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'
```

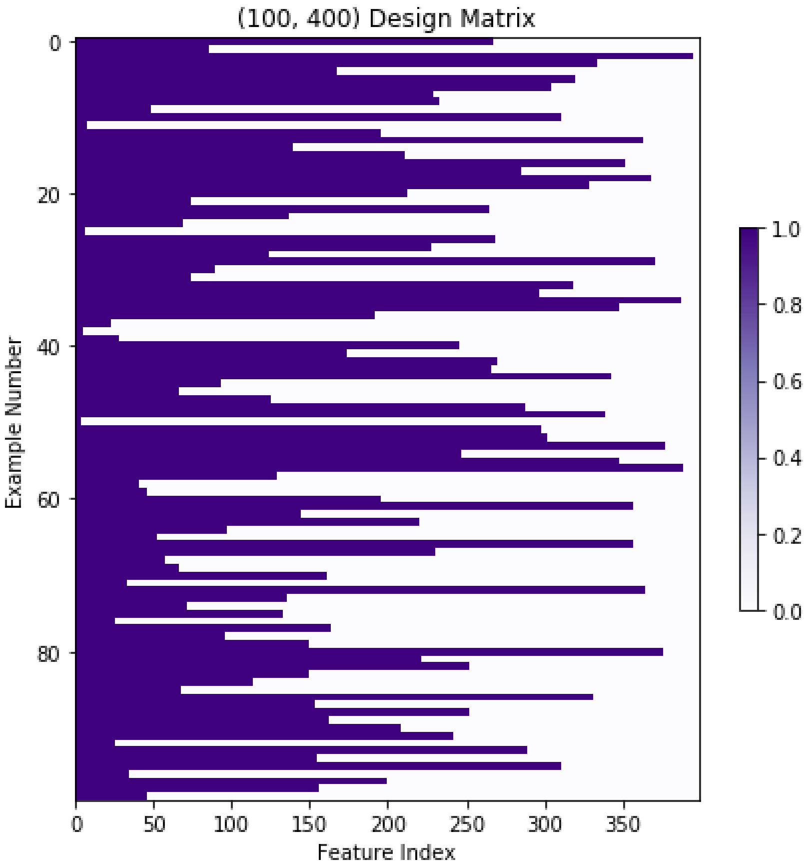
Dataset

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 [6]:

```
#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 [12]:

```
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 [13]:

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

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

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

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

In [14]:

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

```
Hoping this is very close to 0:4.6933160195738277e-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 [15]:

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

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

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

        param_grid = [{'l2reg':params}]

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

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

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

In [16]:

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

In [17]:

