Machine Learning and Computational Statistics
Homework 2: Lasso Regression

2.Ridge Regression

1

```
def do_grid_search_ridge(X_train, y_train, X_val, y_val, y_centered =
 ↪  0):
    # 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.
    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

    # Now we set up and do the grid search over l2reg. The
     ↪  np.concatenate
    # command illustrates my search for the best hyperparameter. In
     ↪  each line,
    # I'm zooming in to a particular hyperparameter range that showed
     ↪  promise
    # in the previous grid. This approach works reasonably well when
    # performance is convex as a function of the hyperparameter, which
     ↪  it seems
    # to be here.
    param_grid =
     ↪  [{'l2reg':np.unique(np.concatenate((10.**np.arange(-6,1,1),
                                            np.arange(1,3,.3)
                                              ))) }]
```

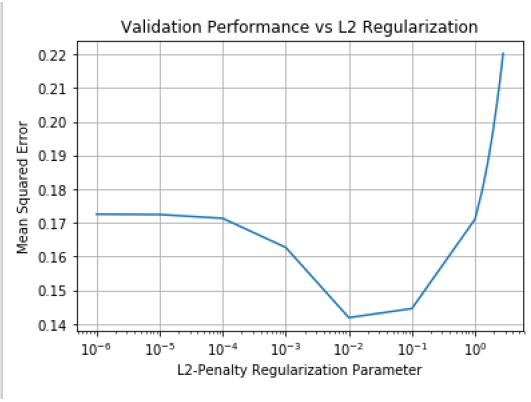
```
    ridge_regression_estimator = RidgeRegression()
    grid = GridSearchCV(ridge_regression_estimator,
                        param_grid,
                        cv = PredefinedSplit(test_fold=val_fold),
                        refit = True,
                        scoring = make_scorer(mean_squared_error,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

Run the ridge regression on the provided dataset, and obtain the following result, table of the parameter values you tried and the validation performance for each,

	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

	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

plot of the results



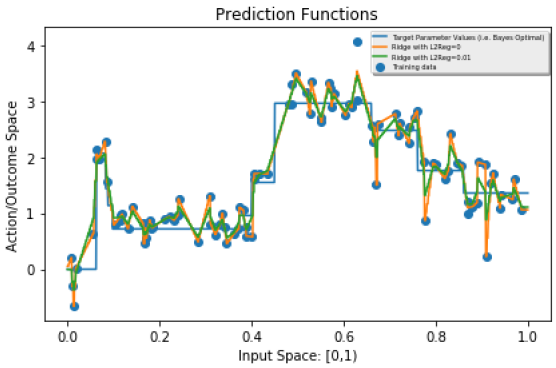
According to the table and the graph, $\lambda = 0.01$ minimizes the emipirical risk.

2

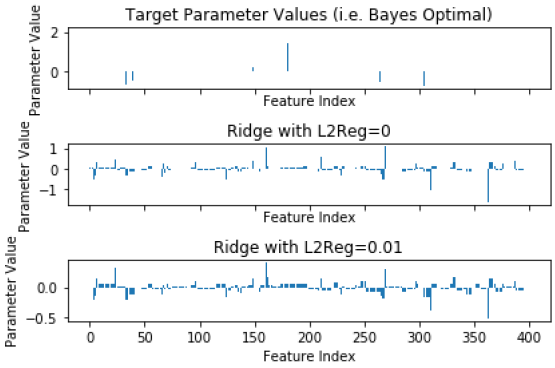
```
def compare_parameter_vectors(pred_fns):
    # Assumes pred_fns is a list of dicts, and each dict has a "name"
     ↪ key and a
    # "coefs" key
    fig, axs = plt.subplots(len(pred_fns),1, sharex=True)
    num_ftrs = len(pred_fns[0]["coefs"])
    for i in range(len(pred_fns)):
        title = pred_fns[i]["name"]
        coef_vals = pred_fns[i]["coefs"]
        axs[i].bar(range(num_ftrs), coef_vals)
        axs[i].set_xlabel('Feature Index')
        axs[i].set_ylabel('Parameter Value')
        axs[i].set_title(title)
    plt.tight_layout()
    fig.subplots_adjust(hspace=0.3)
    return fig

def plot_prediction_functions(x, pred_fns, x_train, y_train,
 ↪ legend_loc="bottom left"):
    # Assumes pred_fns is a list of dicts, and 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
    fig, ax = plt.subplots()
    ax.set_xlabel('Input Space: [0,1)')
    ax.set_ylabel('Action/Outcome Space')
```

```
    ax.set_title("Prediction Functions")
    plt.scatter(x_train, y_train, 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

