hw3: Lasso regression and projected gradient descent

March 10, 2020

Author: Yibo Liu (yl6769)

1 Ridge Regression

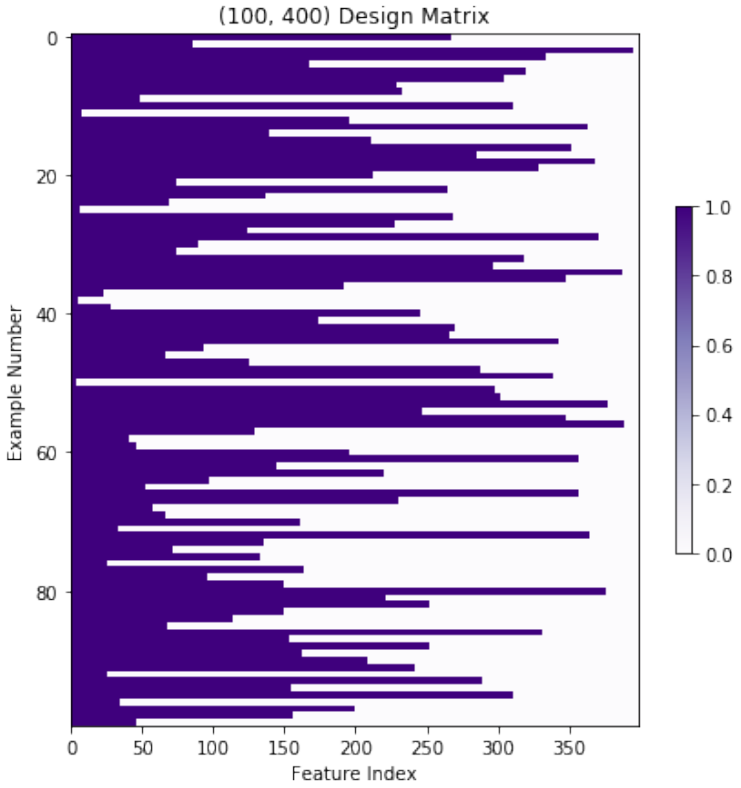
1.1 Choose bset lambda and plot

[1]:
```
import numpy as np
import pandas as pd
import itertools
import matplotlib.pyplot as plt
%matplotlib inline

from scipy.optimize import minimize

from sklearn.base import BaseEstimator, RegressorMixin, clone
from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import GridSearchCV, PredefinedSplit
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.metrics import confusion_matrix

from load_data import load_problem

PICKLE_PATH = 'lasso_data.pickle'
```

[2]:
```
#load data

x_train, y_train, x_val, y_val, target_fn, coefs_true, featurize =␣
 ↪load_problem(PICKLE_PATH)
X_train = featurize(x_train)
X_val = featurize(x_val)
```

[3]:
```
#Visualize training data

fig, ax = plt.subplots(figsize = (7,7))
ax.set_title("({0}, {1}) Design Matrix".format(X_train.shape[0], X_train.
 ↪shape[1]))
```

```
ax.set_xlabel("Feature Index")
ax.set_ylabel("Example Number")
temp = ax.imshow(X_train, cmap=plt.cm.Purples, aspect="auto")
plt.colorbar(temp, shrink=0.5);
```



[3]:
```
class RidgeRegression(BaseEstimator, RegressorMixin):
        """ ridge regression"""

        def __init__(self, l2reg=1):
                if l2reg < 0:
                        raise ValueError('Regularization penalty should be at␣
↪least 0.')
                self.l2reg = l2reg

        def fit(self, X, y=None):
                n, num_ftrs = X.shape
                # convert y to 1-dim array, in case we're given a column vector
                y = y.reshape(-1)
                def ridge_obj(w):
                        predictions = np.dot(X,w)
                        residual = y - predictions
                        empirical_risk = np.sum(residual**2) / n
```

```
                        l2_norm_squared = np.sum(w**2)
                        objective = empirical_risk + self.l2reg * l2_norm_squared
                        return objective
                self.ridge_obj_ = ridge_obj

                w_0 = np.zeros(num_ftrs)
                self.w_ = minimize(ridge_obj, w_0).x
                return self

        def predict(self, X, y=None):
                try:
                        getattr(self, "w_")
                except AttributeError:
                        raise RuntimeError("You must train classifer before␣
↪predicting data!")
                return np.dot(X, self.w_)

        def score(self, X, y):
                # Average square error
                try:
                        getattr(self, "w_")
                except AttributeError:
                        raise RuntimeError("You must train classifer before␣
↪predicting data!")
                residuals = self.predict(X) - y
                return np.dot(residuals, residuals)/len(y)
```

[4]:
```
def compare_our_ridge_with_sklearn(X_train, y_train, l2_reg=1):

        # Fit with sklearn -- need to multiply l2_reg by sample size, since their
        # objective function has the total square loss, rather than average␣
↪square
        # loss.
        n = X_train.shape[0]
        sklearn_ridge = Ridge(alpha=n*l2_reg, fit_intercept=False,␣
↪normalize=False)
        sklearn_ridge.fit(X_train, y_train)
        sklearn_ridge_coefs = sklearn_ridge.coef_

        # Now run our ridge regression and compare the coefficients to sklearn's
        ridge_regression_estimator = RidgeRegression(l2reg=l2_reg)
        ridge_regression_estimator.fit(X_train, y_train)
        our_coefs = ridge_regression_estimator.w_

        print("Hoping this is very close to 0:{}".format(np.sum((our_coefs -␣
↪sklearn_ridge_coefs)**2)))
```

[5]:
```
default_params = np.unique(np.concatenate((10.**np.arange(-6,1,1), np.
↪arange(1,3,.3))))

def do_grid_search_ridge(X_train, y_train, X_val, y_val, params =␣
↪default_params):

        X_train_val = np.vstack((X_train, X_val))
        y_train_val = np.concatenate((y_train, y_val))
        val_fold = [-1]*len(X_train) + [0]*len(X_val) #0 corresponds to␣
↪validation

        param_grid = [{'l2reg':params}]

        ridge_regression_estimator = RidgeRegression()
        grid = GridSearchCV(ridge_regression_estimator,
                                                param_grid,
                                                return_train_score=True,
                                                cv =␣
↪PredefinedSplit(test_fold=val_fold),
                                                refit = True,
                                                scoring =␣
↪make_scorer(mean_squared_error,

↪ greater_is_better = False))
        grid.fit(X_train_val, y_train_val)

        df = pd.DataFrame(grid.cv_results_)
        # Flip sign of score back, because GridSearchCV likes to maximize,
        # so it flips the sign of the score if "greater_is_better=FALSE"
        df['mean_test_score'] = -df['mean_test_score']
        df['mean_train_score'] = -df['mean_train_score']
        cols_to_keep = ["param_l2reg", "mean_test_score","mean_train_score"]
        df_toshow = df[cols_to_keep].fillna('-')
        df_toshow = df_toshow.sort_values(by=["param_l2reg"])
        return grid, df_toshow
```

[24]:
```
grid, results = do_grid_search_ridge(X_train, y_train, X_val, y_val)
```