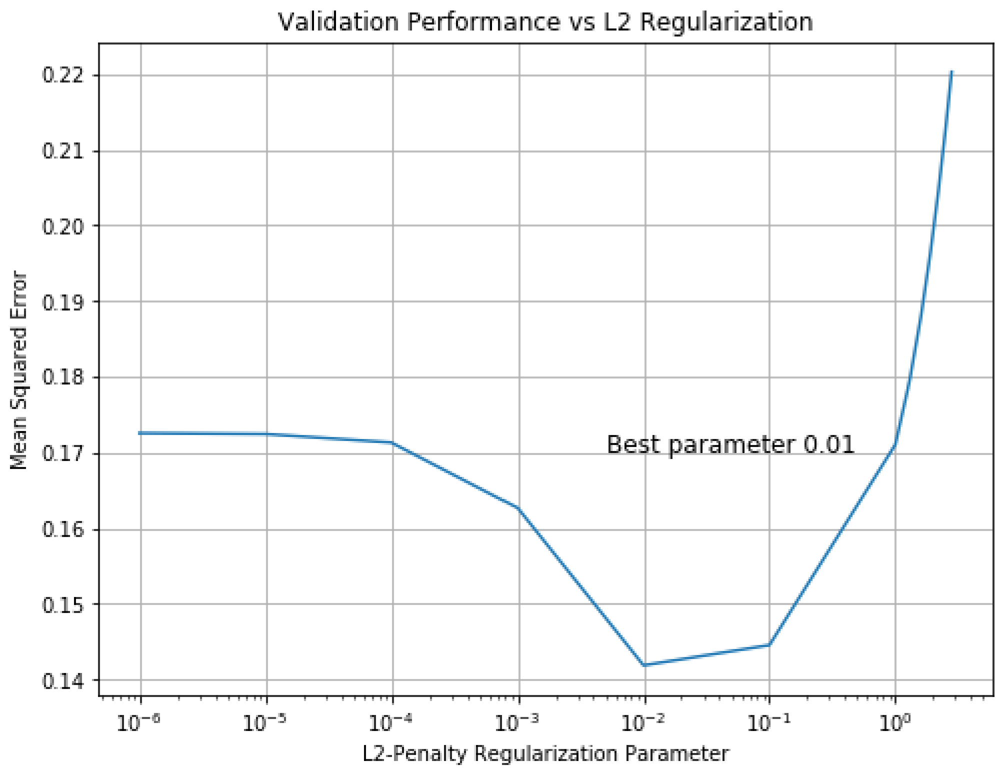
```
results
```

Out[17]:

	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 [18]:

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



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 [19]:

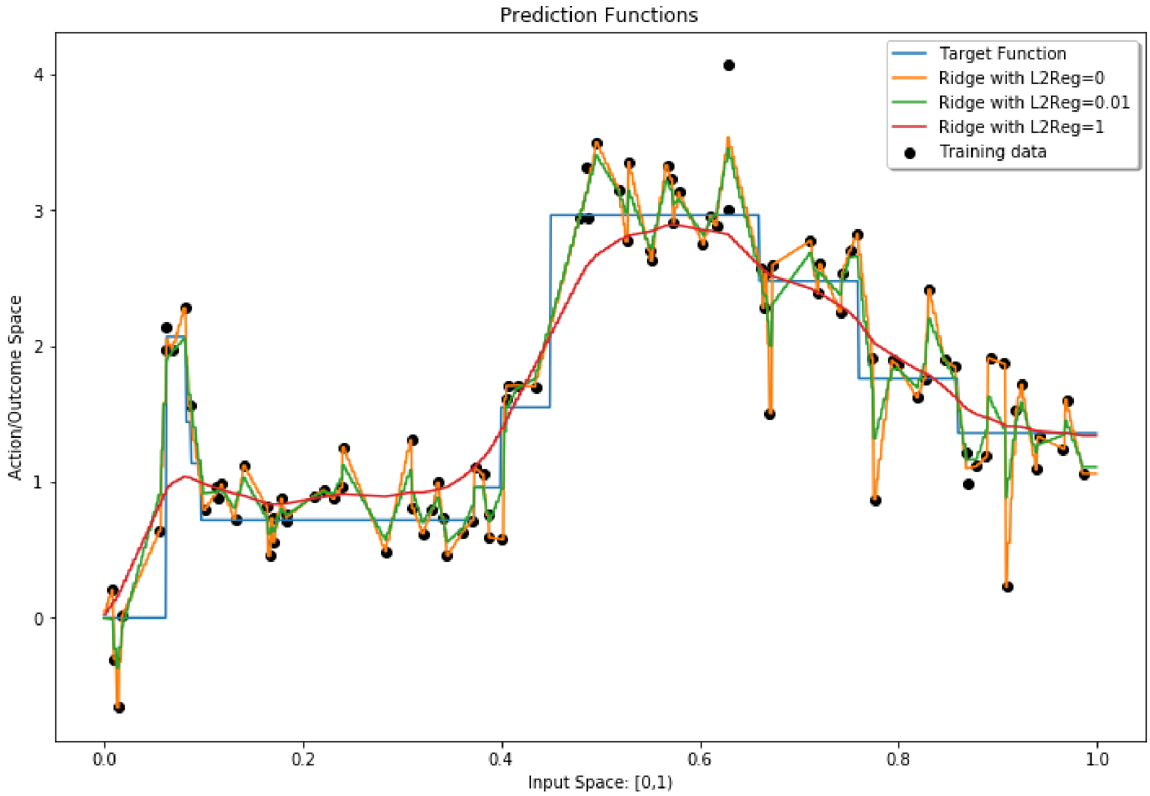
```
pred_fns = []
x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))

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 [20]:

