Ridge Regression

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

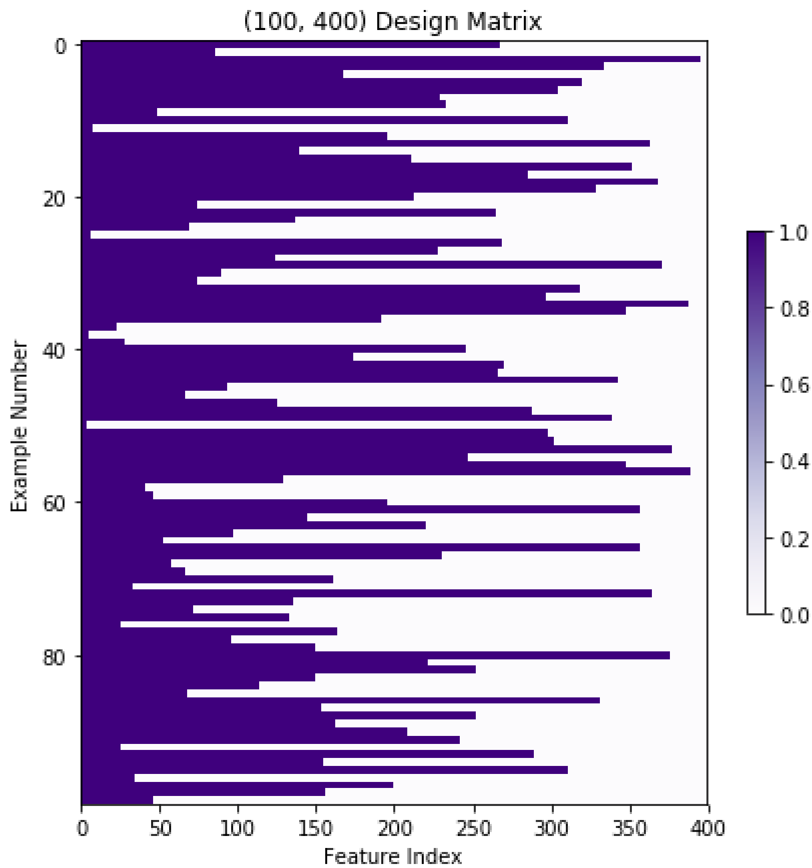
- Determine a class for the model supporting methods
  - fit
  - predict
  - score
- Search for hyperparameters through trial and error
  - evaluate the average training and validating error for each hyperparameter
- Plot the distributions of weight on the features
  - Does Ridge Regression give us sparsity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

In [1]:
```
import numpy as np
from numpy import linalg as LA
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.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

import copy

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

Question 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 [4]:
```
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_nor
m_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 bef
ore 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 bef
ore predicting data!")
                residuals = self.predict(X) - y
                return np.dot(residuals, residuals)/len(y)
```

In [5]:
```
n = X_train.shape[0]
ridge_1 = RidgeRegression(l2reg=1)
#here return the total square loss for sklearn objective function
ridge_1.fit(X_train,y_train)
ridge_1_coefs = ridge_1.w_
```

In [6]:
```
pred_val = ridge_1.predict(X_val)
print(mean_squared_error(y_val, pred_val))
```

```
0.17106788826752428
```

In [7]:
```
default_params = np.unique(np.concatenate((10.**np.arange(-6,1,1), np.ar
ange(1,3,.3))))
print(default_params)

def do_grid_search_ridge(X_train, y_train, X_val, y_val, params = defaul
t_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(tes
t_fold=val_fold),
                                        refit = True,
                                        scoring = make_scorer(me
an_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 maximiz
e,
        # 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_sco
re"]
        df_toshow = df[cols_to_keep].fillna('-')
        df_toshow = df_toshow.sort_values(by=["param_l2reg"])
        return grid, df_toshow
```

```
[1.0e-06 1.0e-05 1.0e-04 1.0e-03 1.0e-02 1.0e-01 1.0e+00 1.3e+00 1.6e+0
0
 1.9e+00 2.2e+00 2.5e+00 2.8e+00]
```

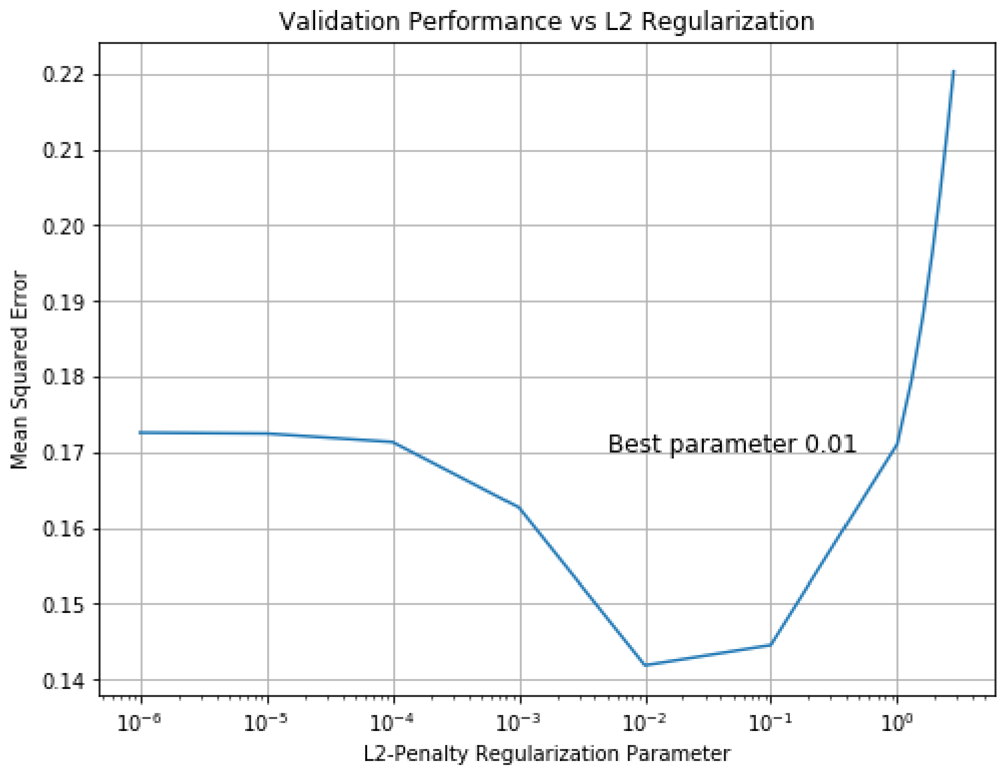
In [8]:
```
grid_1,results_1 = do_grid_search_ridge(X_train,y_train,X_val,y_val)
```

In [9]:
```
results_1
```

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")
#Make a plot with log scaling on the x axis.
ax.semilogx(results_1["param_l2reg"], results_1["mean_test_score"])
#print the best params
ax.text(0.005,0.17,"Best parameter {0}".format(grid_1.best_params_['l2re
g']), fontsize = 12);
```



In [ ]:

Question 2

Comparing to the Target Function

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

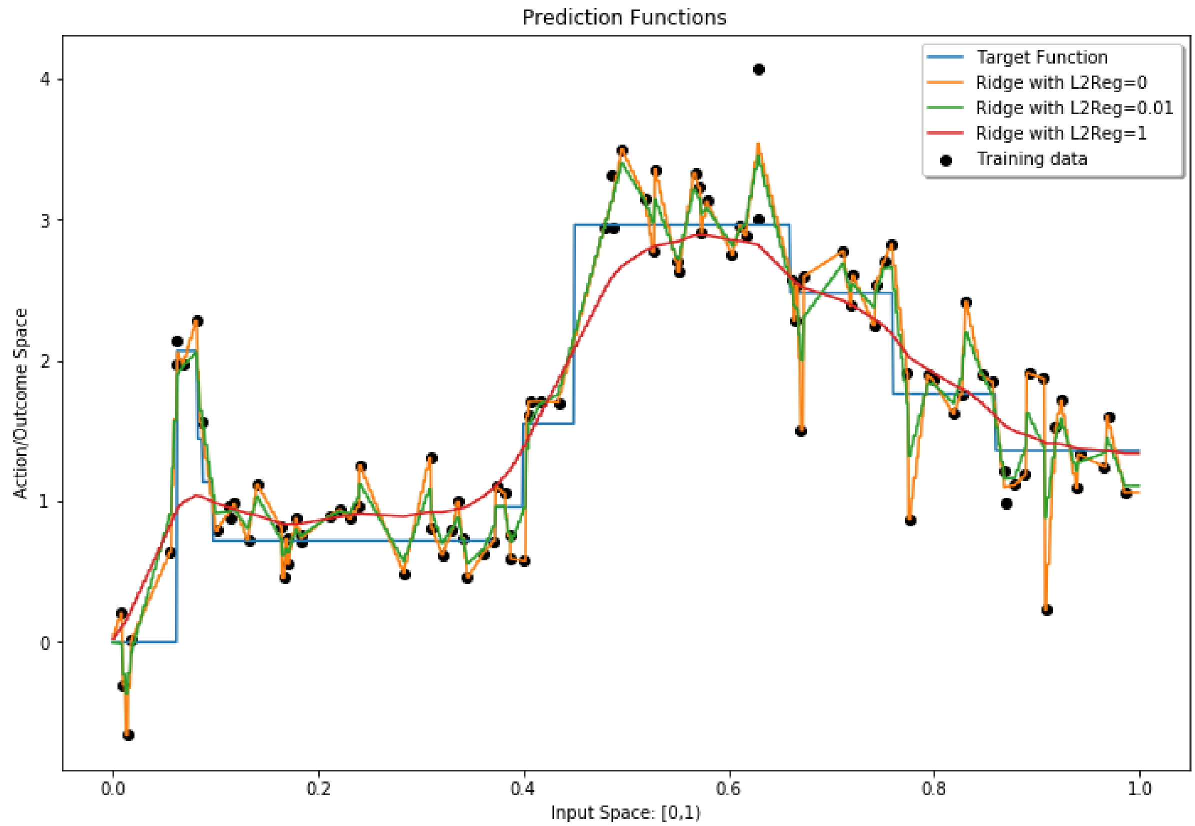
Let's create a list of dicts called `pred_fns` . Each dict has a "name" key and a "preds" key. The value corresponding to the "preds" key is an array of predictions corresponding to the input vector x. x_train and y_train are the input and output values for the training data