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)
                          ↪  })

    f = plot_prediction_functions(x, pred_fns, x_train, y_train,
     ↪  legend_loc="bottom left")
    f.show()

    f = compare_parameter_vectors(pred_fns)
    plt.tight_layout()
```

The prediction functions are as follows,



The coefficients for each of the prediction functions plot are as follows,



We can see from the prediction function plot that the prediction function using $\lambda = 0.01$ is closer to the bayers optimal, while the prediction function with no regulation tend to overfit.

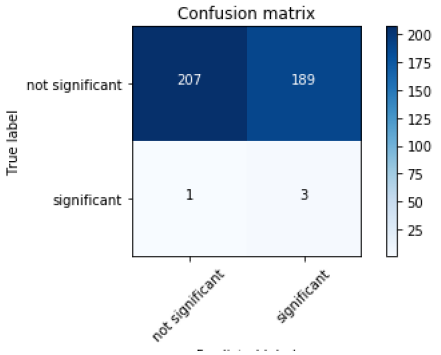
The parameters using $\lambda = 0.01$ has a smaller scale than those without regulation. Also , those parameter that is the significant in the bayers optimal tend to have the most weight in the ridge regression.

3

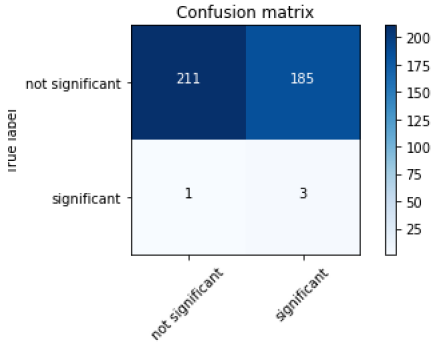
```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

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

    fmt = '.2f' if normalize else 'd'
```

```
    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], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

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


def if_bigger_than_threshold(x, threshold):
    if x > threshold:
        return 1;
    else:
        return 0;

for threshold in 10.**np.array([-6,-3, -1]):
        coefs_true_binary = [if_bigger_than_threshold(x, 0) for x in
         ↪  coefs_true]
        coefs_pred_binary = [if_bigger_than_threshold(x, threshold) for
         ↪  x in pred_fns[1]["coefs"]]
        cnf_matrix = confusion_matrix(coefs_true_binary,
         ↪  coefs_pred_binary)
        classes = ['not significant', 'significant']
        plot_confusion_matrix(cnf_matrix, classes)
```

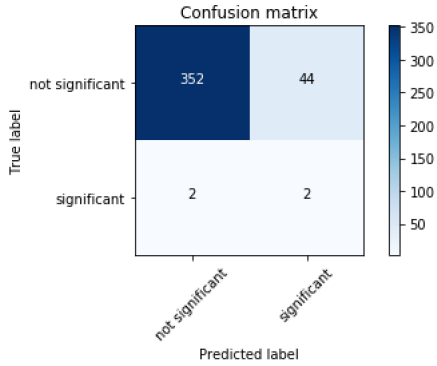
Confusion matrix when threshold is 10^{-6}



Confusion matrix when threshold is 10^{-3}



Confusion matrix when threshold is 10^{-1}



We can see that, the more regulation, the less the false positive, the more the true negative.

3.1 experiments with the shooting algorithm

1

$$a_j = 2 * X_{.j}^T X_{.j}$$

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

2

The function that computes the Lasso solution for a given λ using the shooting algorithm is as follows,

---

```
def delete_all_zero(X):
    delete_index = []
    num_instances, num_features = X.shape[0], X.shape[1]
    for i in range(num_features):
        if all(v == 0 for v in X[:, i]):
            delete_index.append(i)
    X_nz = np.delete(X, delete_index, 1)
    return X_nz, delete_index

def ridge_obj_loss(X, y, w, l1reg):
        num_instances, num_features = X.shape[0], X.shape[1]
        predictions = np.dot(X, w)
        residual = y - predictions
        empirical_risk = np.sum(residual**2) / num_instances
        l1_norm_squared = np.sum(w)
        objective = empirical_risk + l1reg * l1_norm_squared
        return objective
def loss_func(X, y, w):
        num_instances, num_features = X.shape[0], X.shape[1]
        predictions = np.dot(X, w)
        residual = y - predictions
        loss = np.sum(residual**2) / num_instances
        return loss

def lasso_cood_descent_32(X_q, y, X_t, y_t, lambda_reg = 1, stop_diff =
 ↪ 10.0**-8, num_iter = 1000, Murphy = 1, cyclic = 1):
    """
    at each step we optimize over one component of the unknown
 ↪ parameter vector,
    ï¬xingallothercomponents.Thedescentpathsoobtainedisasequenceofsteps,
    each of which is parallel to a coordinate axis in R^d , hence the
 ↪ name.
```

