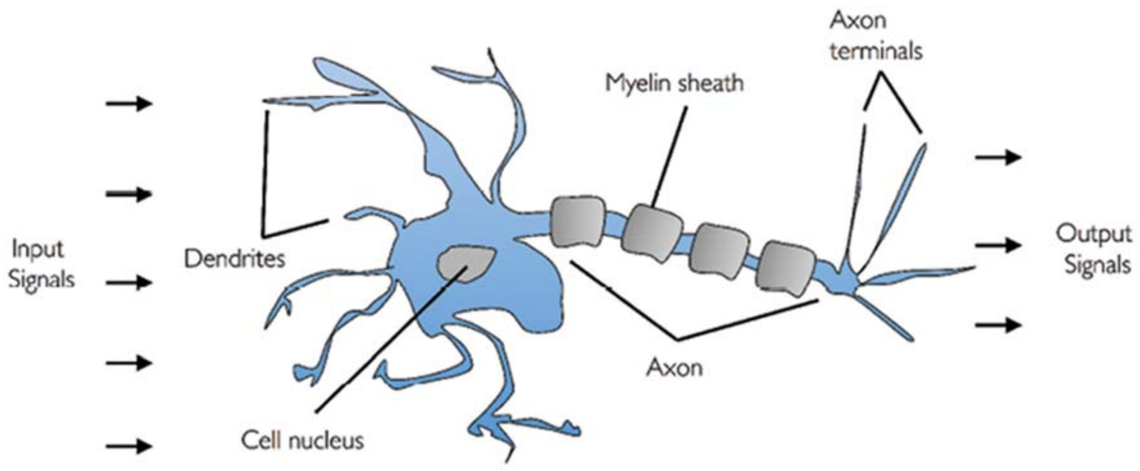
Chapter 6 Machine Learning Algorithms

6.1 Artificial neurons and perceptron learning rule

In this chapter, we will discuss the perceptron and other machine learning algorithms. To start with, we need to review the beginnings of machine learning. Trying to understand how the biological brain works in order to design artificial intelligence (AI), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the **McCulloch-Pitts** (**MCP**) **neuron**, in 1943. Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:



McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

(*A Logical Calculus of the Ideas Immanent in Nervous Activity*, *W. S. McCulloch* and *W. Pitts*, *Bulletin of Mathematical Biophysics*, 5(4): 115-133, *1943*)

Only a few years later, Frank Rosenblatt published the first concept of the **perceptron learning rule** based on the MCP neuron model. With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not. In the context of supervised learning and classification, such an algorithm could then be used to predict whether a new data point belongs to one class or the other.

(*The Perceptron: A Perceiving and Recognizing Automaton*, *F. Rosenblatt*, *Cornell Aeronautical Laboratory*, *1957*)

More formally, we can put the idea behind **artificial neurons** into the context of a binary classification task where we refer to our two classes as 1 (positive class) and –1 (negative class) for simplicity. We can then define a decision function $\phi(z)$ that takes a linear combination of certain input values $\vec{x}$ and a corresponding weight vector $\vec{w}$:

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \qquad \vec{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

where z is the net input defined by:

$$z = \vec{w} \cdot \vec{x} = w_1 x_1 + \cdots + w_m x_m$$

If the net input of a particular example, $\boldsymbol{x}^{(i)}$ is greater than a defined threshold θ, we predict class 1, and class -1 otherwise. In the perceptron algorithm, the decision function is a variant of a **unit step function**:

$$\phi(z) = \begin{cases} 1, & z \geq \theta \\ -1, & z < \theta \end{cases}$$

For simplicity, we can also bring the θ to left side, let $w_0 = -\theta, x_0 = 1$ then:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

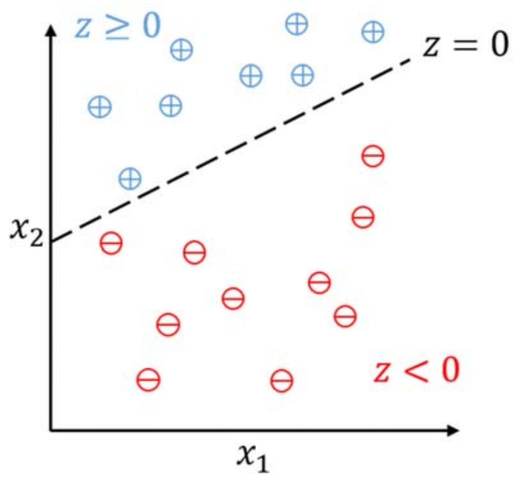
$$\phi(z) = \begin{cases} 1, & z \geq 0 \\ -1, & z < 0 \end{cases}$$

The term $w_0 = -\theta$ is called bias unit.

Let's consider a simple example with only two variables, x_1 and x_2. Suppose after some computations we have the weights $w_0 = -4, w_1 = -1, w_2 = 2$, the net input becomes:

$$z = -4 - x_1 + 2x_2$$

This gives the decision boundary $-4 - x_1 + 2x_2 = 0$, discriminating all data points into two **linearly separable classes**.



Now the problem is, how can we obtain suitable values of the weights, so that the decision line can accurately assign the samples into two classes? The idea behind the perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't. Thus, Rosenblatt's perceptron algorithm can be summarized by the following steps:

(1) Initialize the weights to 0 or small random numbers

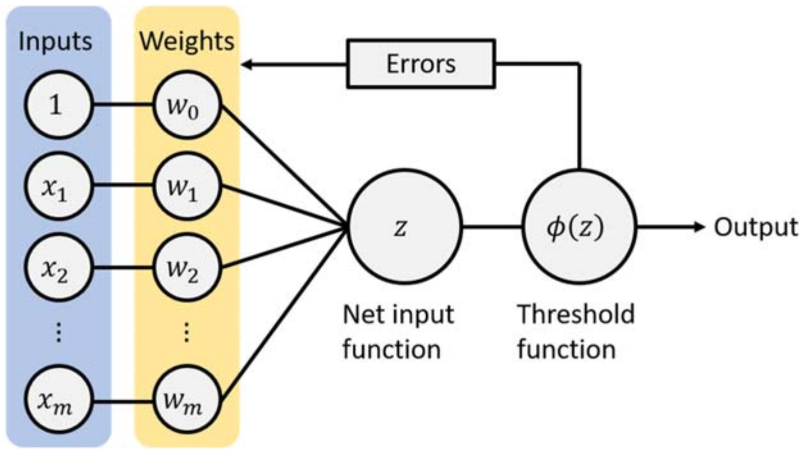
(2) For each training sample $\boldsymbol{x}^{(i)}$, compute the output value $\hat{y}$, then update the weights:

$$w_j := w_j + \Delta w_j$$

where the increment Δw_j for updating w_j is calculated by perceptron learning rule:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

where η is the learning rate (between 0 to 1), $y^{(i)}$ is the true class label and $\hat{y}^{(i)}$ is the predicted class label $\phi(z)$. This algorithm be illustrated by the diagram below.



The interpretation of the perceptron rule is as follows. If the predicted class label $\hat{y}^{(i)}$ is correct, i.e. it equals to the true class label $y^{(i)}$, then in either case of class label 1 or -1 we have $\Delta w_j = 0$, no need to update the weights.

$$\Delta w_j = \eta(1-1)x_j^{(i)} = 0 \qquad \Delta w_j = \eta\left(-1-(-1)\right)x_j^{(i)} = 0$$

If the true class label $y^{(i)} = 1$ and the predicted class label $\hat{y}^{(i)} = -1$ is wrong:

$$\Delta w_j = \eta\left(1-(-1)\right)x_j^{(i)} = 2\eta x_j^{(i)} > 0$$

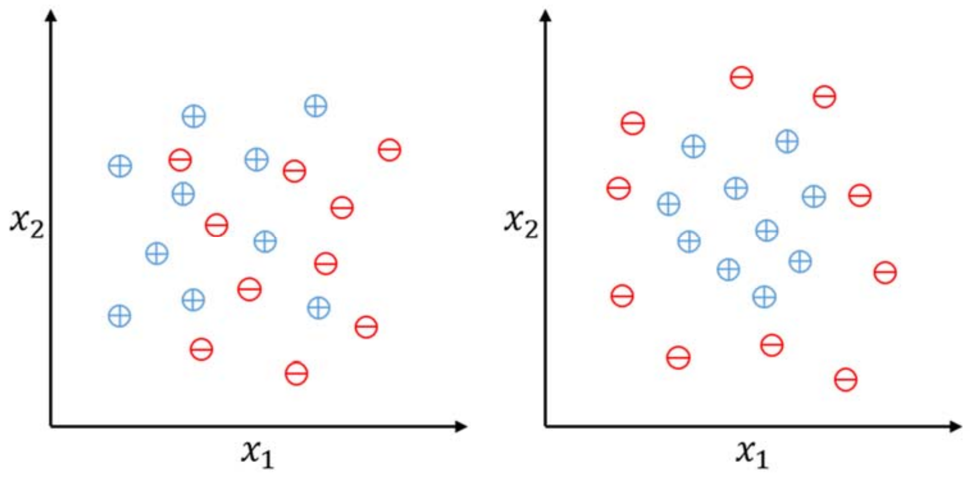
which will push the weight towards the direction of the positive target class.

If the true class label $y^{(i)} = -1$ and the predicted class label $\hat{y}^{(i)} = 1$ is wrong:

$$\Delta w_j = \eta(-1-1)x_j^{(i)} = -2\eta x_j^{(i)} < 0$$

which will push the weight towards the direction of the negative target class.

Notice that the **convergence** of the perceptron model is only guaranteed if the two classes are linearly separable. The following figures show two examples of non-linearly separable points. The one on the left illustrate two classes being highly overlapped and hard to separate. The one on the right is separable but not by a straight line. By observation we might use a circle to separate the space into outer and inner regions for the two classes.



Moreover, if the learning rate η is too large it may also cause the model to diverge. Oppositely, if the learning rate is too small it might take a lot more iterations and running time to obtain a satisfactory result.

6.2 Building a ML model with scikit-learn

The concept of perceptron is simple. However, the algorithm can hardly be computed manually due to the large amount of computations. Instead, we can implement the algorithm in Python with the **scikit-learn** library. It combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms, together with many convenient functions to pre-processed data and to fine-tune and evaluate our models.

To get start with scikit-learn, we will use the Iris dataset as an example. Our target is to use only two features, petal length and petal width to classify the first 100 flowers into two classes. The reason of choosing only two features is for visualization on 2D space. In general you can use any number of features but it would be hard for visualization. For convenience, the class labels have been converted from the species name to 0, 1, 2 stored in "iris_labelled.csv".

First, we read the data from the file and store it as a DataFrame. We further slice the DataFrame into features (X) consisting of column 2 to 3, and target (y) consisting column 4. Notice that we will only need to include the first 100 rows indexed 0 to 99.

```
import pandas as pd
df = pd.read_csv("iris_labelled.csv",header=None)
X = df.loc[0:99, [2,3]]
y = df.loc[0:99, 4]
```

In scikit-learn, there is a method which splits the data of each class label randomly into train data and test data under a given ratio. In our data set, there are 100 rows of 2 class labels, 50 of each. By taking test-size 0.3, there will be 70 rows (35 of each class label) in the train data and 30 rows (15 of each class label) in the test data.

```
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=1, stratify=y)
```

There are many machine learning algorithms available in scikit-learn. We can implement a perceptron model with adjustable parameters, namely the maximum number of iterations `max_iter` and the learning rate `eta`. We can also assign the initial weights to be some small random values so as to avoid 0.

```
from sklearn.linear_model import Perceptron
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train, y_train)
```

The `fit` method under the perceptron model takes in the features and labels of the train data for finding the weights that fits the train data. Now the perceptron model named "ppn" is ready for testifying. To get the final weights after the iterations, we can check for two attributes `coeff_` and `intercept_`. They represents the coefficients $w_1, w_2, \dots, w_m$ and the bias unit w_0 respectively.

```
ppn.coef_
```

```
array([[0.14, 0.28]])
```

```
ppn.intercept_
```

```
array([-0.7])
```

These imply the net input function $z = -0.7 + 0.14x_1 + 0.28x_2$.

Using the model, we can make prediction of a given array of input variables.

```
ppn.predict([[2.7,0.1]])
```

```
array([0], dtype=int64)
```

```
ppn.predict([[4.0,2.1]])
```

```
array([1], dtype=int64)
```

The interpretation is that given a point with $x_1 = 2.7, x_2 = 0.1$ under our model it gives $z = -0.294 < 0$, therefore the model predicts this data point to have class label 0. For another point with $x_1 = 4.0, x_2 = 2.1$, it gives $z = 0.448 > 0$ so it is predicted to have class label 1.

After trained up the model, we shall use it to make prediction on the test data `X_test` resulting in `y_pred` and compare with their actual class labels `y_test`. The number of misclassified samples and the accuracy rate can be displayed as follows:

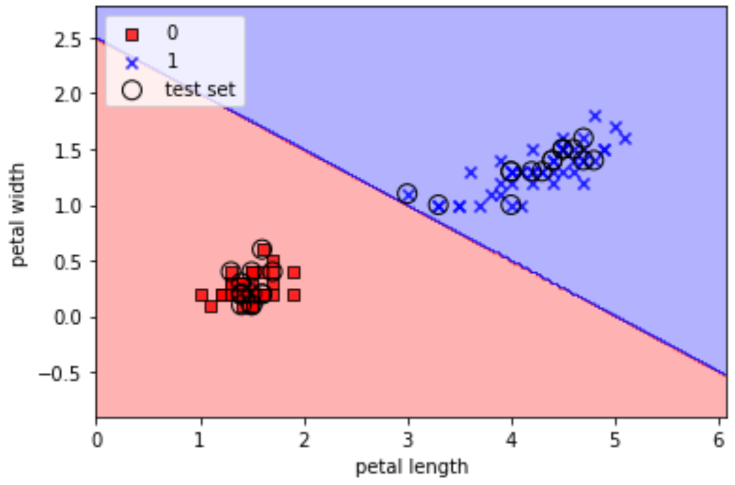
```
y_pred = ppn.predict(X_test)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
from sklearn.metrics import accuracy_score
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

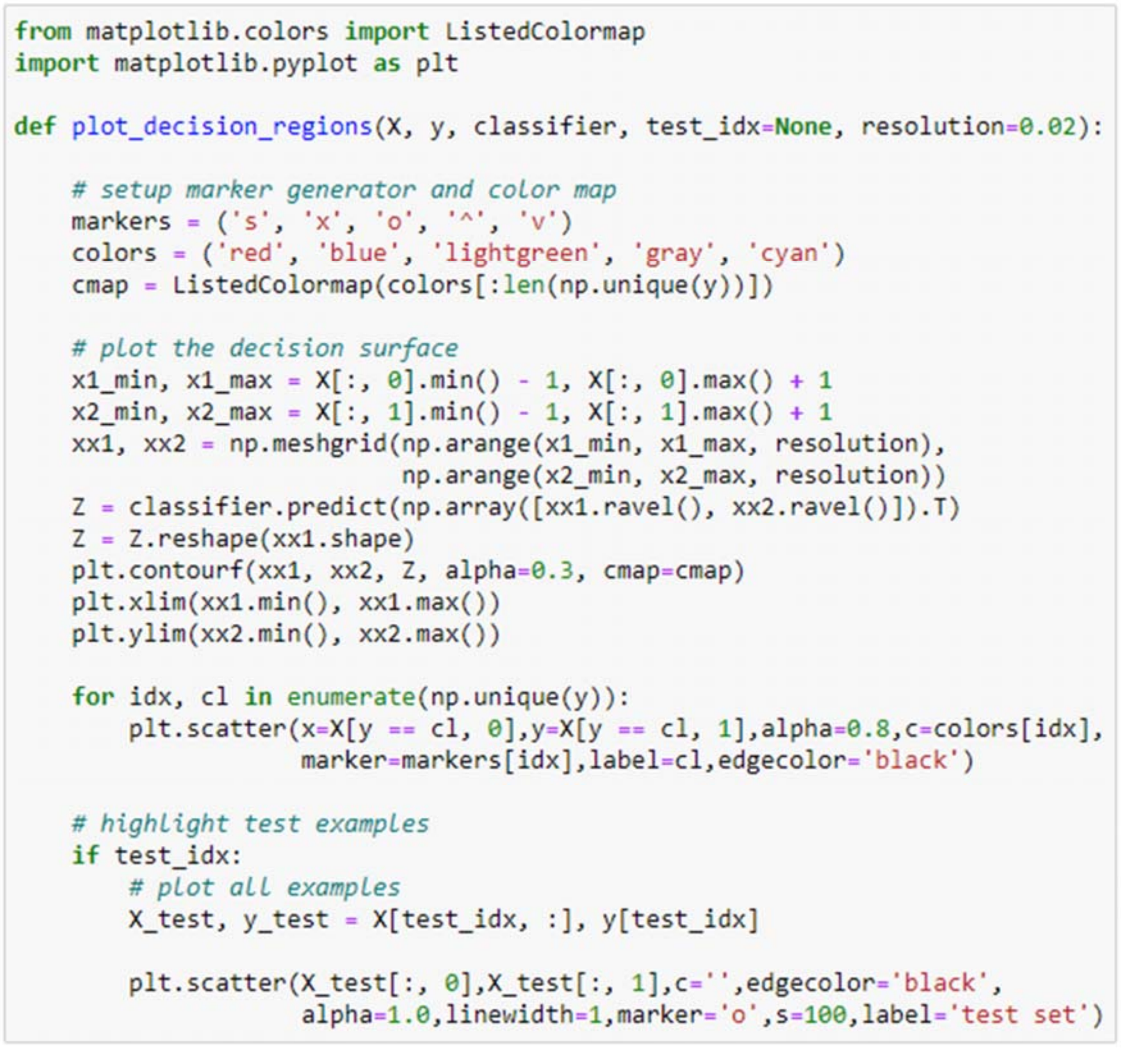
```
Misclassified samples: 0
Accuracy: 1.00
```

To illustrate our model better and compare with other models, we will construct a function in python that displays the data points with labels and decision regions. The coding of this function, however, is made up of more advanced programming techniques and therefore beyond the syllabus of this course. Before using the function to plot the graph, we first combine the train data and test data into a combined data for comparison. In this function, we need to input the features and labels of these data points, together with the name of our model "ppn" and also the index of the test data. Remember that the first 70 data points (indexed 0 to 69) are from the train data. So we should indicate the test data index as `range(70,100)` meaning 70 to 99. As the function is based on matplotlib.pyplot, we can also modify the graph by putting additional information.

```
import numpy as np
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(
    X=X_combined, y=y_combined,classifier=ppn, test_idx=range(70, 100))
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.legend(loc='upper left')
plt.show()
```

The result is as follows, the decision boundary separates the space into two regions coloured red (label 0) and blue (label 1). We can see that the points lie accordingly on the two regions. This suggests our model can perfectly separate the points. The test data is circled for distinguishing from the train data.





```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],y=X[y == cl, 1],alpha=0.8,c=colors[idx],
                    marker=markers[idx],label=cl,edgecolor='black')

    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]

        plt.scatter(X_test[:, 0],X_test[:, 1],c='',edgecolor='black',
                    alpha=1.0,linewidth=1,marker='o',s=100,label='test set')
```

In this example, the two variables representing the petal length and petal width are more or less of the same unit and scale. But in some other application problems, variables might have very diverse unit and scale. This might affect the performance and stability of the machine learning model. In general, we may want to standardize the variables first before running the model. This can be done automatically by the `StandardScalar` method in scikit-learn. Assume data loading and data splitting have been done. The standardized features can be obtained as follows:

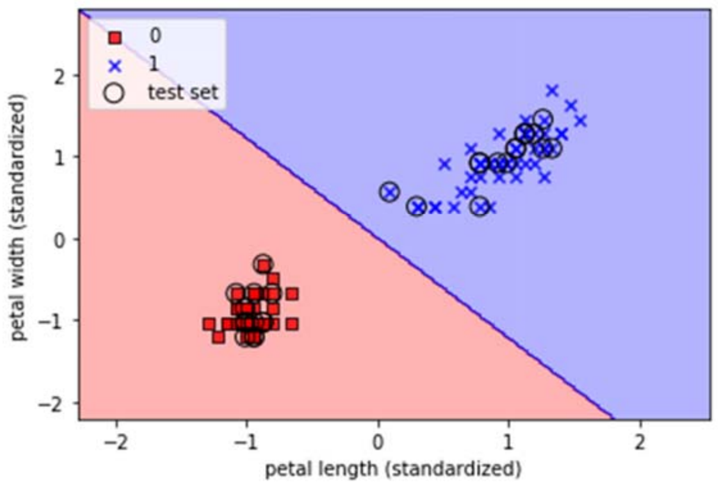
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Next, run the perceptron model "ppn_std" with the standardized train data.

```
from sklearn.linear_model import Perceptron
ppn_std = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn_std.fit(X_train_std, y_train)
```

The result is shown below. The data points are separated clearly into the two regions.

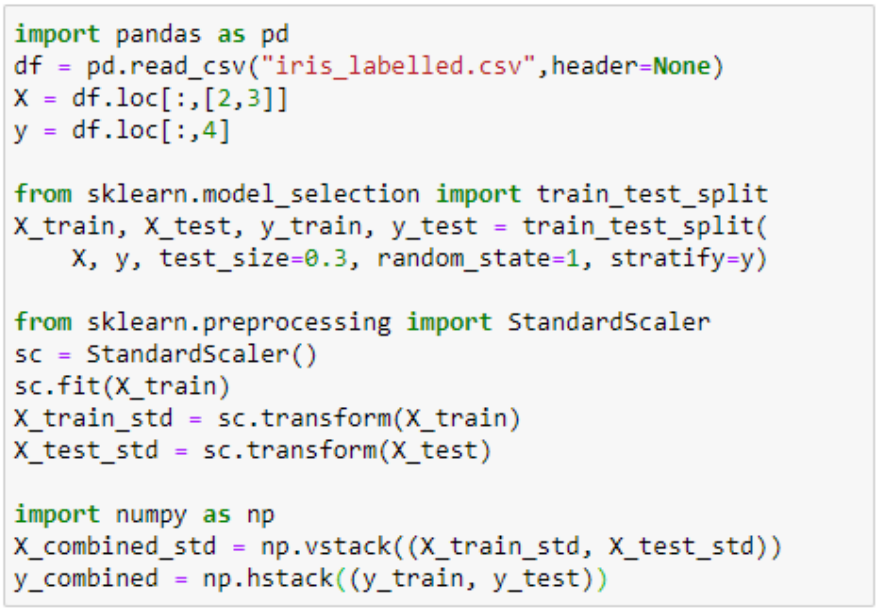
```
import numpy as np
X_combined = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(
    X=X_combined, y=y_combined,classifier=ppn_std, test_idx=range(70, 100))
plt.xlabel('petal length (standardized)')
plt.ylabel('petal width (standardized)')
plt.legend(loc='upper left')
plt.show()
```



6.3 More ML algorithms for classification

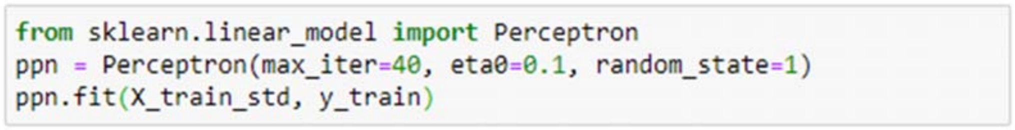
In the previous section, we have trained a perceptron model to classify the iris data of two class labels. The result can be visualized simply as a straight line (decision boundary) dividing the space into two decision regions. However, if the full set of data of all the three class labels is used for classification, the result might not be that obvious. In this section, we will introduce some more algorithms and implement the model on the iris data set (3 class labels) for comparison.

Throughout the examples below, we will use the iris data set with all the 150 rows. The data is then split into train and test data sets and the features are standardized. We will then run different machine learning models. The accuracy of their prediction on the test data and the plot of their decision regions are compared. Notice that in the combined data, the first 105 rows are train data while the last 45 rows (indexed 105 to 149) are test data.



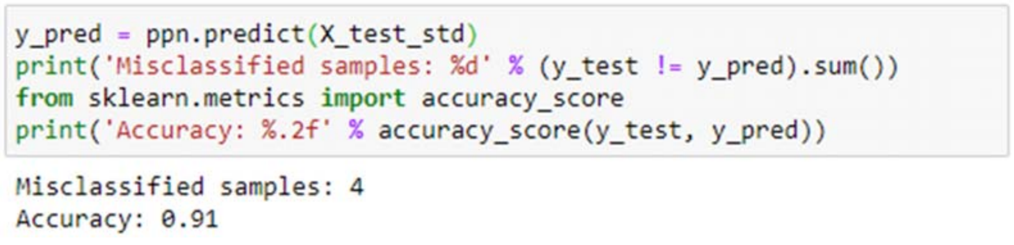
```
import pandas as pd
df = pd.read_csv("iris_labelled.csv",header=None)
X = df.loc[:,[2,3]]
y = df.loc[:,4]

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=1, stratify=y)

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

import numpy as np
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))
```

First, try the perceptron model we have learnt.



```
from sklearn.linear_model import Perceptron
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
```

The accuracy of the test data is evaluated. There are 4 misclassified samples out of 45.

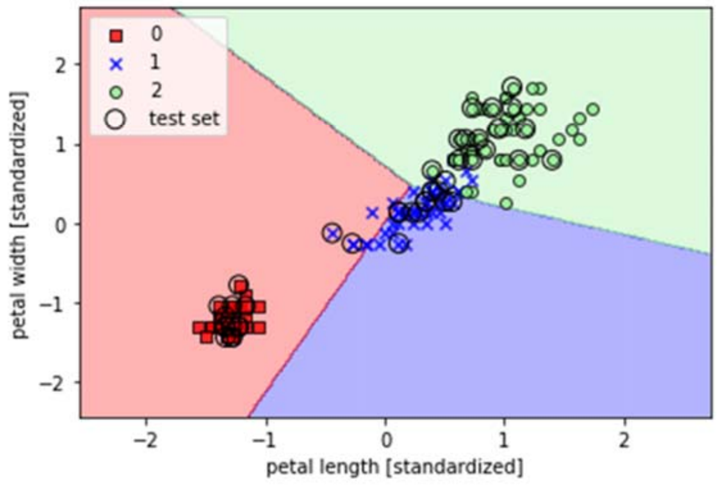


```
y_pred = ppn.predict(X_test_std)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
from sklearn.metrics import accuracy_score
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

```
Misclassified samples: 4
Accuracy: 0.91
```

The decision region is as follows:

```
plot_decision_regions(X=X_combined_std, y=y_combined,
                      classifier=ppn, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.show()
```



We can see that under this method the three iris species cannot be perfectly separated by a linear decision boundary under the perceptron model. There are a number of points near the decision boundary and misclassified. Although the perceptron model is simple and nice, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable.

We will take a look at another more powerful algorithm for classification problems: **logistic regression**. Don't be confused by its name, it is not related to regression. The idea of this algorithm is based on the probability p of a point lying in a desired class. We define the net input z to be the **logit** function of probability p:

$$z = \log \frac{p}{1-p}$$

The decision function ϕ is then defined to be the probability p. We can write down p in terms of z by considering the inverse function:

$$\frac{p}{1-p} = e^z \qquad p = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} = \phi(z)$$

This function is also called the logistic sigmoid function, where the net input z is computed as linear combination of the feature variables:

$$z = w_0 + w_1 x_1 + \cdots + w_m x_m$$

In scikit-learn, we can create an object of `LogisticRegression` as follows. It contains the `fit` method for training.

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, random_state=1, solver='lbfgs', multi_class='ovr')
lr.fit(X_train_std, y_train)
```

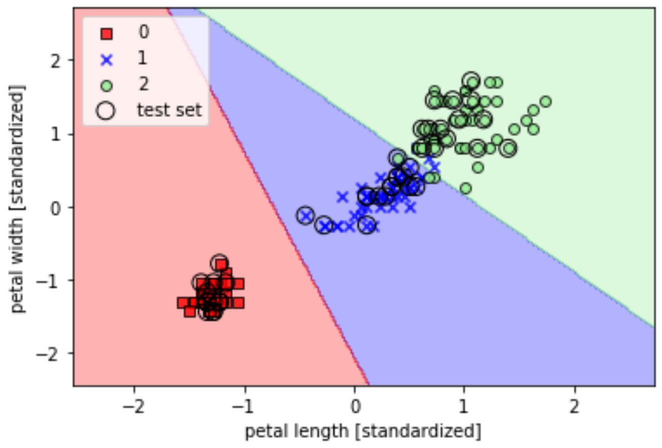
Implement the model to make prediction on the test data, there is only one misclassified sample. The performance seems better than perceptron method.

```
y_pred = lr.predict(X_test_std)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
from sklearn.metrics import accuracy_score
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

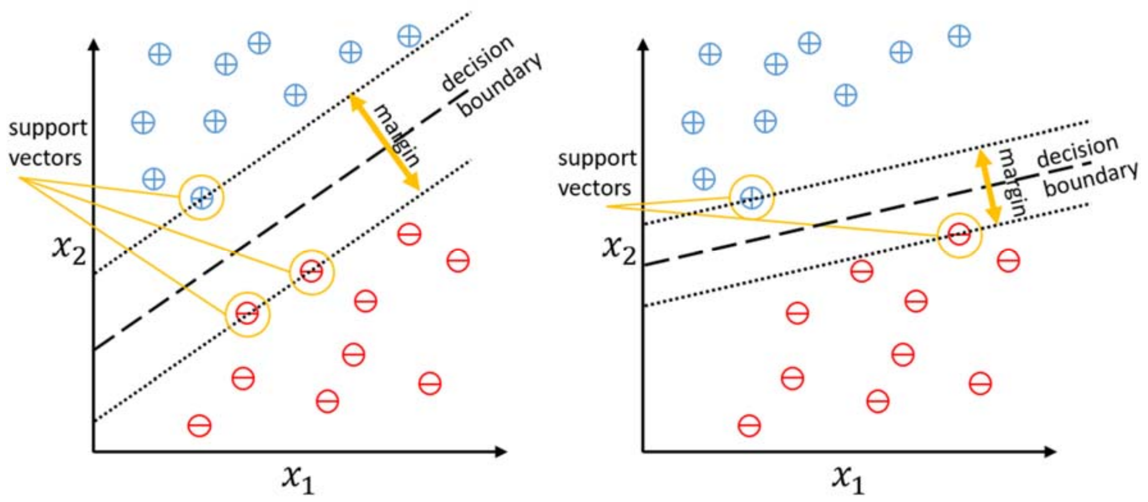
```
Misclassified samples: 1
Accuracy: 0.98
```

The decision regions is plot as follows. The decision boundaries are also straights. Notice that there are fewer points lying near the decision boundary.

```
plot_decision_regions(X_combined_std, y_combined,
                      classifier=lr, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.show()
```



Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs our optimization objective is to maximize the margin. The margin is defined as the distance between the decision boundary and the closest train data points, which are the so-called **support vectors**. This is illustrated in the following figures:



The data points on both figures are the same. Both the two decision boundaries can separate the samples correctly. However, the figure on the left have a wider margin around the boundary compared with the figure on the right. Under SVM, we are looking for the decision boundary that maximize the margin width therefore the one on the left is better.

The algorithm of SVM, however, involve more advanced mathematical knowledge and not to be covered in this course. We will directly run the model with `SVC` in scikit-learn.

```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train_std, y_train)
```

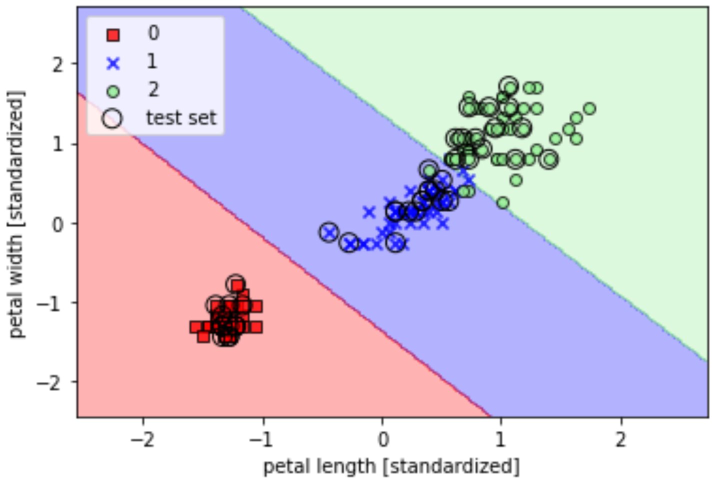
Implement the model to make prediction on the test data. Just like the logistic regression model, there is only one misclassified sample. The performance seems better than perceptron method.

```
y_pred = svm.predict(X_test_std)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
from sklearn.metrics import accuracy_score
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

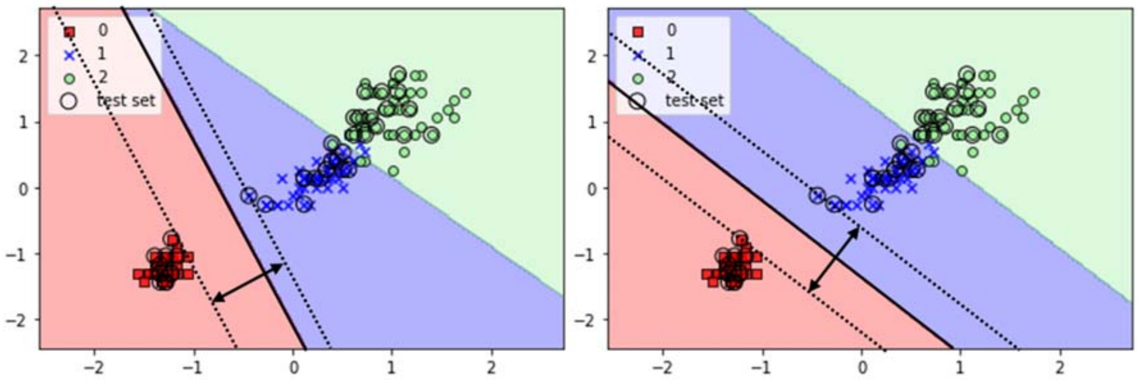
```
Misclassified samples: 1
Accuracy: 0.98
```

The decision boundaries and regions is plot as below. Notice that for class label 1 (blue) and 2 (green), it is not so meaningful to maximize the margin as the points cannot be linearly separated.

```
plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.show()
```



However, if we look at the boundary between label 0 (red) and label 1 (blue), we can see that the margin is maximized with the decision boundary lying in the middle of it (figure on the right). Compare with the decision boundary in the logistic regression model (figure on the left) we can see this difference.

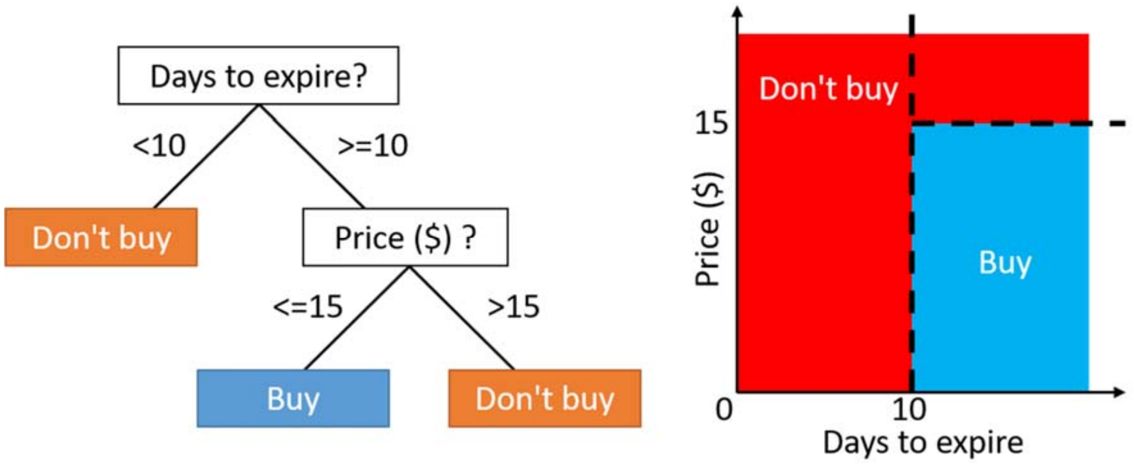


6.4 Decision tree learning

We have introduced some algorithms namely perceptron, logistic regression and support vector machine (SVM). The most common property of them is that their decision boundary is linear. In contrast, their common weakness is that they are not doing well in data that are not linearly separable. There is also a lack of interpretability of how the decision is made.

Another kind of machine learning algorithm, **decision tree** classifiers are attractive models if we care about interpretability. As the name "decision tree" suggests, we can think of this model as breaking down our data by making a decision based on asking a series of questions.

Let's consider a daily life example: should you buy a piece of food or not? Two criteria could be (1) days to expire; (2) price. A person might first look at the best-before tag and exempt those to be expired within 10 days. Even the food can be stored for more than 10 days, price would be another concern. Those more expensive than $15 is not worth to buy. This decision process can be converted to a tree diagram in response to the two questions asked, leading to the leaves (terminal) labelled as either "Don't buy" or "Buy". Based on these criteria, we can draw the decision boundaries and the decision regions. If we regard all the food to be data with two variables (days to expire, price), we can simple plot the point on the graph and make decision according to the region it lies in.



Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples. Although the preceding figure illustrates the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers, like in the Iris dataset. For example, we could simply define a cut-off value along the sepal width feature axis and ask a binary question: "Is the petal width ≥ 2.5 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which is related to the concept of entropy and not to be discussed in this course. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the training examples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to **overfitting**. Thus, we typically want to **prune** the tree by setting a limit for the maximal depth of the tree.

In order to make direct interpretation of the original values, we will not apply standardization on the data. Build the model using `DecisionTreeClassifier` in scikit-learn. Notice that the depth of the tree can be adjusted by the parameter `max_depth`.

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier(criterion='gini',max_depth=4,random_state=1)
dtc.fit(X_train, y_train)
```