In [11]:
```
pred_fns=[]
x = np.sort(np.concatenate([np.arange(0,1,0.001),x_train]))
#this is the testing datasets that we havent seen it in either train or
 val

pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds"
: target_fn(x)})
# later it appends other functions like ridge with L2Reg 0,0.01,1,
# where 0 is the unregularizd, 0.01 is the best one we picked

l2regs = [0, grid_1.best_params_['l2reg'], 1] #[0,0.1,1]

X = featurize(x)
```

In [12]:
```
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 [13]:
```
def plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc=
"best"):

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

In [14]:
```
plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best");
```

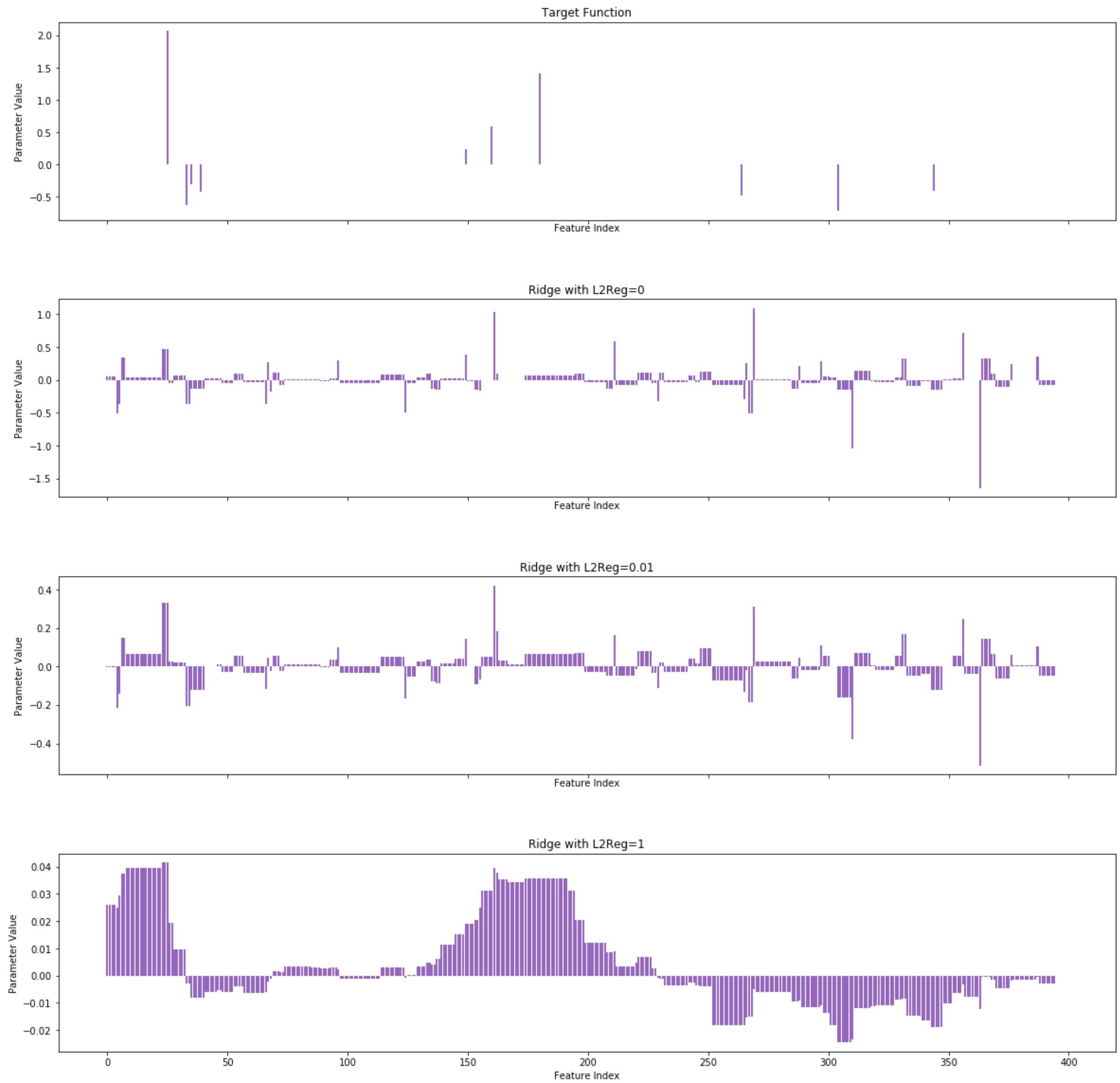


Visualizing the Weights

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

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

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

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

In [16]:
```
compare_parameter_vectors(pred_fns);
```



Paterns for these coefficients:

In the target function, only 10 features have non-zero weights.

In the non-regularized version (L2Reg=0), a lot of features have really small weights close to 0, with few (around 5) features have weights greater than 0.5 and they ae the most significant ones amongst all.

In the best one we have selected above ((L2Reg=0.01)), the features with weights that are close to in the second graph increase slightly, but the significant ones are similar to the ones above.

In the last one, where regularization (L2Reg=1), the number of significant features increase drastically compared to the above three and the magnitude increased as well.

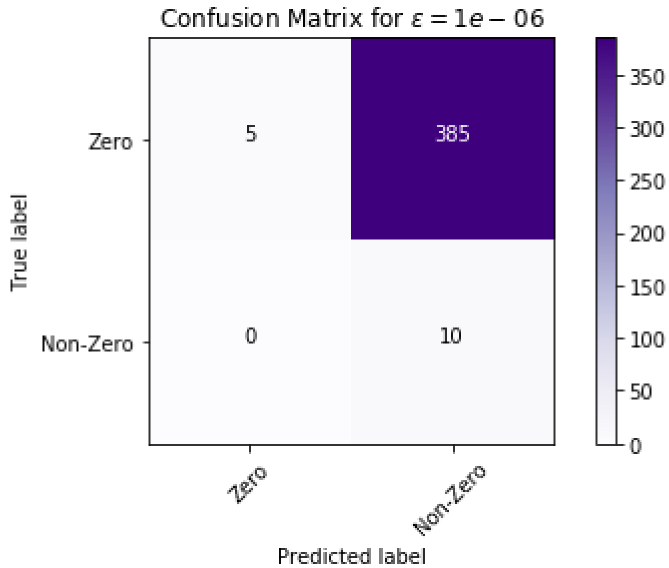
Question 3

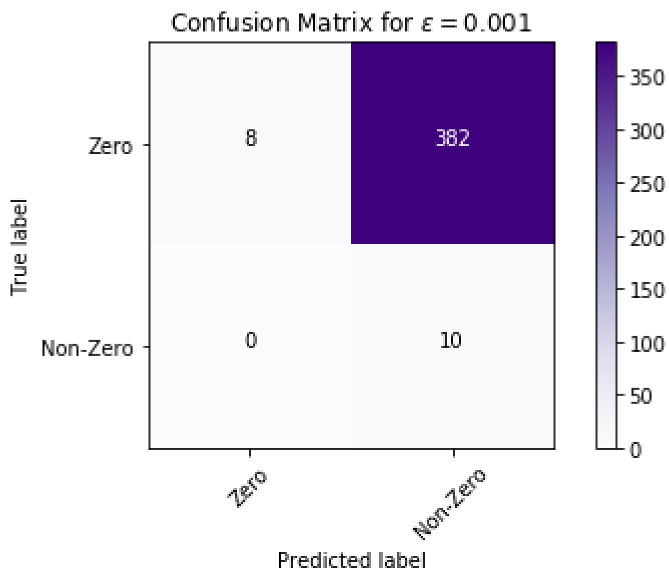
Confusion Matrix

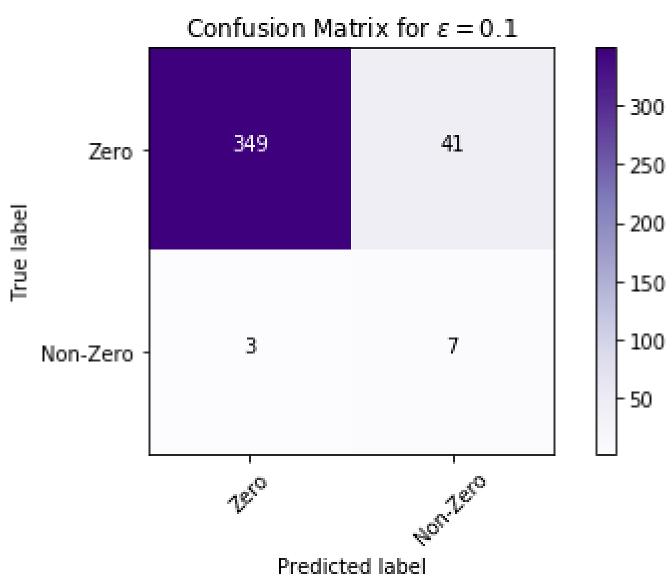
We can try to predict the features with corresponding weight zero. We will fix a threshold `eps` such that any value between `-eps` and `eps` will get counted as zero. We take the remaining features to have positive value. These predictions of can be compared to the weights for the target function.

In [17]:
```
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.shap
e[1])):
                plt.text(j, i, format(cm[i, j], 'd'),
                              horizontalalignment="center",
                              color="white" if cm[i, j] > thresh els
e "black")

        plt.tight_layout()
        plt.ylabel('True label')
        plt.xlabel('Predicted label')
```

In [18]:
```
bin_coefs_true = list(map(lambda a: 0 if abs(a)==0 else 1, pred_fns[0][
"coefs"]))  # your code goes here
eps_list = [10**-6,10**-3, 10**-1 ]# your code goes here
for eps in eps_list:
    bin_coefs_estimated = list(map(lambda a: 0 if abs(a) < eps else 1, p
red_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 $\epsi
lon = {}$".format(eps), classes=["Zero", "Non-Zero"])
```


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


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


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

Lasso Regression

Question 1

$a_i = 2X_{.j}^T X_{.j}$

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

Question 2

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 [19]:
```
def soft_threshold(a, delta):
    ####
    temp = np.abs(a) - delta
    if temp >=0:
        return np.sign(a)*temp
    else:
        return 0
    ####

def compute_sum_sqr_loss(X, y, w):
    ####
    return np.sum(np.power(np.dot(X,w)-y,2))
    ####

def compute_lasso_objective(X, y, w, l1_reg=0):
    ####
    return np.sum(np.power(np.dot(X,w)-y,2)) + l1_reg*LA.norm(w,1)
    ####

def get_ridge_solution(X, y, l2_reg):
    ####
    I = np.eye(X.shape[1])
    return np.dot(LA.inv((np.dot(X.T,X)+l2_reg*I)),np.dot(X.T,y))
    ####
```

Shooting Algorithm

In [20]:
```
def shooting_algorithm(X, y, w0=None, l1_reg = 1., max_num_epochs = 1000
, min_obj_decrease=1e-8, random=False):
    if w0 is None:
        w = np.zeros(X.shape[1])
    else:
        w = np.copy(w0)
    d = X.shape[1]
    epoch = 0
    obj_val = compute_lasso_objective(X, y, w, l1_reg)
    obj_decrease = min_obj_decrease + 1.
    while (obj_decrease>min_obj_decrease) and (epoch<max_num_epochs):
        obj_old = obj_val
        # Cyclic coordinates descent
        coordinates = range(d)
        # Randomized coordinates descent
        if random:
            coordinates = np.random.permutation(d)
        for j in coordinates:
            ####
            aj = 2*np.dot(X[:,j].T,X[:,j])
            cj = 2*(np.dot(X[:,j],y) - np.dot(X[:,j],np.dot(X,w)) + w[j]
*np.dot(X[:,j].T,X[:,j]))
            if aj==0 and cj==0:
                w[j]=0
            else:
                w[j] = soft_threshold(cj/aj, l1_reg/aj)
            ####
        obj_val = compute_lasso_objective(X, y, w, l1_reg)
        obj_decrease = abs(obj_old - obj_val)
#         print('epoch:',epoch,' \n obj_val',obj_val,'obj_decrease',obj_
decrease)
        epoch += 1
    print("Ran for "+str(epoch)+" epochs. " + 'Lowest loss: ' + str(obj_
val))
    return w
```

Class for Lasso Regression

In [21]:
```
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):
        # convert y to 1-dim array, in case we're given a column vector
        y = y.reshape(-1)
        if coef_init is None:
            coef_init = get_ridge_solution(X,y, self.l1_reg)

        ####
        # your code goes here
        self.w_ = shooting_algorithm(X, y,max_num_epochs=max_epochs,
                                     l1_reg=self.l1_reg,
                                     w0 = coef_init,
                                     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 predicti
ng 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 predicti
ng data!")

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

We can compare to the `sklearn` implementation.

In [22]:
```
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, norma
lize=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 sklea
rn's

    ####
    # your code goes here
    lasso_regression_estimator = LassoRegression(l1_reg=l1_reg,randomize
d=False)
    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.m
ean((sklearn_lasso_preds - lasso_regression_preds)**2)))
    # Let's compare differences parameter values
#     print((our_coefs - sklearn_lasso_coefs)**2)
#     print(np.sum((our_coefs - sklearn_lasso_coefs)**2))
    print("Hoping this is very close to 0 (ceofficients): {}".format(np.
sum((our_coefs - sklearn_lasso_coefs)**2)))
```

In [23]:
```
compare_our_lasso_with_sklearn(X_train, y_train, l1_reg=1.5)
```

```
Ran for 704 epochs. Lowest loss: 19.508009470245895
Hoping this is very close to 0 (predictions): 4.41237534530972e-07
Hoping this is very close to 0 (ceofficients): 3.0883607771475425
```

In [24]:
```
#cyclic; ridge optimal
w1 = shooting_algorithm(X_train, y_train, w0=None, l1_reg = 1.5,
                        max_num_epochs = 1000, min_obj_decrease=1e-8, random=
False)
#random; ridge optimal
w2 = shooting_algorithm(X_train, y_train, w0=None, l1_reg = 1.5,
                        max_num_epochs = 1000, min_obj_decrease=1e-8, random=
True)
#cyclic;initialized at 0
w3 = shooting_algorithm(X_train, y_train, w0=np.zeros(X_train.shape[1]),
l1_reg = 1.5,
                        max_num_epochs = 1000, min_obj_decrease=1e-8, random=
False)
#random; initialized at 0
w4 = shooting_algorithm(X_train, y_train, w0=np.zeros(X_train.shape[1]),
l1_reg = 1.5,
                        max_num_epochs = 1000, min_obj_decrease=1e-8, random=
True)
```

```
Ran for 811 epochs. Lowest loss: 19.508009386279696
Ran for 691 epochs. Lowest loss: 19.508009468828806
Ran for 811 epochs. Lowest loss: 19.508009386279696
Ran for 700 epochs. Lowest loss: 19.508009459126775
```

In [25]:
```
loss_1 = compute_sum_sqr_loss(X_val, y_val, w1)
loss_2 = compute_sum_sqr_loss(X_val, y_val, w2)
loss_3 = compute_sum_sqr_loss(X_val, y_val, w3)
loss_4 = compute_sum_sqr_loss(X_val, y_val, w4)
```

Question 2

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 [26]:
```
default_params_lasso = np.unique(np.concatenate((np.array([1e-3,0.01,0.1
,1,2,3]),
                                                np.linspace(0.01,0.1,5
))))
default_params_lasso
```

Out[26]:
```
array([1.00e-03, 1.00e-02, 3.25e-02, 5.50e-02, 7.75e-02, 1.00e-01,
       1.00e+00, 2.00e+00, 3.00e+00])
```

In [27]:
```
def do_grid_search_lasso(X_train, y_train, X_val, y_val,params = default
_params_lasso):
    ####
    ## 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)

    param_grid = [{'l1_reg':params}]

    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_)
    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 [28]:
```
grid_2, results_2 = do_grid_search_lasso(X_train, y_train, X_val, y_val)
```

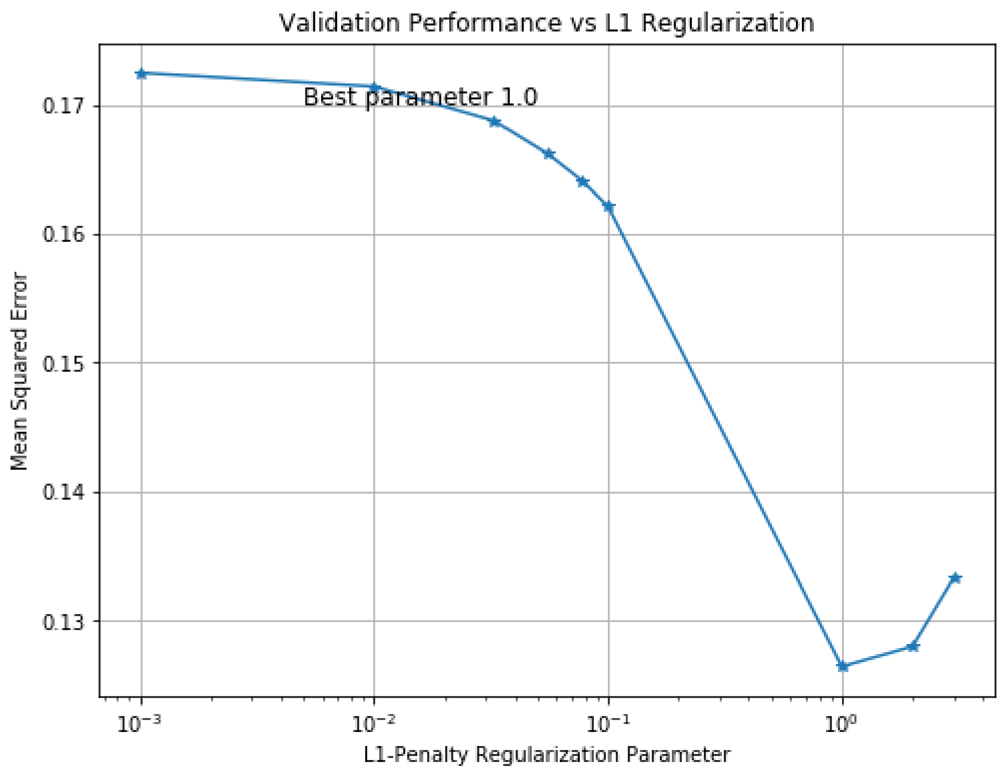
```
Ran for 231 epochs. Lowest loss: 0.7123822291805079
Ran for 571 epochs. Lowest loss: 1.042243250036044
Ran for 741 epochs. Lowest loss: 1.82926917190946
Ran for 822 epochs. Lowest loss: 2.5656234433124023
Ran for 827 epochs. Lowest loss: 3.256724835810272
Ran for 830 epochs. Lowest loss: 3.9049738772510736
Ran for 730 epochs. Lowest loss: 16.197738061447524
Ran for 644 epochs. Lowest loss: 22.624200220734313
Ran for 410 epochs. Lowest loss: 28.33425512384084
Ran for 1000 epochs. Lowest loss: 88.9939378492196
```