```
    It turns out that for the Lasso optimization problem, we can
    ï¬ndaclosedform
    solution for optimization over a single component
    ï¬xingallothercomponents.
    This gives us the following algorithm, known as the shooting
↪   algorithm

    Args:
        X_q - the feature vector, 2D numpy array of size
↪   (num_instances, num_features)
        y - the label vector, 1D numpy array of size (num_instances)
        X_t - the feature vector for tesing, 2D numpy array of size
↪   (num_instances, num_features)
        y_t - the label vector for tesing, 1D numpy array of size
↪   (num_instances)
        lambda_reg - the regulation parameter
        num_iter - number of iterations to run
        stop_diff - stop criteria: if the difference between two
↪   iteration is smaller than stop, then break and return
        Murphy - if Murphy = 1, start at the ridge regression solution
↪   suggested by Murphy, else start at 0
        cyclic - if cyclic = 1, do cyclic coordinate descent, else do
↪   randomized coordinate descent

    """

    #delete columns of all zeros, as these feature are of no use
    X, delete_ind = delete_all_zero(X_q)
    X_td = np.delete(X_t, delete_ind, 1)
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize
     ↪  theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist

    if Murphy == 1:
        #start at the ridge regression solution suggested by Murphy
        w = inv(X.T.dot(X) + lambda_reg *
         ↪  np.identity(num_features)).dot(X.T).dot(y)
        label_M = 'start at the solution suggested by Murphy'
    else:
        w = np.zeros(num_features)
        label_M = 'starting at 0'

    theta_hist[0] = w
    loss_hist[0] = ridge_obj_loss(X, y, w, l1reg = lambda_reg)
```

```
    for i in range(num_iter):
        if cyclic == 1:
            cyclic_label = 'cyclic'
            index_list = np.arange(num_features)
        else:
            cyclic_label = 'randomized'
            index_list = np.arange(num_features)
            np.random.shuffle(index_list)
        # coordinate descent
        for j in index_list:
            a = 2 * X[:,j].dot(X[:,j])
            c = 2 * X[:,j].dot(y - X.dot(w) + w[j] * X[:,j])
            if c < -lambda_reg:
                w[j] = (c + lambda_reg) / a
            elif c > lambda_reg:
                w[j] = (c - lambda_reg) / a
            else:
                w[j] = 0
        theta_hist[i+1] = w
        loss_hist[i+1] = ridge_obj_loss(X, y, w, lambda_reg)
        if abs(loss_hist[i+1]-loss_hist[i]) < stop_diff:
            print ('when lambda = {0}, coordinate descent converge in
             ↪ {1} iteration\n'.format(lambda_reg, i))
            print('for {2} coordinate descent, the test squared loss is
             ↪ {0} using the solution {1}\n'.format(loss_func(X_td,
             ↪ y_t, w), label_M, cyclic_label))
            break
        if(i == num_iter - 1):
            print ('for {1} coordinate descent, the coordinate descent
             ↪ don\'t converge using the solution {0}'.format(label_M,
             ↪ cyclic_label))

def lasso_cood_descent_32_helper(X_q, y, X_t, y_t, lambda_reg = 1,
 ↪ stop_diff = 10.0**-8, num_iter = 1000):
    #do two for loop for different stating point and different styles
     ↪ of coordinate descent(cyclic or randomized)
    for m in [0, 1]:
        for c in [0, 1]:
            lasso_cood_descent_32(X_q, y, X_t, y_t, lambda_reg =
             ↪ lambda_reg, stop_diff = stop_diff, num_iter = num_iter,
             ↪ Murphy = m, cyclic = c)
```

The results for using performance of cyclic coordinate descent to randomized coordinate descent, and starting at 0 versus starting at the ridge regression solution suggested by Murphy is as follows,

when lambda = 1, coordinate descent converge in 807 iteration

for randomized coordinate descent, the test squared loss is 0.12397023201831626 using the solution starting at 0

when lambda = 1, coordinate descent converge in 774 iteration

for cyclic coordinate descent, the test squared loss is 0.16782069948314618 using the solution starting at 0

when lambda = 1, coordinate descent converge in 552 iteration

for randomized coordinate descent, the test squared loss is 0.12554824748102403 using the solution start at the solution suggested by Murphy

when lambda = 1, coordinate descent converge in 199 iteration

for cyclic coordinate descent, the test squared loss is 0.12678590609462204 using the solution start at the solution suggested by Murphy

We can see that the randomized coordinate descent using the solution start at the solution suggested by Murphy gives the best performance and converge the fastest.

3