```
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 [21]:

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



Visualizing the Weights

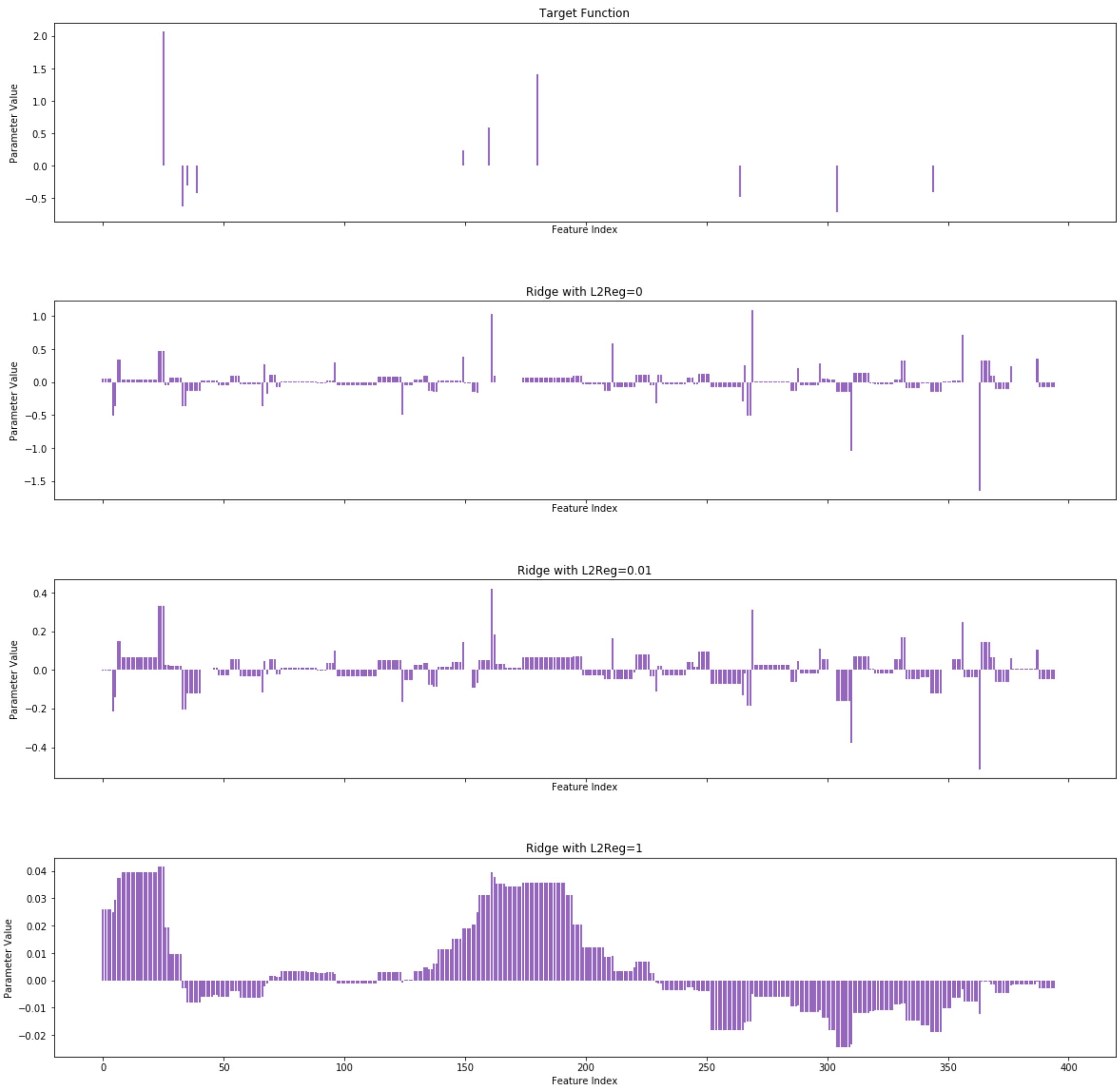
Using `pred_fns` let's try to see how sparse the weights are...

