hw1_report

September 18, 2024

1 HW1 : Regression, Cross-Validation, and Regularization

[1]:
```
import os
import numpy as np
import pandas as pd

import sklearn.preprocessing
import sklearn.pipeline
import sklearn.linear_model
```

[2]:
```
from matplotlib import pyplot as plt

import seaborn as sns
#This sets the default style for all figures.
sns.set('notebook', font_scale=1.25, style='whitegrid')
```

1.0.1 Configuration

You may need to adjust your data directory to point towards the auto data.

[12]:
```
SEED = 12345

DATA_DIR = 'data_auto/'
```

2 Helper Functions

These notebook sections contain helper functions which you **do not need to alter**.

2.0.1 Methods for loading dataset

[4]:
```
def load_2d_arr_from_csv(fname, include_header=False):
    x = np.loadtxt(os.path.join(DATA_DIR, fname), delimiter=',', skiprows=1)
    assert x.ndim == 2
    if include_header:
        header_cols = np.loadtxt(os.path.join(DATA_DIR, fname), delimiter=',',␣
↪dtype=str)[0].tolist()
        return x, header_cols
    else:
        return x

def load_1d_arr_from_csv(fname):
    x = np.loadtxt(os.path.join(DATA_DIR, fname), delimiter=',', skiprows=1)
    if x.ndim == 1:
        return x
    else:
        raise ValueError("Not 1d")
```

2.0.2 Plotting methods

[5]:
```
def plot_train_and_valid_error_vs_hyper(
        hyper_list, err_tr_list=None, err_va_list=None,
        ymax=40,
        leg_loc='upper right',
        xlabel='polynomial degree',
        ylabel='RMSE'):
    if err_va_list is not None:
        plt.plot(hyper_list, err_va_list, 'rs-', label='valid');
    if err_tr_list is not None:
        plt.plot(hyper_list, err_tr_list, 'bd:', label='train');
    plt.ylim([0, ymax]);
    plt.legend(loc=leg_loc);
    plt.xlabel(xlabel);
    plt.ylabel(ylabel);
```

2.0.3 Method to sanitize predictions

For many regression problems certain values might be impossible or improbable (such as in this case where we can't have a negative miles-per-gallon). However, most machine learning models cannot learn these types of strict rules. Thus, a common practice in ML is to **sanatize** predictions made by a model, confining them to be within reasonable bounds.

Here we are predicting MPG, which should * (1) always be positive, and * (2) will probably never exceed 120% of the largest value we see in train data

All model predictions should be sanitized before being used.

[6]:
```
def sanitize(yhat_N):
    yhat_N = np.maximum(yhat_N, 0)
    yhat_N = np.minimum(yhat_N, Y_MAX)
    return yhat_N
```

2.1 Methods for building pipelines

These are sklearn pipelines, which define a series of steps that can then be treated as if they were a single classifier.

```
[7]: def make_poly_linear_regr_pipeline(degree=1):
         pipeline = sklearn.pipeline.Pipeline(
             steps=[
              ('rescaler', sklearn.preprocessing.MinMaxScaler()),
              ('poly_transformer', sklearn.preprocessing.
       ↪PolynomialFeatures(degree=degree, include_bias=False)),
              ('linear_regr', sklearn.linear_model.LinearRegression()),
             ])

         # Return the constructed pipeline
         # We can treat it as if it has a 'regression' API
         # e.g. a fit and a predict method
         return pipeline
```

```
[36]: def make_unscaled_poly_linear_regr_pipeline(degree=1):
          pipeline = sklearn.pipeline.Pipeline(
              steps=[
               # ('rescaler', sklearn.preprocessing.MinMaxScaler()),
               ('poly_transformer', sklearn.preprocessing.
        ↪PolynomialFeatures(degree=degree, include_bias=False)),
               ('linear_regr', sklearn.linear_model.LinearRegression()),
              ])

          # Return the constructed pipeline
          # We can treat it as if it has a 'regression' API
          # e.g. a fit and a predict method
          return pipeline
```

```
[8]: def make_poly_ridge_regr_pipeline(degree=1, alpha=1.0):
         pipeline = sklearn.pipeline.Pipeline(
             steps=[
              ('rescaler', sklearn.preprocessing.MinMaxScaler()),
              ('poly_transformer', sklearn.preprocessing.
       ↪PolynomialFeatures(degree=degree, include_bias=False)),
              ('linear_regr', sklearn.linear_model.Ridge(alpha=alpha)),
             ])

         # Return the constructed pipeline
         # We can treat it as if it has a 'regression' API
         # e.g. a fit and a predict method
         return pipeline
```

2.2 Method to inspect learned weights

Use this function when asked to display learned parameters.

[38]:
```python
def pretty_print_learned_weights(pipeline, xcolnames_F):
    ''' Print the learned parameters of given pipeline
    '''
    my_lin_regr = pipeline.named_steps['linear_regr']

    feat_names = pipeline.named_steps['poly_transformer'].
  ↪get_feature_names_out()
    coef_values = my_lin_regr.coef_

    for feat, coef in zip(feat_names, coef_values):
        print("% 7.2f : %s" % (coef, feat))

    print("where ")
    for ff, colname in enumerate(xcolnames_F):
        print("x%d = %s" % (ff, colname))
```

3 Analysis

The rest of this notebook is an analysis and report of the automobile dataset. Some of the steps of the analysis have already been completed for you. You need to complete the indicated sections in problems 1-3.

4 Load the dataset

First, we need to load the predefined 'x' and 'y' arrays for train/valid/test sets using the predefined helper function.

[13]:
```python
x_tr_MF, xcolnames_F = load_2d_arr_from_csv('x_train.csv', include_header=True)
x_va_NF = load_2d_arr_from_csv('x_valid.csv')
x_te_PF = load_2d_arr_from_csv('x_test.csv')
```

[14]:
```python
y_tr_M = load_1d_arr_from_csv('y_train.csv')
y_va_N = load_1d_arr_from_csv('y_valid.csv')
y_te_P = load_1d_arr_from_csv('y_test.csv')
```

We can then take at various parts of our training data.

[15]:
```python
# Feature names
print("Feature Names:", xcolnames_F)

# First 5 instances in our training data
print("Training data features:\n", x_tr_MF[:5])

# MPG for the first 5 instances
print("Training data MPGs:\n", y_tr_M[:5,np.newaxis])
```

```
Feature Names: ['horsepower', 'weight', 'cylinders', 'displacement']
Training data features:
 [[ 115. 2595.    6.  173.]
 [ 180. 4380.    8.  350.]
 [ 150. 4457.    8.  318.]
 [ 105. 3897.    6.  250.]
 [ 193. 4732.    8.  304.]]
Training data MPGs:
 [[28.8]
 [16.5]
 [14. ]
 [16. ]
 [ 9. ]]
```

Also, we need to set a `Y_MAX` variable so that the sanatize function works.

[17]:
```
# Highest MPG in the training data
y_tr_M.max()
```

[17]:
```
46.6
```

[18]:
```
Y_MAX = y_tr_M.max()*1.2
```

4.1 Load completed code

Now we load in the code you wrote in part of of the homework.