In [29]:
```
results_2
```

Out[29]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.0010	0.172471	0.006752
1	0.0100	0.171410	0.006806
2	0.0325	0.168757	0.007309
3	0.0550	0.166206	0.008224
4	0.0775	0.164092	0.009511
5	0.1000	0.162105	0.011143
6	1.0000	0.126440	0.091950
7	2.0000	0.127986	0.105365
8	3.0000	0.133294	0.121506

In [30]:
```
# 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_2["param_l1_reg"], results_2["mean_test_score"],mark
er= '*')
####

ax.text(0.005,0.17,"Best parameter {0}".format(grid_2.best_params_['l1_r
eg']), fontsize = 12);
```

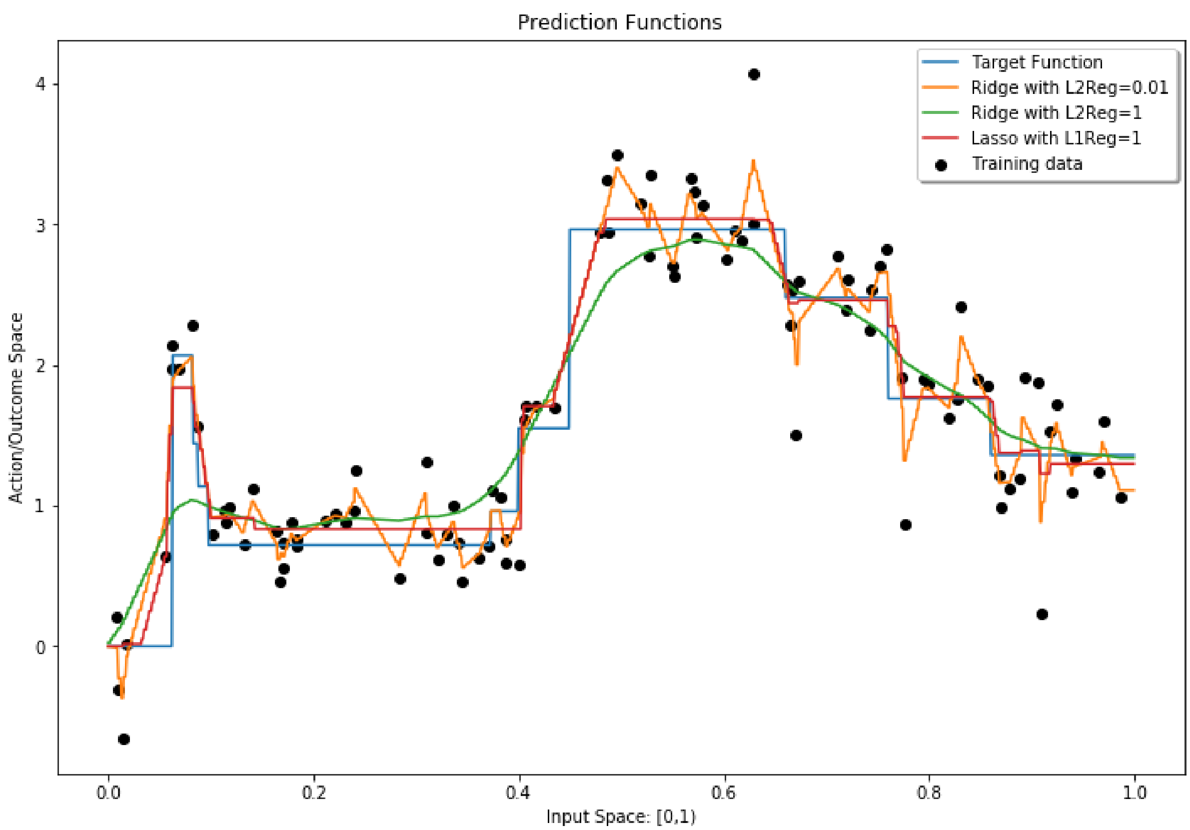


Comparing to the Target Function

In [31]:
```
pred_fns_v1 = copy.deepcopy(pred_fns)
del pred_fns_v1[1]

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

lasso_regression_estimator_v1 = LassoRegression (l1_reg=1.0)
lasso_regression_estimator_v1.fit(X_train, y_train, max_epochs = 1000, c
oef_init=None)

pred_fns_v1.append({"name":'Lasso with L1Reg=1',
                    "coefs":lasso_regression_estimator_v1.w_,
                    "preds": lasso_regression_estimator_v1.predict(X)})
```

```
Ran for 730 epochs. Lowest loss: 16.197738061447524
```

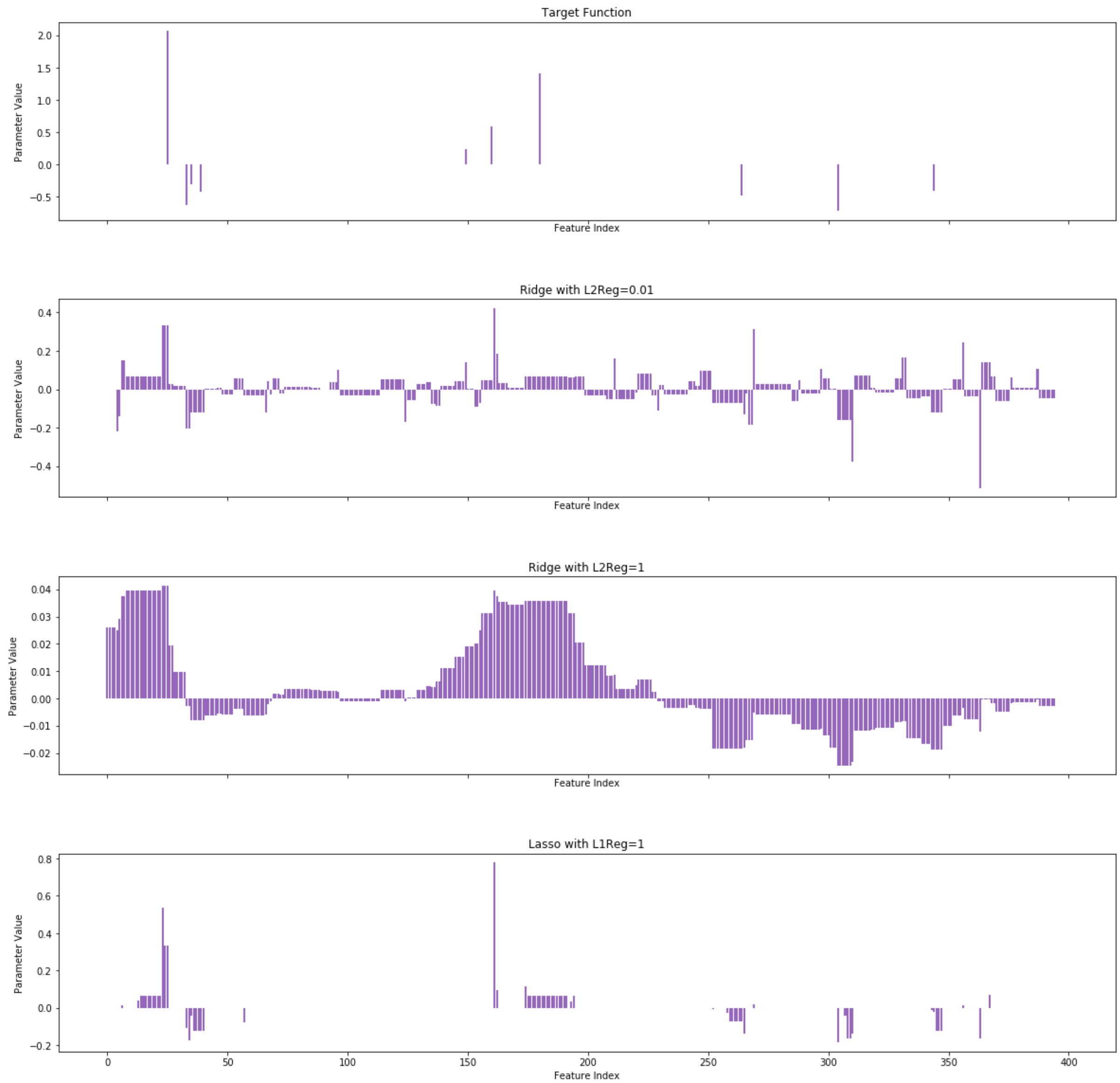
In [32]:
```
plot_prediction_functions(x, pred_fns_v1, x_train, y_train, legend_loc=
"best");
```



Visualizing the Weights

Using `pred_fns_v1` let's try to see how sparse the weights are...

In [33]:
```
compare_parameter_vectors(pred_fns_v1);
```



Comparing Lasso and Ridge:

For Ridge, there are quite a bit of insignificant variables whose params are close to 0 but not zeros.
But for Lasso, it has a function of parameters selection, where the insignificant variables will diminish to 0.

In [34]:
```
#the best ridge we found above is with reg=0.01
ridge_regression_estimator = RidgeRegression(l2reg = 0.01)
ridge_regression_estimator.fit(X_train, y_train)
ridge_mse = mean_squared_error(y_val,ridge_regression_estimator.predict(
X_val))
```

In [35]:
```
#the best lasso we found above is with reg=1
lasso_regression_estimator = LassoRegression(l1_reg = 1)
lasso_regression_estimator.fit(X_train, y_train)
lasso_mse = mean_squared_error(y_val,lasso_regression_estimator.predict(
X_val))
```

```
Ran for 730 epochs. Lowest loss: 16.197738061447524
```

In [36]:
```
print(f'mean_squared_error for Best Ridge is :{ridge_mse}\nAnd best best
Lasso is {lasso_mse}')
```

```
mean_squared_error for Best Ridge is :0.14188682939394098
And best best Lasso is 0.12643956987109234
```

The the best model I have found is Lasso with validation error of 0.1264

Question 4

Homotopy :

In [37]:
```
def homotopy(X_train,y_train,X_val,y_val):
    lambda_max = 2*LA.norm((X_train.T.dot(y_train)),ord=np.inf)
    lambda_homo = [lambda_max*0.8**i for i in range(30)]
    reg_path = dict(zip(lambda_homo,[0]*len(lambda_homo)))

    w = np.zeros(X_train.shape[1])

    for i in range(len(lambda_homo)):
        lasso_regression_estimator = LassoRegression(l1_reg = lambda_hom
o[i] ,
                                                     randomized=False)
        lasso_regression_estimator = lasso_regression_estimator.fit(X_tr
ain,y_train,coef_init=w)
        w = lasso_regression_estimator.w_
        pred_val = lasso_regression_estimator.predict(X_val)
        mse = mean_squared_error(y_val,pred_val)
        reg_path[lambda_homo[i]] = mse
    return reg_path
```

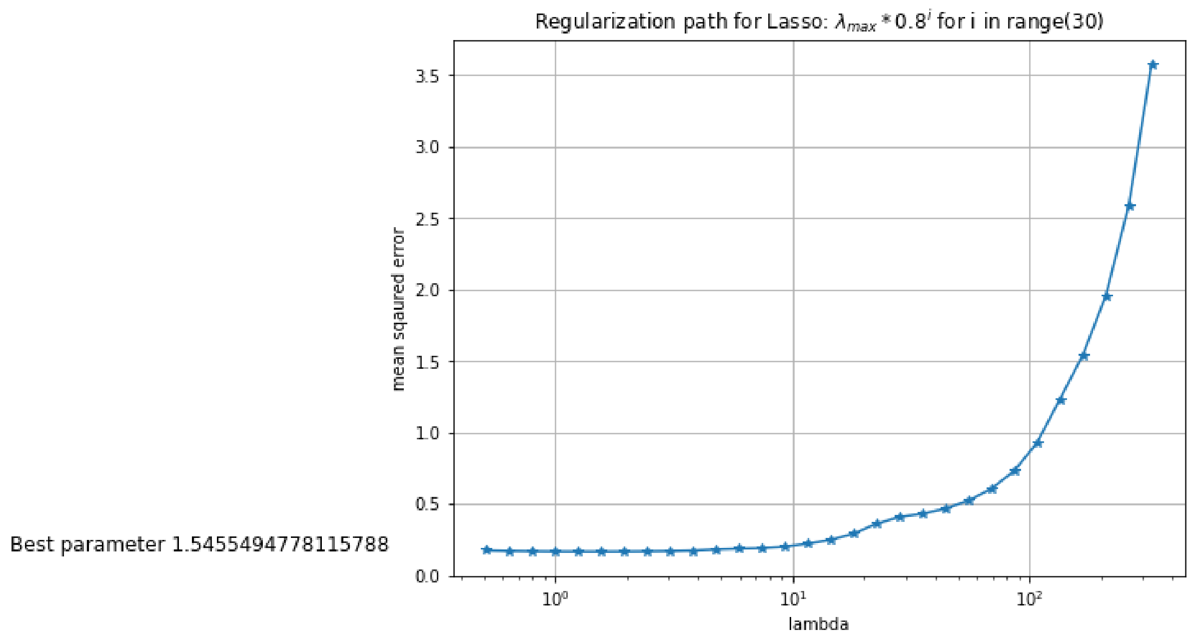
In [38]:
```
reg_path = homotopy(X_train,y_train,X_val,y_val)
```

```
Ran for 1 epochs. Lowest loss: 359.66740028131966
Ran for 594 epochs. Lowest loss: 348.52108633392805
Ran for 578 epochs. Lowest loss: 323.53716482374966
Ran for 705 epochs. Lowest loss: 293.22926472360444
Ran for 671 epochs. Lowest loss: 262.2363733206085
Ran for 136 epochs. Lowest loss: 231.30364679470637
Ran for 132 epochs. Lowest loss: 202.01748534382176
Ran for 127 epochs. Lowest loss: 175.68296871183844
Ran for 303 epochs. Lowest loss: 152.73952217465134
Ran for 319 epochs. Lowest loss: 133.12872392009874
Ran for 300 epochs. Lowest loss: 116.62091752850152
Ran for 280 epochs. Lowest loss: 102.89040503458664
Ran for 319 epochs. Lowest loss: 91.40127958050607
Ran for 351 epochs. Lowest loss: 80.59513427785576
Ran for 332 epochs. Lowest loss: 70.65131127999413
Ran for 313 epochs. Lowest loss: 61.838968974776506
Ran for 334 epochs. Lowest loss: 54.081079699361254
Ran for 320 epochs. Lowest loss: 47.3266690315174
Ran for 301 epochs. Lowest loss: 41.56428576679454
Ran for 614 epochs. Lowest loss: 36.69712945939425
Ran for 723 epochs. Lowest loss: 32.32317412430599
Ran for 700 epochs. Lowest loss: 28.434762283530574
Ran for 669 epochs. Lowest loss: 25.07153045272604
Ran for 643 epochs. Lowest loss: 22.211091350772826
Ran for 657 epochs. Lowest loss: 19.7996973823722
Ran for 630 epochs. Lowest loss: 17.789509032870527
Ran for 603 epochs. Lowest loss: 16.121413497283267
Ran for 604 epochs. Lowest loss: 14.563979468681882
Ran for 575 epochs. Lowest loss: 12.993126809546268
Ran for 543 epochs. Lowest loss: 11.487542271474759
```

In [39]:
```
print('The regularization path and its mean squared error is:\n')
for item in reg_path.items():
    print(item)
print('-'*50,'\n')
best_lambda = min(reg_path, key=reg_path.get)
print(f'The best lambda is {best_lambda}')
```

```
The regularization path and its mean squared error is:

(327.28283232952117, 3.576552934309347 6)
(261.8262658636169, 2.584476900524163)
(209.4610126908936, 1.9538114962054285)
(167.5688101527149, 1.54000559128969)
(134.05504812217188, 1.2307521610707755)
(107.24403849773752, 0.9251205234175048)
(85.79523079819003, 0.7294772775859948)
(68.63618463855202, 0.604231046447266)
(54.90894771084163, 0.5220217532583166)
(43.9271581686733, 0.466784939825105)
(35.14172653493864, 0.4315616203623035)
(28.113381227950914, 0.40912026313874783)
(22.490704982360732, 0.3608077627666701)
(17.992563985888587, 0.290753649077262)
(14.394051188710872, 0.24902881746021532)
(11.515240950968698, 0.22249068710193642)
(9.212192760774958, 0.20131902568814863)
(7.369754208619968, 0.1915081367166798)
(5.895803366895974, 0.187244309616775)
(4.716642693516779, 0.18237640138872596)
(3.773314154813424, 0.17434063310518713)
(3.0186513238507393, 0.17056928068733704)
(2.4149210590805916, 0.1686413537041543)
(1.9319368472644733, 0.16746756396565268)
(1.5455494778115788, 0.1671412710375244)
(1.236439582249263, 0.1674143483395599)
(0.9891516657994105, 0.1678281566686209)
(0.7913213326395284, 0.16922603990680943)
(0.6330570661116228, 0.17169935842072526)
(0.5064456528892983, 0.174975403414017)
--------------------------------------------------

The best lambda is 1.5455494778115788
```

In [40]:
```
fig,ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title('Regularization path for Lasso: $\lambda_{max}*0.8^i$ for i in range(30)')
ax.set_xlabel('lambda')
ax.set_ylabel('mean sqaured error')
# ax.plot(reg_path.keys(),reg_path.values(),marker = '*')
ax.semilogx(reg_path.keys(),reg_path.values(),marker = '*')
ax.text(0.005,0.17,"Best parameter {0}".format(best_lambda), fontsize = 12);
```

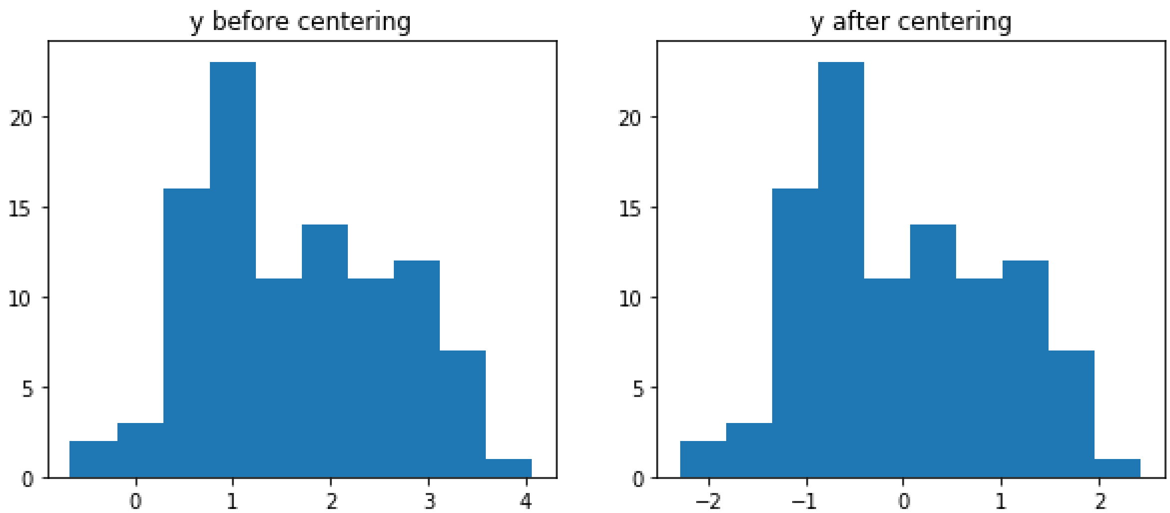


Question 5:

