Homework 3: Lasso Regression and Projected Gradient Descent

Student Name: Kuan-Lin Liu

Net ID: kll482

Packages

In [1]:

```
import numpy as np
np.random.seed(42)
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 Lasso, Ridge
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'
```

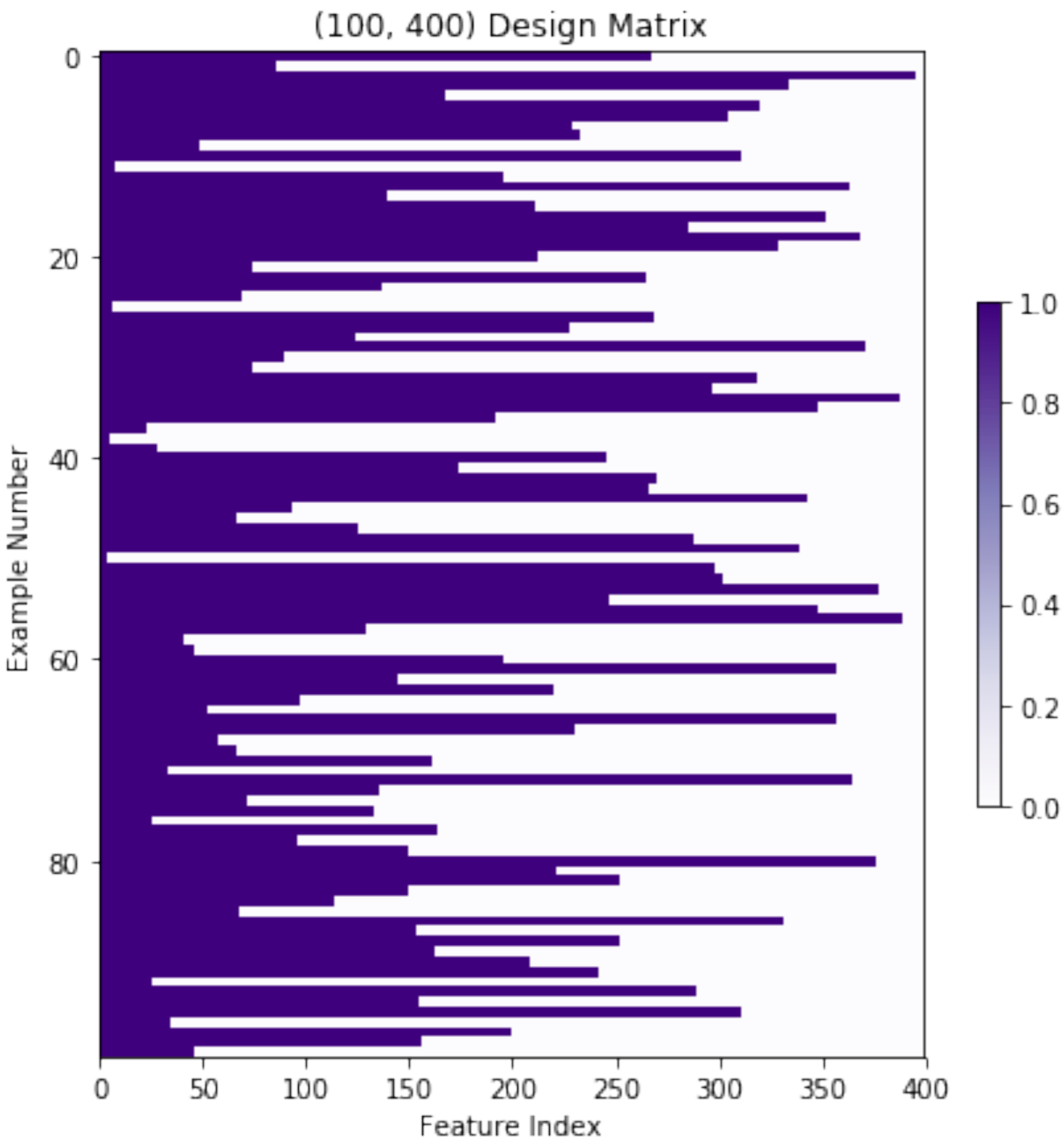
Dataset

In [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)
```

In [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);
```



Ridge Regression

Here we will try to fit the dataset with a Ridge Regression model. The steps are

- Determine a class for the model supporting methods
  - fit
  - predict
  - score
- Search for hyperparameters through trial and error
  - evaluate the average training and validating error for each hyperparameter
- Plot the distributions of weight on the features
  - Does Ridge Regression give us sparsity
- Threshold the values to compare zero/non-zero against the weights of the target function

Class for Ridge Regression

In [4]:

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

        def __init__(self, l2reg=1):
                if l2reg < 0:
                        raise ValueError('Regularization penalty should be at le
ast 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_square
d
                        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 pred
icting 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 pred
icting data!")
                residuals = self.predict(X) - y
                return np.dot(residuals, residuals)/len(y)
```

We can compare to the `sklearn` implementation.

In [5]:

```
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 thei
r
        # objective function has the total square loss, rather than average squa
re
        # loss.
        n = X_train.shape[0]
        sklearn_ridge = Ridge(alpha=n*l2_reg, fit_intercept=False, normalize=Fal
se)
        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 - skl
earn_ridge_coefs)**2)))
```

In [6]:

```
compare_our_ridge_with_sklearn(X_train, y_train, l2_reg=1.5)
```

```
Hoping this is very close to 0:4.6933165148971775e-11
```

1.

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.

In [7]:

```
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 validati
on

        param_grid = [{'l2reg':params}]

        ridge_regression_estimator = RidgeRegression()
        grid = GridSearchCV(ridge_regression_estimator,
                                                 param_grid,
                                                 return_train_score=True,
                                                 cv = PredefinedSplit(test_fold=v
al_fold),
                                                 refit = True,
                                                 scoring = make_scorer(mean_squar
ed_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
```

In [8]:

```
grid, results = do_grid_search_ridge(X_train, y_train, X_val, y_val)
```

In [9]:

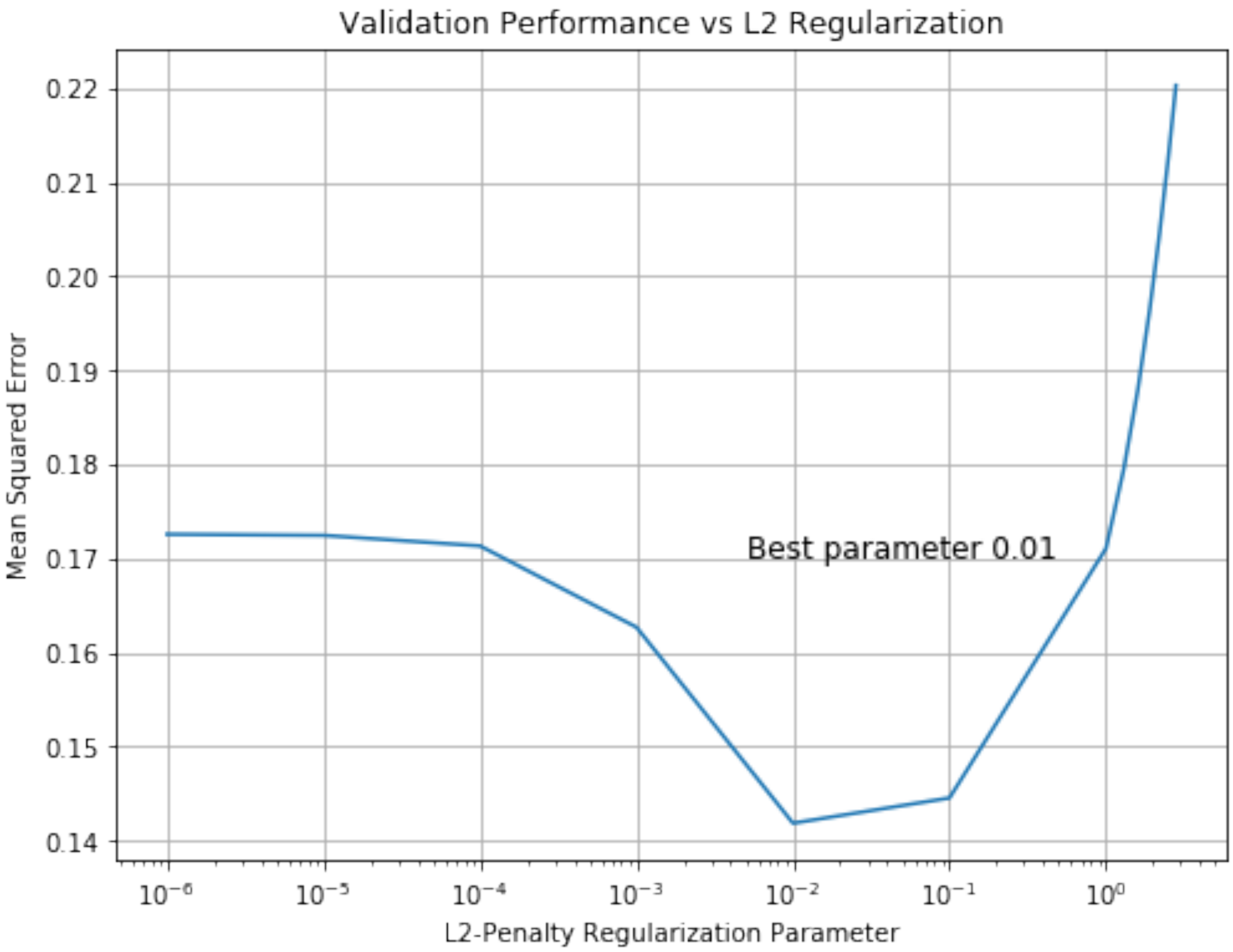
```
results
```

Out[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

In [10]:

```
# 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']), font
size = 12);
```



2.

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

In [11]:

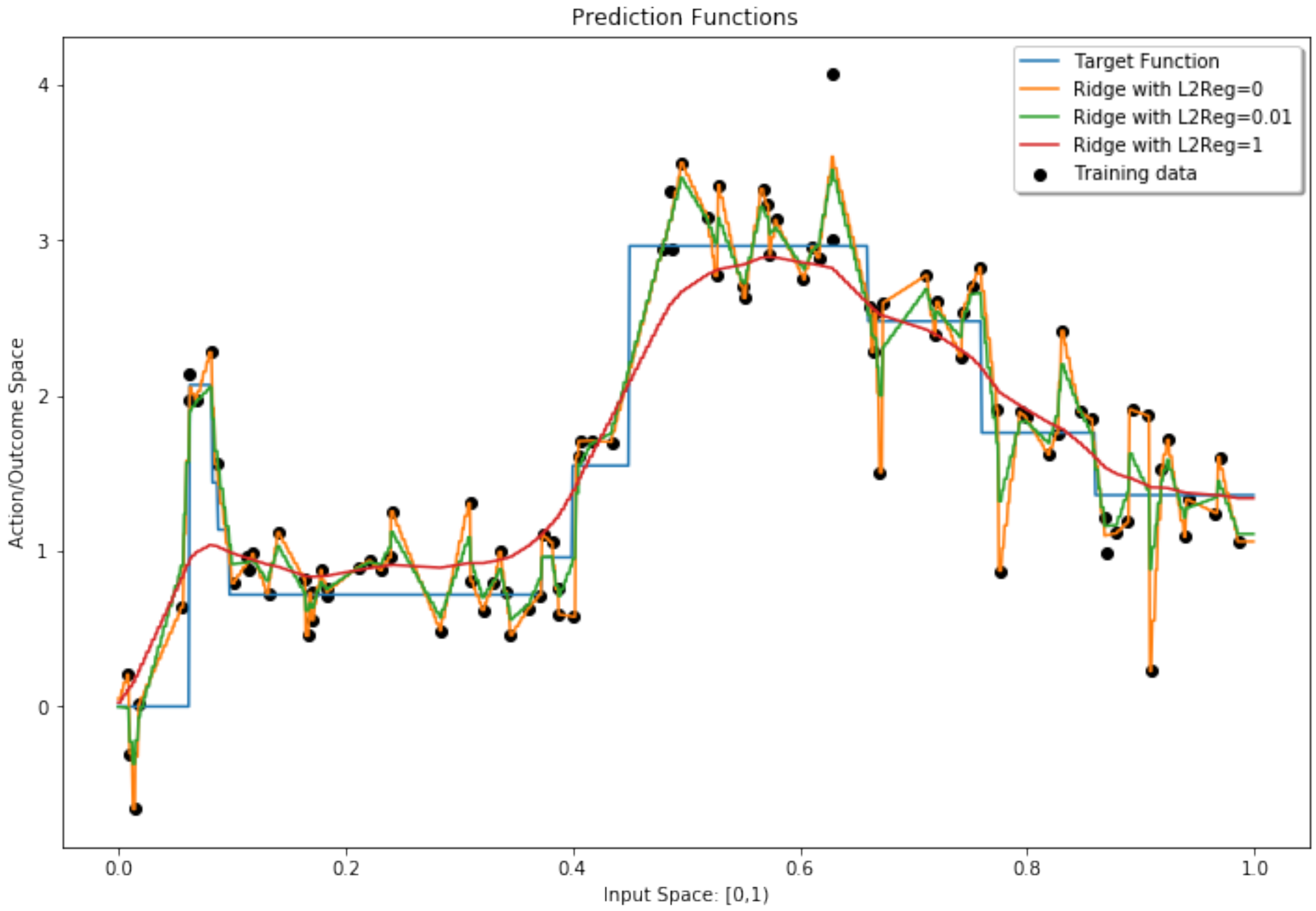
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))

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) })
```

In [12]:

```
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
```

In [13]:

```
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...

In [14]:

```
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
```

In [15]:

```
compare_parameter_vectors(pred_fns);
```

3.

Confusion Matrix

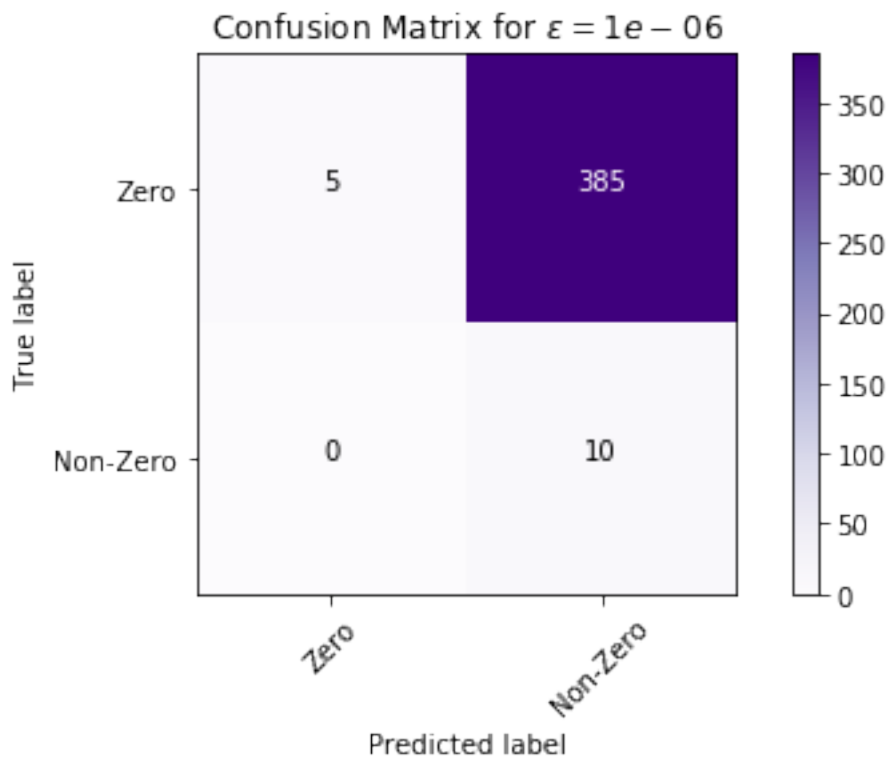
We can try to predict the features with corresponding weight zero. We will fix a threshold `eps` such that any value between `-eps` and `eps` will get counted as zero. We take the remaining features to have positive value. These predictions of can be compared to the weights for the target function.

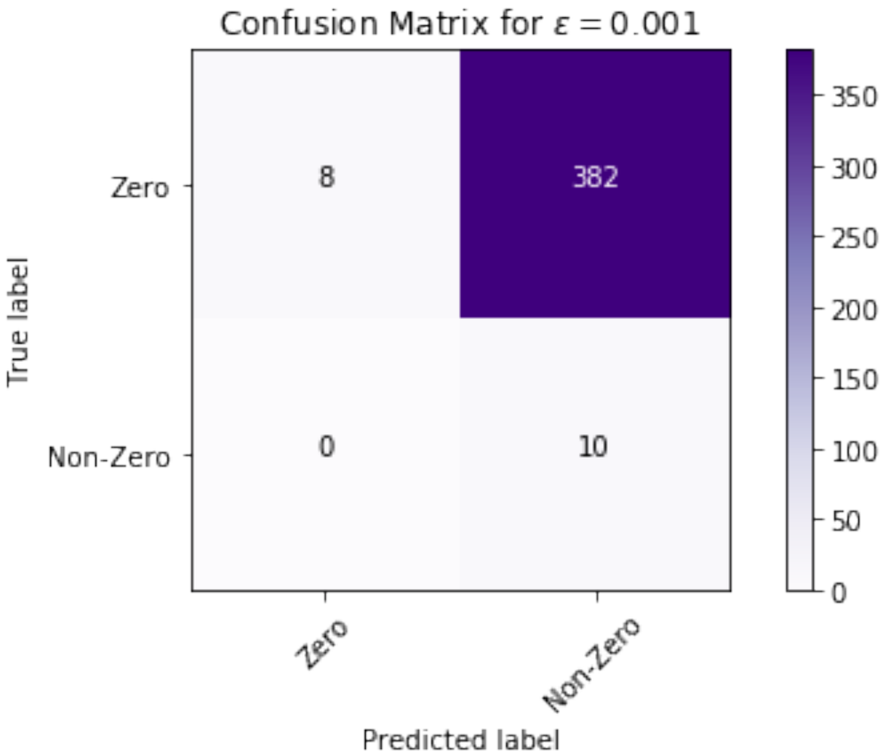
In [16]:

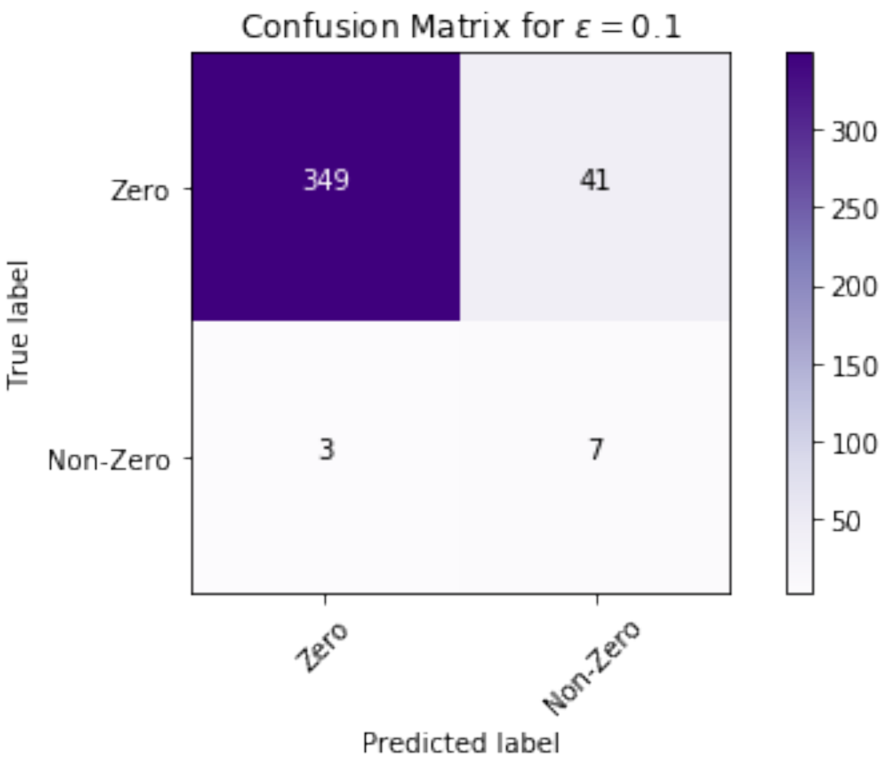
```
def plot_confusion_matrix(cm, title, classes):
        plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Purples)
        plt.title(title)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, rotation=45)
        plt.yticks(tick_marks, classes)

        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')
```

In [17]:

```
bin_coefs_true = [1 if i != 0 else 0 for i in coefs_true]  # your code goes here
eps_list = [10**i for i in [-6, -3, -1]] # your code goes here
for eps in eps_list:
    bin_coefs_estimated = [0 if w <= eps and w >= -eps else 1 for w in pred_fns[
2]["coefs"]] # 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"])
```




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


Confusion Matrix for $\varepsilon = 0.1$
True label
Zero
Non-Zero
349
41
3
7
Predicted label
Zero
Non-Zero
300
250
200
150
100
50

Lasso Regression

We will try to fit the dataset with a Lasso Regression model. The steps are

- Implement the Shooting Algorithm
  - allow for random or non-random order for the coordinates
  - allow for initial weights all zero or the corresponding solution to Ridge Regression
- Determine a class for the model supporting methods
  - fit
  - predict
  - score
- Tune hyperparameters
  - Search for hyperparameters through trial and error
  - Use upper bound on hyperparameter with warm start
- Plot the distributions of weight on the features
  - Does Lasso Regression give us sparsity
- Threshold the values to compare zero/non-zero against the weights of the target function
- Implement Projected Gradient Descent
  - Compare to Shooting Algorithm

1.

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.

In [18]:

```
def soft_threshold(a, delta):
    ####
    # your code goes here
    ####
    if a > 0:
        return max(abs(a)-delta, 0)
    else:
        return -max(abs(a)-delta, 0)

def compute_sum_sqr_loss(X, y, w):
    ####
    # your code goes here
    ####
    return sum(np.square(np.dot(X, w)-y))

def compute_lasso_objective(X, y, w, l1_reg=0):
    ####
    # your code goes here
    ####
    return compute_sum_sqr_loss(X, y, w) + l1_reg*sum(np.abs(w))

def get_ridge_solution(X, y, l2_reg):
    ####
    # your code goes here
    ####
    dim = X_train.shape[1]
    return np.linalg.inv(np.dot(X.T, X)+l2_reg*np.identity(dim)).dot(np.dot(X.T,
y))
```

Remember that we should avoid loops in the implementation because we need to run the algorithm repeatedly for hyperparameter tuning.

Please see Lecture 4 notes for derivation of shooting algorithm.

2.

In [19]:

```
def shooting_algorithm(X, y, w0=None, l1_reg = 1., max_num_epochs = 1000, min_ob
j_decrease=1e-8, random=False):
    if w0 is None:
        w = np.zeros(X.shape[1])
    else:
        w = np.copy(w0)
    d = X.shape[1] # dimension
    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*sum(np.square(X[:, j]))
            c = 2*np.dot(X[:, j], (y-np.dot(X, w)+w[j]*X[:, 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
```

Class for Lasso Regression

In [42]:

```
class LassoRegression(BaseEstimator, RegressorMixin):
    """ Lasso regression"""
    def __init__(self, l1_reg=1.0, randomized=False, coef_init=None):
        if l1_reg < 0:
            raise ValueError('Regularization penalty should be at least 0.')
        self.l1_reg = l1_reg
        self.randomized = randomized
        self.coef_init = coef_init

    def fit(self, X, y, max_epochs = 500):
        # convert y to 1-dim array, in case we're given a column vector
        y = y.reshape(-1)
        if self.coef_init is None:
            self.coef_init = get_ridge_solution(X,y, self.l1_reg)

        ####
        # your code goes here
        self.w_ = shooting_algorithm(X, y, w0=self.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)
```

We can compare to the `sklearn` implementation.

In [43]:

```
def compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=1):

    # 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=Fal
se)
    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=l1_reg)
    lasso_regression_estimator.fit(X_train, y_train)
    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((sk
learn_lasso_preds - lasso_regression_preds)**2)))
    # Let's compare differences parameter values
    print("Hoping this is very close to 0: {}".format(np.sum((our_coefs - sklear
n_lasso_coefs)**2)))
```

In [44]:

```
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=1.5)
```

```
Ran for 500 epochs. Lowest loss: 19.508025464837484
Hoping this is very close to 0 (predictions): 1.330785055280227e-07
Hoping this is very close to 0: 3.089820886577222
```

3.

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.

In [45]:

```
def do_grid_search_lasso(X_train, y_train, X_val, y_val):
    ####
    ## 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)

    my_params = np.unique(np.concatenate((10.**np.arange(-6, 1, 1), np.arange(1,
3, 0.3))))
    randomizedornot = [True, False]
    weight = [None, np.zeros(X_train_val.shape[1])]

    param_grid = [{"l1_reg": my_params, "randomized": randomizedornot, "coef_ini
t": weight}]

    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 the sign of the score because 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", "param_randomized", "param_coef_init", "mean
_test_score", "mean_train_score"]
    df["param_coef_init"] = [0 if row is not None else row for row in df["param_
coef_init"]]
    df_toshow = df[cols_to_keep].fillna("-")
    df_toshow = df_toshow.sort_values(by=["param_l1_reg"])

    return grid, df_toshow
```

In [46]:

```
grid, results = do_grid_search_lasso(X_train, y_train, X_val, y_val)
```

```
Ran for 1 epochs. Lowest loss: 0.6752223426188158
Ran for 1 epochs. Lowest loss: 0.6752223426098317
Ran for 1 epochs. Lowest loss: 0.6755576053468857
Ran for 1 epochs. Lowest loss: 0.6755576056041663
Ran for 10 epochs. Lowest loss: 0.6789098847197851
Ran for 6 epochs. Lowest loss: 0.6789099908354154
Ran for 144 epochs. Lowest loss: 0.7123821578168035
Ran for 231 epochs. Lowest loss: 0.712382229180633I
Ran for 500 epochs. Lowest loss: 1.042244440618114
```

```
Ran for 500 epochs. Lowest loss: 1.042244432218734
Ran for 500 epochs. Lowest loss: 3.9050396287000897
Ran for 500 epochs. Lowest loss: 3.9050571388463577
Ran for 500 epochs. Lowest loss: 16.19774479033197
Ran for 500 epochs. Lowest loss: 16.19776118280682
Ran for 500 epochs. Lowest loss: 18.208746486806547
Ran for 500 epochs. Lowest loss: 18.208756668329794
Ran for 500 epochs. Lowest loss: 20.146341751853328
Ran for 500 epochs. Lowest loss: 20.146346403885268
Ran for 500 epochs. Lowest loss: 22.016058952068086
Ran for 500 epochs. Lowest loss: 22.01605568657856
Ran for 500 epochs. Lowest loss: 23.818264633748598
Ran for 500 epochs. Lowest loss: 23.818258983271782
Ran for 500 epochs. Lowest loss: 25.559320975510417
Ran for 500 epochs. Lowest loss: 25.559312921992206
Ran for 500 epochs. Lowest loss: 27.2429397783529
Ran for 493 epochs. Lowest loss: 27.242934037959618
Ran for 500 epochs. Lowest loss: 0.6783573948805407
Ran for 500 epochs. Lowest loss: 0.6784018094847046
Ran for 500 epochs. Lowest loss: 0.6783356130813115
Ran for 500 epochs. Lowest loss: 0.6787367121492615
Ran for 500 epochs. Lowest loss: 0.682848043522198
Ran for 500 epochs. Lowest loss: 0.682085245983932
Ran for 500 epochs. Lowest loss: 0.7248247672415223
Ran for 500 epochs. Lowest loss: 0.7155234604645673
Ran for 500 epochs. Lowest loss: 1.0731118953392464
Ran for 500 epochs. Lowest loss: 1.0451149665654054
Ran for 500 epochs. Lowest loss: 4.011484805247497
Ran for 500 epochs. Lowest loss: 3.906961546256224
Ran for 500 epochs. Lowest loss: 16.197751573580774
Ran for 500 epochs. Lowest loss: 16.19788555548063
Ran for 500 epochs. Lowest loss: 18.20899147183324
Ran for 500 epochs. Lowest loss: 18.20887434982803
Ran for 500 epochs. Lowest loss: 20.146336186825195
Ran for 500 epochs. Lowest loss: 20.146455618766325
Ran for 500 epochs. Lowest loss: 22.016051737490272
Ran for 500 epochs. Lowest loss: 22.016155665008416
Ran for 500 epochs. Lowest loss: 23.8182698579489
Ran for 500 epochs. Lowest loss: 23.81833966769363
Ran for 500 epochs. Lowest loss: 25.559332288330104
Ran for 500 epochs. Lowest loss: 25.559429210383115
Ran for 500 epochs. Lowest loss: 27.242939829203102
Ran for 500 epochs. Lowest loss: 27.24305646805597
Ran for 500 epochs. Lowest loss: 90.85033171479994
```

In [47]:

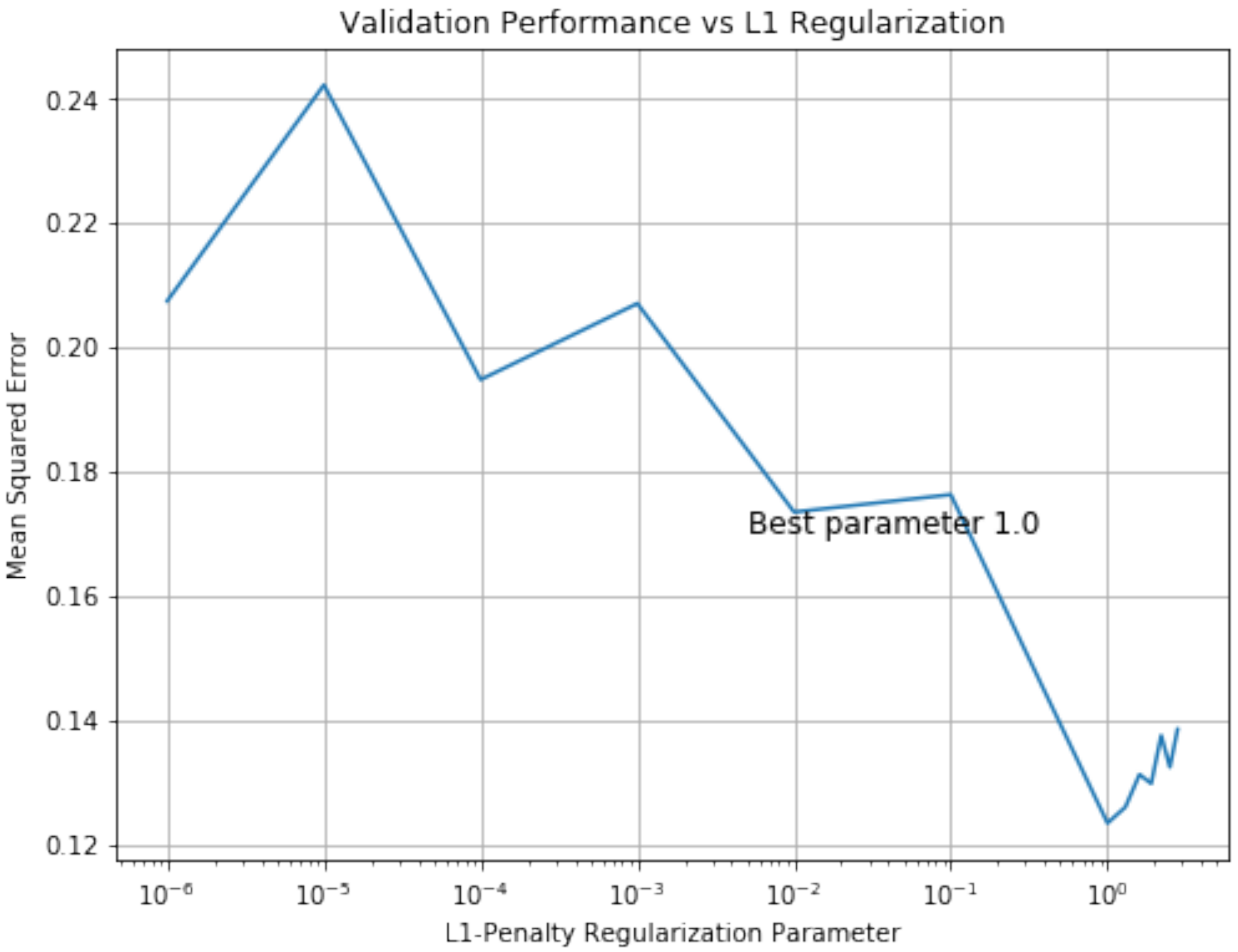
```
grid.best_params_
```

Out[47]:

```
{'coef_init': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0.]),
 'l1_reg': 1.0,
 'randomized': True}
```

In [52]:

```python
# 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.loc[(results["param_randomized"]==True) & (results["param_co
ef_init"] == 0), "param_l1_reg"], results.loc[(results["param_randomized"]==True
) & (results["param_coef_init"] == 0), "mean_test_score"])
####

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



Setting lambda at 1.3 with the randomized selection on features is my best configuration.

Sparsity between Lasso's Shooting Algorithms and Ridge Regression

In [55]:

```
lasso_regression_estimator = LassoRegression(l1_reg=1, randomized=True, coef_ini
t=np.zeros(X_train.shape[1]))
lasso_regression_estimator.fit(X_train, y_train)
lasso_coefs = lasso_regression_estimator.w_
```

```
Ran for 500 epochs. Lowest loss: 16.19776762107677
```

In [56]:

```
ridge_regression_estimator = RidgeRegression(l2reg=0.01)
ridge_regression_estimator.fit(X_train, y_train)
ridge_coefs = ridge_regression_estimator.w_
```

In [57]:

```
zeros_in_lasso = sum([i==0.0 for i in lasso_coefs])/len(lasso_coefs)
zeros_in_ridge = sum([i==0.0 for i in ridge_coefs])/len(ridge_coefs)
```

In [58]:

```
print("The percentage of zero weights on the lasso regression's shooting method:
{0}.\n The percentage of zero weights on the ridge regression: {1}".format(zeros
_in_lasso, zeros_in_ridge))
```

```
The percentage of zero weights on the lasso regression's shooting method: 0.7575.
 The percentage of zero weights on the ridge regression: 0.0
```

Lasso regression with the shooting algorithms has around 76% of the zero variables. On the other hand, ridge regression is not sparse at all.

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

In [59]:

```
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, 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)})
    ####
```

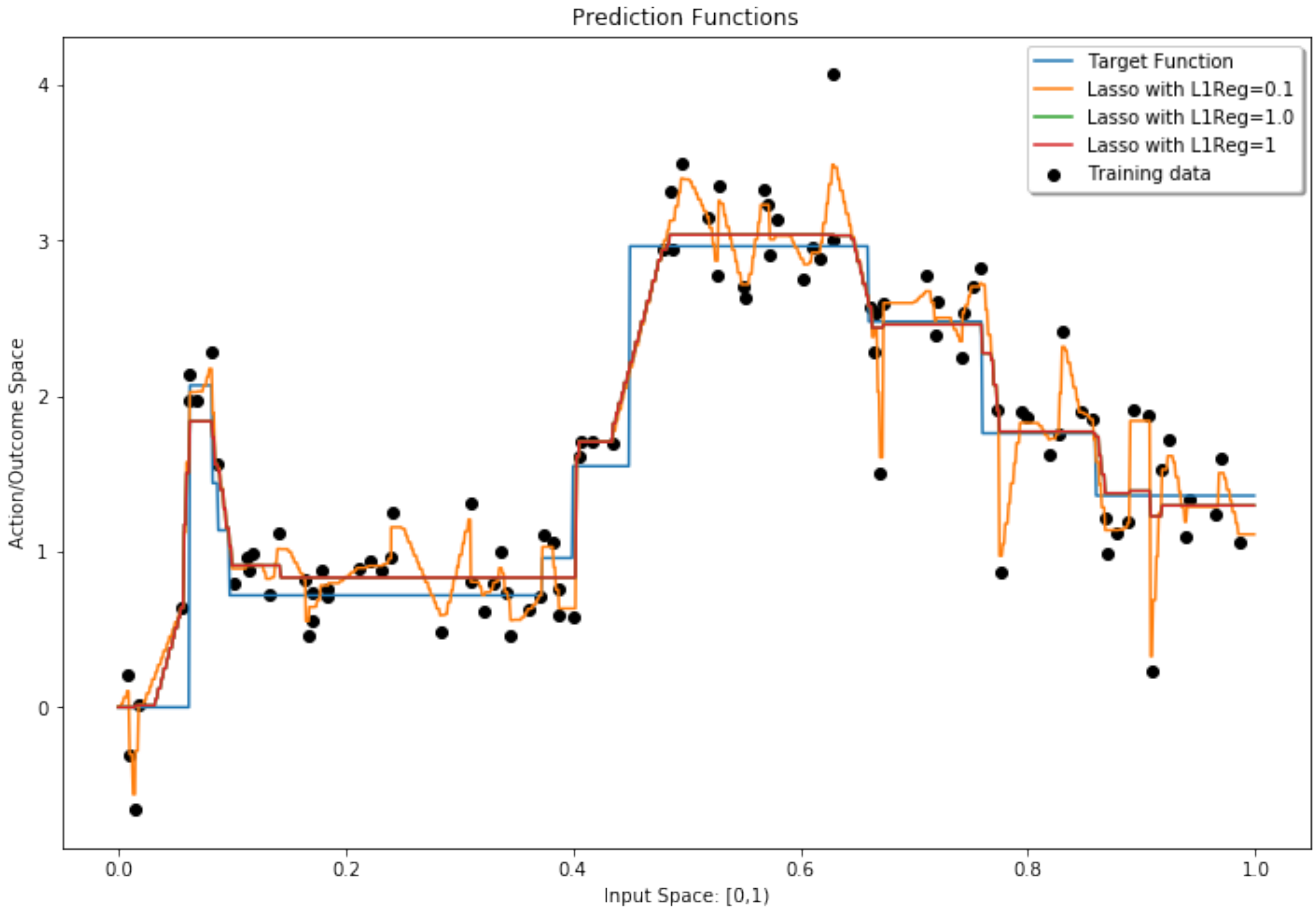
```
Ran for 500 epochs. Lowest loss: 3.9050571388463577
Ran for 500 epochs. Lowest loss: 16.19776118280682
Ran for 500 epochs. Lowest loss: 16.19776118280682
```

In [60]:

```
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
```

In [61]:

```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



Visualizing the Weights

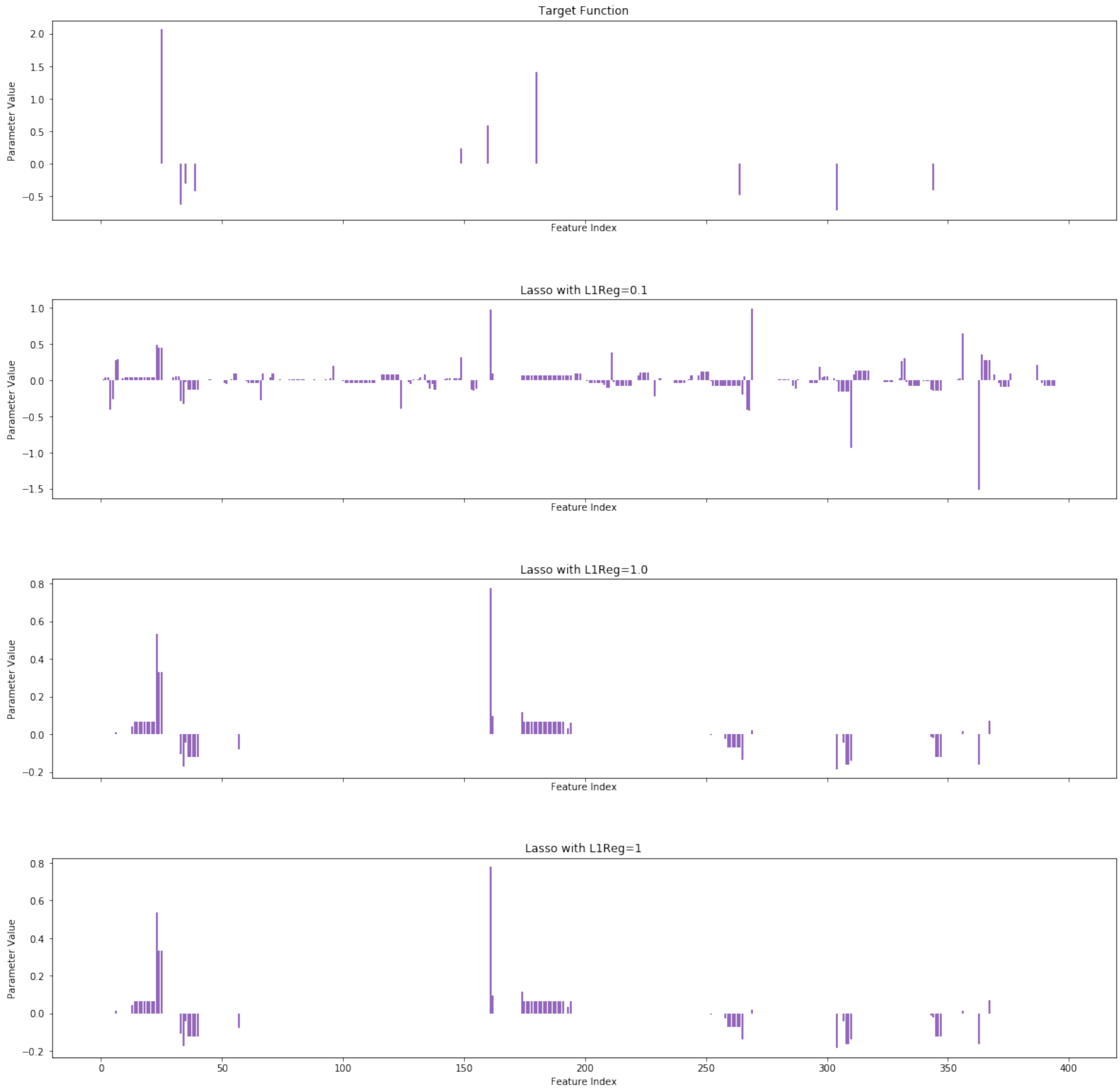
Using `pred_fns` let's try to see how sparse the weights are...

In [62]:

```
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
```

In [63]:

```
compare_parameter_vectors(pred_fns);
```



4.

Continuation Method

We compute the largest value of λ for which the weights can be nonzero.

In [64]:

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

Use homotopy method to compute regularization path for LassoRegression.

In [65]:

```
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.l1_reg = reg_val
            estimator.coef_init = coef_init
            estimator.fit(X, y, max_epochs = 1000)
            coef_init = estimator.w_
            ####

            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
```

In [66]:

```
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_nam
e="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
```

In [67]:

```
lasso_reg_path_estimator, reg_vals = do_grid_search_homotopy(X_train,
                                                              y_train,
                                                              X_val,
                                                              y_val,
                                                              None)
```

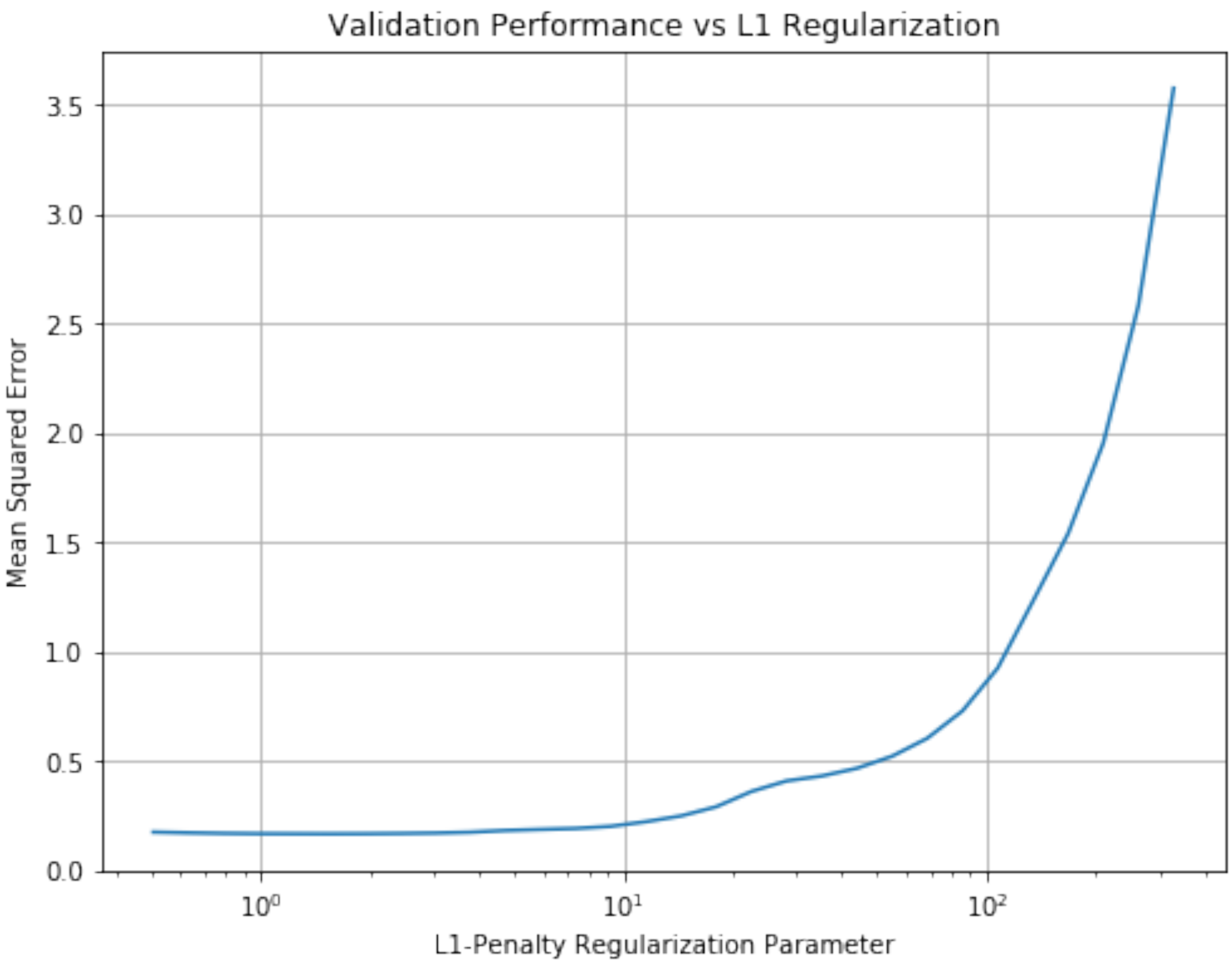
```
Ran for 2 epochs. Lowest loss: 359.6674002813195
Ran for 594 epochs. Lowest loss: 348.5210863339282
Ran for 578 epochs. Lowest loss: 323.53716482374966
Ran for 705 epochs. Lowest loss: 293.2292647236045
Ran for 671 epochs. Lowest loss: 262.23637332060844
Ran for 136 epochs. Lowest loss: 231.30364679470637
Ran for 132 epochs. Lowest loss: 202.01748534382182
Ran for 127 epochs. Lowest loss: 175.6829687118384
Ran for 303 epochs. Lowest loss: 152.73952217465128
Ran for 319 epochs. Lowest loss: 133.12872392009874
Ran for 300 epochs. Lowest loss: 116.62091752850151
Ran for 280 epochs. Lowest loss: 102.89040503458665
Ran for 319 epochs. Lowest loss: 91.40127958050608
Ran for 351 epochs. Lowest loss: 80.59513427785579
Ran for 332 epochs. Lowest loss: 70.65131127999413
Ran for 313 epochs. Lowest loss: 61.838968974776485
Ran for 334 epochs. Lowest loss: 54.08107969936126
Ran for 320 epochs. Lowest loss: 47.3266690315174
Ran for 301 epochs. Lowest loss: 41.56428576679454
Ran for 614 epochs. Lowest loss: 36.69712945939424
Ran for 723 epochs. Lowest loss: 32.323174124306
Ran for 700 epochs. Lowest loss: 28.43476228353057
Ran for 669 epochs. Lowest loss: 25.07153045272605
Ran for 643 epochs. Lowest loss: 22.21109135077282
Ran for 657 epochs. Lowest loss: 19.799697382372194
Ran for 630 epochs. Lowest loss: 17.789509032870523
Ran for 603 epochs. Lowest loss: 16.121413497283264
Ran for 604 epochs. Lowest loss: 14.563979468681875
Ran for 575 epochs. Lowest loss: 12.99312680954627
Ran for 543 epochs. Lowest loss: 11.487542271474759
```

In [68]:

```
# Plot validation performance vs regularization parameter
path_result = lasso_reg_path_estimator.score(X_val, y_val)
x_value, y_value = [i["reg_val"] for i in path_result], [i["score"] for i in pat
h_result]
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")

ax.semilogx(x_value, y_value)


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

Out[68]:

```
[<matplotlib.lines.Line2D at 0x7fa8a1501b70>]
```



5.

In [69]:

```
from sklearn.preprocessing import StandardScaler
```

In [70]:

```
# standardize y_train
std_scaler = StandardScaler()
std_scaler.fit(y_train.reshape(-1, 1))
y_std_train = std_scaler.transform(y_train.reshape(-1, 1)).flatten()
y_std_val = std_scaler.transform(y_val.reshape(-1, 1)).flatten()
```

In [71]:

```
print("Original Max Lambda: {}; Centered Max Lambda: {}".format(get_lambda_max_n
o_bias(X_train, y_std_train), get_lambda_max_no_bias(X_train, y_train)))
```

```
Original Max Lambda: 68.48117686312149; Centered Max Lambda: 327.28283232952117
```

5-1. Lasso

In [77]:

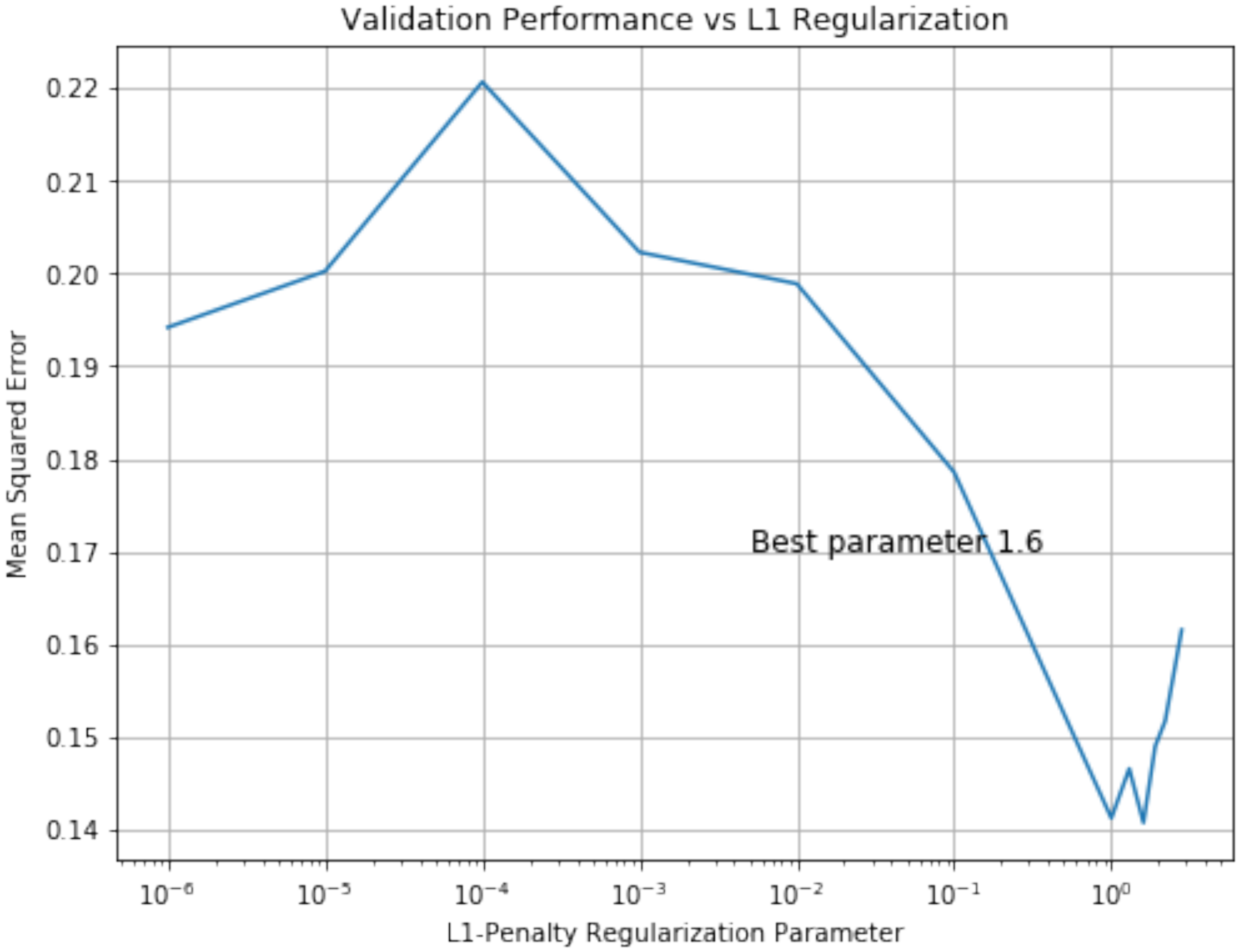
```
grid_lasso, results_lasso = do_grid_search_lasso(X_train, y_std_train, X_val, y_
std_val)
```

```
Ran for 1 epochs. Lowest loss: 0.7156248815786076
Ran for 1 epochs. Lowest loss: 0.7156248815810357
Ran for 1 epochs. Lowest loss: 0.715981306539921
Ran for 1 epochs. Lowest loss: 0.7159813067575387
Ran for 11 epochs. Lowest loss: 0.7195452070850137
Ran for 7 epochs. Lowest loss: 0.7195453126055195
Ran for 142 epochs. Lowest loss: 0.755134717510434
Ran for 229 epochs. Lowest loss: 0.7551347740687375
Ran for 500 epochs. Lowest loss: 1.1062552840244666
Ran for 500 epochs. Lowest loss: 1.1062552715595106
Ran for 500 epochs. Lowest loss: 4.1890078571567315
Ran for 500 epochs. Lowest loss: 4.188974615138752
Ran for 500 epochs. Lowest loss: 18.58373922565912
Ran for 500 epochs. Lowest loss: 18.583716610662762
Ran for 500 epochs. Lowest loss: 21.13827339438507
Ran for 500 epochs. Lowest loss: 21.138254066001714
Ran for 500 epochs. Lowest loss: 23.5504761317787
Ran for 500 epochs. Lowest loss: 23.550440359262872
Ran for 500 epochs. Lowest loss: 25.84175727258252
Ran for 500 epochs. Lowest loss: 25.841749742111745
Ran for 500 epochs. Lowest loss: 28.02024300014
Ran for 465 epochs. Lowest loss: 28.020229969972476
Ran for 500 epochs. Lowest loss: 30.091037644678053
Ran for 490 epochs. Lowest loss: 30.09102689857479
Ran for 500 epochs. Lowest loss: 32.05879664216745
Ran for 467 epochs. Lowest loss: 32.05878875733673
Ran for 500 epochs. Lowest loss: 0.7189575962226185
Ran for 500 epochs. Lowest loss: 0.7189945952685527
Ran for 500 epochs. Lowest loss: 0.718764726933061
Ran for 500 epochs. Lowest loss: 0.7193506487380974
Ran for 500 epochs. Lowest loss: 0.72307322954527
```

```
Ran for 500 epochs. Lowest loss: 0.722910707813501
Ran for 500 epochs. Lowest loss: 0.7618964400930648
Ran for 500 epochs. Lowest loss: 0.7584650218666503
Ran for 500 epochs. Lowest loss: 1.1169319829387216
Ran for 500 epochs. Lowest loss: 1.1093058809840413
Ran for 500 epochs. Lowest loss: 4.189599017773912
Ran for 500 epochs. Lowest loss: 4.19098605962925
Ran for 500 epochs. Lowest loss: 18.58369641790818
Ran for 500 epochs. Lowest loss: 18.584149605766846
Ran for 500 epochs. Lowest loss: 21.13830370107243
Ran for 500 epochs. Lowest loss: 21.138519488508777
Ran for 500 epochs. Lowest loss: 23.5504485995678
Ran for 500 epochs. Lowest loss: 23.55064583891195
Ran for 500 epochs. Lowest loss: 25.84175614067184
Ran for 500 epochs. Lowest loss: 25.841772394728217
Ran for 500 epochs. Lowest loss: 28.02023150084385
Ran for 500 epochs. Lowest loss: 28.02023750144713
Ran for 500 epochs. Lowest loss: 30.09104107419929
Ran for 500 epochs. Lowest loss: 30.091051756152062
Ran for 500 epochs. Lowest loss: 32.058800937059594
Ran for 500 epochs. Lowest loss: 32.05881636320348
Ran for 500 epochs. Lowest loss: 105.9581029678573
```

In [78]:

```python
# 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_lasso.loc[(results_lasso["param_randomized"]==True) & (results_lasso["param_coef_init"] == 0), "param_l1_reg"], results_lasso.loc[(results_lasso["param_randomized"]==True) & (results_lasso["param_coef_init"] == 0), "mean_test_score"])
####

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


Validation Performance vs L1 Regularization
Mean Squared Error
0.22
0.21
0.20
0.19
0.18
0.17
0.16
0.15
0.14
Best parameter 1.6
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
L1-Penalty Regularization Parameter

5-2. Ridge

In [74]:

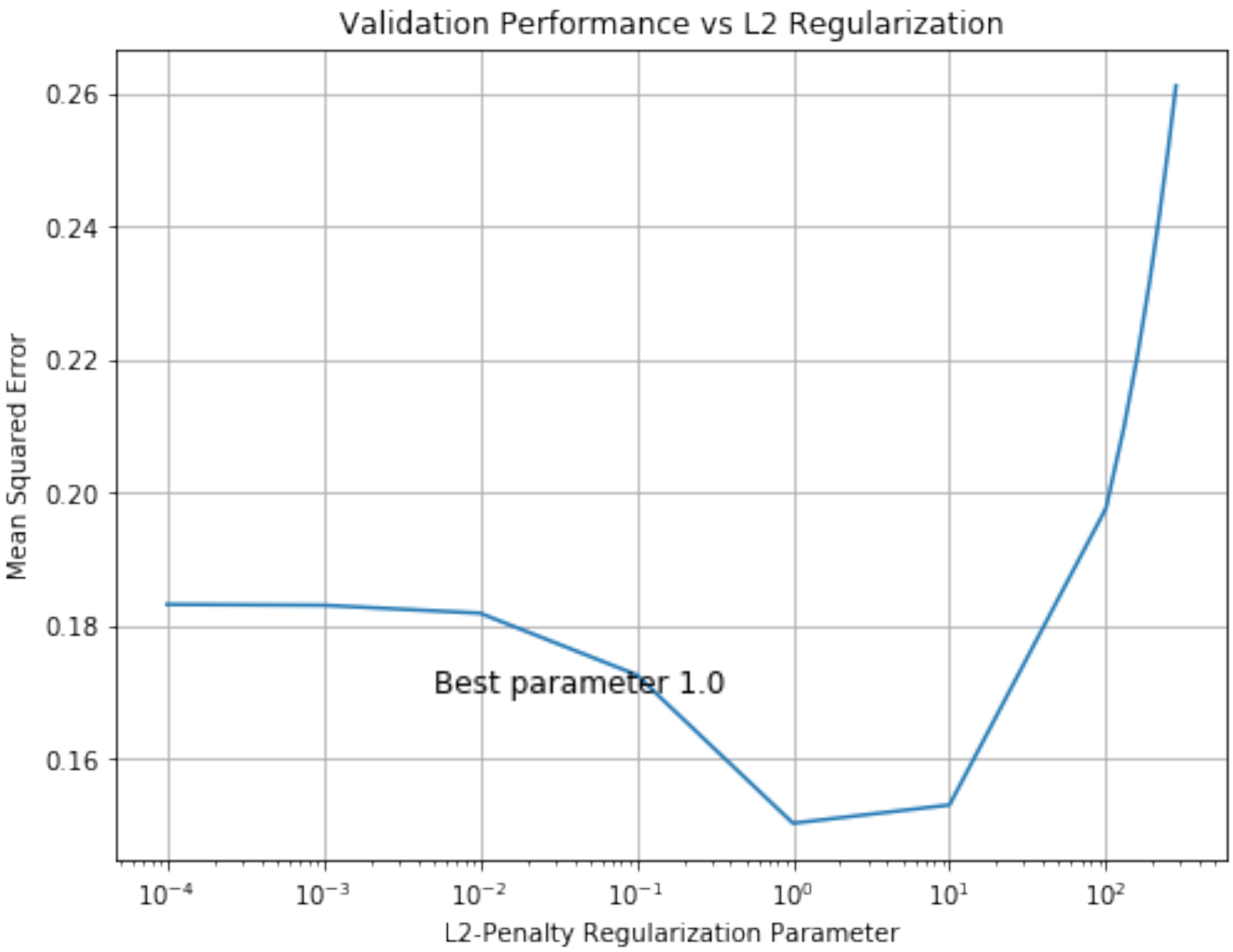
```
default_params = np.unique(np.concatenate((10.**np.arange(-6,1,1), np.arange(1,3
,.3))))

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 validati
on

        param_grid = [{'alpha': X_train.shape[0]*params}]

        ridge_regression_estimator = Ridge()
        grid = GridSearchCV(ridge_regression_estimator,
                                                 param_grid,
                                                 return_train_score=True,
                                                 cv = PredefinedSplit(test_fold=v
al_fold),
                                                 refit = True,
                                                 scoring = make_scorer(mean_squar
ed_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_alpha", "mean_test_score","mean_train_score"]
        df_toshow = df[cols_to_keep].fillna('-')
        df_toshow = df_toshow.sort_values(by=["param_alpha"])
        return grid, df_toshow
```

In [75]:

```
grid_ridge, results_ridge = do_grid_search_ridge(X_train, y_std_train, X_val, y_
std_val)
```

In [76]:

```
# 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")

####
## your code goes here
ax.semilogx(results_ridge["param_alpha"], results_ridge["mean_test_score"])
####

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



After I did standard scaling on y, the validation MSE of both ridge regression and lasso regression became higher with their respective best parameters.

Projected SGD

1.

In [79]:

```
def projection_SGD_split(X, y, theta_positive_0, theta_negative_0, lambda_reg =
1.0, alpha = 0.1, num_iter = 1000):
    # alpha: fixed step size of SGD
    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
            X_i, y_i = X[i], y[i]
            P_i = np.dot(X_i, theta)-y_i
            # theta_positive
            g_pos_deri = 2*np.dot(X_i.T, P_i)+lambda_reg/m
            theta_positive = theta_positive-alpha*g_pos_deri
            # theta_negative
            g_neg_deri = -2*np.dot(X_i.T, P_i)+lambda_reg/m
            theta_negative = theta_negative-alpha*g_neg_deri
            # negative to zero
            #theta_positive = np.clip(theta_positive, 0, None)
            #theta_negative = np.clip(theta_negative, 0, None)
            theta_positive = np.array([i if i > 0 else 0 for i in theta_positive
])
            theta_negative = np.array([i if i > 0 else 0 for i in theta_negative
])
            # theta
            theta = theta_positive-theta_negative
            ####
        loss = compute_sum_sqr_loss(X, y, theta)
        loss_change = np.abs(loss - loss_old)
        times +=1

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

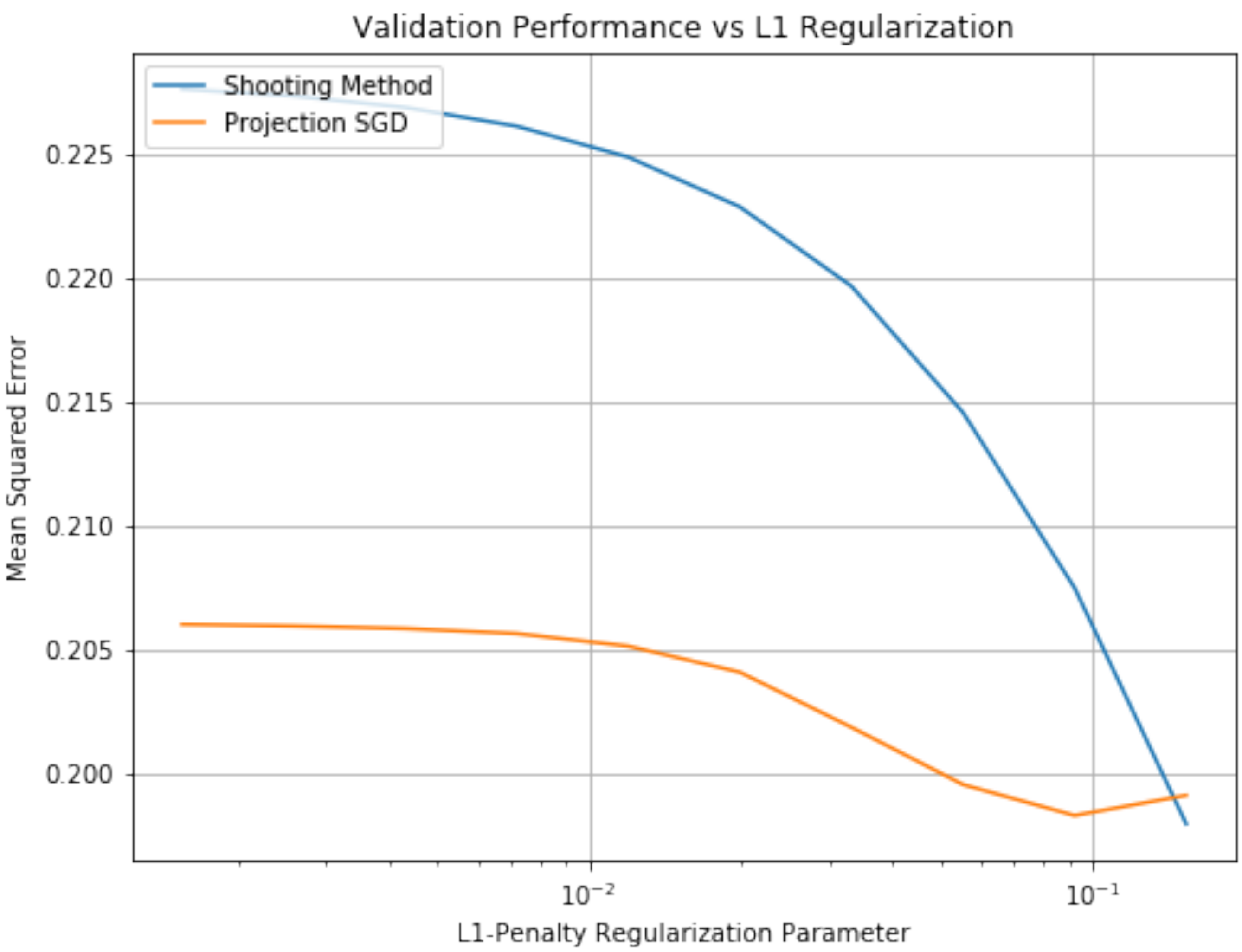
In [80]:

```
x_training, y_training, x_validation, y_validation, target_fn, coefs_true, featu
rize = 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 = []
n_vali = X_validation.shape[0]
for lambda_value in reg_vals:
    ####
    ## your code goes here
    sgd_theta = projection_SGD_split(X_training, y_training, 0, 0, lambda_reg=la
mbda_value, alpha=0.001)
    loss_SGD_list.append(compute_sum_sqr_loss(X_validation, y_validation, sgd_th
eta)/n_vali)

    shooting_theta = shooting_algorithm(X_training, y_training, w0=None, l1_reg
= lambda_value)
    loss_shooting.append(compute_sum_sqr_loss(X_validation, y_validation, shooti
ng_theta)/n_vali)
    ####
```

```
(SGD) Ran for 1000 epochs. Loss:6.548281868701245 Lambda: 0.1538834734708454
Ran for 1000 epochs. Lowest loss: 5.309363932046909
(SGD) Ran for 1000 epochs. Loss:6.003824713811802 Lambda: 0.09233008408250726
Ran for 1000 epochs. Lowest loss: 3.6886309490778286
(SGD) Ran for 1000 epochs. Loss:5.945923717391214 Lambda: 0.05539805044950434
Ran for 1000 epochs. Lowest loss: 2.5782374612036367
(SGD) Ran for 1000 epochs. Loss:6.079914698778654 Lambda: 0.033238830269702604
Ran for 1000 epochs. Lowest loss: 1.8542182080338
(SGD) Ran for 1000 epochs. Loss:6.220528894971988 Lambda: 0.019943298161821565
Ran for 1000 epochs. Lowest loss: 1.3966441974431747
(SGD) Ran for 1000 epochs. Loss:6.282423532254842 Lambda: 0.011965978897092939
Ran for 1000 epochs. Lowest loss: 1.1131523250340585
(SGD) Ran for 1000 epochs. Loss:6.315135363148399 Lambda: 0.007179587338255763
Ran for 1000 epochs. Lowest loss: 0.9398140346202832
(SGD) Ran for 1000 epochs. Loss:6.323934602735171 Lambda: 0.004307752402953458
Ran for 1000 epochs. Lowest loss: 0.834639570135258
(SGD) Ran for 1000 epochs. Loss:6.327205264918045 Lambda: 0.002584651441772074
Ran for 1000 epochs. Lowest loss: 0.7711064552425747
(SGD) Ran for 1000 epochs. Loss:6.328832691674653 Lambda: 0.0015507908650632444
Ran for 1000 epochs. Lowest loss: 0.7328296996266538
```

In [81]:

```
# 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();
```



2.

I will choose the best lambda found above for the projection method. For the shooting method, I set 1 at lambda which is the one I found by grid search, randomly select the features in each step, and initialize all of the weights as zero.

2-1. Projection

In [82]:

```
# Report the best
theta_positive_ini, theta_negative_ini = 0, 0
lambda_best_SGD = reg_vals[np.argmin(loss_SGD_list)]
theta_lasso_SGD_best = projection_SGD_split(X_training, y_training, theta_positi
ve_ini, theta_negative_ini, lambda_reg=lambda_best_SGD, alpha = 0.001)
print('Best lambda for SGD is {0} with loss {1}'.format(lambda_best_SGD, np.min(
loss_SGD_list)))
```

```
(SGD) Ran for 1000 epochs. Loss:6.003824713811802 Lambda: 0.09233008408250726
Best lambda for SGD is 0.09233008408250726 with loss 0.19828465332143744
```

2-2. Shooting Algorithms

In [86]:

```
lasso_regression_estimator = LassoRegression(l1_reg=1, randomized=True, coef_ini
t=np.zeros(X_train.shape[1]))
lasso_regression_estimator.fit(X_train, y_train)
our_coefs = lasso_regression_estimator.w_
```

```
Ran for 500 epochs. Lowest loss: 16.197739158834345
```

In [87]:

```
zeros_in_shooting = sum([i==0.0 for i in our_coefs])/len(our_coefs)
zeros_in_projection = sum([i==0.0 for i in theta_lasso_SGD_best])/len(theta_lass
o_SGD_best)
```

In [88]:

```
print("The percentage of zero weights on the shooting method: {0}.\n The percent
age of zero weights on the projection method: {1}".format(zeros_in_shooting, zer
os_in_projection))
```

```
The percentage of zero weights on the shooting method: 0.8.
 The percentage of zero weights on the projection method: 0.0125
```

The shooting algorithms will lead to more sparse weights in my lasso regression estimator.

In [ ]:

```

```