In [22]:

```
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 [23]:

```
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 [24]:

```
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 "blac
k")

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

Q1

In [62]:

```
lambdas = [1e-5,1e-4,1e-3,1e-2,1e-1,1,10]
losses = []
from tqdm import tqdm

for lambd in lambdas:
    model = RidgeRegression(lambd)
    model.fit(X_train,y_train)
    losses.append(model.score(X_val,y_val))
best_param = lambdas[np.argmin(np.array(losses))]
```

In [63]:

```
result = pd.DataFrame({"lambda":lambdas,"avg loss on val set":losses})
result
```

Out[63]:

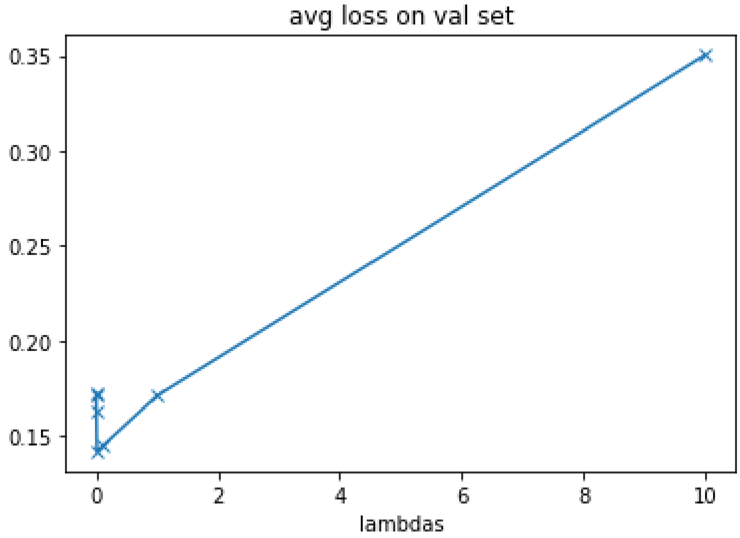
	lambda	avg loss on val set
0	0.00001	0.172464
1	0.00010	0.171345
2	0.00100	0.162705
3	0.01000	0.141887
4	0.10000	0.144566
5	1.00000	0.171068
6	10.00000	0.350321

In [64]:

```
plt.plot(lambdas,losses,"x-")
plt.xlabel("lambdas")
plt.title("avg loss on val set")
```

Out[64]:

```
Text(0.5, 1.0, 'avg loss on val set')
```



In [65]:

