

Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 60

Hello everyone, this is the final lecture of the final week. In fact, this is the final lecture of this course. You may recall that this week we have been working on a machine learning application to implement ridge regression using optimization algorithms. In the previous lecture, we worked out the constrained optimization version of the ridge regression method. In this lecture, we will start with the unconstrained version.

If you look at the screen, please note that in the constrained version we had a parameter 's'. I told you that there will be a μ corresponding to each value of 's'. That μ is the μ^* that we get when we solve the Lagrangian of this particular problem. When you solve this constraint problem by hand, you have the Lagrangian $L(\hat{\beta}, \mu)$ which is the objective function plus μ times the constraint equation. When we solve this problem, we not only solve for $\hat{\beta}$ but we also solve for μ . Let us say the optimal value is μ^* .

The solution of the constrained problem and the solution of the unconstrained problem are the same if we use the μ^* that we obtained from the constrained version. That was one of the reasons why I solved the constrained problem first. Now, when we want to solve the unconstrained problem, we will take the μ^* that we obtain from the constraint problem. Recall that in one of the previous lectures, we actually found the value of μ^* for each of the values of 's'. For point one, the equivalent μ^* was 28771.69. For point two, it was 16834.78, and so on. You can also see that for the value 256, it was zero. The reason is that the constraint was not active. For inactive constraints, you can recall that the Lagrange multiplier will be zero. If you give any 's' value greater than this, you will certainly get your μ^* to be zero, and the sum of squares of the test errors will be the same, 4016.74. You can try this at home.

What we are going to do is use this value of μ^* and solve the unconstrained problem that is boxed below on the screen. Let us do that. We will only have $f(x)$. It is slightly different from the previous formulation, but given that we will use this μ^* , it will be fine. I will also write the text, which is "Ridge Regression Unconstrained". I think I should write a name here as well. Let me do that. This is "Ridge Regression using Penalty Method". All right.

This is the first part, the sum of squares of the errors. With this, we should add μ times the L2 penalty term. We will define the gradient and the Hessian as well. This is something we have defined in other places. This is exactly the gradient of the sum of squares part. For the penalty part, we need the gradient of g that we defined here. Let us take that from here. We will also define the Hessian since we will first solve it with Newton's method. The Hessian of the sum

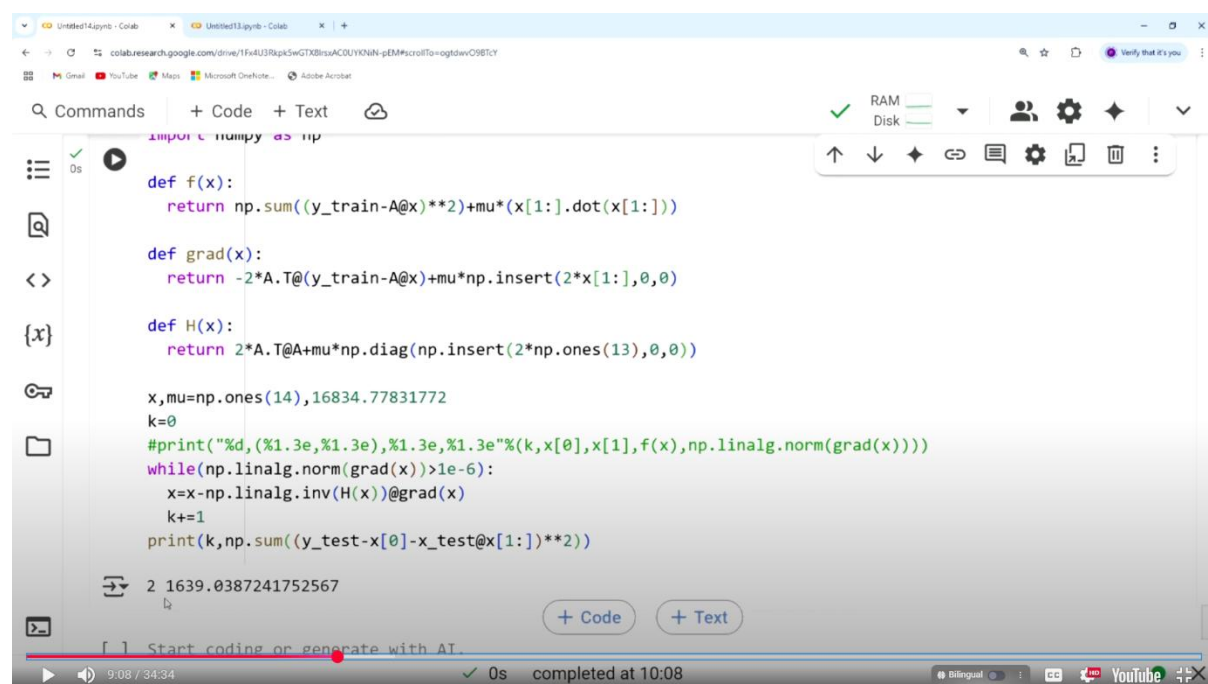
of squares is $2A^T A$. For the penalty term, we will have the Hessian we defined here, which involves `np.diag` and all that part.