```
def lasso_cood_descent_33(X_q, y, x_t, y_t, stop_diff = 10.0**-8,
 ↪ num_iter = 1000, homotopy = 0, homotopy_para = 0.8, y_centered =
 ↪ 0):
    """
    at each step we optimize over one component of the unknown
 ↪ parameter vector,
    ï¬xingallothercomponents.Thedescentpathsoobtainedisasequenceofsteps,
    each of which is parallel to a coordinate axis in R^d , hence the
 ↪ name.
    It turns out that for the Lasso optimization problem, we can
    ï¬ndaclosedform
    solution for optimization over a single component
    ï¬xingallothercomponents.
    This gives us the following algorithm, known as the shooting
 ↪ algorithm

    Args:
        X_q - the feature vector, 2D numpy array of size
 ↪ (num_instances, num_features)
        y - the label vector, 1D numpy array of size (num_instances)
        X_t - the feature vector for tesing, 2D numpy array of size
 ↪ (num_instances, num_features)
```

```
        y_t - the label vector for tesing, 1D numpy array of size
↪  (num_instances)
        lambda_reg - the regulation parameter
        num_iter - number of iterations to run
        stop_diff - stop criteria: if the difference between two
↪  iteration is smaller than stop, then break and return
        Murphy - if Murphy = 1, start at the ridge regression solution
↪  suggested by Murphy, else start at 0
        cyclic - if cyclic = 1, do cyclic coordinate descent, else do
↪  randomized coordinate descent
        homotopy - if homotopy!=0, use homotopy method
        homotopy_para - reduce partmeter for homotopy method
        y_centered - if y_centered != 0, y is centered
    """


    if y_centered != 0:
        #center y
        y_ = (y-np.mean(y)) / np.std(y)
        y_t = (y_t-np.mean(y)) / np.std(y)
        y = y_
    #delete columns of all zeros, as these feature are of no use
    X, delete_ind = delete_all_zero(X_q)
    #x_t = np.sort(x_t)
    X_t = featurize(x_t)
    X_td = np.delete(X_t, delete_ind, 1)
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize
     ↪  theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist

    index_list = np.arange(num_features)
    #record the validation loss for each lambda
    loss_record = []
    #record the prediction function for each lambda
    pred_fns = []
    if homotopy == 0:
        lambda_reg_list = 10.**np.arange(-6, 2)
    else:
        #homotopy method: start lambda from lambda_max, then reduce it
         ↪  repeatly.
        #And the optimization problem is solved using the previous
         ↪  optimal
        # point as the starting point.
        lambda_reg_list = 10.**np.arange(-6, 2)
        lambda_reg_list = []
```

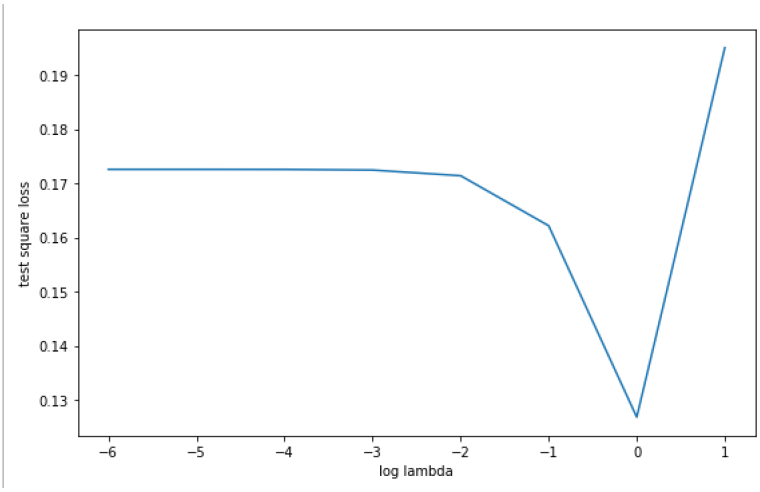
```
    lambda_reg_max = np.max(2*X.T.dot(y))
    lambda_reg_r = lambda_reg_max
    for i in range(35):
        lambda_reg_list.append(lambda_reg_r)
        lambda_reg_r *= homotopy_para
w = np.zeros(num_features)
for lambda_reg in lambda_reg_list:
    if homotopy == 0:
        w = inv(X.T.dot(X) + lambda_reg *
         ↪ np.identity(num_features)).dot(X.T).dot(y)

