Artificial Intelligence & Machine Learning

Experiment No. 9

Implementation of Classifying data using support vectormachine (SVMs)

PRACTICAL NO. 9

Aim: Implementation of Classifying data using support vector machine (SVMs)

Objective: Understand classifying data using support

vector machine (SVMs) Software Requirement:

- **Anaconda Navigator:** Anaconda Navigator is a desktop graphical user interface included in Anaconda that allows you to launch applications and easily manage conda packages, environments and channels without the need to use command line commands.

Theory:

Support vector machine or SVM is one of the most popular supervised learning algorithm which is used for classification as well as regression problem however, primarily, it used for classification problem in machine learning.

Types of SVM

SVM can be of 2 types:

- **Linear SVM:** Linear SVM is **used for linearly separable data**, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-Linear SVM:** Nonlinear classification: **SVM can be extended to solve nonlinear classification tasks when the set of samples cannot be separated linearly**. By applying kernel functions, the samples are mapped onto a high-dimensional feature space, in which the linear classification is possible.
- **Support vectors:** Support vectors are **data points that are closer to the hyperplane and influence the position and orientation of the hyperplane**. Using these support vectors, we maximize the margin of the classifier.
- **Hyperplane:** A hyperplane is **a decision boundary that differentiates the two classes in SVM**. A data point falling on either side of the hyperplane can be attributed to different classes. The dimension of the hyperplane depends on the number of input features in the dataset.

- **Marginal Distance: The distance between the line and the closest data points** is referred to as the margin. The best or optimal line that can separate the two classes is the line that as the largest margin.

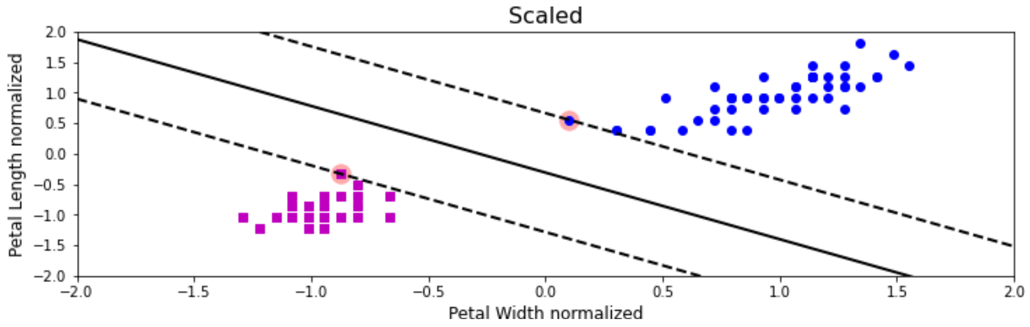
Code & Output:

1) SVM Linear

```
In [1]: %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt

        def plot_svc_decision_boundary(svm_clf, xmin, xmax):
            w = svm_clf.coef_[0]
            b = svm_clf.intercept_[0]

            # At the decision boundary, w0*x0 + w1*x1 + b = 0
            # => x1 = -w0/w1 * x0 - b/w1
            x0 = np.linspace(xmin, xmax, 200)
            decision_boundary = -w[0]/w[1] * x0 - b/w[1]

            margin = 1/w[1]
            gutter_up = decision_boundary + margin
            gutter_down = decision_boundary - margin

            svs = svm_clf.support_vectors_
            plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA')
            plt.plot(x0, decision_boundary, "k-", linewidth=2)
            plt.plot(x0, gutter_up, "k--", linewidth=2)
            plt.plot(x0, gutter_down, "k--", linewidth=2)
```

```
In [2]: from sklearn.svm import SVC
        from sklearn import datasets

        iris = datasets.load_iris()
        #print(iris)
        X = iris["data"][:, (2, 3)]  # petal length, petal width
        #print(X)

        y = iris["target"]

        setosa_or_versicolor = (y == 0) | (y == 1)
        X = X[setosa_or_versicolor]
        y = y[setosa_or_versicolor]

        # SVM Classifier model
        #the hyperparameter control the margin violations
        #smaller C leads to more margin violations but wider street
        #C can be inferred
        svm_clf = SVC(kernel="linear", C=float("inf"))
        svm_clf.fit(X, y)

        svm_clf.predict([[2.4, 3.1]])

        #SVM classifiers do not output a probability like logistic regression classifiers

Out[2]: array([1])
```

```
In [3]: #plot the decision boundaries
        import numpy as np

        plt.figure(figsize=(12,3.2))

        from sklearn.preprocessing import StandardScaler
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
        svm_clf.fit(X_scaled, y)

        plt.plot(X_scaled[:, 0][y==1], X_scaled[:, 1][y==1], "bo")
        plt.plot(X_scaled[:, 0][y==0], X_scaled[:, 1][y==0], "ms")
        plot_svc_decision_boundary(svm_clf, -2, 2)
        plt.xlabel("Petal Width normalized", fontsize=12)
        plt.ylabel("Petal Length normalized", fontsize=12)
        plt.title("Scaled", fontsize=16)
        plt.axis([-2, 2, -2, 2])

Out[3]: (-2.0, 2.0, -2.0, 2.0)
```

