1 Ridge Regression

In the problems below, you do not need to implement ridge regression. You may use any of the code provided in the assignment, or you may use other packages. However, your results must correspond to the ridge regression objective function that we use, namely

$$J(w;\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(w^T x_i - y_i\right)^2 + \lambda\|w\|^2.$$

1. Run ridge regression on the provided training dataset. Choose the λ that minimizes the empirical risk (i.e. the average square loss) on the validation set. Include a **table of the parameter values** you tried and the validation performance for each. Also include **a plot of the results**.

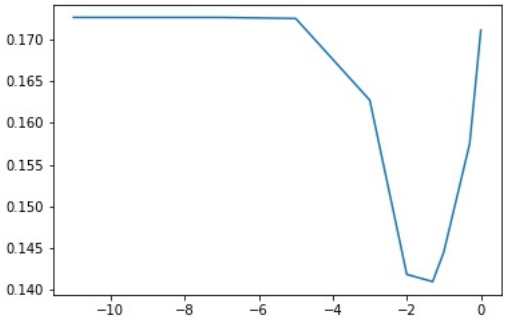
```
#we can choose the lambda from 1e−11 to 1.0
#using the lambda we choose, to implement the RidgeRegression function, getting the w value
#fitting the X_train and y_train
#predicting the X_val and y_val
l2reg = [1e−11, 1e−9, 1e−7,1e−5,1e−3,1e−2,0.05, 0.1, 0.5, 1.0]
    error_record = []
    for lambda_ in l2reg:
        regression_estimator = RidgeRegression(l2reg = lambda_)
        regression_estimator.fit(X_train, y_train)
        w_opt = regression_estimator.w_
        error = 1/len(y_val)*np.dot( np.dot(w_opt,X_val.T) − y_val, np.dot(w_opt,X_val.T) − y_val)
        error_record.append(error)

    error_df = pd.DataFrame(columns = ['lambda','empirical risk'])
    error_df['lambda'] = l2reg
    error_df['empirical risk'] = error_record
    print(error_df)
    #plot
    plt.figure()
    plt.plot(np.log10(error_df['lambda']),error_df['empirical risk'])
    plt.savefig("different lambda.jpg")
```

Function 1: different lambda

Ridge Regression

	lambda	empirical risk
0	1.000000e-11	0.172592
1	1.000000e-09	0.172592
2	1.000000e-07	0.172590
3	1.000000e-05	0.172464
4	1.000000e-03	0.162705
5	1.000000e-02	0.141887
6	5.000000e-02	0.141019
7	1.000000e-01	0.144566
8	5.000000e-01	0.157506
9	1.000000e+00	0.171068

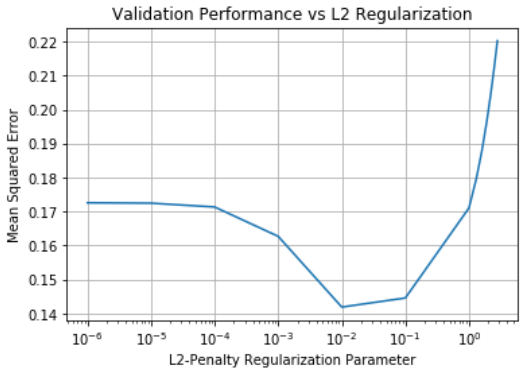
(a) risk value table



(b) risk value plot

	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

(c) risk value table



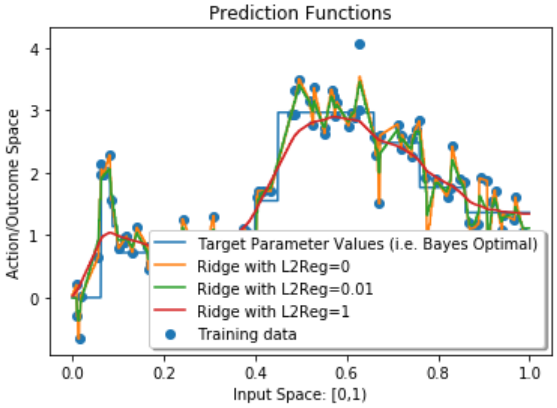
(d) risk value plot

For this question, I try the lambda with $[1e-11, 1e-9, 1e-7, 1e-5, 1e-3, 1e-2, 0.05, 0.1, 0.5, 1.0]$. The risk table is as below, and we can see if we choose lambda to 0.05 then we can get the best risk value. Also, checking the risk value plot, we can notice the risk value decrease as first several lambda attempt, then after 0.05, risk value increase. So for the lambda I choose for this question, the best one is 0.05. Also, if we look at the table with the source code(the code in the homework), we can find when l2reg equal to 0.01, mean test score will get the best.

