Mini project #3

算法设计与分析

16337266 徐原

Part1

(a) (4 points) Least-squares regression has the closed form solution $\mathbf{a} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, which minimizes the squared error on the data. (Here $\mathbf{X}$ is the $n \times d$ data matrix as in the code above, with one row per data point, and $\mathbf{y}$ is the n-vector of their labels.) Solve for $\mathbf{a}$ and report the value of the objective function using this value $\mathbf{a}$. For comparison, what is the total squared error if you just set $\mathbf{a}$ to be the all 0's vector?

Comment: Computing the closed-form solution requires time $O(nd^2 + d^3)$, which is slow for large d. Although gradient descent methods will not yield an exact solution, they do give a close approximation in much less time. For the purpose of this assignment, you can use the closed form solution as a good sanity check in the following parts.

Solution:

```
d=100
n=1000
X=np.random.normal(0,1,size=(n,d))
a_true=np.random.normal(0,1,size=(d,1))
y=X.dot(a_true)+np.random.normal(0,0.5,size=(n,1))

def get_a(flag):
    if flag:
        return np.zeros(shape=(d,1))
    Xt=X.T
    XtX=np.matmul(Xt,X)
    XtXinv=np.linalg.inv(XtX)
    XtXinvXt=np.matmul(XtXinv,Xt)
    a=np.matmul(XtXinvXt,y)#a:d*1
    return a

def get_error(a):
    sqrd_error=0
    at=a.T# at:1*d
    for i in range(n):
        xi=X[i].reshape((d,1))
        sqrd_error+=np.power(at.dot(xi)-y[i],2)
    return sqrd_error

def part_a():
    flag=0
    a=get_a(flag)
    error=get_error(a)
    flag=1
    aa=get_a(flag)
    errorr=get_error(aa)
    print("normal error:",error)
    print("set a all 0's error:",errorr)

#part_a
part_a()
```

Result:

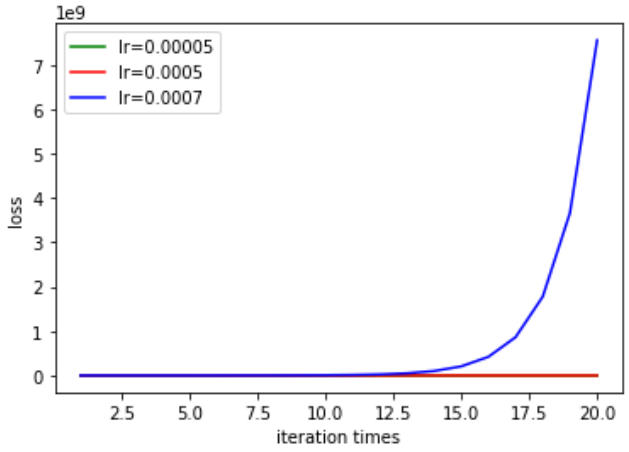
```
normal error: [[226.8584088]]
set a all 0's error: [[86613.31172669]]
```

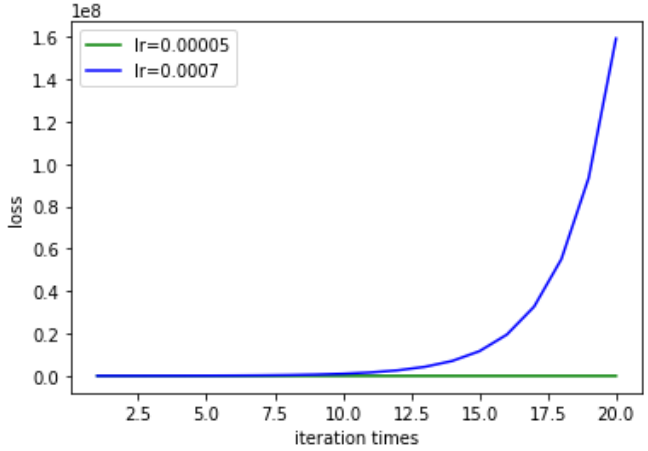
So, the total squared error is 86613.31172669 if just set **a** to be the all 0's vector.(This result has randomness)

(b) (6 points) In this part, you will solve the same problem via gradient descent on the squared-error objective function $f(\mathbf{a}) = \sum_{i=1}^{n} f_i(\mathbf{a})$. Recall that the gradient of a sum of functions is the sum of their gradients. Given a point $\mathbf{a}_t$, what is the gradient of f at $\mathbf{a}_t$?

