train,y_train)
```



[145]:
```
training_data_fit_display(best_lasso_2,X_train,y_train)
```

```
Here is the MSE score:
4.044953021119381
```



Finally this model got a kaggle score of: 18.58684 Which at least at the time of me making/submitting this assgiment had me in 1st place on the leader board for the class, which i think i kind of cool!