IQRA University

CSC 471 Artificial Intelligence

Lab# 11
KNN, along with Data Exploration, Visualization, Accuracy Measures

Objective:

The lab implements the KNN classifier for the Iris dataset. The students are also introduced to some basic techniques related to Data Loading, Data Exploration, Visualization, and implementation of Accuracy Measures after Model training

Name of Student: Faraz Alam

Roll No: 13948 **Sec.** Saturday (12:00pm-14:00pm)

Date of Experiment: 06/12/24

Classifying Iris Species

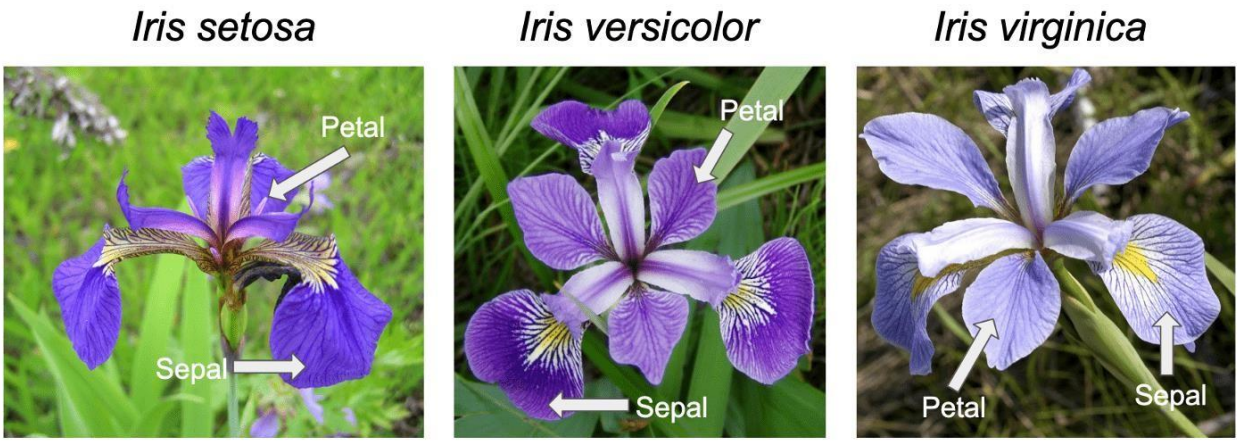
In this section, we will go through a simple machine learning application and create our first model. In the process, we will introduce some core concepts and terms.

Data Loading

The data we will use for this example is the Iris dataset, a classical dataset in machine learning and statistics. It is included in scikit-learn in the datasets module. We can load it by calling the load_iris function:

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
```

The iris object that is returned by load_iris is a Bunch object, which is very similar to a dictionary. It contains keys and values. The value of the key DESCR is a short description of the dataset. The value of the key target_names is an array of strings, containing the species of flower that we want to predict. The value of feature_names is a list of strings, giving the description of each feature. The data itself is contained in the target and data fields. data contains the numeric measurements of sepal length, sepal width, petal length, and petal width in a NumPy array.



```
print(iris_dataset['DESCR'][:193] + "\n...")
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, pre
...
```

```
print("Target names:", iris_dataset['target_names'])
```

```
Target names: ['setosa' 'versicolor' 'virginica']
```

```
print("Feature names:\n", iris_dataset['feature_names'])
```

```
Feature names:
 ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
print("Type of data:", type(iris_dataset['data']))
```

```
Type of data: <class 'numpy.ndarray'>
```

```
print("Shape of data:", iris_dataset['data'].shape)
```

```
Shape of data: (150, 4)
```

We see that the array contains measurements for 150 different flowers. Remember that the individual items are called samples in machine learning, and their properties are called features. The shape of the data array is the number of samples multiplied by the number of features. This is a convention in scikit-learn, and your data will always be assumed to be in this shape. The feature values for the first five samples are provided in the snippet below. From this data, we can see that all of the first five flowers have a petal width of 0.2 cm and that the first flower has the longest sepal, at 5.1 cm. The target array contains the species of each of the flowers that were measured, also as a NumPy array. The species are encoded as integers from 0 to 2.

```
print("First five rows of data:\n", iris_dataset['data'][:5])
```

```
First five rows of data:
 [[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]]
```

```
print("Type of target:", type(iris_dataset['target']))
```

```
Type of target: <class 'numpy.ndarray'>
```

```
print("Shape of target:", iris_dataset['target'].shape)
```

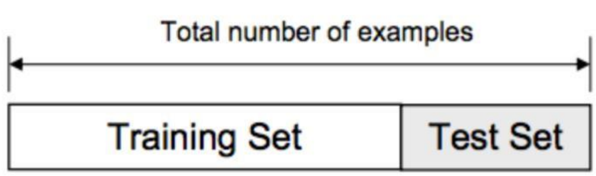
```
Shape of target: (150,)
```

```
print("Target:\n", iris_dataset['target'])
```

```
Target:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 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
 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 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Data Splitting

In machine learning we usually split our data into two subsets: training data and testing data (and sometimes to three: train, validate and test), and fit our model on the train data, in order to make predictions on the test data. Training data and test data are two important concepts in machine learning.



Training Data

The training set contains a known output and the model learns on this data in order to be generalized to other data later on. The observations in the training set form the experience that the algorithm uses to learn.

Test Data

The test dataset (or subset) in order to test our model's prediction on this subset. The test set is a set of observations used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it.

A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples. A program that memorizes its observations may not perform its task well, as it could memorize relations and structures that are noise or coincidence.

Scikit-learn contains a function that shuffles the dataset and splits it for you: the train_test_split function. This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data. The remaining 25% of the data, together with the remaining labels, is declared as the test set. Deciding how much data you want to put into the training and the test set respectively is somewhat arbitrary, but using a test set containing 25% of the data is a good rule of thumb. In scikit-learn, data is usually denoted with a capital X, while labels are denoted by a lowercase y. This is inspired by the standard formulation f(x)=y in mathematics, where x is the input to a function and y is the output. Following more conventions from mathematics, we use a capital X because the data is a two-dimensional array (a matrix) and a lowercase y because the target is a one-dimensional array (a vector). Let us call train_test_split on our data and assign the outputs using this nomenclature:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

```
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
```

```
X_train shape: (112, 4)
y_train shape: (112,)
```

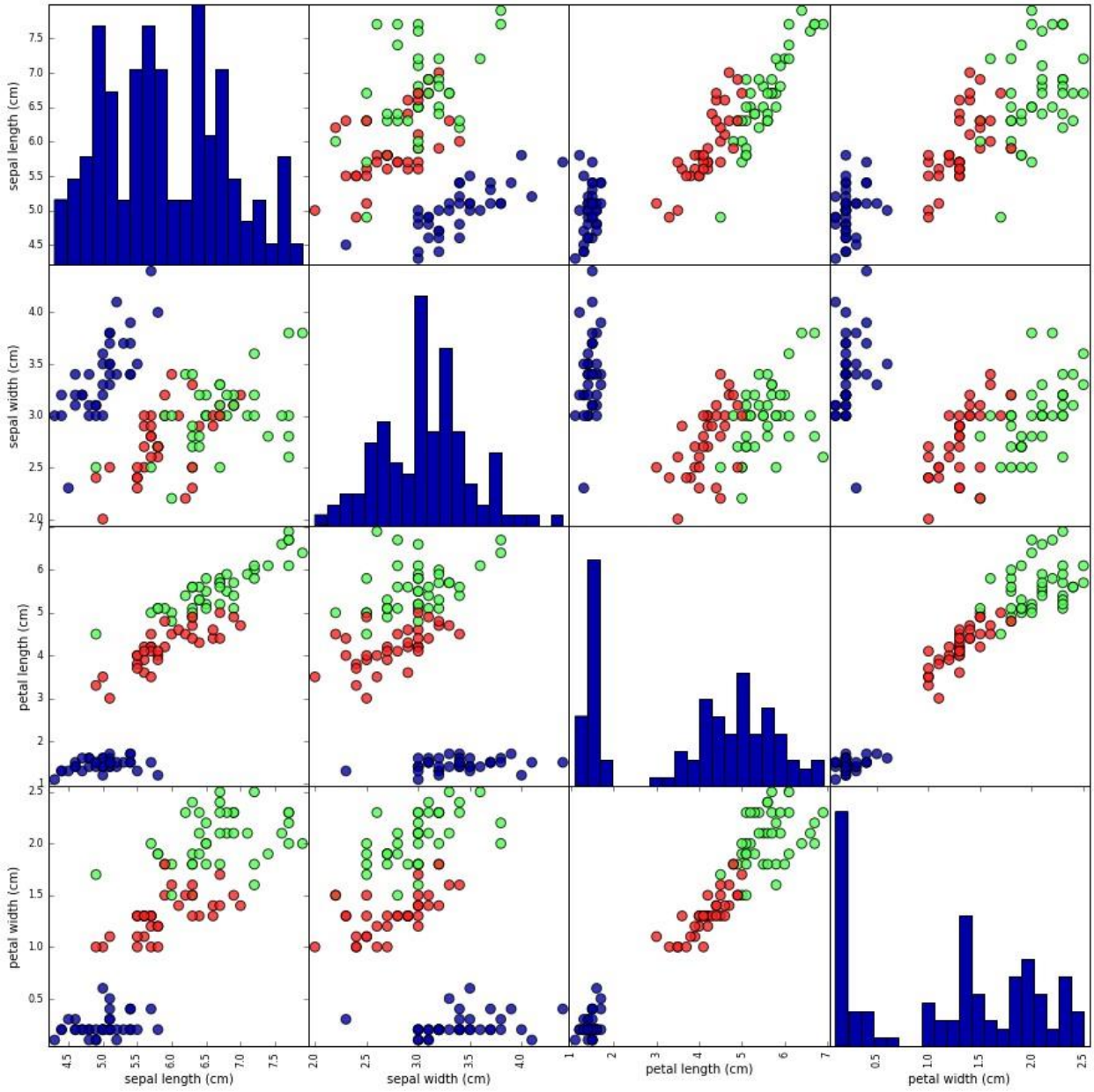
```
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
X_test shape: (38, 4)
y_test shape: (38,)
```

Look at Your Data

Before building a machine learning model it is often a good idea to inspect the data, to see if the task is easily solvable without machine learning, or if the desired information might not be contained in the data. One of the best ways to inspect data is to visualize it. One way to do this is by using a scatter plot. A scatter plot of the data puts one feature along the x-axis and another along the y-axis, and draws a dot for each data point. Unfortunately, computer screens have only two dimensions, which allows us to plot only two (or maybe three) features at a time. It is difficult to plot datasets with more than three features this way. One way around this problem is to do a pair plot, which looks at all possible pairs of features. If you have a small number of features, such as the four we have here, this is quite reasonable. You should keep in mind, however, that a pair plot does not show the interaction of all of features at once, so some interesting aspects of the data may not be revealed when visualizing it this way.

```
# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15),
                           marker='o', hist_kwds={'bins': 20}, s=60,
                           alpha=.8)
```



Model Fitting: k-Nearest Neighbors

The K-nearest neighbors (KNN) algorithm is a type of supervised machine learning algorithms. KNN is extremely easy to implement in its most basic form, and yet performs quite complex classification tasks. It is a lazy learning algorithm since it doesn't have a specialized training phase. Rather, it uses all of the data for training while classifying a new data point or instance. KNN is a non-parametric learning algorithm, which means that it doesn't assume anything about the

underlying data. This is an extremely useful feature since most of the real-world data doesn't really follow any theoretical assumption e.g. linear-separability, uniform distribution, etc.

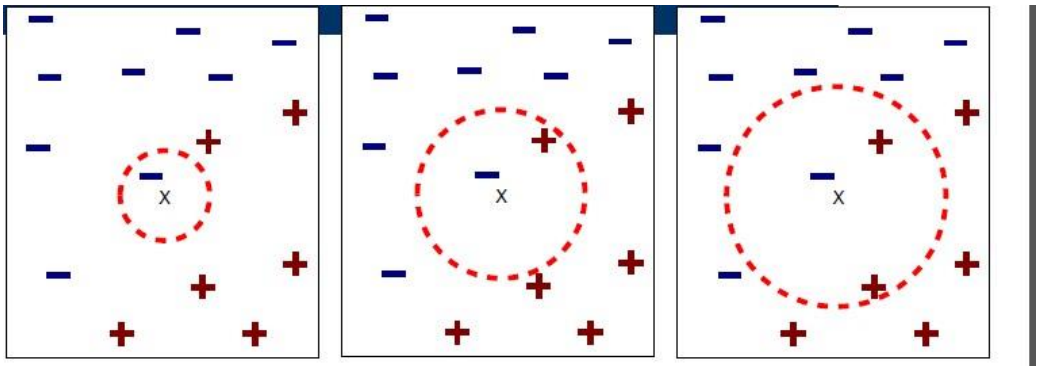
The intuition behind the KNN algorithm is one of the simplest of all the supervised machine learning algorithms. It simply calculates the distance of a new data point to all other training data points. The distance can be of any type e.g. Euclidean or Manhattan etc. It then selects the Knearest data points, where K can be any integer. Finally it assigns the data point to the class to which the majority of the K data points belong.

k-NN algorithm requires three things:

- The set of stored records
- Distance Metric to compute distance between records
- The value of *k*, the number of nearest neighbors to retrieve

To classify an unknown record:

- Compute distance to other training records
- Identify *k* nearest neighbors
- Use class labels of nearest neighbors to determine the class label of unknown record (e.g., by taking majority vote)



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

To compute distance between two points:

– Euclidean distance

$$d(p,q) = \sqrt{\sum_i (p_i - q_i)_2} \qquad d(p,q) = \sum_i abs(p_i - q_i)$$

To determine the class from nearest neighbor list:

– take the majority vote of class labels among the k-nearest neighbors

– Weigh the vote according to distance

KNN in Scikit-Learn

All machine learning models in scikit-learn are implemented in their own classes, which are called Estimator classes. The k-nearest neighbors classification algorithm is implemented in the KNeighborsClassifier class in the neighbors module. Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The knn object encapsulates the algorithm that will be used to build the model from the training data, as well the algorithm to make predictions on new data points. It will also hold the information that the algorithm has extracted from the training data. In the case of KNeighborsClassifier, it will just store the training set. To build the model on the training set, we call the fit method of the knn object, which takes as arguments the NumPy array X_train containing the training data and the NumPy array y_train of the corresponding training labels

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

```
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier(n_neighbors=1)
```

Making Predictions

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape:", X_new.shape)
```

```
X_new.shape: (1, 4)
```

```
prediction = knn.predict(X_new)
print("Prediction:", prediction)
print("Predicted target name:",
       iris_dataset['target_names'][prediction])
```

```
Prediction: [0]
Predicted target name: ['setosa']
```

Evaluating the Model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species is for each iris in the test set. Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species). We can measure how well the model works by computing the accuracy, which is the fraction of flowers for which the right species was predicted

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n", y_pred)
```

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2]
```

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

```
Test set score: 0.97
```

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

```
Test set score: 0.97
```

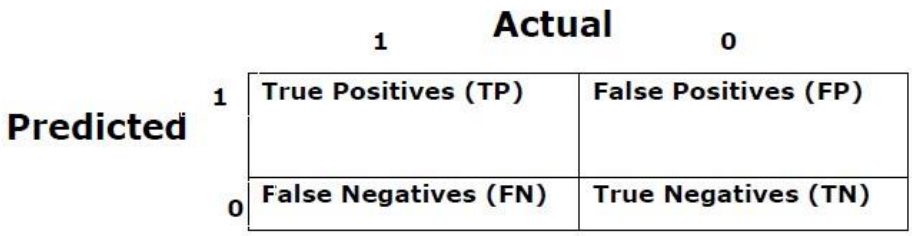
Performance of a Classifier

After implementing a machine learning algorithm, we need to find out how effective the model is. The criteria for measuring the effectiveness may be based upon datasets and metric. For

evaluating different machine learning algorithms, we can use different performance metrics. For example, suppose if a classifier is used to distinguish between images of different objects, we can use the classification performance metrics such as average accuracy, AUC, etc. In one or other sense, the metric we choose to evaluate our machine learning model is very important because the choice of metrics influences how the performance of a machine learning algorithm is measured and compared. Following are some of the metrics:

Confusion Matrix

Basically it is used for classification problem where the output can be of two or more types of classes. It is the easiest way to measure the performance of a classifier. A confusion matrix is basically a table with two dimensions namely "Actual" and "Predicted". Both the dimensions have "True Positives (TP)", "True Negatives (TN)", "False Positives (FP)", "False Negatives (FN)".



Confusion Matrix

In the confusion matrix above, 1 is for positive class and 0 is for negative class.

Following are the terms associated with Confusion matrix:

- **True Positives:** TPs are the cases when the actual class of data point was 1 and the predicted is also 1.
- **True Negatives:** TNs are the cases when the actual class of the data point was 0 and the predicted is also 0.
- **False Positives:** FPs are the cases when the actual class of data point was 0 and the predicted is also 1.
- **False Negatives:** FNs are the cases when the actual class of the data point was 1 and the predicted is also 0.

```
from sklearn.metrics import confusion_matrix
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
```

```
Confusion Matrix:
[[13  0  0]
 [ 0 15  1]
 [ 0  0  9]]
```

Accuracy

The confusion matrix itself is not a performance measure as such but almost all the performance matrices are based on the confusion matrix. One of them is accuracy. In classification problems, it may be defined as the number of correct predictions made by the model over all kinds of predictions made. The formula for calculating the accuracy is as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Precision

It is mostly used in document retrievals. It may be defined as how many of the returned documents are correct. Following is the formula for calculating the precision:

$$Precision = \frac{TP}{TP + FP}$$

Recall

It may be defined as how many of the positives do the model return. Following is the formula for calculating the recall/sensitivity of the model:

$$Recall = \frac{TP}{TP + FN}$$

```
from sklearn.metrics import classification_report, accuracy_score
result1 = classification_report(y_test, y_pred)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,y_pred)
print("Accuracy:",result2)
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       1.00      0.94      0.97        16
           2       0.90      1.00      0.95         9

    accuracy                           0.97        38
   macro avg       0.97      0.98      0.97        38
weighted avg       0.98      0.97      0.97        38

Accuracy: 0.9736842105263158
```

Summary

This lab introduced you to the basics of supervised machine learning. The objective was to practice the loading of a dataset within the scikit-learn library, visualize it and then fit a supervised learning algorithm to make future predictions.

Student Exercise

Task 1

During the data visualization phase, we constructed paired scatter plots among different attributes of the iris dataset. Construct a single three dimensional scatter plot among any three attributes of the iris dataset. Also construct a Correlation Matrix heatmap among the four attributes.

Task 2

Explore at least 2 more classification datasets present within scikit-learn. Implement the explored KNN classification mechanism, along with the data visualization techniques discussed on the 2 datasets of your choice.

Task3

Among the popular data sharing formats Excel file (xlsx), Comma Separated Values (CSV), and data file (.data) are three popular formats. You are provided with one dataset for each of these formats. Find a way to upload these datasets and get them into scikit-learn for exploration. Next perform some basic data visualization and try to fit in a KNN model for each of these classification problems. Evaluate your model's performance through the 80/20 training testing split.

(Note: you might have to use other libraries introduced at the start of the lab apart from scikit-learn).

Task 4

Identify at least 3 famous data repositories where datasets for different machine learning exercises are available. Pick any one classification dataset from among these repositories and fit in a KNN classifier on your dataset. Report the accuracy of the model for different values of N (Neighbors).