Method 1: Center y

In [41]:
```
y_train_center = y_train-np.mean(y_train)
y_val_center = y_val-np.mean(y_train)
```

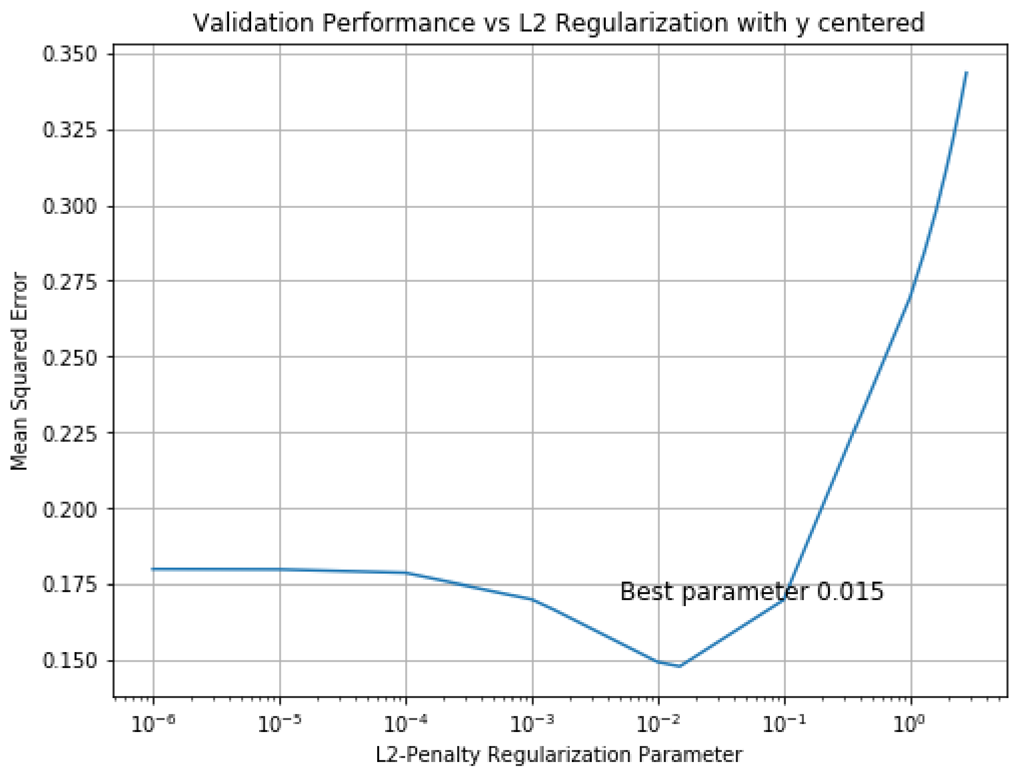
In [42]:
```
fig = plt.subplots(figsize = (10,4))

plt.subplot(1, 2, 1)
plt.hist(y_train)
plt.title('y before centering')

plt.subplot(1, 2, 2)
plt.hist(y_train_center)
plt.title('y after centering')

plt.show()
```



Ridge Grid Search based on centered y

In [43]:
```
ridge_params_1 = np.unique(np.concatenate((10.**np.arange(-6,1,1),np.arr
ay([0.0015,0.015]) ,
                                           np.arange(1,3,.3))))
grid_3, results_3 = do_grid_search_ridge(X_train, y_train_center,
                                         X_val, y_val_center,
                                         params = ridge_params_1)
```

In [44]:
```
# Plot validation performance vs regularization parameter
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L2 Regularization with y centere
d")
ax.set_xlabel("L2-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")

####
## your code goes here
ax.semilogx(results_3["param_l2reg"], results_3["mean_test_score"])
####

ax.text(0.005,0.17,"Best parameter {0}".format(grid_3.best_params_['l2re
g']), fontsize = 12);
```



In [45]:
```
results_3
```

Out[45]:

	param_l2reg	mean_test_score	mean_train_score
0	0.000001	0.179844	0.006752
1	0.000010	0.179730	0.006752
2	0.000100	0.178616	0.006773
3	0.001000	0.169996	0.008264
4	0.001500	0.166603	0.009631
5	0.010000	0.149334	0.032714
6	0.015000	0.147861	0.041963
7	0.100000	0.169800	0.112772
8	1.000000	0.269492	0.263542
9	1.300000	0.284440	0.282292
10	1.600000	0.297735	0.298239
11	1.900000	0.310046	0.312516
12	2.200000	0.321682	0.325676
13	2.500000	0.332805	0.338013
14	2.800000	0.343509	0.349706

In [46]:
```
print('lowest validation loss before y centered: \n',min(results_1.loc
[:,'mean_test_score']))
print('lowest validation loss after y centered: \n',min(results_3.loc[:,
'mean_test_score']))
```

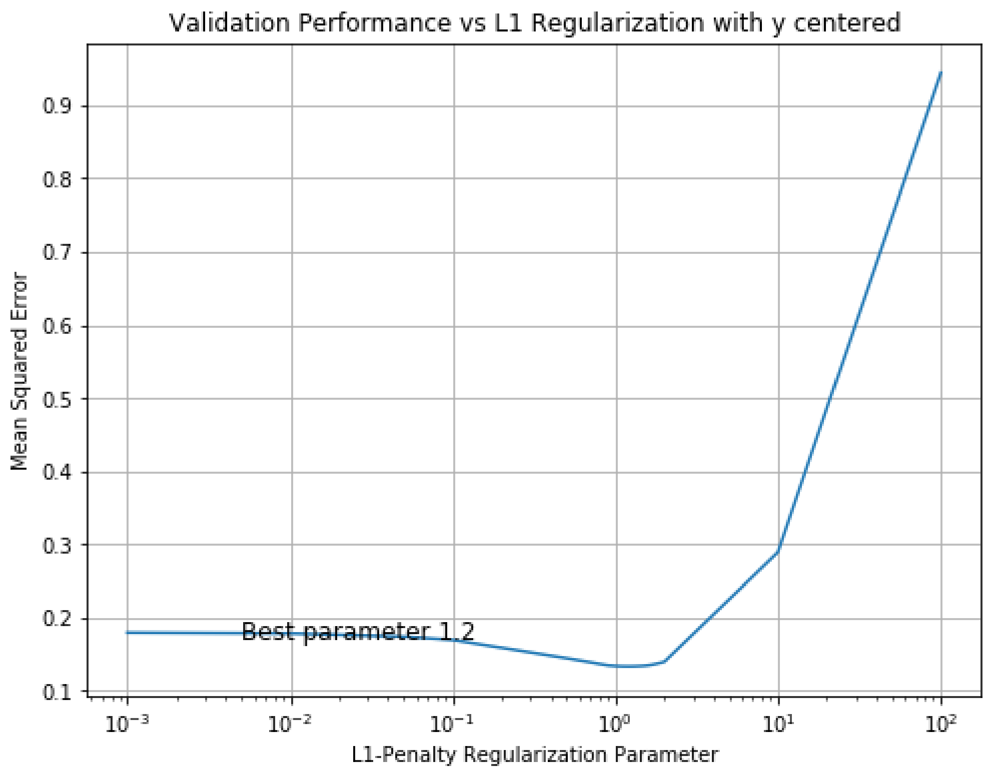
```
lowest validation loss before y centered:
 0.14188682939394098
lowest validation loss after y centered:
 0.1478608087150924
```

Lasso Grid Search based on centered y

In [47]:
```
params_lasso_v1 = [1e-3,1e-2,0.01,0.05,0.1,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2
,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2,10,100]
grid_4, results_4 = do_grid_search_lasso(X_train, y_train_center,
                                         X_val, y_val_center,
                                         params = params_lasso_v1)
```

```
Ran for 230 epochs. Lowest loss: 0.7136003976829023
Ran for 572 epochs. Lowest loss: 1.0545148177740025
Ran for 572 epochs. Lowest loss: 1.0545148177740025
Ran for 808 epochs. Lowest loss: 2.4693777019593357
Ran for 830 epochs. Lowest loss: 4.036689597165474
Ran for 843 epochs. Lowest loss: 12.176048013718372
Ran for 845 epochs. Lowest loss: 13.557993212205965
Ran for 847 epochs. Lowest loss: 14.795006741144213
Ran for 824 epochs. Lowest loss: 15.901680175891173
Ran for 803 epochs. Lowest loss: 16.888675702839947
Ran for 691 epochs. Lowest loss: 17.78352321994182
Ran for 697 epochs. Lowest loss: 18.621016053951923
Ran for 702 epochs. Lowest loss: 19.438774666245465
Ran for 706 epochs. Lowest loss: 20.242324404681675
Ran for 710 epochs. Lowest loss: 21.031665260340482
Ran for 713 epochs. Lowest loss: 21.806797243790704
Ran for 545 epochs. Lowest loss: 22.567832737633122
Ran for 540 epochs. Lowest loss: 23.315992387507112
Ran for 528 epochs. Lowest loss: 24.05160986378963
Ran for 515 epochs. Lowest loss: 24.77468512262412
Ran for 493 epochs. Lowest loss: 25.48521814126093
Ran for 348 epochs. Lowest loss: 58.41505764686252
Ran for 3 epochs. Lowest loss: 94.35424556084891
Ran for 1000 epochs. Lowest loss: 93.5465628712471
```

In [48]:
```
# Plot validation performance vs regularization parameter
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L1 Regularization with y centere
d")
ax.set_xlabel("L1-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")

####
## your code goes here
ax.semilogx(results_4["param_l1_reg"], results_4["mean_test_score"])
####

ax.text(0.005,0.17,"Best parameter {0}".format(grid_4.best_params_['l1_r
eg']), fontsize = 12);
```



In [49]:
```
results_4
```

Out[49]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.001	0.179746	0.006752
1	0.010	0.178728	0.006804
2	0.010	0.178728	0.006804
3	0.050	0.174250	0.007963
4	0.100	0.169786	0.011043
5	0.500	0.144763	0.048494
6	0.600	0.141765	0.057259
7	0.700	0.139246	0.066048
8	0.800	0.136921	0.075442
9	0.900	0.135010	0.085135
10	1.000	0.134176	0.091880
11	1.100	0.133748	0.095473
12	1.200	0.133656	0.097107
13	1.300	0.133755	0.098884
14	1.400	0.134045	0.100802
15	1.500	0.134429	0.102862
16	1.600	0.135131	0.104963
17	1.700	0.136171	0.107032
18	1.800	0.137357	0.109226
19	1.900	0.138595	0.111545
20	2.000	0.140015	0.113989
21	10.000	0.289350	0.296943
22	100.000	0.944463	0.943542

In [50]:
```
print('lowest validation loss before y centered: \n',min(results_2.loc
[:,'mean_test_score']))
print('lowest validation loss after y centered: \n',min(results_4.loc[:,
'mean_test_score']))
```

```
lowest validation loss before y centered:
 0.12643956987109234
lowest validation loss after y centered:
 0.1336556811488883
```

Projected Gradient Descent

Question 1:

In [61]:
```
def compute_sqaure_loss(X,y,theta):
    return np.mean((np.dot(X,theta)-y)**2)
```

In [62]:
```
def compute_gradient_plus(X, y, theta_plus,theta_minus,l1_reg):
    m = len(X)
    grad = 1/m*l1_reg + 2/m*(np.dot(X.T, (np.dot(X,theta_plus)-np.dot(X,
theta_minus)-y)))
    return grad
```

In [63]:
```
def compute_gradient_minus(X, y, theta_plus,theta_minus,l1_reg):
    m = len(X)
    grad = 1/m*l1_reg - 2/m*(np.dot(X.T, (np.dot(X,theta_plus)-np.dot(X,
theta_minus)-y)))
    return grad
```

In [100]:
```
def Pojected_SGD(X, y, alpha=0.05,l1_reg = 1., max_num_epochs = 1000, mi
n_obj_decrease=1e-8):
    num_instances,num_features = X.shape[0],X.shape[1]
    theta = np.zeros(num_features)

#     theta_hist = np.zeros((max_num_epochs,num_features))
#     loss_hist = np.zeros(max_num_epochs)

    t=1 # becaseu we need to cal 1/t
    n=0
    C=0.1

    obj_val = compute_lasso_objective(X, y, theta, l1_reg)
    obj_decrease = min_obj_decrease + 1.

    while (obj_decrease>min_obj_decrease) and (n < max_num_epochs):
        for _ in range(num_instances):
            obj_old = obj_val

            #pick random x given it is stochastic
            i = np.random.randint(0,num_instances)

            if alpha == "1/sqrt(t)":
                alpha = C/np.sqrt(t)
            if alpha == "1/t":
                alpha = C/t
            if isinstance(alpha,float):
                alpha = alpha
            else:
                raise Exception("Sorry, alpha type wrong")

            theta_plus = np.where(theta>=0,theta,0)
            theta_minus = np.where(theta<=0,-theta,0)

            #cal grad for theta plus and minus
            grad_plus = compute_gradient_plus(np.array(X[i,:]),np.array(
y[i]),theta_plus,theta_minus,l1_reg)
            grad_minus = compute_gradient_minus(np.array(X[i,:]),np.arra
y(y[i]),theta_plus,theta_minus,l1_reg)

            #update and projection: if theta_i is less than 0, we projec
t it to 0
            theta_plus -= alpha * grad_plus
            theta_plus = np.maximum(theta_plus,0)

            theta_minus -= alpha * grad_minus
            theta_minus = np.maximum(theta_minus,0)

            #cal theta
            theta = np.array(theta_plus) - np.array(theta_minus)

            loss = compute_sqaure_loss(X[i,:],y[i],theta)

            obj_val = compute_lasso_objective(X, y, theta, l1_reg)
            obj_decrease = abs(obj_old - obj_val)
            t+=1
```

```
        n+=1
    print(f'number of epoch: {n-1}, minimum loss: {loss}')
    return theta
```

In [101]:
```
class LassoRegression_proj_SGD(BaseEstimator, RegressorMixin):
    """ Lasso regression"""
    def __init__(self, l1_reg=1.0, alpha="1/t"):
        if l1_reg < 0:
            raise ValueError('Regularization penalty should be at least 
 0.')
        self.l1_reg = l1_reg
        self.alpha = alpha


    def fit(self, X, y, max_epochs = 1000,):
        # convert y to 1-dim array, in case we're given a column vector
        y = y.reshape(-1)
        ####
        # your code goes here
        self.w_ = Pojected_SGD(X, y,
                               max_num_epochs = max_epochs,alpha=self.al
pha,
                               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 predicti
ng 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 predicti
ng data!")

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

In [102]:
```
def do_grid_search_lasso_projected(X_train, y_train, X_val, y_val,params
= default_params_lasso):
    ####
    ## 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)

    param_grid = [{'l1_reg':params}]

    lasso_regression_estimator = LassoRegression_proj_SGD()
    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_)
    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 [103]:
```
grid_5, results_5 = do_grid_search_lasso_projected(X_train, y_train, X_v
al, y_val,params = default_params_lasso)
```

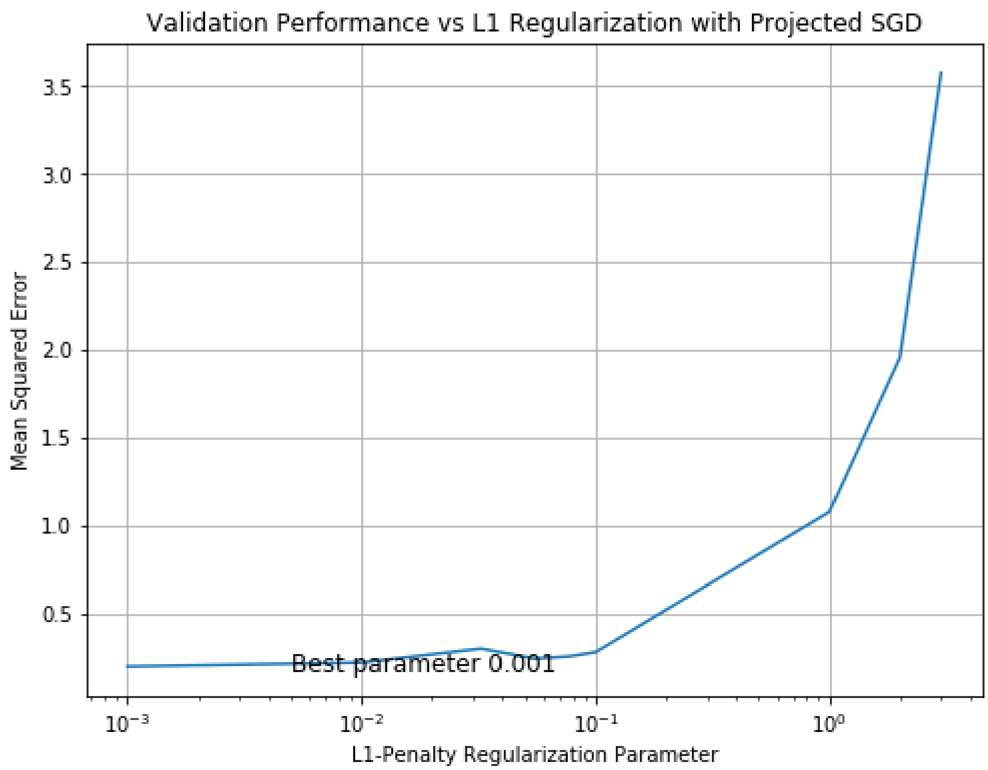
```
number of epoch: 999, minimum loss: 1.298489098022087e-05
number of epoch: 999, minimum loss: 1.2462239619669872e-05
number of epoch: 999, minimum loss: 0.4810186743060247
number of epoch: 999, minimum loss: 0.008237254982516932
number of epoch: 999, minimum loss: 0.26198852459051875
number of epoch: 999, minimum loss: 0.03893088276147973
number of epoch: 999, minimum loss: 0.05792662558129987
number of epoch: 999, minimum loss: 0.3942755818528236
number of epoch: 40, minimum loss: 0.9204745591751082
number of epoch: 999, minimum loss: 0.00846508561492384
```