Now use gradient descent to find a coefficient vector **a** that approximately minimizes the least squares objective function over the data. Run gradient descent three times, once with each of the step sizes 0:00005, 0:0005, and 0:0007. You should initialize **a** to be the all-zero vector for all three runs. Plot the objective function value for 20 iterations for all 3 step sizes on the same graph. Comment in 3-4 sentences on how the step size can affect the convergence of gradient descent (feel free to experiment with other step sizes). Also report the step size that had the best final objective function value and the corresponding objective function value.

```python
#part_b
learning_rates=[0.00005, 0.0005, 0.0007]
iterations=20
'''
y=(y_predict-y_true)^2
y'=2*y_predict-y_true
'''
def gradient_descent():
    gdresult = {}
    for i in learning_rates:
        a=np.zeros(shape=(d,1))
        for j in range(iterations):
            gd=0
            for elem in range(n):
                xi=X[elem].reshape((d,1))
                gd+=2*xi*(a.T.dot(xi)-y[elem])
            a-=i*gd
            loss=get_error(a)
            gdresult.setdefault(i,[]).extend(loss)
    return gdresult

def make_plot(gdresult,iterations,outFileName):
    xlabs=[i for i in range(1,iterations+1)]
    plt.xlabel('iteration times')
    plt.ylabel('loss')
    plt.plot(xlabs,gdresult[learning_rates[0]],color='green',label='lr=0.00005')
    plt.plot(xlabs,gdresult[learning_rates[1]],color='red',label='lr=0.0005')
    plt.plot(xlabs,gdresult[learning_rates[2]],color='blue',label='lr=0.0007')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig(outFileName, format = 'png')

gdresult=gradient_descent()
make_plot(gdresult,iterations,"part_b.png")
```





(To be more clear)

The calculate function is : f'(x)=2* $\sum_{k=1}^{n} x^k(a^T x^k - y^k)$

From result , we can see The loss of 0.0007 is most serious.
And the objective loss value of the three learn rates are:

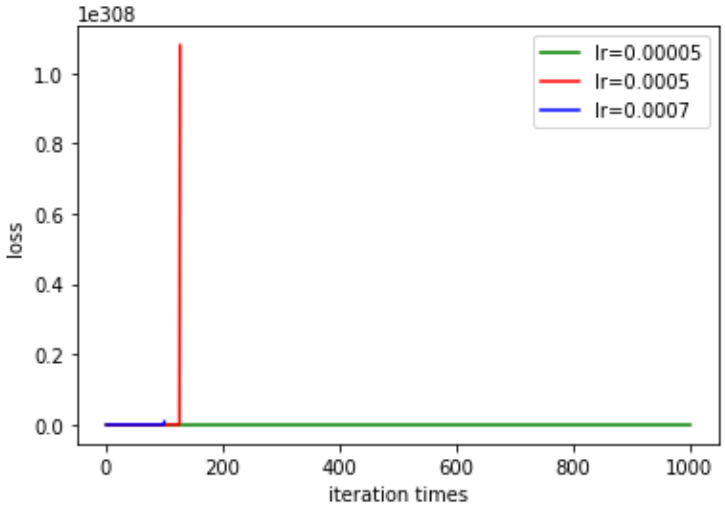
```
0.00005: [2669.76564882]
0.0005: [239.25296324]
0.0007: [1.07103966e+08]
```

Solution:

If the step is too small, it will not loss right direction, but the convergence speed is too low. Like step 0.000005, although we have calculated 20 times, it still be away from best a .But if the strp is too large, it may have wrong direction. Like the step 0.0007. With times increasing,it has larger loss. So we choose the 0.0005 as the best step size. It has right direction and good convergence speed. Finally , we got the best loss value:239.25296324.

