Rishith Kyatham
CSE353
Professor : Yifan Sun

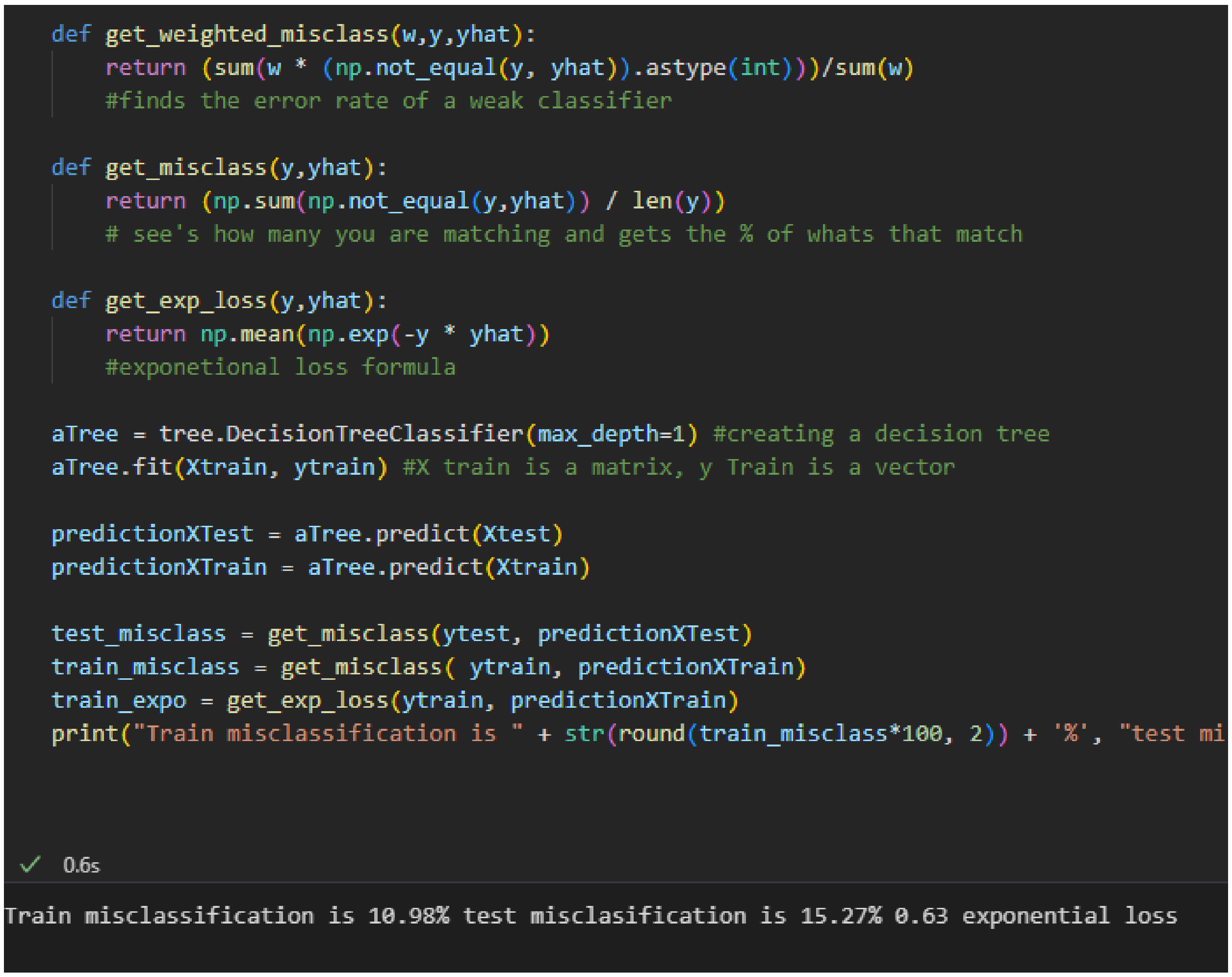
HW 4

1. a) When splitting a decision tree, we split by the feature of where the data that gives the most information gained. When you look at the path you took, we have all the decision nodes. We could even print the path of all the nodes to any given node we want to. In the path of the nodes, we would go through, each node is focused on a single feature of where it split. So, if we were to sort the decision nodes by the information gain, we could see that the ones with the higher information gain would have a higher contribution of the feature and would be more important than the ones that had a lower information gain.