Fine, we have defined $f(x)$ and its gradient and Hessian. We will also initialize the values. Since we are only going to use Newton's method first, we can possibly think about other methods a little later. This is Newton's method for general functions. The initial point is all ones, which we have defined. We will start with that. The print statements are not all required. I think this is a slightly different method, but nevertheless we can change this very easily.

Let us take this particular method. After finishing, we will print the value of k , meaning we will see how many iterations it took, and the test error which we have here. We can run the code. Since we are starting afresh, I need to import NumPy. We will have to run the code for x_{train} and all of that so that the values of x_{train} , x_{test} , y_{train} , y_{test} are all defined. I have run this now, and now we will run our function. I should have defined the value of μ . Though I mentioned it in the beginning, yes, this is the value of μ . Let us take that.

It says the size is different from fourteen. I suppose the problem is with the gradient. Let us debug this straight away. Let us print the shape of $H(x)$ and the shape of $\text{grad}(x)$. That might help to understand the issue. It is (1,14) instead of just fourteen. I think I can modify this to be the following way. I suppose now it may work. Yes, done. Now we can remove this print statement. All right.

(Refer Slide Time 9:08)



```
import numpy as np

def f(x):
    return np.sum((y_train-A*x)**2)+mu*(x[1:].dot(x[1:]))

def grad(x):
    return -2*A.T@(y_train-A*x)+mu*np.insert(2*x[1:],0,0)

def H(x):
    return 2*A.T@A+mu*np.diag(np.insert(2*np.ones(13),0,0))

x,mu=np.ones(14),16834.77831772
k=0
#print("%d, (%1.3e,%1.3e),%1.3e,%1.3e"%(k,x[0],x[1],f(x),np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
    x=x-np.linalg.inv(H(x))@grad(x)
    k+=1
print(k,np.sum((y_test-x[0]-x_test*x[1:])**2))

2 1639.0387241752567
```