    #w = np.zeros(num_features)
    theta_hist[0] = w
    loss_hist[0] = ridge_obj_loss(X, y, w, l1reg = lambda_reg)
    # coodinate descent
    for i in range(num_iter):
        np.random.shuffle(index_list)
        for j in range(num_features):
            a = 2 * X[:,j].dot(X[:,j])
            c = 2 * X[:,j].dot(y - X.dot(w) + w[j] * X[:,j])
            if c < -lambda_reg:
                w[j] = (c + lambda_reg) / a
            elif c > lambda_reg:
                w[j] = (c - lambda_reg) / a
            else:
                w[j] = 0

            #return theta_hist, loss_hist
        theta_hist[i+1] = w
        loss_hist[i+1] = ridge_obj_loss(X, y, w, l1reg =
         ↪ lambda_reg)
        if abs(loss_hist[i+1]-loss_hist[i]) < stop_diff:
            loss_record.append(loss_func(X_td, y_t, w))

            print ('when lambda = {0}, coodinate descent converge
             ↪ in {1} iteration)\n'.format(lambda_reg, i))
            break
        if(i == num_iter - 1):
            loss_record.append(-1)
            print ('don\'t converge')

    # append the prediction function
    name = 'lambda = {}'.format(lambda_reg)

    x_ts = np.sort(x_t)
    X_ts = featurize(x_ts)
```

```
        X_tsd = np.delete(X_ts, delete_ind, 1)


        pred_fns.append({"name": name,
                 "coefs": w,
                 "preds": X_tsd.dot(w)})
    ii = 0
    for record in loss_record:
        print ('lambda = 10^{0}, loss =
         ↪ {1}'.format(np.log10(lambda_reg_list[ii]), record))
        ii += 1

    plt.figure(figsize=(8, 5))
    plt.plot(np.log10(lambda_reg_list), loss_record)
    plt.xlabel("log lambda")
    plt.ylabel('test square loss')
    plt.legend()
    params = {'legend.fontsize': 5,
          'legend.handlelength': 2}
    plt.rcParams.update(params)
    plt.tight_layout()
    #plt.savefig()
    plt.show()
    return pred_fns

pred_fns = lasso_cood_descent_33(X_train, y_train, x_val, y_val,
 ↪ stop_diff = 10.0**-8, num_iter = 1000, homotopy = 0)
f332 = compare_parameter_vectors(pred_fns)
f332.show()
pred_fns.append({"name":"Bayes Optimal", "coefs":coefs_true, "preds":
 ↪ target_fn(x_train) })

f331 = plot_prediction_functions(np.sort(x_val), pred_fns, x_train,
 ↪ y_train, legend_loc="bottom left")
f331.show()
```

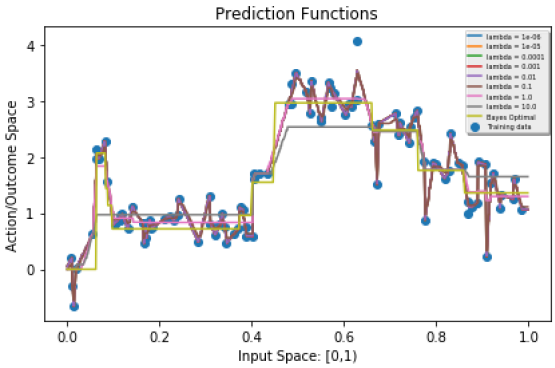
The table of the parameter values you tried and the validation performance for each is as follows,