```
print("Best parameter is: ", best_param)
```

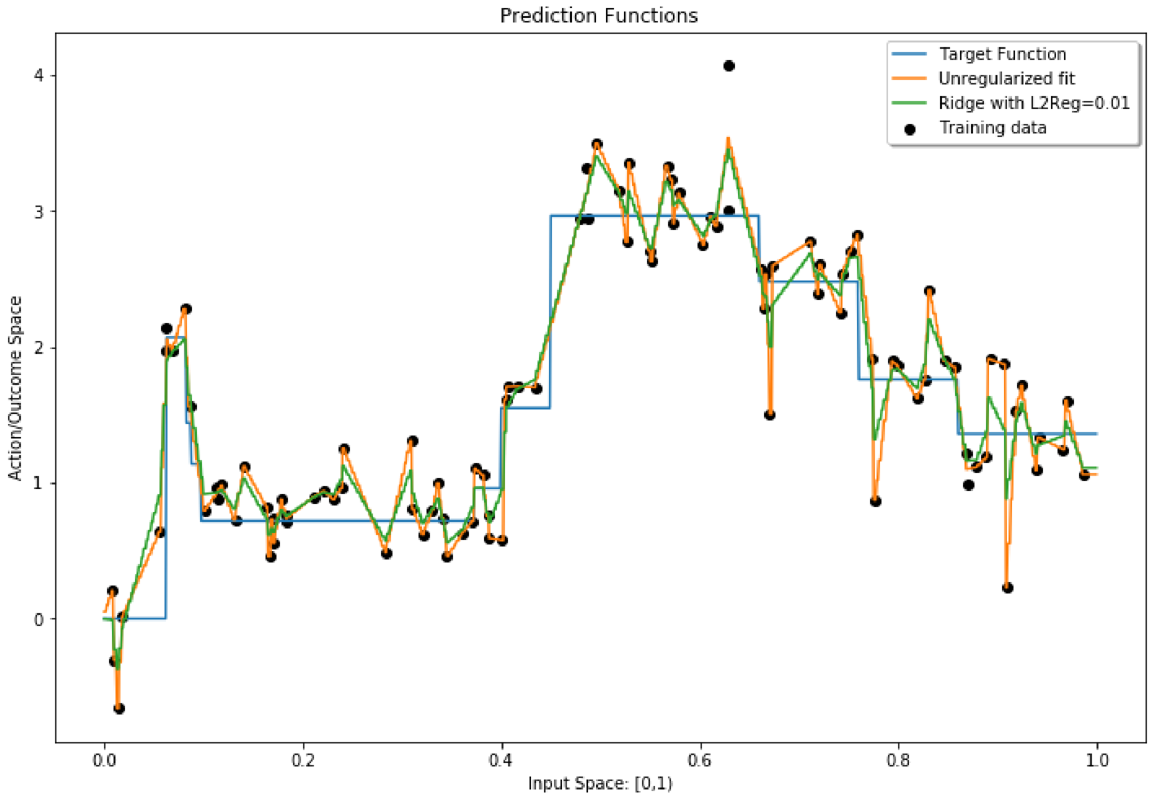
```
Best parameter is:  0.01
```

Q2

Plot the training data, the target function, an unregularized least squares fit (still using the featurizeddata), and the prediction function chosen in the previous proble

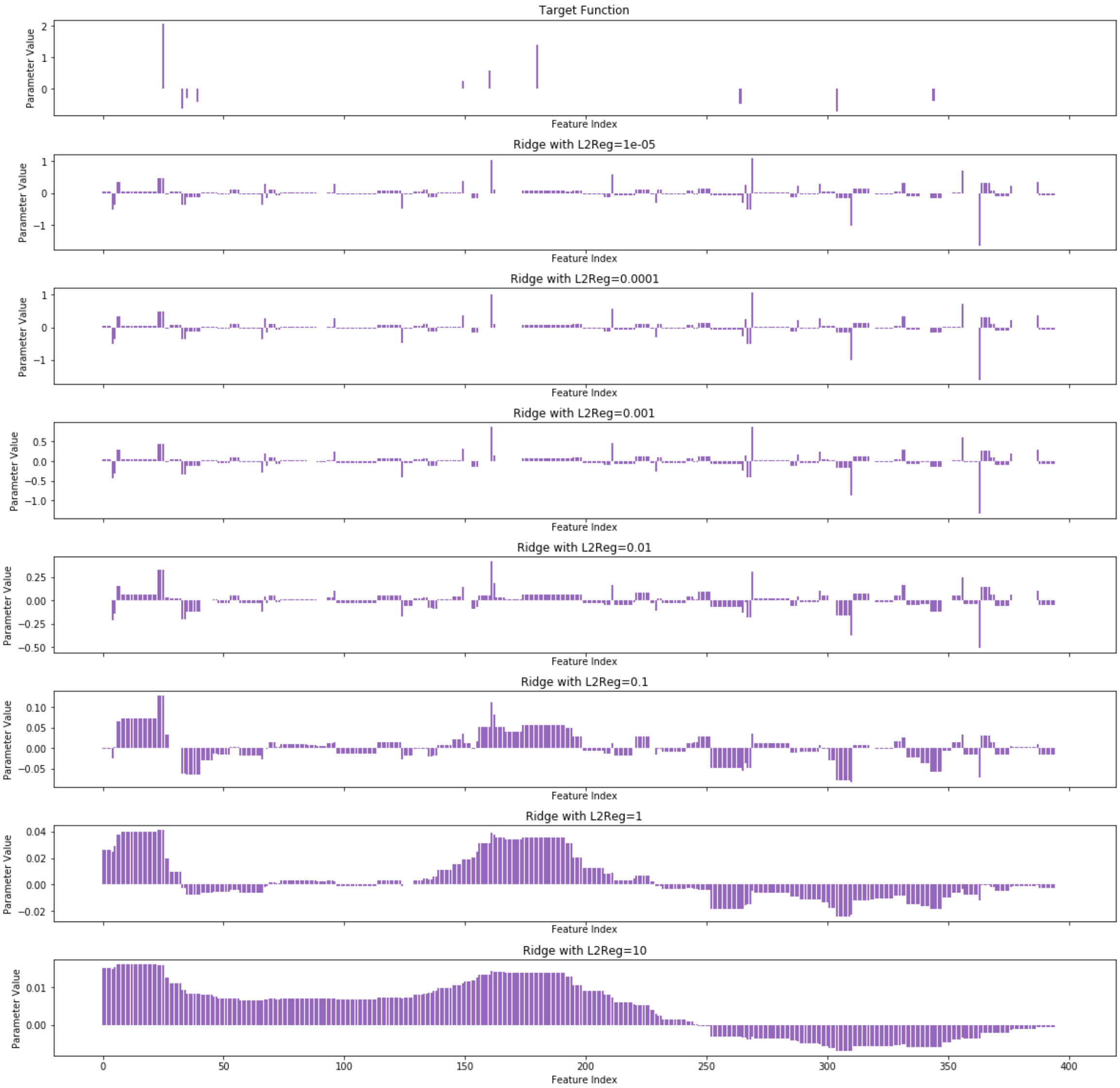
In [66]:

```
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)
lambdas = [0, best_param]
for l2reg in lambdas:
    ridge_regression_estimator = RidgeRegression(l2reg=l2reg)
    ridge_regression_estimator.fit(X_train, y_train)
    if l2reg != 0:
        name = "Ridge with L2Reg="+str(l2reg)
    else:
        name = "Unregularized fit"
    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 [70]:

```
pred_fns = []
pred_fns.append({"name": "Target Function", "coefs": coefs_true, "preds": target
_fn(x)})
lambdas = [1e-5,1e-4,1e-3,1e-2,1e-1,1,10]
for l2reg in lambdas:
    ridge_regression_estimator = RidgeRegression(l2reg=l2reg)
    ridge_regression_estimator.fit(X_train, y_train)
    if l2reg != 0:
        name = "Ridge with L2Reg="+str(l2reg)
    else:
        name = "Unregularized fit"
    pred_fns.append({"name":name,
                     "coefs":ridge_regression_estimator.w_,
                     "preds": ridge_regression_estimator.predict(X) })
compare_parameter_vectors(pred_fns);
```

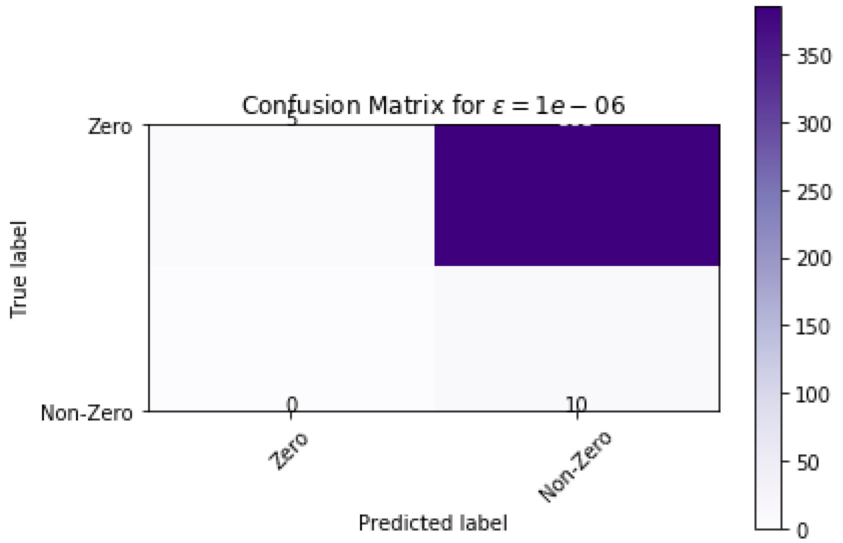


As we can see, as the regularization gets larger and larger, the coefficents will become smaller and smaller in absolute value. This confirms with how regularization works. Also, as regularization becomes smaller and closer to 0, the coefficients approach to those of the target function more and more.