(c) (6 points) In this part you will run stochastic gradient descent to solve the same problem. Recall that in stochastic gradient descent, you pick one datapoint at a time, say (**x**(i)*;* *y*(i)), and update your current value of **a** according to the gradient of *fi*(**a**) = (**a**T**x**(i) - *y*(i))2.
Run stochastic gradient descent using step sizes f0.0005*;* 0.005*;* 0.01g and 1000 iterations. Plot the objective function value vs. the iteration number for all 3 step sizes on the same graph. Comment 3-4 sentences on how the step size can affect the convergence of stochastic gradient descent and how it compares to gradient descent. Compare the performance of the two methods. How do the best final objective function values compare? How many times does each algorithm use each data point? Also report the step size that had the best final objective function value and the corresponding objective function value.

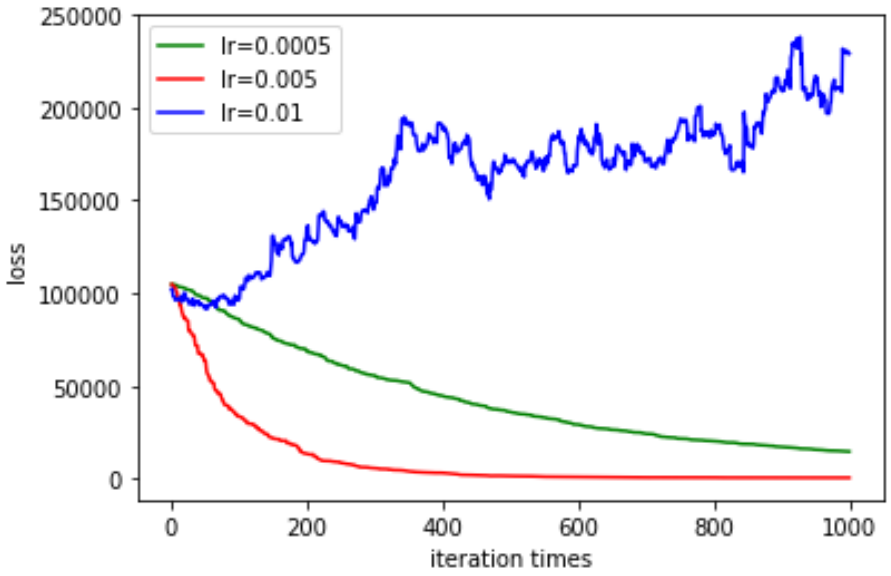
Solution:
Firstly, I try to use the same a like part_b. the loss has overflow upper bound.



Obviously, it is wrong method.

So, we use stochastic gradient descent.

```
#part_c
learning_rates=[0.0005,0.005, 0.01]
iterations=1000
def stochastic_gradient_descent():
    gdresult = {}
    for i in learning_rates:
        a=np.zeros(shape=(d,1))
        for j in range(iterations):
            random_choose=np.random.randint(0,n-1)
            xi=X[random_choose].reshape((d,1))
            gd=2*xi*(a.T.dot(xi)-y[random_choose])
            a-=i*gd
            loss=get_error(a)
            gdresult.setdefault(i,[]).extend(loss)
    return gdresult

gdresult=stochastic_gradient_descent()
make_plot(gdresult,iterations,"part_c.png")
print("0.0005:",gdresult[learning_rates[0]][-1])
print("0.005:",gdresult[learning_rates[1]][-1])
print("0.01:",gdresult[learning_rates[2]][-1])
```




```
0.0005: [14578.88111626]
0.005: [417.52891107]
0.01: [229022.27019279]
```

Solution:

From the result, we can got when lr=0.005, the function has best performance. The same to partB, when lr is small, it has right direction but its convergence speed is too low. When lr is large, it is apt to lose right direction. With times increasing, the loss will become larger.The step 0.005 is still a good choice. It has right direction and best convergence speed.

Each data point is used once on average because there are 1000 iterations and 1000 data points.