Apply the predict method on the unstandardized test data X_test. We can see the accuracy of decision tree classifier is quite high with only one misclassified sample.

```
y_pred = dtc.predict(X_test)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
from sklearn.metrics import accuracy_score
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

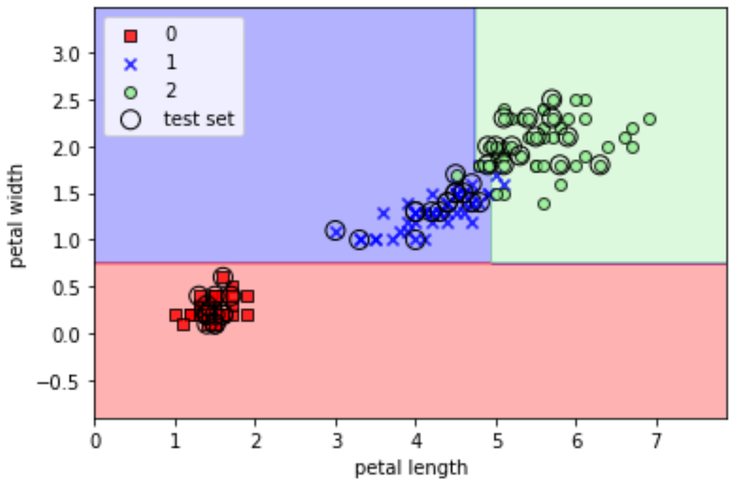
```
Misclassified samples: 1
Accuracy: 0.98
```

Combine the train and test data together for visualization:

```
import numpy as np
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
```

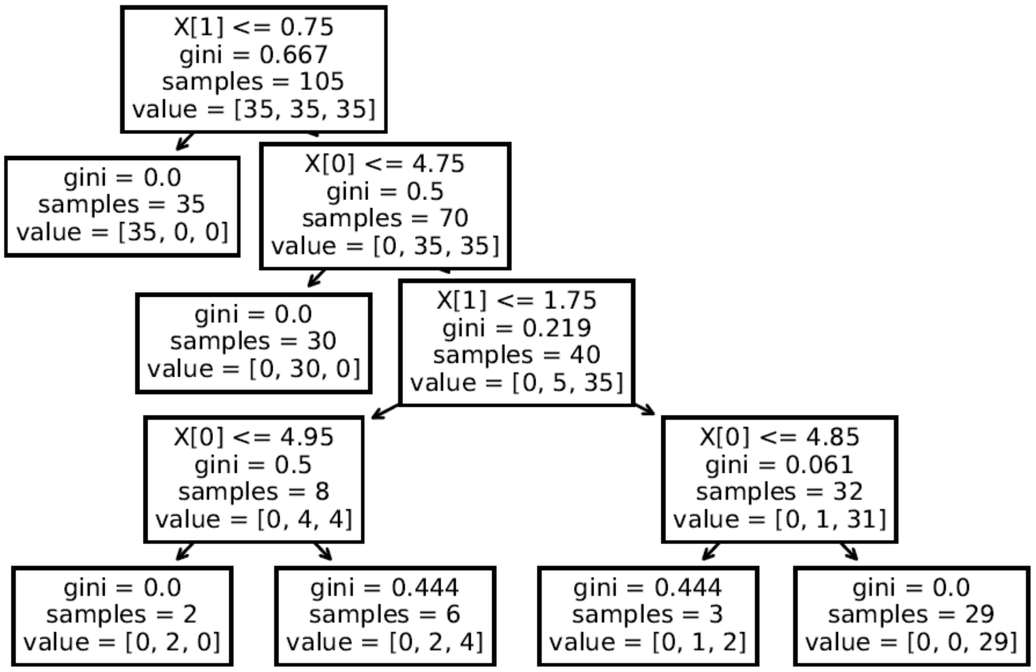
The decision boundaries and regions are plotted on the figure. Notice that the decision boundary between the regions of label 1 (blue) and label 2 (green) looks like a staircase formed by vertical and horizontal line segments. None of the decision boundary in a decision tree is an inclined line. This is quite different from the previous algorithms we have introduced.

```
plot_decision_regions(X_combined, y_combined,
                      classifier=dtc, test_idx=range(105, 150))
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.legend(loc='upper left')
plt.show()
```



In scikit-learn, we can also plot the decision tree. Since the tree diagram is too complicated to be shown under the default resolution in Jupyter Notebook, we can save it as a file which support higher resolution such as svg or pdf.

```
from sklearn import tree
tree.plot_tree(dtc)
plt.savefig("tree.pdf")
```



X[1] <= 0.75
gini = 0.667
samples = 105
value = [35, 35, 35]
gini = 0.0
samples = 35
value = [35, 0, 0]
X[0] <= 4.75
gini = 0.5
samples = 70
value = [0, 35, 35]
gini = 0.0
samples = 30
value = [0, 30, 0]
X[1] <= 1.75
gini = 0.219
samples = 40
value = [0, 5, 35]
X[0] <= 4.95
gini = 0.5
samples = 8
value = [0, 4, 4]
X[0] <= 4.85
gini = 0.061
samples = 32
value = [0, 1, 31]
gini = 0.0
samples = 2
value = [0, 2, 0]
gini = 0.444
samples = 6
value = [0, 2, 4]
gini = 0.444
samples = 3
value = [0, 1, 2]
gini = 0.0
samples = 29
value = [0, 0, 29]