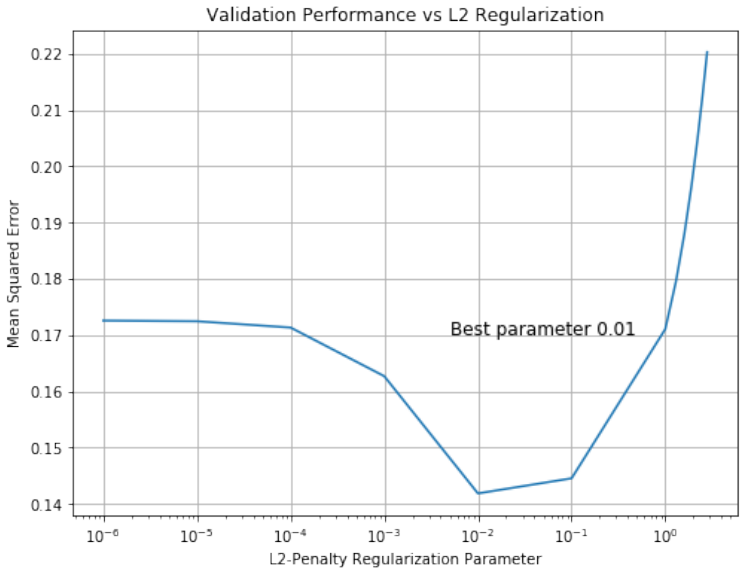
[9]:
```
results
```

[9]:

	param_l2reg	mean_test_score	mean_train_score
0	0.000001	0.172579	0.006752
1	0.000010	0.172464	0.006752
2	0.000100	0.171345	0.006774
3	0.001000	0.162705	0.008285
4	0.010000	0.141887	0.032767

5	0.100000	0.144566	0.094953
6	1.000000	0.171068	0.197694
7	1.300000	0.179521	0.216591
8	1.600000	0.187993	0.233450
9	1.900000	0.196361	0.248803
10	2.200000	0.204553	0.262958
11	2.500000	0.212530	0.276116
12	2.800000	0.220271	0.288422

[15]:
```
# Plot validation performance vs regularization parameter
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L2 Regularization")
ax.set_xlabel("L2-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")
ax.semilogx(results["param_l2reg"], results["mean_test_score"])
ax.text(0.005,0.17,"Best parameter {0}".format(grid.best_params_['l2reg']),␣
 ↪fontsize = 12);
```



Conclusion: The best parameter is l2_reg = 0.01.

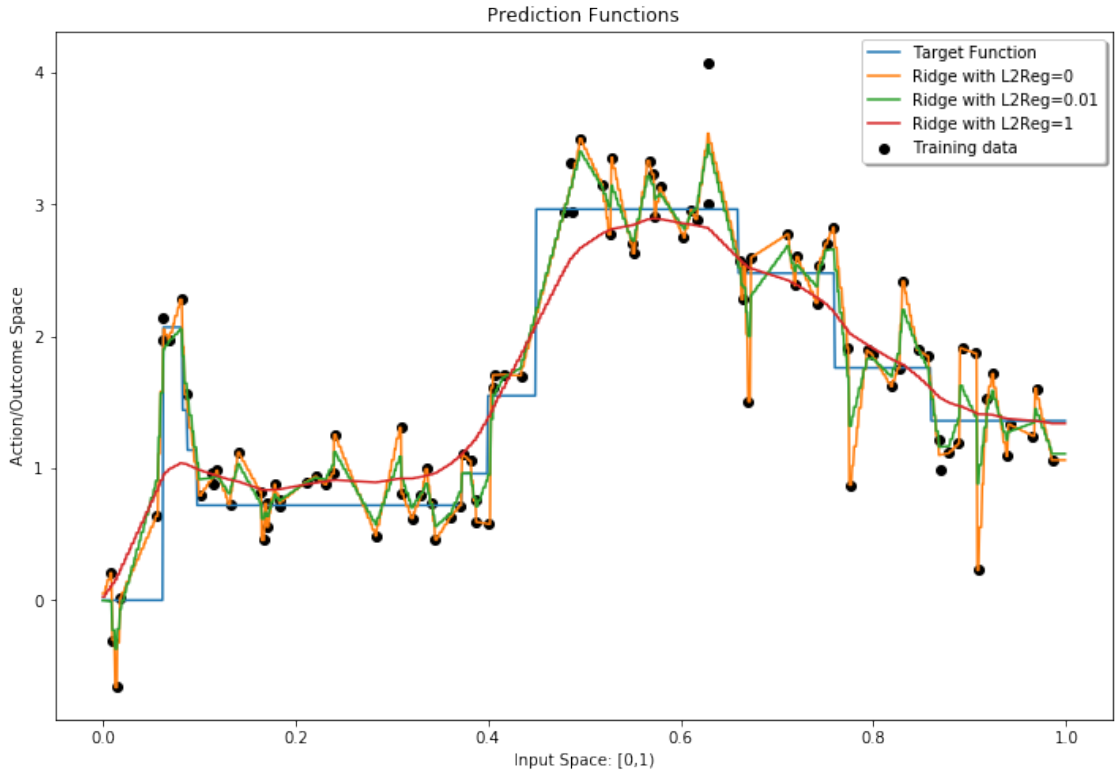
1.2 Visualize prediction function

[16]:
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))

pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds":␣
 ↪target_fn(x)})
```

```
l2regs = [0, grid.best_params_['l2reg'], 1]
X = featurize(x)
for l2reg in l2regs:
    ridge_regression_estimator = RidgeRegression(l2reg=l2reg)
    ridge_regression_estimator.fit(X_train, y_train)
    name = "Ridge with L2Reg="+str(l2reg)
    pred_fns.append({"name":name,
                     "coefs":ridge_regression_estimator.w_,
                     "preds": ridge_regression_estimator.predict(X) })
```

[18]:
```
def plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best"):

        fig, ax = plt.subplots(figsize = (12,8))
        ax.set_xlabel('Input Space: [0,1)')
        ax.set_ylabel('Action/Outcome Space')
        ax.set_title("Prediction Functions")
        plt.scatter(x_train, y_train, color="k", label='Training data')
        for i in range(len(pred_fns)):
                ax.plot(x, pred_fns[i]["preds"], label=pred_fns[i]["name"])
        legend = ax.legend(loc=legend_loc, shadow=True)
        return fig
```

[19]:
```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



Visualizing weights

[20]:
```
def compare_parameter_vectors(pred_fns):

        fig, axs = plt.subplots(len(pred_fns),1, sharex=True, figsize = (20,20))
        num_ftrs = len(pred_fns[0]["coefs"])
        for i in range(len(pred_fns)):
                title = pred_fns[i]["name"]
                coef_vals = pred_fns[i]["coefs"]
                axs[i].bar(range(num_ftrs), coef_vals, color = "tab:purple")
                axs[i].set_xlabel('Feature Index')
                axs[i].set_ylabel('Parameter Value')
                axs[i].set_title(title)

        fig.subplots_adjust(hspace=0.4)
        return fig
```

[21]:
```
compare_parameter_vectors(pred_fns);
```

Conclusion: With L2 regularization factor increasing, the weights tend to be less sparse and have smaller scale. As we can see in the figures above, the weights roughly range from -1.5 to 1 when l2_reg=0, from -0.4 to 0.4 when l2_reg=0.01, from -0.02 to 0.04 when l2_reg=1. The coefficients whose indexes are close to the weights in the target function have the most weight.

1.3 Examine model coefficients

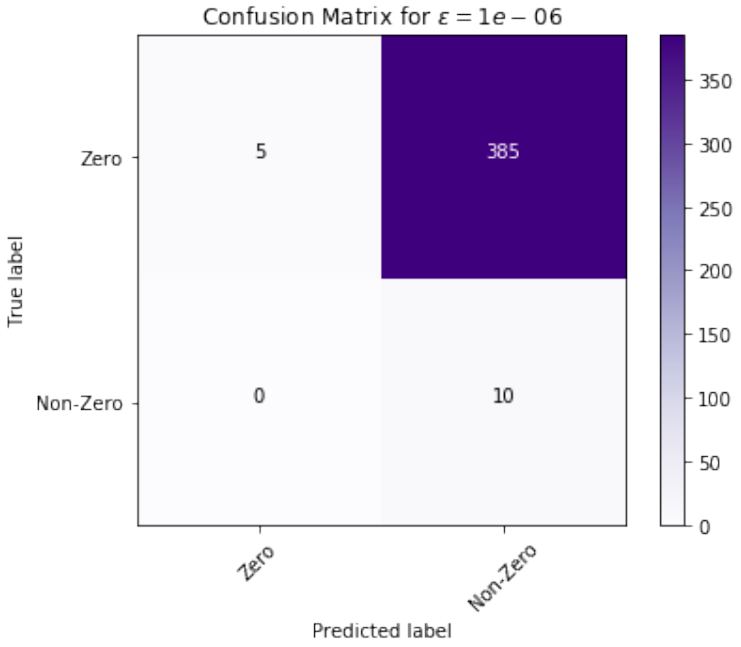
Confusion Matrix: We can try to predict the features with corresponding weight zero. We will fix a threshold `eps` such that any value between `-eps` and `eps` will get counted as zero. We take the remaining features to have positive value. These predictions of can be compared to the weights for the target function.

