Machine Learning in Python – a step by step tutorial

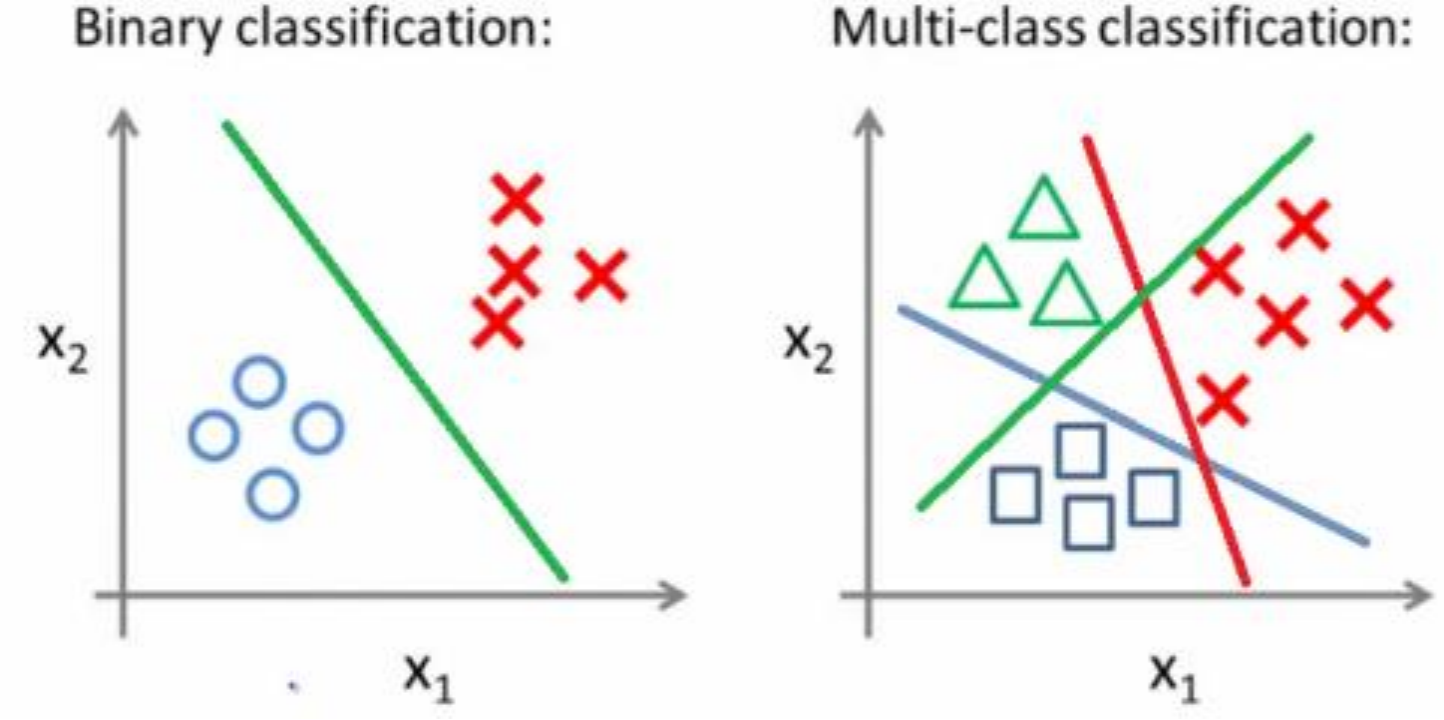
Dr. Junya Michanan

Goals

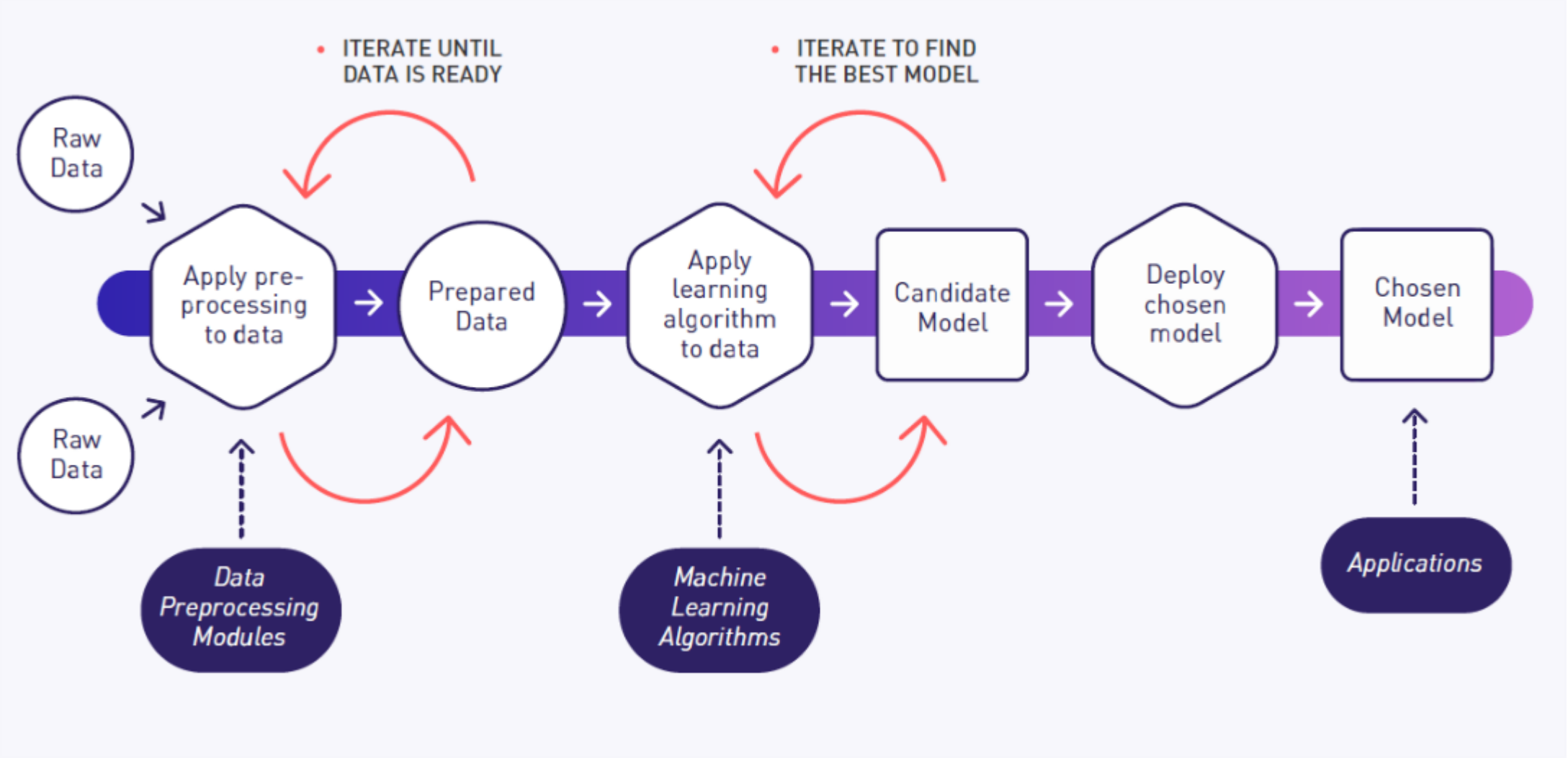
- Get a quick start to Python in machine learning using **ML** packages
- Learn the ML process from a hands-on project
- Learn ML Process
- Learn how to choose a ML algorithm
- Learn how to perform a Supervised Learning Classification Problem

Types of Classification Problems

Binary classification – the problem of classifying instances into two classes, e.g. Yes/No, True/False, Male/Female.
Multi-class classification - the problem of classifying instances into one of three or more classes



The process of a machine learning project



Steps (not linear)

DEFINE PROBLEM

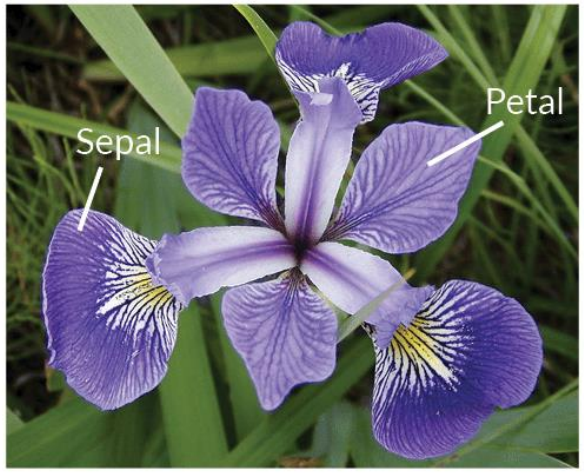
PREPARE DATA

EVALUATE ALGORITHMS

IMPROVE RESULTS

PRESENT RESULTS

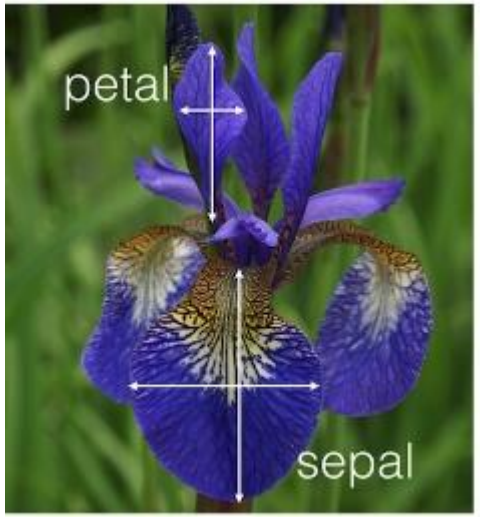
Supervised Learning **Classification** problem – using the Iris flower dataset



Iris Versicolor

Iris Setosa

Iris Virginica



Iris Dataset

- Attributes are numeric so you have to figure out how to load and handle data.
- It is a classification problem, allowing you to practice with perhaps an easier type of supervised learning algorithms.
- It is a mutli-class classification problem (multi-nominal) that may require some specialized handling.
- It only has 4 attribute and 150 rows, meaning it is small and easily fits into memory (and a screen or A4 page).
- All of the numeric attributes are in the same units and the same scale not requiring any special scaling or transforms to get started.

Steps

1. Installing the Python and SciPy platform.
2. Loading the dataset.
3. Summarizing the dataset.
4. Visualizing the dataset.
5. Evaluating some algorithms.
6. Making some predictions.

Step 1: Install packages

There are 5 key libraries that you will need to install. Below is a list of the Python SciPy libraries required for this tutorial:

- Scipy ->
- numpy
- matplotlib
- pandas
- sklearn

```
# Python version
import sys
print('Python: {}'.format(sys.version))
# scipy
import scipy
print('scipy: {}'.format(scipy.__version__))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
# matplotlib
import matplotlib
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

```
C:\AI\CS5902-1-63\tensorflowenv\Scripts\python.exe C:/AI/CS5902-1-63/CheckMLLib.py
Python: 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)]
scipy: 1.4.1
numpy: 1.18.3
matplotlib: 3.2.1
pandas: 1.0.3
sklearn: 0.23.0

Process finished with exit code 0
```

Step 2: Load libraries

```
# Load libraries
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Step 4: Load The Data

- IRIS data is a public data and can be downloaded publicly.
- You now has the iris dataset in a variable.

```
# Load dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = read_csv(url, names=names)
```

Step 5: Data Exploration

In this step we are going to take a look at the data a few different ways:

1. Dimensions of the dataset.
2. Peek at the data itself.
3. Statistical summary of all attributes.
4. Breakdown of the data by the class variable.

Dimensions of Dataset

3.1 Dimensions of Dataset

We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the shape property.

```
# shape
print(dataset.shape)
```

You should see 150 instances and 5 attributes:

```
(150, 5)
```

Peek at the data

It is also always a good idea to actually eyeball your data.

```
# head
print(dataset.head(20))
```

You should see the first 20 rows of the data:

```
    sepal-length  sepal-width  petal-length  petal-width        class
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
10           5.4          3.7           1.5          0.2  Iris-setosa
11           4.8          3.4           1.6          0.2  Iris-setosa
12           4.8          3.0           1.4          0.1  Iris-setosa
13           4.3          3.0           1.1          0.1  Iris-setosa
14           5.8          4.0           1.2          0.2  Iris-setosa
15           5.7          4.4           1.5          0.4  Iris-setosa
16           5.4          3.9           1.3          0.4  Iris-setosa
17           5.1          3.5           1.4          0.3  Iris-setosa
18           5.7          3.8           1.7          0.3  Iris-setosa
19           5.1          3.8           1.5          0.3  Iris-setosa
```

Statistical Summary

```
# descriptions
print(dataset.describe())
```

We can see that all of the numerical values have the same scale (centimeters) and similar ranges between 0 and 8 centimeters.

```
       sepal-length  sepal-width  petal-length  petal-width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.054000      3.758667     1.198667
std        0.828066     0.433594      1.764420     0.763161
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

Class Distribution

Let's now take a look at the number of instances (rows) that belong to each class. We can view this as an absolute count.

```
# class distribution
print(dataset.groupby('class').size())
```

We can see that each class has the same number of instances (50 or 33% of the dataset).

```
class
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
```

Data Visualization

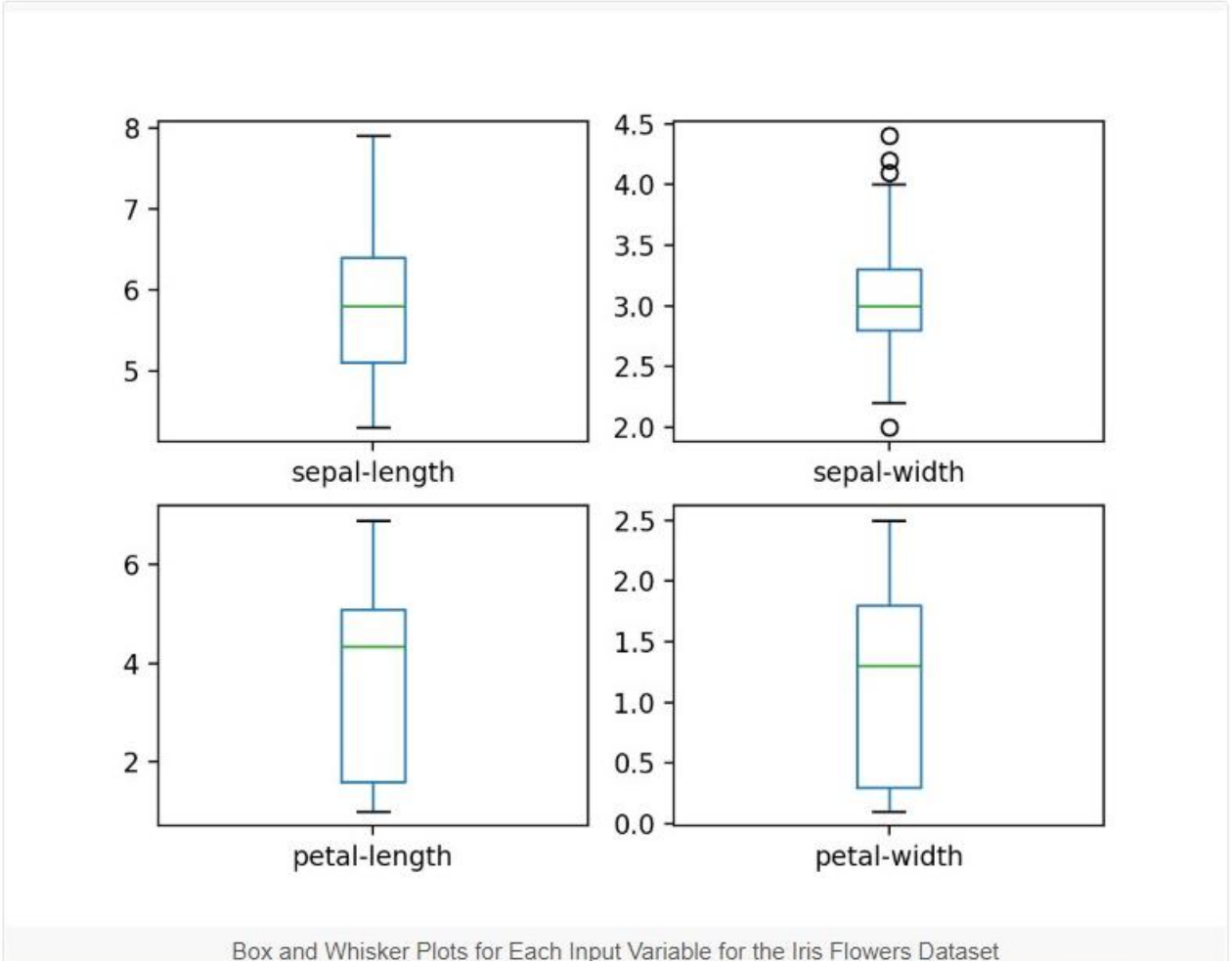
- Examples:
  - Univariate plots to better understand each attribute.
  - Multivariate plots to better understand the relationships between attributes

Univariate plots

- Univariate plots using box and whisker plots

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
pyplot.show()
```

This gives us a much clearer idea of the distribution of the input attributes:

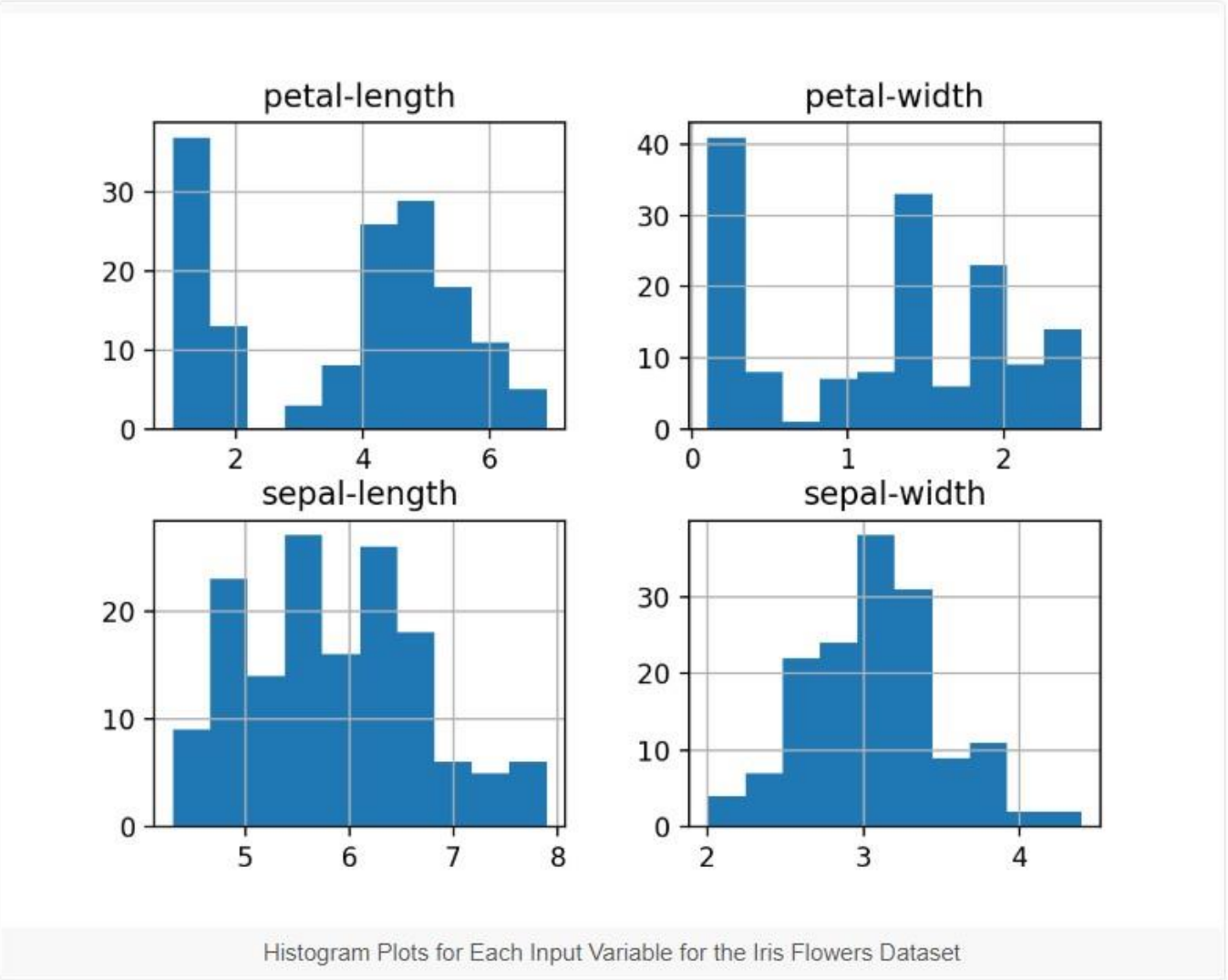


Box and Whisker Plots for Each Input Variable for the Iris Flowers Dataset

Univariate plots

- Gaussian distribution using histograms

```
# histograms
dataset.hist()
pyplot.show()
```

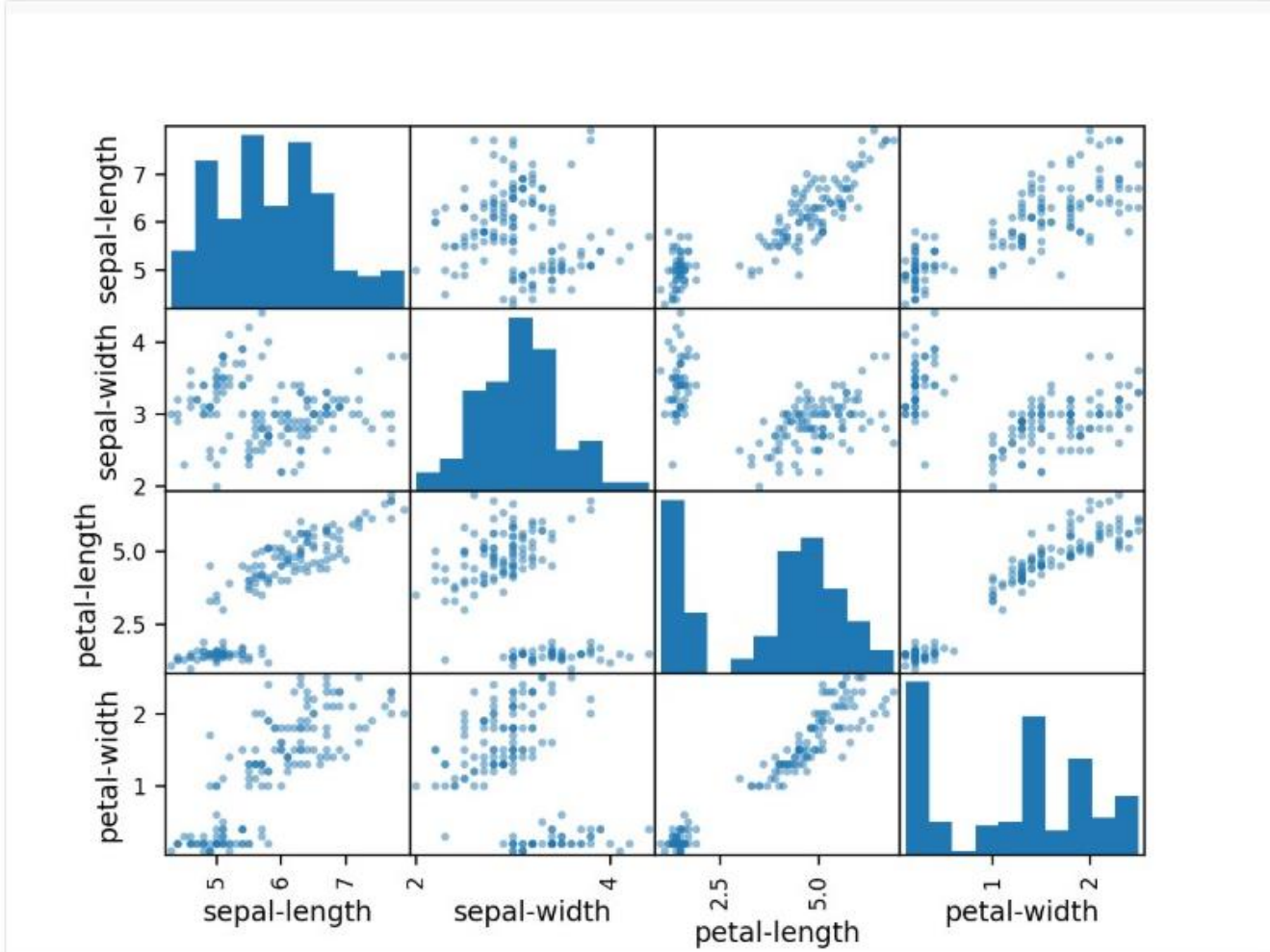


Histogram Plots for Each Input Variable for the Iris Flowers Dataset

Multivariate Plots

```
1 # scatter plot matrix
2 scatter_matrix(dataset)
3 pyplot.show()
```

- Interactions between the variables using Scatter plot matrix



Scatter Matrix Plot for Each Input Variable for the Iris Flowers Dataset

Step 6: Machine Learning Modeling

- Machine Learning Modeling Steps:
  1. Separate out a validation dataset.
  2. Set-up the test harness to use 10-fold cross validation.
  3. Build multiple different models to predict species from flower measurements
  4. Select the best model.

6.1 Create a Validation Dataset

- Validation dataset →some holdback data for validation that the algorithms will not get to see. It will be used to get a second and independent idea of how accurate the best model might be.
- Split the loaded dataset into two, 80% of which we will use to train, evaluate and select among our models, and 20% that we will hold back as a validation dataset

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y, test_size=0.2
```

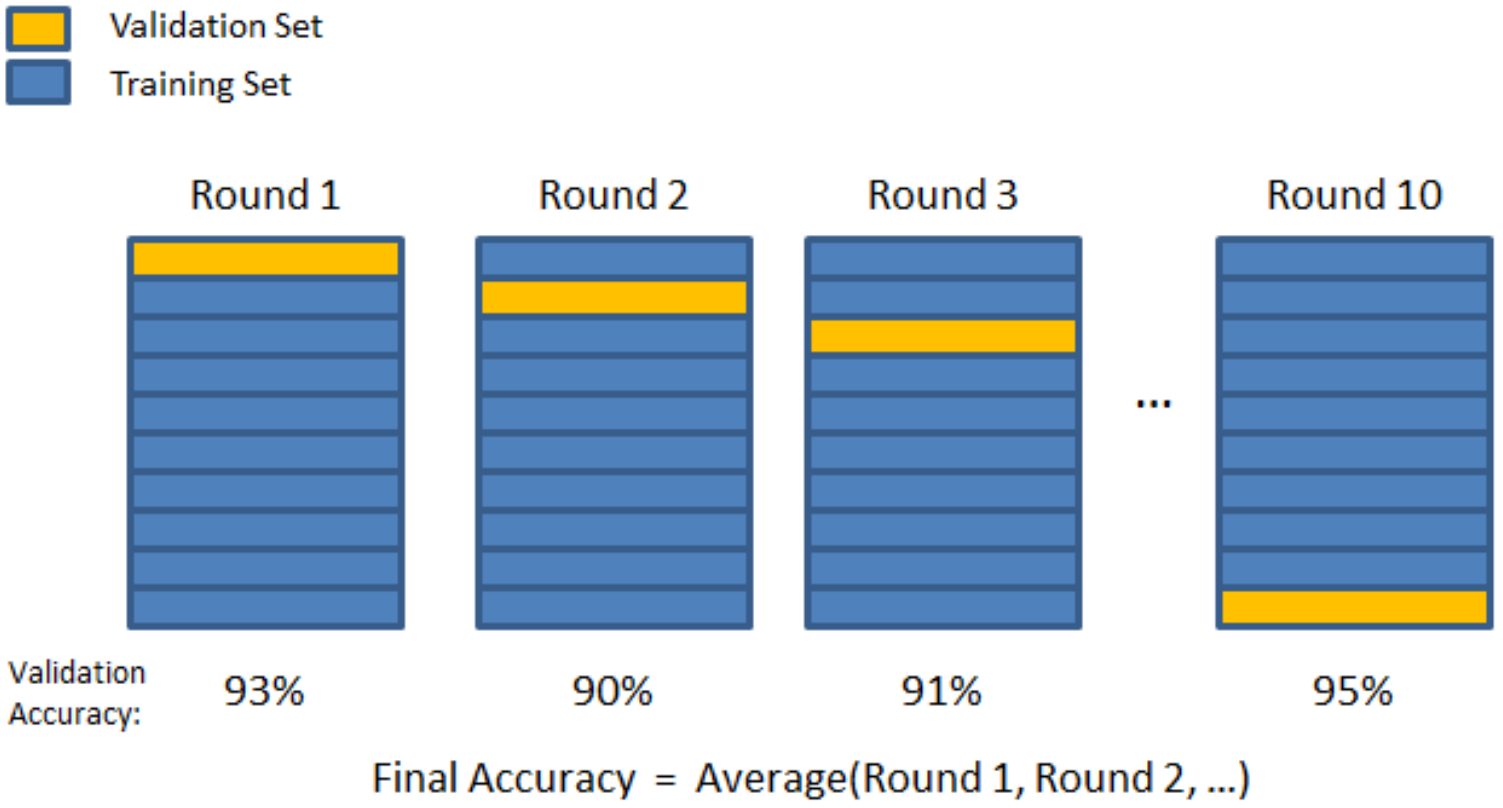
You now have training data in the X_train and Y_train for preparing models and a X_validation and Y_validation sets that we can use later.

6.2 Test Harness

- We will use stratified **10-fold cross validation** to estimate model accuracy.
  - This will split our dataset into 10 parts, train on 9 and test on 1 and repeat for all combinations of train-test splits.
- random seed via the random_state argument to a fixed number to ensure that each algorithm is evaluated on the same splits of the training dataset.
- We are using the metric of '*accuracy*' to evaluate models.
  - A ratio of the number of correctly predicted instances divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the scoring variable when we run build and evaluate each model next.

10-fold Cross Validation

“The Goal is to reduce bias”



6.3 Build Models

Evaluate 6 different algorithms:

1. Logistic Regression (LR)
2. Linear Discriminant Analysis (LDA)
3. K-Nearest Neighbors (KNN).
4. Classification and Regression Trees (CART).
5. Gaussian Naive Bayes (NB).
6. Support Vector Machines (SVM).

```
# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression(solver='liblinear', multi_class='ovr')))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
# evaluate each model in turn
results = []
names = []
for name, model in models:
    kfold = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    print('%s: %f (%f)' % (name, cv_results.mean(), cv_results.std()))
```

6.4 Select Best Model

```
LR: 0.960897 (0.052113)
LDA: 0.973974 (0.040110)
KNN: 0.957191 (0.043263)
CART: 0.957191 (0.043263)
NB: 0.948858 (0.056322)
SVM: 0.983974 (0.032083)
```

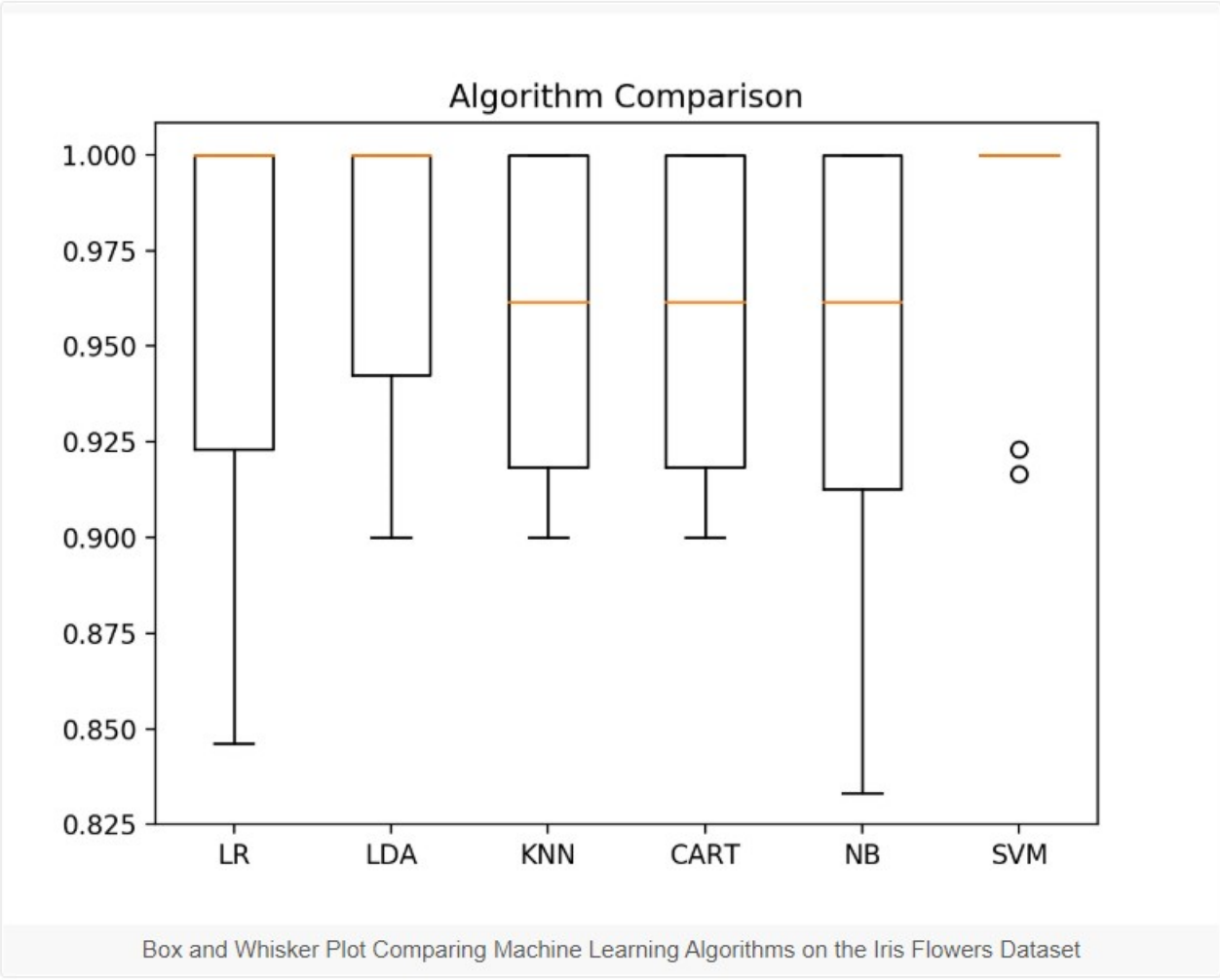
- It looks like Support Vector Machines (SVM) has the largest estimated accuracy score at about 0.98 or 98%

6.4 Select Best Model