b) How a linear regression model works is that it predicts lines using an estimated line. The slope value (theta), shows how each slope at given points would impact the end label. Now, this could be easily calculated at a given point by just taking the slope respective to the point. Therefore, each feature is important when determining the contributed answer at the end.

c) A graphical model uses the inputs given and follows a probability pathway. Each of the nodes of the graphical model corresponds to a feature so, from any label, we can trace path to the final node. This is similar to the decision tree as you trace your path in the same manner and you could find all the nodes (given features) and their respective probabilities to see how it came to to the final contribution. A higher probability indicates that the respective node has a bigger importance in classifying the label.

2. a) **Deep Trees :**



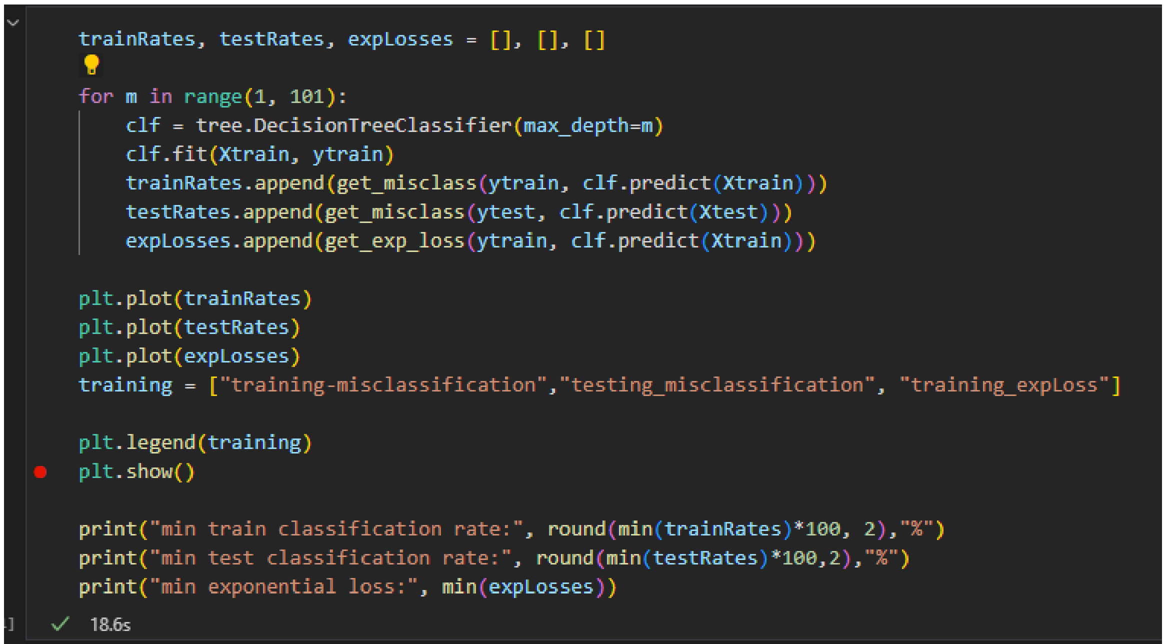
```
def get_weighted_misclass(w,y,yhat):
    return (sum(w * (np.not_equal(y, yhat)).astype(int)))/sum(w)
    #finds the error rate of a weak classifier

def get_misclass(y,yhat):
    return (np.sum(np.not_equal(y,yhat)) / len(y))
    # see's how many you are matching and gets the % of whats that match

def get_exp_loss(y,yhat):
    return np.mean(np.exp(-y * yhat))
    #exponetional loss formula

aTree = tree.DecisionTreeClassifier(max_depth=1) #creating a decision tree
aTree.fit(Xtrain, ytrain) #X train is a matrix, y Train is a vector

predictionXTest = aTree.predict(Xtest)
predictionXTrain = aTree.predict(Xtrain)

test_misclass = get_misclass(ytest, predictionXTest)
train_misclass = get_misclass( ytrain, predictionXTrain)
train_expo = get_exp_loss(ytrain, predictionXTrain)
print("Train misclassification is " + str(round(train_misclass*100, 2)) + '%', "test mi
```

✓ 0.6s

```
Train misclassification is 10.98% test misclasification is 15.27% 0.63 exponential loss
```

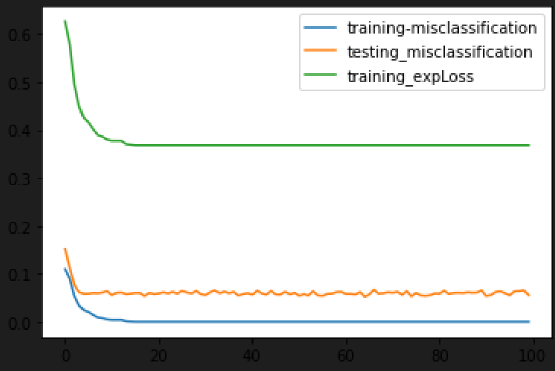
On the bottom, I got the values using a Deep Tree.

Part of Deep Trees :



```python
trainRates, testRates, expLosses = [], [], []

for m in range(1, 101):
    clf = tree.DecisionTreeClassifier(max_depth=m)
    clf.fit(Xtrain, ytrain)
    trainRates.append(get_misclass(ytrain, clf.predict(Xtrain)))
    testRates.append(get_misclass(ytest, clf.predict(Xtest)))
    expLosses.append(get_exp_loss(ytrain, clf.predict(Xtrain)))

plt.plot(trainRates)
plt.plot(testRates)
plt.plot(expLosses)
training = ["training-misclassification","testing_misclassification", "training_expLoss"]

plt.legend(training)
plt.show()

print("min train classification rate:", round(min(trainRates)*100, 2),"%")
print("min test classification rate:", round(min(testRates)*100,2),"%")
print("min exponential loss:", min(expLosses))
```

✓ 18.6s



```
min train classification rate: 0.0 %
min test classification rate: 5.27 %
min exponential loss: 0.3678794411714424
```

Boosted Decision Stumps:

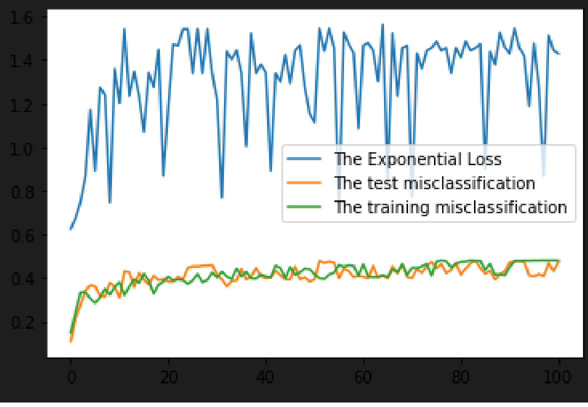
(Code is submitted separately)

Final Results For Adaboost:

```python
plt.plot(theExpLoss, label = "The Exponential Loss")
plt.plot(trainingMisclassifications, label = "The test misclassification")
plt.plot(testingMisclassifications, label = "The training misclassification")
plt.legend()

print("minimum Training misclassification", min(trainingMisclassifications))
print("minimum Testing misclassification", min(testingMisclassifications))
print("minimum Exponential Loss", min(theExpLoss))
#question : the boosting decision tree did not perform as well as the deep trees because
#deep creates create large trees up to a depth of 100 and the boosting we were doing did not
#make up the difference in accuracy (Deep trees are more accurate) Exponential loss is lower
```

✓ 0.1s

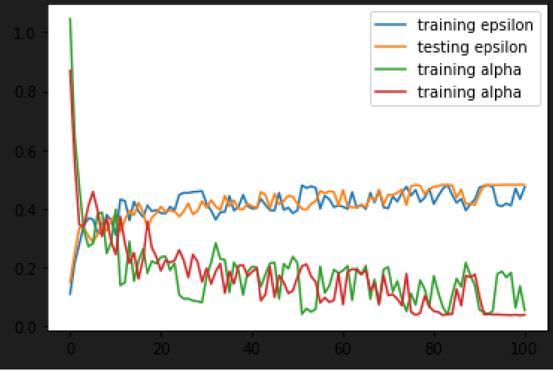
```
minimum Training misclassification 0.10980592441266993
minimum Testing misclassification 0.1496735308889984
minimum Exponential Loss 0.6259675480492947
```



```python
plt.plot(trainingMisclassifications, label="training epsilon")
plt.plot(testingMisclassifications, label="testing epsilon")
plt.plot(alphaWeightTrains, label="training alpha")
plt.plot(alphaWeightTests, label="training alpha")
plt.legend()
#Question :
```

✓ 0.1s

```
<matplotlib.legend.Legend at 0x1e584acf130>
```

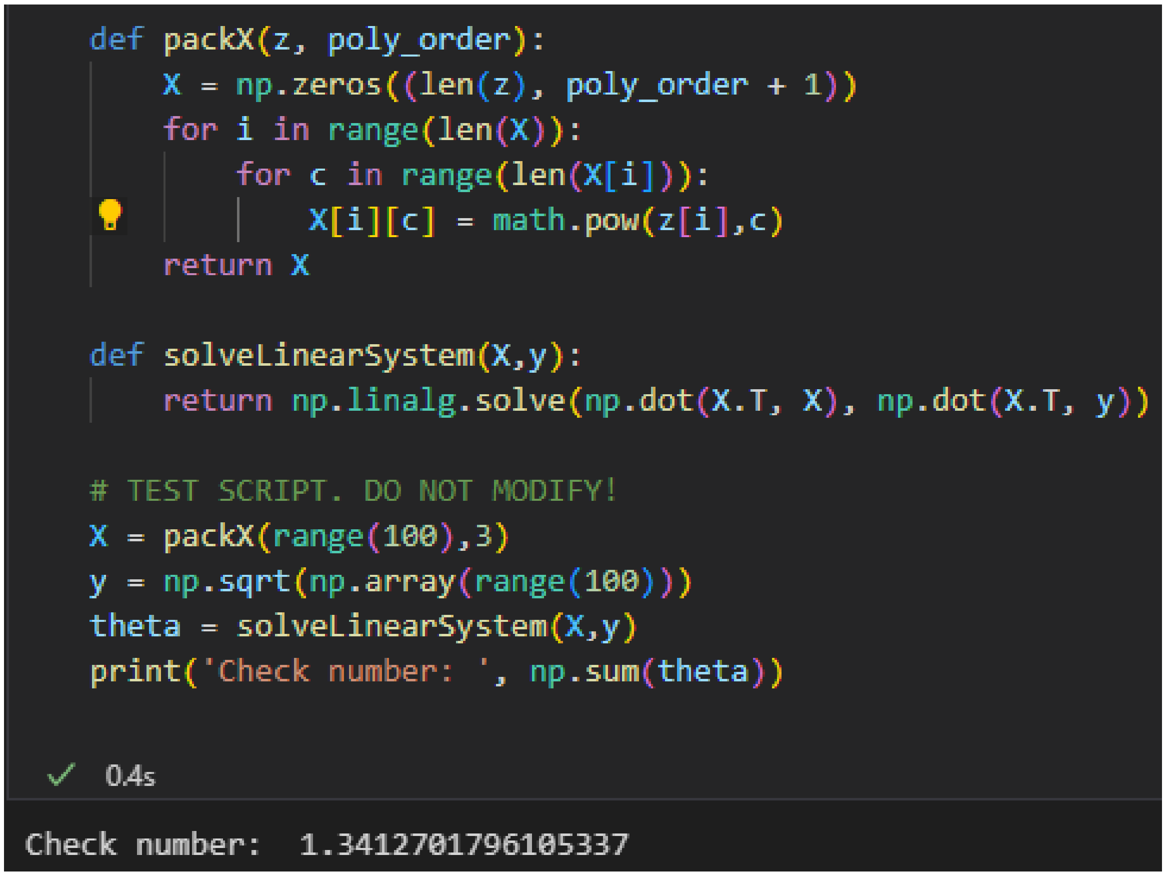


The minimum test misclassification rate, train misclassification rate, and smallest train exponential losses are stated in the pictures above.

First Part of Question: The boosted decision stumps performed worse than deep trees because they have a much higher minimum exponential Loss. The deep tree training misclassification seems to approach 0 as the iterations increase but this same action does not occur in the boosted decision stumps.

Second Part Of Question: As the number of iterations increases, the epsilon gets bigger. For the higher values, epsilon is really large and positive. As the number of iterations increases, alpha goes down and is really negative at the higher iteration values. The alpha values go closer and get closer to 0 as the number of iterations increases. I believe the functionality exists as this because our main objective is to increase the weights of misclassified weights and to adjust the other weights accordingly. This would lead to the epsilon values getting bigger and the alpha values getting smaller. So, in conclusion, the misclassified values get a higher weight so they get priority of identification in the next iteration.

3. **Part A** :
i) The normal equation is **(X^T)Xθ = (X^T)Y**
ii)