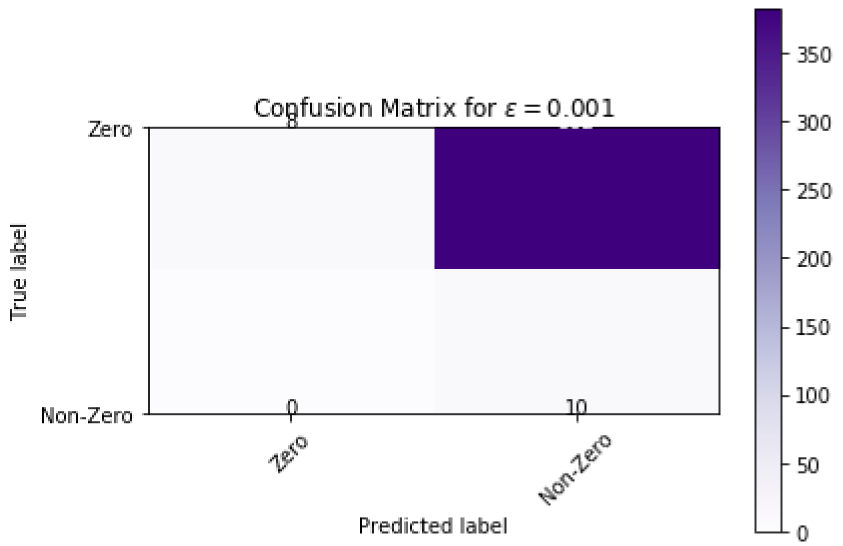
Q3

In [84]:

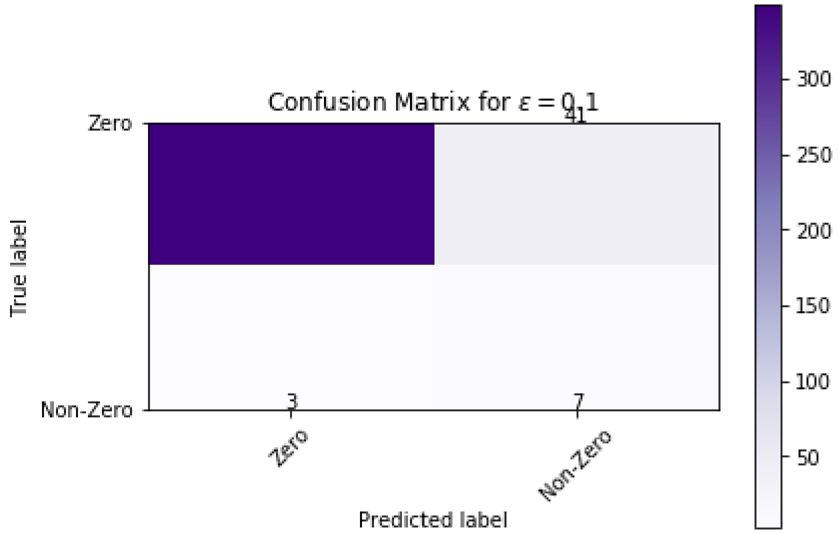
```
bin_coefs_true = coefs_true != 0  # your code goes here
eps_list = [1e-6,1e-3,1e-1]
ridge_regression_estimator = RidgeRegression(l2reg=best_param)
ridge_regression_estimator.fit(X_train, y_train)
w_tilde = ridge_regression_estimator.w_

for eps in eps_list:
    bin_coefs_estimated = np.abs(w_tilde) > eps  # your code goes here
    cnf_matrix = confusion_matrix(bin_coefs_true, bin_coefs_estimated)
    plt.figure()
    plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon = {}
$".format(eps), classes=["Zero", "Non-Zero"])
```



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



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


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

In [ ]: