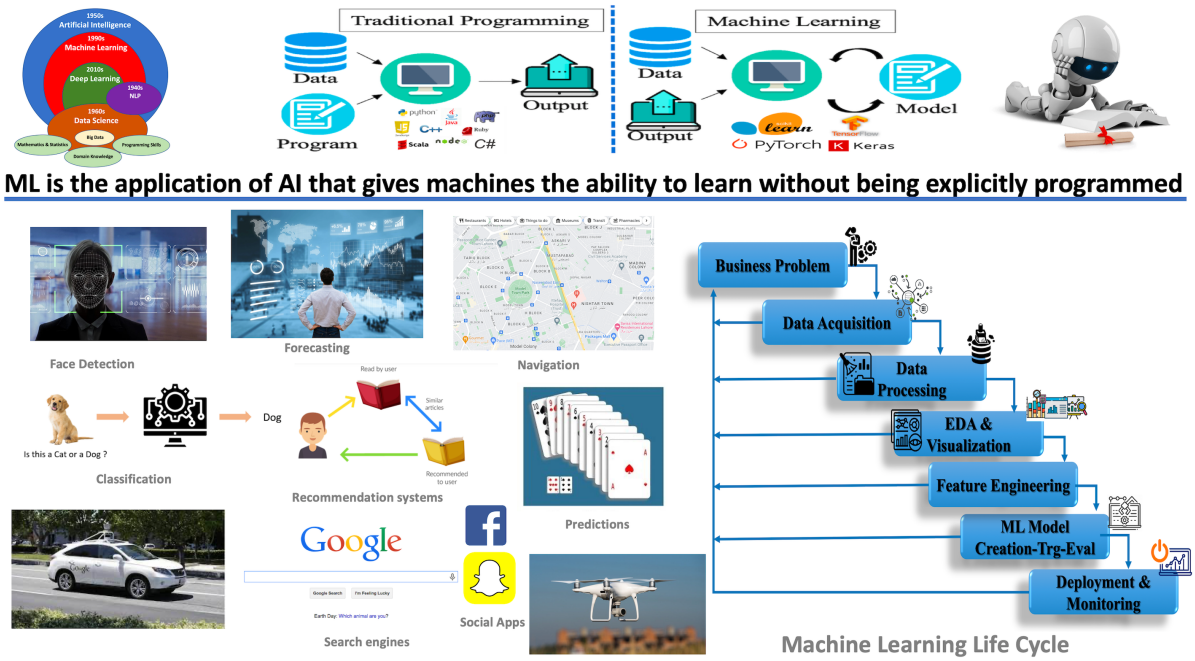
Department of Data Science

Course: Tools and Techniques for Data Science

Instructor: Muhammad Arif Butt, Ph.D.

Lecture 6.23 (Logistic Regression: Part-IV)

Open in Colab (https://colab.research.google.com/github/arifpucit/data-science/blob/master/Section-4-Mathematics-for-Data-Science/Lec-4.1(Descriptive-Statistics).ipynb)



Learning agenda of this notebook

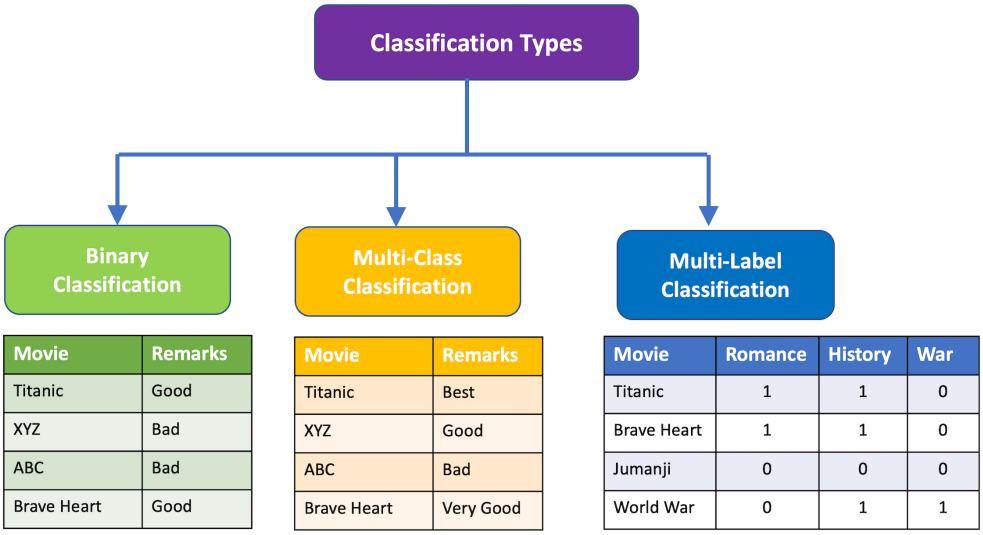
- Theoratical Background for Multinomial Classification
  - Binary vs Multi-Class vs Multi-Label Classification
  - One vs Rest (aka One vs All)
  - Multinomial / Softmax Logistic Regression
- Multi-Class Classification on `iris` Dataset
  - Load Dataset
  - Exploratory Data Analysis

    - Model Training (`ovr` and `multinomial` Logistic Regression)
- Evaluation Metrics for Multi-Class Classification
    - Recap of Confusion Matrix and Evaluation Metrics for Binary Classification
    - Confusion Matrix for Multi-Class Classifier
    - Accuracy, Precision, Recall and F-1 Scores for Multi-Class Problem
    - `Macro Average` Precision, Recall and F-1 Scores for Multi-Class Problem
    - `Micro Average` Precision, Recall and F-1 Scores for Multi-Class Problem
    - `Weighted Average` Precision, Recall and F-1 Scores for Multi-Class Problem
- Evaluate the Logistic Regression Model (`iris dataset`)
- Visualize Decision Boundries for `iris dataset`
- Task To Do (Assignment)

1. Theoratical Background for Multinomial Classification

a. Binary vs Multi-Class vs Multi-Label Classification

- **Binary**



Movie	Remarks
Titanic	Good
XYZ	Bad
ABC	Bad
Brave Heart	Good

Movie	Remarks
Titanic	Best
XYZ	Good
ABC	Bad
Brave Heart	Very Good

Movie	Romance	History	War
Titanic	1	1	0
Brave Heart	1	1	0
Jumanji	0	0	0
World War	0	1	1

  Classification is where each data sample is assigned one and only one label from two mutually exclusive classes.
    - Email: Spam/Not Spam
    - Online Transaction: Fraudlent (yes/no)
    - Sentiments: Positive/Negative
    - Tumor: Malignant/Benign
    - `Algorithms:` Logistic Regression, Support Vector Machine, k-Nearest Neighbors, Decision Trees, Naive Bayes

- **Multi-class Classification** is where each data sample is assigned one and only one label from more than two classes.
    - Sentiments: Positive/Negative/Neutral
    - Emotions: Happy/Sad/Surprised/Angry
    - Fruit Images: Apple/Orange/Banana/Pear
    - Digit Recognition: Zero/One/Two/.../Nine
    - `Algorithms:` K-Nearest Neighbors, Decision Trees, Naive Bayes, Random Forest, Gradient Boosting. (Logistic Regression and SVM uses one-vs-rest to implement multi-class classification)

- **Multi-label Classification** is where each data sample can be assigned zero or more labels.

- Genere of a Movie: Romance/History/War
- Document Clustering: Sports/Politics/Cricket/Economics/Religion
- Objects in an image: House/Sun/Road/Car/Cat/Dog
- `Algorithms:` Scikit-multilearn is a library built on top of scikit-learn that is used for for multi-label classification. Multi-label K Nearest Neighbours, Multi-label Decision Trees, Multi-label Random Forests, Multi-label Gradient Boosting

> - A multiclass classifier must assign one and only one class or label to each data sample, while a multilabel classifier can assign zero or more classes or labels to the same data sample.

Logistic Regression's Hyperparameter `multi_class` :

In [ ]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.get_params()
```

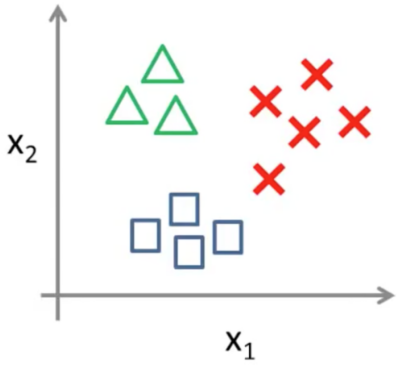
- The `multi_class` hyperparameter of Logistic Regression model can take three values: `auto` , `ovr` , and `multinomial` , with `auto` as the default value
  - If `multi_class = 'ovr'` , then it will create multiple binary classification models and fit one for each output label
  - If `multi_class = 'multinomial'` , then Sklearn will use a bit different loss function (multinomial loss), and is also known as softmax Logistic Regression. Can be applied even for binary classification problem. However, remember that `multinomial` is unavailable when `solver = 'liblinear'`
  - If `multi_class = 'auto'` , then
    - `'ovr'` is used if the data is binary, or if `solver = 'liblinear'` ,
    - otherwise selects `multinomial`

b. One vs Rest (aka One vs All)

OVR technique use multiple binary classifiers to implement multi-class classification

Main Dataset

Features		Classes
x1	x2	
-	-	G
-	-	B
-	-	R
-	-	G
-	-	B
-	-	R

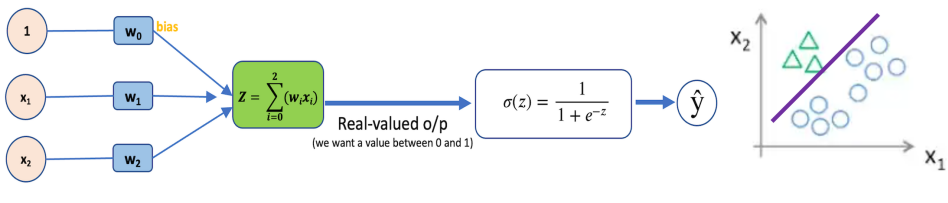


Features		Green	Blue	Red
x1	x2			
-	-	1	0	0
-	-	0	1	0
-	-	0	0	1
-	-	1	0	0
-	-	0	1	0
-	-	0	0	1

$$J(w) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

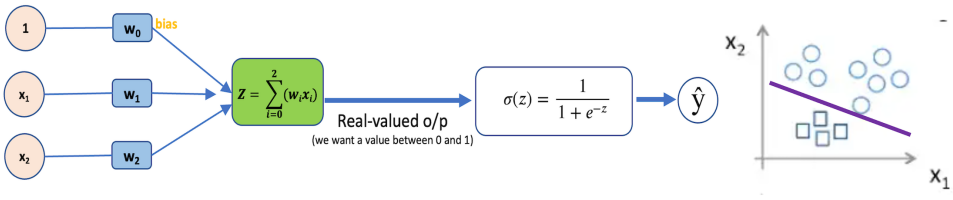
Training Dataset 1
Class: Green Triangle

Features		Green
x1	x2	
-	-	1
-	-	0
-	-	0
-	-	1
-	-	0
-	-	0



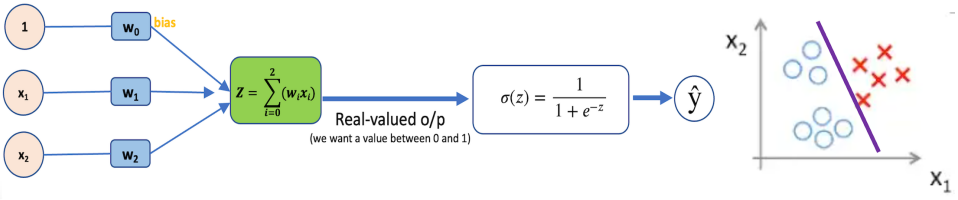
Training Dataset 2
Class: Blue Square

Features		Blue
x1	x2	
-	-	0
-	-	1
-	-	0
-	-	0
-	-	1
-	-	0



Training Dataset 3
Class: Red Cross

Features		Red
x1	x2	
-	-	0
-	-	0
-	-	1
-	-	0
-	-	0
-	-	1



Working of OVR: Multiple binary classifiers to perform a multi-class classification

- After training is done, we will have a total of nine coefficients or weights (three for each dataset)
- Given a new input vector, it will be fed to all of the above three models and each model will output a probability value of belonging of the input vector to Green, Blue and Red class
- The maximum probability value class will be assigned to the new input vector

Limitation of OVR:

- For `K` classes, we need to train `K` binary classifiers. In above example

c. Multinomial / Softmax Logistic Regression

Multinomial Logistic Regression uses a bit modified Loss Function (cross entropy loss) for training and softmax function instead of sigmoid function

Loss Function for Multinomial Logistic Regression

Features		Green	Blue	Red
x1	x2	$y_{k=1}$	$y_{k=2}$	$y_{k=3}$
x_{11}	x_{12}	1	0	0
x_{21}	x_{22}	0	1	0
x_{31}	x_{32}	0	0	1

$$J(w) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$

- Let us expand the above loss function for the dataset having just three rows:

$$J(w) = y_1^{(1)}log(\hat{y}_1^{(1)}) + y_2^{(1)}log(\hat{y}_2^{(1)}) + y_3^{(1)}log(\hat{y}_3^{(1)}) + y_1^{(2)}log(\hat{y}_1^{(2)}) + y_2^{(2)}log(\hat{y}_2^{(2)}) + y_3^{(2)}log(\hat{y}_3^{(2)}) + y_1^{(3)}log(\hat{y}_1^{(3)}) + y_2^{(3)}log(\hat{y}_2^{(3)}) + y_3^{(3)}log(\hat{y}_3^{(3)})$$

$$J(w) = y_1^{(1)}log(\hat{y}_1^{(1)}) + y_2^{(2)}log(\hat{y}_2^{(2)}) + y_3^{(3)}log(\hat{y}_3^{(3)})$$

- Now we have $y_1^{(1)}$, $y_2^{(2)}$ and $y_3^{(3)}$ in our dataset, and we have to calculate $\hat{y}_1^{(1)}$, $\hat{y}_2^{(2)}$ and $\hat{y}_3^{(3)}$

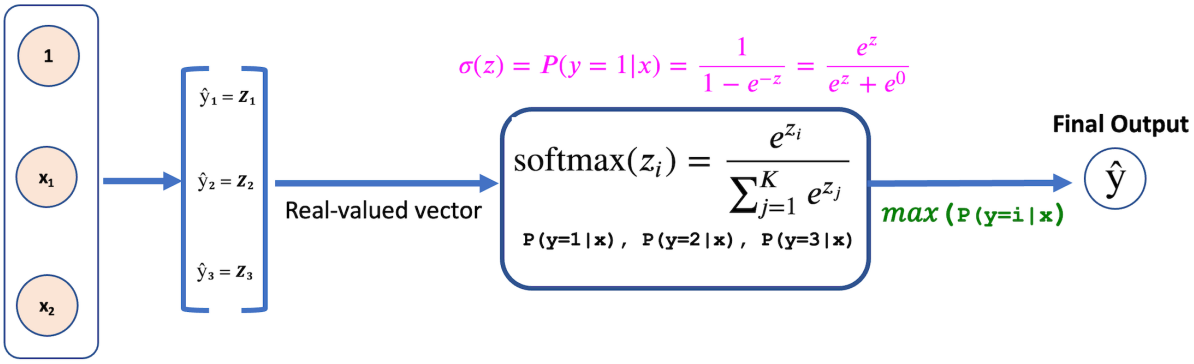
$$\hat{y}_1^{(1)} = w_1^{(1)}x_{11} + w_2^{(1)}x_{12} + w_0^{(1)}$$

$$\hat{y}_2^{(2)} = w_1^{(2)}x_{21} + w_2^{(2)}x_{22} + w_0^{(2)}$$

$$\hat{y}_3^{(3)} = w_1^{(3)}x_{31} + w_2^{(3)}x_{32} + w_0^{(3)}$$

- The nine coefficients (weights) can be calculated using Gradient Descent, which will calculate the nine partial derivatives:

$$\frac{\partial J}{\partial w_1^{(1)}}, \frac{\partial J}{\partial w_2^{(1)}}, \frac{\partial J}{\partial w_0^{(1)}} \qquad \frac{\partial J}{\partial w_1^{(2)}}, \frac{\partial J}{\partial w_2^{(2)}}, \frac{\partial J}{\partial w_0^{(2)}} \qquad \frac{\partial J}{\partial w_1^{(3)}}, \frac{\partial J}{\partial w_2^{(3)}}, \frac{\partial J}{\partial w_0^{(3)}}$$

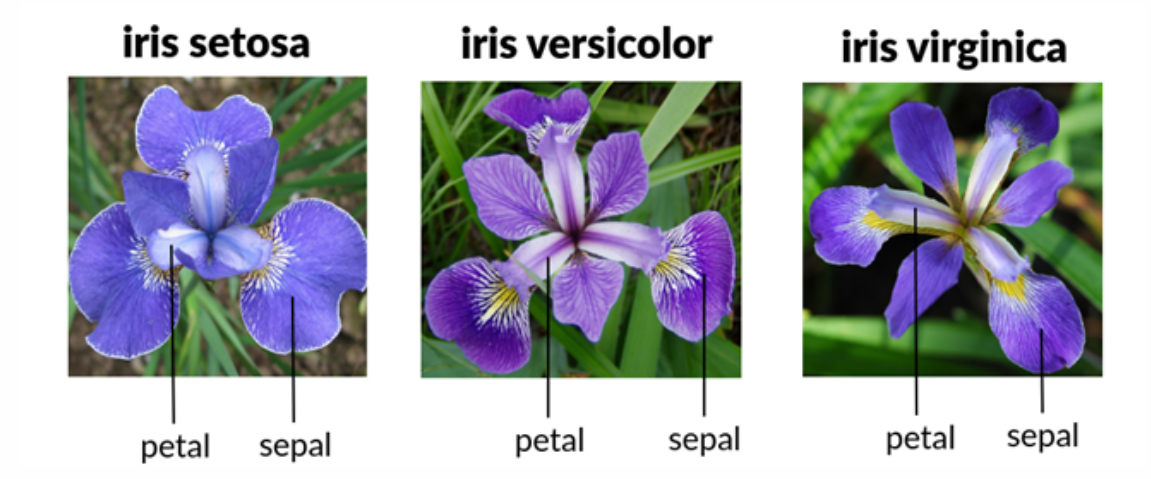


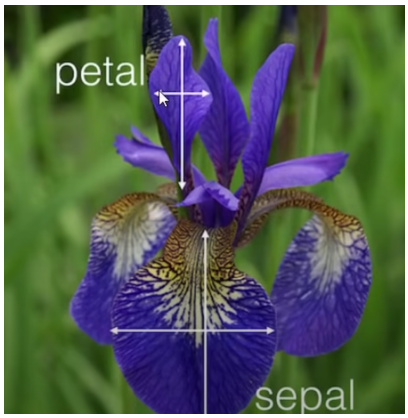
In [ ]:
```
# softmax() is passed three z values and it maps them to three probabili
# The maximum probability specifies that the input vector belong to that
from numpy import exp
def softmax(x):
    return exp(x) / exp(x).sum()

zs = [6.5, 7.2, 5.9]
print(softmax(zs))
```

2. Multi-Class Classification on `iris` Dataset

a. Load Dataset

- The use of multiple measurements in Taxonomic Problems (R.A. Fisher): https://hekyll.services.adelaide.edu.au/dspace/bitstream/2440/15227/1/138.pdf (https://hekyll.services.adelaide.edu.au/dspace/bitstream/2440/15227/1/138.pdf)
- Download Dataset: https://archive.ics.uci.edu/ml/datasets/Iris (https://archive.ics.uci.edu/ml/datasets/Iris)





Datasets that come bundled with Seaborn Library's `load_dataset()` method:

In [1]:
```
import seaborn as sns
print(sns.get_dataset_names())
```

```
['anagrams', 'anscombe', 'attention', 'brain_networks', 'car_crashes',
'diamonds', 'dots', 'dowjones', 'exercise', 'flights', 'fmri', 'geyse
r', 'glue', 'healthexp', 'iris', 'mpg', 'penguins', 'planets', 'seaic
e', 'taxis', 'tips', 'titanic']
```

Load `iris` dataset using Seaborn's `load_dataset('iris')` method:

In [2]:
```
df = sns.load_dataset('iris') # Returns a dataframe
df.sample(10, random_state=54)
```

Out[2]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
45	4.8	3.0	1.4	0.3	setosa
91	6.1	3.0	4.6	1.4	versicolor
103	6.3	2.9	5.6	1.8	virginica
94	5.6	2.7	4.2	1.3	versicolor
96	5.7	2.9	4.2	1.3	versicolor
42	4.4	3.2	1.3	0.2	setosa
79	5.7	2.6	3.5	1.0	versicolor
116	6.5	3.0	5.5	1.8	virginica
4	5.0	3.6	1.4	0.2	setosa

Load `iris` dataset using Scikit-Learn Library's `load_iris()` method:

In [3]:
```
from sklearn import datasets
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np

iris = datasets.load_iris() # Returns a Sklearn's bunch object
type(iris)
```

Out[3]:
```
sklearn.utils.Bunch
```

In [4]:
```
iris.feature_names
```

Out[4]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

In [5]:
```
iris.target_names
```

Out[5]:
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

In [6]:
```
iris.data
```

Out[6]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
       [5.8, 4. , 1.2, 0.2],
       [5.7, 4.4, 1.5, 0.4],
       [5.4, 3.9, 1.3, 0.4],
       [5.1, 3.5, 1.4, 0.3],
       [5.7, 3.8, 1.7, 0.3],
```

In [7]:
```
print(iris.DESCR)
```

.. _iris_dataset:

Iris plants dataset

Data Set Characteristics:

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the cl
ass
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min Max   Mean   SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length: 4.3 7.9   5.84   0.83   0.7826
    sepal width: 2.0 4.4   3.05   0.43   -0.4194
    petal length: 1.0 6.9   3.76   1.76   0.9490 (high!)
    petal width: 0.1 2.5   1.20   0.76   0.9565 (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset i
s taken
from Fisher's paper. Note that it's the same as in R, but not as in th
e UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature. Fisher's paper is a classic in the fi
eld and
is referenced frequently to this day. (See Duda & Hart, for example.)
The
data set contains 3 classes of 50 instances each, where each class ref
ers to a
type of iris plant. One class is linearly separable from the other 2;
the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic probl
ems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributio
ns to
     Mathematical Statistics" (John Wiley, NY, 1950).

```
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene
Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New Sys
tem
     Structure and Classification Rule for Recognition in Partially Ex
posed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Tra
nsactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLA
SS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

In [8]:
```
# Convert the Sklearn's bunch object to a dataframe
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target
df.sample(10, random_state=54)
```

Out[8]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
45	4.8	3.0	1.4	0.3	0
91	6.1	3.0	4.6	1.4	1
103	6.3	2.9	5.6	1.8	2
94	5.6	2.7	4.2	1.3	1
96	5.7	2.9	4.2	1.3	1
42	4.4	3.2	1.3	0.2	0
79	5.7	2.6	3.5	1.0	1
116	6.5	3.0	5.5	1.8	2
4	5.0	3.6	1.4	0.2	0

In [9]:
```
df.rename(columns={'sepal length (cm)':'sepal_length',
                   'sepal width (cm)':'sepal_width',
                   'petal length (cm)':'petal_length',
                   'petal width (cm)': 'petal_width'},
                    inplace=True)
df.sample(10, random_state=54)
```

Out[9]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	0
45	4.8	3.0	1.4	0.3	0
91	6.1	3.0	4.6	1.4	1
103	6.3	2.9	5.6	1.8	2
94	5.6	2.7	4.2	1.3	1
96	5.7	2.9	4.2	1.3	1
42	4.4	3.2	1.3	0.2	0
79	5.7	2.6	3.5	1.0	1
116	6.5	3.0	5.5	1.8	2
4	5.0	3.6	1.4	0.2	0

b. Exploratory Data Analysis (EDA)

Since EDA has no real set methodolgy, the following is a short check list you might want to walk through:

1. What question(s) are you trying to solve (or prove wrong)?
2. What's missing from the data and how do you deal with it?
3. Are there any outliers and how to treat them?
4. What kind of data do you have and how do you treat different types?
5. How can you add, change or remove features to get more out of your data?
6. How your data is distributed and the correlation between different variables?

In [10]:
```
# Check out the data types and null values if any
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    int64  
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

In [11]:
```
# Check out the different descriptive statistics of numeric columns
df.describe()
```

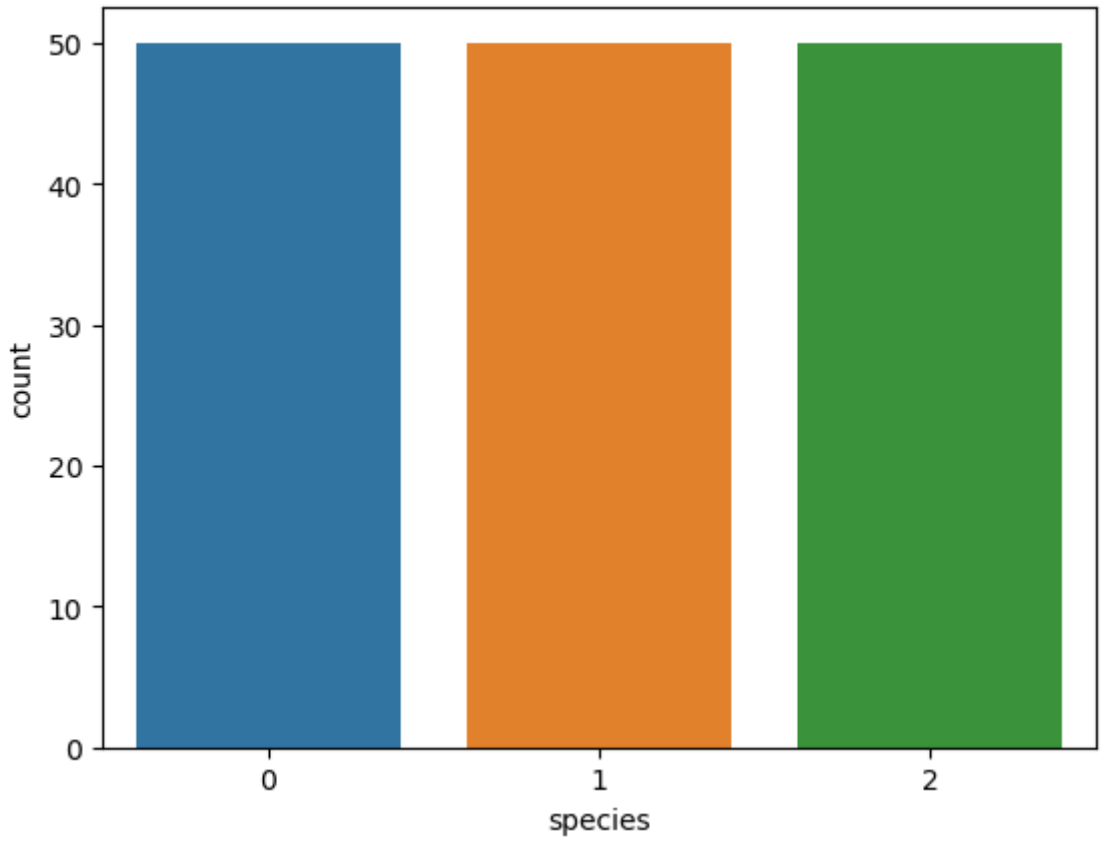
Out[11]:

	sepal_length	sepal_width	petal_length	petal_width	species
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

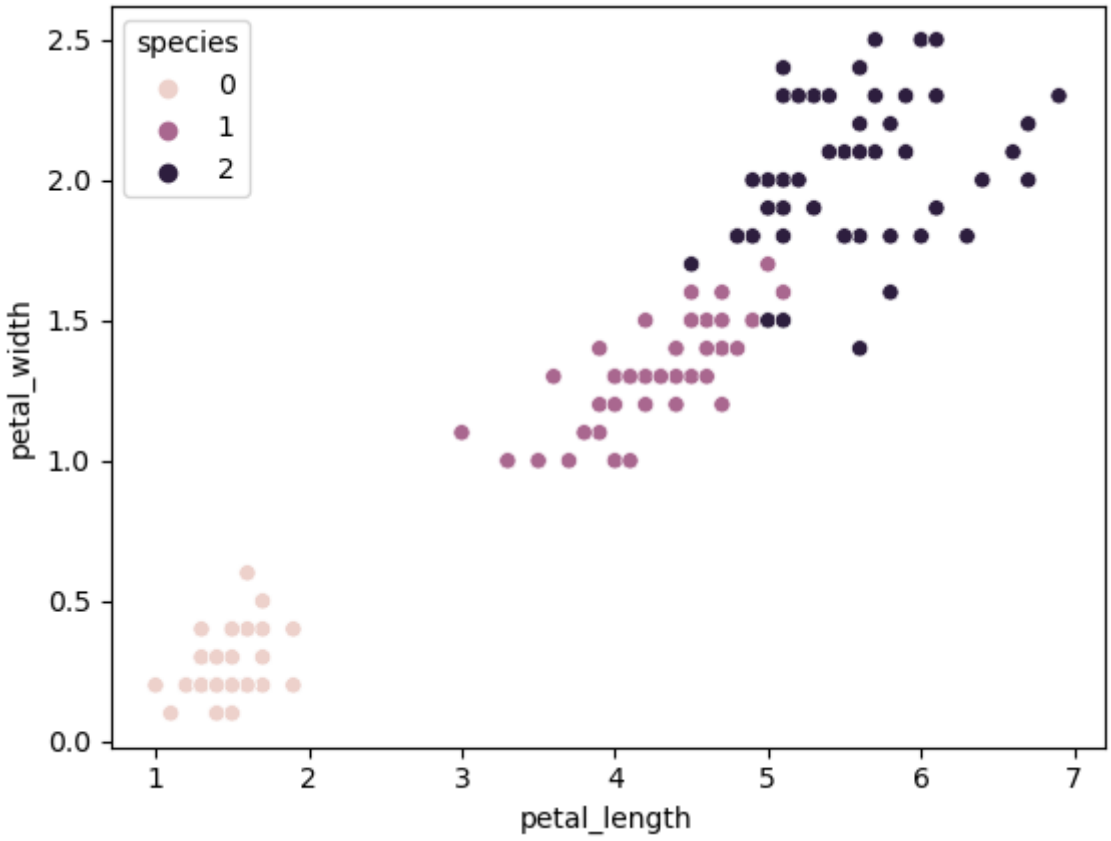
In [13]:
```
# Checkout the number of observations of each class in dataset
df['species'].value_counts()
```

Out[13]:
```
0    50
1    50
2    50
Name: species, dtype: int64
```

In [14]:
```
# Visually checkout the number of observations of each class in dataset
sns.countplot(x=df['species']);
```

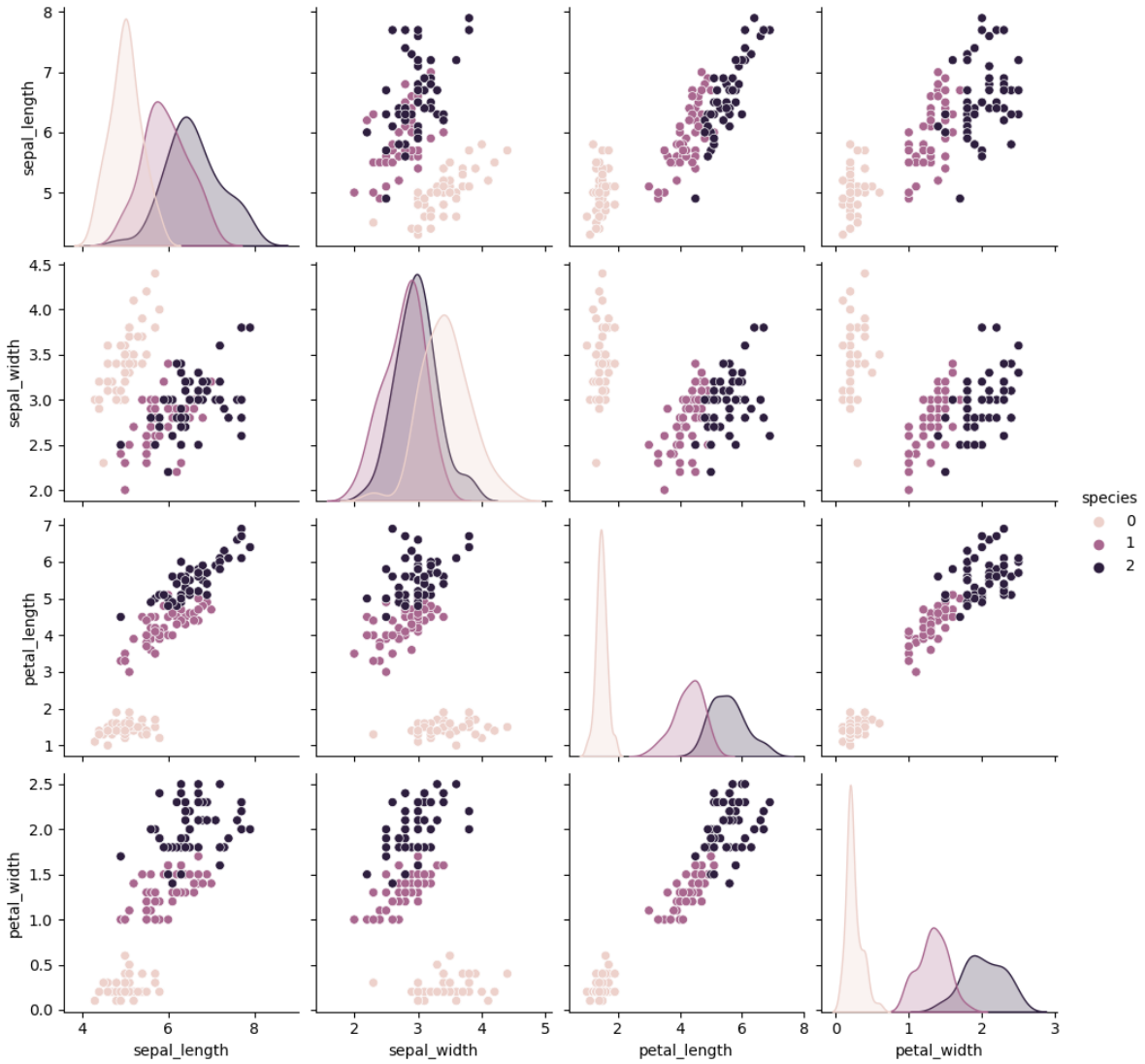


In [17]:
```
# Visually checkout how separated the data points of the three classes a
sns.scatterplot(x='petal_length', y='petal_width', data=df, hue='species
```

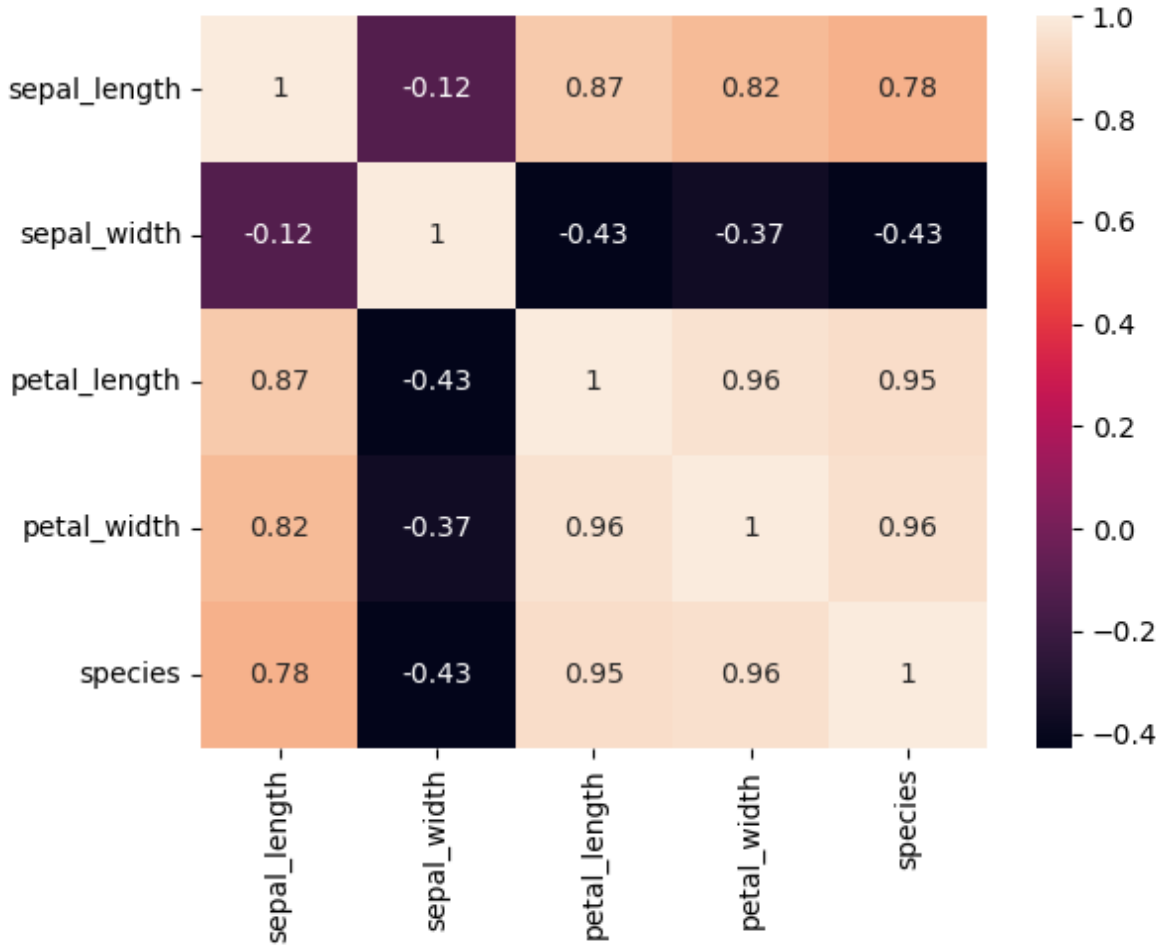


- Note that class 0 (Setosa) is easily or linearly separable from the other two classes i.e., Versicolor and Virginica based on two features petal_length and petal_width
- However, Versicolor and Virginica are bit similar and not linearly separable

In [19]:
```
sns.pairplot(df, hue='species');
```



In [20]:
```
sns.heatmap(df.corr(), annot=True);
```



c. Model Training (`ovr` and `multinomial` Logistic Regression)

- Since the dataset has
  - No missing values
  - No outliers
  - No categorical columns
  - All input feature columns has same unit of measurement (centimeter)
- Therefore, we need not to perform any preprocessing on this dataset, and this is all ready to be fed to the machine learning model

In [22]:
```
df
```

Out[22]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

Do a Train-Test Split:

In [23]:
```
from sklearn.model_selection import train_test_split

X = df.drop('species', axis=1) # df.iloc[:,0:2]
y = df['species']              # df.iloc[:,-1]

X = df.drop('species', axis=1)
y = df['species']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[23]:
```
(120, 120, 30, 30)
```

Instentiate and Train the `LogisticRegression()` Model

In [24]:
```
from sklearn.linear_model import LogisticRegression
```

In [25]:
```
#model = LogisticRegression(multi_class='ovr', solver='liblinear',penalt
model = LogisticRegression(multi_class='multinomial', solver='sag', pena
model.fit(X_train, y_train)
```

Out[25]:
```
LogisticRegression(max_iter=5000, multi_class='multinomial', solver='s
ag')
```

Solver Parameters

The solver parameter in logistic regression is used to specify the algorithm that will be used to find the optimal set of coefficients for the model. The five options for the solver parameter are `newton-cg` , `lbfgs` , `liblinear` , `sag` , and `saga` . The choice of solver will depend on the size and characteristics of your dataset, and the desired trade-off between speed and

accuracy. `newton-cg` , `lbfgs` and `sag` are more suited for larger datasets, while `liblinear` is more efficient for smaller datasets. `saga` is a solver that combines the benefits of `sag` and `lbfgs` and is generally recommended.

newton-cg: The Newton-Conjugate Gradient (Newton-CG) solver is an optimization algorithm used for finding the minimum of a function. It is a combination of Newton's method and the Conjugate Gradient method, and it is particularly well-suited for solving large-scale optimization problems in logistic regression.

lbfgs: The Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) solver is an optimization algorithm used for finding the minimum of a function. LBFGS is particularly well-suited for problems with a large number of parameters, such as logistic regression, as it can handle a large number of features and still converge quickly.

liblinear: The liblinear solver is an optimization algorithm used for solving linear classification problems, such as logistic regression. It is based on a linear support vector machine (SVM) and is implemented in the C++ library LIBLINEAR.

saga: The SAGA (Stochastic Average Gradient Descent) solver is an optimization algorithm used for solving large-scale optimization problems, such as logistic regression. It is an extension of the standard stochastic gradient descent (SGD) algorithm and can handle both L1 and L2 regularization.

sag: The Stochastic Average Gradient (SAG) solver is an optimization algorithm used for solving large-scale optimization problems, such as logistic regression. It is a variant of the SAGA algorithm and also uses a mini-batch approach to update the parameters of the model by computing the gradient of the loss function with respect to the parameters using a small random subset of the training data.

Supported Penalties by Different Solvers are:

- `lbfgs` --> ['l2', None]
- `liblinear` --> ['l1', 'l2']
- `newton-cg` --> ['l2', None]
- `newton-cholesky` --> ['l2', None]
- `sag` --> ['l2', None]

Checkout the Model Coefficients and Intercepts

In [27]:
```
# Since there are four i/p features and three output classes, so for eac
print("Model Coefficients:\n",model.coef_)
```

```
Model Coefficients:
 [[-0.18100766  0.97389237 -2.2881706  -1.01122744]
 [ 0.45768068 -0.61827181 -0.12348637 -0.86240594]
 [-0.27667303 -0.35562056  2.41165697  1.87363337]]
```

In [28]:
```
# For each output class we will have one intercept term
print("Model Intercepts:\n",model.intercept_)
```

```
Model Intercepts:
 [  7.54061556   2.86691887 -10.40753443]
```

Carry out the Prediction on `X_test`

In [29]:
```
y_pred = model.predict(X_test)
y_pred
```

Out[29]:
```
array([0, 0, 1, 2, 1, 1, 0, 1, 2, 0, 0, 2, 2, 2, 1, 1, 2, 2, 0, 0, 1,
2,
       1, 1, 2, 1, 1, 0, 1, 1])
```

In [30]:
```
# Check out the Probabilities of each Class
probs = model.predict_proba(X_test)
probs
```

Out[30]:
```
array([[9.80708912e-01, 1.92909701e-02, 1.18058279e-07],
       [9.64745530e-01, 3.52541598e-02, 3.10042449e-07],
       [8.01872426e-03, 8.07131447e-01, 1.84849829e-01],
       [9.76144787e-05, 1.21042856e-01, 8.78859529e-01],
       [1.85201385e-02, 9.05163101e-01, 7.63167606e-02],
       [2.37989093e-02, 9.01745377e-01, 7.44557139e-02],
       [9.83928919e-01, 1.60709626e-02, 1.17953825e-07],
       [7.18845901e-02, 9.22860604e-01, 5.25480559e-03],
       [1.72353840e-04, 1.66787615e-01, 8.33040031e-01],
       [9.84506456e-01, 1.54934388e-02, 1.04758573e-07],
       [9.85139652e-01, 1.48602480e-02, 9.98586785e-08],
       [3.21191064e-04, 1.55089721e-01, 8.44589088e-01],
       [5.34987989e-04, 3.47338711e-01, 6.52126301e-01],
       [4.88858001e-05, 5.37492838e-02, 9.46201830e-01],
       [3.96813394e-02, 9.50444085e-01, 9.87457527e-03],
       [4.15694737e-03, 9.26044045e-01, 6.97990080e-02],
       [7.48753823e-07, 6.21938470e-03, 9.93779867e-01],
       [1.22956593e-03, 4.60673337e-01, 5.38097097e-01],
       [9.91737020e-01, 8.26293460e-03, 4.53058189e-08],
       [9.71476242e-01, 2.85234057e-02, 3.51896089e-07],
       [1.26180223e-02, 9.28312881e-01, 5.90690964e-02],
       [3.87238833e-05, 6.27593197e-02, 9.37201956e-01],
       [5.13265797e-03, 8.56169018e-01, 1.38698324e-01],
       [1.13006770e-03, 5.32391709e-01, 4.66478223e-01],
       [8.07357394e-05, 9.89491379e-02, 9.00970126e-01],
       [7.11821464e-04, 5.79634309e-01, 4.19653869e-01],
       [1.73432072e-02, 7.18718297e-01, 2.63938496e-01],
       [9.60948290e-01, 3.90513892e-02, 3.21097334e-07],
       [4.99021686e-03, 7.91978086e-01, 2.03031697e-01],
       [2.29304511e-02, 8.67351014e-01, 1.09718535e-01]])
```

- Let's create a Dataframe of the above numPy array containing probabilities for better understanding.

In [39]:
```
df_probs = pd.DataFrame(data=probs, columns=['Setosa', 'Versicolour', 'V
df_probs.sample(n=10, random_state=54)
```

Out[39]:

	Setosa	Versicolour	Virginica
10	9.851397e-01	0.014860	9.985868e-08
28	4.990217e-03	0.791978	2.030317e-01
12	5.349880e-04	0.347339	6.521263e-01
16	7.487538e-07	0.006219	9.937799e-01
11	3.211911e-04	0.155090	8.445891e-01
24	8.073574e-05	0.098949	9.009701e-01
27	9.609483e-01	0.039051	3.210973e-07
3	9.761448e-05	0.121043	8.788595e-01
22	5.132658e-03	0.856169	1.386983e-01
18	9.917370e-01	0.008263	4.530582e-08

In [36]:
```
#The sum of the probabilities must always be 1. We can see here:
df_probs['sum'] = df_probs.sum(axis=1)
df_probs.sample(n=10, random_state=54)
```

Out[36]:

	Setosa	Versicolour	Virginica	sum
10	9.851397e-01	0.014860	9.985868e-08	1.0
28	4.990217e-03	0.791978	2.030317e-01	1.0
12	5.349880e-04	0.347339	6.521263e-01	1.0
16	7.487538e-07	0.006219	9.937799e-01	1.0
11	3.211911e-04	0.155090	8.445891e-01	1.0
24	8.073574e-05	0.098949	9.009701e-01	1.0
27	9.609483e-01	0.039051	3.210973e-07	1.0
3	9.761448e-05	0.121043	8.788595e-01	1.0
22	5.132658e-03	0.856169	1.386983e-01	1.0
18	9.917370e-01	0.008263	4.530582e-08	1.0

In [37]:
```
y_pred = model.predict(X_test)
df_probs['predicted_class'] = y_pred
df_probs.sample(n=10, random_state=54)
```

Out[37]:

	Setosa	Versicolour	Virginica	sum	predicted_class
10	9.851397e-01	0.014860	9.985868e-08	1.0	0
28	4.990217e-03	0.791978	2.030317e-01	1.0	1
12	5.349880e-04	0.347339	6.521263e-01	1.0	2
16	7.487538e-07	0.006219	9.937799e-01	1.0	2
11	3.211911e-04	0.155090	8.445891e-01	1.0	2
24	8.073574e-05	0.098949	9.009701e-01	1.0	2
27	9.609483e-01	0.039051	3.210973e-07	1.0	0
3	9.761448e-05	0.121043	8.788595e-01	1.0	2
22	5.132658e-03	0.856169	1.386983e-01	1.0	1
18	9.917370e-01	0.008263	4.530582e-08	1.0	0

In [38]:
```
df_probs['actual_class'] = y_test.to_frame().reset_index().drop(columns=
df_probs.sample(n=10, random_state=54)
```

Out[38]:

	Setosa	Versicolour	Virginica	sum	predicted_class	actual_class
10	9.851397e-01	0.014860	9.985868e-08	1.0	0	0
28	4.990217e-03	0.791978	2.030317e-01	1.0	1	1
12	5.349880e-04	0.347339	6.521263e-01	1.0	2	2
16	7.487538e-07	0.006219	9.937799e-01	1.0	2	2
11	3.211911e-04	0.155090	8.445891e-01	1.0	2	2
24	8.073574e-05	0.098949	9.009701e-01	1.0	2	2
27	9.609483e-01	0.039051	3.210973e-07	1.0	0	0
3	9.761448e-05	0.121043	8.788595e-01	1.0	2	2
22	5.132658e-03	0.856169	1.386983e-01	1.0	1	1
18	9.917370e-01	0.008263	4.530582e-08	1.0	0	0

Evaluate the Model using Accuracy Score

In [40]:
```
# Calculate Accuracy score on 20% of the test data
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
print('Accuracy: {:.2f}'.format(accuracy_score(y_test, y_pred)))
```

```
Accuracy: 0.93
```

Cross Validation using `cross_val_score()` :

In [44]:
```
# Calculate Accuracy score on entire dataset using cross_val_score() met
from sklearn.model_selection import cross_val_score

scores_array = cross_val_score(model, X, y, cv=5, scoring='accuracy')
scores_array
```

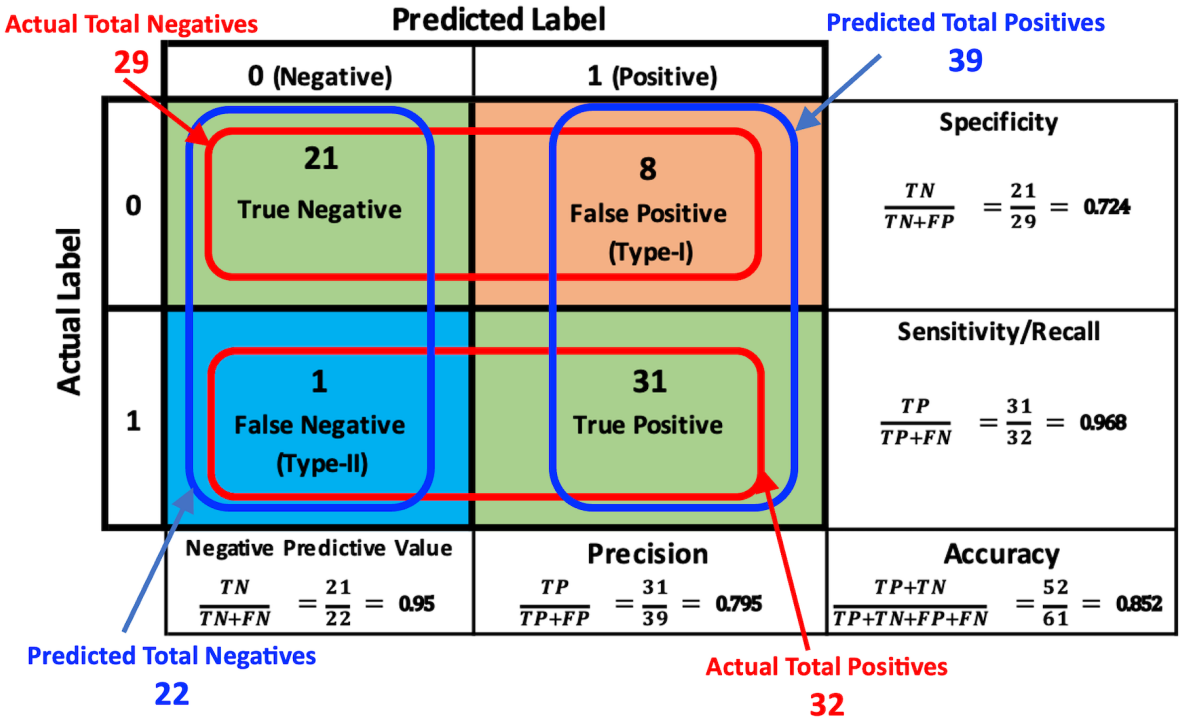
Out[44]:
```
array([0.96666667, 1.        , 0.93333333, 0.96666667, 1.        ])
```

In [46]:
```
print("Accuracy: %0.2f (+/- %0.2f)" % (scores_array.mean(), scores_array
```

```
Accuracy: 0.97 (+/- 0.05)
```

3. Evaluation Metrics for Multi-Class Classification

a. Recap of Confusion Matrix and Evaluation Metrics for Binary Classification



b. Confusion Matrix for Multi-Class Classifier

In [47]:
```
import numpy as np
from sklearn.metrics import confusion_matrix

y_actual = np.array([0,0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,2,0,1,1,0,0,0,0,2,2
y_predicted = np.array([1,0,0,1,0,0,1,2,1,0,0,0,0,0,2,1,1,1,1,0,0,0,1,0,
print("Actual label:    ", y_actual)
print("Predicted label: ", y_predicted)
confusion_matrix(y_actual,y_predicted)
```

```
Actual label:     [0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 2 0 1 1 0 0 0 0 2 2
2 1 1 2]
Predicted label:  [1 0 0 1 0 0 1 2 1 0 0 0 0 0 2 1 1 1 1 0 0 0 1 0 0 0
0 2 2 2]
```

Out[47]:
```
array([[11,  5,  2],
       [ 2,  3,  2],
       [ 3,  1,  1]])
```

		Predicted Label			Total:
		Class A (0)	Class B (1)	Class C (2)	
Actual Label	Class A (0)	11	5	2	18
	Class B (1)	2	3	2	7
	Class C (2)	3	1	1	5
Total:		16	9	5	30

Constructing Confusion Matrix for Class A

Confusion Matrix for a Multi-Class Classifier

		Predicted Label			Total:
		Class A (0)	Class B (1)	Class C (2)	
Actual Label	Class A (0)	11	5	2	18
	Class B (1)	2	3	2	7
	Class C (2)	3	1	1	5
Total:		16	9	5	30

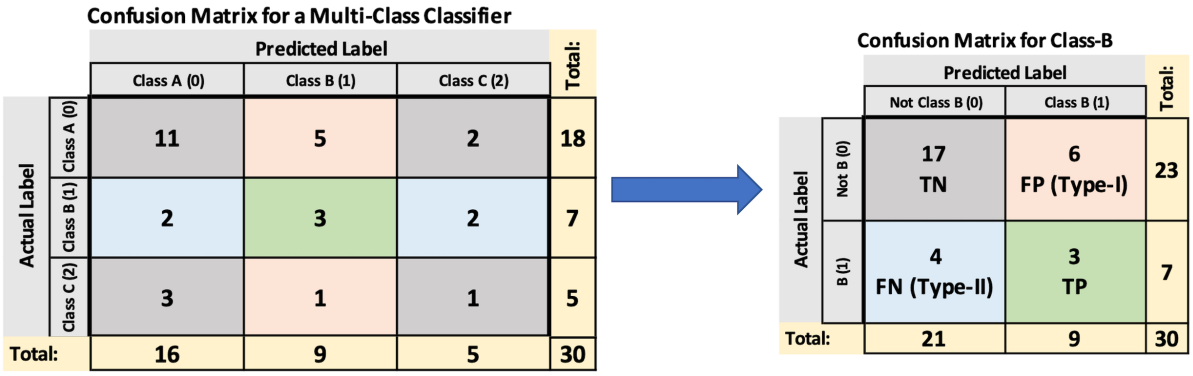
Confusion Matrix for Class-A

		Predicted Label		Total:
		Not Class A (0)	Class A (1)	
Actual Label	Not A (0)	7 TN	5 FP (Type-I)	12
	A (1)	7 FN (Type-II)	11 TP	18
Total:		14	16	30

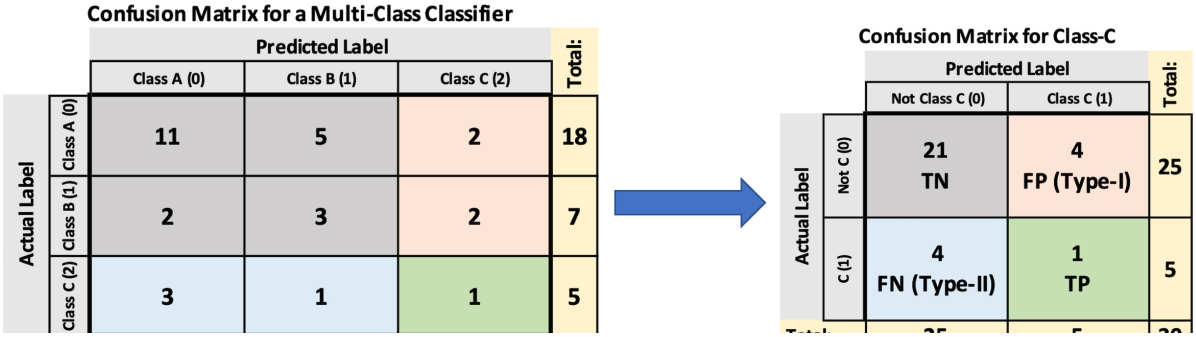
- **TP:** the True-positive value is where the actual value and predicted value are the same.

- **FP:** The False-positive value for a class will be the sum of values of the corresponding `column` except for the TP value.
- **FN:** The False-negative value for a class will be the sum of values of the corresponding `row` except for the TP value.
- **TN:** The True-negative value for a class will be the sum of values of all columns and rows except the values of that class that we are calculating the values for.

Constructing Confusion Matrix for Class B



Constructing Confusion Matrix for Class C



c. Accuracy, Precision, Recall and F-1 Scores for Multi-

Class Problem

		Predicted Label			Total:	Sensitivity/Recall $\frac{TP}{TP+FN}$
		Class A (0)	Class B (1)	Class C (2)		
bel	Class A (0)	11	5	2	18	$\frac{11}{11+5+2} = 0.611$

In [48]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_scor

accuracy = accuracy_score(y_actual, y_predicted)
precision = precision_score(y_actual, y_predicted, average=None)
recall = recall_score(y_actual, y_predicted, average=None)
f1 = f1_score(y_actual, y_predicted, average=None)

print("Accuracy score: ", accuracy)
print("Precision score: ", precision)
print("Recall score: ", recall)
print("F1 score: ", f1)
print("\n Classification Report:\n", classification_report(y_actual, y_p
```

```
Accuracy score:  0.5
Precision score:  [0.6875     0.33333333 0.2       ]
Recall score:  [0.61111111 0.42857143 0.2       ]
F1 score:  [0.64705882 0.375      0.2       ]

 Classification Report:
               precision    recall  f1-score   support

           0       0.69      0.61      0.65        18
           1       0.33      0.43      0.38         7
           2       0.20      0.20      0.20         5

    accuracy                           0.50        30
   macro avg       0.41      0.41      0.41        30
weighted avg       0.52      0.50      0.51        30
```

d. `Macro Average` Precision, Recall and F-1 Scores for Multi-Class Problem

We use **macro averaging** when we want to give an equal weight to each class. If you are working with an **imbalanced dataset** where all classes are equally important, macro average is a good choice as it treats all classes equally.

Macro Average Precision is the arithmetic mean of `Precision` of all classes

$$= \frac{Precision_A + Precision_B + Precision_C}{3} = \frac{0.6875+0.333+0.2}{3} = 0.4069$$

Macro Average Recall is the arithmetic mean of `Recall` of all classes

		Predicted Label			Total:	Sensitivity/Recall $\frac{TP}{TP+FN}$
		Class A (0)	Class B (1)	Class C (2)		
Actual Label	Class A (0)	11	5	2	18	$\frac{11}{11+5+2} = 0.611$
	Class B (1)	2	3	2	7	$\frac{3}{3+2+2} = 0.4286$
	Class C (2)	3	1	1	5	$\frac{1}{1+3+1} = 0.2$
Total:		16	9	5	30	
Precision $\frac{TP}{TP+FP}$		$\frac{11}{11+2+3} = 0.687$	$\frac{3}{3+5+1} = 0.333$	$\frac{1}{1+2+2} = 0.2$		Accuracy $\frac{15}{30} = 0.5$

$$F1(Class\ A) = \frac{2*P*R}{P+R} = \frac{2*0.687*0.611}{0.687+0.611} = 0.647$$

$$F1(Class\ B) = \frac{2*P*R}{P+R} = \frac{2*0.333*0.4286}{0.333+0.4286} = 0.375$$

$$F1(Class\ C) = \frac{2*P*R}{P+R} = \frac{2*0.2*0.2}{0.2+0.2} = 0.2$$

$$= \frac{Recall_A+Recall_B+Recall_C}{3} = \frac{0.611+0.428+0.2}{3} = 0.413$$

Macro Average F1 is the arithmetic mean of `F1` of all classes

$$\frac{F1_A+F1_B+F1_C}{} = \frac{0.647+0.375+0.2}{} = 0.407$$

In [49]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, cla

macro_precision = precision_score(y_actual, y_predicted, average='macro'
macro_recall = recall_score(y_actual, y_predicted, average='macro')
macro_f1 = f1_score(y_actual, y_predicted, average='macro')

print("Macro Precision score: ", macro_precision)
print("Macro Recall score: ", macro_recall)
print("Macro F1 score: ", macro_f1)
print("\n Classification Report:\n", classification_report(y_actual, y_p
```

```
Macro Precision score:  0.4069444444444444
Macro Recall score:  0.4132275132275132
Macro F1 score:  0.40735294117647053

 Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.61      0.65        18
           1       0.33      0.43      0.38         7
           2       0.20      0.20      0.20         5

    accuracy                           0.50        30
   macro avg       0.41      0.41      0.41        30
weighted avg       0.52      0.50      0.51        30
```

e. `Micro Average` Precision, Recall and F-1 Scores for Multi-Class Problem

> We use **micro averaging** when we want to give an equal weight to each instance or prediction.

Micro Average Precision: is the sum of all TPs divided by the sum of all TPs plus the sum of all FPs.

		Predicted Label			Total:
		Class A (0)	Class B (1)	Class C (2)	
Actual Label	Class A (0)	11	5	2	18
	Class B (1)	2	3	2	7
	Class C (2)	3	1	1	5
Total:		16	9	5	30

$$= \frac{TP_A+TP_B+TP_C}{(TP_A+TP_B+TP_C)+(FP_A+FP_B+FP_C)} = \frac{11+3+1}{(11+3+1)+(5+6+4)} = 0.5$$

Micro Average Recall: is the sum of all TPs divided by the sum of all TPs plus the sum of all FNs.

$$= \frac{TP_A+TP_B+TP_C}{(TP_A+TP_B+TP_C)+(FN_A+FN_B+FN_C)} = \frac{11+3+1}{(11+3+1)+(7+4+4)} = 0.5$$

Micro Average F1:

$$F1 = \frac{2PR}{P+R}$$

$$F1 = \frac{TP}{TP+0.5\times(FP+FN)} = \frac{TP_A+TP_B+TP_C}{TP_A+TP_B+TP_C+0.5\times(FP_A+FP_B+FP_C+FN_A+FN_B+FN}$$

- **TP:** is the sum of values of the diagonal, where the actual value and predicted value are the same.($TP_A = 11, TP_B = 3, TP_C = 1$)
- **FP:** is the sum of values of corresponding column except for the TP value. ($FP_A = 5, FP_B = 6, FP_C = 4$)
- **FN:** is the sum of values of corresponding row except for the TP value. ($FN_A = 7, FN_B = 4, FN_C = 4$)
- **TN:** is the sum of values of all columns and rows except the values of the class that we are calculating the TN for. ($TN_A = 7, TN_B = 17, TN_C = 21$)

In [50]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, cla

micro_precision = precision_score(y_actual, y_predicted, average='micro'
micro_recall = recall_score(y_actual, y_predicted, average='micro')
micro_f1 = f1_score(y_actual, y_predicted, average='micro')

print("Micro Precision score: ", micro_precision)
print("Micro Recall score: ", micro_recall)
print("Micro F1 score: ", micro_f1)
print("\n Classification Report:\n", classification_report(y_actual, y_p
```

```
Micro Precision score:  0.5
Micro Recall score:  0.5
Micro F1 score:  0.5

 Classification Report:
               precision    recall  f1-score   support

           0       0.69      0.61      0.65        18
           1       0.33      0.43      0.38         7
           2       0.20      0.20      0.20         5

    accuracy                           0.50        30
   macro avg       0.41      0.41      0.41        30
weighted avg       0.52      0.50      0.51        30
```

In [ ]:

The difference between *macro* and *micro* averaging is that **macro weighs each class equally** whereas **micro weighs each sample equally**. If you have an equal number of samples for each class, then macro and micro will result in the same score.

f. `Weighted Average` Precision, Recall and F-1 Scores for Multi-Class Problem

We use **weighted averaging**, while working with **imbalanced datasets**, when we want to assign greater contribution to classes with more number of samples in the dataset.

Weighted Average Precision is the average of the actual instances of classes times the `Precision` of the classes

$$= \frac{18*0.6875+7*0.333+5*0.2}{18+7+5} = 0.523$$

Weighted Average Recall is the average of the actual instances of classes times the `Recall` of the classes

		Predicted Label			Total:	Sensitivity/Recall $\frac{TP}{TP+FN}$
		Class A (0)	Class B (1)	Class C (2)		
Actual Label	Class A (0)	11	5	2	18	$\frac{11}{11+5+2} = 0.611$
	Class B (1)	2	3	2	7	$\frac{3}{3+2+2} = 0.4286$
	Class C (2)	3	1	1	5	$\frac{1}{1+3+1} = 0.2$
Total:		16	9	5	30	
Precision $\frac{TP}{TP+FP}$		$\frac{11}{11+2+3} = 0.687$	$\frac{3}{3+5+1} = 0.333$	$\frac{1}{1+2+2} = 0.2$		Accuracy $\frac{15}{30} = 0.5$

$$F1(Class\ A) = \frac{2*P*R}{P+R} = \frac{2*0.687*0.611}{0.687+0.611} = 0.647$$

$$F1(Class\ B) = \frac{2*P*R}{P+R} = \frac{2*0.333*0.4286}{0.333+0.4286} = 0.375$$

$$F1(Class\ C) = \frac{2*P*R}{P+R} = \frac{2*0.2*0.2}{0.2+0.2} = 0.2$$

$$= \frac{18*0.611+7*0.428+5*0.2}{18+7+5} = 0.5$$

Weighted Average F1 is the average of the actual instances of classes times the `Recall` of the classes

In [51]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, cla

weighted_precision = precision_score(y_actual, y_predicted, average='wei
weighted_recall = recall_score(y_actual, y_predicted, average='weighted'
weighted_f1 = f1_score(y_actual, y_predicted, average='weighted')

print("Weighted Precision score: ", weighted_precision)
print("Weighted Recall score: ", weighted_recall)
print("Weighted F1 score: ", weighted_f1)
print(classification_report(y_actual, y_predicted))
```

```
Weighted Precision score:  0.523611111111111
Weighted Recall score:  0.5
Weighted F1 score:  0.5090686274509805
              precision    recall  f1-score   support

           0       0.69      0.61      0.65        18
           1       0.33      0.43      0.38         7
           2       0.20      0.20      0.20         5

    accuracy                           0.50        30
   macro avg       0.41      0.41      0.41        30
weighted avg       0.52      0.50      0.51        30
```

g. Other Metrics

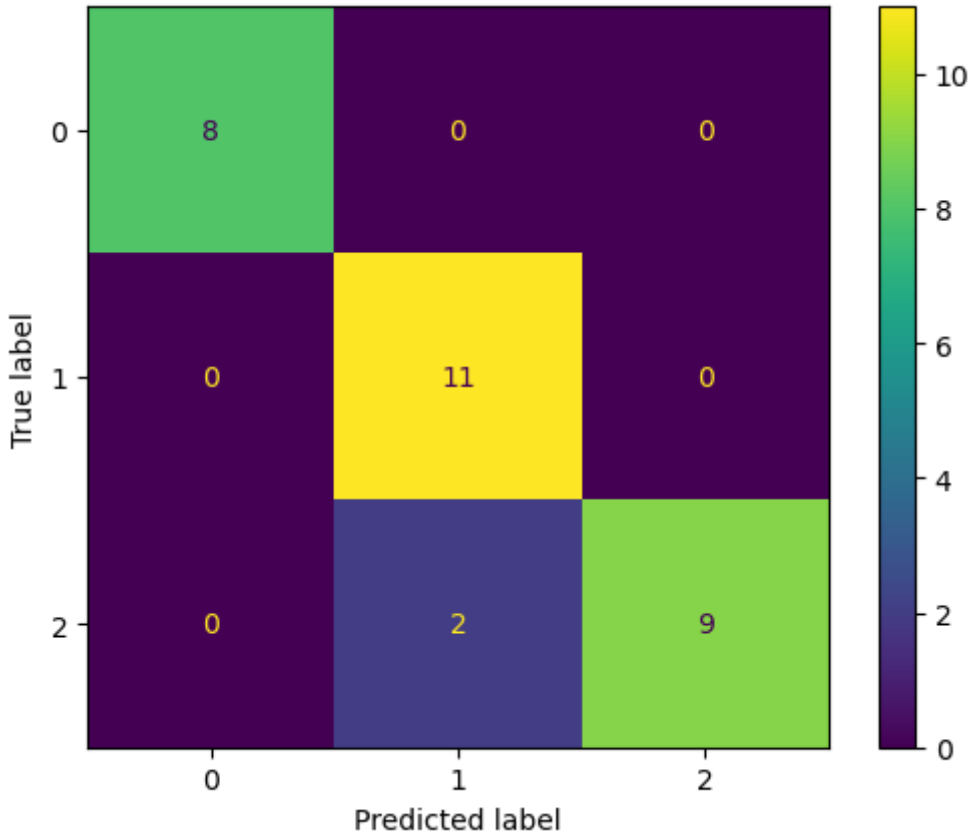
Cohen's Kappa: This is one of the best metrics for evaluating multi-class classifiers on imbalanced datasets. The traditional metrics from the classification report are biased towards the majority class and assumes an identical distribution of the actual and predicted classes. In contrast, Cohen's Kappa Statistic measures the proximity of the predicted classes to the actual classes when compared to a random classification. The output is normalized between 0 and 1 the metrics for each classifier, therefore can be directly compared across the classification task. Generally closer the score is to one, better the classifier.

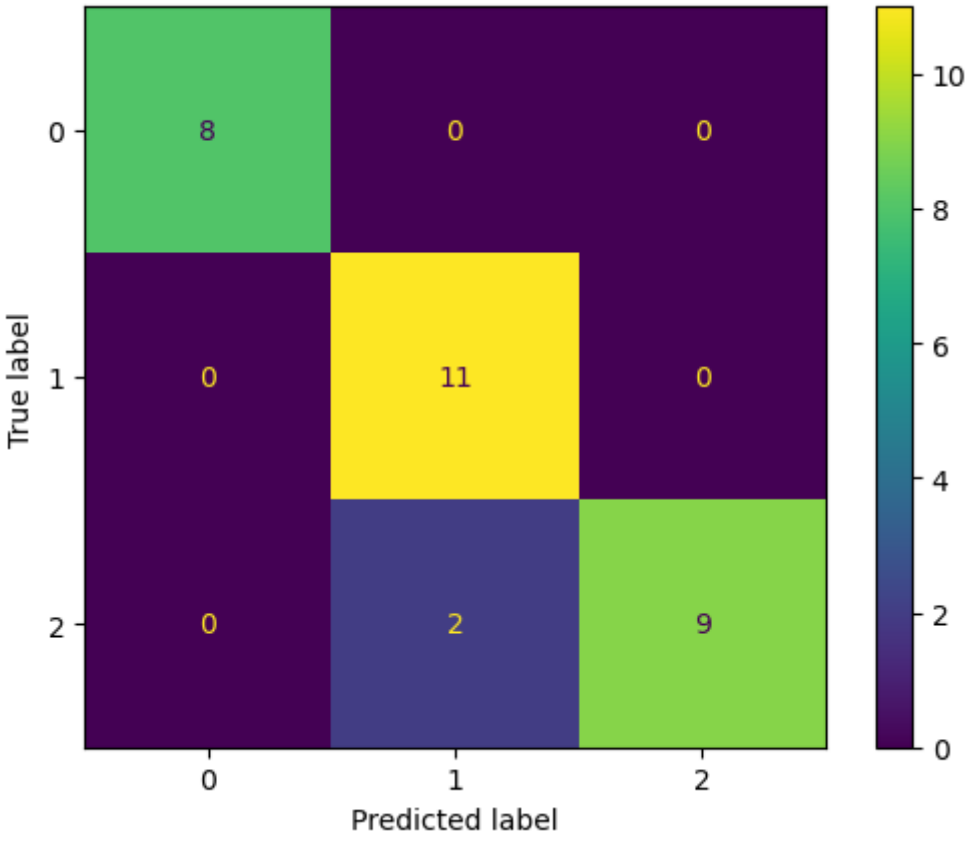
Mathews Correlation Coefficient (MCC): MCC, originally devised for binary classification on unbalanced classes, has been extended to evaluates multiclass classifiers by computing the correlation coefficient between the observed and predicted classifications. A coefficient of +1 represents a perfect prediction, 0 is similar to a random prediction and −1 indicates an inverse prediction.

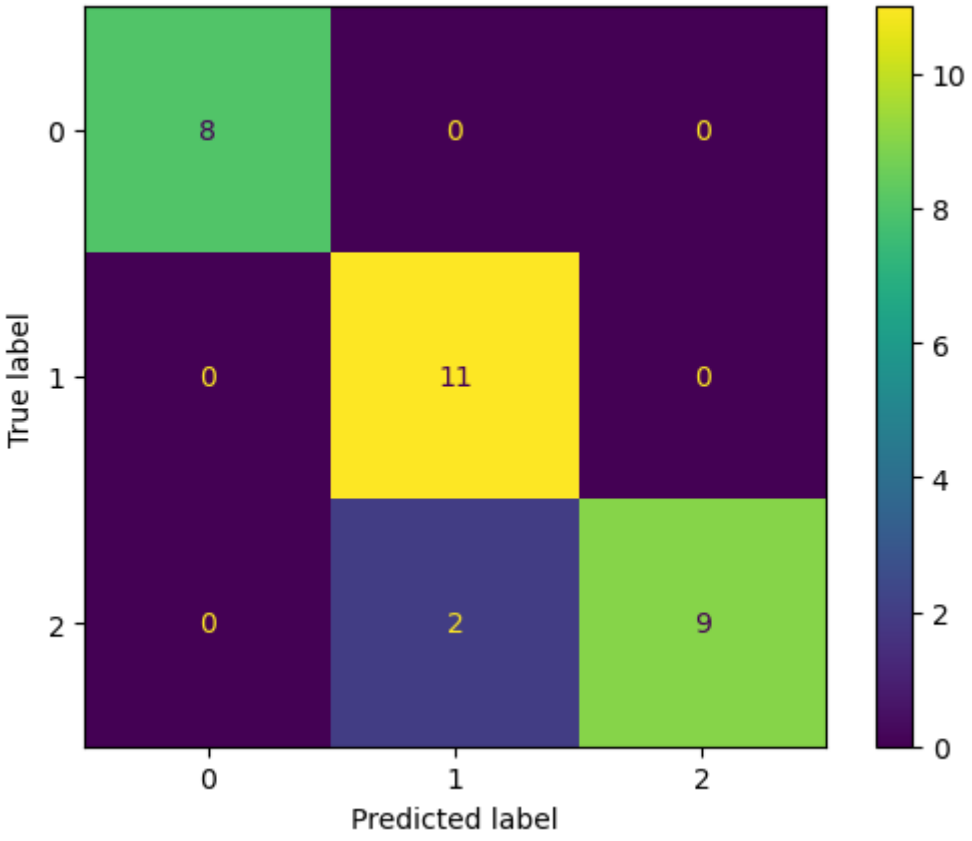
4. Evaluate the Logistic Regression Model (`iris dataset`)

a. Confusion Matrix

In [53]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

disp = ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred))
disp.plot();

ConfusionMatrixDisplay.from_predictions(y_test, y_pred);

ConfusionMatrixDisplay.from_estimator(model, X_test, y_test);
```




8
0
0
0
11
0
0
2
9
0
1
2
True label
Predicted label
10
8
6
4
2
0


8
0
0
0
11
0
0
2
9
0
1
2
True label
Predicted label
10
8
6
4
2
0

b. `Macro Average` Precision, Recall and F-1 Scores

In [54]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

macro_precision = precision_score(y_test, y_pred, average='macro')
macro_recall = recall_score(y_test, y_pred, average='macro')
macro_f1 = f1_score(y_test, y_pred, average='macro')

print("Macro Precision score: ", macro_precision)
print("Macro Recall score: ", macro_recall)
print("Macro F1 score: ", macro_f1)
```

```
Macro Precision score:  0.9487179487179488
Macro Recall score:  0.9393939393939394
Macro F1 score:  0.9388888888888888
```

c. `Micro Average` Precision, Recall and F-1 Scores

In [55]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

micro_precision = precision_score(y_test, y_pred, average='micro')
micro_recall = recall_score(y_test, y_pred, average='micro')
micro_f1 = f1_score(y_test, y_pred, average='micro')

print("Micro Precision score: ", micro_precision)
print("Micro Recall score: ", micro_recall)
print("Micro F1 score: ", micro_f1)
```

```
Micro Precision score:  0.9333333333333333
Micro Recall score:  0.9333333333333333
Micro F1 score:  0.9333333333333333
```

d. `Weighted Average` Precision, Recall and F-1 Scores

In [56]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

weighted_precision = precision_score(y_test, y_pred, average='weighted')
weighted_recall = recall_score(y_test, y_pred, average='weighted')
weighted_f1 = f1_score(y_test, y_pred, average='weighted')

print("Weighted Precision score: ", weighted_precision)
print("Weighted Recall score: ", weighted_recall)
print("Weighted F1 score: ", weighted_f1)
print(classification_report(y_test, y_pred))
```

```
Weighted Precision score:  0.9435897435897436
Weighted Recall score:  0.9333333333333333
Weighted F1 score:  0.9327777777777778
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       0.85      1.00      0.92        11
           2       1.00      0.82      0.90        11

    accuracy                           0.93        30
   macro avg       0.95      0.94      0.94        30
weighted avg       0.94      0.93      0.93        30
```

e. Visualize Decision Boundries

In [57]:
```
df
```

Out[57]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In [58]:
```
df = df[['sepal_length','petal_length','species']]
df
```

Out[58]:

	sepal_length	petal_length	species
0	5.1	1.4	0
1	4.9	1.4	0
2	4.7	1.3	0
3	4.6	1.5	0
4	5.0	1.4	0
...	...	...	...
145	6.7	5.2	2
146	6.3	5.0	2
147	6.5	5.2	2
148	6.2	5.4	2
149	5.9	5.1	2

150 rows × 3 columns

In [59]:
```
X = df.drop('species', axis=1) # df.iloc[:,0:2]
y = df['species']              # df.iloc[:,-1]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
model = LogisticRegression(solver='saga', multi_class='ovr', max_iter=50
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

In [ ]:
```
```

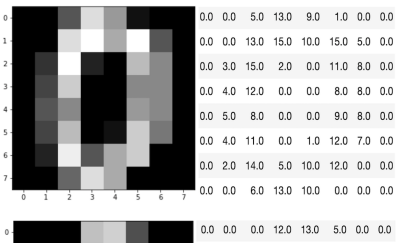
In [ ]:
```
# https://rasbt.github.io/mlxtend/
#import sys
#!{sys.executable} -m pip install mlxtend --upgrade --no-deps
```

In [60]:
```
from mlxtend.plotting import plot_decision_regions

plot_decision_regions(X=X.values, y=y.values, clf=model)

plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.title('Logistic (Softmax) Regression on Iris')
plt.show();
```

```
/opt/anaconda3/lib/python3.9/site-packages/sklearn/base.py:450: UserWa
rning: X does not have valid feature names, but LogisticRegression was
fitted with feature names
  warnings.warn(
```

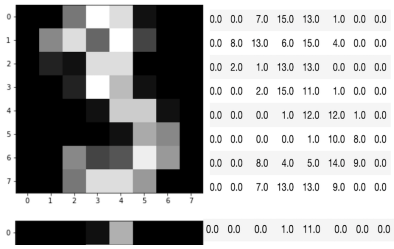
Task To Do (Digit Classification):

Toy Datasets of Scikit-Learn Library: [https://scikit-learn.org/stable/datasets/toy_dataset.html (https://scikit-learn.org/stable/datasets/toy_dataset.html)](https://scikit-learn.org/stable/datasets/toy_dataset.html)



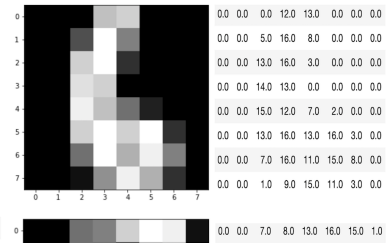
0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0
0.0	0.0	13.0	15.0	10.0	15.0	5.0	0.0
0.0	3.0	15.0	2.0	0.0	11.0	8.0	0.0
0.0	4.0	12.0	0.0	0.0	8.0	8.0	0.0
0.0	5.0	8.0	0.0	0.0	9.0	8.0	0.0
0.0	4.0	11.0	0.0	1.0	12.0	7.0	0.0
0.0	2.0	14.0	5.0	10.0	12.0	0.0	0.0
0.0	0.0	6.0	13.0	10.0	0.0	0.0	0.0

0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0



0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0
0.0	8.0	13.0	6.0	15.0	4.0	0.0	0.0
0.0	2.0	1.0	13.0	13.0	0.0	0.0	0.0
0.0	0.0	2.0	15.0	11.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0	12.0	12.0	1.0	0.0
0.0	0.0	0.0	0.0	1.0	10.0	8.0	0.0
0.0	0.0	8.0	4.0	5.0	14.0	9.0	0.0
0.0	0.0	7.0	13.0	13.0	9.0	0.0	0.0

0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0



0.0	0.0	0.0	12.0	13.0	0.0	0.0	0.0
0.0	0.0	5.0	16.0	8.0	0.0	0.0	0.0
0.0	0.0	13.0	16.0	3.0	0.0	0.0	0.0
0.0	0.0	14.0	13.0	0.0	0.0	0.0	0.0
0.0	0.0	15.0	12.0	7.0	2.0	0.0	0.0
0.0	0.0	13.0	16.0	13.0	16.0	3.0	0.0
0.0	0.0	7.0	16.0	11.0	15.0	8.0	0.0
0.0	0.0	1.0	9.0	15.0	11.0	3.0	0.0

0.0	0.0	7.0	8.0	13.0	16.0	15.0	1.0