[146]:
```
from performance_metrics import calc_root_mean_squared_error
from cross_validation import train_models_and_calc_scores_for_n_fold_cv
```

5 Problem 1: Polynomial Degree Selection on Fixed Validation Set

Implementation Step 1A:

Fit a linear regression model to a polynomial feature transformation of the provided training set at each of these possible degrees: [1, 2, 3, 4, 5, 6, 7]. For each hyperparameter setting, record the training set error and the validation set error in terms of RMSE.

[28]:
```
degree_list = [1, 2, 3, 4, 5, 6, 7]
fv_err_tr_list = []
fv_err_va_list = []

fv_pipeline_list = []
for degree in degree_list:

    # TODO create a pipeline using features with current degree value
    # TODO train this pipeline on provided training data (x_tr_MF, y_train_M)
    poly_linear_pipeline_problem_1 = make_poly_linear_regr_pipeline(degree)
    poly_linear_pipeline_problem_1.fit(x_tr_MF, y_tr_M)
    # Compute training error
    yhat_tr_M = sanitize(poly_linear_pipeline_problem_1.predict(x_tr_MF))  #␣
↪TODO fixme, be sure to sanitize predictions
    yhat_va_N = sanitize(poly_linear_pipeline_problem_1.predict(x_va_NF))  #␣
↪TODO fixme, be sure to sanitize predictions

    err_tr = calc_root_mean_squared_error(y_tr_M, yhat_tr_M) # TODO fixme
    err_va = calc_root_mean_squared_error(y_va_N, yhat_va_N) # TODO fixme

    fv_err_tr_list.append(err_tr)
    fv_err_va_list.append(err_va)

    # TODO store current pipeline for future use
    fv_pipeline_list.append(poly_linear_pipeline_problem_1)
```

[29]:
```
# Overview of polynomial results
for idx, deg in enumerate(degree_list):
    print("Degree {} produced the following RMSEs: Training Data: {},␣
↪Validation Data: {}".format(deg, fv_err_tr_list[idx], fv_err_va_list[idx]))
```

```
Degree 1 produced the following RMSEs: Training Data: 4.237313, Validation Data:
4.360352
Degree 2 produced the following RMSEs: Training Data: 3.734677, Validation Data:
3.974075
Degree 3 produced the following RMSEs: Training Data: 3.38421, Validation Data:
4.157264
Degree 4 produced the following RMSEs: Training Data: 2.9005, Validation Data:
5.891522
Degree 5 produced the following RMSEs: Training Data: 2.290656, Validation Data:
14.872595
Degree 6 produced the following RMSEs: Training Data: 0.546188, Validation Data:
26.661354
Degree 7 produced the following RMSEs: Training Data: 0.117892, Validation Data:
27.589121
```

5.0.1 Implementation 1B: Score on the test set using the chosen model

Select the model hyperparameters that *minimize* your fixed validation set error. Using your already-trained LinearRegression model with these best hyperparameters, compute error on the *test* set. Save this test set error value for later.

[32]:
```
print("Selected Parameters:")
print("degree 2")
print("Fixed validation set estimate of heldout error:")
print("3.97")
print("Error on the test-set:")
print("{}".format(
    calc_root_mean_squared_error(y_te_P, sanitize(fv_pipeline_list[1].
 ↪predict(x_te_PF)))
))
```

```
Selected Parameters:
degree 2
Fixed validation set estimate of heldout error:
3.97
Error on the test-set:
3.991503
```

[34]:
```
# TODO store score on test set for later
prob1_polydeg2_testerr = calc_root_mean_squared_error(y_te_P,␣
 ↪sanitize(fv_pipeline_list[1].predict(x_te_PF)))
```

5.0.2 Figure 1: Error vs degree

Once `fv_err_tr_list` and `fv_err_va_list` contain values from part 1A, you can re-run the plotting cell to generate figure 1 with the values you found.

[35]:
```
plot_train_and_valid_error_vs_hyper(
    degree_list, fv_err_tr_list, fv_err_va_list, leg_loc='upper left');
plt.title('RMSE vs. Degree');
```

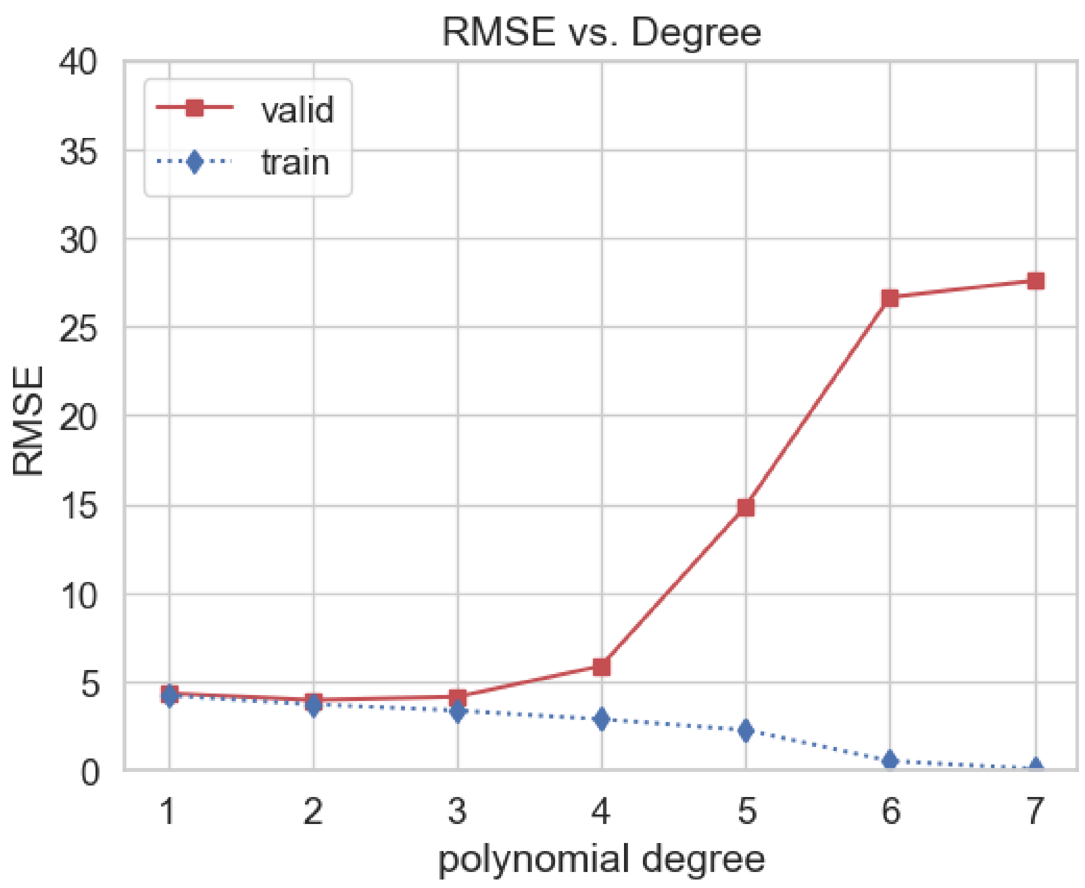


Figure 1: RMSE vs. Degree *Provide a 2 sentence caption answering the questions: How do validation and training error change as degree increases? What degree do you recommend based on this plot?*

Training error: It decreases monotonically as the "degree" increases. The decrease appears linear from degrees 1 through 5. At degrees 6 and 7, there is a rapid drop approaching 0, which suggests the data is being overfit.