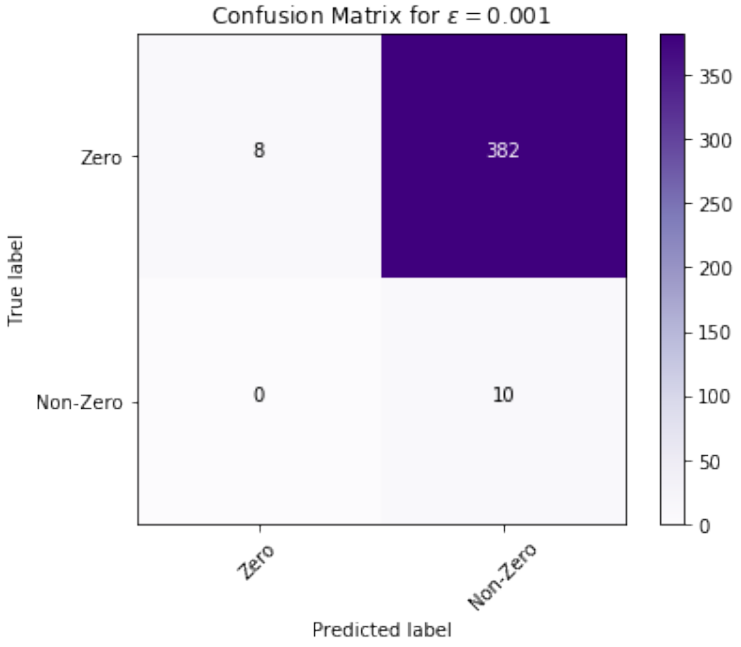
[34]:
```
def plot_confusion_matrix(cm, title, classes):
        plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Purples)
```

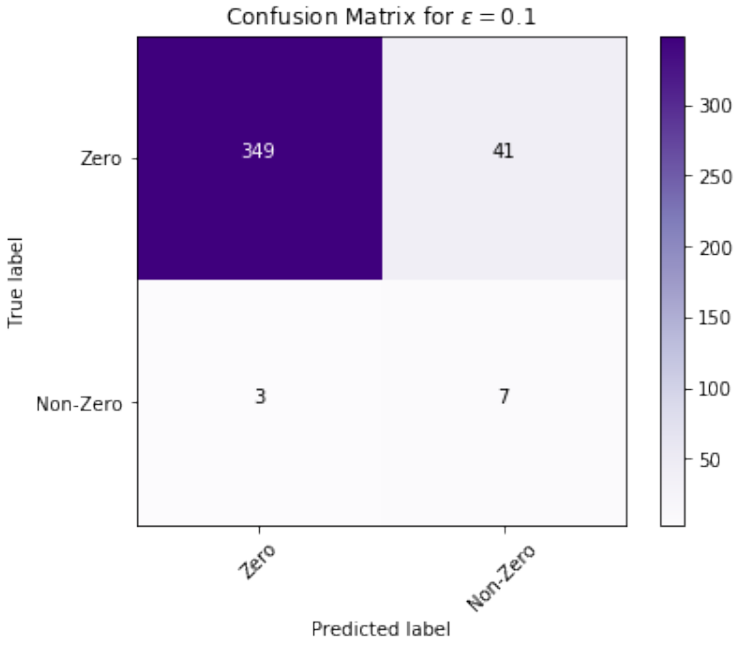
```
        plt.title(title)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, rotation=45)
        plt.yticks(tick_marks, classes)

        thresh = cm.max() / 2.
        for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
                plt.text(j, i, format(cm[i, j], 'd'),
                                 horizontalalignment="center",
                                 color="white" if cm[i, j] > thresh else␣
↪"black")

        plt.tight_layout()
        plt.ylabel('True label')
        plt.xlabel('Predicted label')
        plt.ylim(1.5,-0.5)
```

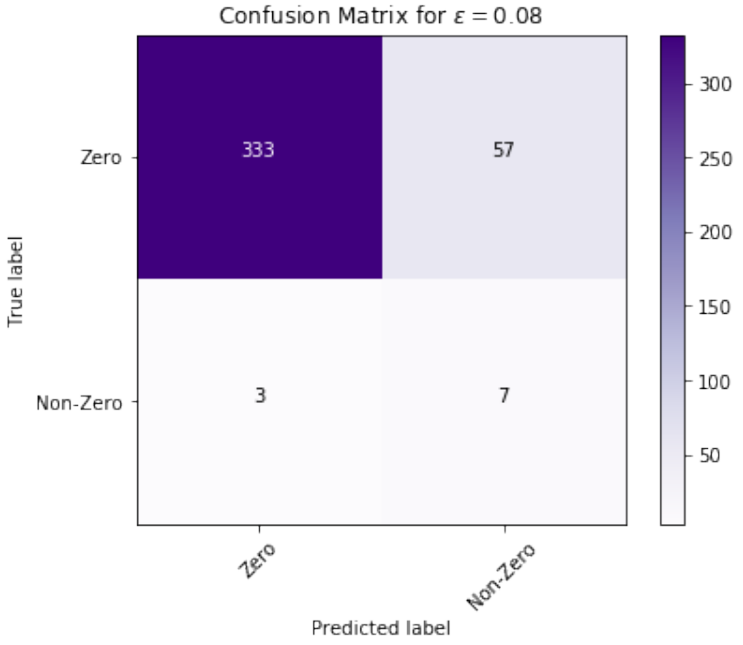
[43]:
```
print(pred_fns[2]["name"])
coef_vals = pred_fns[2]["coefs"]
bin_coefs_true = [0 if i==0 else 1 for i in coefs_true] # your code goes here
eps_list = [1e-6,1e-3,1e-1] # your code goes here
for eps in eps_list:
    bin_coefs_estimated = [0 if abs(i)<eps else 1 for i in coef_vals] # your␣
↪code goes here
    cnf_matrix = confusion_matrix(bin_coefs_true, bin_coefs_estimated)
    plt.figure()
    plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon =␣
↪{}$".format(eps), classes=["Zero", "Non-Zero"])
```

```
Ridge with L2Reg=0.01
```


Confusion Matrix for $\epsilon = 1e-06$
Zero
Non-Zero
True label
5
385
0
10
Zero
Non-Zero
Predicted label
350
300
250
200
150
100
50
0


Confusion Matrix for $\epsilon = 0.001$
Zero
Non-Zero
True label
8
382
0
10
Zero
Non-Zero
Predicted label
350
300
250
200
150
100
50
0





2 Lasso Regression

Coordinate Descent for Lasso Regression (Shooting Algorithm): For the shooting algorithm, we need to compute the Lasso Regression objective for the stopping condition. Moreover we need a

threshold function at each iteration along with the solution to Ridge Regression for initial weights.

2.1 Vector expression of a and c

Answer:
$a_j = 2X_j^T X_j$
$c_j = 2X_j^T(y - Xw + w_j X_j)$

2.2 Compute Lasso solution

[6]:
```
def soft_threshold(a, delta):
    return np.sign(a)*(abs(a)-delta) if np.abs(a)-delta >= 0 else 0

def compute_sum_sqr_loss(X, y, w):
    return np.dot(np.dot(X,w)-y, np.dot(X,w)-y)

def compute_lasso_objective(X, y, w, l1_reg=0):
    return np.dot(np.dot(X,w)-y, np.dot(X,w)-y) + l1_reg*np.linalg.norm(w, ord=1)

def get_ridge_solution(X, y, l2_reg=0.01):
    ridge_regression_estimator = RidgeRegression(l2reg=l2_reg)
    ridge_regression_estimator.fit(X_train, y_train)
    return ridge_regression_estimator.w_
```

[7]:
```
def shooting_algorithm(X, y, w0=None, l1_reg = 1., max_num_epochs = 1000,␣
↪min_obj_decrease=1e-8, random=False):
    if w0 is None:
        w = np.zeros(X.shape[1])
    else:
        w = np.copy(w0)
    d = X.shape[1]
    epoch = 0
    obj_val = compute_lasso_objective(X, y, w, l1_reg)
    obj_decrease = min_obj_decrease + 1.
    while (obj_decrease>min_obj_decrease) and (epoch<max_num_epochs):
        obj_old = obj_val
        # Cyclic coordinates descent
        coordinates = range(d)
        # Randomized coordinates descent
        if random:
            coordinates = np.random.permutation(d)
        for j in coordinates:

            ####
            # your code goes here
            a =2*np.dot(X.T[j],X.T[j])
            c =2*np.dot((y-np.dot(X,w)+w[j]*X.T[j]),X.T[j])
```

```
            if a == 0 and c==0:
                w[j]=0
            else:
                w[j] = soft_threshold(c/a,l1_reg/a)
            ####

        obj_val = compute_lasso_objective(X, y, w, l1_reg)
        obj_decrease = obj_old - obj_val
        epoch += 1
    print("Ran for "+str(epoch)+" epochs. " + 'Lowest loss: ' + str(obj_val))
    return w
```

[8]:
```
class LassoRegression(BaseEstimator, RegressorMixin):
    """ Lasso regression"""
    def __init__(self, l1_reg=1.0, randomized=False):
        if l1_reg < 0:
            raise ValueError('Regularization penalty should be at least 0.')
        self.l1_reg = l1_reg
        self.randomized = randomized


    def fit(self, X, y, max_epochs = 1000, coef_init=None): # 1000
        # convert y to 1-dim array, in case we're given a column vector
        y = y.reshape(-1)
        if coef_init is None:
            coef_init = get_ridge_solution(X,y, self.l1_reg)

        ####
        # your code goes here
#         self.lasso_obj_ = compute_lasso_objective(X, y, w, self.l1_reg)
        self.w_ = shooting_algorithm(X, y, w0=coef_init, l1_reg = self.l1_reg,
                                     max_num_epochs = max_epochs,␣
 ↪min_obj_decrease=1e-8, random=self.randomized)
        ####

        return self

    def predict(self, X, y=None):
        try:
            getattr(self, "w_")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!
 ↪")

        return np.dot(X, self.w_)

    def score(self, X, y):
```

```
        try:
            getattr(self, "w_")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!
↪")

        return compute_sum_sqr_loss(X, y, self.w_)/len(y)
```

[9]:
```
def compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=1, coef_init=None,␣
↪randomized = False):

    # Fit with sklearn -- need to divide l1_reg by 2*sample size, since they
    # use a slightly different objective function.
    n = X_train.shape[0]
    sklearn_lasso = Lasso(alpha=l1_reg/(2*n), fit_intercept=False,␣
↪normalize=False)
    sklearn_lasso.fit(X_train, y_train)
    sklearn_lasso_coefs = sklearn_lasso.coef_
    sklearn_lasso_preds = sklearn_lasso.predict(X_train)

    # Now run our lasso regression and compare the coefficients to sklearn's

    ####
    # your code goes here
    lasso_regression_estimator = LassoRegression(l1_reg, randomized)
    lasso_regression_estimator.fit(X_train, y_train, max_epochs = 1000,␣
↪coef_init=coef_init)
    our_coefs = lasso_regression_estimator.w_
    lasso_regression_preds = lasso_regression_estimator.predict(X_train)
    ####

    # Let's compare differences in predictions
    print("Hoping this is very close to 0 (predictions): {}".format( np.
↪mean((sklearn_lasso_preds - lasso_regression_preds)**2)))
    # Let's compare differences parameter values
    print("Hoping this is very close to 0 (param): {}".format(np.sum((our_coefs␣
↪- sklearn_lasso_coefs)**2)))
```

[14]:
```
reg = 1
print('cyclic, ridge init, l1 reg=', reg)
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=reg,
                               coef_init=None ,randomized = False) #
print('cyclic, zero init, l1 reg=', reg)
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=reg,
                               coef_init=np.zeros(X_train.shape[1])␣
↪,randomized = False) #
```

```
print('random, ridge init, l1 reg=', reg)
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=reg,
                                    coef_init=None ,randomized = True) #
print('random, zero init, l1 reg=', reg)
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=reg,
                                    coef_init=np.zeros(X_train.shape[1])␣
 ↪,randomized = True) #
```

```
cyclic, ridge init, l1 reg= 1
Ran for 818 epochs. Lowest loss: 16.19773797905091
Hoping this is very close to 0 (predictions): 3.548427634086183e-07
Hoping this is very close to 0 (param): 0.8259772498441471
cyclic, zero init, l1 reg= 1
Ran for 824 epochs. Lowest loss: 16.19773797187303
Hoping this is very close to 0 (predictions): 3.55488803386613e-07
Hoping this is very close to 0 (param): 8.306670429853805e-05
random, ridge init, l1 reg= 1
Ran for 697 epochs. Lowest loss: 16.197738076995925
Hoping this is very close to 0 (predictions): 4.5090327410700293e-07
Hoping this is very close to 0 (param): 3.9427230411616017
random, zero init, l1 reg= 1
Ran for 687 epochs. Lowest loss: 16.197738149460825
Hoping this is very close to 0 (predictions): 4.52580049464455e-07
Hoping this is very close to 0 (param): 3.47349148975904
```

Conclusion: For the chosen lambda=1, we found that cyclic coordinate descent achieves smaller validation error than randomized coordinate descent. For cyclic coordinate descent, ridge initiation converges faster. In th following experiments, we set *cyclic, ridge init* as default setting.

2.3 Select lambda

Grid Search to Tune Hyperparameter: Now let's use sklearn to help us do hyperparameter tuning GridSearchCv.fit by default splits the data into training and validation itself; we want to use our own splits, so we need to stack our training and validation sets together, and supply an index (validation_fold) to specify which entries are train and which are validation.

[10]:
```
def do_grid_search_lasso(X_train, y_train, X_val, y_val, params):
    ####
    ## your code goes here
        X_train_val = np.vstack((X_train, X_val))
        y_train_val = np.concatenate((y_train, y_val))
        val_fold = [-1]*len(X_train) + [0]*len(X_val) #0 corresponds to␣
 ↪validation

        param_grid = [{'l1_reg':params, 'randomized':[False]}]

        lasso_regression_estimator = LassoRegression()
        grid = GridSearchCV(lasso_regression_estimator,
```

```
                                                    param_grid,
                                                    return_train_score=True,
                                                    cv =␣
↪PredefinedSplit(test_fold=val_fold),
                                                    refit = True,
                                                    scoring =␣
↪make_scorer(mean_squared_error,
                                                                                    ␣
↪ greater_is_better = False))
      grid.fit(X_train_val, y_train_val)

      df = pd.DataFrame(grid.cv_results_)
      # Flip sign of score back, because GridSearchCV likes to maximize,
      # so it flips the sign of the score if "greater_is_better=FALSE"
      df['mean_test_score'] = -df['mean_test_score']
      df['mean_train_score'] = -df['mean_train_score']
      cols_to_keep = ["param_l1_reg", "mean_test_score","mean_train_score"]
      df_toshow = df[cols_to_keep].fillna('-')
      df_toshow = df_toshow.sort_values(by=["param_l1_reg"])
      return grid, df_toshow
  ####
```

[ ]:
```
# regs = np.unique(np.concatenate((10.**np.arange(-6,1,1), np.arange(1,3,.3))))
# default_params = np.array([0.01])
regs = np.array([1e-2,1e-1,0.5,0.8,1,1.3,1.6,2])
grid, results = do_grid_search_lasso(X_train, y_train, X_val, y_val, params =␣
↪regs)
```

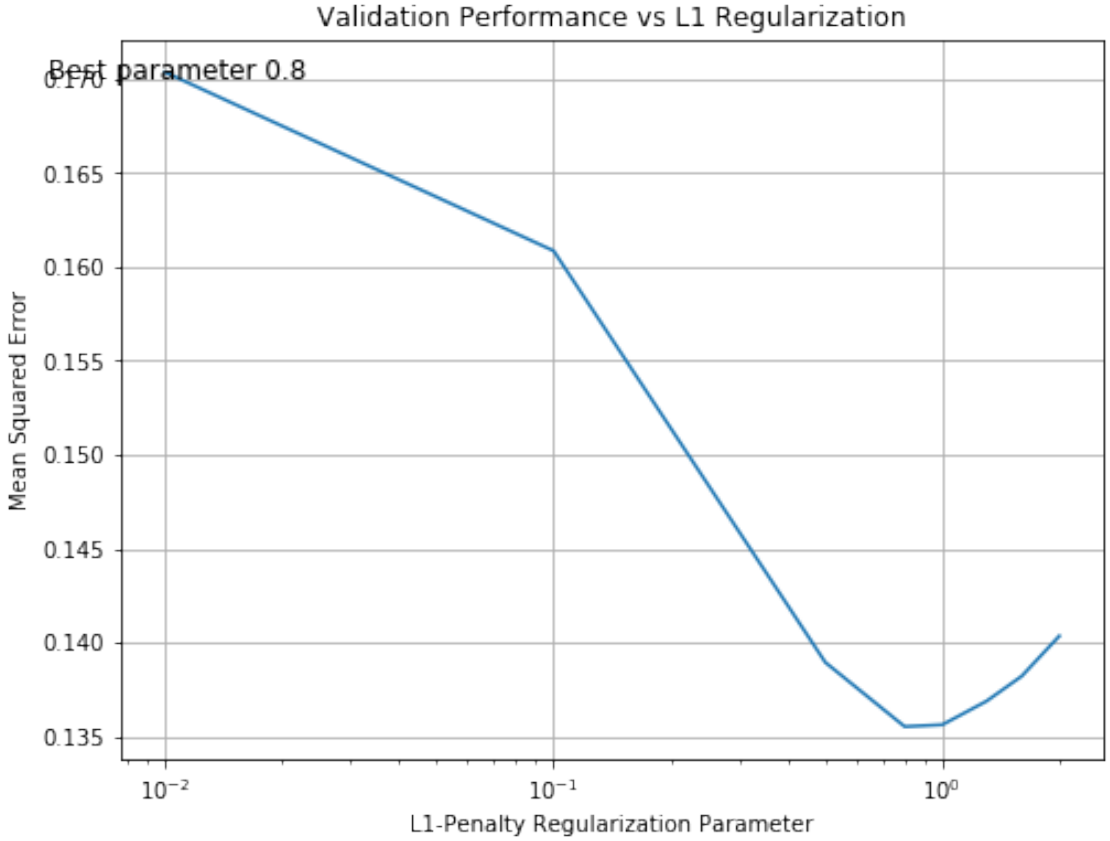
```
Ran for 1000 epochs. Lowest loss: 1.047100066017412
Ran for 1000 epochs. Lowest loss: 3.904984445421853
Ran for 936 epochs. Lowest loss: 11.40399746525369
Ran for 919 epochs. Lowest loss: 14.640976534049317
Ran for 818 epochs. Lowest loss: 16.19773797905091
Ran for 810 epochs. Lowest loss: 18.208737599347504
Ran for 799 epochs. Lowest loss: 20.146331963438833
```

[47]:
```
results
```

[47]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.01	0.170345	0.006805
1	0.10	0.160840	0.011138
2	0.50	0.138948	0.048919
3	0.80	0.135533	0.075868
4	1.00	0.135630	0.091950
5	1.30	0.136889	0.096653
6	1.60	0.138213	0.099934
7	2.00	0.140353	0.105364

[48]:
```
# Plot validation performance vs regularization parameter

fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L1 Regularization")
ax.set_xlabel("L1-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")

####
## your code goes here
ax.semilogx(results["param_l1_reg"], results["mean_test_score"])
####

ax.text(0.005,0.17,"Best parameter {0}".format(grid.best_params_['l1_reg']),␣
 ↪fontsize = 12);
```



Comparing to the Target Function: Let's plot prediction functions and compare coefficients for several fits and the target function.

Let's create a list of dicts called `pred_fns`. Each dict has a "name" key and a "preds" key. The value corresponding to the "preds" key is an array of predictions corresponding to the input vector x.

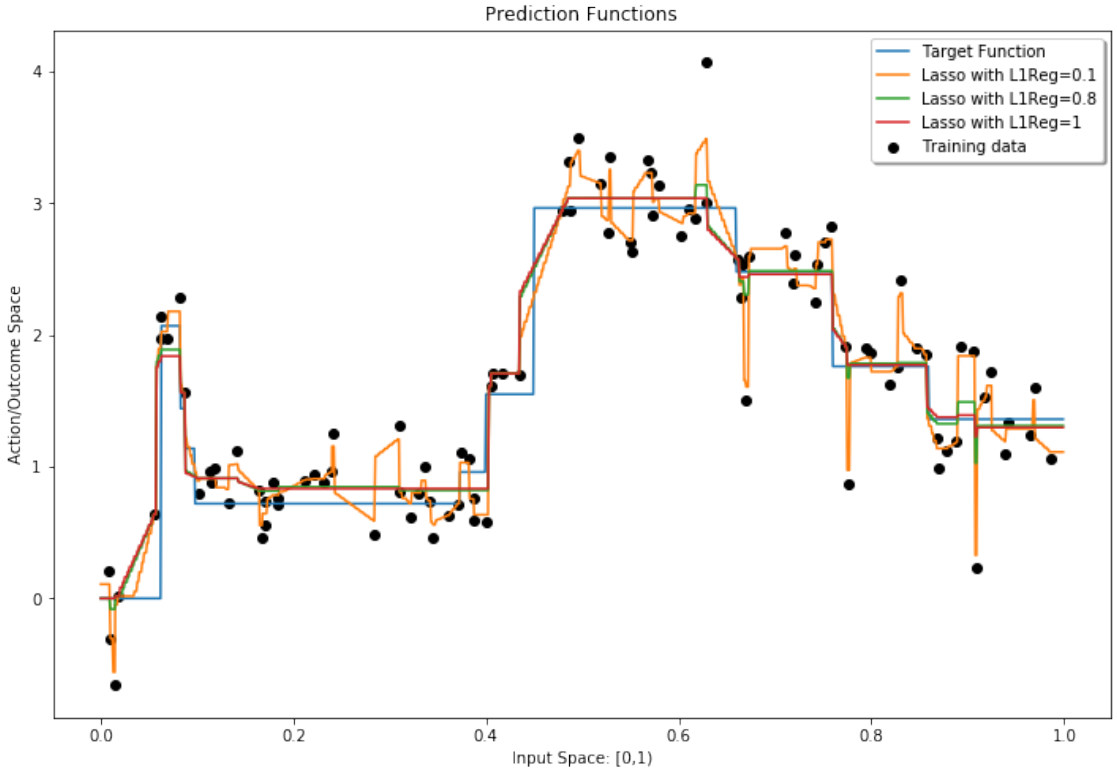
x_train and y_train are the input and output values for the training data

[12]:
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))

pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds":␣
 ↪target_fn(x)})

l1regs = [0.1, 0.8, 1]
# l1regs = [0.1, grid.best_params_['l1_reg'], 1]
X = featurize(x)
for l1reg in l1regs:
    lasso_regression_estimator = LassoRegression(l1_reg=l1reg)
    lasso_regression_estimator.fit(X_train, y_train)
    name = "Lasso with L1Reg="+str(l1reg)

    ####
    ## your code goes here
    pred_fns.append({"name":name,
                     "coefs":lasso_regression_estimator.w_,
                     "preds": lasso_regression_estimator.predict(X) })
    ####
```

```
Ran for 1000 epochs. Lowest loss: 3.904984445421853
Ran for 919 epochs. Lowest loss: 14.640976534049317
Ran for 818 epochs. Lowest loss: 16.19773797905091
```

[13]:
```
def plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best"):

        fig, ax = plt.subplots(figsize = (12,8))
        ax.set_xlabel('Input Space: [0,1)')
        ax.set_ylabel('Action/Outcome Space')
        ax.set_title("Prediction Functions")
        plt.scatter(x_train, y_train, color="k", label='Training data')
        for i in range(len(pred_fns)):
                ax.plot(x, pred_fns[i]["preds"], label=pred_fns[i]["name"])
        legend = ax.legend(loc=legend_loc, shadow=True)
        return fig
```

[59]:
```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



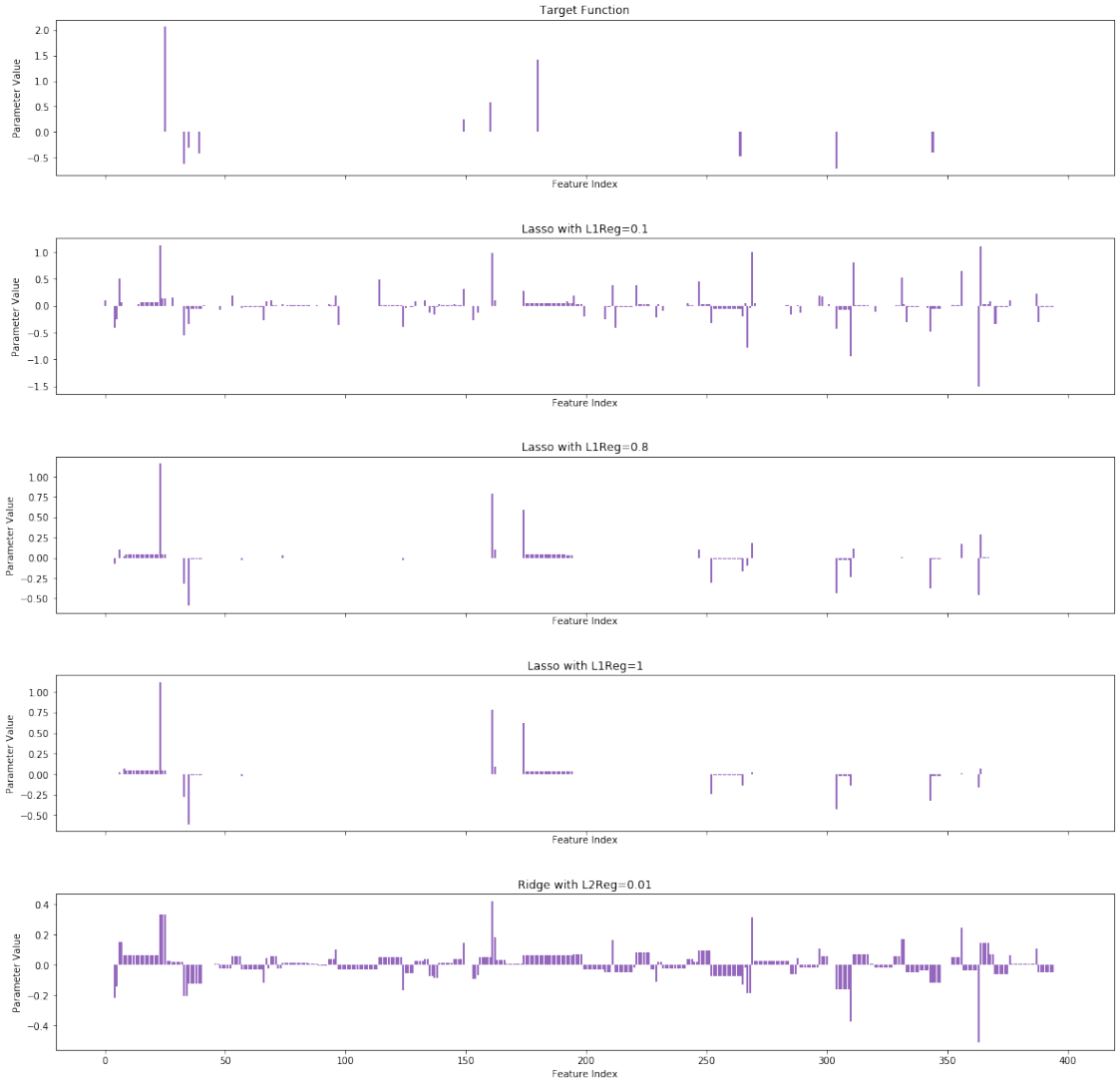
Visualizing the Weights: Using `pred_fns` let's try to see how sparse the weights are...

[14]:
```
# best ridge config
ridge_regression_estimator = RidgeRegression(l2reg=0.01)
ridge_regression_estimator.fit(X_train, y_train)
pred_fns.append({"name": "Ridge with L2Reg=0.01",
                    "coefs": ridge_regression_estimator.w_,
                    "preds": ridge_regression_estimator.predict(X) })
```

[15]:
```
def compare_parameter_vectors(pred_fns):

        fig, axs = plt.subplots(len(pred_fns),1, sharex=True, figsize = (20,20))
        num_ftrs = len(pred_fns[0]["coefs"])
        for i in range(len(pred_fns)):
                title = pred_fns[i]["name"]
                coef_vals = pred_fns[i]["coefs"]
                axs[i].bar(range(num_ftrs), coef_vals, color = "tab:purple")
                axs[i].set_xlabel('Feature Index')
                axs[i].set_ylabel('Parameter Value')
                axs[i].set_title(title)

        fig.subplots_adjust(hspace=0.4)
```

```
        return fig
```

[19]:
```
compare_parameter_vectors(pred_fns)
```



Conclusion: From the weights visualization, we can see that (1) the larger the l1 regularization factor is, the larger the sparsity the weights have; (2) the weights has higher sparsity with lasso regression comparing to ridge regression. The best model is lasso regression with lambda=0.8, using cyclic coordiniate descent and w0 initialized with ridge solution. The validation error is 0.135.

2.4 Implement homotopy method

[16]:
```
def get_lambda_max_no_bias(X, y):
    return 2 * np.max(np.abs(np.dot(y, X)))
```

[12]:
```
class LassoRegularizationPath:
    def __init__(self, estimator, tune_param_name):
        self.estimator = estimator
        self.tune_param_name = tune_param_name

    def fit(self, X, y, reg_vals, coef_init=None, warm_start=True):
        # reg_vals is a list of regularization parameter values to solve for.
        # Solutions will be found in the order given by reg_vals.

        #convert y to 1-dim array, in case we're given a column vector
        y = y.reshape(-1)

        if coef_init is not None:
            coef_init = np.copy(coef_init)

        self.results = []
        for reg_val in reg_vals:
            estimator = clone(self.estimator)

            ####
            ## your code goes here
            estimator.__init__(reg_val)
            estimator.fit(X_train, y_train, coef_init=coef_init)
            ####

            self.results.append({"reg_val":reg_val, "estimator":estimator})

        return self

    def predict(self, X, y=None):
        predictions = []
        for i in range(len(self.results)):
            preds = self.results[i]["estimator"].predict(X)
            reg_val = self.results[i]["reg_val"]
            predictions.append({"reg_val":reg_val, "preds":preds})
        return predictions

    def score(self, X, y=None):
        scores = []
        for i in range(len(self.results)):
            score = self.results[i]["estimator"].score(X, y)
            reg_val = self.results[i]["reg_val"]
```

```
            scores.append({"reg_val":reg_val, "score":score})
        return scores
```

[13]:
```
def do_grid_search_homotopy(X_train, y_train, X_val, y_val,
                            reg_vals=None, w0=None):
    if reg_vals is None:
        lambda_max = get_lambda_max_no_bias(X_train, y_train)
        reg_vals = [lambda_max * (.8**n) for n in range(0, 30)]

    ####
    ## your code goes here
    estimator = LassoRegression()
    lasso_reg_path_estimator = LassoRegularizationPath(estimator,␣
 ↪tune_param_name="l1_reg")
    lasso_reg_path_estimator.fit(X_train, y_train,
                                 reg_vals=reg_vals[:], coef_init=w0,
                                 warm_start=True)
    ####

#     estimator = LassoRegression()
#     lasso_reg_path_estimator = LassoRegularizationPath(estimator,␣
 ↪tune_param_name="l1_reg")
#     lasso_reg_path_estimator.fit(X_train, y_train,
#                                  reg_vals=reg_vals[:], coef_init=w0,
#                                  warm_start=True)

    return lasso_reg_path_estimator, reg_vals
```

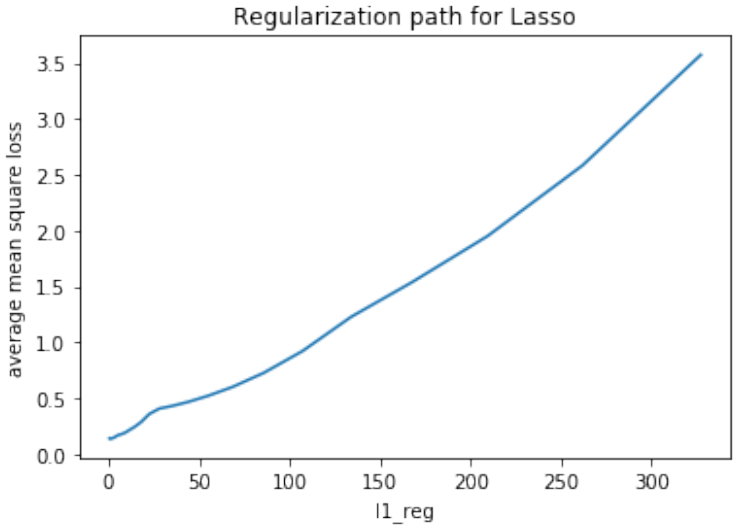
[18]:
```
lasso_reg_path_estimator, reg_vals = do_grid_search_homotopy(X_train,
                                                             y_train,
                                                             X_val,
                                                             y_val,
                                                             None)
```

```
Ran for 2 epochs. Lowest loss: 359.6674002813196
Ran for 595 epochs. Lowest loss: 348.52108633392805
Ran for 648 epochs. Lowest loss: 323.53716482932157
Ran for 603 epochs. Lowest loss: 293.2292649086329
Ran for 738 epochs. Lowest loss: 262.2363733418835
Ran for 739 epochs. Lowest loss: 231.30364706207007
Ran for 741 epochs. Lowest loss: 202.01748558953204
Ran for 742 epochs. Lowest loss: 175.68296893418443
Ran for 744 epochs. Lowest loss: 152.73952247982828
Ran for 745 epochs. Lowest loss: 133.12872422255808
Ran for 746 epochs. Lowest loss: 116.62091783025068
Ran for 746 epochs. Lowest loss: 102.89040534141196
Ran for 747 epochs. Lowest loss: 91.40127987759519
```

```
Ran for 747 epochs. Lowest loss: 80.59513458329275
Ran for 747 epochs. Lowest loss: 70.6513115830218
Ran for 747 epochs. Lowest loss: 61.83896928106218
Ran for 747 epochs. Lowest loss: 54.08108000167425
Ran for 746 epochs. Lowest loss: 47.326669329763874
Ran for 741 epochs. Lowest loss: 41.56428606891879
Ran for 764 epochs. Lowest loss: 36.6971294461177
Ran for 811 epochs. Lowest loss: 32.32317412217333
Ran for 830 epochs. Lowest loss: 28.434762287699513
Ran for 826 epochs. Lowest loss: 25.071530449274626
Ran for 784 epochs. Lowest loss: 22.211091299891443
Ran for 801 epochs. Lowest loss: 19.79969729565775
Ran for 812 epochs. Lowest loss: 17.789508948933666
Ran for 819 epochs. Lowest loss: 16.121413407945717
Ran for 921 epochs. Lowest loss: 14.563979323761105
Ran for 933 epochs. Lowest loss: 12.993126650524049
Ran for 936 epochs. Lowest loss: 11.487542082572876
```

[23]:
```
val_scores = lasso_reg_path_estimator.score(X_val, y_val)
reg_vals = [result["reg_val"] for result in val_scores]
scores = [result["score"] for result in val_scores]
plt.xlabel("l1_reg")
plt.ylabel("average mean square loss")
plt.title("Regularization path for Lasso")
plt.plot(reg_vals, scores)
```

[23]:
```
[<matplotlib.lines.Line2D at 0x7ff1009fb950>]
```



2.5 Additional experiment on centering y

[20]:
```
# centering y
offset = (max(y_train)+min(y_train))/2
y_train_ct = y_train-offset
y_val_ct = y_val-offset
print(max(y_train),min(y_train))
print(offset)
print(max(y_train_ct),min(y_train_ct))
```

```
4.0693497310902735 -0.6608769948624821
1.7042363681138957
2.365113362976378 -2.365113362976378
```

[21]:
```
# lasso regression (centered)
regs = np.array([1e-2,1e-1,0.5,0.8,1,1.3,1.6,2])
grid, results = do_grid_search_lasso(X_train, y_train_ct, X_val, y_val_ct,␣
 ↪params = regs)
```

```
Ran for 1000 epochs. Lowest loss: 1.0601255608125273
Ran for 1000 epochs. Lowest loss: 4.04423935378151
Ran for 1000 epochs. Lowest loss: 12.213744561516709
Ran for 981 epochs. Lowest loss: 15.961994336494122
Ran for 824 epochs. Lowest loss: 17.858915920765458
Ran for 817 epochs. Lowest loss: 20.340334944349987
Ran for 657 epochs. Lowest loss: 22.688461025243868
Ran for 651 epochs. Lowest loss: 25.636003648373357
Ran for 1000 epochs. Lowest loss: 87.65773373458256
```

[22]:
```
# lasso regression (centered)
results
```

[22]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.01	0.170421	0.006804
1	0.10	0.161361	0.011042
2	0.50	0.139746	0.048494
3	0.80	0.136331	0.075441
4	1.00	0.136398	0.091880
5	1.30	0.137609	0.098884
6	1.60	0.139261	0.104960
7	2.00	0.142973	0.113989

[47]:
```
# lasso regression (uncentered)
results
```

[47]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.01	0.170345	0.006805
1	0.10	0.160840	0.011138

```
2          0.50          0.138948          0.048919
3          0.80          0.135533          0.075868
4          1.00          0.135630          0.091950
5          1.30          0.136889          0.096653
6          1.60          0.138213          0.099934
7          2.00          0.140353          0.105364
```

[23]:
```
# ridge regression (centered)
grid, results = do_grid_search_ridge(X_train, y_train_ct, X_val, y_val_ct)
```

[24]:
```
# ridge regression (centered)
results
```

[24]:
```
    param_l2reg  mean_test_score  mean_train_score
0      0.000001          0.180448          0.006752
1      0.000010          0.180334          0.006752
2      0.000100          0.179222          0.006773
3      0.001000          0.170612          0.008264
4      0.010000          0.150020          0.032739
5      0.100000          0.171803          0.114165
6      1.000000          0.277469          0.269884
7      1.300000          0.293173          0.289101
8      1.600000          0.307081          0.305417
9      1.900000          0.319917          0.320013
10     2.200000          0.332029          0.333464
11     2.500000          0.343596          0.346084
12     2.800000          0.354709          0.358045
```

[27]:
```
# ridge regression (uncentered)
results
```

[27]:
```
    param_l2reg  mean_test_score  mean_train_score
0      0.000001          0.172579          0.006752
1      0.000010          0.172464          0.006752
2      0.000100          0.171345          0.006774
3      0.001000          0.162705          0.008285
4      0.010000          0.141887          0.032767
5      0.100000          0.144566          0.094953
6      1.000000          0.171068          0.197694
7      1.300000          0.179521          0.216591
8      1.600000          0.187993          0.233450
9      1.900000          0.196361          0.248803
10     2.200000          0.204553          0.262958
11     2.500000          0.212530          0.276116
12     2.800000          0.220271          0.288422
```

Conclusion: For lasso regression, the best lambda is still 0.8, but the validation error increased

from 0.135630 to 0.136331 after centering y. For Ridge regression, the best lambda is still 0.01, but the validation error increased from 0.141887 to 0.150020 after centering y.

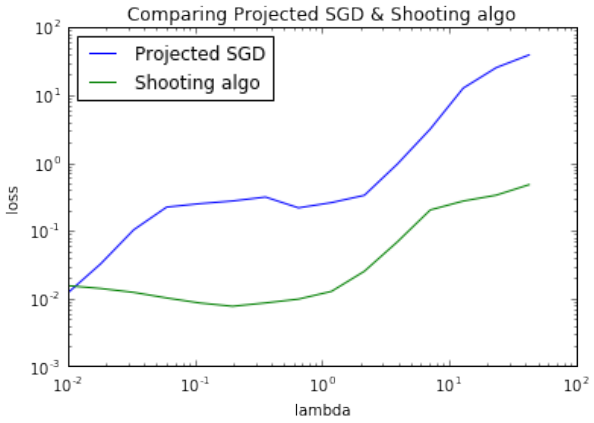
3 Projected gradient descent

3.1 Implement projected SGD

[14]:
```
def projection_SGD_split(X, y, theta_positive_0, theta_negative_0, lambda_reg =
↪1.0, alpha = 0.1, num_iter = 1000):
    m, n = X.shape
    theta_positive = np.zeros(n)
    theta_negative = np.zeros(n)
    theta_positive[0:n] = theta_positive_0
    theta_negative[0:n] = theta_negative_0
    times = 0
    theta = theta_positive - theta_negative
    loss = compute_sum_sqr_loss(X, y, theta)
    loss_change = 1.
    while (loss_change>1e-6) and (times<num_iter):
        loss_old = loss
        for i in range(m):
            ####
            ## your code goes here
            diff = np.dot(X[i], theta)-y[i]
            grad_p = 2*np.dot(diff, X[i]) + lambda_reg
            grad_n = -2*np.dot(diff, X[i]) + lambda_reg
            theta_positive = theta_positive - alpha*grad_p
            theta_negative = theta_negative - alpha*grad_n
#             print(theta_positive)
            for j in range(n):
                if theta_positive[j] < 0:
                    theta_positive[j] = 0
                elif theta_negative[j] < 0:
                    theta_negative[j] = 0
            theta = theta_positive - theta_negative
            ####
        loss = compute_sum_sqr_loss(X, y, theta)
        loss_change = np.abs(loss - loss_old)
        print(loss, loss_change)
        times +=1

    print('(SGD) Ran for {} epochs. Loss:{} Lambda: {}'.
↪format(times,loss,lambda_reg))
    return theta
```

[17]:
```
x_training, y_training, x_validation, y_validation, target_fn, coefs_true,␣
 ↪featurize = load_problem(PICKLE_PATH)
X_training = featurize(x_training)
X_validation = featurize(x_validation)
D = X_training.shape[1]

lambda_max = get_lambda_max_no_bias(x_training, y_training)
reg_vals = [lambda_max * (.6**n) for n in range(15, 25)]

loss_SGD_list = []
loss_shooting = []
loss_GD_list = [] # ?

for lambda_value in reg_vals:
    ####
    ## your code goes here
    w0 = np.zeros(D)
    lasso_shooting_estimator = LassoRegression(lambda_value)
    lasso_shooting_estimator.fit(X_train, y_train, coef_init = w0)
    val_loss_shooting = lasso_shooting_estimator.score(X_val, y_val)
    loss_shooting.append(val_loss_shooting)

    theta_positive_0 = np.zeros(D)
    theta_negative_0 = np.zeros(D)
    theta = projection_SGD_split(X_train, y_train, theta_positive_0,␣
 ↪theta_negative_0, lambda_reg=lambda_value)
    val_loss_sgd = compute_sum_sqr_loss(X_val, y_val, theta)/len(y_val)
    loss_SGD_list.append(val_loss_sgd)
    ####
```

[ ]:
```
# Plot validation performance vs regularization parameter

fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L1 Regularization")
ax.set_xlabel("L1-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")

plt.semilogx(reg_vals, loss_shooting, label = 'Shooting Method')
plt.semilogx(reg_vals, loss_SGD_list, label = 'Projection SGD')
plt.legend(loc='upper left')
plt.show();
```



3.2 Choose best lambda

[21]:
```
# Report the best

lambda_best_SGD = reg_vals[np.argmin(loss_SGD_list)]
theta_lasso_SGD_best = projection_SGD_split(X_training, y_training,␣
 ↪theta_positive_ini, theta_negative_ini, lambda_reg=lambda_best_SGD, alpha = 0.
 ↪01)
print('Best lambda for SGD is {0} with loss {1}'.format(lambda_best_SGD, np.
 ↪min(loss_SGD_list)))
```

4 Deriving lambda_max

4.1 Compute oneside directional derivative

Answer:

$$J'(0;v) = lim_{h\to 0}\frac{||Xhv-y||_2^2+\lambda||hv||_1-||y||^2-\lambda||0||}{h} = -2(Xv)^Ty+\lambda||v||$$

4.2 Write expression for C

Answer:

$$-2(Xv)^Ty+\lambda||v|| \geq 0$$

$$\Rightarrow \lambda \geq \frac{2vX^Ty}{||v||_1}$$

So

$$C = \frac{2vX^Ty}{||v||_1}$$

4.3 Prove w=0 with conditions

Answer:

$$\frac{2vX^Ty}{||v||_1} = \frac{\sum_{i=1}^n 2v_i x_i y_i}{\sum_{i=1}^n v_i} = \frac{\sum_{i=1}^n 2v_i (x_i y_i)}{\sum_{i=1}^n v_i} \leq \frac{\sum_{i=1}^n 2v_i max(x_i y_i)}{\sum_{i=1}^n v_i} = 2||X^Ty||_\infty$$

The maximum of lower bounds is $2||X^Ty||_\infty$ if and only if $\lambda \geq C$. Therefore, $\lambda = C$ is the most strict condition on λ. So $w = 0$ is a minimizer of $J(w)$ if and only if $\lambda \geq 2||X^Ty||_\infty$.