```
lambda = 10^-6.0, loss = 0.17258980775649865
lambda = 10^-5.0, loss = 0.17258864793737194
lambda = 10^-4.0, loss = 0.17257694907967214
lambda = 10^-3.0, loss = 0.17246223374524508
lambda = 10^-2.0, loss = 0.17141523852877646
lambda = 10^-1.0, loss = 0.16216326851977303
lambda = 10^0.0, loss = 0.1267859060946221
lambda = 10^1.0, loss = 0.19507421415078688
```

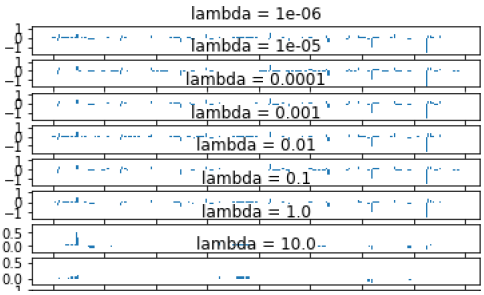
plot of these results.



plot of the prediction functions



bar charts of coefficients.



From the table and the plot of validation loss using different lambda, we can see that the best model is the one using $\lambda = 1$, and the average validation loss of the model is 0.1267.

We can see from the prediction function plot that the prediction function using $\lambda = 1$ is closer to the bayers optimal, while the prediction function with less regulation tend to overfit.

With regard to parameter sparsity, the parameter sparsity given by lasso regression is much higher than that given by ridge regression. And the more regulation, the more sparse the parameters.

4

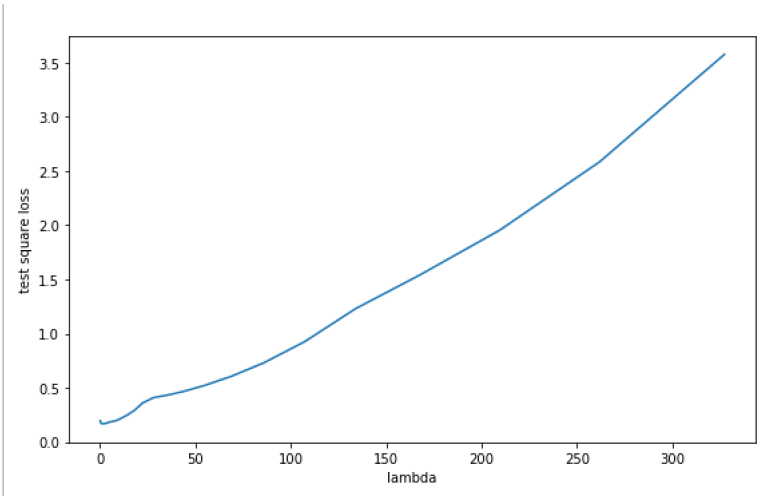
The function is the same as the one in question 3. We implement it by using

```
pred_fns = lasso_cood_descent_33(X_train, y_train, x_val, y_val,
 ↪  stop_diff = 10.0**-8, num_iter = 1000, homotopy = 1)
```

The table of the parameter values you tried and the validation performance for each lambda is as follows,

```
lambda = 327.2828323295212, loss = 3.5765529343093485
lambda = 261.826265863617, loss = 2.5844779406182616
lambda = 209.4610126908936, loss = 1.9538125351795286
lambda = 167.5688101527149, loss = 1.5401284307349365
lambda = 134.0550481221719, loss = 1.2308743485727163
lambda = 107.24403849773753, loss = 0.9251649796396434
lambda = 85.79523079819003, loss = 0.7294642859574281
lambda = 68.63618463855202, loss = 0.6041738489644687
lambda = 54.90894771084162, loss = 0.521986282116319
lambda = 43.9271581686733, loss = 0.4667694179661933
lambda = 35.14172653493864, loss = 0.43155104269186695
lambda = 28.113381227950914, loss = 0.409114377064572
lambda = 22.490704982360732, loss = 0.36079257206778576
lambda = 17.992563985888587, loss = 0.29096162693587857
lambda = 14.39405118871087, loss = 0.24900084658196212
lambda = 11.515240950968696, loss = 0.22261713764885044
lambda = 9.212192760774958, loss = 0.20129556003556376
lambda = 7.369754208619966, loss = 0.1914919613557205
lambda = 5.895803366895973, loss = 0.1872337414270721
lambda = 4.7166426935167785, loss = 0.18271915447407855
lambda = 3.773314154813423, loss = 0.17448543941177516
lambda = 3.0186513238507384, loss = 0.17070352287444251
lambda = 2.4149210590805907, loss = 0.16960648489525448
lambda = 1.9319368472644727, loss = 0.17007694114087626
lambda = 1.5455494778115781, loss = 0.16862589179609272
lambda = 1.2364395822492626, loss = 0.16755026100379125
lambda = 0.9891516657994102, loss = 0.1682913572636202
lambda = 0.7913213326395282, loss = 0.1698169394822542
lambda = 0.6330570661116226, loss = 0.1720062801925209
lambda = 0.5064456528892981, loss = 0.17497093213155288
lambda = 0.4051565223114385, loss = 0.1787257345953635
lambda = 0.3241252178491508, loss = 0.18245754007778298
lambda = 0.25930017427932067, loss = 0.18703537865750913
lambda = 0.20744013942345654, loss = 0.19168978991874294
lambda = 0.16595211153876524, loss = 0.19636116801965095
```

plot of these results.



From the table and the plot of validation loss using different lambda, we can see that the best model is the one using $\lambda = 0.99$, and the average validation loss of the model is 0.168.

5

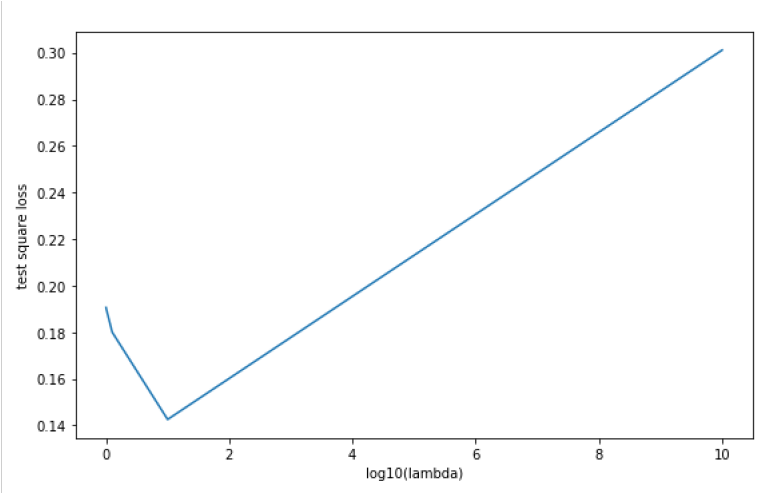
The function is the same as the one in question 3. We implement it by using