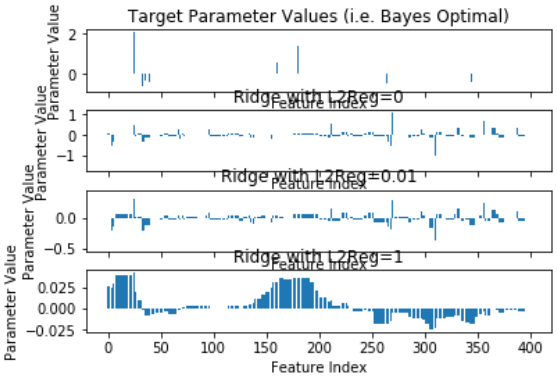
2. Now we want to visualize the prediction functions. On the same axes, plot the following: **the training data, the target function, an unregularized least squares fit (still using the featurized data), and the prediction function** chosen in the previous problem. Next, along the lines of the bar charts produced by the code in compare_parameter_vectors, **visualize the coefficients** for each of the prediction functions plotted, including the target function. **Describe the patterns**, including the scale of the coefficients, as well as which coefficients have the most weight.

```
pred_fns = []
  x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))
  name = "Target Parameter Values (i.e. Bayes Optimal)"
  pred_fns.append({"name":name, "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) })
  f = plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best")
  f.show()
  f = compare_parameter_vectors(pred_fns)
  f.show()
# to get the top coefficient
print(np.argmax(np.abs(coefs_true)),np.max(np.abs(coefs_true)))
# the  result is 25 2.06957209208771

```

Function 2: visualize the prediction functions



(e) prediction function



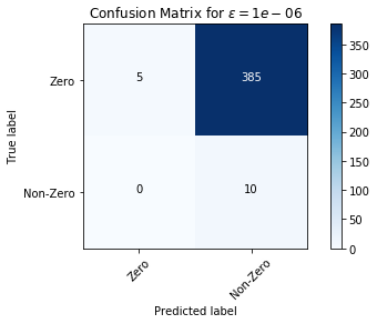
(f) target parameter value

Looking at the prediction functions plot, we can see the predict with lambda = 0.01 is very dense. For the scale of coefficients, the target function have parameters with much larger scale compared with the ridge regression. For the target function and regularized regression, the 25th coefficient have most weight which is 2.069.

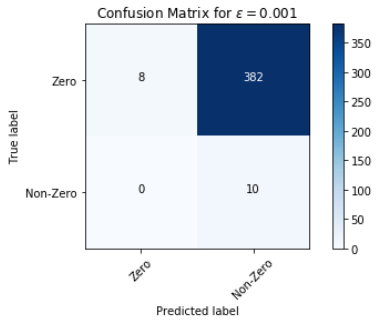
3. For the chosen λ, examine the model coefficients. For ridge regression, we don't expect any parameters to be exactly 0. However, let's investigate whether we can predict the sparsity pattern of the true parameters (i.e. which parameters are 0 and which are nonzero) by thresholding the parameter estimates we get from ridge regression. **We'll predict that $w_i = 0$ if $|\hat{w}_i| < \varepsilon$ and $w_i \neq 0$ otherwise.** Give the confusion matrix for $\varepsilon = 10^{-6}, 10^{-3}, 10^{-1}$, and any other thresholds you would like to try.

```
"""question 2.3"""
    regression_estimator = RidgeRegression(l2reg = 0.01)
    regression_estimator.fit(X_train, y_train)
    w_opt = regression_estimator.w_
    w = coefs_true
    w_ture = [(lambda i: 0 if i ==0 else 1) (i) for i in w]
    w_0 = [(lambda i: 0 if np.abs(i) < 1e-6 else 1) (i) for i in w_opt]
    w_1 = [(lambda i: 0 if np.abs(i) < 1e-3 else 1) (i) for i in w_opt]
    w_2 = [(lambda i: 0 if np.abs(i) < 1e-1 else 1) (i) for i in w_opt]

  cnf_matrix = confusion_matrix(w_ture, w_0)
  plt.figure()
  plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon = {}$".format(1e-6), classes
      ↪ =["Zero", "Non-Zero"])
  plt.show()
    cnf_matrix = confusion_matrix(w_ture, w_1)
  plt.figure()
  plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon = {}$".format(1e-3), classes
      ↪ =["Zero", "Non-Zero"])
  plt.show()
    cnf_matrix = confusion_matrix(w_ture, w_2)
  plt.figure()
  plot_confusion_matrix(cnf_matrix, title="Confusion Matrix for $\epsilon = {}$".format(1e-1), classes
      ↪ =["Zero", "Non-Zero"])
  plt.show()
```

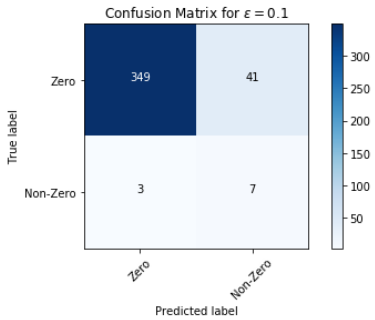
Function 3: confusion matrix with lambda = 0.01



(g) matrix 1



(h) matrix 2



(i) matrix 3

2 Coordinate Descent for Lasso (a.k.a. The Shooting algorithm)

The Lasso optimization problem can be formulated as

$$\hat{w} \in \arg\min_{w \in \mathbf{R}^d} \sum_{i=1}^{m} (h_w(x_i) - y_i)^2 + \lambda \|w\|_1,$$

where $h_w(x) = w^T x$, and $\|w\|_1 = \sum_{i=1}^{d} |w_i|$. Note that to align with Murpy's formulation below, and for historical reasons, we are using the total square loss, rather than the average square loss, in the objective function.

Since the ℓ_1-regularization term in the objective function is non-differentiable, it's not immediately clear how gradient descent or SGD could be used to solve this optimization problem directly. (In fact, as we'll see in the next homework on SVMs, we can use "subgradient" methods when the objective function is not differentiable, in addition to the two methods discussed in this homework assignment.)

Another approach to solving optimization problems is coordinate descent, in which at each step we optimize over one component of the unknown parameter vector, fixing all other components. The descent path so obtained is a sequence of steps, each of which is parallel to a coordinate axis in $\mathbf{R}^d$, hence the name. It turns out that for the Lasso optimization problem, we can find a closed form solution for optimization over a single component fixing all other components. This gives us the following algorithm, known as the **shooting algorithm**:

Algorithm 13.1: Coordinate descent for lasso (aka shooting algorithm)

Initialize $\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$;
repeat
  for $j = 1, \ldots, D$ **do**
    $a_j = 2\sum_{i=1}^{n} x_{ij}^2$;
    $c_j = 2\sum_{i=1}^{n} x_{ij}(y_i - \mathbf{w}^T\mathbf{x}_i + w_j x_{ij})$;
    $w_j = \text{soft}(\frac{c_j}{a_j}, \frac{\lambda}{a_j})$;
until *converged*;

Figure 1: Shooting algorithm

The "soft thresholding" function is defined as

$$\text{soft}\,(a, \delta) = \text{sign}(a)\,(|a| - \delta)_+\,,$$

for any $a, \delta \in \mathbf{R}$.

NOTE: Algorithm 13.1 does not account for the case that $a_j = c_j = 0$, which occurs when the jth column of X is identically 0. One can either eliminate the column (as it cannot possibly help the solution), or you can set $w_j = 0$ in that case since it is, as you can easily verify,

the coordinate minimizer. Note also that Murphy is suggesting to initialize the optimization with the ridge regession solution. Although theoretically this is not necessary (with exact computations and enough time, coordinate descent will converge for lasso from any starting point), in practice it's helpful to start as close to the solution as we're able.

There are a few tricks that can make selecting the hyperparameter λ easier and faster. First, as we'll see in a later problem, you can show that for any $\lambda \geq 2\|X^T(y - \bar{y})\|_\infty$, the estimated weight vector $\hat{w}$ is entirely zero, where $\bar{y}$ is the mean of values in the vector y, and $\|\cdot\|_\infty$ is the infinity norm (or supremum norm), which is the maximum over the absolute values of the components of a vector. Thus we need to search for an optimal λ in $[0, \lambda_{\max}]$, where $\lambda_{\max} = 2\|X^T(y - \bar{y})\|_\infty$. (Note: This expression for $\lambda_{\max}$ assumes we have an unregularized bias term in our model. That is, our decision functions are of the form $h_{w,b}(x) = w^T x + b$. In our the experiments, we do not have an unregularized bias term, so we should use $\lambda_{\max} = 2\|X^T y\|_\infty$.)

The second trick is to use the fact that when λ and λ' are close, the corresponding solutions $\hat{w}(\lambda)$ and $\hat{w}(\lambda')$ are also close. Start with $\lambda = \lambda_{\max}$, for which we know $\hat{w}(\lambda_{\max}) = 0$. You can run the optimization anyway, and initialize the optimization at $w = 0$. Next, λ is reduced (e.g. by a constant factor close to 1), and the optimization problem is solved using the previous optimal point as the starting point. This is called **warm starting** the optimization. The technique of computing a set of solutions for a chain of nearby λ's is called a **continuation** or **homotopy method**. The resulting set of parameter values $\hat{w}(\lambda)$ as λ ranges over $[0, \lambda_{\max}]$ is known as a **regularization path**.

2.1 Experiments with the Shooting Algorithm

1. The algorithm as described above is not ready for a large dataset (at least if it has being implemented in Python) because of the implied loop in the summation signs for the expressions for a_j and c_j. Give an expression for computing a_j and c_j using matrix and vector operations, without explicit loops. This is called "vectorization" and can lead to dramatic speedup when implemented in languages such as Python, Matlab, and R. Write your expressions using X, w, $y = (y_1, \ldots, y_n)^T$ (the column vector of responses), $X_{\cdot j}$ (the jth column of X, represented as a column matrix), and w_j (the jth coordinate of w – a scalar).

   For this question, we got the $a_j = 2\sum_{i=1}^n x_{ij}^2$ and $c_j = 2\sum_{i=1}^n x_{ij}(y_i - w^T x_i + w_j x_{ij})$. So, my result is:

$$a_j = 2\sum_{i=1}^{n} x_{ij}^2 = 2X_{\cdot j}^T X_{\cdot j}$$

$$c_j = 2\sum_{i=1}^{n} x_{ij}(y_i - w^T x_i + w_j x_{ij}) = 2X_{\cdot j}^T(y - Xw + w_j X_{\cdot j})$$

2. Write a function that computes the Lasso solution for a given λ using the shooting algorithm described above. For convergence criteria, continue coordinate descent until a pass through the coordinates reduces the objective function by less than 10^{-8}, or you have taken 1000 passes through the coordinates. **Compare performance of cyclic coordinate descent to randomized coordinate descent**, where in each round we pass through the coordinates in a different random order (for your choices of λ). Compare also the solutions attained (following the convergence criteria above) for **starting at** 0 **versus starting at the ridge regression solution suggested by Murphy (again, for your choices of** λ**)**. If you like, you may adjust the convergence criteria to try to attain better results (or the same results faster).

```
# soft function follow the instructor
def soft_func(a,delta):
    sign = np.sign(a)
    if np.abs(a) - delta >=0:
        return np.dot(sign, np.abs(a) - delta)
    else:
        return 0

```

Function 4: soft function

```
def shooting_algorithm(X, y, w, lambda_reg = 0.01, max_steps = 1000, tor = 1e-8):
    converge = False
    steps = 0
    d = X.shape[1]
    a = np.zeros(d)
    c = np.zeros(d)
    # loss function
    loss = np.dot(np.dot(X,w) - y, np.dot(X,w) - y) + lambda_reg*np.linalg.norm(w,ord = 1)
    while converge == False and steps <= max_steps:
        loss_prev = loss
        for i in range(d):
            a[i] = 2*np.dot(X.T[i],X.T[i])
            c[i] = 2*np.dot(X.T[i],y-np.dot(X,w)+np.dot(w[i],X.T[i]))
            if a[i] ==0 and c[i] ==0:
                w[i] = 0
            else:
                w[i] = soft_func(c[i]/a[i], lambda_reg/a[i])
        loss = np.dot(np.dot(X,w) - y, np.dot(X,w) - y) + lambda_reg*np.linalg.norm(w,ord = 1)
        change = loss_prev - loss
        if np.abs(change)>=tor:
            converge = False
        else:
            converge = True
        steps += 1
    return a,c,w

```

Function 5: cycle shooting function

```
def random_shooting_algorithm(X, y, w, lambda_reg = 0.01, max_steps = 1000, tor = 1e-8):
    converge = False
    steps = 0
    d = X.shape[1]
    a = np.zeros(d)
    c = np.zeros(d)
    # loss function
    loss = np.dot(np.dot(X,w) - y, np.dot(X,w) - y) + lambda_reg*np.linalg.norm(w,ord = 1)
    while converge == False and steps <= max_steps:
        loss_prev = loss
        random = np.arange(X.shape[0])
        np.random.shuffle(random)
        X = X[random]
        y = y[random]
        for i in range(d):
            a[i] = 2*np.dot(X.T[i],X.T[i])
            c[i] = 2*np.dot(X.T[i],y-np.dot(X,w)+np.dot(w[i],X.T[i]))
            if a[i] ==0 and c[i] ==0:
                w[i] = 0
            else:
                w[i] = soft_func(c[i]/a[i], lambda_reg/a[i])
        loss = np.dot(np.dot(X,w) - y, np.dot(X,w) - y) + lambda_reg*np.linalg.norm(w,ord = 1)
        change = loss_prev - loss
        if np.abs(change)>=tor:
            converge = False
        else:
            converge = True
        steps += 1
    return a,c,w

```

Function 6: random shooting function

```
    #lambda equal to 0.01 which is recommend
    w = shooting_algorithm(X_train, y_train, w_opt, lambda_reg = 0.01, max_steps = 1000, tor = 1e
    ↪ -8)[2]
    loss_val = np.dot(np.dot(X_val,w) - y_val,np.dot(X_val,w) - y_val)/X.shape[0]
    print("Murphy loss value from cycle shooting algorithm with lambda equal to 0.01\t" + str(loss_val))
    w = shooting_algorithm(X_train, y_train, np.zeros(400), lambda_reg = 0.01, max_steps = 1000, tor
    ↪ = 1e-8)[2]
    loss_val = np.dot(np.dot(X_val,w) - y_val,np.dot(X_val,w) - y_val)/X.shape[0]
    print("from zero loss value from cycle shooting algorithm with lambda equal to 0\t" + str(loss_val))
    w = random_shooting_algorithm(X_train, y_train, w_opt, lambda_reg = 0.01, max_steps = 1000,
    ↪ tor = 1e-8)[2]
    loss_val = np.dot(np.dot(X_val,w) - y_val,np.dot(X_val,w) - y_val)/X.shape[0]
    print("Murphy loss value from random shooting algorithm with lambda equal to 0\t" + str(loss_val))
```

```
    w = random_shooting_algorithm(X_train, y_train, np.zeros(400), lambda_reg = 0.01, max_steps =
      ↪ 1000, tor = 1e−8)[2]
    loss_val = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
    print("from zero loss value from random shooting algorithm with lambda equal to 0.01\t" + str(
      ↪ loss_val))

    #lambda equal to 0.1
    w = shooting_algorithm(X_train, y_train, w_opt, lambda_reg = 0.1, max_steps = 1000, tor = 1e
      ↪ −8)[2]
    loss_val = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
    print("Murphy loss value from cycle shooting algorithm with lambda equal to 0.1\t" + str(loss_val))
    w = shooting_algorithm(X_train, y_train, np.zeros(400), lambda_reg = 0.1, max_steps = 1000, tor
      ↪ = 1e−8)[2]
    loss_val = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
    print("from zero loss value from cycle shooting algorithm with lambda equal to 0\t" + str(loss_val))
    w = random_shooting_algorithm(X_train, y_train, w_opt, lambda_reg = 0.1, max_steps = 1000,
      ↪ tor = 1e−8)[2]
    loss_val = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
    print("Murphy loss value from random shooting algorithm with lambda equal to 0.1\t" + str(loss_val))
    w = random_shooting_algorithm(X_train, y_train, np.zeros(400), lambda_reg = 0.1, max_steps =
      ↪ 1000, tor = 1e−8)[2]
    loss_val = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
    print("from zero loss value from random shooting algorithm with lambda equal to 0.01\t" + str(
      ↪ loss_val))

```

Function 7: compute loss

The results are:

```
Murphy loss value from cycle shooting algorithm with lambda equal to 0.01       0.14657656248948403
from zero loss value from cycle shooting algorithm with lambda equal to 0       0.18441464087338894
Murphy loss value from random shooting algorithm with lambda equal to 0 0.14657661099101094
from zero loss value from random shooting algorithm with lambda equal to 0.01   0.18441464087338919
Murphy loss value from cycle shooting algorithm with lambda equal to 0.1        0.13138214483183014
from zero loss value from cycle shooting algorithm with lambda equal to 0       0.16865931484348173
Murphy loss value from random shooting algorithm with lambda equal to 0.1       0.13138215035777645
from zero loss value from random shooting algorithm with lambda equal to 0.01   0.16865931484348184
```

Figure 2: different lambda

For the first, I test with the lambda which we choose before is 0.01, and got the risk with the cycle shooting algorithm and the random shooting algorithm, and random shooting algorithm are similar error, and cycle shooting algorithm will give us a little small error. Compare the Murphy method and start from 0, Murphy will give us a small risk, and for cycle shooting algorithm, Murphy method result is 0.14657656248948403, but the initial w with 0 will have result: 0.18441464087338894. Also you can see the figure 2, I also test with lambda equal to 0.1, Murphy will give us a better result as well.

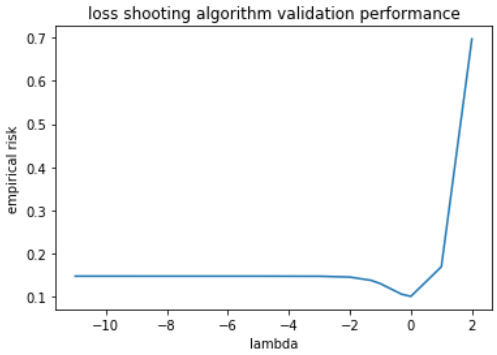
3. Run your best Lasso configuration on the training dataset provided, and **select the λ that minimizes the square error on the validation set**. Include a **table** of the parameter values you tried and the validation performance for each. Also include a **plot** of these results. Similarly, add the lasso coefficients to the bar charts of coefficients generated in the ridge regression setting. Comment on the results, with particular attention to parameter sparsity and how the ridge and lasso solutions compare. What's the best model you found, and what's its validation performance?

```
l1reg = [1e−11, 1e−9, 1e−7,1e−5,1e−3,1e−2,0.05, 0.1, 0.5, 1.0, 10, 100]
    error_record = []
    for lambda_ in l1reg:
        w = random_shooting_algorithm(X_train, y_train, w_opt, lambda_reg = lambda_, max_steps =
    ↪  1000, tor = 1e−8)[2]
        error = np.dot(np.dot(X_val,w) − y_val,np.dot(X_val,w) − y_val)/X.shape[0]
        error_record.append(error)
    error_df = pd.DataFrame(columns = ['lambda','empirical risk'])
    error_df['lambda'] = l1reg
    error_df['empirical risk'] = error_record
    print(error_df)
    #plot
    plt.figure()
    plt.plot(np.log10(error_df['lambda']),error_df['empirical risk'])
    plt.xlabel("lambda")
    plt.ylabel("empirical risk")
    plt.title("loss shooting algorithm validation performance")
    plt.savefig("different lambda.jpg")

```

Function 8: compute loss with different lambda

	lambda	empirical risk
0	1.000000e-11	0.148393
1	1.000000e-09	0.148387
2	1.000000e-07	0.148387
3	1.000000e-05	0.148387
4	1.000000e-03	0.148172
5	1.000000e-02	0.146285
6	5.000000e-02	0.138692
7	1.000000e-01	0.131091
8	5.000000e-01	0.106632
9	1.000000e+00	0.101652
10	1.000000e+01	0.170187
11	1.000000e+02	0.696648

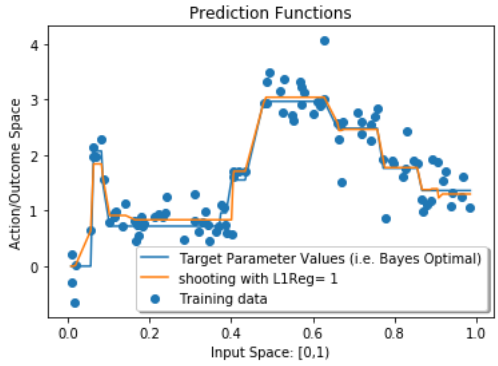
(a) table



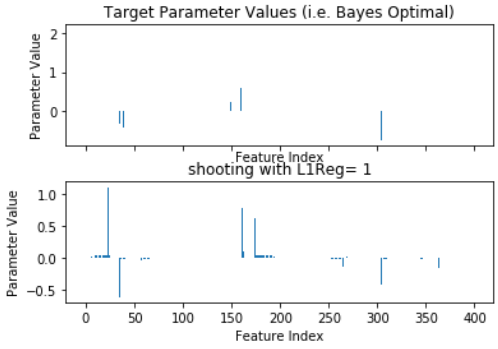
(b) plot

```
# Let's plot prediction functions and compare coefficients for several fits
  # and the target function.
  pred_fns = []
  x = np.sort(np.concatenate([np.arange(0.1,0.001), x_train]))
  name = "Target Parameter Values (i.e. Bayes Optimal)"
  pred_fns.append({"name":name, "coefs":coefs_true, "preds": target_fn(x) })

  X = featurize(x)
  w = random_shooting_algorithm(X_train, y_train, ridge_regression_estimator.w_, lambda_reg = 1,
      ↪ max_steps = 1000, tor = 1e-8)[2]
  name = "shooting with L1Reg= 1"
  pred_fns.append({"name":name,
          "coefs":w,
          "preds": np.dot(X,w)})

  f = plot_prediction_functions(x, pred_fns, x_train, y_train, legend_loc="best")
  f.show()

  f = compare_parameter_vectors(pred_fns)
  f.show()

```

Function 9: prediction function with losso shooting algorithm



(c) prediction function with losso



(d) parameter

In my implement, lambda equal to 1 gives me the least square error on validation set. And the error is 0.1017. The lasso is very sparse compared with ridge solution, looking at the prediction function plot. And lasso have large scale then ridge.

4. Implement the homotopy method described above. Compute the Lasso solution for (at least) the regularization parameters in the set $\{\lambda = \lambda_{\max} 0.8^i \mid i = 0, \ldots, 29\}$. **Plot the results (average validation loss vs λ).**

```
"""
    question 3.4
    """
    lambda_max = max(2*np.abs(X_train.T.dot(y_train)))
    lambda_lasso= [lambda_max*0.8**i for i in range(30)]
    lambda_lasso
    w=np.zeros((30,400))
    loss=np.zeros(30)
    for i in range(30):
        # shooting algorithm
        w[i]= shooting_algorithm(X_train,y_train,w[i-1],lambda_lasso[i],max_steps = 1000,tor = 1e-8)
      ↪ [2]
        # compute the loss
        loss[i] = (1/X_train.shape[0])*np.dot(np.dot(X_val,w[i])-y_val,np.dot(X_val,w[i])-y_val)
    print(loss)
    error_df = pd.DataFrame(columns = ['lambda','empirical risk'])
    error_df['lambda'] = lambda_lasso
    error_df['empirical risk'] = loss
    plt.figure()
    plt.plot(error_df['lambda'],error_df['empirical risk'])
    plt.xlabel("lambda")
    plt.ylabel("empirical risk")
    plt.title("loss shooting algorithm")
    plt.savefig("question3_4.jpg")

```

Function 10: Implement the homotopy method

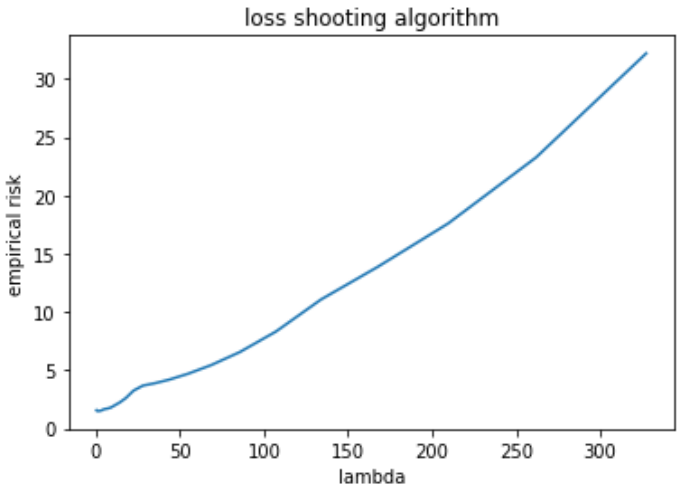


Figure 3: Implement the homotopy method

2.2 Optional: Deriving $\lambda_{\max}$

In this problem we will derive an expression for $\lambda_{\max}$. Use the Lasso objective function excluding the bias term i.e, $J(w) = \|Xw - y\|_2^2 + \lambda \|w\|_1$. We will show that for any $\lambda \geq 2\|X^T y\|_\infty$, the estimated weight vector $\hat{w}$ is entirely zero, where $\|\cdot\|_\infty$ is the infinity norm (or supremum norm), which is the maximum absolute value of any component of the vector.

1. The one-sided directional derivative of $f(x)$ at x in the direction v is defined as:

$$f'(x; v) = \lim_{h \downarrow 0} \frac{f(x + hv) - f(x)}{h}$$

Compute $J'(0; v)$. That is, compute the one-sided directional derivative of $J(w)$ at $w = 0$ in the direction v. [Hint: the result should be in terms of $X, y, \lambda,$ and v.]
For this question, looking at the

$$\begin{aligned} J(w) &= \|Xw - y\|_2^2 + \lambda \|w\|_1 \\ &= (Xw - y)^T (Xw - y) + \lambda \|w\|_1 \end{aligned}$$

Then using the $f'(x; v) = \lim_{h \downarrow 0} \frac{f(x+hv) - f(x)}{h}$

$$\begin{aligned} J'(x; w) &= \lim_{h \downarrow 0} \frac{J(x + hw) - J(x)}{h} \\ &= \lim_{h \downarrow 0} \frac{X^T X h^2 v^T v - 2h v^T X^T y + y^T y - \lambda \|v\|_1}{h} \\ &= -2v^T X^T y - \lambda \|v\|_1 \end{aligned}$$

2. Since the Lasso objective is convex, w^* is a minimizer of $J(w)$ if and only if the directional derivative $J'(w^*; v) \geq 0$ for all $v \neq 0$. Show that for any $v \neq 0$, we have $J'(0; v) \geq 0$ if and only if $\lambda \geq C$, for some C that depends on $X, y,$ and v. You should have an explicit expression for C. For this question: first, from last question we got:

$$\begin{aligned} J'(x; w) &= \lim_{h \downarrow 0} \frac{J(x + hw) - J(x)}{h} \\ &= -2v^T X^T y - \lambda \|v\|_1 \end{aligned}$$

And we want $J'(x; w) \geq 0$, then:

$$-2v^T X^T y - \lambda \|v\|_1 \geq 0$$

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

So $C = \frac{2v^T X^T y}{\|v\|_1}$

3. In the previous problem, we get a different lower bound on λ for each choice of v. Show that the maximum of these lower bounds on λ is $\lambda_{\max} = 2\|X^T y\|_\infty$. Conclude that $w = 0$ is a minimizer of $J(w)$ if and only if $\lambda \geq 2\|X^T y\|_\infty$.
   We need to prove that $w = 0$ is a minimizer of $J(x)$ if and only if $\lambda \geq 2\|X^T y\|_\infty$

$$-2v^T X^T y - \lambda \|v\|_1 \geq 0$$

$$\lambda \geq \frac{2v^T X^T y}{\|v\|_1} for\ all\ v \neq 0$$

   which is euqal to

$$= 2\ max\ v^T (X^T y) for\ v : \|v\|_1 = 1$$
$$= 2\lambda \left\|X^T y\right\|_\infty$$

   So we can conclude that $w = 0$ is a minimizer of $J(x)$ if and only if $\lambda \geq 2\|X^T y\|_\infty$.

3 Projected SGD via Variable Splitting

In this question, we consider another general technique that can be used on the Lasso problem. We first use the variable splitting method to transform the Lasso problem to a differentiable problem with linear inequality constraints, and then we can apply a variant of SGD.

Representing the unknown vector θ as a difference of two non-negative vectors θ^+ and θ^-, the ℓ_1-norm of θ is given by $\sum_{i=1}^{d} \theta_i^+ + \sum_{i=1}^{d} \theta_i^-$. Thus, the optimization problem can be written as

$$(\hat{\theta}^+, \hat{\theta}^-) = \underset{\theta^+, \theta^- \in \mathbf{R}^d}{\arg\min} \sum_{i=1}^{m} (h_{\theta^+, \theta^-}(x_i) - y_i)^2 + \lambda \sum_{i=1}^{d} \theta_i^+ + \lambda \sum_{i=1}^{d} \theta_i^-$$
$$\text{such that } \theta^+ \geq 0 \text{ and } \theta^- \geq 0,$$

where $h_{\theta^+, \theta^-}(x) = (\theta^+ - \theta^-)^T x$. The original parameter θ can then be estimated as $\hat{\theta} = (\hat{\theta}^+ - \hat{\theta}^-)$.

This is a convex optimization problem with a differentiable objective and linear inequality constraints. We can approach this problem using projected stochastic gradient descent, as discussed in lecture. Here, after taking our stochastic gradient step, we project the result back into the feasible set by setting any negative components of θ^+ and θ^- to zero.

1. Implement projected SGD to solve the above optimization problem for the same λ's as used with the shooting algorithm. Since the two optimization algorithms should find essentially the same solutions, you can check the algorithms against each other. Report the differences in validation loss for each λ between the two optimization methods. (You can make a table or plot the differences.)