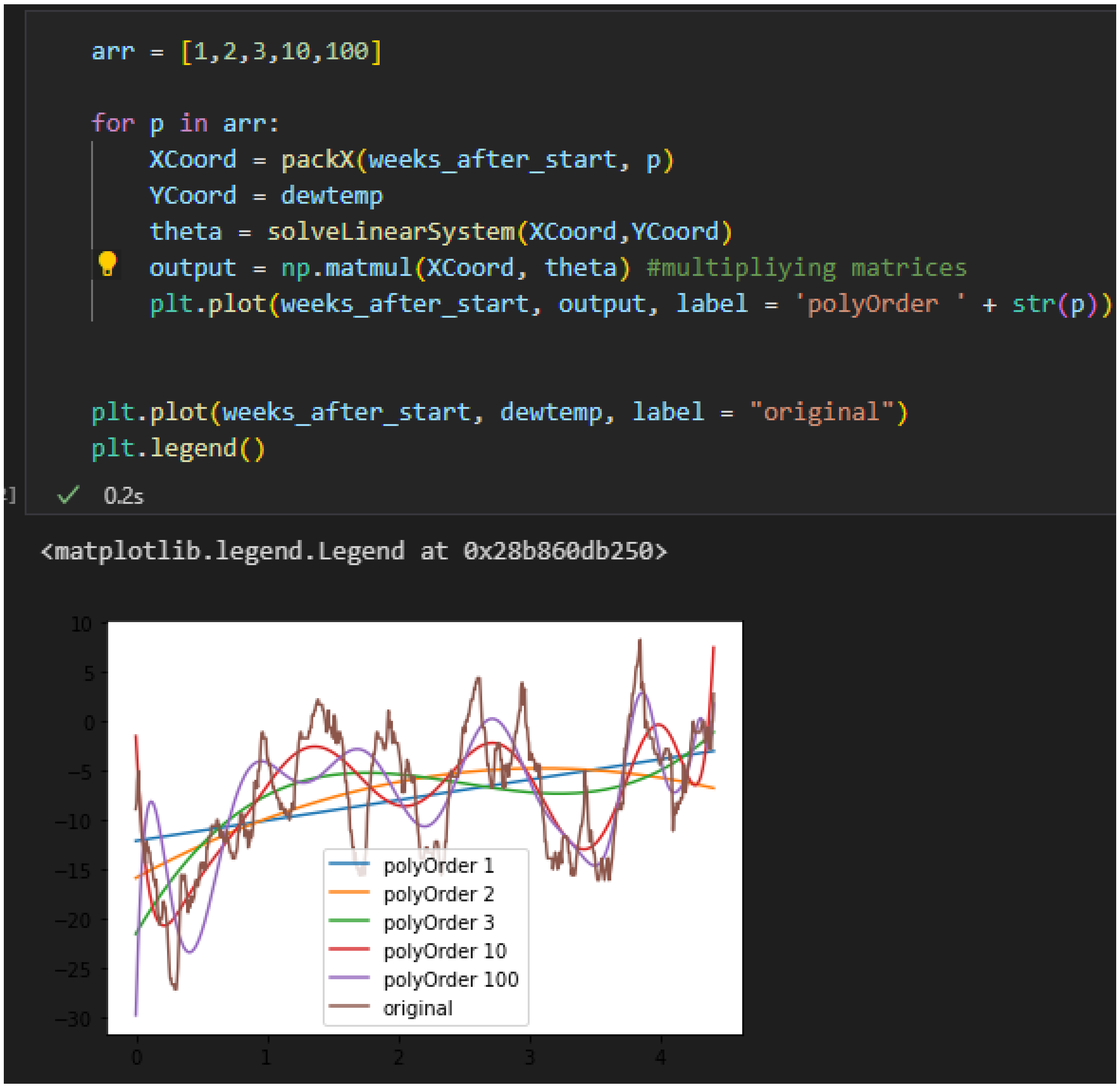


```
def packX(z, poly_order):
    X = np.zeros((len(z), poly_order + 1))
    for i in range(len(X)):
        for c in range(len(X[i])):
            X[i][c] = math.pow(z[i],c)
    return X

def solveLinearSystem(X,y):
    return np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))

# TEST SCRIPT. DO NOT MODIFY!
X = packX(range(100),3)
y = np.sqrt(np.array(range(100)))
theta = solveLinearSystem(X,y)
print('Check number: ', np.sum(theta))
```

✓ 0.4s

```
Check number:  1.3412701796105337
```

iii)



```
arr = [1,2,3,10,100]

for p in arr:
    XCoord = packX(weeks_after_start, p)
    YCoord = dewtemp
    theta = solveLinearSystem(XCoord,YCoord)
    output = np.matmul(XCoord, theta) #multipliying matrices
    plt.plot(weeks_after_start, output, label = 'polyOrder ' + str(p))


plt.plot(weeks_after_start, dewtemp, label = "original")
plt.legend()
```

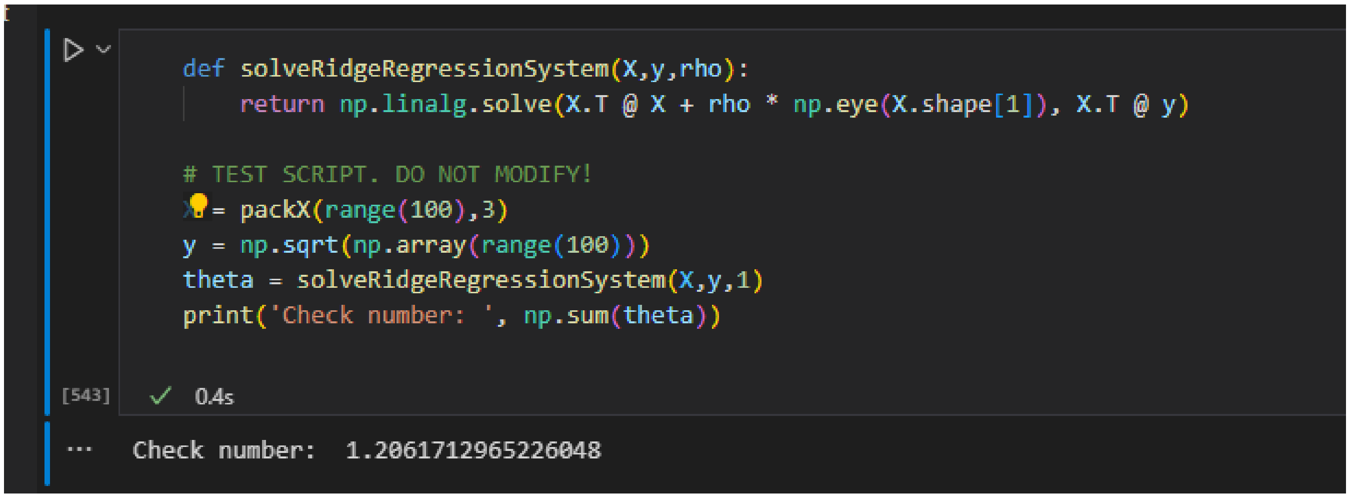
```
✓ 0.2s
```

```
<matplotlib.legend.Legend at 0x28b860db250>
```

For each polyOrder, it seems to be following the trend nicely. I believe that linear regression would be a very good fit to the graph. The lines on the graph follow similarly to the actual data there.

Part B :

i) The normal equation is **((X^T)X + ρI)θ = (X^T)Y** ii)

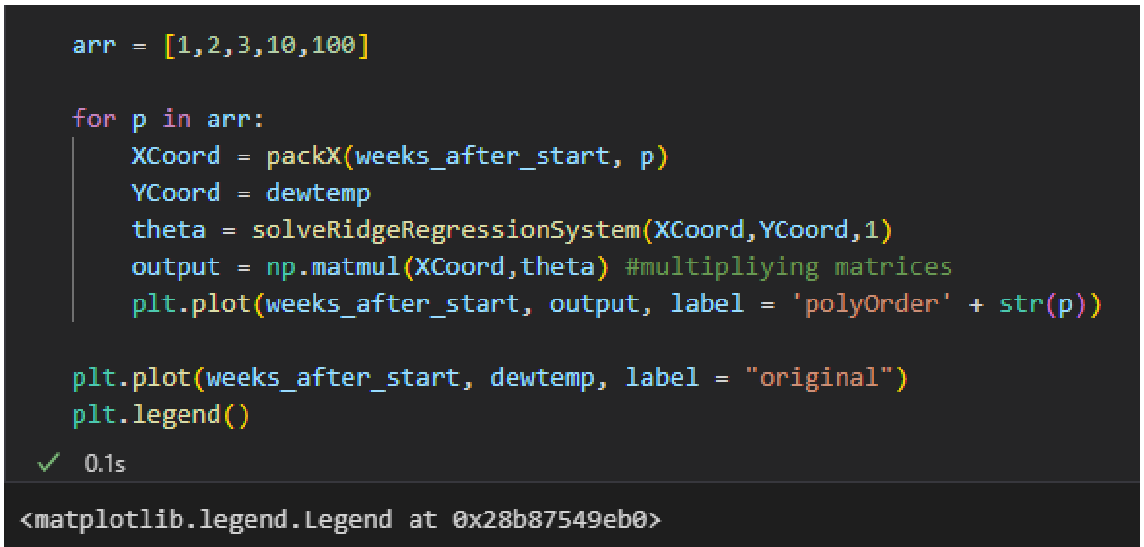


```
def solveRidgeRegressionSystem(X,y,rho):
    return np.linalg.solve(X.T @ X + rho * np.eye(X.shape[1]), X.T @ y)

# TEST SCRIPT. DO NOT MODIFY!
X = packX(range(100),3)
y = np.sqrt(np.array(range(100)))
theta = solveRidgeRegressionSystem(X,y,1)
print('Check number: ', np.sum(theta))
```

```
[543]  ✓ 0.4s
```

```
Check number:  1.2061712965226048
```

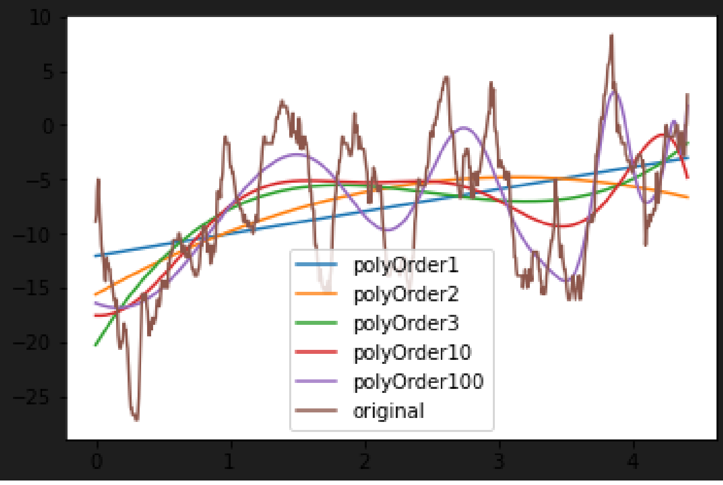
iii)



```
arr = [1,2,3,10,100]

for p in arr:
    XCoord = packX(weeks_after_start, p)
    YCoord = dewtemp
    theta = solveRidgeRegressionSystem(XCoord,YCoord,1)
    output = np.matmul(XCoord,theta) #multipliying matrices
    plt.plot(weeks_after_start, output, label = 'polyOrder' + str(p))

plt.plot(weeks_after_start, dewtemp, label = "original")
plt.legend()
```

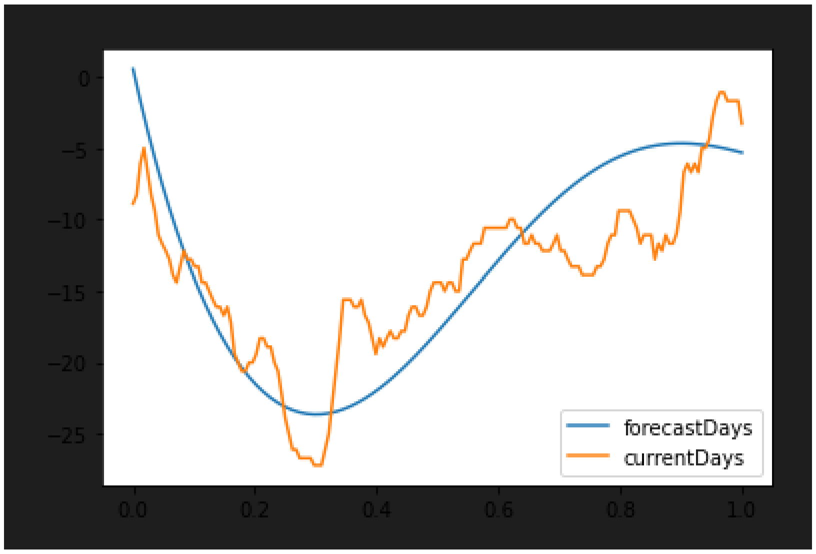
```
✓ 0.1s
```

```
<matplotlib.legend.Legend at 0x28b87549eb0>
```



This has a more lenient fit whereas the previous part had a more strict fit. With a more lenient fit, the lines curve and follow the original line's data much better. I believe this graph is a very good fit and is better than the linear regression fit.

Part D :



When I picked **pVal = 50** and **rho = 0.0005,** I got this graph. I believe the forecast as the trend is followed pretty closely. A couple of outliers are fine because I tested multiple other values and as pVal increases, it seems to get closer to the actual trend. It also does not have a lot of variance as well.