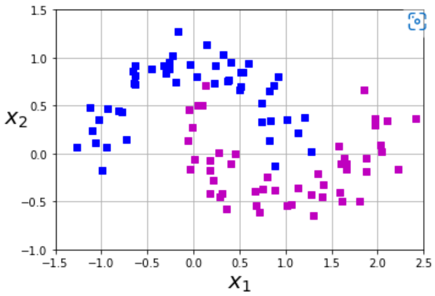


2) SVM Non-Linear

In [1]:
```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

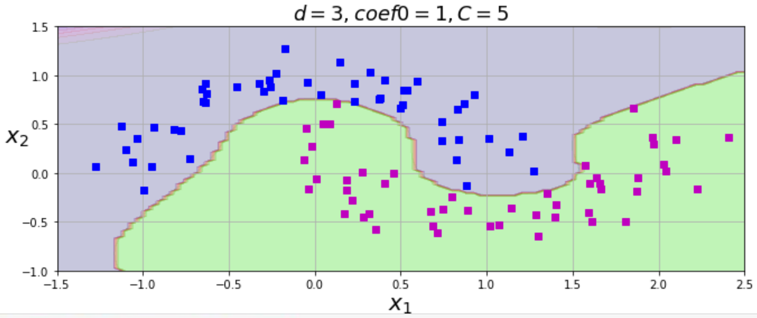
In [2]:
```
import numpy as np
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

In [3]:
```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

#define a function to plot the dataset
def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "ms")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

#Let's have a look at the data we have generated
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```



In [4]:
```
#define a function plot the decision boundaries
def plot_predictions(clf, axes):
    #create data in continous linear space
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)
```

In [6]:
```
#plot the decision boundaries
plt.figure(figsize=(11, 4))

#plot the decision boundaries
plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])

#plot the dataset
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.title(r"$d=3, coef0=1, C=5$", fontsize=18)
plt.show()
```



In [5]:
```
#C controls the width of the street
#Degree of data

#create a pipeline to create features, scale data and fit the model
polynomial_svm_clf = Pipeline((
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scalar", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=10, coef0=1, C=5))
))

#call the pipeline
polynomial_svm_clf.fit(X,y)
```

Out[5]:
```
Pipeline(steps=[('poly_features', PolynomialFeatures(degree=3)),
                ('scalar', StandardScaler()),
                ('svm_clf', SVC(C=5, coef0=1, degree=10, kernel='poly'))])
```

Implementation of Bagging Algorithm: Decision Tree,Random Forest

PRACTICAL NO. 10

Aim: Implementation of Bagging Algorithm: Decision Tree, Random Forest

Objective: To Learn decision tree , different ensemble techniques like bagging, Random forest classification and regression.

- **Anaconda Navigator:** Anaconda Navigator is a desktop graphical user interface included in Anaconda that allows you to launch applications and easily manage conda packages, environments and channels without the need to use command line commands.

Theory:

1) Decision Tree:

Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree- structured classifier, where internal nodes represent the features of a dataset, branches representthe decision rules and each leaf node represents the outcome.

2) Random Forest:

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of **ensemble learning,** which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model.

a) **Classification:** A random forest **produces good predictions that can be understood easily**. It can handle large datasets efficiently. The random forest algorithm provides a higher level of accuracy in predicting outcomes over the decision tree algorithm.

b) **Regression:** Random Forest Regression is **a supervised learning algorithm that uses ensemble learning method for regression**.

Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

Code &

Output:

1.Decision

Tree

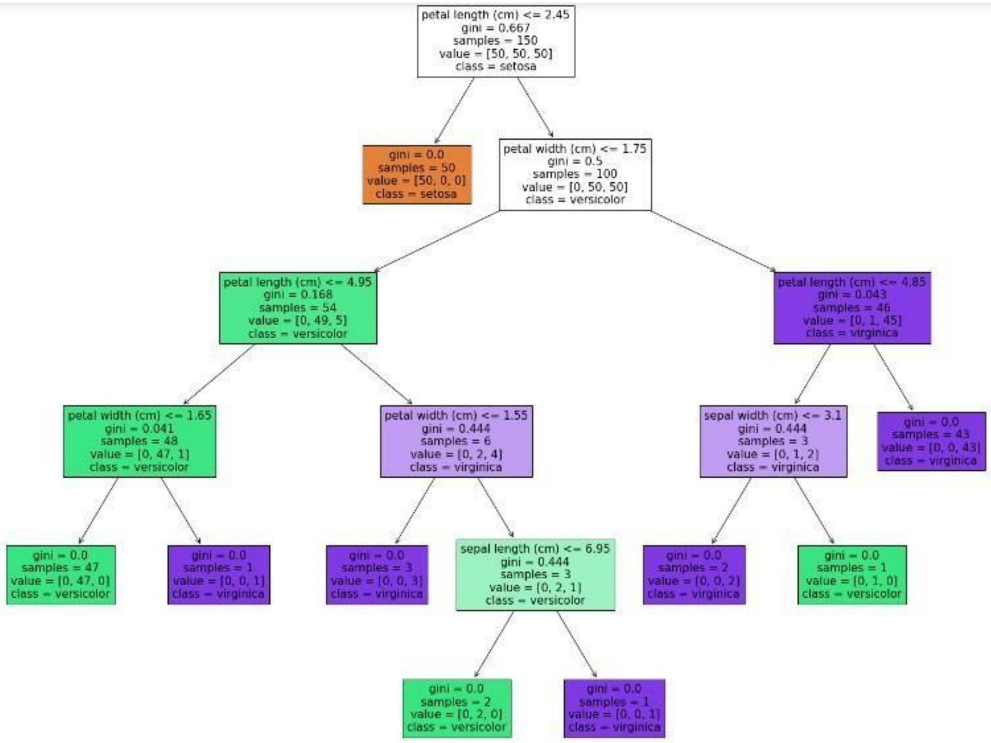
```
In [15]: from matplotlib import pyplot as plt
         from sklearn import datasets
         from sklearn.tree import DecisionTreeClassifier
         from sklearn import tree