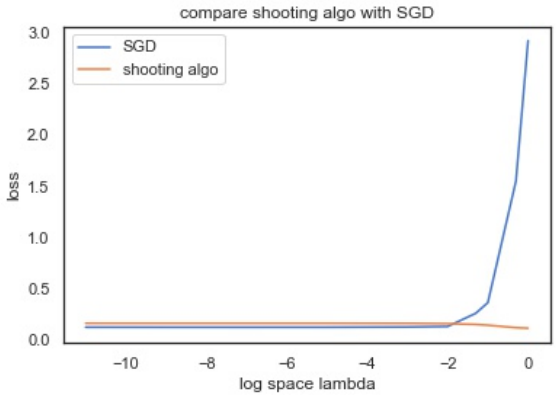
```
''' Implement stochastic gradient descent '''
num_instances, num_features = X.shape[0], X.shape[1]
theta_positive = np.zeros(num_features)
theta_negative = np.zeros(num_features)
loss_hist =np.zeros(max_iter)

# implement same with SGD assignment but with theta_positive and theta_negative
def compute_stochastic_gradient(x_i, y_i, theta_positive, theta_negative, lambda_reg):
    num_features = len(x_i)
    predict = x_i.dot(theta_positive - theta_negative)
    error = y_i - predict
    #gradient for theta_positive
    grad_1 =  x_i * -1 * error  + 2*lambda_reg * np.ones(num_features)
    #gradient for theta_negative
    grad_2 =  x_i * error  + 2*lambda_reg * np.ones(num_features)
    return grad_1, grad_2


```

```
#return the max theta
def max(array):
    for i, theta in enumerate(array):
        array[i] = max(theta, 0)
    return array
def SGD(X, y, alpha='1/t', lambda_reg =0.01, max_iter=1000 , tor=1e-8):
    for i in np.arange(max_iter):
        theta_old = theta_positive - theta_negative
        index = np.random.permutation(num_instances)
        for it,ix in enumerate(index):
            x_i, y_i = X[ix],y[ix]
            #implement same with SGD assignment
            if type(alpha)==float:
                step_size = alpha
            elif alpha == '1/t':
                step_size = 1.0/(num_instances*i+it+1)
            else:
                step_size = 1.0/np.sqrt(num_instances*i+it+1)

            grad_1, grad_2 = compute_stochastic_gradient(x_i, y_i, theta_positive,theta_negative,
    ↪ lambda_reg)
            theta_positive -= step_size * grad_1
            theta_negative -= step_size * grad_2
            theta_positive = max(theta_positive)
            theta_negative = max(theta_negative)
        theta = theta_positive - theta_negative
        loss_hist[i] = np.dot(np.dot(X,theta) - y,np.dot(X,theta) - y)/X.shape[0]
        diff =   diff = np.linalg.norm(theta_old - theta,ord=1)
        if diff < tor:
            print ('finish %d iteration' %i)
            break
        return theta, loss_hist

result_SGD =[]
reg = [1e-11, 1e-9, 1e-7,1e-5,1e-3,1e-2,0.05, 0.1, 0.5, 1.0]
for lambda_ in reg:
    theta, loss_hist= SGD_lasso(X_train, y_train,alpha =0.0001,lambda_reg=lambda_,max_iter=1000)
    loss = np.dot(np.dot(X_val,theta) - y_val,np.dot(X_val,theta) - y_val)/X.shape[0]
    result_SGD.append(loss)
```

Function 11: Implement the homotopy method

```
error_df = pd.DataFrame(columns = ['lambda','SGD ','ridge regression'])
error_df['lambda'] = l2reg
error_df['SGD'] = result_SGD
error_df['ridge regression'] = error_record
print(error_df)
#plot
plt.figure()
plt.xlabel('log space lambda')
plt.ylabel('loss')
plt.title('compare shooting algo with SGD')
plt.plot(np.log10(error_df['lambda']),error_df['SGD'], label = 'SGD')
plt.plot(np.log10(error_df['lambda']),error_df['ridge regression'], label = 'shooting algo')
plt.legend()
plt.savefig("SGD.jpg")
```

Function 12: plot compare



(a) plot: SGD compare with shooting algorithm

	lambda	SGD	ridge regression	SGD
0	1.000000e-11	NaN	0.159531	0.121112
1	1.000000e-09	NaN	0.159531	0.120774
2	1.000000e-07	NaN	0.159531	0.120592
3	1.000000e-05	NaN	0.159531	0.120852
4	1.000000e-03	NaN	0.158845	0.123136
5	1.000000e-02	NaN	0.155780	0.127872
6	5.000000e-02	NaN	0.148049	0.257009
7	1.000000e-01	NaN	0.140659	0.361556
8	5.000000e-01	NaN	0.116613	1.552560
9	1.000000e+00	NaN	0.110847	2.926271

(b) table: SGD compare with shooting algorithm

The difference we can see in the plot, both two algorithms can get the least loss with lambda around to 0.01 or smaller than 0.01. The loss from using SGD will high than using shooting algorithm. When lambda >= 0.01, using SGD will highly increase the loss.

2. How does the sparsity compare to the solution from the shooting algorithm?
   Sparse is better than shooting algorithm.