In [104]:
```
results_5
```

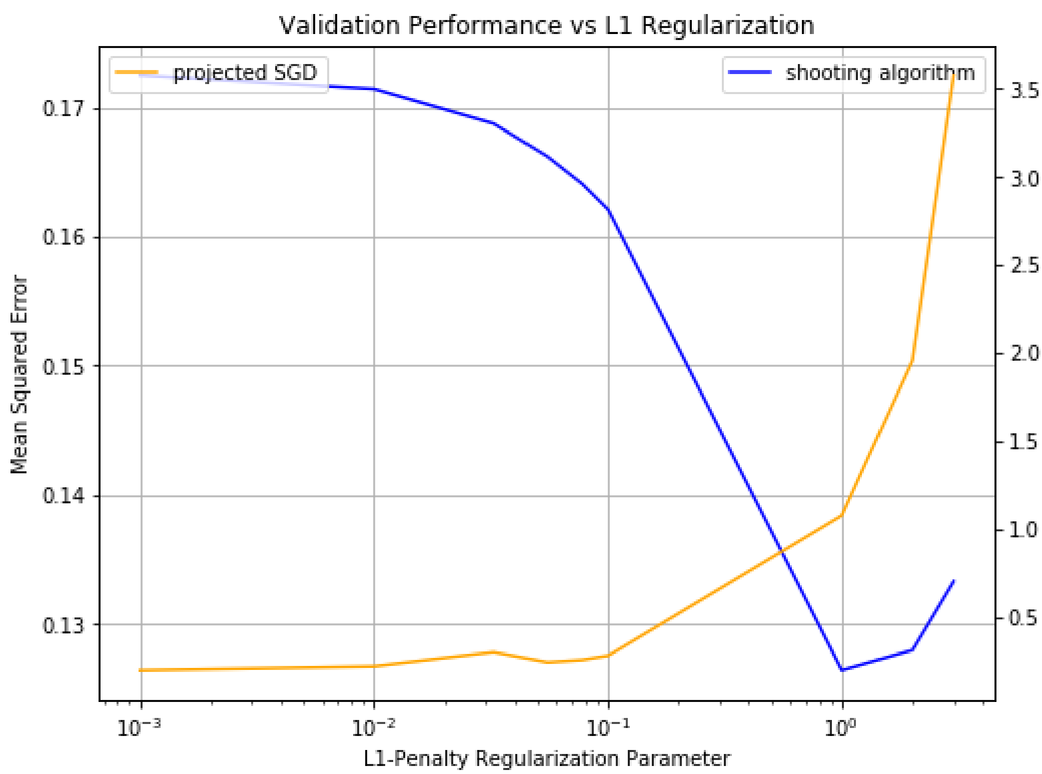
Out[104]:

	param_l1_reg	mean_test_score	mean_train_score
0	0.0010	0.196479	0.218404
1	0.0100	0.218748	0.255930
2	0.0325	0.297967	0.347254
3	0.0550	0.240658	0.291885
4	0.0775	0.254115	0.304381
5	0.1000	0.277917	0.342583
6	1.0000	1.074738	1.108979
7	2.0000	1.956401	1.975677
8	3.0000	3.576553	3.596674

In [105]:
```
# Plot validation performance vs regularization parameter
fig, ax = plt.subplots(figsize = (8,6))
ax.grid()
ax.set_title("Validation Performance vs L1 Regularization with Projected SGD")
ax.set_xlabel("L1-Penalty Regularization Parameter")
ax.set_ylabel("Mean Squared Error")

####
## your code goes here
ax.semilogx(results_5["param_l1_reg"], results_5["mean_test_score"])
####

ax.text(0.005,0.17,"Best parameter {0}".format(grid_5.best_params_['l1_reg']), fontsize = 12);
```



In [113]:
```
# Plot validation performance vs regularization parameter
fig, ax1 = plt.subplots(figsize = (8,6))
ax1.grid()
ax1.set_title("Validation Performance vs L1 Regularization")
ax1.set_xlabel("L1-Penalty Regularization Parameter")
ax1.set_ylabel("Mean Squared Error")

####
## your code goes here
ax1.semilogx(results_2["param_l1_reg"], results_2["mean_test_score"], la
bel ='shooting algorithm',color='blue')

ax2 = ax1.twinx()
ax2.semilogx(results_5["param_l1_reg"], results_5["mean_test_score"],lab
el = 'projected SGD',color='orange')

ax1.legend();
ax2.legend();
####

# ax.text(0.005,0.17,"Best parameter {0}".format(grid_5.best_params_['l1
_reg']), fontsize = 12);
```



Difference:

In [140]:
```
results_5["mean_test_score"] -  results_2["mean_test_score"]
```

Out[140]:
```
0    0.024009
1    0.047339
2    0.129210
3    0.074452
4    0.090023
5    0.115812
6    0.948298
7    1.828415
8    3.443259
Name: mean_test_score, dtype: float64
```

Question 2:

Based on the grid search, the best λ is 0.01

In [117]:
```
theta_psgd= Pojected_SGD(X_train, y_train,
                                    alpha="1/t",l1_reg = 0.01,
                                    max_num_epochs = 1000, min_obj_decre
ase=1e-8)
```

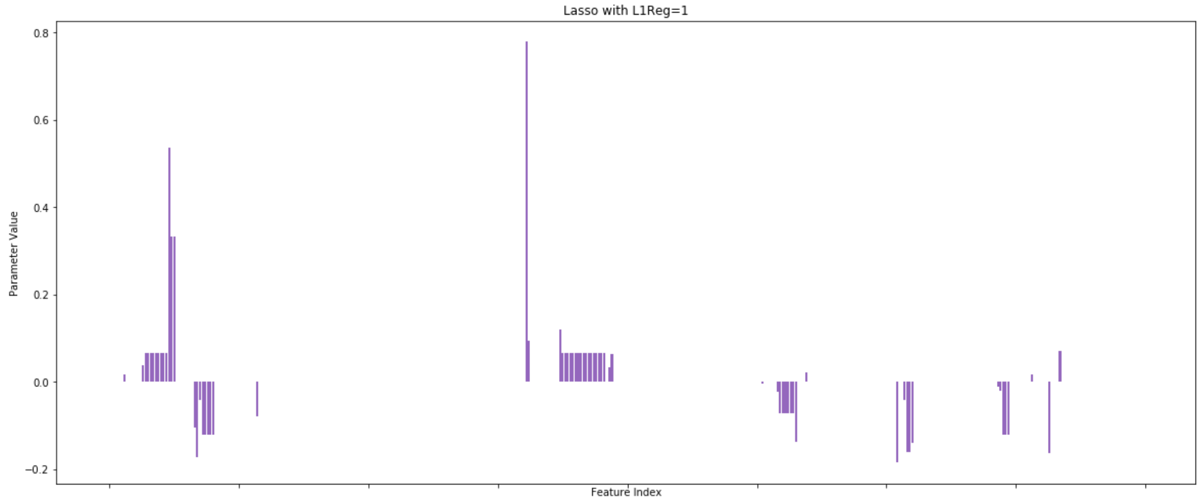
```
number of epoch: 999, minimum loss: 0.011373823760189755
```

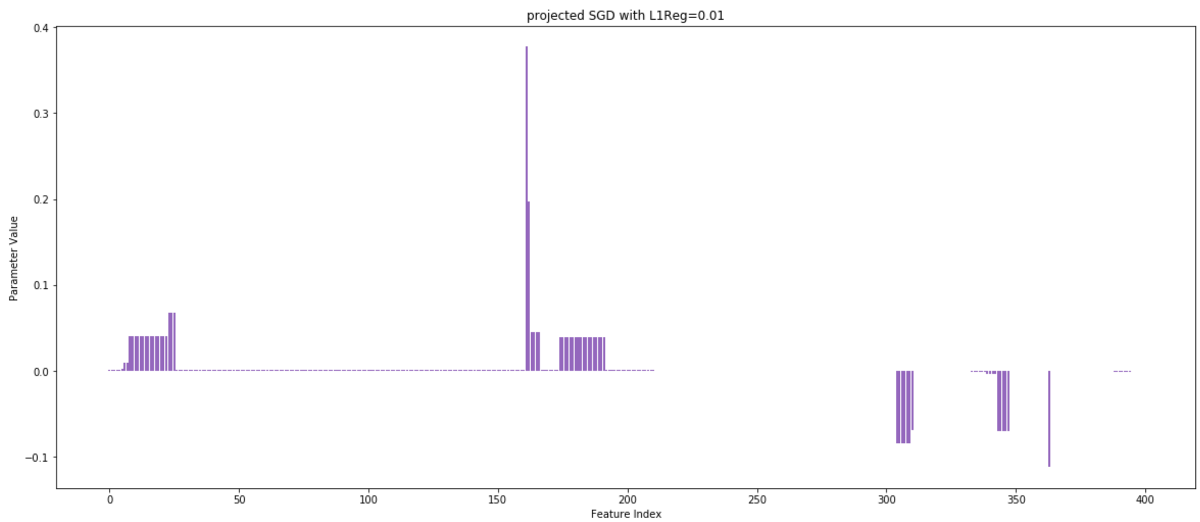
In [137]:
```
pred_fns_v2 = copy.deepcopy(pred_fns_v1)
del pred_fns_v2[0]
del pred_fns_v2[0]
del pred_fns_v2[0]
```

In [138]:
```
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))
X = featurize(x)

lasso_regression_estimator_v2 = LassoRegression_proj_SGD(l1_reg=0.01,alp
ha="1/t")
lasso_regression_estimator_v2.fit(X_train, y_train, max_epochs = 1000)

pred_fns_v2.append({"name":'projected SGD with L1Reg=0.01',
                     "coefs":lasso_regression_estimator_v2.w_,
                     "preds": lasso_regression_estimator_v2.predict(X)})
```

```
number of epoch: 999, minimum loss: 0.0030668228986437066
```

In [139]:
```
compare_parameter_vectors(pred_fns_v2);
```





Except for the significant ones, most of the coefficients are 0.
Compared to the shooring algorithm, the sparsity are pretty much similar between them two based on the two graph above.

In [ ]: