Lab session #6
Linear Models for Regression and Classification

Dr Zied M'NASRI

Introduction

This lab session aims to implement the linear models for regression and classification, as seen in Lecture 6. To do the exercises of this lab session, you need to use the following:

1. Python tutorial (see Lab 1 & Lab 2 material)
2. Python libraries tutorial (see Lab 3 material)
3. Lecture 6 notes
4. Google's Colab or Jupyter Notebook (on Horizon)
5. The attached csv files *Salary_data.csv* and *play_tennis.csv*

Part I Linear regression

Tutorial 1 - Linear regression

In this demo, we want to reproduce the example seen in Lecture 6, to predict the model that allows the output *y*=[1,3,3] match the input vector *x*=[1,2,4].

Steps

1. Set-up: Import the python library *numpy*

```
import numpy as np
```

2. Training data: Create a training_Data vector made of *(xi,yi)* values

```
training_data = [
    (1, 1),
    (2, 3),
    (4, 3)
]
```

3. Feature extractor: Define a function *phi(x)* that returns the vector [1,*x*]

```
def phi(x):
    return np.array([1, x])
```

4. Loss function: Define a function `trainLoss(w)` that computes the training loss as:

$$TrainLoss(w) = \frac{1}{N_{train}} \sum_{i=1}^{N} (w.\varphi(x) - y)^2$$

Hint: Use the **np.dot** command to calculate the dot product

```
def trainLoss(w):
```

```
    return sum((np.dot(w, phi(x)) - y)**2 for x, y in
    training_data)/len(training_data)
```

5. Define the function `gradientTrainLoss(w)` that computes the gradient of the train loss as:

$$\nabla_w TrainLoss(w) = \frac{1}{N_{train}} \sum_{i=1}^{N} 2(w.\varphi(x) - y) \quad \varphi(x)$$

```
def gradientTrainLoss(w):
```

```
    return sum(2*(np.dot(w, phi(x)) - y)*phi(x) for x, y
    in  training_data)/len(training_data)
```

6. Define the function `gradientDescent(F,gradF,w0)` that updates the gradient descent along 500 epochs (iterations)

Hints:
- Use the method `.copy()` to initialise *w* with the values of *w0*
- Set the learning rate *eta* to 0.1
- For standard notations, call the iteration counter as `epoch`
- At the end of each iteration, print the epoch number, the value of *w*, the Train Loss (*F*) and the gradient of the Train Loss (*gradF*)

```
def gradientDescent(F, gradF, w0):

    w = w0.copy()

    eta = 0.1

    for epoch in range(500):

        gradient = gradF(w)

        w -= eta*gradient
```

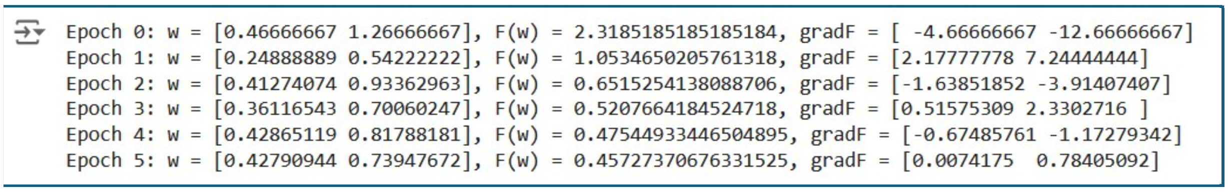
```
    print(f'Epoch {epoch}: w = {w}, F(w) = {F(w)},  gradF
    = {gradient}')
```

```
    return(w)
```

7. Call the function `gradientDescent (trainLoss, gradientTrainLoss, w0)` with *w0*=[0,0]. If the code is right, you should visualise the update of *w* until the end of eopchs.

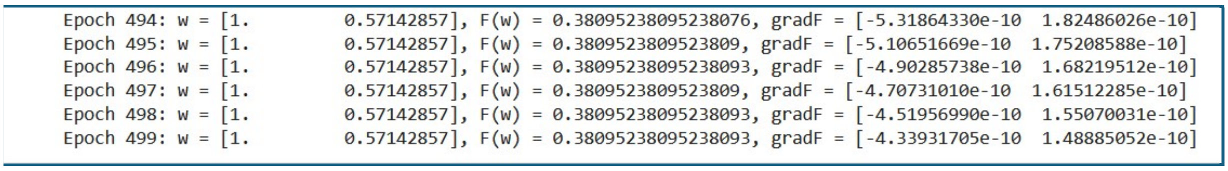
```
gradientDescent(trainLoss, gradientTrainLoss, np.zeros(2))
```

Training Result



```
Epoch 0: w = [0.46666667 1.26666667], F(w) = 2.3185185185185184, gradF = [ -4.66666667 -12.66666667]
Epoch 1: w = [0.24888889 0.54222222], F(w) = 1.0534650205761318, gradF = [2.17777778 7.24444444]
Epoch 2: w = [0.41274074 0.93362963], F(w) = 0.6515254138088706, gradF = [-1.63851852 -3.91407407]
Epoch 3: w = [0.36116543 0.70060247], F(w) = 0.5207664184524718, gradF = [0.51575309 2.3302716 ]
Epoch 4: w = [0.42865119 0.81788181], F(w) = 0.47544933446504895, gradF = [-0.67485761 -1.17279342]
Epoch 5: w = [0.42790944 0.73947672], F(w) = 0.45727370676331525, gradF = [0.0074175  0.78405092]
```

...

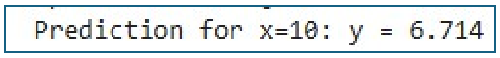


```
Epoch 494: w = [1.         0.57142857], F(w) = 0.38095238095238076, gradF = [-5.31864330e-10  1.82486026e-10]
Epoch 495: w = [1.         0.57142857], F(w) = 0.3809523809523809, gradF = [-5.10651669e-10  1.75208588e-10]
Epoch 496: w = [1.         0.57142857], F(w) = 0.38095238095238093, gradF = [-4.90285738e-10  1.68219512e-10]
Epoch 497: w = [1.         0.57142857], F(w) = 0.3809523809523809, gradF = [-4.70731010e-10  1.61512285e-10]
Epoch 498: w = [1.         0.57142857], F(w) = 0.38095238095238093, gradF = [-4.51956990e-10  1.55070031e-10]
Epoch 499: w = [1.         0.57142857], F(w) = 0.38095238095238093, gradF = [-4.33931705e-10  1.48885052e-10]
```

Once training is finished, predict y for any x given as input, using the dot product of the weight vector *w* (result of the function `gradientDescent)` and the feature extractor *phi(x)*. Print the result with only 3 decimals, e.g.

```
x=int(input())
y = np.dot(gradientDescent(trainLoss, gradientTrainLoss,
np.zeros(2)), phi(x))
print(f'Prediction for x={x}: {y = :.3f}')
```

Sample result



```
Prediction for x=10: y = 6.714
```

Exercise I – Univariate linear model for regression

In this exercise, we aim at implanting a linear regression model that estimates the salary from the experience duration. Therefore, we will use the csv file Salary_data.csv containing two columns: YearsExperience (as input), as Salary (as output).

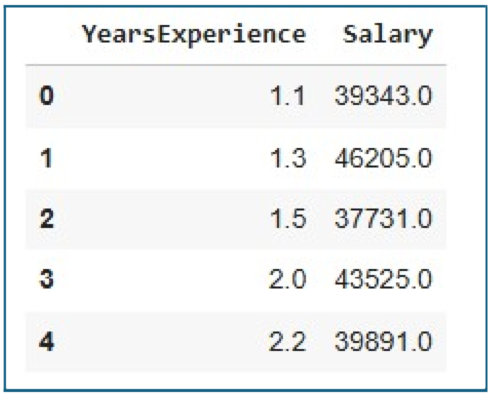
Steps:

1. Upload the *.csv* file in Colab using the following script

```
from google.colab import files
uploaded = files.upload()
```

2. Data preparation
   2.1. Import `numpy`, `pandas` and `matplotlib.pyplot`, as `np`, `pd` and `plt`, respectively.
   2.2. Read the attached csv file into a dataframe using the pandas method `pd.read_csv('filename.csv')`
   2.3. Display the first 5 records using the method `dataFrameName.head()`

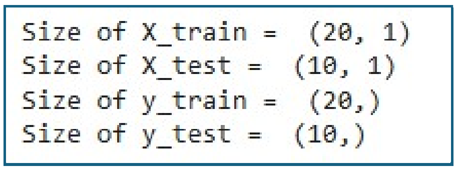
   Sample result



	YearsExperience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0

3. Read *x* and *y* from the dataset, using `dataFrameName.iloc[].values()` method (integer-location based indexing for selection by position)
   Hint: The index of *x* in the dataframe is `[:-1]` and for *y*, it is `[:1]`

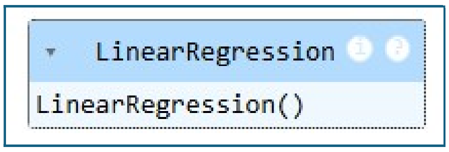
4. From `sklearn.model_selection` import the method `train_test_split`, to split the vectors *x* and *y* into `X_train, y_train, X_test` and `y_test`, respectively.
   Hint: Set `test_size = 1/3` and `random_state=0`
   Print the size of each subset (X_train, y_train, X_trest, y_test) using the attribute `subsetName.shape`
   Sample result



```
Size of X_train =  (20, 1)
Size of X_test =  (10, 1)
Size of y_train =  (20,)
Size of y_test =  (10,)
```

5. From `sklearn.linear_model` import the method `LinearRegression()` and define a regression model called `regressorName` using the imported function. Train the regressor with `x_train` and `y_train` using the method `regressorName.fit()`

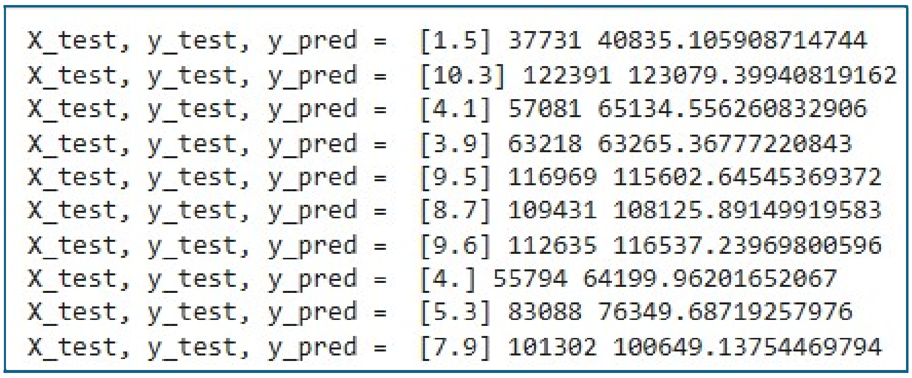
   Sample result



```
▾ LinearRegression
LinearRegression()
```

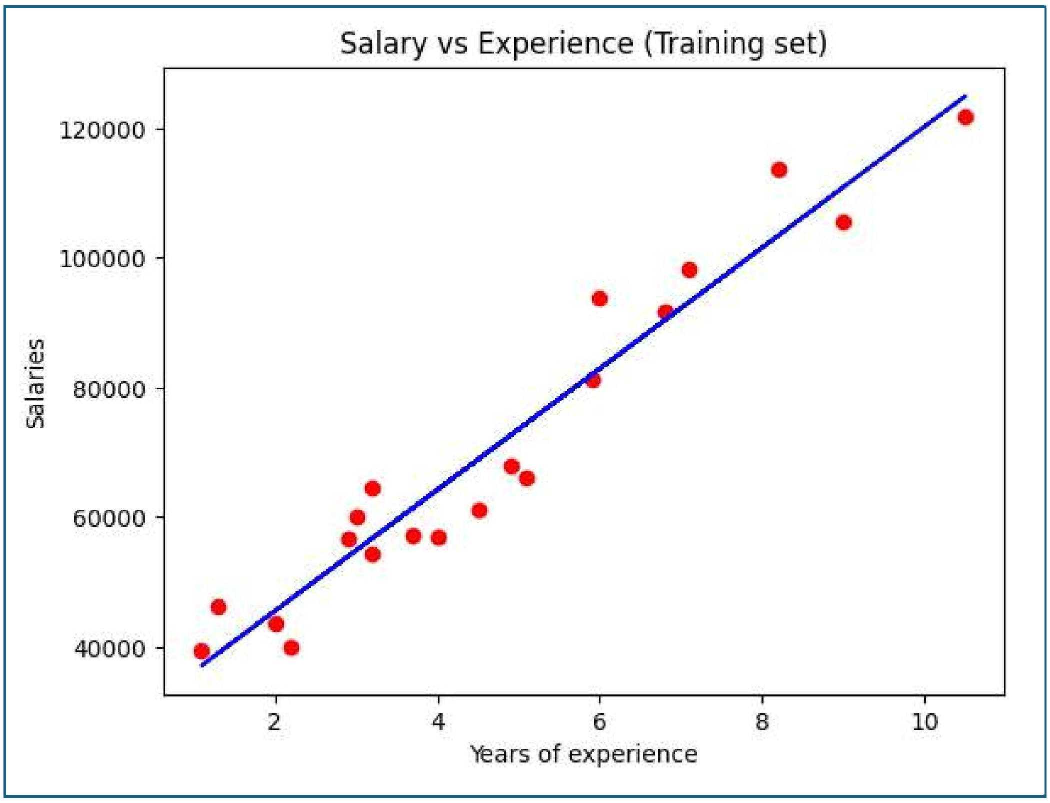
6. Once training is finished, predict all the values of *y_pred* from *x_test* subset using the regressor's method `regressorName.predict() and print the values of X_test, y_test and y_pred`

   Sample result



```
X_test, y_test, y_pred =  [1.5] 37731 40835.105908714744
X_test, y_test, y_pred =  [10.3] 122391 123079.39940819162
X_test, y_test, y_pred =  [4.1] 57081 65134.556260832906
X_test, y_test, y_pred =  [3.9] 63218 63265.36777220843
X_test, y_test, y_pred =  [9.5] 116969 115602.64545369372
X_test, y_test, y_pred =  [8.7] 109431 108125.89149919583
X_test, y_test, y_pred =  [9.6] 112635 116537.23969800596
X_test, y_test, y_pred =  [4.] 55794 64199.96201652067
X_test, y_test, y_pred =  [5.3] 83088 76349.68719257976
X_test, y_test, y_pred =  [7.9] 101302 100649.13754469794
```

7. To view the results, plot the original test data *(x_test,y_test)* as scattered red points, using the method `plt.scatter()`, and the prediction results *(x_test,y_pred)* as a blue line, using the method `plt.plot()`
   Sample result



8. To evaluate the regression model's error, import the function `mean_squared_error` from `sklearn` and apply it to *(y_test,y_pred)*. Then calculate the RMSE (Root Mean Square Error) using the *numpy* method `np.sqrt()`. Print the result.

9. To understand the actual value of the regression error, compute the relative RMSE error, by dividing it by the average value of *y_test*, given by `np.mean()`. Print the relative RMSE.

   Sample results

```
Root Mean Square Error (RMSE) on train data: 6070.662959214961
Relative Root Mean Square Error (RRMSE) on train data: 8.5475%
```

10. Go back to (4.) and repeat the training using a different `test_size` (< ½). Draw your conclusion regarding the effect of train/test proportion on the model's performance.

Part II Linear classification

Tutorial 2 - Binary linear classification with Hinge Loss

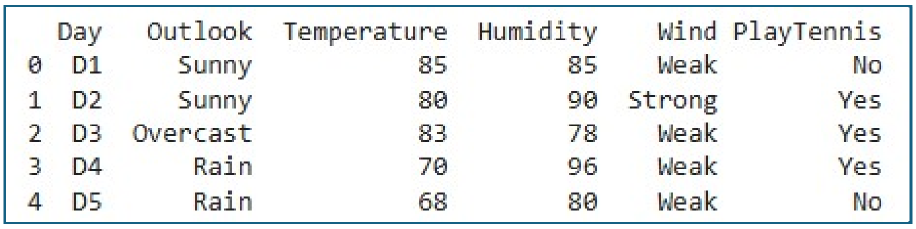
In this tutorial, we aim to implement the linear classification model, as seen in lecture 6. We use the dataset presented in the file “Play_Tennis.csv” taking 2 attributes, Temperature and Humidity, to predict the outcome (Yes/No).

1. Import the file “playTennis.csv” dataset, using the same procedure used in Lab session 5, exercise 1.

```
from google.colab import files
uploaded = files.upload()
```

2. Import the dataset using the pandas method `read_csv()` and display the first 5 elements using the method `datasetName.head()`

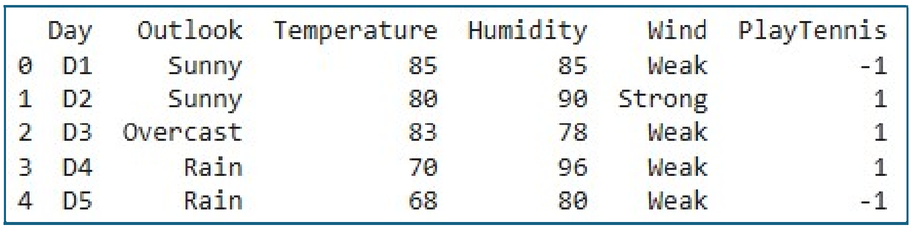
```
dataset = pd.read_csv('play_tennis.csv')
print(dataset.head())
```



```
  Day   Outlook  Temperature  Humidity    Wind PlayTennis
0  D1     Sunny           85        85    Weak         No
1  D2     Sunny           80        90  Strong        Yes
2  D3  Overcast           83        78    Weak        Yes
3  D4      Rain           70        96    Weak        Yes
4  D5      Rain           68        80    Weak         No
```

3. Replace the values of Play Tennis (No, Yes) by (-1,1), respectively, using the method `datasetName(ColumnName).replace()` and print the header again.

```
dataset['PlayTennis'].replace(['No', 'Yes'], [-1,1], inplace =
True)
print(dataset.head())
```



```
  Day   Outlook  Temperature  Humidity    Wind  PlayTennis
0  D1     Sunny           85        85    Weak          -1
1  D2     Sunny           80        90  Strong           1
2  D3  Overcast           83        78    Weak           1
3  D4      Rain           70        96    Weak           1
4  D5      Rain           68        80    Weak          -1
```

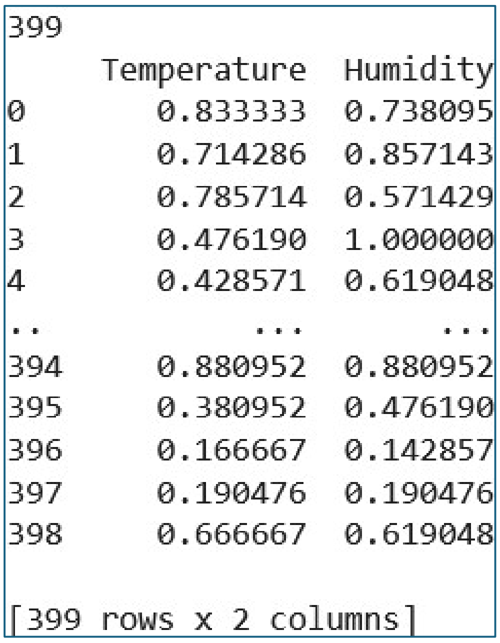
4. Define the input data as X using only 2 attributes(use `datasetName.iloc[-2:2]`), and y using `datasetName.iloc[:,5]`

```
X = dataset.iloc[:,2:-2]
print(len(X))
y = dataset.iloc[:,5]
print(y.head())
```

5. Normalize the values of X such as each column receives the following values:
X(column)=X(column_j)-min(column_j)/(max(column_j)-min(column_j)
Hint! For each column j of X, max and min are defined as
`maxj = max(X.iloc[:,j])` and `minj=min(X.iloc[:,j])`

```
for j in range(2):
    maxj = max(X.iloc[:,j])
    minj = min(X.iloc[:,j])
    X.iloc[:,j] = (X.iloc[:,j]-minj)/(maxj-minj)
print(X)
```

So that the normalized values of Temperature and humidity should be as follows:

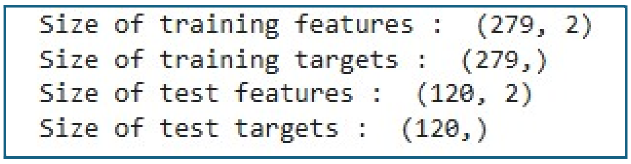


```
399
     Temperature  Humidity
0       0.833333  0.738095
1       0.714286  0.857143
2       0.785714  0.571429
3       0.476190  1.000000
4       0.428571  0.619048
..           ...       ...
394     0.880952  0.880952
395     0.380952  0.476190
396     0.166667  0.142857
397     0.190476  0.190476
398     0.666667  0.619048

[399 rows x 2 columns]
```

Figure 2.1.1. Normalized values of the input features

6. Split *X* and *y* into *X_train*, *y_train*, *X_test* and *y_test* using a split ratio 0.3 and `random_state= 42` using the method `train_test_split()`, and display the size of each subset
   Hint! You need to import `train_test_split()` from `sklearn.model_selection`

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
print("Size of training features : ", X_train.shape)
print("Size of training targets : ", y_train.shape)
print("Size of test features : ", X_test.shape)
print("Size of test targets : ", y_test.shape)
```



```
Size of training features :  (279, 2)
Size of training targets :  (279,)
Size of test features :  (120, 2)
Size of test targets :  (120,)
```

7. Define the input features function (`phi(x)`), as follows:

```
def phi(x):
    return np.array(x)
```

8. Define the boundary decision equation function

```
def boundary(w,x):
     y = -w[0]/w[1]*x
     return y
```

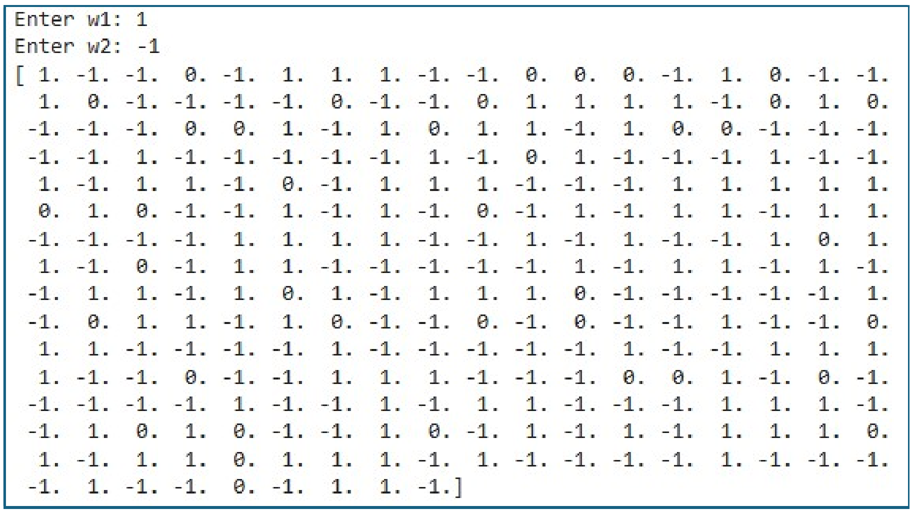
9. Define the prediction function based on an input assumed `weight w=(w1,w2)` using the sign function

```
def predict(w, x):
   return np.sign(np.dot(w, phi(x)))
```

10. Test the linear classifier with random values of the weights (w1,w2)

```
w=np.zeros(2)
w[0] = input('Enter w1: ')
w[1] = input('Enter w2: ')
output = np.zeros((len(X_train)))
prediction = np.zeros(len(X_train))
for i in range(len(X_train)):
    output[i]=predict(w,X_train.iloc[i])
print(output)
```

Sample result

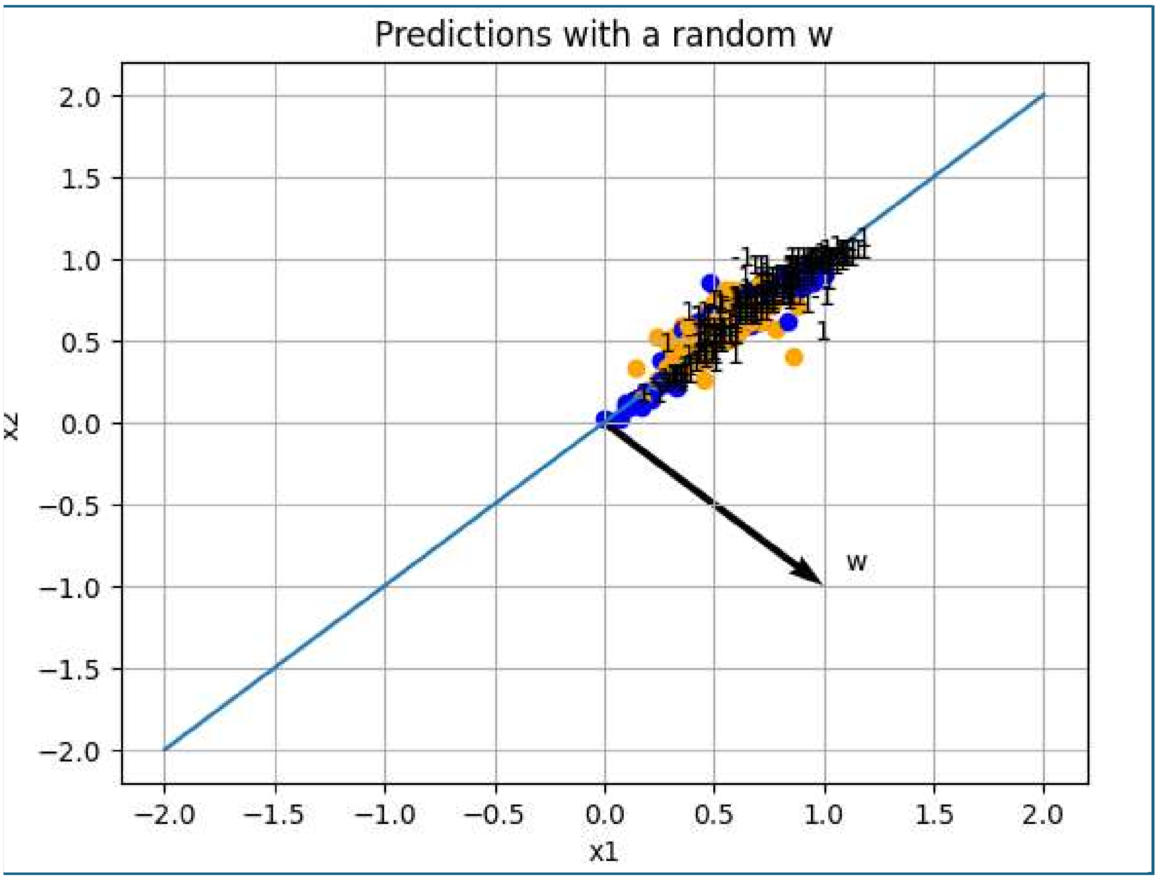


```
Enter w1: 1
Enter w2: -1
[ 1. -1. -1.  0. -1.  1.  1.  1. -1. -1.  0.  0.  0. -1.  1.  0. -1. -1.
  1.  0. -1. -1. -1. -1.  0. -1. -1.  0.  1.  1.  1.  1. -1.  0.  1.  0.
 -1. -1. -1.  0.  0.  1. -1.  1.  0.  1.  1. -1.  1.  0.  0. -1. -1. -1.
 -1. -1.  1. -1. -1. -1. -1. -1.  1. -1.  0.  1. -1. -1. -1.  1. -1. -1.
  1. -1.  1.  1. -1.  0. -1.  1.  1.  1. -1. -1. -1.  1.  1.  1.  1.  1.
  0.  1.  0. -1. -1.  1. -1.  1. -1.  0. -1.  1. -1.  1.  1. -1.  1.  1.
 -1. -1. -1. -1.  1.  1.  1.  1. -1. -1.  1. -1.  1. -1. -1.  1.  0.  1.
  1. -1.  0. -1.  1.  1. -1. -1. -1. -1. -1.  1. -1.  1.  1. -1.  1. -1.
 -1.  1.  1. -1.  1.  0.  1. -1.  1.  1.  1.  0. -1. -1. -1. -1. -1.  1.
 -1.  0.  1.  1. -1.  1.  0. -1. -1.  0. -1.  0. -1. -1.  1. -1. -1.  0.
  1.  1. -1. -1. -1. -1.  1. -1. -1. -1. -1. -1.  1. -1. -1.  1.  1.  1.
  1. -1. -1.  0. -1. -1.  1.  1.  1. -1. -1. -1.  0.  0.  1. -1.  0. -1.
 -1. -1. -1. -1.  1. -1. -1.  1. -1.  1.  1. -1. -1. -1.  1.  1.  1. -1.
 -1.  1.  0.  1.  0. -1. -1.  1.  0. -1.  1. -1.  1. -1.  1.  1.  1.  0.
  1. -1.  1.  1.  0.  1.  1.  1. -1.  1. -1. -1. -1. -1.  1. -1. -1. -1.
 -1.  1. -1. -1.  0. -1.  1.  1. -1.]
```

11. Plot the predictions made with the entered value of w and check the classification result

```
from matplotlib import pyplot as plt

plot = plt.figure
plt.grid(True)
for i in range(len(y_train)):
  if (y_train.iloc[i]==1):
    plt.scatter
(X_train.iloc[i][0],X_train.iloc[i][1],color='orange')
    plt.text(X_train.iloc[i][0]+0.1,X_train.iloc[i][1]+0.1,str
(y_train.iloc[i]))
  else:
    plt.scatter
(X_train.iloc[i][0],X_train.iloc[i][1],color='blue')
    plt.text(X_train.iloc[i][0]+0.1,X_train.iloc[i][1]+0.1,str
(y_train.iloc[i]))

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predictions with a random w')
x = np.linspace(-2,2)
plt.plot(x,boundary(w,x))
plt.quiver(0,0,w[0],w[1], angles='xy', scale_units='xy',
scale=1)
plt.text(w[0]+0.1,w[1]+0.1,str('w'))
plt.show()
```

Sample result



12. Define the Hinge Loss function

```
def hinge_loss(w, x, y):
    return max(0, 1 - y * np.dot(w, phi(x)))
```

13. Define the TrainLoss function as the average Hinge Loss on the training data

```
def train_loss(w, training_data):
    loss = 0
    for x, y in training_data:
        loss += hinge_loss(w, x, y)
    return loss / len(training_data)
```

14. Define the `gradient_train_loss()` function as the average gradient of the Hinge Loss on the training data

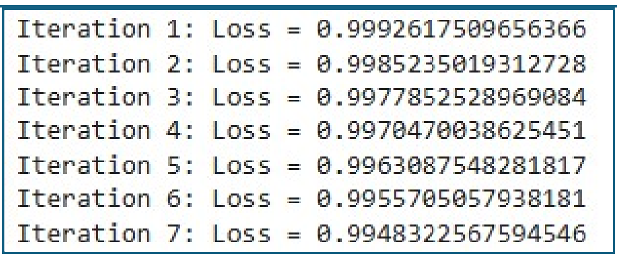
```
def gradient_train_loss(w, training_data):
  gradient = np.zeros(len(w))
  for x, y in training_data:
    if hinge_loss(w, x, y) > 0:
      gradient += -y * phi(x)
    else:
      gradient += 0
  return gradient / len(training_data)
```

15. Define the `gradient_descent()` function

```
def gradient_descent(training_data, learning_rate,
num_iterations):
  w = np.zeros(2)
  for i in range(num_iterations):
    gradient = gradient_train_loss(w, training_data)
    w -= learning_rate * gradient
    print(f"Iteration {i+1}: Loss = {train_loss(w,
training_data)}")
  return w
```

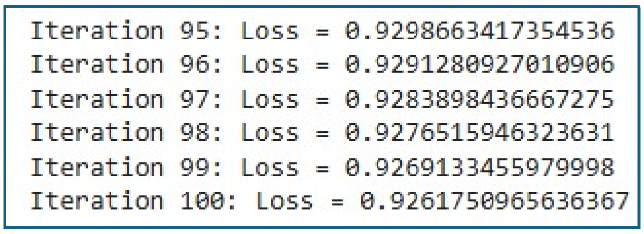
16. Run the `gradient_descent()` function on the training data using a learning rate (step size of the GD algorithm) of 0.1 and 100 iterations

```
learning_rate = 0.1
num_iterations = 100
training_data = list(zip(X_train.values, y_train.values))
w = gradient_descent(training_data, learning_rate,
num_iterations)
```



```
Iteration 1: Loss = 0.9992617509656366
Iteration 2: Loss = 0.9985235019312728
Iteration 3: Loss = 0.9977852528969084
Iteration 4: Loss = 0.9970470038625451
Iteration 5: Loss = 0.9963087548281817
Iteration 6: Loss = 0.9955705057938181
Iteration 7: Loss = 0.9948322567594546
```

…

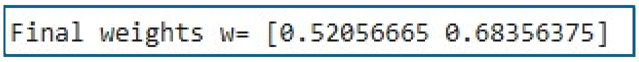


```
Iteration 95: Loss = 0.9298663417354536
Iteration 96: Loss = 0.9291280927010906
Iteration 97: Loss = 0.9283898436667275
Iteration 98: Loss = 0.9276515946323631
Iteration 99: Loss = 0.9269133455979998
Iteration 100: Loss = 0.9261750965636367
```

17. Print the final weights obtained by the gradient descent algorithm

```
print(f"Final weights w= {w}")
```

Sample result

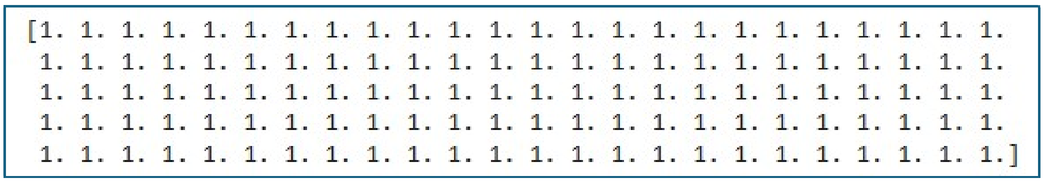


```
Final weights w= [0.52056665 0.68356375]
```

18. Calculate the prediction outcomes based on the obtained final weight

```
predictions = np.zeros(len(y_test))
for i in range(len(y_test)):
    predictions[i]=predict(w,X_test.iloc[i])
print(predictions)
```

Sample result



```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

19. Plot the predictions made with the entered value of w and check the classification result

```
from matplotlib import pyplot as plt

plot = plt.figure
plt.grid(True)
for i in range(len(y_test)):
  if (predictions[i]==1):
    plt.scatter
(X_test.iloc[i][0],X_test.iloc[i][1],color='orange')
    plt.text(X_test.iloc[i][0]+0.1,X_test.iloc[i][1]+0.1,str(p
redictions[i]))
  else:
    plt.scatter
(X_test.iloc[i][0],X_test.iloc[i][1],color='blue')
    plt.text(X_test.iloc[i][0]+0.1,X_test.iloc[i][1]+0.1,str(p
redictions[i]))

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predictions with a trained w')
x = np.linspace(-2,2)
plt.plot(x,boundary(w,x))
plt.quiver(0,0,w[0],w[1], angles='xy', scale_units='xy',
scale=1)
plt.text(w[0]+0.1,w[1]+0.1,str('w'))
plt.show()
```

Sample result

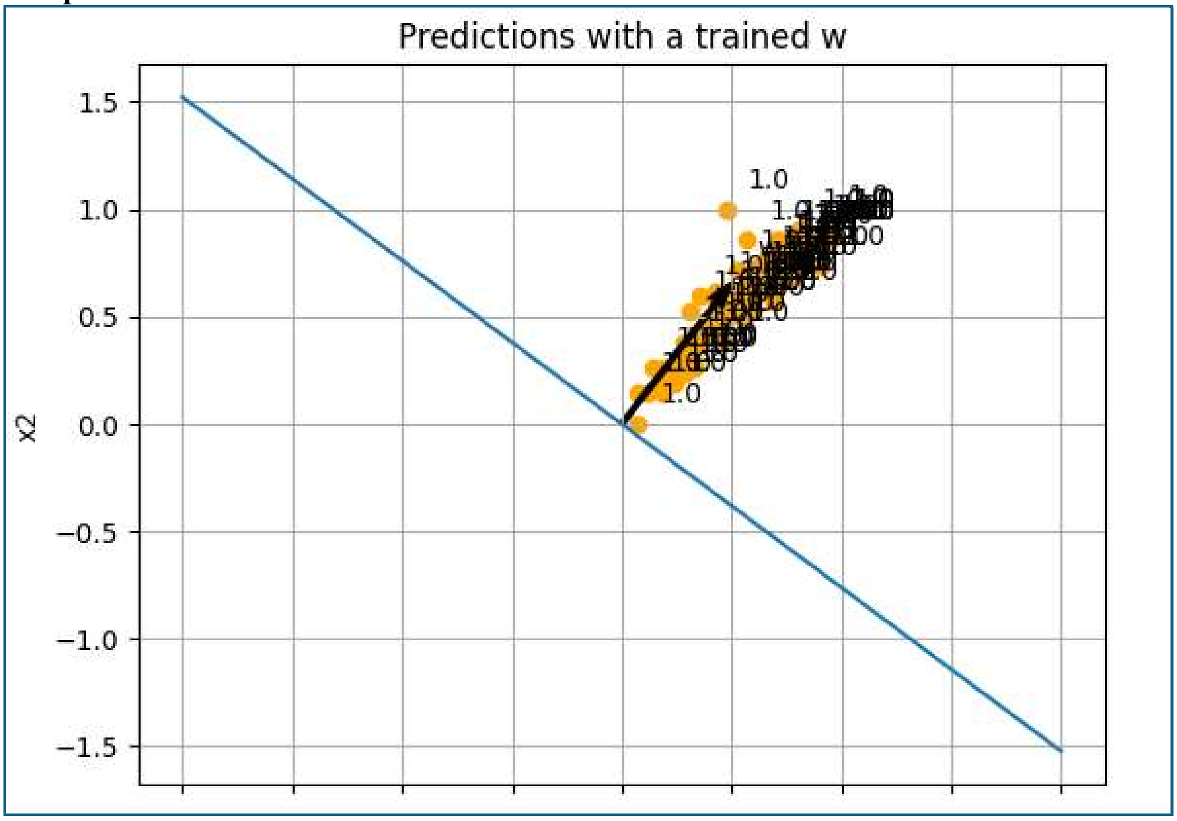
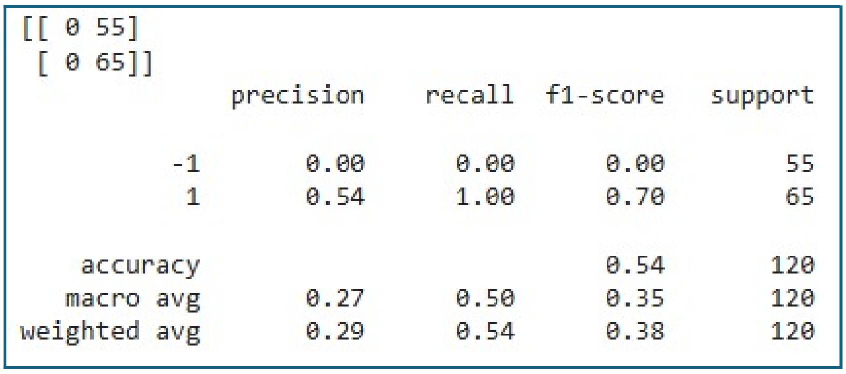


Figure 2.1.2. Result of the binary linear classifier with Hinge Loss

20. Calculate the confusion matrix and produce the classification report

```
from sklearn.metrics import confusion_matrix,
classification_report
cm = confusion_matrix(y_test, predictions)
cr = classification_report(y_test, predictions)
print(cm)
print(cr)
```

Sample result

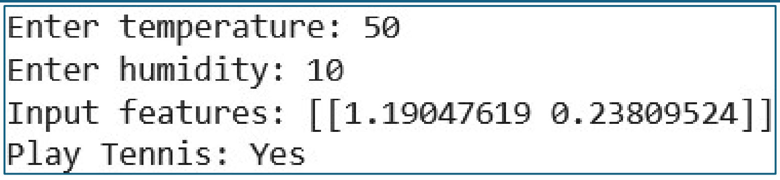


```
[[ 0 55]
 [ 0 65]]
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00        55
           1       0.54      1.00      0.70        65

    accuracy                           0.54       120
   macro avg       0.27      0.50      0.35       120
weighted avg       0.29      0.54      0.38       120
```

Figure 2.1.3. Confusion matrix and classification report

21. Apply the trained linear classifier to make new predictions by entering the new values of Temperature and humidity

```
k = float(input('Enter temperature: '))/(max(dataset.iloc[:,2])-min(dataset.iloc[:,2]))
l = float(input('Enter humidity: '))/(max(dataset.iloc[:,3])-min(dataset.iloc[:,3]))
X_new=(k,l)
print(f"Input features: {X_new}")
playTennis=predict(w,X_new)
if playTennis == -1:
    playTennis = 'No'
else:
    playTennis = 'Yes'
print(f"Play Tennis: {playTennis}")
```



```
Enter temperature: 50
Enter humidity: 10
Input features: [[1.19047619 0.23809524]]
Play Tennis: Yes
```

Figure 2.1.4. Prediction results with normalized new input data

Exercise 2 - Multi-class Linear classification with Hinge Loss

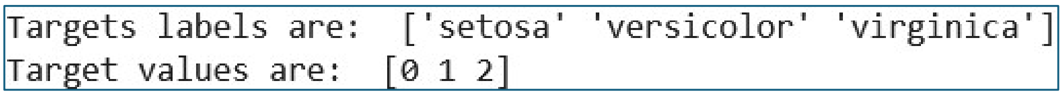
In this exercise, we implement a linear classifier and run it on the Iris flower dataset. Our goals are: **a)** Perform classification, **b)** Compute and visualize the hinge loss, **c)** Understand how the output values are converted into predicted classes.

1. Set-up Open a new notebook and import the following packages: numpy as np and pandas as pd

2. Import and upload the Iris dataset and visualize the names and the values of the target labels

Hints!

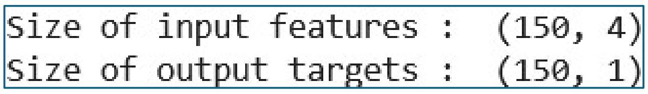
- Import the library datasets from sklearn, and use the method `datasets.load_iris()`
- To visualize the names and values of the targets, use the dataset's attributes `target_names` and `target`



```
Targets labels are:  ['setosa' 'versicolor' 'virginica']
Target values are:  [0 1 2]
```

Figure 2.2.1. Names and values of the target labels

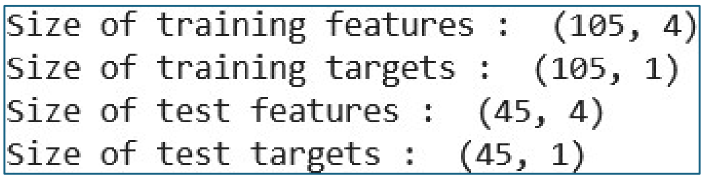
3.Define input features and output targets as dataframes and specify their columns, then visualize the size of input features and output targets in the dataset



```
Size of input features :  (150, 4)
Size of output targets :  (150, 1)
```

Figure 2.2.2. Size of input and output data

4. Split data into training and test sets using a split rate of 0.3, and visualize the size of each split

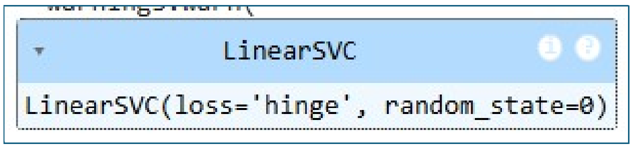


```
Size of training features :  (105, 4)
Size of training targets :  (105, 1)
Size of test features :  (45, 4)
Size of test targets :  (45, 1)
```

Figure 2.2.3. Size of input and output data in the training and test sets

5. Create a linear classifier based on support vector machines with hinge loss using the method `svm.LinearSVC()` with the following attributes: `loss="hinge"` and `random_state=0`
   Hint! You need to import `svm` from `sklearn`

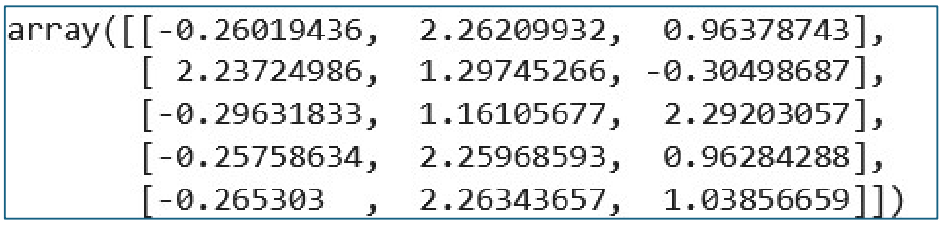
6. Train the linear classifier on the input and output training subsets using the Method `linearClassifierName.fit()`



```
LinearSVC
LinearSVC(loss='hinge', random_state=0)
```

Figure 2.2.4. Linear SVM classifier based on Hinge loss

7. Run the trained model on the test set using the method `linearClassifierName.decision_function()` and visualize the first 5 (raw) output values . The result should look like this:

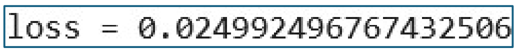


```
array([[-0.26019436,  2.26209932,  0.96378743],
       [ 2.23724986,  1.29745266, -0.30498687],
       [-0.29631833,  1.16105677,  2.29203057],
       [-0.25758634,  2.25968593,  0.96284288],
       [-0.265303  ,  2.26343657,  1.03856659]])
```

Figure 2.2.5. First 5 output results : Notice that for each row, there 3 values, which the score of the linear classifier for each class

8. Calculate the mean hinge loss on the test set using the target and the output values using the method `hinge_loss()`

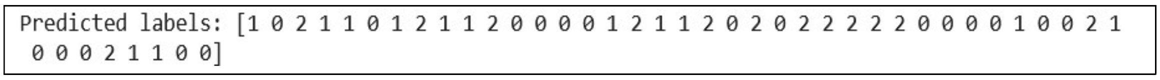
Hints! The input variables to the method `hinge_loss()` are the target values and the predicted output values (not the predicted classes) !



```
loss = 0.024992496767432506
```

Figure 2.2.6. Obtained loss

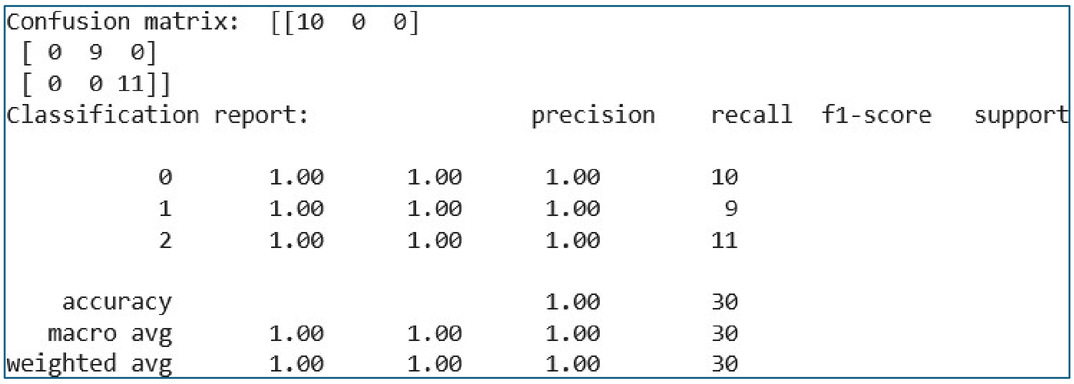
9. Calculate the values of the predicted classes using the method `linearClassifierName.predict()`



```
Predicted labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0]
```

Figure 2.2.7. Predicted labels

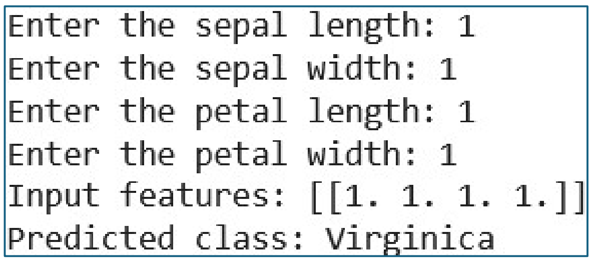
10. Compute and visualize the confusion matrix and the evaluation metrics (Accuracy, precision, recall and F1-score) using the methods `confusion matrix()` and `classification_report()`



```
Confusion matrix:  [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
Classification report:                precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Figure 2.2.8. Confusion matrix and classification report of the Decision Tree model trained on Fisher's Iris dataset

11. Make new predictions by entering new input features and using the method `linearClassifierName.predict().` Convert the predict values to the names of the flowers (0: 'Setosa', 1: 'Versicolor', 2: 'Virginica')



```
Enter the sepal length: 1
Enter the sepal width: 1
Enter the petal length: 1
Enter the petal width: 1
Input features: [[1. 1. 1. 1.]]
Predicted class: Virginica
```

Figure 2.2.9. Classification result for new input data.