Validation error: It decreases from degree 1 to 2, with the lowest value being attained at degree 2. Then it increases rapidly, plateauing at about 26 at degree 6.

Short Answer 1A The starter code pipelines include a *preprocessing* step that rescales each feature column to be in the unit interval from 0 to 1. Why is this necessary for this particular dataset? What happens (in terms of both training error and test error) if this step is omitted? *Hint: Try removing this step and see.*

[ ]:
```
prob1_unscaled = make_unscaled_poly_linear_regr_pipeline(degree)
prob1_unscaled.fit(x_tr_MF, y_tr_M)
```

[49]:
```
print("Parameters for model that uses feature scaling.")
pretty_print_learned_weights(fv_pipeline_list[1], xcolnames_F),
print(
    "Error metrics\n",
    "Train {} Validation {} Test {}".format(fv_err_tr_list[1],␣
  ↪fv_err_va_list[1], prob1_polydeg2_testerr)
)
print("Parameters for model that doesn't use feature scaling.")
pretty_print_learned_weights(prob1_unscaled, xcolnames_F)
print(
    "Error metrics",
    "Train {} Validation {} Test {}".format(
        calc_root_mean_squared_error(y_tr_M, sanitize(prob1_unscaled.
  ↪predict(x_tr_MF))),
        calc_root_mean_squared_error(y_va_N, sanitize(prob1_unscaled.
  ↪predict(x_va_NF))),
        calc_root_mean_squared_error(y_te_P, sanitize(prob1_unscaled.
  ↪predict(x_te_PF))),
    )
)
```

```
Parameters for model that uses feature scaling.
 -40.72 : x0
  -7.20 : x1
 -11.00 : x2
 -11.10 : x3
  -4.45 : x0^2
   8.80 : x0 x1
  26.38 : x0 x2
   8.98 : x0 x3
 -31.77 : x1^2
  27.17 : x1 x2
  13.43 : x1 x3
  21.76 : x2^2
 -80.86 : x2 x3
  52.65 : x3^2
where
x0 = horsepower
x1 = weight
x2 = cylinders
x3 = displacement
Error metrics
 Train 3.734677 Validation 3.974075 Test 3.991503
Parameters for model that doesn't use feature scaling.
  -0.00 : x0
  -0.00 : x1
   0.00 : x2
 0.00 : x3
-0.00 : x0^2
-0.00 : x0 x1
 0.00 : x0 x2
-0.00 : x0 x3
 0.00 : x1^2
 0.00 : x1 x2
 0.00 : x1 x3
 0.00 : x2^2
-0.00 : x2 x3
-0.00 : x3^2
-0.00 : x0^3
 0.00 : x0^2 x1
 0.00 : x0^2 x2
 0.00 : x0^2 x3
-0.00 : x0 x1^2
 0.00 : x0 x1 x2
-0.00 : x0 x1 x3
-0.00 : x0 x2^2
-0.00 : x0 x2 x3
 0.00 : x0 x3^2
 0.00 : x1^3
 0.00 : x1^2 x2
-0.00 : x1^2 x3
-0.00 : x1 x2^2
-0.00 : x1 x2 x3
-0.00 : x1 x3^2
-0.00 : x2^3
 0.00 : x2^2 x3
-0.00 : x2 x3^2
-0.00 : x3^3
-0.00 : x0^4
-0.00 : x0^3 x1
-0.00 : x0^3 x2
-0.00 : x0^3 x3
-0.00 : x0^2 x1^2
-0.00 : x0^2 x1 x2
-0.00 : x0^2 x1 x3
-0.00 : x0^2 x2^2
-0.00 : x0^2 x2 x3
-0.00 : x0^2 x3^2
-0.00 : x0 x1^3
-0.00 : x0 x1^2 x2
-0.00 : x0 x1^2 x3
-0.00 : x0 x1 x2^2
-0.00 : x0 x1 x2 x3
-0.00 : x0 x1 x3^2
-0.00 : x0 x2^3
```

```
-0.00 : x0 x2^2 x3
-0.00 : x0 x2 x3^2
-0.00 : x0 x3^3
 0.00 : x1^4
 0.00 : x1^3 x2
-0.00 : x1^3 x3
 0.00 : x1^2 x2^2
-0.00 : x1^2 x2 x3
-0.00 : x1^2 x3^2
 0.00 : x1 x2^3
-0.00 : x1 x2^2 x3
-0.00 : x1 x2 x3^2
-0.00 : x1 x3^3
-0.00 : x2^4
-0.00 : x2^3 x3
-0.00 : x2^2 x3^2
-0.00 : x2 x3^3
 0.00 : x3^4
 0.00 : x0^5
-0.00 : x0^4 x1
-0.00 : x0^4 x2
 0.00 : x0^4 x3
-0.00 : x0^3 x1^2
-0.00 : x0^3 x1 x2
 0.00 : x0^3 x1 x3
-0.00 : x0^3 x2^2
-0.00 : x0^3 x2 x3
 0.00 : x0^3 x3^2
-0.00 : x0^2 x1^3
-0.00 : x0^2 x1^2 x2
 0.00 : x0^2 x1^2 x3
-0.00 : x0^2 x1 x2^2
-0.00 : x0^2 x1 x2 x3
-0.00 : x0^2 x1 x3^2
-0.00 : x0^2 x2^3
-0.00 : x0^2 x2^2 x3
-0.00 : x0^2 x2 x3^2
-0.00 : x0^2 x3^3
-0.00 : x0 x1^4
-0.00 : x0 x1^3 x2
 0.00 : x0 x1^3 x3
-0.00 : x0 x1^2 x2^2
-0.00 : x0 x1^2 x2 x3
-0.00 : x0 x1^2 x3^2
-0.00 : x0 x1 x2^3
-0.00 : x0 x1 x2^2 x3
-0.00 : x0 x1 x2 x3^2
-0.00 : x0 x1 x3^3
```

```
-0.00 : x0 x2^4
-0.00 : x0 x2^3 x3
 0.00 : x0 x2^2 x3^2
-0.00 : x0 x2 x3^3
-0.00 : x0 x3^4
 0.00 : x1^5
-0.00 : x1^4 x2
-0.00 : x1^4 x3
-0.00 : x1^3 x2^2
-0.00 : x1^3 x2 x3
 0.00 : x1^3 x3^2
-0.00 : x1^2 x2^3
-0.00 : x1^2 x2^2 x3
-0.00 : x1^2 x2 x3^2
 0.00 : x1^2 x3^3
-0.00 : x1 x2^4
-0.00 : x1 x2^3 x3
 0.00 : x1 x2^2 x3^2
 0.00 : x1 x2 x3^3
 0.00 : x1 x3^4
-0.00 : x2^5
 0.00 : x2^4 x3
 0.00 : x2^3 x3^2
 0.00 : x2^2 x3^3
 0.00 : x2 x3^4
 0.00 : x3^5
-0.00 : x0^6
 0.00 : x0^5 x1
-0.00 : x0^5 x2
-0.00 : x0^5 x3
 0.00 : x0^4 x1^2
-0.00 : x0^4 x1 x2
-0.00 : x0^4 x1 x3
-0.00 : x0^4 x2^2
 0.00 : x0^4 x2 x3
-0.00 : x0^4 x3^2