Part2

(a) (2 points) We will first setup a baseline, by finding the test error of the linear regression solution a = X-1y without any regularization. This is the closed-form solution for the minimizer of the objective function f(a). (Note the formula is simpler than in 1(a) because now X is square.) Report the training error and test error of this approach, averaged over 10 trials. For better interpretability, report the normalized test error f^(a) rather than the value of the objective function f(a), where by definition

$$\hat{f}(\mathbf{a}) = \frac{\| \mathbf{Xa} - \mathbf{y} \|_2}{\| \mathbf{y} \|_2}.$$

```
def f_error(X,a,y):
    n=np.matmul(X,a)
    n-=y
    n=math.sqrt(np.sum(np.square(n)))
    d=math.sqrt(np.sum(np.square(y)))
    return float(n)/d

def get_a(X_train,y_train):
    inv = np.linalg.inv(X_train)
    a = np.matmul(inv, y_train)
    return a

def part_2(func,choice,c):
    train_err=0
    test_err=0

    for i in range(trials):
        X_train = np.random.normal(0,1, size=(train_n,d))
        a_true = np.random.normal(0,1, size=(d,1))
        y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
        X_test = np.random.normal(0,1, size=(test_n,d))
        y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))
        if choice=='a':
            a=func(X_train,y_train)
        else:
            a=func(X_train,y_train,c)
        tra=f_error(X_train,a,y_train)
        tes=f_error(X_test,a,y_test)
        train_err+=tra
        test_err+=tes
    train_err=float(train_err)/trials
    test_err=float(test_err)/trials
```

```
train error: 2.882382222311432e-14
test error: 1.0103870540770568
```

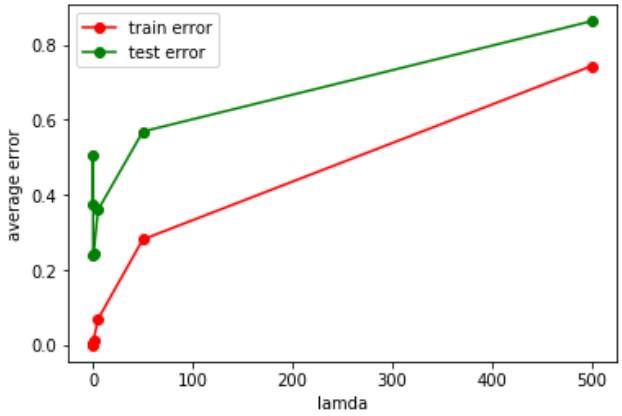
(b) (5 points) We will now examine '**2** regularization as a means to prevent overfitting. The '**2** regularized objective function is given by the following expression:

$$\sum_{i=1}^{m} (\mathbf{a}^{\mathbf{T}}\mathbf{x}^{(\mathbf{i})} - y^{(i)})^2 + \lambda \| \mathbf{a} \|_2^2 .$$

This has a closed-form solution $a = (XTX + \lambda I)\text{-}1XTy$. Using this closed-form solution, present a plot of the normalized training error and normalized test error f^(a) for λ = f0:0005; 0:005; 0:05; 0:5; 5; 50; 500g.As before, you should average over 10 trials. Discuss the characteristics of your plot, and also compare it to your answer to (a)。

```
def get_a2(X, y,C):
    res=np.matmul(X.T, X) + C * np.identity(d)
    res=np.linalg.inv(res)
    a = np.matmul(res, X.T)
    a = np.matmul(a, y)
    return a


def part_2b():
    error_tra=[]
    error_tes=[]
    for i in c:
        tra,tes=part_2(get_a2,'b',i)
        error_tra.append(tra)
        error_tes.append(tes)
    plt.xlabel('lamda')
    plt.ylabel('average error')
    plt.plot(c,error_tra,color='red',label='train error',marker='o')
    plt.plot(c,error_tes,color='green',label='test error',marker='o')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_b", format = 'png')
'''
part_2b()
'''
```

```
part b train error: 0.00038057850645065806
part b test error: 0.3734706357531815
part b train error: 0.0029658224110029625
part b test error: 0.5074906549170194
part b train error: 0.0062282000695934
part b test error: 0.24099396877376095
part b train error: 0.01436489626960405
part b test error: 0.2438324613916946
part b train error: 0.06992605546498847
part b test error: 0.360644718883716
part b train error: 0.2818764269616459
part b test error: 0.5691640816410323
part b train error: 0.7433535012773835
part b test error: 0.8627740018327075
```



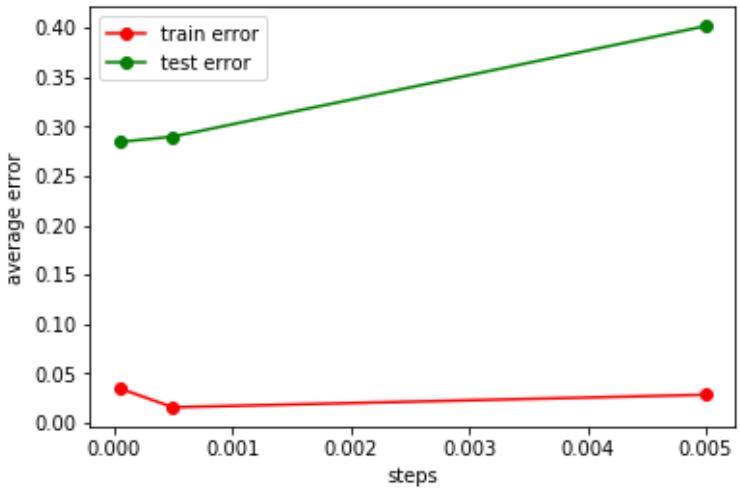
Solution:

From this figure we can get:

Compared part a, both train error and test error have decreased, especially train error. But different lamda have different impact on data. If the value of lamda is too large, the error increases. if the value of lamda is small and not equal to 0, it has good impact on test . Such as lamdas be tested,0.005 is a good choice. Regularization truly matters with the suitable lamda.

(c) (5 points) Run stochastic gradient descent (SGD) on the original objective function f(**a**), with the initial guess of **a** set to be the all 0's vector. Run SGD for 1,000,000 iterations for each different choice of the step size, f0:00005; 0:0005; 0:005g. Report the normalized training error and the normalized test error for each of these three settings, averaged over 10 repetitions/trials. How does the SGD solution compare with the solutions obtained using '2 regularization? Note that SGD is minimizing the original objective function, which does *not* have any regularization. In Part (a) of this problem, we found the *optimal* solution to the original objective function with respect to the training data. How does the training and test error of the SGD solutions compare with those of the solution in (a)? Can you explain your observations? (It may be helpful to also compute the normalized training and test error corresponding to the true coefficient vector f(**a**⇤), for comparison.

```
lef part_2c():
    for step in steps:
        train_error=0
        test_error=0
        for trial in range(trials):
            X_train = np.random.normal(0,1, size=(train_n,d))
            a_true = np.random.normal(0,1, size=(d,1))
            y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
            X_test = np.random.normal(0,1, size=(test_n,d))
            y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))
            a=np.zeros(shape=(d,1))
            for i in range(iterations):
                random_i=np.random.randint(0,train_n-1)
                train=X_train[random_i].reshape((d,1))
                gradient=2*train*(a.T.dot(train)-y_train[random_i])
                a-=step*gradient
            tra=f_error(X_train,a,y_train)
            tes=f_error(X_test,a,y_test)
            train_error+=tra
            test_error+=tes
        train_error/=10
        test_error/=10
        steps_train_error.append(train_error)
        steps_test_error.append(test_error)
    plt.xlabel('steps')
    plt.ylabel('average error')
    plt.plot(steps,steps_train_error,color='red',label='train error',marker='o')
    plt.plot(steps,steps_test_error,color='green',label='test error',marker='o')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_c", format = 'png')
```



Solution:

Compared with part a , its error also has decreased. But there is special case occurring.from step 0.00005 to 0.0005, we can see the train error decreased , but the test error increased. That means although we choose small step and SGD, it also overfit. So it indicates us regularization may be a good way to avoid overfitting.

(d) (7 points) We will now examine the behavior of SGD in more detail. For step sizes f0:00005; 0:005g and 1,000,000 iterations of SGD.

(i) Plot the normalized training error vs. the iteration number. On the plot of training error, draw a line parallel to the x-axis indicating the error **f**^(a∗) of the true model a∗:

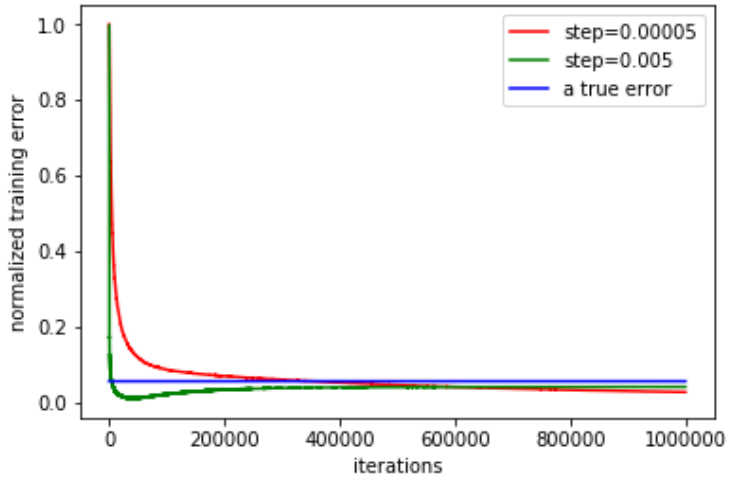
(ii) Plot the normalized test error vs. the iteration number. Your code might take a long time to run if you compute the test error after every SGD step|feel free to compute the test error every 100 iterations of SGD to make the plots.

(iii) Plot the **l**2 norm of the SGD solution vs. the iteration number.

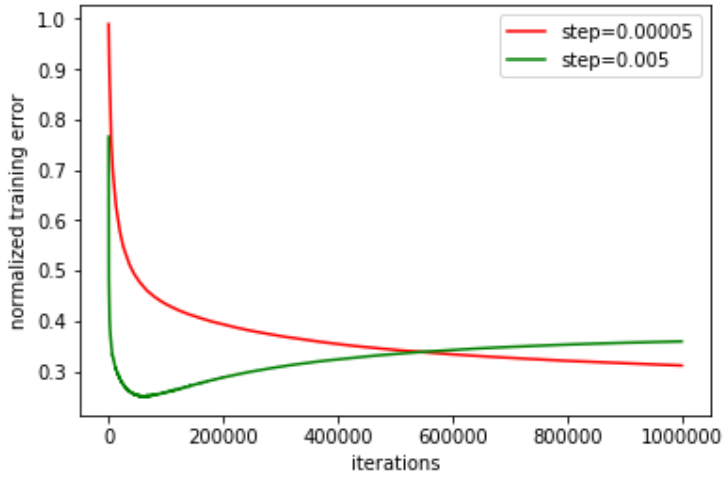
```
def part_2d():

    for step in steps:
        a=np.zeros(shape=(d,1))
        for i in range(1,iterations+1):
            random_i=np.random.randint(0,train_n-1)
            train=X_train[random_i].reshape((d,1))
            gradient=2*train*(a.T.dot(train)-y_train[random_i])
            a-=step*gradient
            error=f_error(X_train,a,y_train)
            train_errors.setdefault(step,[]).append(error)
            if i%100==0:
                error=f_error(X_test,a,y_test)
                test_errors.setdefault(step,[]).append(error)
            l2_errors.setdefault(step,[]).append(np.linalg.norm(a))
    part2_d_plot1()
    part2_d_plot2()
    part2_d_plot3()
```

```
def part2_d_plot1():
    plt.xlabel('iterations')
    plt.ylabel('normalized training error')
    plt.plot(xx,train_errors[steps[0]],color='red',label='step=0.00005')
    plt.plot(xx,train_errors[steps[1]],color='green',label='step=0.005')
    fa=[f_error(X_train,a_true,y_train) for i in range(iterations)]
    plt.plot(xx,fa,color='blue',label='a true error')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_d1.png", format = 'png')
    plt.close()
def part2_d_plot2():
    plt.xlabel('iterations')
    plt.ylabel('normalized training error')
    plt.plot(xx100,test_errors[steps[0]],color='red',label='step=0.00005')
    plt.plot(xx100,test_errors[steps[1]],color='green',label='step=0.005')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_d2.png", format = 'png')
    plt.close()
def part2_d_plot3():
    plt.xlabel('iterations')
    plt.ylabel('normalized training error')
    plt.plot(xx,l2_errors[steps[0]],color='red',label='step=0.00005')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_d3.png", format = 'png')
    plt.close()
```