```
pred_fns = lasso_cood_descent_33(X_train, y_train, x_val, y_val,
 ↪  stop_diff = 10.0**-8, num_iter = 1000, homotopy = 0, y_centered =
 ↪  1)
```

After centering the y, the table of the parameter values you tried and the validation performance for each lambda is as follows,

```
lambda = 10^-6.0, loss = 0.19062340202685546
lambda = 10^-5.0, loss = 0.19062216705025983
lambda = 10^-4.0, loss = 0.19060982578195165
lambda = 10^-3.0, loss = 0.19048883609831033
lambda = 10^-2.0, loss = 0.18943414789658425
lambda = 10^-1.0, loss = 0.18005780050145112
lambda = 10^0.0, loss = 0.14258089324141743
lambda = 10^1.0, loss = 0.3011299211954395
```

plot of these results.



The result using centered y is not much different from those using regular y. The reason could be the bias for the method using regular y is very small, changing it to centered y don't have a large influence on the result.

3.2 Deriving the Coordinate Minimizer for Lasso

1

$$w_j = 0$$

2

$$f(w_j) = a_j - c_j + \lambda sign(w_j)$$

3

$$w_j = \begin{cases} \frac{c_j - \lambda}{a_j} & w_j > 0 \\ \frac{c_j + \lambda}{a_j} & w_j < 0 \end{cases}$$

$c_j > \lambda$ implies minimizer $w_j > 0$.$c_j < \lambda$ implies minimizer $w_j < 0$
so it can also be written as

$$w_j = \begin{cases} \frac{c_j - \lambda}{a_j} & c_j > \lambda \\ \frac{c_j + \lambda}{a_j} & c_j < -\lambda \end{cases}$$

4

When $w_j \to 0^+$

$$\lim_{w_j \to 0^+} \frac{f(w_j) - f(0)}{w_j} = \lambda \geq 0$$

When $w_j \to 0^-$

$$\lim_{w_j \to 0^-} \frac{f(w_j) - f(0)}{w_j} = -\lambda \leq 0$$

Therefore, w_j is a minimizer.

5

From question 3, we conclude

$$w_j = \begin{cases} \frac{c_j - \lambda}{a_j} & c_j > \lambda \\ \frac{c_j + \lambda}{a_j} & c_j < -\lambda \end{cases}$$

From question 4, we conclude $w_j = 0 \quad c_j \in [-\lambda, \lambda]$
Combine the above two results, we get

$$w_j = \begin{cases} \frac{c_j - \lambda}{a_j} & c_j > \lambda \\ w_j = 0 \quad c_j \in [-\lambda, \lambda] \\ \frac{c_j + \lambda}{a_j} & c_j < -\lambda \end{cases}$$

4.1 Deriving λ_{max}

1

$$\begin{aligned} & J^{'}(0;v) \\ &= \lim_{h\to 0} \frac{(xhv-y)^T(xhv-y) - y^Ty + \lambda\|hv\|_1}{h} \\ &= -2y^Txv + \lambda\|v\|_1 \end{aligned}$$

2

$$J^{'}(0;v) = -2y^Txv + \lambda\|v\|_1 \geq 0$$

$$\lambda \geq \frac{2y^Txv}{\|v\|_1}$$

$$C = \frac{2y^Txv}{\|v\|_1}$$

3

$$\lambda_{max} = \max_v C = \max_v 2y^Tx\frac{v}{\|v\|_1}$$

Assume $\left\|X^Ty\right\|_\infty$ is obtained at the ith element of X^Ty, then to obtain $\max_v 2y^Tx\frac{v}{\|v\|_1}$, the v has to be [0,0,..0,1,0,..0], where 1 is at the ith element of the array.

given v

$$\lambda_{max} = 2\left\|X^Ty\right\|_\infty$$

$J^{'}(0;v) > 0$ when $\lambda \geq \lambda_{max}$
Therefore, w=0 is a minimizer of J(w) if and only if $\lambda \geq \max_v C = \lambda_{max} = 2\left\|X^Ty\right\|_\infty$

4