```
# Compare Algorithms
pyplot.boxplot(results, labels=names)
pyplot.title('Algorithm Comparison')
pyplot.show()
```

We can see that the box and whisker plots are squashed at the top of the range, with many evaluations achieving 100% accuracy, and some pushing down into the high 80% accuracies.

- We can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (via 10 fold-cross validation).



Box and Whisker Plot Comparing Machine Learning Algorithms on the Iris Flowers Dataset

Step 7: Make Predictions

- **We must choose an algorithm to use to make predictions..**
- SVM was perhaps the most accurate model. We will use this model as our final model.

```
# Make predictions on validation dataset
model = SVC(gamma='auto')
model.fit(X_train, Y_train)
predictions = model.predict(X_validation)
```

Evaluate Predictions

- We can evaluate the predictions by comparing them to the expected results in the validation set, then calculate classification accuracy
- Build a confusion matrix
- Build a classification report.

```
# Evaluate predictions
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

We can see that the accuracy is 0.966 or about 96% on the hold out dataset.

The confusion matrix provides an indication of the three errors made.

Finally, the classification report provides a breakdown of each class by precision, recall, f1-score and support showing excellent results (granted the validation dataset was small).

```
0.9666666666666667
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      0.92      0.96        13
 Iris-virginica       0.86      1.00      0.92         6

       accuracy                           0.97        30
      macro avg       0.95      0.97      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

Confusion Matrix

- A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known.

n=165	**Predicted: NO**	**Predicted: YES**
Actual: NO	50	10
Actual: YES	5	100

example confusion matrix for a binary classifier

- **true positives (TP):** These are cases in which we predicted yes (they have the disease), and they do have the disease.
- **true negatives (TN):** We predicted no, and they don't have the disease.
- **false positives (FP):** We predicted yes, but they don't actually have the disease. (Also known as a "Type I error.")
- **false negatives (FN):** We predicted no, but they actually do have the disease. (Also known as a "Type II error.")

Confusion Matrix

n=165	**Predicted: NO**	**Predicted: YES**	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

List of rates that are often computed from a confusion matrix for a binary classifier:

- **Accuracy:** Overall, how often is the classifier correct?
  - (TP+TN)/total = (100+50)/165 = 0.91
- **Misclassification Rate:** Overall, how often is it wrong?
  - (FP+FN)/total = (10+5)/165 = 0.09
  - equivalent to 1 minus Accuracy
  - also known as "Error Rate"
- **True Positive Rate:** When it's actually yes, how often does it predict yes?
  - TP/actual yes = 100/105 = 0.95
  - also known as "Sensitivity" or "Recall"
- **False Positive Rate:** When it's actually no, how often does it predict yes?
  - FP/actual no = 10/60 = 0.17
- **Specificity:** When it's actually no, how often does it predict no?
  - TN/actual no = 50/60 = 0.83
  - equivalent to 1 minus False Positive Rate
- **Precision:** When it predicts yes, how often is it correct?
  - TP/predicted yes = 100/110 = 0.91
- **Prevalence:** How often does the yes condition actually occur in our sample?
  - actual yes/total = 105/165 = 0.64

Definitions

- Recall (Sensitivity) - Recall is the ratio of correctly predicted positive observations to the all observations in actual class - yes. The question recall answers is: Of all the passengers that truly survived, how many did we label? We have got recall of more than 0.9 which is good for this model as it's above 0.5.

  Recall = TP/TP+FN

- F1 score - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall.

  F1 Score = 2*(Recall * Precision) / (Recall + Precision)

Q & A