In [16]: # Prepare the data data
         iris = datasets.load_iris()
         X = iris.data
         y = iris.target

In [17]: # Fit the classifier with default hyper-parameters
         clf = DecisionTreeClassifier(random_state=1234)
         model = clf.fit(X, y)
```

```
In [18]: text_representation = tree.export_text(clf)
         print(text_representation)

|--- feature_2 <= 2.45
|   |--- class: 0
|--- feature_2 >  2.45
|   |--- feature_3 <= 1.75
|   |   |--- feature_2 <= 4.95
|   |   |   |--- feature_3 <= 1.65
|   |   |   |   |--- class: 1
|   |   |   |--- feature_3 >  1.65
|   |   |   |   |--- class: 2
|   |   |--- feature_2 >  4.95
|   |   |   |--- feature_3 <= 1.55
|   |   |   |   |--- class: 2
|   |   |   |--- feature_3 >  1.55
|   |   |   |   |--- feature_0 <= 6.95
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- feature_0 >  6.95
|   |   |   |   |   |--- class: 2
|   |--- feature_3 >  1.75
|   |   |--- feature_2 <= 4.85
|   |   |   |--- feature_1 <= 3.10
|   |   |   |   |--- class: 2
|   |   |   |--- feature_1 >  3.10
|   |   |   |   |--- class: 1
|   |   |--- feature_2 >  4.85
|   |   |   |--- class: 2
```

s

```
In [19]: with open("decistion_tree.log", "w") as fout:
             fout.write(text_representation)
```

```
In [20]: fig = plt.figure(figsize=(25,20))
         _ = tree.plot_tree(clf,
                            feature_names=iris.feature_names,
                            class_names=iris.target_names,
                            filled=True)
```



2. Random Forest:

a) Classification:

Random Forest for Classifying Digits

```
In [7]: from sklearn.datasets import load_digits  #load digits dataset from sklearn libraries
        import matplotlib.pyplot as plt
        digits = load_digits()
```

```
In [8]: digits   #shows different elements...
```

```
Out[8]: {'data': array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ..., 10.,  0.,  0.],
                [ 0.,  0.,  0., ..., 16.,  9.,  0.],
                ...,
                [ 0.,  0.,  1., ...,  6.,  0.,  0.],
                [ 0.,  0.,  2., ..., 12.,  0.,  0.],
                [ 0.,  0., 10., ..., 12.,  1.,  0.]]),
         'target': array([0, 1, 2, ..., 8, 9, 8]),
         'frame': None,
         'feature_names': ['pixel_0_0',
          'pixel_0_1',
          'pixel_0_2',
          'pixel_0_3',
          'pixel_0_4',
          'pixel_0_5',
          'pixel_0_6',
          'pixel_0_7',
          'pixel_1_0',
          'pixel_1_1',
```

```
In [9]: digits.data   #data element is 2D array.
```

```
Out[9]: array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
               [ 0.,  0.,  0., ..., 10.,  0.,  0.],
               [ 0.,  0.,  0., ..., 16.,  9.,  0.],
               ...,
               [ 0.,  0.,  1., ...,  6.,  0.,  0.],
               [ 0.,  0.,  2., ..., 12.,  0.,  0.],
               [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

```
In [10]: digits.keys()  #shows datapoints.
```

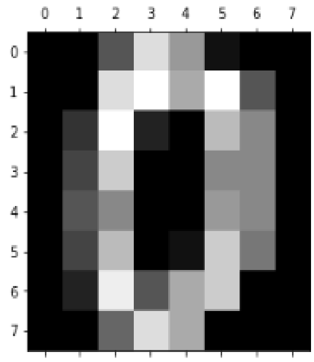
```
Out[10]: dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

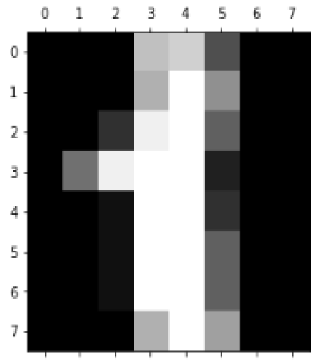
```
In [11]: digits.data[0]  #64 length array. 8*8 digit colour map.
```

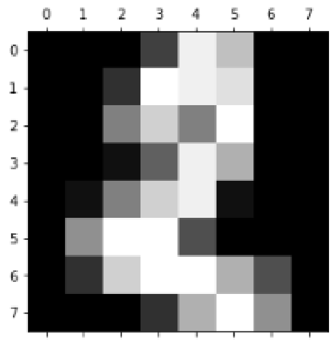
```
Out[11]: array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
                15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
                12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
                 0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
                10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