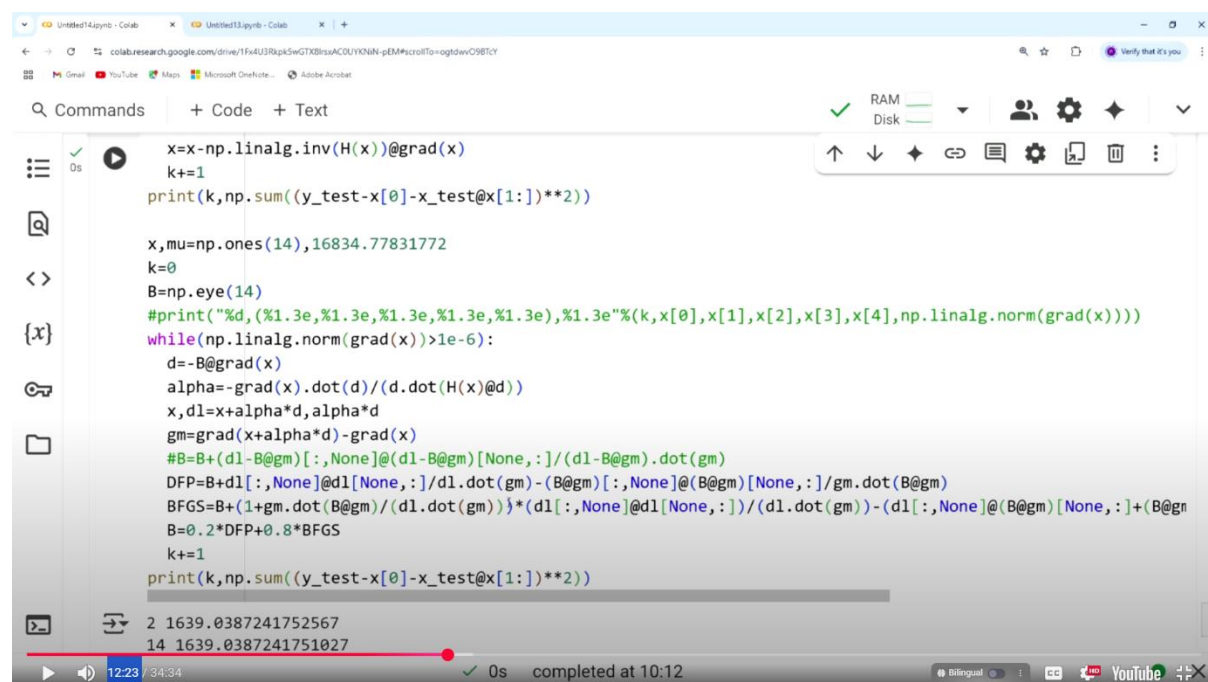
Good. You can see that it just took two steps. If you recall the constrained version which we solved in the last class, we were struggling with a lot of steps. Note that here we just got the answer in two steps. This should not be surprising because the reason is that this is actually a quadratic function. Newton's method solves a quadratic function in a trice, is it not? That is the reason. The problem is large, but since the form of the objective function is just quadratic,

Newton's method solves this quickly, and that is what you have observed here. You can also observe that the answer is exactly the same, 1639.04. That is what you had here as well.

We are just using the algorithms we have learned to solve this ridge regression problem where you get the solution $\hat{\beta}_0, \hat{\beta}_1, \dots, \hat{\beta}_p$. This is using Newton's method. You can also solve using other methods. We will just solve using one other method, which is the BFGS method. For the other ones, I think you can repeat the code and find the solutions yourself. This part will remain the same. We will now move to the BFGS algorithm.

This is a quasi-Newton method. You have the BFGS algorithm here. We will start from this and complete it here. This is not just for BFGS; it is actually for the Broyden family of algorithms, where we have $B = 0.2$ times DFP plus 0.8 times BFGS. You can recall it is ϕ times DFP plus $(1 - \phi)$ times BFGS, where ϕ could take any value between zero and one. Here we have taken 0.2 . Let us see if this needs a change. I do not need to print the values. This needs to be fourteen. This has to be $H(x)$. I think everything else is fine. Let us check how long this takes. I did not print the answer here. Yes, so this has taken fourteen iterations, and you can see it has given the same answer.

(Refer Slide Time 12:23)



```

x=x-np.linalg.inv(H(x))@grad(x)
k+=1
print(k,np.sum((y_test-x[0]-x_test@x[1:])**2))

x,mu=np.ones(14),16834.77831772
k=0
B=np.eye(14)
#print("%d, (%1.3e,%1.3e,%1.3e,%1.3e,%1.3e,%1.3e,%1.3e"%(k,x[0],x[1],x[2],x[3],x[4],np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
    d=-B@grad(x)
    alpha=-grad(x).dot(d)/(d.dot(H(x)@d))
    x,d1=x+alpha*d,alpha*d
    gm=grad(x+alpha*d)-grad(x)
    #B=B+(d1-B@gm)[ :,None]@(d1-B@gm)[None, :]/(d1-B@gm).dot(gm)
    DFP=B+d1[ :,None]@d1[None, :]/d1.dot(gm)-(B@gm)[ :,None]@(B@gm)[None, :]/gm.dot(B@gm)
    BFGS=B+(1+gm.dot(B@gm)/(d1.dot(gm)))*(d1[ :,None]@d1[None, :])/(d1.dot(gm))-(d1[ :,None]@(B@gm)[None, :])+(B@gm)
    B=0.2*DFP+0.8*BFGS
    k+=1
print(k,np.sum((y_test-x[0]-x_test@x[1:])**2))

```

2 1639.0387241752567
14 1639.0387241751027

12:23 / 34:34 0s completed at 10:12

You can try this with the other algorithms as well, like the conjugate gradient algorithm and also the gradient descent algorithm. When you try gradient descent, the solution will take quite a long time to complete, like millions of iterations. The reason is, if you recall what we learned in the fourth or fifth week, if the eigenvalues of the Hessian matrix are not close enough to each other—if the ratio of the highest eigenvalue to the lowest eigenvalue is not close to one—then gradient descent takes a long time. Note that the Hessian here is large. The function itself has four hundred data points. The objective function itself is large, and the Hessian has a wide

range of eigenvalues. The ratio would be far off from one. Please do not expect all the eigenvalues to be close enough. In fact, the highest and lowest eigenvalues will be as far apart as possible.

You can try it at home using gradient descent, and you will see that the number of iterations you need runs into the millions, like two million or three million. That is why we try to avoid gradient descent as much as possible. Gradient descent, as you all know, is the simplest method. It just goes in the direction of the negative gradient. But when the Hessian does not satisfy the desirable properties, it turns out the method is very, very slow. Just note this: a problem where Newton's method finds the answer in two steps and the Broyden family finds the answer in fourteen steps, your gradient descent is taking millions of steps. That is something we try to avoid.

This completes the application. We have taken an application and used many of the algorithms we discussed throughout this course to solve this ridge regression problem. Before completing this application, I want to tell you two things. The first one is, given that this is a machine learning problem, you might be asking if people who are actually implementing ridge regression do they implement all of these algorithms, like Newton's method or the penalty method or augmented Lagrangian method. Of course, if what you want to do is just solve this ridge regression problem, Python has a library to solve all of these problems. It is implemented as a subroutine and given as an inbuilt command.

I would like to let you know about that particular command. It uses a different library. We have only used NumPy, Matplotlib, and Pandas. Pandas was just to input the data. There is a library called scikit-learn. If you have come this far, you can look this up. I will just tell you the library and the command so you get to know how you would do ridge regression. This library is called sklearn. Its full form is scikit-learn. You can look it up. In this, we are going to choose the sub-library called linear_model. I will take this as 'skllm'. You can name it anything you want. This is for scikit-learn linear model.

In this, I should use a command called 'Ridge'. Note it is with a capital R. If you put a small 'r', it will not work. You have to give the μ parameter and not the 's' parameter we used in constrained optimization. The μ parameter, if you recall, is 16834.778 or whatever. You can call this maybe as 'clf' for classifier. For regression, you could call it 'reg' or something. This is the model we are after. What you want to do is to fit 'x_train' and 'y_train'. The feature vectors are given in 'x_train', and the labels are given in 'y_train'. This will be a 506 by thirteen matrix, and 'y_train' is a 506-length vector. The model is fit with 'x_train' and 'y_train'.

After you fit, this command computes the values of $\hat{\beta}_0, \hat{\beta}_1, \dots, \hat{\beta}_p$. If you want to predict for a test vector, you use 'clf.predict'. We will predict for all 'x_test'. You will get a 106-dimensional vector. Rather than that, we are interested in the test error. Find the difference between 'y_test' and 'clf.predict(x_test)', take the square, and then sum it over. We will try to print this. We can straight away see that the answer is 1639.038. In case your objective is just to solve a ridge regression problem, you do not need to go via the optimization way.

In case you are asking why we learned this application, it is because if you are learning more sophisticated machine learning, you will see that you cannot just use the inbuilt commands for all the methods. For example, neural networks. The TensorFlow and PyTorch libraries used to construct neural networks have limitations. If you want to construct a more sophisticated neural network or use more sophisticated loss functions, there are certain loss functions for which the inbuilt commands in Python will not be helpful. You might have to use optimization algorithms to construct the loss function and find an answer. That is where the algorithms we learned will be useful.

Finally, I said optimization algorithms will be useful. The final question we need to answer is which of these algorithms are used in machine learning applications. We have learned quite a lot of methods, starting from gradient descent, conjugate gradient, Newton's method, quasi-Newton method, to the ones in constrained optimization for linear and nonlinear programming problems. The answer is that, unfortunately, none of these are actually used directly for large-scale machine learning. The algorithms used for machine learning applications are actually called stochastic optimization algorithms. These are more advanced than what we have learned until now. What we have learned are, to an extent, basic algorithms.

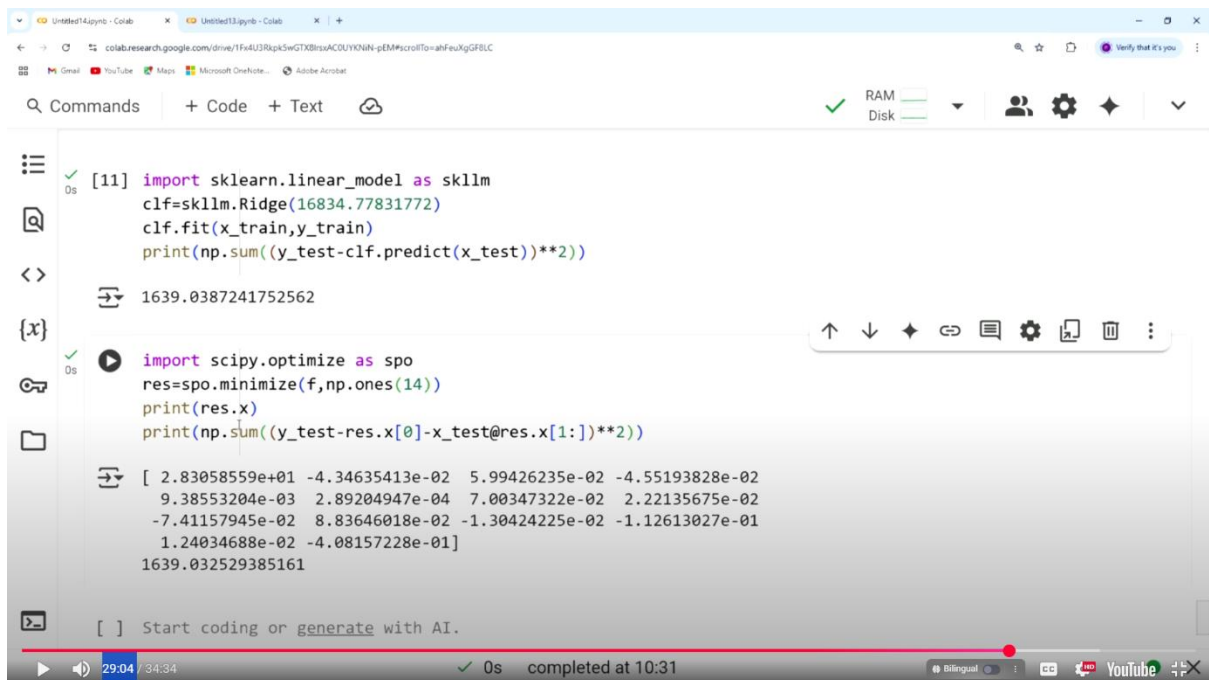
If you are looking for the names of those advanced algorithms used in ML, three major ones are Stochastic Gradient Descent (SGD), L-BFGS, and the Adam optimizer. Stochastic Gradient Descent is an advanced version of gradient descent. L-BFGS is an advanced version of BFGS. The Adam optimizer is a slightly different kind of algorithm. These are the three important optimization algorithms useful in machine learning applications. If you want to learn them, you will have to possibly take a different course. I might offer that sometime later. I would like to invite you all who have completed this course to that particular course as well.

I would like to give you a final note about Python codes for optimization. What we have learned is a step-by-step procedure. When you have an optimization problem to solve, I have given some routines. You just put them in and solve it. As I said for ridge regression, we solved it using an optimization algorithm from scratch. There is a command which implements all of this and gives you the answer. In a similar way, there are commands in Python which solve an optimization problem. I will just give you those commands and solve one constrained and one unconstrained problem. We will end the course with that.

Suppose you just want to solve an optimization problem using Python, you can use these commands. I want to let you know about that. In case you want to learn more, you can read about it. This is from another library called `'scipy.optimize'`. SciPy is the basic library, and optimize is a sub-library. I am defining this as `'spo'`. You can define it whatever you want. Suppose you want to minimize our function f . We have defined f anyway. You can define an $f(x)$ and solve this unconstrained problem.

What you need to do is define the result as `'scipy.optimize.minimize(f, the_initial_point)'`. The initial point we know is all ones, fourteen ones. We will print the value of x , which is the beta value. We have not printed this anywhere until now, but that is fine. We are more interested in the test error. We will again take this, but instead of x , we need to write `'res.x'`. You can see it is again the same, 1639.03. It is as simple as that to minimize an unconstrained function.

(Refer Slide Time 29:05)



```
[11] import sklearn.linear_model as sklmm
      clf=sklmm.Ridge(16834.77831772)
      clf.fit(x_train,y_train)
      print(np.sum((y_test-clf.predict(x_test))**2))

1639.0387241752562

import scipy.optimize as spo
res=spo.minimize(f,np.ones(14))
print(res.x)
print(np.sum((y_test-res.x[0]-x_test@res.x[1:])**2))

[ 2.83058559e+01 -4.34635413e-02  5.99426235e-02 -4.55193828e-02
 9.38553204e-03  2.89204947e-04  7.00347322e-02  2.22135675e-02
-7.41157945e-02  8.83646018e-02 -1.30424225e-02 -1.12613027e-01
 1.24034688e-02 -4.08157228e-01]
1639.032529385161
```

If you want to do a constrained optimization, you also have to define a constraint. I will just do that quickly and end the course. Let us take a different problem, the consumer utility maximization problem we defined in the beginning. We will maximize x_1x_2 , which is equivalent to

minimizing $-x_1x_2$.

The constraint is $p_1x_1 + p_2x_2 \leq w$.

As a simple example,

we will say $x_1 + x_2 \leq 1$ and $x_1 \geq 0, x_2 \geq 0$.

The answer is actually half and half. Since I just want to give you an example for constrained optimization, I am doing this.

(Refer Slide Time 30:20)

Other advanced algorithms:

1. Stochastic Gradient Descent (SGD)
2. L-BFGS
3. Adam's optimizer.

$$\min_{x_1, x_2} (-x_1, x_2)$$

$$\text{s.t. } x_1 + x_2 \leq 1, x_1 \geq 0, x_2 \geq 0.$$

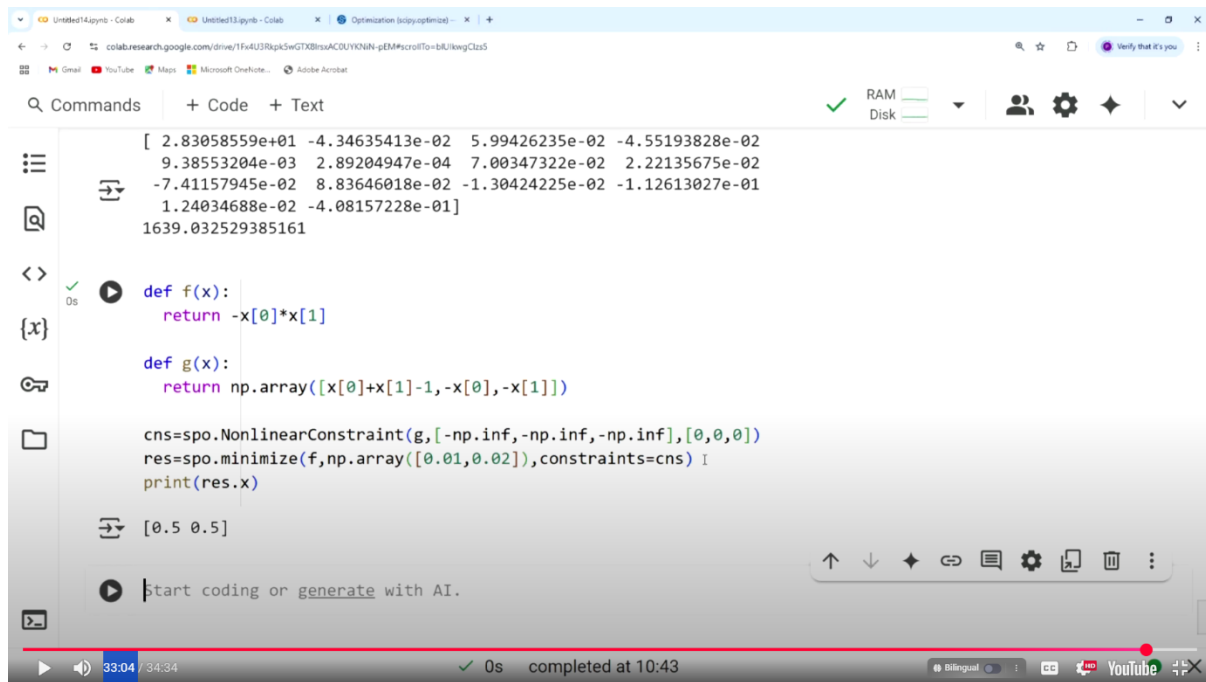
We will define $f(x)$ to be $-x[0]*x[1]$.

We define $g(x)$ to have all the constraints: $x[0] + x[1] - 1$, $-x[0]$, and $-x[1]$.

We will write this as an `np.array`. You have to define the bounds as well. This is something new. I am defining a nonlinear constraint with `spo.NonlinearConstraint`. For g , note that it is all less than or equal to zero. The lower limit is $-\infty$, and the upper limit is zero. You need this. We can represent infinity as a large negative number for the lower bound for the inequality, but the library handles it. The upper bound is zeros for three constraints.

You will just have this. The result is `spo.minimize(f, initial_point, constraints=constraints)`. We need an initial point. We will just put it as zeros, `np.zeros(2)`. You need to say the constraints are in place. You just say `constraints=cons`. We will print `res.x` as the answer. I have to mention this as constraints. There was a small error in my variable name. Using zeros might be a problem because of the objective function. We will use a different initial point, maybe 0.1 and 0.2. You can see the answer is 1/2 comma 1/2. You can use something else as well, like 0.01 and 0.02, and you will see the answer is half comma half as well.

(Refer Slide Time 33:04)



```
[ 2.83058559e+01 -4.34635413e-02  5.99426235e-02 -4.55193828e-02
 9.38553204e-03  2.89204947e-04  7.00347322e-02  2.22135675e-02
 -7.41157945e-02  8.83646018e-02 -1.30424225e-02 -1.12613027e-01
 1.24034688e-02 -4.08157228e-01]
1639.032529385161
```

```
def f(x):
    return -x[0]*x[1]

def g(x):
    return np.array([x[0]+x[1]-1, -x[0], -x[1]])

cns=spo.NonlinearConstraint(g,[-np.inf,-np.inf,-np.inf],[0,0,0])
res=spo.minimize(f,np.array([0.01,0.02]),constraints=cns)
print(res.x)
```

```
[0.5 0.5]
```

This is an inbuilt function I wanted to discuss before finishing the course. If you are asking how the computer solves this, it uses the algorithms we discussed throughout this course. For those who would like to do research on this, I would welcome you to actually improve these algorithms. There are certain algorithms that have been implemented. You can use these algorithms or, in fact, design more efficient algorithms which can give answers even faster.

With this, we would like to end the course. I would like to thank all the students who have come this far. I hope the world of optimization algorithms was very interesting. Let us meet again in some other course which I might offer sometime later. Thank you all.