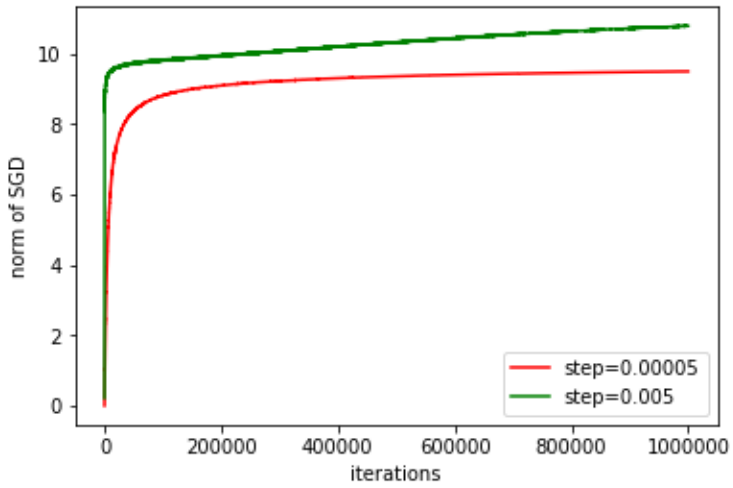
i.


step=0.00005
step=0.005
a true error
normalized training error
iterations
0.0
0.2
0.4
0.6
0.8
1.0
0
200000
400000
600000
800000
1000000

ii.


step=0.00005
step=0.005
normalized training error
iterations
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
200000
400000
600000
800000
1000000

iii.


step=0.00005
step=0.005
norm of SGD
iterations
0
2
4
6
8
10
0
200000
400000
600000
800000
1000000

Solution:

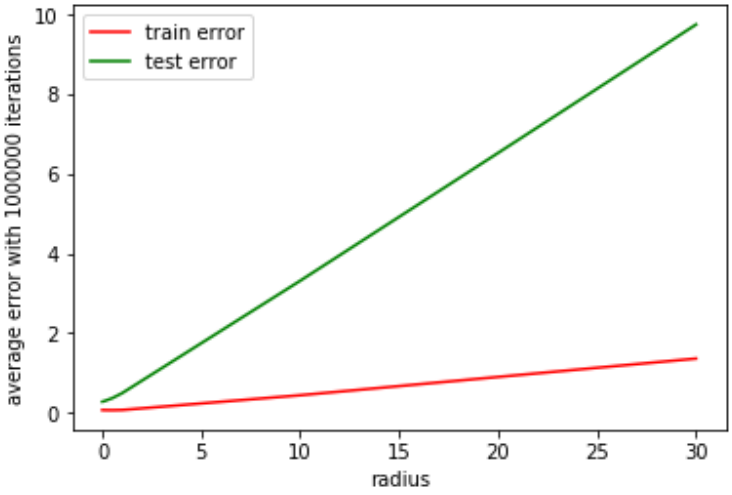
From figure i for train error, we can get that when step=0.005, It has a faster convergence speed. At the same time, it easily become overfitting (compared with true error).

From figure ii for test error, we can get that the bigger step has better performance when iterations is low, step0.00005 and step0.005 has the same impact between [400000,600000]. Although the train error begin increasing (about occur at 20000), the test error keep decreasing(lasts up 50000). It explains that train error can't symbolize test error. When train error is too low, there is overfitting case appearing. Sometimes train error is increasing but test error is decreasing.

From figure iii for norm, we get that with the iterations added, it also increases. And step=0.005 has larger norm, which means it has larger penalty factor. This can be used to avoid overfitting.