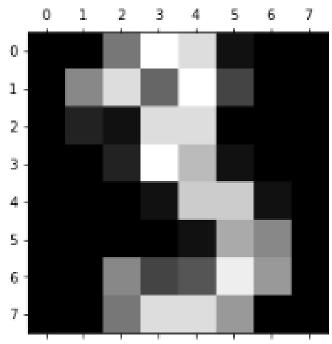
```
In [12]: plt.gray() #plot in gray
         for i in range(4):  #first 5 elt
             plt.matshow(digits.images[i]) #shows matrix as image
```

<Figure size 432x288 with 0 Axes>

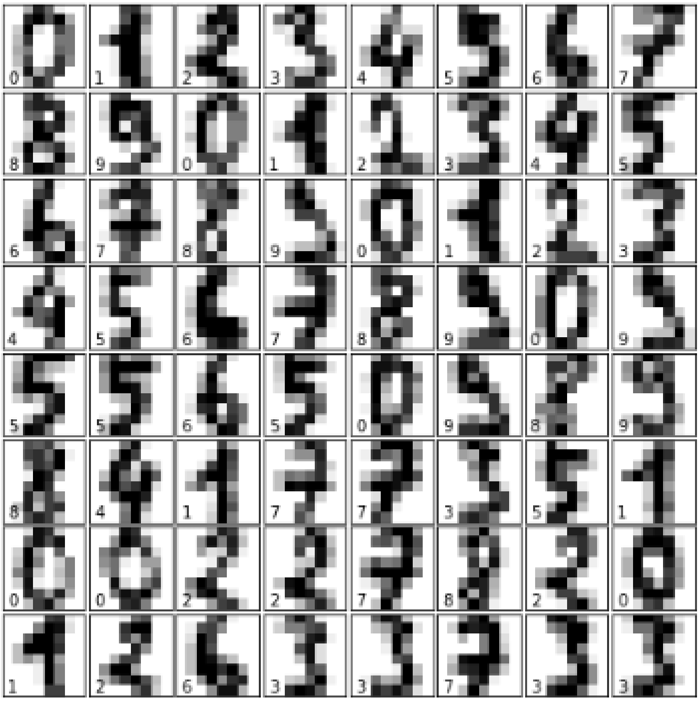








In [2]:
```
# set up the figure
fig = plt.figure(figsize=(6, 6))  # figure size in inches
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

# plot the digits: each image is 8x8 pixels
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')

    # label the image with the target value
    ax.text(0, 7, str(digits.target[i]))
```



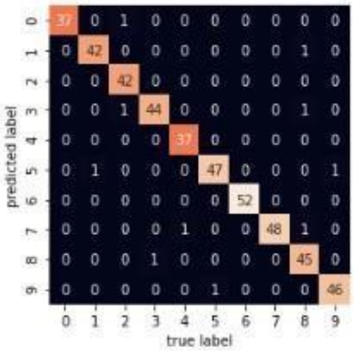
In [18]:
```
from sklearn.model_selection import train_test_split  #divide dataset into train and test set
from sklearn.ensemble import RandomForestClassifier

Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,
                                                random_state=0)
#create model
model = RandomForestClassifier(n_estimators=100) #n_estimator shows number of trees in the forest. accuracy depends on tht.
model.fit(Xtrain, ytrain) # fit model. it is training step put X and y
ypred = model.predict(Xtest) #calculate ypred value for Xtest
```

In [19]:
```
from sklearn import metrics
print(metrics.classification_report(ypred, ytest)) #comparing ypred with ytest and giving score
```

```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99        38
           1       0.98      0.95      0.97        44
           2       0.95      1.00      0.98        42
           3       0.98      0.96      0.97        46
           4       0.97      0.97      0.97        38
           5       0.98      0.96      0.97        49
           6       1.00      1.00      1.00        52
           7       0.98      0.96      0.97        49
           8       0.94      0.98      0.96        46
           9       0.96      0.98      0.97        46

    accuracy                           0.97       450
   macro avg       0.97      0.97      0.97       450
weighted avg       0.97      0.97      0.97       450
```

from 100 samples 97 are correctly classified.

```
In [16]: from sklearn.metrics import confusion_matrix
         import seaborn as sns
         mat = confusion_matrix(ytest, ypred)
         sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
         plt.xlabel('true label')
         plt.ylabel('predicted label');
```

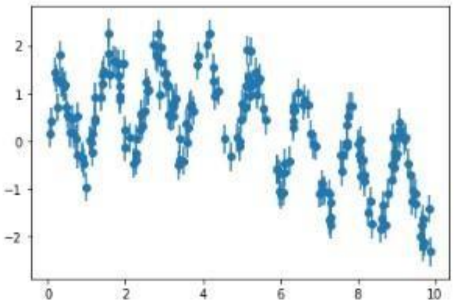


Confusion matrix will show mistakes of your model. just check diagonally against true label and predicted label.

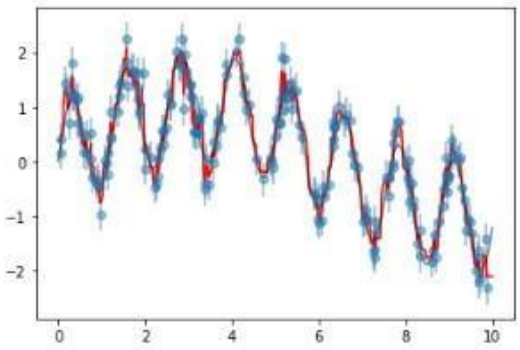
b) Regression:

```
In [3]: from sklearn.datasets import load_digits
        import matplotlib.pyplot as plt
        from sklearn.model_selection import train_test_split
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.metrics import confusion_matrix
        import seaborn as sns
        import numpy as np
```

```
In [4]: rng = np.random.RandomState(42)
        x = 10 * rng.rand(200) #crate array of specified shape & fill random values in given shape.
        #draw fast and slow oscillation...
        def model(x, sigma=0.3):
            fast_oscillation = np.sin(5 * x)
            slow_oscillation = np.sin(0.5 * x)
            noise = sigma * rng.randn(len(x))

            return slow_oscillation + fast_oscillation + noise

        y = model(x)
        plt.errorbar(x, y, 0.3, fmt='o');
```



In [5]:
```
#using random forest regressor we can find best fit curve.

from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(200)
forest.fit(x[:, None], y)

xfit = np.linspace(0, 10, 1000)
yfit = forest.predict(xfit[:, None])
ytrue = model(xfit, sigma=0)

plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```



output shows true model in the smooth gray curve, while random forest model is shown by the jagged red curve.

Artificial Intelligence & Machine LearningExperiment No. 11

Implementation of Boosting Algorithms: AdaBoost, Stochastic Gradient Boosting, Voting Ensemble.

PRACTICAL NO 11

Aim: Implementation of Boosting Algorithms: AdaBoost, Stochastic Gradient Boosting, Voting Ensemble.

Objective: To learn AdaBoost, Stochastic Gradient Boosting, Voting Ensemble.

Software Requirement:

- **Anaconda Navigator:** Anaconda Navigator is a desktop graphical user interface included in Anaconda that allows you to launch applications and easily manage conda packages, environments and channels without the need to use command line commands.

Theory:

- **Boosting:** Boosting is a method used in machine learning to reduce errors in predictive data analysis. Data scientists train machine learning software, called machine learning models, on labeled data to make guesses about unlabeled data. A single machine learning model might make prediction errors depending on the accuracy of the training dataset.
- **AdaBoost:** AdaBoost also called Adaptive Boosting is a technique in Machine Learning used as an Ensemble Method. The most common algorithm used with AdaBoost is decision trees with one level that means with Decision trees with only 1 split. These trees are also called **Decision Stumps.**
- **Ensemble Methods:** Ensemble methods is a machine learning technique that combines several base models in order to produce one optimal predictive model.

  Ensemble learning is a powerful machine learning algorithm that is used across industries by data science experts. The beauty of ensemble learning techniques is that they combine the prediction of multiple machine learning Models.
- **Soft Voting:** combining the probabilities of each prediction in each model and picking the prediction with the highest total probability.
- **Hard Voting:** Hard voting entails picking the prediction with the highest number of votes

- * Steps for Adaboost Algorithm

  1] Initialize the weights as 1/n to every n observation

  2] Select the 1 feature according to lowest Gini / Highest Information given and calculate total error.

  3] Calculate the performance of the Setup.

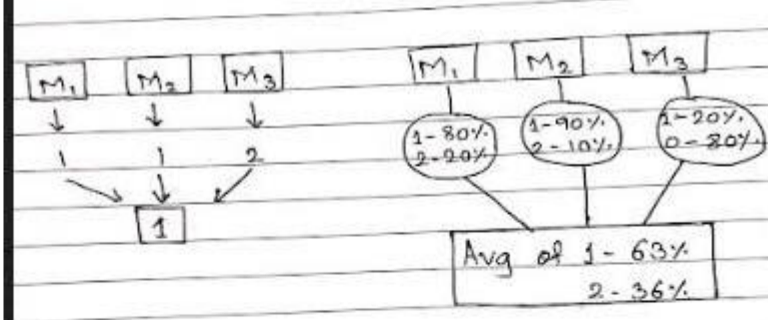
  4] Calculate the new weights for each misclassification (increase) and right classification (decrease)

  5] Normalize the new weights so that the sum of the weight is 1.

  6] Now, Repeat from Step 2 and so on till the configured number of estimators reached or the accuracy achieved.

- Softvoting -
  Classifies input data based on probabilities of all prediction made by classifiers.

  Hardvoting -
  Based on majority vote



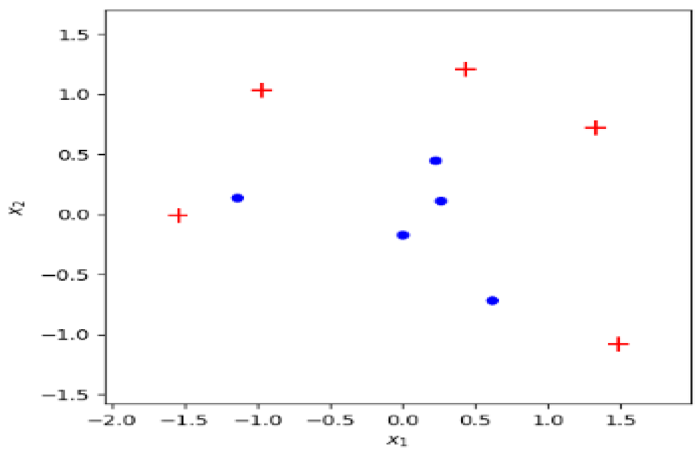
1) AdaBoost:

In [1]:
```
from typing import Optional
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
```

In [2]:
```
def plot_adaboost(X: np.ndarray,
                  y: np.ndarray,
                  clf=None,
                  sample_weights: Optional[np.ndarray] = None,
                  annotate: bool = False,
                  ax: Optional[mpl.axes.Axes] = None) -> None:
    """ Plot ± samples in 2D, optionally with decision boundary """