The one side directional derivative of $J(b)$ at $b = b$is,

$$J^{'}(b;w) = \lim_{h\to 0} \frac{(xw + (b+\epsilon)I - y)^T(xw + (b+\epsilon)I - y) - (xw + bI - y)^T(xw + bI - y)}{h} = 2I^T xw - 2I^T y + 2b$$

When $w = 0$, let

$$J_b^{'}(b;0) = -2I^T y + 2b = 0$$

we obtain $b = \bar{y}$

The one side directional derivative of $J(w)$ at $w = 0$ and $b = \bar{y}$ in the direction v is,

$$J_w^{'}(0;v,\bar{y}) = -2v^T x^T(\bar{y}I - y) + \lambda\|v\|_1$$

let $J_w^{'}(0;v,\bar{y}) > 0$,
we obtain,

$$\lambda \geq 2\frac{v^T x^T(\bar{y}I - y)}{\|v\|_1}$$

using the logic in question 4, we get,

$$\lambda_{max} = 2\left\|X^T(y - \bar{y})\right\|_\infty$$

Therefore, $(w^*, b^*) = (0, \bar{y})$ is a minimizer of J(w,b) if and only if $\lambda \geq \lambda_{max} = 2\left\|X^T(y - \bar{y})\right\|_\infty$

4.2 Feature Correlation

1

we can obtain the following results by computing derivative of $J(\theta)$. While computing, I exchange all the X_2 to X_1 as they are the same

$$\frac{dJ(\theta)}{d\theta_1} = 2X_1^T X_1\theta_1 + 2X_1^T X_1\theta_2 + 2X_1^T(X_r\theta_r - y) + \lambda sign(\theta_1)$$

$$\frac{dJ(\theta)}{d\theta_2} = 2X_1^T X_1\theta_1 + 2X_1^T X_1\theta_2 + 2X_1^T(X_r\theta_r - y) + \lambda sign(\theta_2)$$

While $\theta = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$, $J(\theta)$ is minimized, therefore, we can say that,

$$\frac{dJ(\theta)}{d\theta_1} = \frac{dJ(\theta)}{d\theta_2} = 0$$

when $\theta_1 = a$, $\theta_2 = b$

And when $\theta = 0$, $J(\theta)$ in indifferentiable.
Therefore, we can conclude that a and b must have the same sign, or at least one of them is zero.

Using the conclusion we obtained above, we can compute that,

$$a + b = X_1^{T-1}X_1^{-1}(X_1^T(X_r\theta_r - y) + \frac{\lambda sign(a)}{2}) = C$$

given $a \neq 0$ and $b \neq 0$
When $a = 0$ or $b = 0$,
The results is the same as the above one(change sign(a) to sign(b) if a=0)
When $a = 0, b = 0$, $a + b = 0$
Therefore, $c + d = a + b$, and c and d must have the same sign, or at least one of them is zero.

2

$$\frac{dJ(\theta)}{d\theta_1} = 2X_1^T X_1\theta_1 + 2X_1^T X_1\theta_2 + 2X_1^T(X_r\theta_r - y) + 2\lambda\theta_1 = 0 \quad (1)$$

$$\frac{dJ(\theta)}{d\theta_2} = 2X_1^T X_1\theta_1 + 2X_1^T X_1\theta_2 + 2X_1^T(X_r\theta_r - y) + 2\lambda\theta_2 = 0 \quad (2)$$

we subtract (2) from (1) to obtain,

$$\theta_1 = \theta_2$$

that is,

$$a = b$$

5 The Ellipsoids in the l1 /lâ¢2 regularization picture

1

By, substituting $\hat{w}$ with $\left(X^T X\right)^{-1} X^T y$ we can show that,

$$X\hat{w} = y$$

$$\begin{aligned}
\hat{R}_n(\hat{w}) &= \frac{1}{n}\sum_{i=1}^{n}\left(w^T x_i - y_i\right)^2 \\
&= \frac{1}{n}\left(Xw - y\right)^T\left(Xw - y\right). \\
&= \frac{1}{n}(\hat{w}^T X^T X\hat{w} - 2y^T X\hat{w} + y^T y) \\
&= \frac{1}{n}\left(-y^T X\hat{w} + y^T y\right)
\end{aligned}$$