(e) (4 points) We will now examine the effect of the starting point on the SGD solution. Fixing the step size at 0:00005 and the maximum number of iterations at 1,000,000, choose the initial point randomly from the d-dimensional sphere with radius r = **f**0; 0:1; 0:5; 1; 10; 20; 30**g**, and plot the average normalized training error and the average normalized test error over 10 iterations vs r. Comment on the results, in relation to the results from part (b) where you explored different '2 regularization coefficients. Can you provide an explanation for the behavior seen in this plot?

```
def part_2e():
    for r in rs:
        a=np.ones(shape=(d,1))*r
        train_e=0
        test_e=0
        for i in range(iterations):
            random_i=np.random.randint(0,train_n-1)
            train=X_train[random_i].reshape((d,1))
            gradient=2*train*(a.T.dot(train)-y_train[random_i])
            a-=step*gradient
            error=f_error(X_train,a,y_train)
            train_e+=error
            error=f_error(X_test,a,y_test)
            test_e+=error
        train_errors.append(float(train_e)/iterations)
        test_errors.append(float(test_e)/iterations)
    plt.xlabel('radius')
    plt.ylabel('average error with 1000000 iterations')
    plt.plot(rs,train_errors,color='red',label='train error')
    plt.plot(rs,test_errors,color='green',label='test error')
    plt.legend() # 显示图例
    plt.show()
    plt.savefig("part2_e.png", format = 'png')
    plt.close()

part_2e()
```



Solution:

As can be seen from the figure, if the weight selection at the beginning is too large, the error will be relatively large. However, the influence of large weight on training error and test error is different. Training focuses on constantly adjusting weights, but it is difficult to adjust on test sets. So the size of the starting vector is also important.

Part3

(a) (11 points: 4 for performance, 7 for analysis and discussion) The goal of this problem is to achieve the best test error that you can, using the techniques from the previous two parts and/or by other means. (Of course, your learning algorithm can only use the training data for this purpose, and cannot refer to a_true.) You will receive credit based on your accuracy. Report the average test error you obtain, averaged over 1000 trials (where you re-pick a_true and the data in each trial). Feel free to use regularization, SGD, gradient descent, or any other algorithm you want to try, but clearly describe the algorithm you use in human-readable pseudo-code. **Briefly** discuss the approach you used, your thought process that informed your decisions, and the extent to which you believe a better test error is achievable. Your score will be based on a combination of the short discussion and the average test error you obtain. A paragraph of analysis is enough to earn full credit|don't go overboard unless you really want to.

From part 2b ,we choose lamda=0.005

```
def part3():
    traerror = 0.0
    teserror = 0.0
    c_for_train = 0.0005
    c_for_test=0.05
    for trial in range(trials):
        X_train = np.random.normal(0,1, size=(train_n,d))
        a_true = np.random.normal(0,1, size=(d,1))
        y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
        X_test = np.random.normal(0,1, size=(test_n,d))
        y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))
        a = get_a(X_train,y_train,c_for_train)
        tra= f_error(X_train, a, y_train)
        a = get_a(X_test,y_test,c_for_test)
        tes= f_error(X_test, a, y_test)
        traerror += tra
        teserror += tes
    ave_train_err=traerror/trials
    ave_test_err=teserror/trials
    print("my result")
    print("average train error:",ave_train_err)
    print("average test error:",ave_test_err)

part3()
```

get_a():

```
#from part2
def get_a(X, y,C):
    res=np.matmul(X.T, X) + C * np.identity(d)
    res=np.linalg.inv(res)
    a = np.matmul(res, X.T)
    a = np.matmul(a, y)
    return a
#from part 2
def f_error(X,a,y):
    n=np.matmul(X,a)
    n-=y
    n=math.sqrt(np.sum(np.square(n)))
    d=math.sqrt(np.sum(np.square(y)))
    return float(n)/d
```

Result:

```
In [1]: runfile('D:/算法分析与设计/minipro3/part3.py', wdir='D:/算法分析与设计/minipro3')
my result
average train error: 3.5921243776418618e-06
average test error: 0.03513201707169233
```

Explaination: From part2_b, we learn one function to get a

$$\mathbf{a} = (\mathbf{X}^{\mathbf{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{y}.$$

The regularization one reduce the probability of overfitting and it truly impact result as we expected. So I think it is a good way to calculate.