    assert set(y) == {-1, 1}, 'Expecting response labels to be ±1'

    if not ax:
        fig, ax = plt.subplots(figsize=(5, 5), dpi=100)
        fig.set_facecolor('white')

    pad = 1
    x_min, x_max = X[:, 0].min() - pad, X[:, 0].max() + pad
    y_min, y_max = X[:, 1].min() - pad, X[:, 1].max() + pad

    if sample_weights is not None:
        sizes = np.array(sample_weights) * X.shape[0] * 100
    else:
        sizes = np.ones(shape=X.shape[0]) * 100

    X_pos = X[y == 1]
    sizes_pos = sizes[y == 1]
    ax.scatter(*X_pos.T, s=sizes_pos, marker='+', color='red')

    X_neg = X[y == -1]
    sizes_neg = sizes[y == -1]
    ax.scatter(*X_neg.T, s=sizes_neg, marker='.', c='blue')

    if clf:
        plot_step = 0.01
        xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                             np.arange(y_min, y_max, plot_step))

        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        # If all predictions are positive class, adjust color map acordingly
        if list(np.unique(Z)) == [1]:
            fill_colors = ['r']
        else:
            fill_colors = ['b', 'r']

        ax.contourf(xx, yy, Z, colors=fill_colors, alpha=0.2)

    if annotate:
        for i, (x, y) in enumerate(X):
            offset = 0.05
            ax.annotate(f'$x_{i + 1}$', (x + offset, y - offset))

    ax.set_xlim(x_min+0.5, x_max-0.5)
    ax.set_ylim(y_min+0.5, y_max-0.5)
    ax.set_xlabel('$x_1$')
    ax.set_ylabel('$x_2$')
```

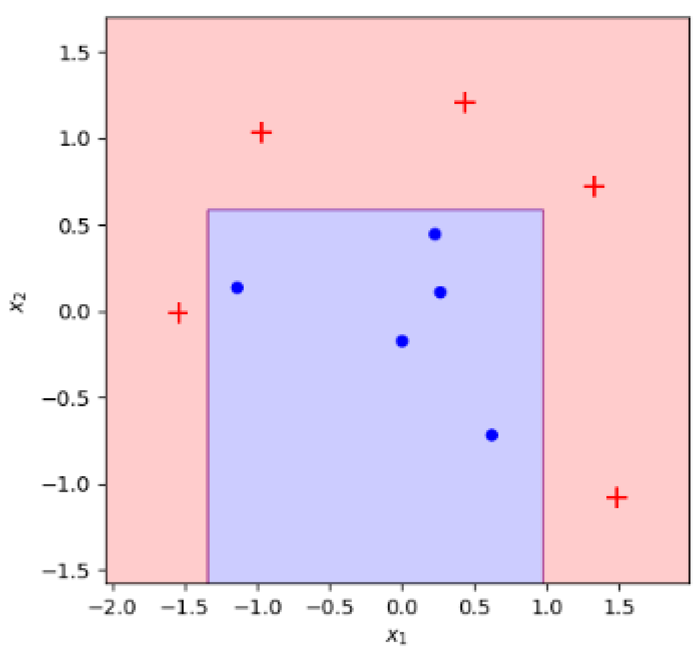
s

In [3]:
```
from sklearn.datasets import make_gaussian_quantiles
from sklearn.model_selection import train_test_split

def make_toy_dataset(n: int = 100, random_seed: int = None):
    """ Generate a toy dataset for evaluating AdaBoost classifiers """

    n_per_class = int(n/2)

    if random_seed:
        np.random.seed(random_seed)

    X, y = make_gaussian_quantiles(n_samples=n, n_features=2, n_classes=2)

    return X, y*2-1

X, y = make_toy_dataset(n=10, random_seed=10)
plot_adaboost(X, y)
```



In [4]:
```
from sklearn.ensemble import AdaBoostClassifier

bench = AdaBoostClassifier(n_estimators=10, algorithm='SAMME').fit(X, y)
plot_adaboost(X, y, bench)

train_err = (bench.predict(X) != y).mean()
print(f'Train error: {train_err:.1%}')
```

```
Train error: 0.0%
```



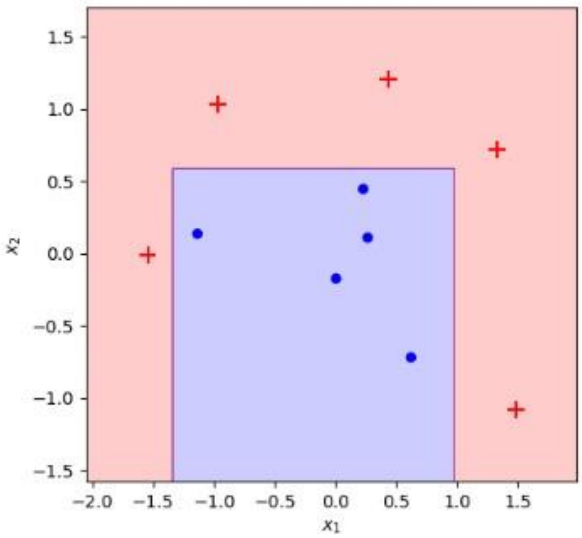
In [5]:
```
class AdaBoost:


    def __init__(self):
        self.stumps = None
        self.stump_weights = None
        self.errors = None
        self.sample_weights = None

    def _check_X_y(self, X, y):
        """ Validate assumptions about format of input data"""
        assert set(y) == {-1, 1}, 'Response variable must be ±1'
        return X, y
```

In [6]:
```
from sklearn.tree import DecisionTreeClassifier

def fit(self, X: np.ndarray, y: np.ndarray, iters: int):
    """ Fit the model using training data """

    X, y = self._check_X_y(X, y)
    n = X.shape[0]

    # init numpy arrays
    self.sample_weights = np.zeros(shape=(iters, n))
    self.stumps = np.zeros(shape=iters, dtype=object)
    self.stump_weights = np.zeros(shape=iters)
    self.errors = np.zeros(shape=iters)

    # initialize weights uniformly
    self.sample_weights[0] = np.ones(shape=n) / n

    for t in range(iters):
        # fit  weak learner
        curr_sample_weights = self.sample_weights[t]
        stump = DecisionTreeClassifier(max_depth=1, max_leaf_nodes=2)
        stump = stump.fit(X, y, sample_weight=curr_sample_weights)

        # calculate error and stump weight from weak learner prediction
        stump_pred = stump.predict(X)
        err = curr_sample_weights[(stump_pred != y)].sum()# / n
        stump_weight = np.log((1 - err) / err) / 2
```

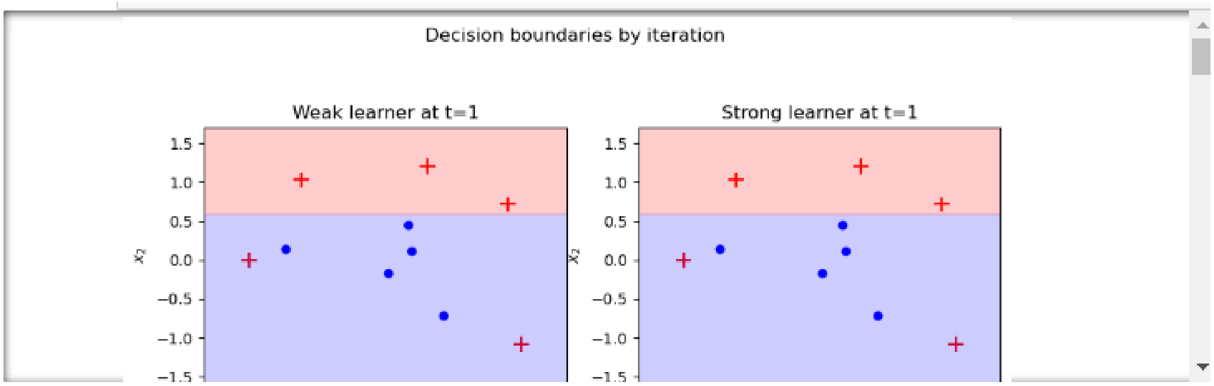
```
        # update sample weights
        new_sample_weights = (
            curr_sample_weights * np.exp(-stump_weight * y * stump_pred)
        )

        new_sample_weights /= new_sample_weights.sum()

        # If not final iteration, update sample weights for t+1
        if t+1 < iters:
            self.sample_weights[t+1] = new_sample_weights

        # save results of iteration
        self.stumps[t] = stump
        self.stump_weights[t] = stump_weight
        self.errors[t] = err

    return self
#Making predictions
#We make a final prediction by taking a “weighted majority vote”, calculated as the sign (±) of the linear combination of each s

#$$ H_t(x) = \text{sign} \Big( \sum_{t=1}^T a_t h_t(x) \Big) $$

def predict(self, X):
    """ Make predictions using already fitted model """
    stump_preds = np.array([stump.predict(X) for stump in self.stumps])
    return np.sign(np.dot(self.stump_weights, stump_preds))
```

