Ridge Regression

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

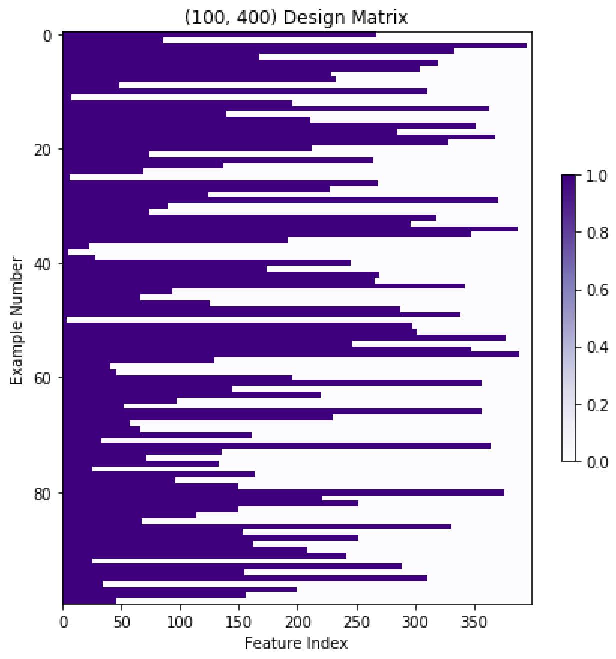
- Determine a class for the model supporting methods
  - fit
  - predict
  - score
- Search for hyperparameters through trial and error
  - evaluate the average training and validating error for each hyperparameter
- Plot the distributions of weight on the features
  - Does Ridge Regression give us sparsity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

In [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
from sklearn.linear_model import 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);
```



Class for Ridge Regression

```
In [7]: 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)
```

We can compare to the `sklearn` implementation.

```
In [8]: 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)))
```

```
In [9]: compare_our_ridge_with_sklearn(X_train, y_train, l2_reg=1.5)
```

```
Hoping this is very close to 0:4.690139613337759e-11
```

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 [11]:
```
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
```

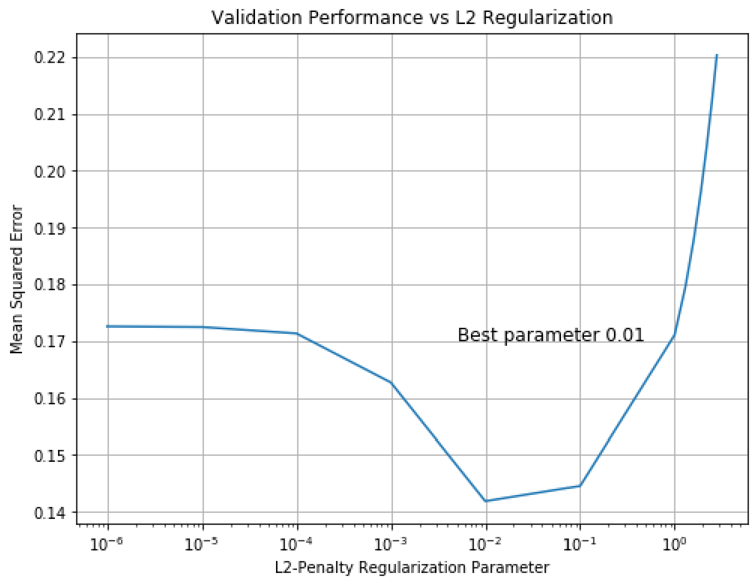
In [12]:
```
grid, results = do_grid_search_ridge(X_train, y_train, X_val, y_val)
```

In [13]:
```
results
```

Out[13]:

	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 [14]:
```python
# 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);
```



Comparing to the Target Function

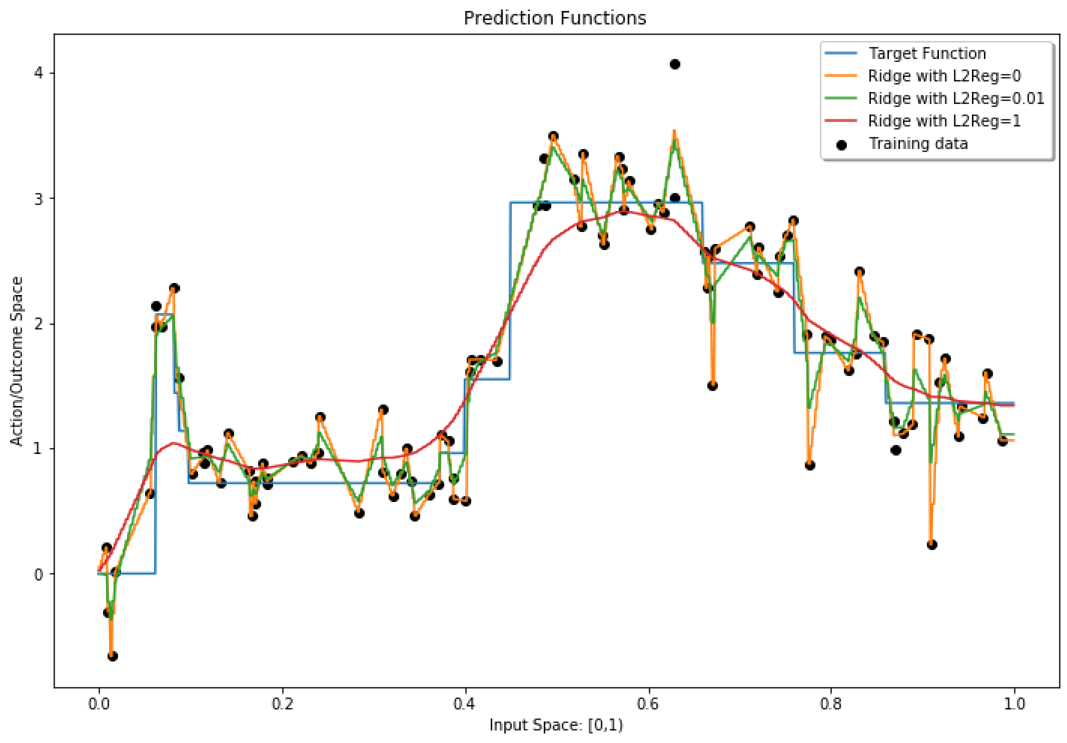
Let's plot prediction functions and compare coefficients for several fits and the target function.

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 [15]:
```python
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 [16]:
```python
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 [17]: plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```

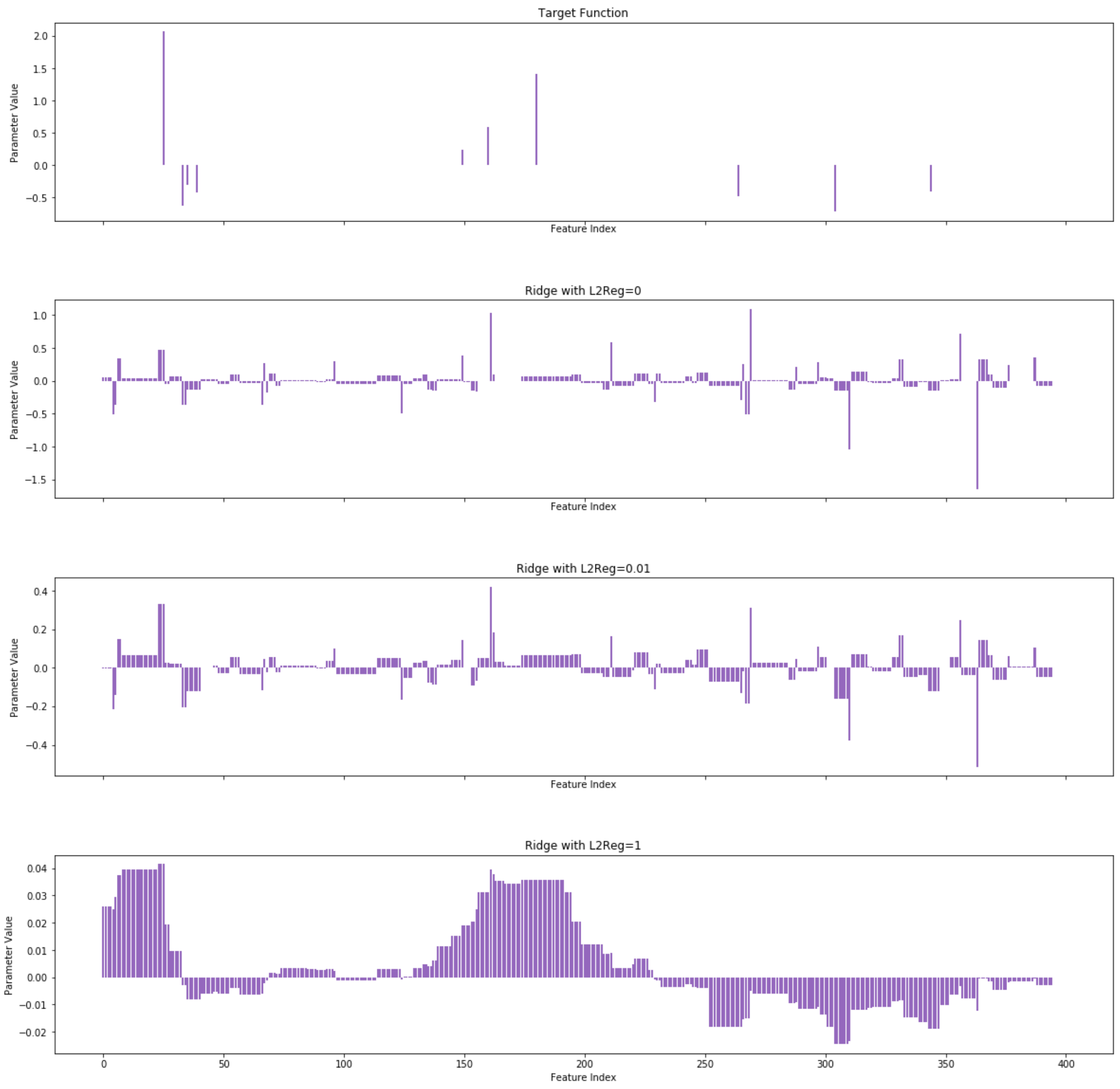


Visualizing the Weights

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

```
In [18]: 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 [19]:
```
compare_parameter_vectors(pred_fns);
```



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 [26]:
```
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')
```

Ridge Regression 1.

In [119]:
```
params = np.unique(np.concatenate((10.**np.arange(-6,1,0.2), np.arange(1,2,.5))))
```

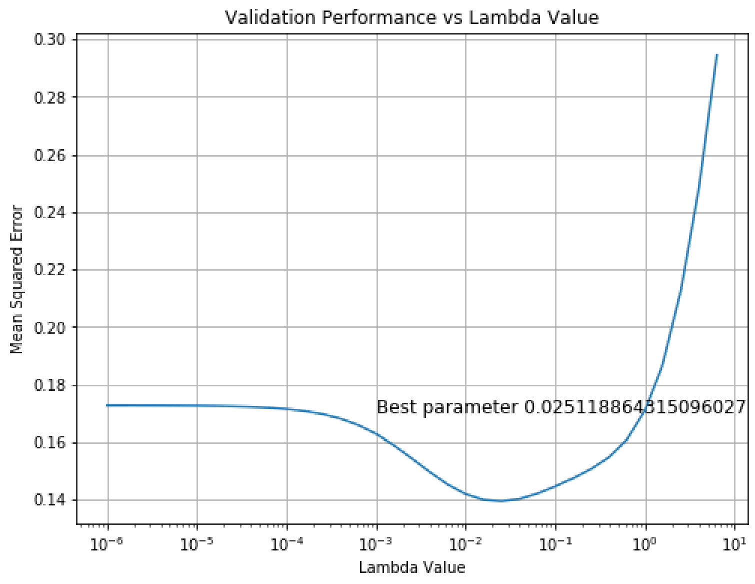
In [120]:
```
grid, results = do_grid_search_ridge(X_train, y_train, X_val, y_val,params = params)
```

```
In [121]: results
```

Out[121]:

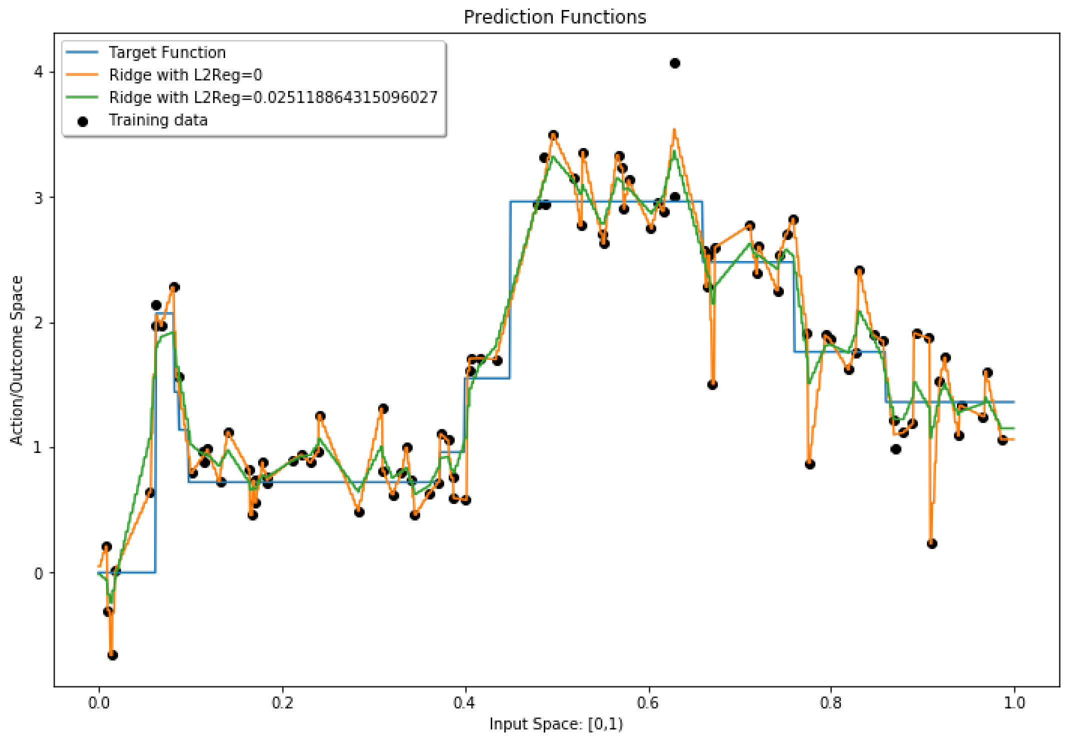
	param_l2reg	mean_test_score	mean_train_score
0	0.000001	0.172579	0.006752
1	0.000002	0.172571	0.006752
2	0.000003	0.172560	0.006752
3	0.000004	0.172541	0.006752
4	0.000006	0.172511	0.006752
5	0.000010	0.172464	0.006752
6	0.000016	0.172390	0.006752
7	0.000025	0.172272	0.006753
8	0.000040	0.172087	0.006755
9	0.000063	0.171798	0.006761
10	0.000100	0.171345	0.006774
11	0.000158	0.170649	0.006805
12	0.000251	0.169590	0.006881
13	0.000398	0.168023	0.007056
14	0.000631	0.165776	0.007450
15	0.001000	0.162705	0.008285
16	0.001585	0.158797	0.009925
17	0.002512	0.154260	0.012853
18	0.003981	0.149546	0.017533
19	0.006310	0.145225	0.024202
20	0.010000	0.141887	0.032767
21	0.015849	0.139916	0.042934
22	0.025119	0.139392	0.054400
23	0.039811	0.140208	0.066953
24	0.063096	0.142061	0.080470
25	0.100000	0.144566	0.094953
26	0.158489	0.147453	0.110587
27	0.251189	0.150713	0.127761
28	0.398107	0.154825	0.147152
29	0.630957	0.160968	0.169897
30	1.000000	0.171068	0.197694
31	1.000000	0.171068	0.197694
32	1.500000	0.185175	0.228017
33	1.584893	0.187568	0.232640
34	2.511886	0.212841	0.276619
35	3.981072	0.248408	0.330405
36	6.309573	0.294445	0.393206

In [122]:
```
# Plot validation performance vs regularization parameter
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs Lambda Value")
ax.set_xlabel("Lambda Value")
ax.set_ylabel("Mean Squared Error")
ax.semilogx(results["param_l2reg"], results["mean_test_score"])
ax.text(0.001,0.17,"Best parameter {0}".format(grid.best_params_['l2reg']), fontsize = 12);
```

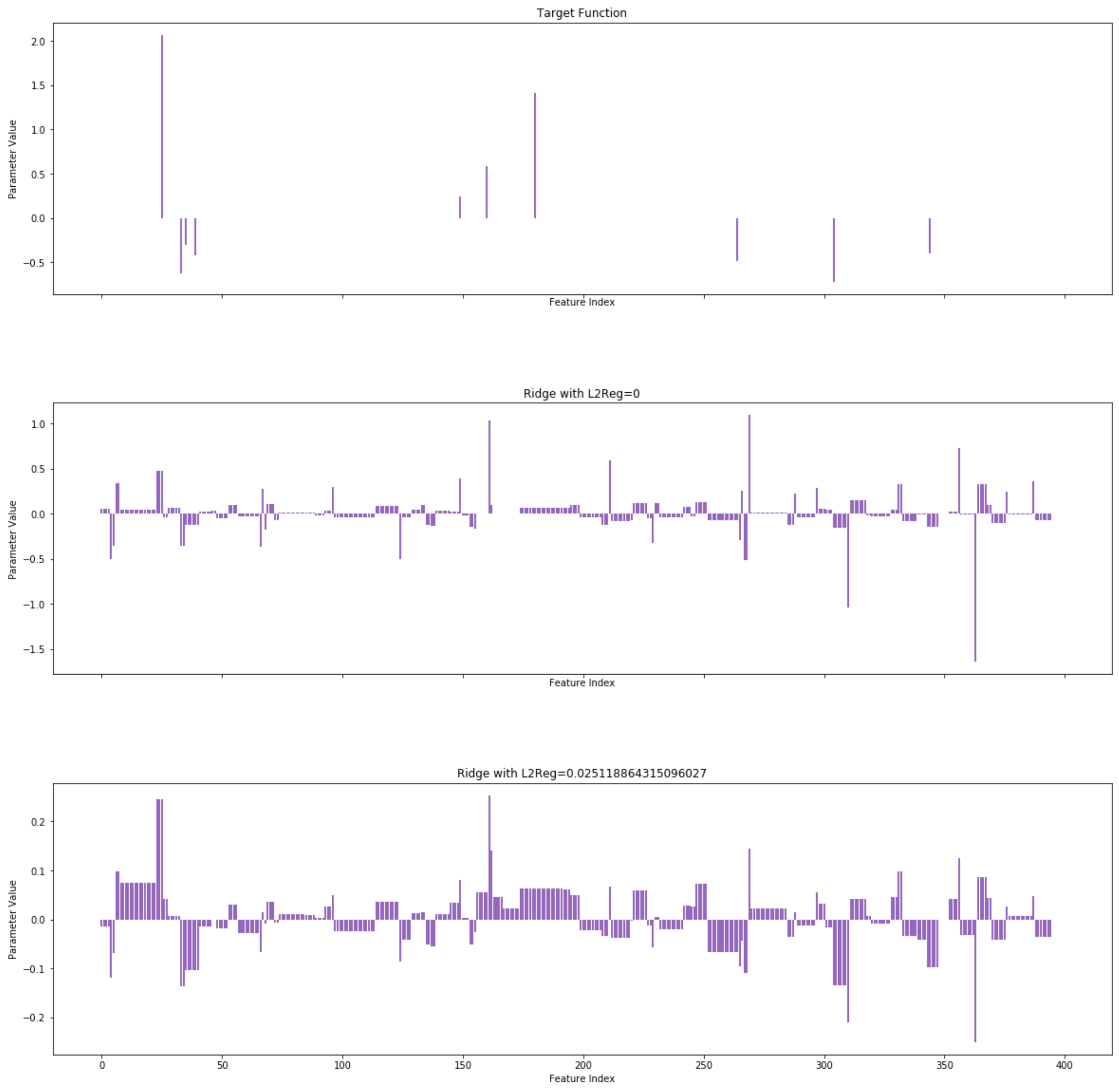


Ridge Regression 2.

In [123]:
```
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']]
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) })
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



In [128]:
```
compare_parameter_vectors(pred_fns);
```



In [135]:
```
for i in range(len(pred_fns)):
    min_w = min(pred_fns[i]['coefs'])
    max_w = max(pred_fns[i]['coefs'])
    print("The most positive weight for",pred_fns[i]['name'],"is",max_w,"at feature index",list(pred_fns[i]['c
oefs']).index(max_w))
    print("The most negative weight for",pred_fns[i]['name'],"is",min_w,"at feature index",list(pred_fns[i]['c
oefs']).index(min_w))
```

```
The most positive weight for Target Function is 2.06957209208771 at feature index 25
The most negative weight for Target Function is -0.7178177644454445 at feature index 304
The most positive weight for Ridge with L2Reg=0 is 1.0935155997863646 at feature index 269
The most negative weight for Ridge with L2Reg=0 is -1.6442202830317005 at feature index 363
The most positive weight for Ridge with L2Reg=0.025118864315096027 is 0.25370489768508436 at feature index 16
1
The most negative weight for Ridge with L2Reg=0.025118864315096027 is -0.2503710373587774 at feature index 36
3
```

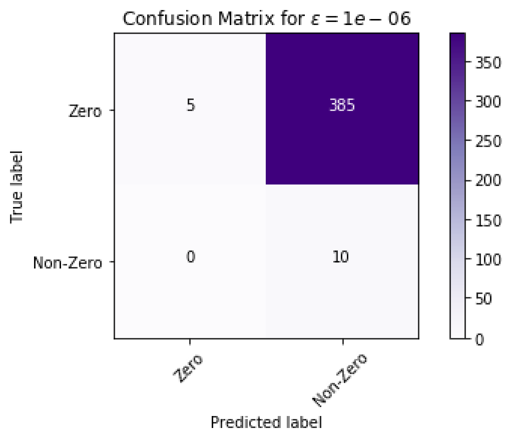
Pattern: We observe that the weights of unregularized predicton function and the prediction function with the optimal lambda value have very similar patterns, i.e. the indexes of both high and low weight value spikes for these two functions are very close or even exactly same. For all of the bars in the target function, we can find corresponding spikes in both prediction functions at approximately the same index position. Although the prediction is very sparse, both of these two prediction functions are not sparse. The unregularized prediction function is relatively more sparse than the optimal prediction function.

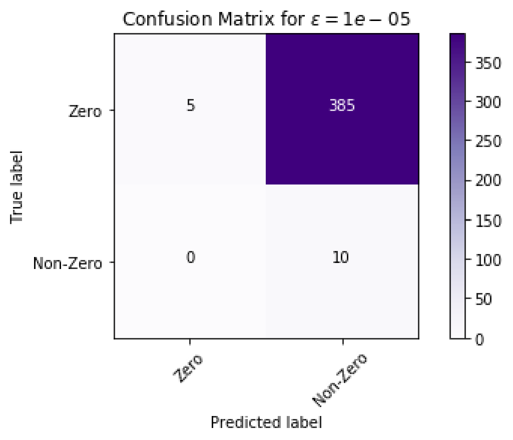
Scale: We observe that the weight ranges for the target function, unregularized predicton function and the prediction function with the optimal lambda value are (-0.72,2.07), (-1.64,1.09), (-0.25,0.25). It is obvious that the prediction function with the chosen lambda value has the lowest maximum weight, the highest minimum weight and the most balanced range. Compared with the target function, both of these two prediction functions have lower maximum weight. However, the unregularized prediction function has the lowest minimum weight among these three functions.

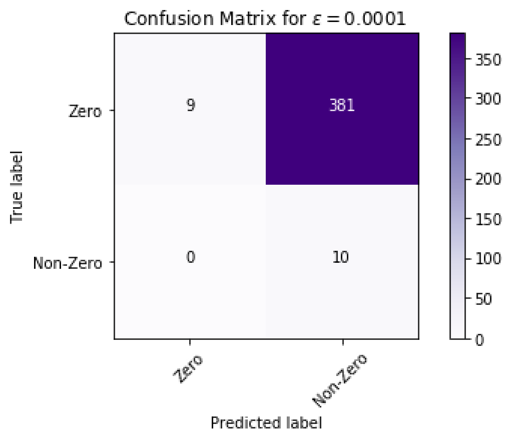
Coefficients with the most weight: please refer to the print out above.

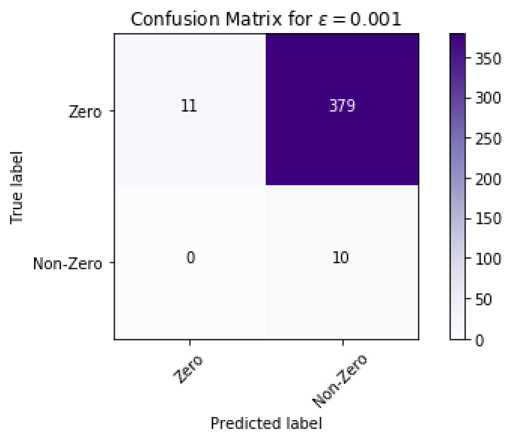
Ridge Regression 3.

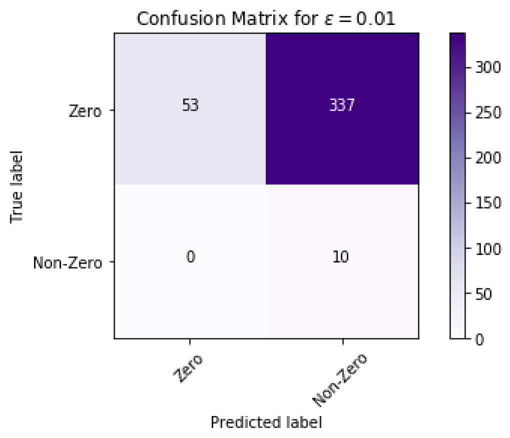
In [124]:
```
bin_coefs_true = [0 if i==0 else 1 for i in coefs_true] # your code goes here
eps_list = 10.**np.arange(-6,1)# your code goes here
for eps in eps_list:
    bin_coefs_estimated = [0 if np.abs(i)<eps else 1 for i in pred_fns[2]['coefs']] # your code goes here
    cnf_matrix = confusion_matrix(bin_coefs_true, bin_coefs_estimated)
    plt.figure()
    plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon = {}$".format(eps), classes=["Zer
o", "Non-Zero"])
```

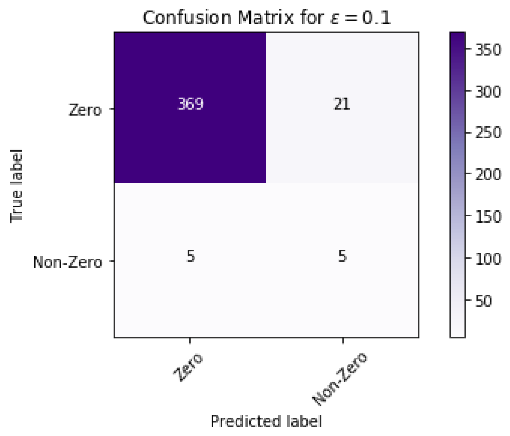

Confusion Matrix for $\varepsilon = 1e-06$
True label
Zero
Non-Zero
5
385
0
10
Predicted label
Zero
Non-Zero
0
50
100
150
200
250
300
350

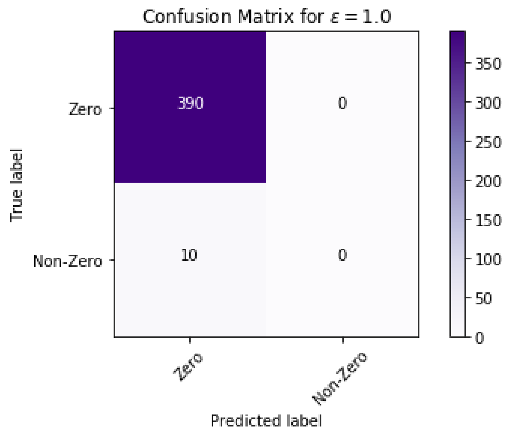

Confusion Matrix for $\varepsilon = 1e-05$
True label
Zero
Non-Zero
5
385
0
10
Predicted label
Zero
Non-Zero
0
50
100
150
200
250
300
350


Confusion Matrix for $\varepsilon = 0.0001$
True label
Zero
Non-Zero
9
381
0
10
Predicted label
Zero
Non-Zero
0
50
100
150
200
250
300
350


Confusion Matrix for $\varepsilon = 0.001$
True label
Zero
Non-Zero
11
379
0
10
Predicted label
Zero
Non-Zero
0
50
100
150
200
250
300
350


Confusion Matrix for $\varepsilon = 0.01$
True label
Zero
Non-Zero
Predicted label
53
337
0
10
300
250
200
150
100
50
0


Confusion Matrix for $\varepsilon = 0.1$
True label
Zero
Non-Zero
Predicted label
369
21
5
5
350
300
250
200
150
100
50


Confusion Matrix for $\varepsilon = 1.0$
True label
Zero
Non-Zero
Predicted label
390
0
10
0
350
300
250
200
150
100
50
0

Xinmeng Li xl1575 HW3

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

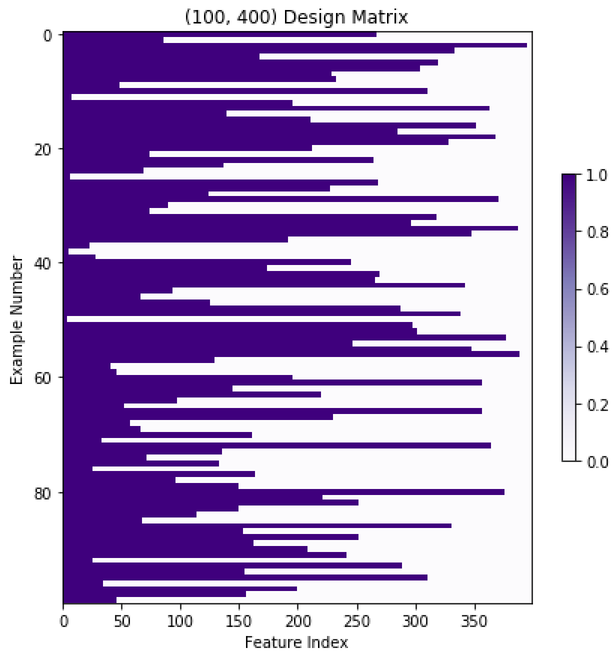
1. a_j = 2$X_{.,j}^T X_{.,j}$
c_j = 2($X_{.,j}^T y - X_{.,j}^T Xw + w_j X_{.,j}^T X_{.,j}$)

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
from sklearn.linear_model import 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'
```

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);
```



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.

In [4]:
```
def soft_threshold(a, delta):
    return np.sign(a)*max(0,(np.abs(a)-delta))

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

def compute_lasso_objective(X, y, w, l1_reg=1.):
    l1 = compute_sum_sqr_loss(X, y, w)
    l2 = l1_reg*np.sum(np.abs(w))
    return l1+l2

from numpy.linalg import inv
def get_ridge_solution(X, y, l1_reg):
    i = inv(np.matmul(X.T,X)+l1_reg*np.eye(X.shape[1]))
    return np.dot(i,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.

```
In [5]: def shooting_algorithm(X, y, w0=None, l1_reg = 1., max_num_epochs = 1000, min_obj_decrease=1e-8, random=False,
        plot=True):
            if w0 is None:
                w = np.zeros(X.shape[1])
            else:
                w = np.copy(w0)
            #print("l1_reg",l1_reg,"w0",sum(w))
            d = X.shape[1]
            epoch = 0
            obj_val = compute_lasso_objective(X, y, w, l1_reg)
            obj_decrease = min_obj_decrease + 1.
            loss_hist = []
            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:
                    if sum(X[:,j]**2) == 0:
                        continue
                    aj = 2*np.dot(X[:,j],X[:,j])
                    cj = 2*(np.dot(X[:,j],y)-np.dot(np.dot(X[:,j].T,X),w)+w[j]*np.dot(X[:,j],X[:,j]))
                    w[j] = soft_threshold(float(cj)/float(aj), float(l1_reg)/float(aj))
                obj_val = compute_lasso_objective(X, y, w, l1_reg)
                loss_hist.append(obj_val)
                obj_decrease = obj_old - obj_val
                epoch += 1
                #print(epoch,": loss decrease",obj_decrease)
            print("Ran for "+str(epoch)+" epochs. " + 'Lowest loss: ' + str(obj_val))
            if plot == True:
                plt.plot(np.arange(epoch),loss_hist)
                plt.title("Loss History for Shooting Algorithm")
                plt.xlabel("Epoch")
                plt.ylabel("Lasso Regression Objective Value")
                plt.show()
            return w
```

Class for Lasso Regression

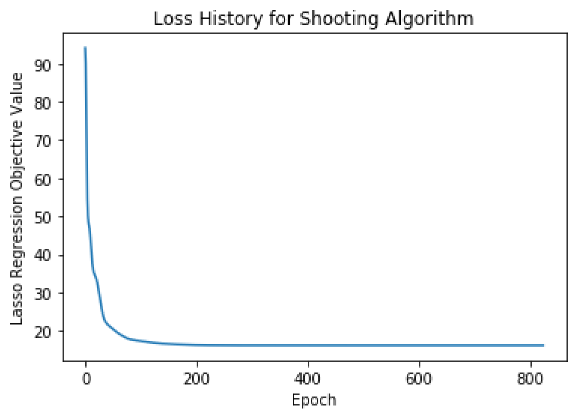
```
In [6]: class LassoRegression(BaseEstimator, RegressorMixin):
            """ Lasso regression"""
            def __init__(self, l1_reg=1.0, randomized=False):
                if l1_reg < 0:
                    raise ValueError('Regularization penalty should be at least 0.')
                self.l1_reg = l1_reg
                self.randomized = randomized


            def fit(self, X, y, max_epochs = 1000, coef_init=None,min_obj_decrease=1e-8,plot=True):
                # 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)
                self.w_ = shooting_algorithm(X, y, w0=coef_init, max_num_epochs=max_epochs,
                                             random=self.randomized,min_obj_decrease=min_obj_decrease,
                                             plot=plot,l1_reg = self.l1_reg)
                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 [7]: 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=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

            lasso_regression_estimator = LassoRegression(l1_reg=l1_reg)
            lasso_regression_estimator.fit(X_train, y_train,coef_init = np.zeros(X_train.shape[1]))
            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_regr
        ession_preds)**2)))
            # Let's compare differences parameter values
            print("Hoping this is very close to 0: {}".format(np.sum((our_coefs - sklearn_lasso_coefs)**2)))
```

In [8]:
```python
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=1)
```

```
Ran for 824 epochs. Lowest loss: 16.197737971873025
```



```
Hoping this is very close to 0 (predictions): 3.554888033867842e-07
Hoping this is very close to 0: 8.306670429825411e-05
```

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.

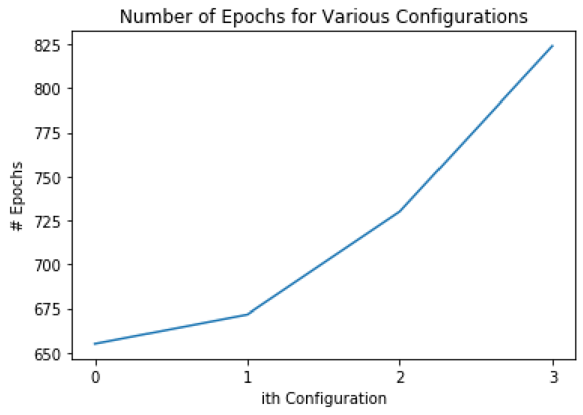
In [18]:
```python
def result(X_train,y_train,X_val,y_val,random=False,coef_init=None,min_obj_decrease=1e-8,plot=True):
    lasso_regression_estimator = LassoRegression(randomized=random)
    lasso_regression_estimator.fit(X_train, y_train,coef_init = coef_init,min_obj_decrease=min_obj_decrease,pl
ot=plot)
    print("+++++++ For configuration random =",random,", the initial w is",
          "Murphy w, " if coef_init is None else "0","+++++++")
    trainerr = lasso_regression_estimator.score(X_train,y_train)
    valerr = lasso_regression_estimator.score(X_val,y_val)
    print("training error:",trainerr)
    print("validation error:",valerr)
    return trainerr,valerr
```

In [19]:
```
tr = []
vr = []
for r in [True,False]:
    for coef in [None,np.zeros(X_train.shape[1])]:
        trerr = 0
        verr = 0
        k = 5 if r==True else 1
        for i in range(k):
            trainerr,valerr= result(X_train,y_train,X_val,y_val,random=r,coef_init=coef,plot=False)
            trerr = trerr + trainerr
            verr = verr + valerr
        tr.append(trerr/k)
        vr.append(verr/k)
```

```
Ran for 653 epochs. Lowest loss: 16.197738229745823
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195068283832321
validation error: 0.1260434085471485
Ran for 674 epochs. Lowest loss: 16.19773807650279
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195070299309847
validation error: 0.12526420582462167
Ran for 662 epochs. Lowest loss: 16.197738049516563
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195050200451924
validation error: 0.12470623842107793
Ran for 645 epochs. Lowest loss: 16.197738136973626
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195022714114953
validation error: 0.12576183048206707
Ran for 641 epochs. Lowest loss: 16.197738206498517
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.0919504870666686
validation error: 0.1246108370235637
Ran for 701 epochs. Lowest loss: 16.197738050302117
+++++++ For configuration random = True , the initial w is 0 +++++++
training error: 0.0919504133216184
validation error: 0.11811834368769462
Ran for 711 epochs. Lowest loss: 16.197738076309893
+++++++ For configuration random = True , the initial w is 0 +++++++
training error: 0.0919505407694667
validation error: 0.13033319241810062
Ran for 654 epochs. Lowest loss: 16.197738044268323
+++++++ For configuration random = True , the initial w is 0 +++++++
training error: 0.09195056652366851
validation error: 0.1468991707078634
Ran for 571 epochs. Lowest loss: 16.19773805902513
+++++++ For configuration random = True , the initial w is 0 +++++++
training error: 0.09195099112550356
validation error: 0.12440082249344152
Ran for 721 epochs. Lowest loss: 16.197738078887262
+++++++ For configuration random = True , the initial w is 0 +++++++
training error: 0.0919504811130107
validation error: 0.12694457604610485
Ran for 730 epochs. Lowest loss: 16.197738061447524
+++++++ For configuration random = False , the initial w is Murphy w,  +++++++
training error: 0.0919503631069907
validation error: 0.12643956987109525
Ran for 824 epochs. Lowest loss: 16.197737971873025
+++++++ For configuration random = False , the initial w is 0 +++++++
training error: 0.09194964844585712
validation error: 0.16781937496866564
```

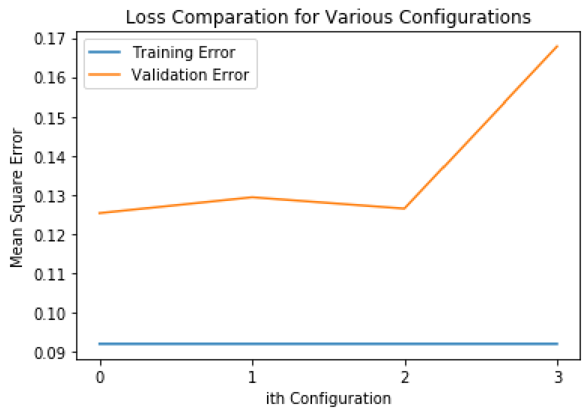
In [22]:
```
plt.plot(np.arange(4),[(653+674+662+645+641)/5,(701+711+654+571+721)/5,730,824])
plt.title("Number of Epochs for Various Configurations")
plt.xlabel("ith Configuration")
plt.ylabel("# Epochs")
plt.xticks(np.arange(4))
plt.plot()
```

Out[22]:
```
[]
```



In [23]:
```
plt.plot(np.arange(4),tr,label="Training Error")
plt.plot(np.arange(4),vr,label="Validation Error")
plt.title("Loss Comparation for Various Configurations")
plt.xlabel("ith Configuration")
plt.ylabel("Mean Square Error")
plt.xticks(np.arange(4))
plt.legend()
plt.plot()
```

Out[23]:
```
[]
```



In [24]:
```
print("The",vr.index(min(vr)),"th configuration has the lowest validation error",min(vr))
```

```
The 0 th configuration has the lowest validation error 0.12527730405969578
```

For the randomized coordinates descent, to reduce the variance, we run 5 iterations and then compute the average training error, validation error, and number of epochs.

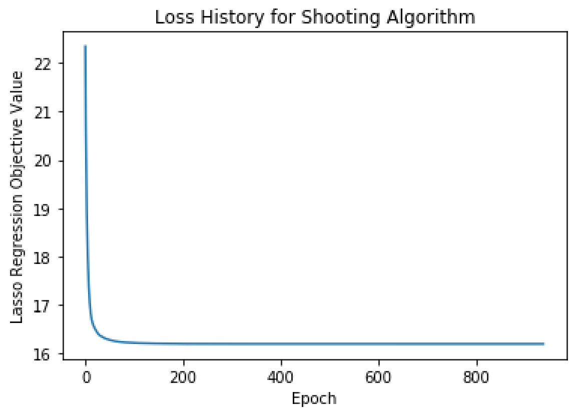
We observe that the training errors across these four configurations are very close. The validation errors rank is (randomized coordinates descent, starting at Murphy's ridge regression solution) < (cyclic coordinates descent, starting at Murphy's ridge regression solution) < (randomized coordinates descent, starting at 0) < (cyclic coordinates descent, starting at 0).

We observe that
when the coordinates descent is randomized, starting at Murphy's ridge regression solution converges faster,
when the coordinates descent is cyclic, starting at Murphy's ridge regression solution converges faster,
when starting at Murphy's ridge regression solution, randomized coordinates descent converges faster,
when starting at 0, randomized coordinates descent converges faster.

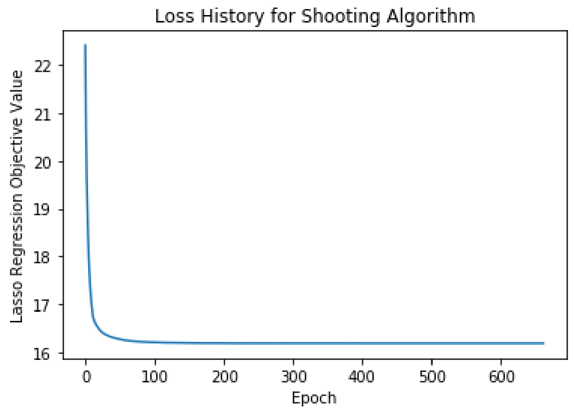
Thereforem (randomized coordinates descent, starting at Murphy's ridge regression solution) is the best configuration due to its excellent performance on validation set and fast convergence speed.

In [25]:
```
conv = 10.**np.arange(-10,-1,2)
for i in conv:
    print('----------------- min_obj_decrease =',i,'------------------')
    result(X_train,y_train,X_val,y_val,random=True,coef_init=None,min_obj_decrease=i)
```

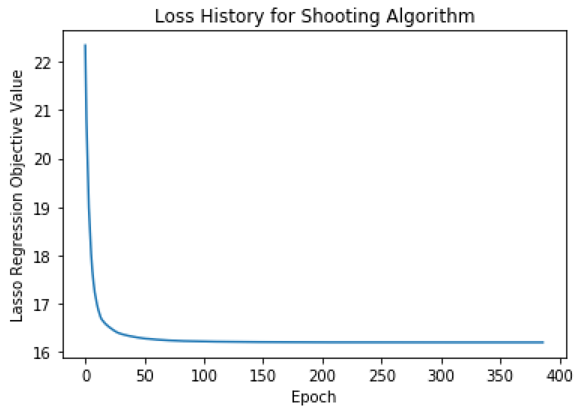
```
----------------- min_obj_decrease = 1e-10 ------------------
Ran for 938 epochs. Lowest loss: 16.19773747870058
```



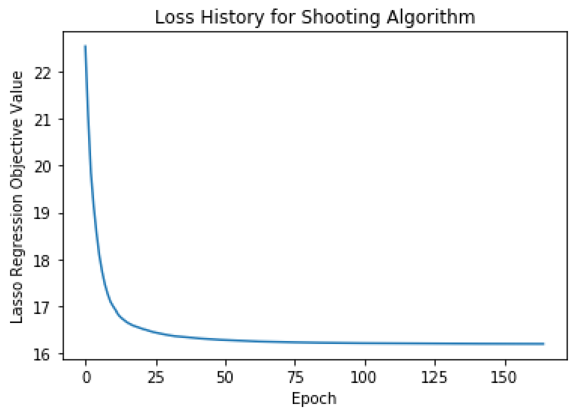
```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195019035579939
validation error: 0.12552979230875197
----------------- min_obj_decrease = 1e-08 ------------------
Ran for 662 epochs. Lowest loss: 16.197738042086463
```



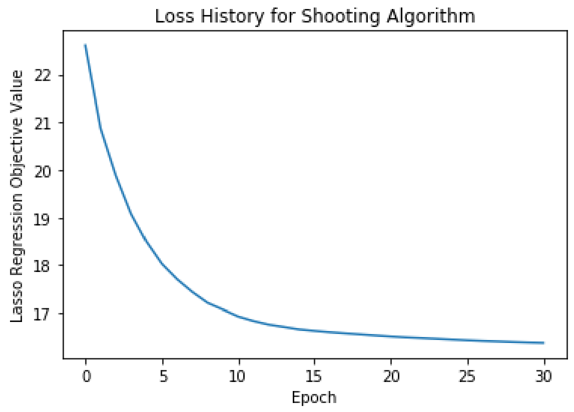
```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195029107112734
validation error: 0.1252889862397355
----------------- min_obj_decrease = 1e-06 ------------------
Ran for 387 epochs. Lowest loss: 16.197794245533284
```



```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195053112415973
validation error: 0.12549330541172524
----------------- min_obj_decrease = 0.0001 ------------------
Ran for 165 epochs. Lowest loss: 16.202749832736437
```



```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09181826389762485
validation error: 0.12683012642430738
----------------- min_obj_decrease = 0.01 ------------------
Ran for 31 epochs. Lowest loss: 16.374510946654173
```



```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.08687599868042913
validation error: 0.13164152362800985
```

We chose the optimal configuration from the four candidates above. Next, by varying the convergence rule, i.e. value of min_obj_decrease, we observe that when min_obj_decrease = 10^{-6}, the training error and validation error are very close to those of min_obj_decrease = default value 10^{-8}. min_obj_decrease = 10^{-6} (387 epochs) converges much faster than 10^{-8} (662 epochs). Thus, we select (randomized coordinates descent, starting at Murphy's ridge regression solution,min_obj_decrease = 10^{-6}) as our final optimal configuration for the experiments below.

3.

In [26]:
```
default_params = 10.**np.arange(-5,3,1)
def do_grid_search_lasso(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 = [{'l1_reg':params}]
        lasso_regression_estimator = LassoRegression(randomized=True)
        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 = Fa
lse))
        grid.fit(X_train_val, y_train_val,coef_init = None,plot=False,min_obj_decrease=1e-6)
        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
```

In [27]:
```
grid, results = do_grid_search_lasso(X_train, y_train, X_val, y_val)
```

```
Ran for 1 epochs. Lowest loss: 0.6755576053974903
Ran for 1 epochs. Lowest loss: 0.6789100309054874
Ran for 10 epochs. Lowest loss: 0.7123993557373929
Ran for 147 epochs. Lowest loss: 1.0423067551595389
Ran for 464 epochs. Lowest loss: 3.9050710024458515
Ran for 384 epochs. Lowest loss: 16.197817227917803
Ran for 386 epochs. Lowest loss: 56.819035258639765
Ran for 130 epochs. Lowest loss: 221.86400029089822
Ran for 1000 epochs. Lowest loss: 89.02315412088316
```

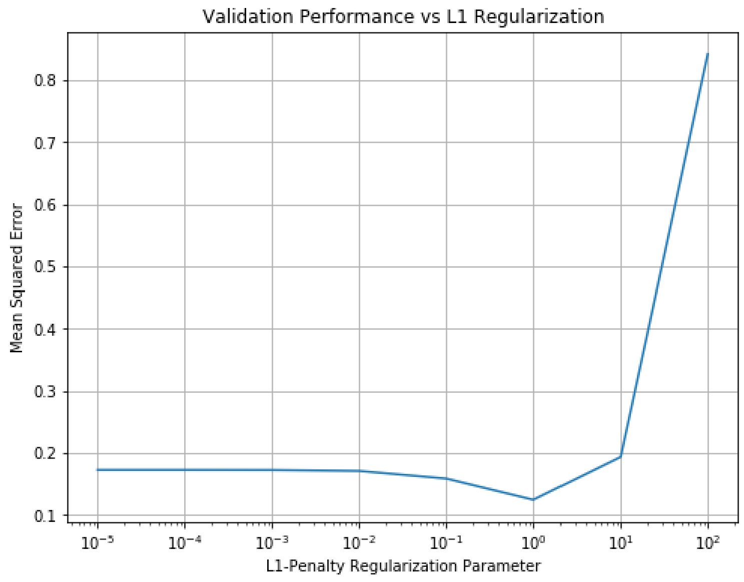
In [28]:
```
results
```

Out[28]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.00001	0.172589	0.006752
1	0.00010	0.172576	0.006752
2	0.00100	0.172435	0.006752
3	0.01000	0.170909	0.006800
4	0.10000	0.158666	0.011115
5	1.00000	0.124585	0.091952
6	10.00000	0.193493	0.226208
7	100.00000	0.841730	0.883711

In [29]:
```
# 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")
ax.semilogx(results["param_l1_reg"], results["mean_test_score"]);
print("Best parameter {0}".format(grid.best_params_['l1_reg']))
```

```
Best parameter 1.0
```



In [37]:
```
grid, results = do_grid_search_lasso(X_train, y_train, X_val, y_val,params = np.arange(0.8,1.8,0.2))
results
```

```
Ran for 410 epochs. Lowest loss: 14.641035429836066
Ran for 373 epochs. Lowest loss: 16.19779535441917
Ran for 394 epochs. Lowest loss: 17.54784743828502
Ran for 380 epochs. Lowest loss: 18.862205924179015
Ran for 373 epochs. Lowest loss: 20.146391592104592
Ran for 1000 epochs. Lowest loss: 89.02346709468117
```

Out[37]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.8	0.125172	0.075832
1	1.0	0.123994	0.091951
2	1.2	0.125109	0.095712
3	1.4	0.126845	0.097673
4	1.6	0.126132	0.099934

In [38]:
```
print("Best parameter {0}".format(grid.best_params_['l1_reg']))
```

```
Best parameter 1.0
```

After grid search, we conclude that the optimal λ is 1.0.

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

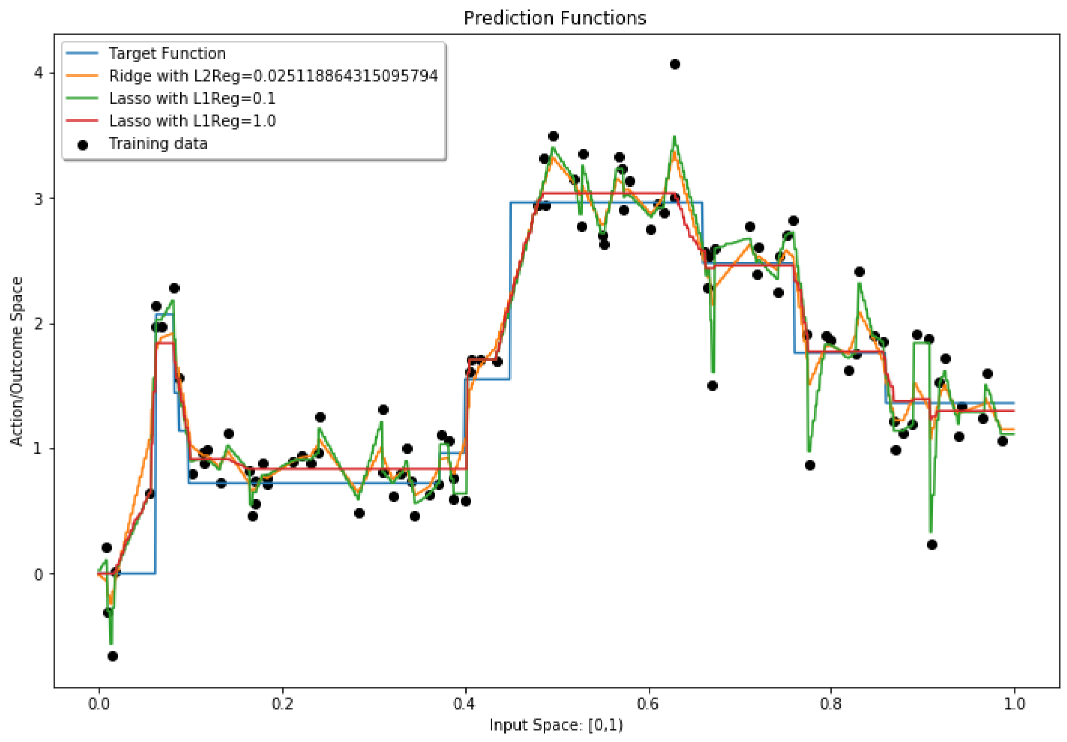
In [39]:
```
class RidgeRegression(BaseEstimator, RegressorMixin):
        """ ridge regression"""

        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)
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
```

In [40]:
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))
X = featurize(x)
pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds": target_fn(x)})

l1regs = [0.1, grid.best_params_['l1_reg']]
ridge_regression_estimator = RidgeRegression(l2reg=10.**-1.6)
ridge_regression_estimator.fit(X_train, y_train)
pred_fns.append({"name": "Ridge with L2Reg="+str(10.**-1.6), "coefs": ridge_regression_estimator.w_,
                 "preds": ridge_regression_estimator.predict(X)})
for l1_reg in l1regs:
    lasso_regression_estimator = LassoRegression(l1_reg=l1_reg,randomized=True)
    lasso_regression_estimator.fit(X_train, y_train,coef_init = None,min_obj_decrease=1e-6,plot=False)
    name = "Lasso with L1Reg="+str(l1_reg)
    pred_fns.append({"name":name,
                     "coefs":lasso_regression_estimator.w_,
                     "preds": lasso_regression_estimator.predict(X) })
```

```
Ran for 459 epochs. Lowest loss: 3.9050767049280837
Ran for 384 epochs. Lowest loss: 16.19779635339426
```

In [41]:
```
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 [42]:
```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



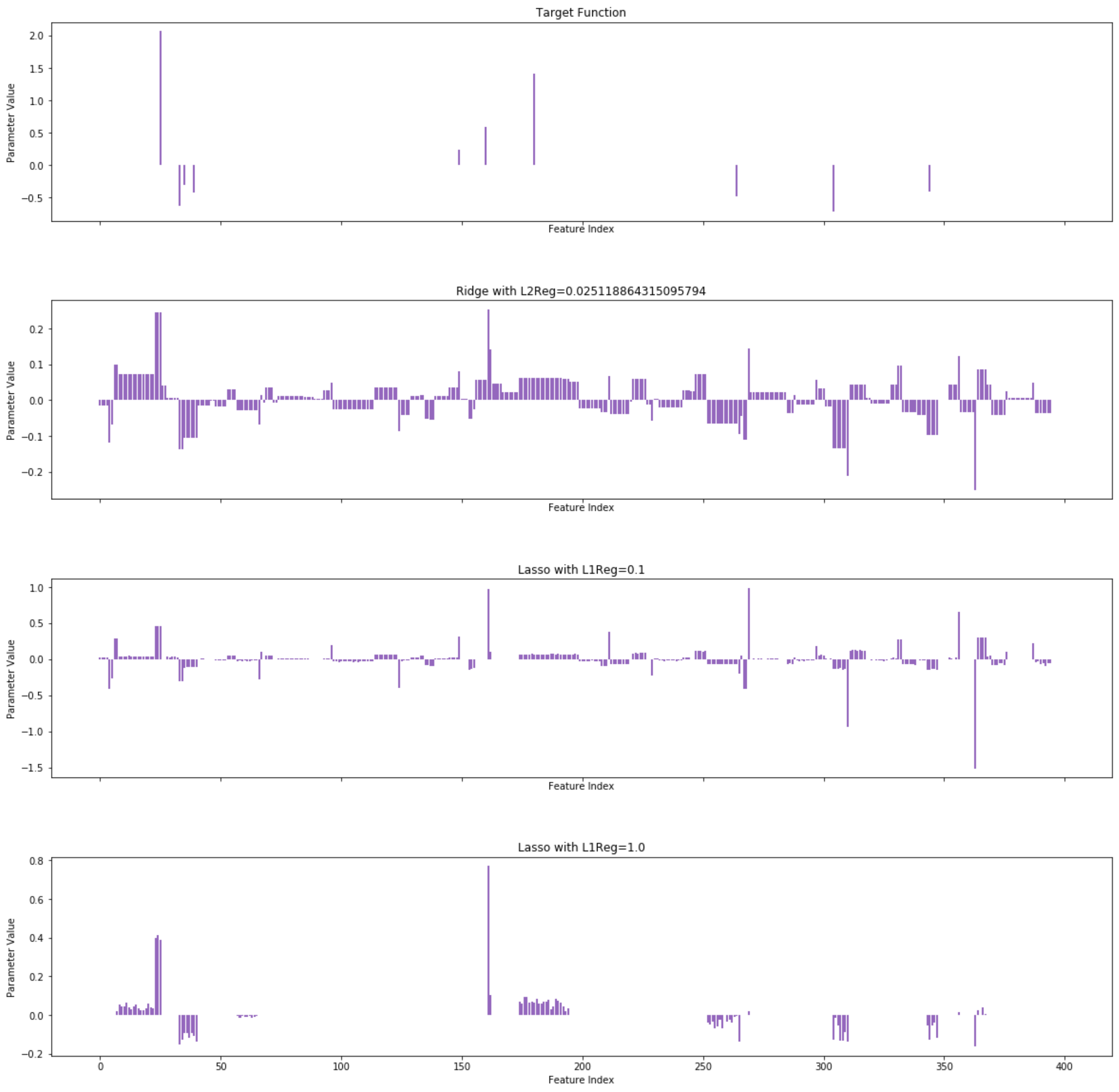
When $\lambda = 1$, the mean test loss of the lasso regression is 0.124585, which is higher than ridge regression's 0.139392. It is obvious to tell this performance difference on the graph. Although the optimal ridge regression prediction function and the lasso regression prediction function with l1_reg = 0.1 fits the training data well, they have too many flutuations to fit the target function, which indicates overfitting. The optimal lasso regression prediction function with l1_reg = 1 has a flatter shape and gets close to the target function. It turns out that lasso reduces overfitting.

Visualizing the Weights

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

In [43]:
```python
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 [44]:
```python
compare_parameter_vectors(pred_fns);
```



We observe that the target function and the lasso prediction function with l1_reg = 1 are very sparse. Ridge regression prediction is the least sparse. The sparcity of the lasso regression is more consistent with the target function.

Theoratically, lasso regression is supposed to perform well when the parameter is sparse and the number of features is greater than the number of instances. Compared with the l2 regularization in ridge regession, penalizing the l1 norm in the lasso regression is an more efficient way of promoting sparsity. From two figures above, we observe that the lasso regression performs better than the ridge regression after fitting the training set with dimension mxn, where n(400) > m(100). We conclude that the lasso regression performs as expected in practice.

4.

Continuation Method

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

```
In [45]: 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.

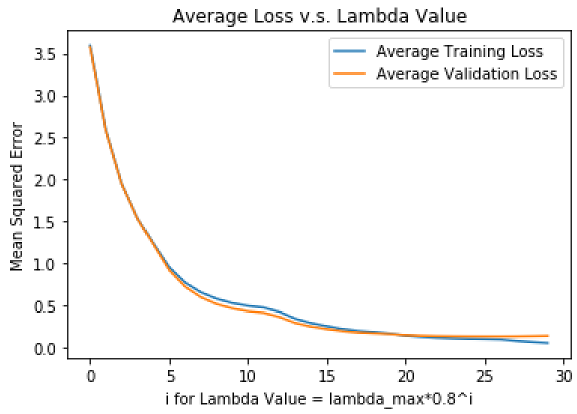
In [46]:
```python
from sklearn.base import clone
class LassoRegularizationPath:
    def __init__(self, estimator, tune_param_name):
        self.estimator = estimator
        self.tune_param_name = tune_param_name


    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 = []
        train_reg = []
        val_reg = []
        w_start = coef_init
        for reg_val in reg_vals:
            estimator = clone(self.estimator)
            estimator.l1_reg = reg_val
            estimator.randomized = True
            estimator.fit(X_train, y_train,plot=False,coef_init=w_start,min_obj_decrease=1e-6)
            self.results.append({"reg_val":reg_val, "estimator":estimator})
            if warm_start == True:
                w_start = estimator.w_
              # print(sum(estimator.w_))
        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
```

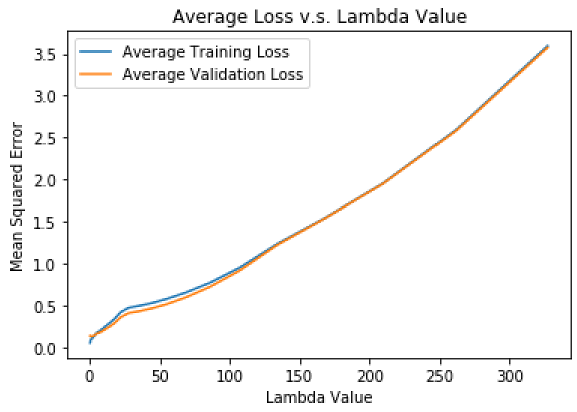
In [47]:
```python
def do_grid_search_homotopy(X_train, y_train, X_val, y_val,
                            reg_vals=None, w0=None,warm_start=True):
    if reg_vals is None:
        lambda_max = get_lambda_max_no_bias(X_train, y_train)
        reg_vals = [lambda_max * (.8**n) for n in range(0, 30)]

    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=warm_start)

    return lasso_reg_path_estimator, reg_vals
```

In [48]:
```
def lasso_reg_path_plot(lasso_reg_path_estimator,reg_vals):
    score_train = lasso_reg_path_estimator.score(X_train, y_train)
    score_val = lasso_reg_path_estimator.score(X_val, y_val)
    sc_train = []
    sc_val = []
    for s in score_train:
        sc_train.append(s['score'])
    for s in score_val:
        sc_val.append(s['score'])
    plt.plot(np.arange(0,len(reg_vals)),sc_train,label = "Average Training Loss")
    plt.plot(np.arange(0,len(reg_vals)),sc_val,label = "Average Validation Loss")
    plt.title("Average Loss v.s. Lambda Value")
    plt.xlabel("i for Lambda Value = lambda_max*0.8^i")
    plt.ylabel("Mean Squared Error")
    plt.legend()
    plt.show()
    plt.plot(reg_vals,sc_train,label = "Average Training Loss")
    plt.plot(reg_vals,sc_val,label = "Average Validation Loss")
    plt.title("Average Loss v.s. Lambda Value")
    plt.xlabel("Lambda Value")
    plt.ylabel("Mean Squared Error")
    plt.legend()
    plt.show()
    min_val_ind = sc_val.index(min(sc_val))
    min_train_ind = sc_train.index(min(sc_train))
    print("The lowest validation error is ",min(sc_val),"when lambda = lambda_max*0.8^"+str(min_val_ind),
          "=",reg_vals[min_val_ind])
    print("The lowest training error is ",min(sc_train),"when lambda = lambda_max*0.8^"+str(min_train_ind),
          "=",reg_vals[min_train_ind])
```

```
In [49]: lasso_reg_path_estimator, reg_vals = do_grid_search_homotopy(X_train, y_train, X_val,  y_val,  None)
         lasso_reg_path_plot(lasso_reg_path_estimator,reg_vals)
```

```
Ran for 2 epochs. Lowest loss: 359.6674002813196
Ran for 39 epochs. Lowest loss: 348.5211666646745
Ran for 164 epochs. Lowest loss: 323.5371684531793
Ran for 157 epochs. Lowest loss: 293.22932629913595
Ran for 117 epochs. Lowest loss: 262.23640163312444
Ran for 138 epochs. Lowest loss: 231.30369921867901
Ran for 52 epochs. Lowest loss: 202.017514713506
Ran for 201 epochs. Lowest loss: 175.68303057604152
Ran for 227 epochs. Lowest loss: 152.73959817232725
Ran for 90 epochs. Lowest loss: 133.12878381568868
Ran for 75 epochs. Lowest loss: 116.6209862497147
Ran for 11 epochs. Lowest loss: 102.89046468541488
Ran for 46 epochs. Lowest loss: 91.40133206459613
Ran for 74 epochs. Lowest loss: 80.59514965528945
Ran for 66 epochs. Lowest loss: 70.65133839454879
Ran for 54 epochs. Lowest loss: 61.83897583680391
Ran for 133 epochs. Lowest loss: 54.08109738742876
Ran for 59 epochs. Lowest loss: 47.326681666302235
Ran for 69 epochs. Lowest loss: 41.564315636468734
Ran for 116 epochs. Lowest loss: 36.69715720494378
Ran for 151 epochs. Lowest loss: 32.32320291769676
Ran for 143 epochs. Lowest loss: 28.4347907543766
Ran for 131 epochs. Lowest loss: 25.071561627942835
Ran for 151 epochs. Lowest loss: 22.211127120499434
Ran for 190 epochs. Lowest loss: 19.79973413206443
Ran for 175 epochs. Lowest loss: 17.789543442426897
Ran for 165 epochs. Lowest loss: 16.121450780618837
Ran for 245 epochs. Lowest loss: 14.564042309934502
Ran for 259 epochs. Lowest loss: 12.993180061024713
Ran for 256 epochs. Lowest loss: 11.487591697765343
```

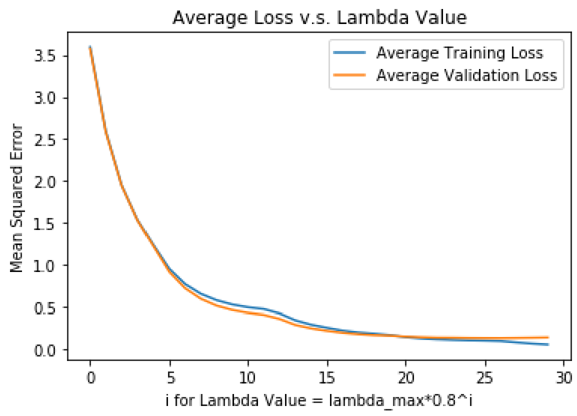


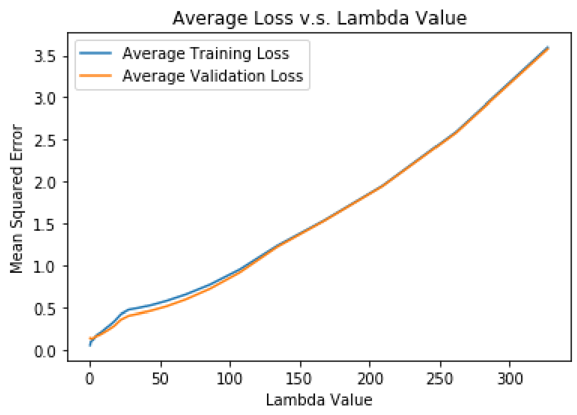


```
The lowest validation error is  0.12701703872952738 when lambda = lambda_max*0.8^26 = 0.9891516657994105
The lowest training error is  0.04961238021137428 when lambda = lambda_max*0.8^29 = 0.5064456528892983
```

```
In [50]: lasso_reg_path_estimator2, reg_vals = do_grid_search_homotopy(X_train, y_train, X_val,  y_val,warm_start=False
         )
         lasso_reg_path_plot(lasso_reg_path_estimator2,reg_vals)
```

```
Ran for 2 epochs. Lowest loss: 359.6674002813196
Ran for 78 epochs. Lowest loss: 348.52115687028925
Ran for 150 epochs. Lowest loss: 323.53717377151094
Ran for 76 epochs. Lowest loss: 293.22929551791174
Ran for 224 epochs. Lowest loss: 262.2364198974681
Ran for 131 epochs. Lowest loss: 231.30366691300418
Ran for 104 epochs. Lowest loss: 202.01751497078916
Ran for 347 epochs. Lowest loss: 175.683019148752
Ran for 386 epochs. Lowest loss: 152.7395819710298
Ran for 404 epochs. Lowest loss: 133.12878632505294
Ran for 392 epochs. Lowest loss: 116.62098310674956
Ran for 404 epochs. Lowest loss: 102.89047148373982
Ran for 393 epochs. Lowest loss: 91.40133940148172
Ran for 368 epochs. Lowest loss: 80.59519666165876
Ran for 367 epochs. Lowest loss: 70.65137666437599
Ran for 378 epochs. Lowest loss: 61.83904339709373
Ran for 386 epochs. Lowest loss: 54.08113880946387
Ran for 383 epochs. Lowest loss: 47.32674007849685
Ran for 364 epochs. Lowest loss: 41.5643592106635
Ran for 394 epochs. Lowest loss: 36.697198691309815
Ran for 363 epochs. Lowest loss: 32.32323514538814
Ran for 392 epochs. Lowest loss: 28.43482302713611
Ran for 389 epochs. Lowest loss: 25.071600207572576
Ran for 387 epochs. Lowest loss: 22.211158339219043
Ran for 394 epochs. Lowest loss: 19.7997674502548
Ran for 388 epochs. Lowest loss: 17.7895713408946
Ran for 364 epochs. Lowest loss: 16.12148394725622
Ran for 405 epochs. Lowest loss: 14.564039331262691
Ran for 398 epochs. Lowest loss: 12.993189150775276
Ran for 382 epochs. Lowest loss: 11.487613665075473
```





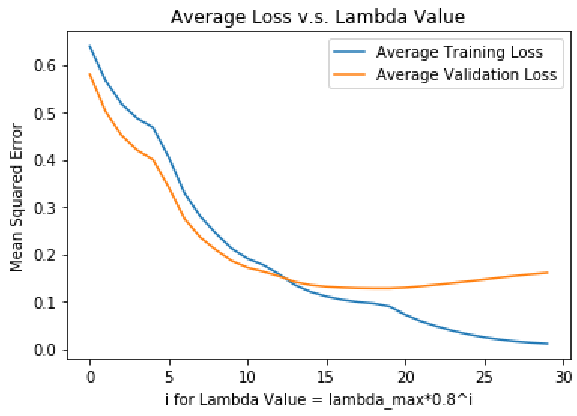
```
The lowest validation error is  0.12594148916705775 when lambda = lambda_max*0.8^26 = 0.9891516657994105
The lowest training error is  0.049540391229877904 when lambda = lambda_max*0.8^29 = 0.5064456528892983
```

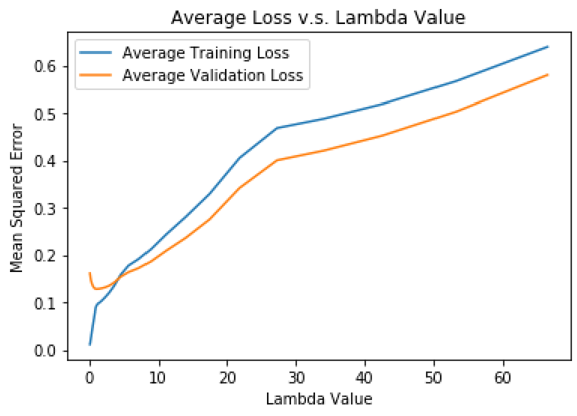
Comparing the output of the lasso_reg_path_estimator with and without warm_start, we observe that although these two estimators have very similar average loss pattern and almost same lowest loss, the estimator with warm_start generally converges faster. This makes sense because we expect starting with the theta learned in the previous iteration is an approach to saving time.

5.

```
In [55]: y_train_centered = y_train-np.mean(y_train)
         y_val_centered = y_val-np.mean(y_val)
         lasso_reg_path_estimator3, reg_vals = do_grid_search_homotopy(X_train, y_train_centered, X_val,  y_val_centere
         d)
         lasso_reg_path_plot(lasso_reg_path_estimator3,reg_vals)
```

```
Ran for 359 epochs. Lowest loss: 172.25599324167302
Ran for 220 epochs. Lowest loss: 149.79054213163022
Ran for 224 epochs. Lowest loss: 130.63132804756674
Ran for 112 epochs. Lowest loss: 114.53457778011821
Ran for 113 epochs. Lowest loss: 101.16478952494161
Ran for 90 epochs. Lowest loss: 89.85218861751893
Ran for 177 epochs. Lowest loss: 79.13764995350124
Ran for 72 epochs. Lowest loss: 69.34477190064281
Ran for 66 epochs. Lowest loss: 60.69174726328947
Ran for 121 epochs. Lowest loss: 53.07231358368207
Ran for 82 epochs. Lowest loss: 46.459140468128595
Ran for 55 epochs. Lowest loss: 40.83153979427125
Ran for 123 epochs. Lowest loss: 36.05809529814436
Ran for 156 epochs. Lowest loss: 31.746323904717677
Ran for 152 epochs. Lowest loss: 27.93109980453006
Ran for 145 epochs. Lowest loss: 24.64078865914845
Ran for 158 epochs. Lowest loss: 21.84672623453983
Ran for 187 epochs. Lowest loss: 19.494538515808465
Ran for 175 epochs. Lowest loss: 17.536638838243753
Ran for 160 epochs. Lowest loss: 15.903047884481616
Ran for 259 epochs. Lowest loss: 14.342601781912606
Ran for 255 epochs. Lowest loss: 12.77648585335596
Ran for 256 epochs. Lowest loss: 11.283771896336637
Ran for 258 epochs. Lowest loss: 9.887652549614671
Ran for 338 epochs. Lowest loss: 8.602351773558706
Ran for 398 epochs. Lowest loss: 7.437388949228355
Ran for 385 epochs. Lowest loss: 6.395138451990898
Ran for 354 epochs. Lowest loss: 5.477539605688921
Ran for 341 epochs. Lowest loss: 4.675347234903969
Ran for 313 epochs. Lowest loss: 3.9866355091306613
```





```
The lowest validation error is  0.12764437697867023 when lambda = lambda_max*0.8^19 = 0.9586535641133789
The lowest training error is  0.011415528210449515 when lambda = lambda_max*0.8^29 = 0.1029346426515201
```

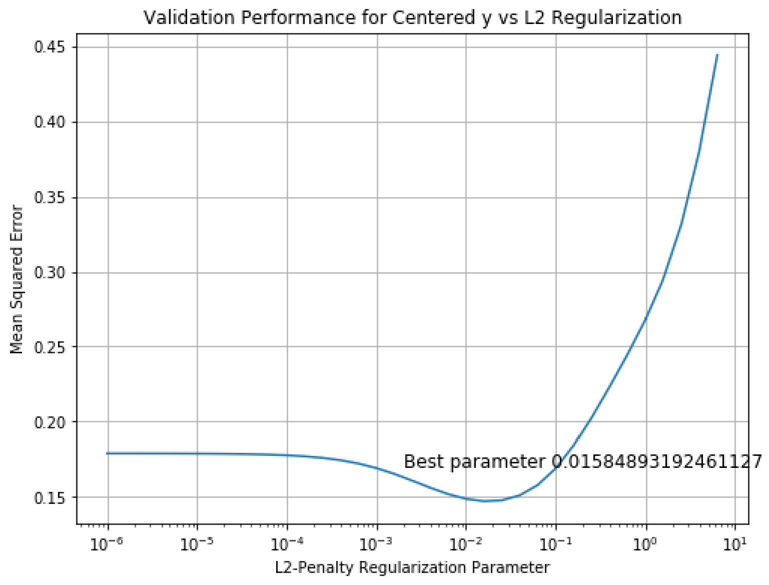
We observe that the average loss range around (0,0.6) with zero centered y is smaller than that (0,3.5) with the original y. The optimal validation loss of the lasso regession with homotopy method slightly increases from 0.1270 to 0.1276. It is interesting to see that the lowest training error decreases from 0.0495 to 0.0114. The phenomena observed is probably because the λ_{max} value decreases significantly with zero-centered y, which causes a chain of λ values decreases.

In [52]:
```
params = np.concatenate((10.**np.arange(-6,1,0.2), np.arange(1,2,.5)))
grid, results = do_grid_search_ridge(X_train, y_train_centered, X_val, y_val_centered,params = params)
results
```

Out[52]:

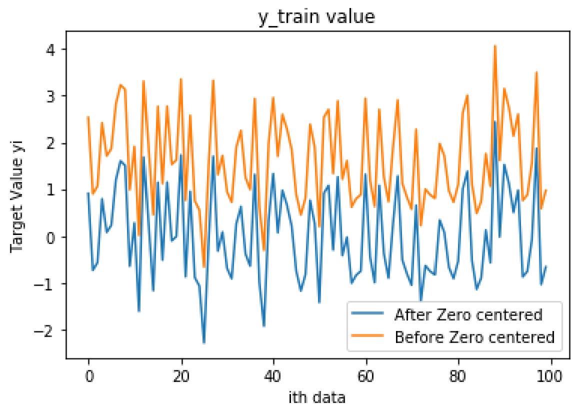
	param_l2reg	mean_test_score	mean_train_score
0	0.000001	0.178923	0.006752
1	0.000002	0.178916	0.006752
2	0.000003	0.178904	0.006752
3	0.000004	0.178885	0.006752
4	0.000006	0.178856	0.006752
5	0.000010	0.178809	0.006752
6	0.000016	0.178736	0.006752
7	0.000025	0.178619	0.006753
8	0.000040	0.178436	0.006755
9	0.000063	0.178149	0.006761
10	0.000100	0.177703	0.006773
11	0.000158	0.177014	0.006804
12	0.000251	0.175967	0.006879
13	0.000398	0.174410	0.007052
14	0.000631	0.172181	0.007441
15	0.001000	0.169136	0.008264
16	0.001585	0.165261	0.009881
17	0.002512	0.160756	0.012771
18	0.003981	0.156062	0.017403
19	0.006310	0.151790	0.024046
20	0.010000	0.148593	0.032714
21	0.015849	0.147030	0.043337
22	0.025119	0.147615	0.056040
23	0.039811	0.150957	0.071310
24	0.063096	0.157811	0.089960
25	0.100000	0.168841	0.112772
26	0.158489	0.184143	0.139833
27	0.251189	0.202827	0.169976
28	0.398107	0.223430	0.201277
29	0.630957	0.245028	0.232334
35	1.000000	0.268280	0.263542
30	1.000000	0.268280	0.263542
36	1.500000	0.292177	0.293148
31	1.584893	0.295823	0.297482
32	2.511886	0.331886	0.338489
33	3.981072	0.380806	0.391060
34	6.309573	0.444376	0.457167

In [54]:
```
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance for Centered y 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(0.002,0.17,"Best parameter {0}".format(grid.best_params_['l2reg']), fontsize = 12);
```



Comparing this graph with the one we got in Ridge Regression Q1, we find that the validation loss generally increases. The optimal validation loss of the ridge regession increases from 0.139392 to 0.147030. The maximum validation loss also increases from 0.294445 to 0.444376.

In [66]:
```
plt.title("y_train value")
plt.ylabel("Target Value yi")
plt.xlabel("ith data")
plt.plot(y_train_centered,label="After Zero centered")
plt.plot(y_train,label="Before Zero centered")
plt.legend()
plt.show()
```



We observe that after my processing, the y_train is zero-centered now.

Projected SGD

1.

```
In [190]: from sklearn.utils import shuffle
          def projection_SGD_split(X, y, theta_positive_0, theta_negative_0, lambda_reg = 1.0, alpha = 0.1, num_iter = 1
          000):
              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
              t = 0
              theta = theta_positive - theta_negative
              loss = compute_sum_sqr_loss(X, y, theta)
              loss_change = 1.
              alpha_curr = alpha
              while (loss_change>1e-6) and (times<num_iter):
                  loss_old = loss
                  X, y = shuffle(X, y)
                  for i in range(m):
                      t = t+1
                      alpha_curr = alpha
                      grad_positive = 2*X[i]*(np.dot(X[i],theta)-y[i])+lambda_reg*np.ones(n)
                      grad_negative = -2*X[i]*(np.dot(X[i],theta)-y[i])+lambda_reg*np.ones(n)
                      theta_positive = theta_positive - alpha_curr*grad_positive
                      theta_negative = theta_negative - alpha_curr*grad_negative
                      theta_positive = np.clip(theta_positive,0,None)
                      theta_negative = np.clip(theta_negative,0,None)
                      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
```

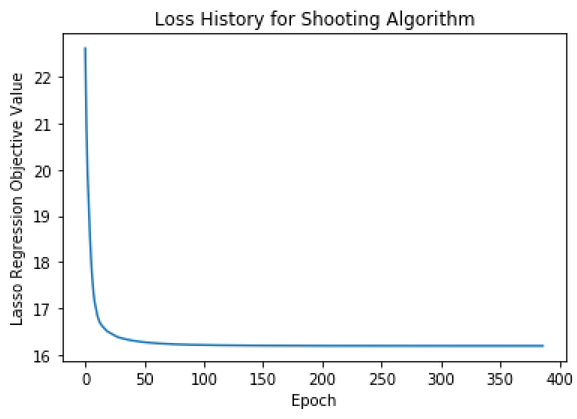
```
In [106]: x_training, y_training, x_validation, y_validation, target_fn, coefs_true, featurize = load_problem(PICKLE_PAT
          H)
          X_training = featurize(x_training)
          X_validation = featurize(x_validation)
          D = X_training.shape[1]
```

```
In [218]: theta = projection_SGD_split(X_training, y_training, np.zeros(D), np.zeros(D),alpha = 0.00009)
          print("Training Error for projected sgd is",compute_sum_sqr_loss(X_training, y_training, theta)/len(y_training
          ))
          print("Validation Error for projected sgd is",compute_sum_sqr_loss(X_validation, y_validation, theta)/len(y_va
          lidation))
```

```
(SGD) Ran for 1000 epochs. Loss:88.48996125141022 Lambda: 1.0
Training Error for projected sgd is 0.8848996125141022
Validation Error for projected sgd is 0.8422935708237576
```

```
In [88]: _,_=result(X_training, y_training,X_validation, y_validation,random=True,min_obj_decrease=1e-6)
```

```
Ran for 387 epochs. Lowest loss: 16.19779975084048
```



```
+++++++ For configuration random = True , the initial w is Murphy w,  +++++++
training error: 0.09195170185035917
validation error: 0.12710724862701128
```

We observe that for $\lambda = 1$, the average validation error of the shooting algorithm 0.127 is lower than that of the projected SGD algorith 0.842. They are not very close but still in the same order of magnitude. Now let's explore more λ value to compare the difference.

In [237]:
```python
lambda_max = get_lambda_max_no_bias(x_training, y_training)
reg_vals = [lambda_max * (.6**n) for n in range(15, 25)]
```

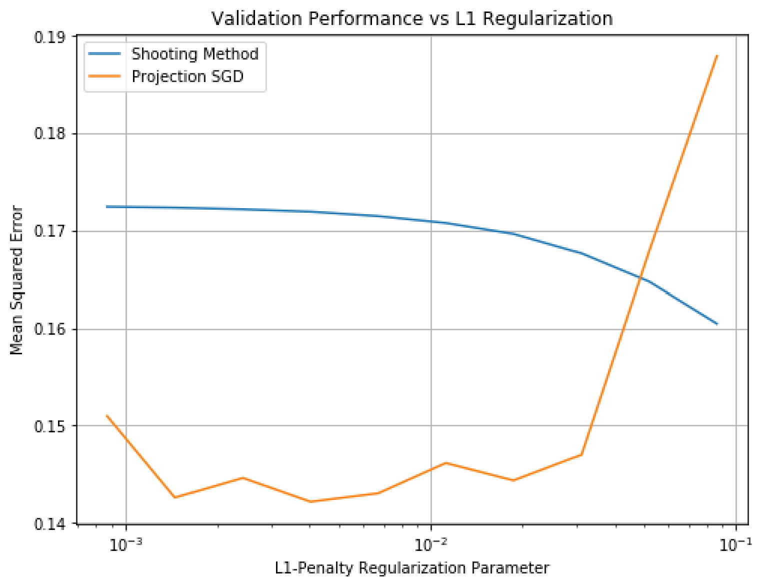
In [158]:
```python
loss_shooting = []
for lambda_value in reg_vals:
    lasso_regression_estimator = LassoRegression(l1_reg = lambda_value,randomized=True)
    lasso_regression_estimator.fit(X_training, y_training,plot=False)
    loss_shooting.append(lasso_regression_estimator.score(X_validation,y_validation))
```

```
Ran for 1000 epochs. Lowest loss: 3.517884401866023
Ran for 1000 epochs. Lowest loss: 2.464848785330584
Ran for 857 epochs. Lowest loss: 1.7818821032811232
Ran for 701 epochs. Lowest loss: 1.351505101982832
Ran for 574 epochs. Lowest loss: 1.0854372300051507
Ran for 449 epochs. Lowest loss: 0.9229544111996358
Ran for 345 epochs. Lowest loss: 0.8244383935702673
Ran for 251 epochs. Lowest loss: 0.7649524261255178
Ran for 182 epochs. Lowest loss: 0.7291232329491543
Ran for 129 epochs. Lowest loss: 0.7075760465724138
```

In [210]:
```python
 = []
for lambda_value in reg_vals:
    theta = projection_SGD_split(X_training, y_training, np.zeros(D), np.zeros(D)
                             ,alpha = 0.00009,lambda_reg = lambda_value)
    loss_SGD_list.append(compute_sum_sqr_loss(X_validation, y_validation, theta)/len(y_validation))
```

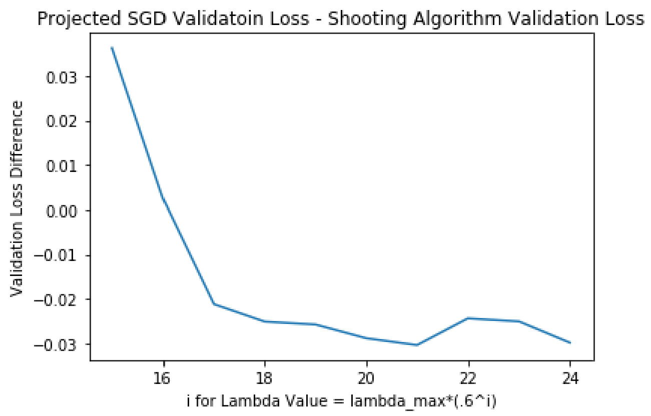
```
(SGD) Ran for 1000 epochs. Loss:21.432908590004075 Lambda: 0.08639124693688117
(SGD) Ran for 1000 epochs. Loss:18.116474101211146 Lambda: 0.0518347481621287
(SGD) Ran for 1000 epochs. Loss:13.488125744248325 Lambda: 0.031100848897277218
(SGD) Ran for 1000 epochs. Loss:11.76124887893916 Lambda: 0.01866050933836633
(SGD) Ran for 1000 epochs. Loss:10.981063903221322 Lambda: 0.0111963056030198
(SGD) Ran for 1000 epochs. Loss:9.768666672666905 Lambda: 0.006717783361811879
(SGD) Ran for 1000 epochs. Loss:8.517273138361404 Lambda: 0.004030670017087127
(SGD) Ran for 1000 epochs. Loss:7.725563108132105 Lambda: 0.002418402010252276
(SGD) Ran for 1000 epochs. Loss:7.131443976430828 Lambda: 0.0014510412061513654
(SGD) Ran for 1000 epochs. Loss:7.411839501819506 Lambda: 0.0008706247236908194
```

In [211]:
```
# 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();
```



In [238]:
```
diff_loss = [(loss_SGD_list[i]-loss_shooting[i]) for i in range(len(reg_vals))]
```

In [241]:
```
plt.plot(np.arange(15,25),diff_loss)
plt.title("Projected SGD Validatoin Loss - Shooting Algorithm Validation Loss")
plt.xlabel("i for Lambda Value = lambda_max*(.6^i)")
plt.ylabel("Validation Loss Difference")
plt.show()
```



We observe that as the lambda value decreases, (Projected SGD Validatoin Loss - Shooting Algorithm Validation Loss) decreases. From the graph we observe that the difference is about 0 when i=16, $\lambda = \lambda_{max} * 0.6^{16} \approx 0.05$. When $\lambda > 0.05$, the Shooting Algorithm has a lower validation loss. When $\lambda <= 0.05$, Projected SGD has a lower validation loss.

2.

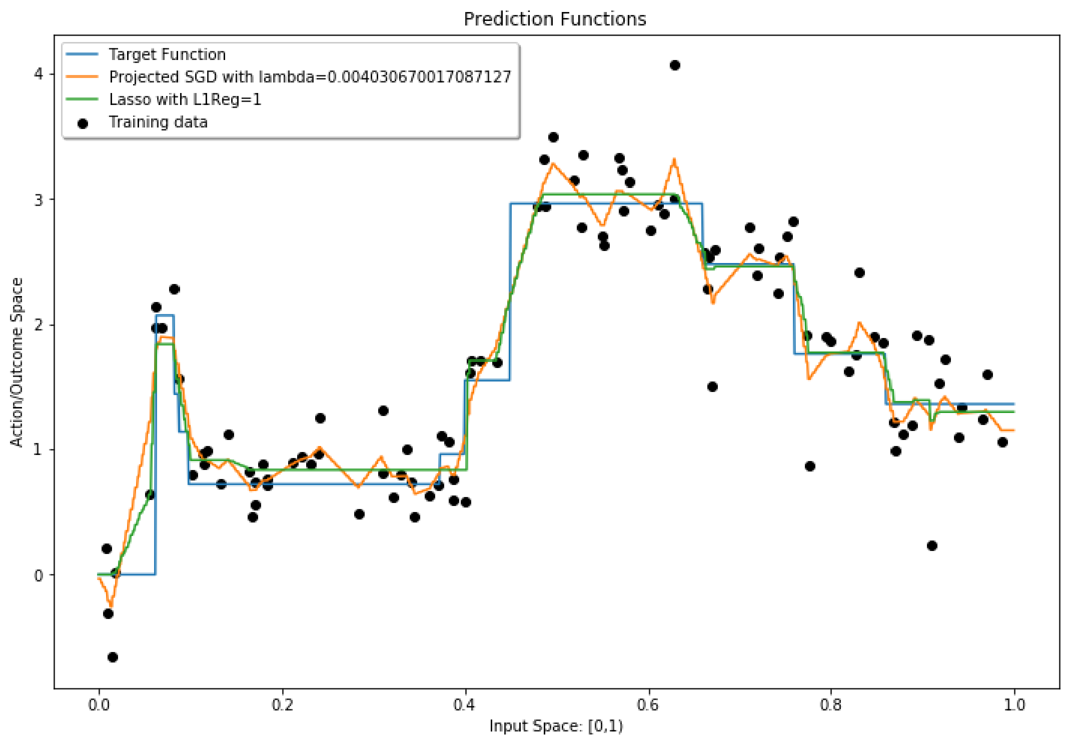
In [212]:
```
# Report the best
lambda_best_SGD = reg_vals[np.argmin(loss_SGD_list)]
theta_lasso_SGD_best = projection_SGD_split(X_training, y_training, np.zeros(D),np.zeros(D),
                                             lambda_reg=lambda_best_SGD, alpha = 0.001)
print('Best lambda for SGD is {0} with loss {1}'.format(lambda_best_SGD, np.min(loss_SGD_list)))
```

```
(SGD) Ran for 1000 epochs. Loss:6.3778949104473694 Lambda: 0.004030670017087127
Best lambda for SGD is 0.004030670017087127 with loss 0.14217598710841406
```

In [229]:
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_training]))
X = featurize(x)
pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds": target_fn(x)})

theta = projection_SGD_split(X_training, y_training, np.zeros(D), np.zeros(D)
                                  ,alpha = 0.00009,lambda_reg = lambda_value)
pred_fns.append({"name": "Projected SGD with lambda="+str(lambda_best_SGD), "coefs": theta,
                 "preds": np.dot(X,theta)})
lasso_regression_estimator = LassoRegression(randomized=True)
lasso_regression_estimator.fit(X_training, y_training,min_obj_decrease=1e-6,plot=False)
name = "Lasso with L1Reg="+str(1)
pred_fns.append({"name":name,"coefs":lasso_regression_estimator.w_,"preds": lasso_regression_estimator.predict
(X) })
```

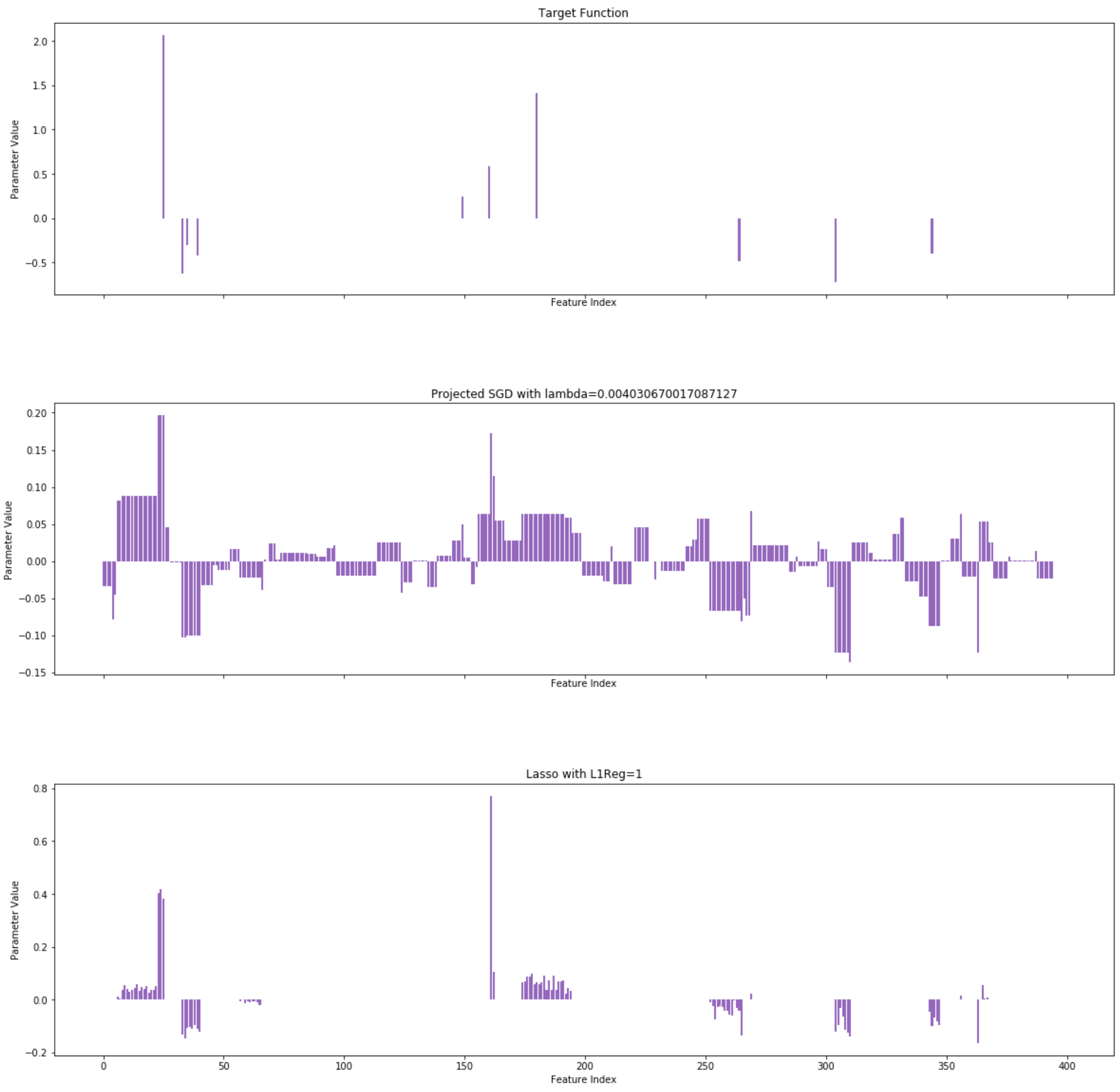
```
(SGD) Ran for 1000 epochs. Loss:6.890613957552057 Lambda: 0.00087062472369O8194
Ran for 371 epochs. Lowest loss: 16.197801391305568
```

In [231]:
```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```



From the above graph, we observe that the projected SGD has more flutuations and thus is more prone to overfitting than the shooting algorithm.

```
In [230]: compare_parameter_vectors(pred_fns);
```



We observe that both the target function and the shooting algorithm has a sparse parameter weights. The weights generated by the projected SGD algorithm is not sparse, we barely find a feature index such that the parameter valud is 0. This comparison again verifies that the lasso regression has an excellent performance on the sparse data.

Deriving λ_{max}

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

= $lim_{h->0} \frac{(Xhv-y)^T(Xhv-y)+\lambda||hv||_1-y^Ty}{h}$

= $lim_{h->0} \frac{h^2v^TX^TXv-2hv^TX^Ty+y^Ty+\lambda||hv||_1-y^Ty}{h}$

= $lim_{h->0} hv^TX^TXv - 2v^TX^Ty + \lambda||v||_1$

= $-2v^TX^Ty + \lambda||v||_1$

2. $J'(0;v) >= 0$
$-2v^TX^Ty + \lambda||v||_1 >= 0$
$\lambda >= \frac{2v^TX^Ty}{||v||_1}$

$\lambda >= \frac{2v^TX^Ty}{||v||_1}$
$\lambda||v||_1 - 2v^TX^Ty >= 0$
$J'(0;v) >= 0$

Thus, $J'(0;v) >= 0$ **iff** $\lambda >= C = \frac{2v^TX^Ty}{||v||_1}$ **for** $v \neq 0$

3.

$\frac{2v^TX^Ty}{||v||_1}$ **=** $\frac{2\sum_i v_i(X^Ty)_i}{||v||_1}$

<= $\frac{2||X^Ty||_\infty \sum_i v_i}{||v||_1}$ **<=** $\frac{2||X^Ty||_\infty ||v||_1}{||v||_1}$ **=** $2||X^Ty||_\infty$

Thus, $\lambda_{max} = 2||X^Ty||_\infty$

If $\lambda > \lambda_{max}$**, then** $J'(0,v) > 0$**, which implies that** $J(w;v)$ **increases for** $w \neq 0$**. Thus, any** $w \neq 0$ **has a greater** $J(w;v)$ **than** $w = 0$**. If** $\lambda = \lambda_{max}$**, then** $w = 0$**. If** $\lambda >= \lambda_{max} = 2||X^Ty||_\infty$**,** $w = 0$ **is the minimizer for** $J(w;v)$**.**

If $w = 0$ **is the minimizer, then** $J'(0,v) >= 0, \lambda >= \lambda_{max} = 2||X^Ty||_\infty$

In conclusion, $w = 0$ **is the minimizer of** $J(w,v)$ **iff** $\lambda >= \lambda_{max} = 2||X^Ty||_\infty$