-0.00 : x0^3 x1^3
-0.00 : x0^3 x1^2 x2
 0.00 : x0^3 x1^2 x3
-0.00 : x0^3 x1 x2^2
 0.00 : x0^3 x1 x2 x3
-0.00 : x0^3 x1 x3^2
 0.00 : x0^3 x2^3
 0.00 : x0^3 x2^2 x3
 0.00 : x0^3 x2 x3^2
 0.00 : x0^3 x3^3
 0.00 : x0^2 x1^4
 0.00 : x0^2 x1^3 x2
```

```
-0.00 : x0^2 x1^3 x3
-0.00 : x0^2 x1^2 x2^2
 0.00 : x0^2 x1^2 x2 x3
 0.00 : x0^2 x1^2 x3^2
-0.00 : x0^2 x1 x2^3
 0.00 : x0^2 x1 x2^2 x3
 0.00 : x0^2 x1 x2 x3^2
 0.00 : x0^2 x1 x3^3
-0.00 : x0^2 x2^4
 0.00 : x0^2 x2^3 x3
 0.00 : x0^2 x2^2 x3^2
 0.00 : x0^2 x2 x3^3
-0.00 : x0^2 x3^4
-0.00 : x0 x1^5
 0.00 : x0 x1^4 x2
 0.00 : x0 x1^4 x3
-0.00 : x0 x1^3 x2^2
-0.00 : x0 x1^3 x2 x3
-0.00 : x0 x1^3 x3^2
-0.00 : x0 x1^2 x2^3
-0.00 : x0 x1^2 x2^2 x3
-0.00 : x0 x1^2 x2 x3^2
 0.00 : x0 x1^2 x3^3
-0.00 : x0 x1 x2^4
 0.00 : x0 x1 x2^3 x3
 0.00 : x0 x1 x2^2 x3^2
-0.00 : x0 x1 x2 x3^3
-0.00 : x0 x1 x3^4
-0.00 : x0 x2^5
 0.00 : x0 x2^4 x3
 0.00 : x0 x2^3 x3^2
 0.00 : x0 x2^2 x3^3
-0.00 : x0 x2 x3^4
-0.00 : x0 x3^5
-0.00 : x1^6
-0.00 : x1^5 x2
 0.00 : x1^5 x3
 0.00 : x1^4 x2^2
 0.00 : x1^4 x2 x3
-0.00 : x1^4 x3^2
-0.00 : x1^3 x2^3
-0.00 : x1^3 x2^2 x3
-0.00 : x1^3 x2 x3^2
 0.00 : x1^3 x3^3
-0.00 : x1^2 x2^4
 0.00 : x1^2 x2^3 x3
 0.00 : x1^2 x2^2 x3^2
-0.00 : x1^2 x2 x3^3
```

```
-0.00 : x1^2 x3^4
-0.00 : x1 x2^5
 0.00 : x1 x2^4 x3
 0.00 : x1 x2^3 x3^2
 0.00 : x1 x2^2 x3^3
-0.00 : x1 x2 x3^4
 0.00 : x1 x3^5
-0.00 : x2^6
 0.00 : x2^5 x3
 0.00 : x2^4 x3^2
 0.00 : x2^3 x3^3
 0.00 : x2^2 x3^4
 0.00 : x2 x3^5
 0.00 : x3^6
 0.00 : x0^7
-0.00 : x0^6 x1
-0.00 : x0^6 x2
 0.00 : x0^6 x3
 0.00 : x0^5 x1^2
-0.00 : x0^5 x1 x2
-0.00 : x0^5 x1 x3
-0.00 : x0^5 x2^2
-0.00 : x0^5 x2 x3
 0.00 : x0^5 x3^2
-0.00 : x0^4 x1^3
 0.00 : x0^4 x1^2 x2
 0.00 : x0^4 x1^2 x3
-0.00 : x0^4 x1 x2^2
-0.00 : x0^4 x1 x2 x3
-0.00 : x0^4 x1 x3^2
 0.00 : x0^4 x2^3
 0.00 : x0^4 x2^2 x3
-0.00 : x0^4 x2 x3^2
 0.00 : x0^4 x3^3
 0.00 : x0^3 x1^4
 0.00 : x0^3 x1^3 x2
-0.00 : x0^3 x1^3 x3
-0.00 : x0^3 x1^2 x2^2
 0.00 : x0^3 x1^2 x2 x3
 0.00 : x0^3 x1^2 x3^2
 0.00 : x0^3 x1 x2^3
 0.00 : x0^3 x1 x2^2 x3
-0.00 : x0^3 x1 x2 x3^2
 0.00 : x0^3 x1 x3^3
 0.00 : x0^3 x2^4
 0.00 : x0^3 x2^3 x3
 0.00 : x0^3 x2^2 x3^2
-0.00 : x0^3 x2 x3^3
```

```
-0.00 : x0^3 x3^4
-0.00 : x0^2 x1^5
-0.00 : x0^2 x1^4 x2
 0.00 : x0^2 x1^4 x3
-0.00 : x0^2 x1^3 x2^2
 0.00 : x0^2 x1^3 x2 x3
-0.00 : x0^2 x1^3 x3^2
 0.00 : x0^2 x1^2 x2^3
 0.00 : x0^2 x1^2 x2^2 x3
 0.00 : x0^2 x1^2 x2 x3^2
 0.00 : x0^2 x1^2 x3^3
 0.00 : x0^2 x1 x2^4
 0.00 : x0^2 x1 x2^3 x3
 0.00 : x0^2 x1 x2^2 x3^2
-0.00 : x0^2 x1 x2 x3^3
-0.00 : x0^2 x1 x3^4
 0.00 : x0^2 x2^5
 0.00 : x0^2 x2^4 x3
 0.00 : x0^2 x2^3 x3^2
 0.00 : x0^2 x2^2 x3^3
-0.00 : x0^2 x2 x3^4
 0.00 : x0^2 x3^5
 0.00 : x0 x1^6
 0.00 : x0 x1^5 x2
-0.00 : x0 x1^5 x3
 0.00 : x0 x1^4 x2^2
-0.00 : x0 x1^4 x2 x3
 0.00 : x0 x1^4 x3^2
-0.00 : x0 x1^3 x2^3
-0.00 : x0 x1^3 x2^2 x3
 0.00 : x0 x1^3 x2 x3^2
-0.00 : x0 x1^3 x3^3
-0.00 : x0 x1^2 x2^4
 0.00 : x0 x1^2 x2^3 x3
-0.00 : x0 x1^2 x2^2 x3^2
-0.00 : x0 x1^2 x2 x3^3
 0.00 : x0 x1^2 x3^4
-0.00 : x0 x1 x2^5
 0.00 : x0 x1 x2^4 x3
 0.00 : x0 x1 x2^3 x3^2
-0.00 : x0 x1 x2^2 x3^3
 0.00 : x0 x1 x2 x3^4
-0.00 : x0 x1 x3^5
-0.00 : x0 x2^6
 0.00 : x0 x2^5 x3
 0.00 : x0 x2^4 x3^2
 0.00 : x0 x2^3 x3^3
 0.00 : x0 x2^2 x3^4
```

```
   0.00 : x0 x2 x3^5
   0.00 : x0 x3^6
   0.00 : x1^7
   0.00 : x1^6 x2
  -0.00 : x1^6 x3
   0.00 : x1^5 x2^2
  -0.00 : x1^5 x2 x3
   0.00 : x1^5 x3^2
  -0.00 : x1^4 x2^3
  -0.00 : x1^4 x2^2 x3
   0.00 : x1^4 x2 x3^2
  -0.00 : x1^4 x3^3
  -0.00 : x1^3 x2^4
   0.00 : x1^3 x2^3 x3
   0.00 : x1^3 x2^2 x3^2
  -0.00 : x1^3 x2 x3^3
   0.00 : x1^3 x3^4
  -0.00 : x1^2 x2^5
  -0.00 : x1^2 x2^4 x3
   0.00 : x1^2 x2^3 x3^2
   0.00 : x1^2 x2^2 x3^3
   0.00 : x1^2 x2 x3^4
  -0.00 : x1^2 x3^5
  -0.00 : x1 x2^6
  -0.00 : x1 x2^5 x3
   0.00 : x1 x2^4 x3^2
   0.00 : x1 x2^3 x3^3
   0.00 : x1 x2^2 x3^4
  -0.00 : x1 x2 x3^5
   0.00 : x1 x3^6
  -0.00 : x2^7
   0.00 : x2^6 x3
   0.00 : x2^5 x3^2
   0.00 : x2^4 x3^3
   0.00 : x2^3 x3^4
   0.00 : x2^2 x3^5
   0.00 : x2 x3^6
  -0.00 : x3^7
where
x0 = horsepower
x1 = weight
x2 = cylinders
x3 = displacement
Error metrics Train 2.962887 Validation 11.309984 Test 11.725465
```

Anwer: As you can see above, all the parameters of the regressor that doesn't use feature scaling get set to very small numbers. The error on the train set isn't too different but the errors on the validation and test set are almost 4 times higher. Feature scaling is necessary so that features whose values are very high (weight) don't get weighed differently than features whose values are low (cylinders) as the difference in the magnitude of these values affects the way the model sets the weights.

Short Answer 1B Consider the model with degree 1. Following the starter code, print out the values of **all** the learned weight parameters (aka coefficients). From these values, which feature has the highest positive impact on MPG? Which has the highest negative impact? Do these make sense?

[51]:
```
# TODO call pretty_print_learned_weights on your pipeline with degree=1 from␣
↪above
# Hint: The names of the original F=4 features are already in your workspace
pretty_print_learned_weights(fv_pipeline_list[0], xcolnames_F)
```

```
 -10.43 : x0
 -18.23 : x1
  -1.15 : x2
   0.58 : x3
where
x0 = horsepower
x1 = weight
x2 = cylinders
x3 = displacement
```

The feature with the largest positive impact is **x3** or "displacement." The feature with the largest negative impact is **x1** or "weight."

It makes sense that a car will utilize less gasoline moving a vehicle over a given distance if this vehicle is comparatively lighter than other vehicles. The impact of displacement is small (weight of 0.58), which suggests that there might be some basis on which you could claim that the larger the displacement of gasoline in an engine, the more efficient it is. A larger displacement means a car can burn more air and fuel, producing more power, and therefore, displacement is often an indicator of an engine's potential performance. Manufacturers often design their cars such that the engine coupling with the rest of the car meets some kind of efficiency standard.

6 Problem 2: Alpha Selection on Fixed Val Set

Implementation Step 2A Fix the degree at 4. Consider the below possible `alpha` values for L2-penalized linear regression, aka `Ridge`.

Fit a L2-penalized linear regression pipeline for each alpha value above, then record that model's training set error and the validation set error.

[52]:
```
my_degree = 4
alpha_list = np.asarray([1.e-10, 1.e-08, 1.e-06, 1.e-04, 1.e-02, 1.e+00, 1.
↪e+02, 1.e+04, 1.e+06])
print(alpha_list)
```

```
[1.e-10 1.e-08 1.e-06 1.e-04 1.e-02 1.e+00 1.e+02 1.e+04 1.e+06]
```

[54]:
```
fv2_err_tr_list = []
fv2_err_va_list = []
fv2_pipeline_list = []

for alpha in alpha_list:

    # TODO create a pipeline using features with current degree value
    # TODO train this pipeline on provided training data
    alpha_pipeline_problem_2 = make_poly_ridge_regr_pipeline(degree = 4, alpha␣
 ↪= alpha)
    alpha_pipeline_problem_2.fit(x_tr_MF, y_tr_M)

    # Compute predictions
    yhat_tr_M = sanitize(alpha_pipeline_problem_2.predict(x_tr_MF))  # TODO␣
 ↪fixme, be sure to sanitize predictions
    yhat_va_N = sanitize(alpha_pipeline_problem_2.predict(x_va_NF))  # TODO␣
 ↪fixme, be sure to sanitize predictions

    assert np.all(yhat_va_N >= 0.0)
    assert np.all(yhat_va_N <= Y_MAX)

    # Calculate errors
    err_tr = calc_root_mean_squared_error(y_tr_M, yhat_tr_M) # TODO fixme
    err_va = calc_root_mean_squared_error(y_va_N, yhat_va_N) # TODO fixme

    fv2_err_tr_list.append(err_tr)
    fv2_err_va_list.append(err_va)

    # TODO store current pipeline for future use
    fv2_pipeline_list.append(alpha_pipeline_problem_2)
```

[55]:
```
# Overview of ridge results
for idx, alph in enumerate(alpha_list):
    print("Degree {} produced the following RMSEs: Training Data: {},␣
 ↪Validation Data: {}".format(alph, fv2_err_tr_list[idx],␣
 ↪fv2_err_va_list[idx]))
```

```
Degree 1e-10 produced the following RMSEs: Training Data: 2.900556, Validation
Data: 5.884819
Degree 1e-08 produced the following RMSEs: Training Data: 2.923625, Validation
Data: 5.834062
Degree 1e-06 produced the following RMSEs: Training Data: 3.069503, Validation
Data: 4.280301
Degree 0.0001 produced the following RMSEs: Training Data: 3.360597, Validation
Data: 4.057066
Degree 0.01 produced the following RMSEs: Training Data: 3.616048, Validation
Data: 3.931388
Degree 1.0 produced the following RMSEs: Training Data: 3.916736, Validation
```

```
Data: 3.998782
Degree 100.0 produced the following RMSEs: Training Data: 5.260844, Validation
Data: 5.321071
Degree 10000.0 produced the following RMSEs: Training Data: 7.953661, Validation
Data: 7.37561
Degree 1000000.0 produced the following RMSEs: Training Data: 8.228089,
Validation Data: 7.589769
```

6.0.1 Figure 2 in report

Make a line plot of mean-squared error on y-axis vs. alpha on x-axis.

[56]:
```
plot_train_and_valid_error_vs_hyper(
    alpha_list, fv2_err_tr_list, fv2_err_va_list,
    xlabel='alpha (L2 penalty)', leg_loc='upper left');
plt.gca().set_ylim([0, 10]);
plt.gca().set_xscale('log');
plt.title('Error vs. Alpha');
```

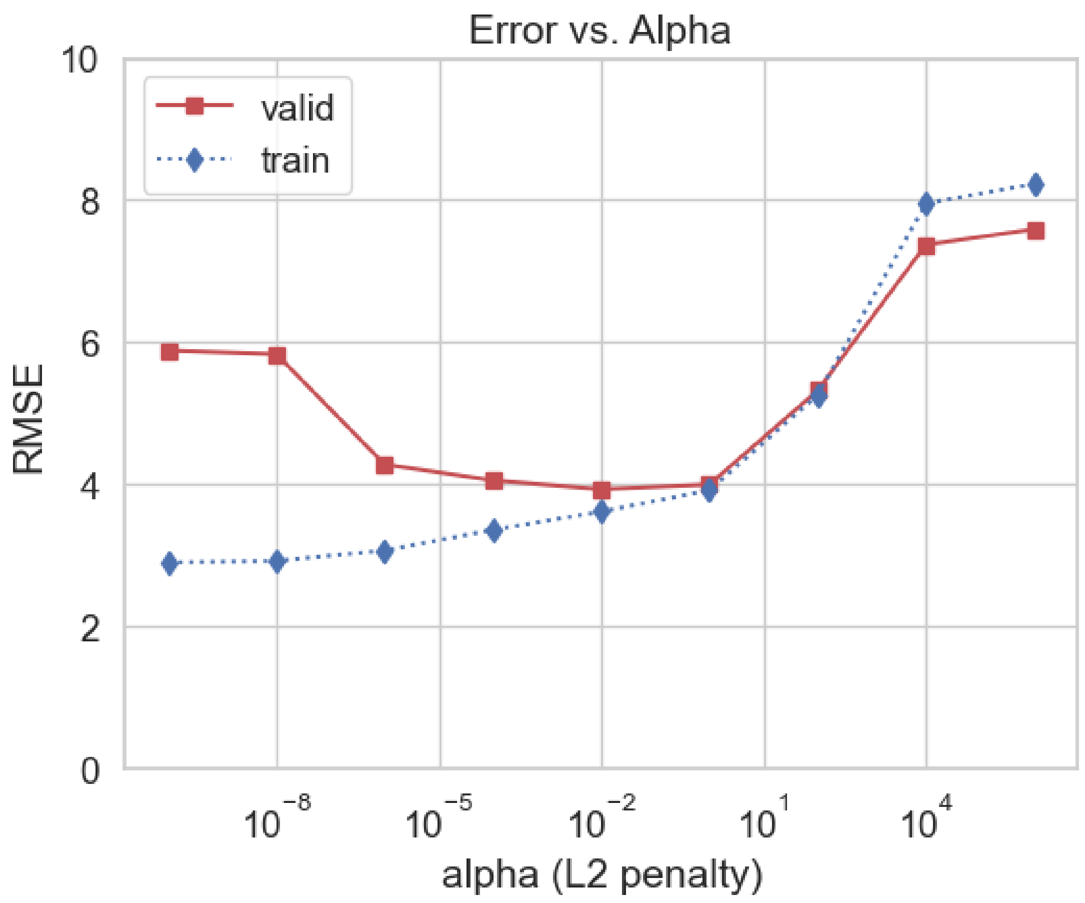


Figure 2: Error vs. Alpha Provide a 2 sentence caption answering the questions: How do validation and training set RMSE change with alpha? Do any trends emerge? What alpha value do you recommend based on this plot?

Training Error: RMSE increases monotonically, with an increasing gradient until it plateaus at an alpha value of (10^4).

Validation Error: RMSE decreases as alpha increases until (10^{-2}), when it achieves its lowest value at 3.93. After (10^{-2}), it follows the same increasing trend as the training error, increasing rapidly and plateauing at ($\alpha = 10^4$).

Implementation Step 2B Select the model hyperparameters that *minimize* your fixed validation set error. Using your already-trained model with these best hyperparameters, compute error on the *test* set.

[61]:
```
from IPython.display import display, Math, Latex
```

[65]:
```
print("Selected Parameters:")
display(Math(r"L2 = 10^{-2}"))
print("Fixed validation set estimate of heldout error:")
print("3.93")
print("Error on the test-set:")
print("{}".format(
    calc_root_mean_squared_error(y_te_P, sanitize(fv2_pipeline_list[4].
 ↪predict(x_te_PF)))
))
```

```
Selected Parameters:
```

$L2 = 10^{-2}$

```
Fixed validation set estimate of heldout error:
3.93
Error on the test-set:
3.877668
```

[66]:
```
# TODO Save this test set error value for later.
prob2_ridge_testerr = calc_root_mean_squared_error(y_te_P,␣
 ↪sanitize(fv2_pipeline_list[4].predict(x_te_PF)))
```

Short Answer 2a Inspect the learned weight parameters of your chosen degree-4 ridge regression model. How do their relative magnitudes compare to 1c above?

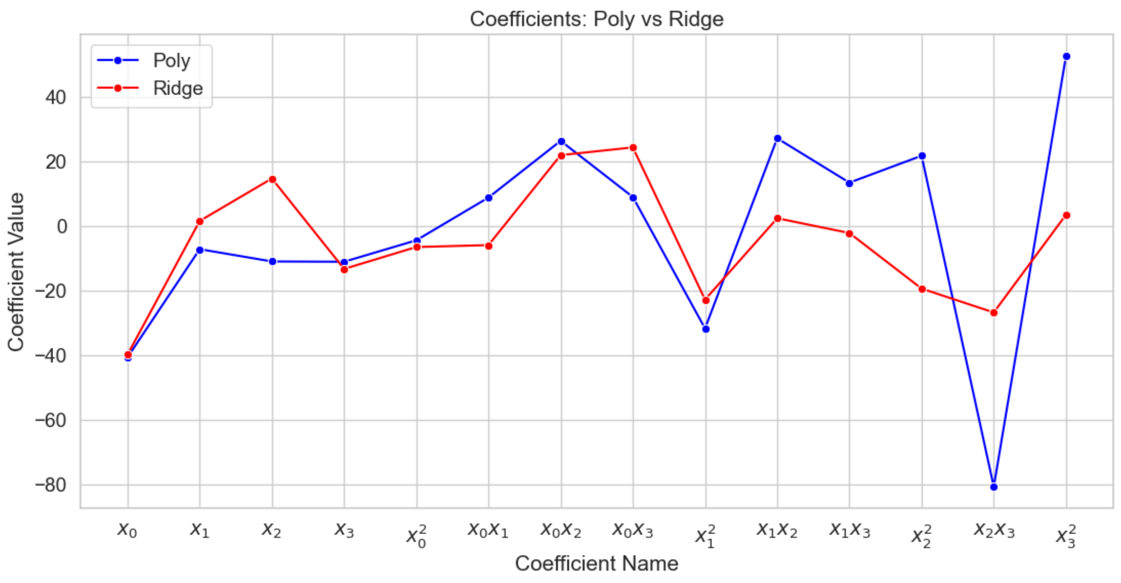
What is 1c? There is no c part to problem 1. Did you mean the magnitudes of the chosen model from problem 1?

[77]:
```
my_ridge_regr = fv2_pipeline_list[4].named_steps['linear_regr']
my_poly_regr = fv_pipeline_list[1].named_steps['linear_regr']
```

```
my_ridge_feat_names = fv2_pipeline_list[4].named_steps['poly_transformer'].
  ↪get_feature_names_out()
my_poly_feat_names = fv_pipeline_list[1].named_steps['poly_transformer'].
  ↪get_feature_names_out()
my_ridge_coef_values = my_ridge_regr.coef_
my_poly_coef_values = my_poly_regr.coef_
```

[141]:
```
def format_coeffs(coeff):
    coeff_list = coeff.split(" ")
    sep_exps = [x.split("^") for x in coeff_list]
    is_product = len(sep_exps) > 1

    if not is_product:
        base = "${}_{}".format(coeff[0], coeff[1])
        if "^" in coeff:
            return "^".join([base, coeff.split("^")[-1]])+"$"
        return base+"$"

    if is_product:
        final_str = ""
        for term in sep_exps:
            exp = "^".join(term)
            prefx = "_".join([exp[0], exp[1:]])
            final_str = final_str+" {}".format(prefx)

        final_str = "$"+final_str+"$"
        return final_str
```

[144]:
```
# Set the plot width
plt.figure(figsize=(13, 6))

# First line plot (Poly)
sns.lineplot(
    x=[format_coeffs(x) for x in my_poly_feat_names],
    y=my_poly_coef_values,
    color='blue',
    marker='o',
    label='Poly'
)

# Second line plot (Ridge)
sns.lineplot(
    x = [format_coeffs(x) for x in my_ridge_feat_names[:
  ↪len(my_poly_feat_names)]],
    y = my_ridge_coef_values[:len(my_poly_feat_names)],
    color='red',
    marker='o',
    label='Ridge'
)

# Add title and axis labels
plt.title('Coefficients: Poly vs Ridge')
plt.xlabel('Coefficient Name')
plt.ylabel('Coefficient Value')

# Show the legend
plt.legend()

# Display the plot
plt.show()
```



The plot above compares the coefficients of polynomial regression and ridge regression for the shared coefficients (note that ridge regression includes additional coefficients due to its higher degree). The values are visibly similar, and the contours of the two plots are alike. Both regressions evaluated their shared coefficients to approximately the same values.

Short Answer 2b in Report Your colleague suggests that you can determine the regularization strength `alpha` by minimizing the following loss on the *training* set:

$$\min_{w \in \mathbb{R}^F, b \in \mathbb{R}, \alpha \geq 0} \quad \sum_{n=1}^{N} (y_n - \hat{y}(x_n, w, b))^2 + \alpha \sum_{f=1}^{F} w_f^2$$

What value of α would you pick if you did this? Why is this problematic if your goal is to generalize to new data well?

Hint: Which value of α would minimize this loss function?

THis function would reach it's minimum (assuming $\alpha \geq 0$) at $\alpha = 0$. This would be problematic as it would eliminate the penality given to the weights, essentially yielding an unpenalized model which defeats the purpose of using α in the first place.

7 Data preprocessing for Problem 3

For this problem, you'll again use the provided training set and validation sets. However, you'll *merge* these into a large "development" set that contains 292 examples total.

[148]:
```
x_trva_LF = np.vstack([x_tr_MF, x_va_NF])
y_trva_L = np.hstack([y_tr_M, y_va_N])

print(x_trva_LF.shape)
```

```
(292, 4)
```

8 Problem 3: Cross Validation for Polynomial Feature Regression

8.0.1 Implementation step 3A

For each possible `alpha` value as well as each possible polynomial degree, train and evaluate a `Ridge` regression model across the entire train+validation set using 10-fold cross validation. Use the CV methods you implemented in `cross_validation.py`. For each possible hyperparameter (alpha value and degree value), your 10-fold CV procedure will give you an estimate of the training error and heldout validation error (averaged across all folds).

[ ]:
```
# # TEmplate
# def train_models_and_calc_scores_for_n_fold_cv(
#         estimator, x_NF, y_N, n_folds=3, random_state=0):
```

[ ]:
```
fv2_err_tr_list = []
fv2_err_va_list = []
fv2_pipeline_list = []

for alpha in alpha_list:

    # TODO create a pipeline using features with current degree value
    # TODO train this pipeline on provided training data
    alpha_pipeline_problem_2 = make_poly_ridge_regr_pipeline(degree = 4, alpha␣
 ↪= alpha)
    alpha_pipeline_problem_2.fit(x_tr_MF, y_tr_M)

    # Compute predictions
    yhat_tr_M = sanitize(alpha_pipeline_problem_2.predict(x_tr_MF))  # TODO␣
 ↪fixme, be sure to sanitize predictions
```

```
    yhat_va_N = sanitize(alpha_pipeline_problem_2.predict(x_va_NF))  # TODO␣
  ↪fixme, be sure to sanitize predictions

    assert np.all(yhat_va_N >= 0.0)
    assert np.all(yhat_va_N <= Y_MAX)

    # Calculate errors
    err_tr = calc_root_mean_squared_error(y_tr_M, yhat_tr_M) # TODO fixme
    err_va = calc_root_mean_squared_error(y_va_N, yhat_va_N) # TODO fixme

    fv2_err_tr_list.append(err_tr)
    fv2_err_va_list.append(err_va)

    # TODO store current pipeline for future use
    fv2_pipeline_list.append(alpha_pipeline_problem_2)
```

[149]:
```
K = 10 # num folds of CV
degree_list = [1, 2, 3, 4, 5, 6, 7]
alpha_list = np.logspace(-10, 6, 17)

ridge_param_list = []
my_rand_state = np.random.RandomState(seed=42)
for alpha in alpha_list:
    for degree in degree_list:
        ridge_param_list.append(dict(degree=degree, alpha=alpha))

cv_train_err_list = []
cv_valid_err_list = []
for param in ridge_param_list:
    # TODO make pipeline
    problem3_pipe = make_poly_ridge_regr_pipeline(degree = param['degree'],␣
  ↪alpha = param['alpha'])

    # TODO call your function to train a separate model for each fold and␣
  ↪return train and valid errors
    # Don't forget to pass random_state = SEED (where SEED is defined above) so␣
  ↪its reproducible
    # tr_error_K, valid_error_K = train_models_and_calc_scores_for_n_fold_cv()␣
  ↪# TODO
    tr_K, te_K = train_models_and_calc_scores_for_n_fold_cv(
        problem3_pipe,
        x_trva_LF,
        y_trva_L,
        n_folds = K,
        random_state = my_rand_state
    )
    err_tr = np.average(tr_K) # TODO fixme, compute average error across all␣
↪train folds
    err_va = np.average(te_K) # TODO fixme, compute average error across all␣
↪heldout folds

    cv_train_err_list.append(err_tr)
    cv_valid_err_list.append(err_va)
```

8.0.2 Implementation step 3B

Select the model hyperparameters that *minimize* your estimated cross-validation error. Using these best hyperparameters, retrain the model using the full development set (concatenating the predefined training and validation sets). Then compute that (retrained) model's error on the test set.

Save this test set error value for later.

[158]:
```
[np.where(cv_valid_err_list == np.min(cv_valid_err_list))[0][0]]
```

[158]:
```
[58]
```

[170]:
```
best_fit_idx = np.where(cv_valid_err_list == np.min(cv_valid_err_list))[0][0]
print(
    "Best fit params: ",
    ridge_param_list[best_fit_idx],
    "\nTrain error:",
    cv_train_err_list[best_fit_idx],
    "\nHeldout error:",
    cv_valid_err_list[best_fit_idx]
)
```

```
Best fit params:  {'degree': 3, 'alpha': 0.01}
Train error: 3.7000230000000003
Heldout error: 3.7912391
```

[159]:
```
best_fit = make_poly_ridge_regr_pipeline(degree = 3, alpha = 0.01)
best_fit.fit(x_trva_LF, y_trva_L)
```

[159]:
```
Pipeline(steps=[('rescaler', MinMaxScaler()),
                ('poly_transformer',
                 PolynomialFeatures(degree=3, include_bias=False)),
                ('linear_regr', Ridge(alpha=0.01))])
```

[171]:
```
print("Selected Parameters:")
print("Degree 3, alpha = 0.01")
print("10-fold CV estimate of heldout error:")
print("3.7912391")
print("Error on the test-set:")
print(
    calc_root_mean_squared_error(y_te_P, sanitize(best_fit.predict(x_te_PF)))
)
print("Error on train set:")
print(calc_root_mean_squared_error(y_trva_L, sanitize(best_fit.
 ↪predict(x_trva_LF))))
```

```
Selected Parameters:
Degree 3, alpha = 0.01
10-fold CV estimate of heldout error:
3.7912391
Error on the test-set:
3.790084
Error on train set:
3.707136
```

8.0.3 Table 3: Comparing Pipelines on the test set

In one neat table, please compare the *test set* root-mean-squared-error (RMSE) performance for the following regressors:

- Baseline: A predictor that always guesses the *mean* y value of the training set, regardless of the new test input
- The best Poly+Linear pipeline, picking degree to minimize val set error (from 1B)
- The best Poly+Ridge pipeline, fixing degree=4 and picking alpha to minimize val set error (from 2B)
- The best Poly+Ridge pipeline, picking degree and alpha to minimize 10-fold cross validation error (from 3B)

[167]:
```
print(
    "Base mean prediction. Train Err <<{}>> Test Err<<{}>>\n".format(
            calc_root_mean_squared_error(y_trva_L, [np.
 ↪mean(y_trva_L)]*len(y_trva_L)),
            calc_root_mean_squared_error(y_te_P, [np.mean(y_trva_L)]*len(y_te_P))
        )
)
```

```
Base mean prediction. Train Err <<8.015037>> Test Err<<7.131415>>
```

TODO make a table in your report using the saved values from 1B, 2B and 3B above. You can fill in the below Markdown table.

Name	Hyperparameter Value(s)	Test RMSE	Train RMSE	Heldout
Baseline mean prediction	None	7.13	8.02	N/A
Poly+Linear pipeine (1B)	Degree = 2	3.99	3.73	3.97
Poly+Ridge Alpha Search (2B)	Degree = 4, Alpha = 0.01	3.88	3.62	3.93

Name	Hyperparameter Value(s)	Test RMSE	Train RMSE	Heldout
Poly + Ridge Grid Search (3B)	Degree = 3, Alpha = 0.01	3.79	3.70	3.79

Table 3: *Provide a 2-4 sentence caption answering the question: What pipeline in Table 3 performed best in terms of heldout error, and why do you thik that pipeline performed the best? What pipeline in Table 3 performed worst in terms of heldout error, and why do you thik that pipeline performed the worst?*

8.0.4 Reflection

The Poly + Ridge Grid Search (3B) performed the best in the heldout error, and also in the test RMSE. I believe that it managed to achieve a higher granularity of analaysis on the impact of the features due to its higher degree, while also maintaining a balance of the weights using a proper alpha value. It is interesting that the best alpha value for the 3B model was the same as the best for the 2B model. To me this suggests that 2B was a good stepping stone.
The 1B model performed the worst on the heldout. It probably wasn't able to capture much granularity due to its low degree and couldn't balance the weights appropriately because it had no regularization.

Further reflection In order to get a better view into which model I would finally recommend, I will re-train all the models on the concatenated train+validation data and evaluate them on the test data.
Note that while 3B was the best model above, it was also trained using more data.

[172]:
```
# Let's evaluate the performance over the entire data set

# Re-Making Best Fits from P1 and P2
p1_best_fit = make_poly_linear_regr_pipeline(degree = 2)
p2_best_fit = make_poly_ridge_regr_pipeline(degree = 4, alpha = 0.01)
# Fitting
p1_best_fit.fit(x_trva_LF, y_trva_L)
p2_best_fit.fit(x_trva_LF, y_trva_L)
# Getting final errors
p1_tr_err = calc_root_mean_squared_error(y_trva_L, sanitize(p1_best_fit.
 ↪predict(x_trva_LF)))
p1_te_err = calc_root_mean_squared_error(y_te_P, sanitize(p1_best_fit.
 ↪predict(x_te_PF)))

p2_tr_err = calc_root_mean_squared_error(y_trva_L, sanitize(p2_best_fit.
 ↪predict(x_trva_LF)))
p2_te_err = calc_root_mean_squared_error(y_te_P, sanitize(p2_best_fit.
 ↪predict(x_te_PF)))
```

```
p3_tr_err = calc_root_mean_squared_error(y_trva_L, sanitize(best_fit.
 ↪predict(x_trva_LF)))
p3_te_err = calc_root_mean_squared_error(y_te_P, sanitize(best_fit.
 ↪predict(x_te_PF)))


print("After retraining and predicting on the same data set, we get the␣
 ↪following errors.\n\n",
      "Base mean prediction. Train Err <<{}>> Test Err<<{}>>\n".format(
          calc_root_mean_squared_error(y_trva_L, [np.
 ↪mean(y_trva_L)]*len(y_trva_L)),
          calc_root_mean_squared_error(y_te_P, [np.mean(y_trva_L)]*len(y_te_P))
      ),
      "Poly linear. Train Err <<{}>> Test Err <<{}>>\n".format(
          p1_tr_err, p1_te_err
      ),
      "Poly Ridge. Train Err <<{}>> Test Err <<{}>>\n".format(
          p2_tr_err, p2_te_err
      ),
      "Poly + Ridge CV. Train Err <<{}>> Test Err <<{}>>.".format(
          p3_tr_err, p3_te_err
      )

)
```

```
After retraining and predicting on the same data set, we get the following
errors.

 Base mean prediction. Train Err <<8.015037>> Test Err<<7.131415>>
 Poly linear. Train Err <<3.781584>> Test Err <<3.915952>>
 Poly Ridge. Train Err <<3.665313>> Test Err <<3.781111>>
 Poly + Ridge CV. Train Err <<3.707136>> Test Err <<3.790084>>.
```

Your caption here

TODO make a table in your report using the saved values from 1B, 2B and 3B above. You can fill in the below Markdown table.

Name	Hyperparameter Value(s)	TrVa RMSE	Test RMSE
Baseline mean prediction	None	8.02	7.13
Poly+Linear pipeine (1B)	Degree = 2	3.78	3.91
Poly+Ridge Alpha Search (2B)	Degree = 4, Alpha = 0.01	3.66	3.78
Poly + Ridge Grid Search (3B)	Degree = 3, Alpha = 0.01	3.70	3.79

Last Reflection After training and evaluating all the models on the same data set, we observe that 2B works best. It achieved the lowest error scores on the TrVa set used for training, and on the test set as well, beating 3B on the test set by 0.01.

8.1 Reflection

Please fill in the following information about your HW1 solution.

How many hours did HW1 take you? __________

Did you use any resources outside of class materials and official documentation, such as StackOverflow threads or ChatGPT? If so, please list such resources below:

It took me one hour to do the .py parts of the homework, and about 2-3 hrs to complete the jupyter notebook. Totaling somewhere between 3-4.
I used the numpy docs, and asked Windows Co-Pilot questions about the syntax of certain numpy functions (sometimes faster than looking up the docs).

[ ]:
```

```