```
In [7]: # assign our individually defined functions as methods of our classifier
AdaBoost.fit = fit
AdaBoost.predict = predict

clf = AdaBoost().fit(X, y, iters=10)
plot_adaboost(X, y, clf)

train_err = (clf.predict(X) != y).mean()
print(f'Train error: {train_err:.1%}')
```

```
Train error: 0.0%
```



```
In [8]: def truncate_adaboost(clf, t: int):
            """ Truncate a fitted AdaBoost up to (and including) a particular iteration """
            assert t > 0, 't must be a positive integer'
            from copy import deepcopy
            new_clf = deepcopy(clf)
            new_clf.stumps = clf.stumps[:t]
            new_clf.stump_weights = clf.stump_weights[:t]
            return new_clf


        def plot_staged_adaboost(X, y, clf, iters=10):
            """ Plot weak learner and cumulaive strong learner at each iteration. """

            # Larger grid
            fig, axes = plt.subplots(figsize=(8, iters*3),
                                     nrows=iters,
                                     ncols=2,
                                     sharex=True,
                                     dpi=100)

            fig.set_facecolor('white')

            _ = fig.suptitle('Decision boundaries by iteration')
            for i in range(iters):
                ax1, ax2 = axes[i]

                # Plot weak learner
                _ = ax1.set_title(f'Weak learner at t={i + 1}')
                plot_adaboost(X, y, clf.stumps[i],
                              sample_weights=clf.sample_weights[i],
                              annotate=False, ax=ax1)

                # Plot strong learner
                trunc_clf = truncate_adaboost(clf, t=i + 1)
                _ = ax2.set_title(f'Strong learner at t={i + 1}')
                plot_adaboost(X, y, trunc_clf,
                              sample_weights=clf.sample_weights[i],
                              annotate=False, ax=ax2)

            plt.tight_layout()
            plt.subplots_adjust(top=0.95)
            plt.show()

        clf = AdaBoost().fit(X, y, iters=10)
        plot_staged_adaboost(X, y, clf)
```




```
In [ ]:
```

2) Adaboost using decision tree:

```
In [1]: #importing libraries

        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from random import sample
        import random
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.metrics import confusion_matrix
        from sklearn import tree
        from math import log,exp
```

```
In [2]: pd.set_option('display.max_rows', 500)
        pd.set_option('display.max_columns', 500)
```

```
In [3]: #importing file
        iris = pd.read_csv("iris.csv")
```

```
In [4]: iris = iris.drop('Unnamed: 0', axis=1)
```

```
In [5]: iris.head(1)
```

Out[5]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa

```
In [6]: #considering only two classes
        example = iris[(iris['Species'] == 'versicolor') | (iris['Species'] == 'virginica')]
```

```
In [7]: example.head(2)
```

Out[7]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
50	7.0	3.2	4.7	1.4	versicolor
51	6.4	3.2	4.5	1.5	versicolor

```
In [8]: #replacing the two classes with +1 and -1
        example['Label'] = example['Species'].replace(to_replace = ['versicolor','virginica'], value=[1,-1])
```

```
<ipython-input-8-241c08c9f205>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  example['Label'] = example['Species'].replace(to_replace = ['versicolor','virginica'], value=[1,-1])
```

```
In [9]: example = example.drop('Species', axis = 1)
```

```
In [10]: #Initially assign same weights to each records in the dataset
         example['probR1'] = 1/(example.shape[0])
```

```
In [11]: example.head(5)
```

Out[11]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1
50	7.0	3.2	4.7	1.4	1	0.01
51	6.4	3.2	4.5	1.5	1	0.01
52	6.9	3.1	4.9	1.5	1	0.01
53	5.5	2.3	4.0	1.3	1	0.01
54	6.5	2.8	4.6	1.5	1	0.01

In [12]:
```
#simple random sample with replacement
random.seed(10)
example1 = example.sample(len(example), replace = True, weights = example['probR1'])
```

In [13]:
```
example1
```

Out[13]:

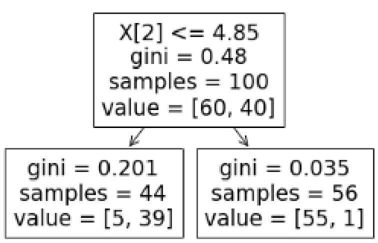
	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1
137	6.4	3.1	5.5	1.8	-1	0.01
84	5.4	3.0	4.5	1.5	1	0.01
66	5.6	3.0	4.5	1.5	1	0.01
87	6.3	2.3	4.4	1.3	1	0.01
66	5.6	3.0	4.5	1.5	1	0.01
76	6.8	2.8	4.8	1.4	1	0.01
75	6.6	3.0	4.4	1.4	1	0.01
84	5.4	3.0	4.5	1.5	1	0.01
118	7.7	2.6	6.9	2.3	-1	0.01
147	6.5	3.0	5.2	2.0	-1	0.01
85	6.0	3.4	4.5	1.6	1	0.01
79	5.7	2.6	3.5	1.0	1	0.01

In [14]:
```
#X_train and Y_train split
X_train = example1.iloc[0:len(iris),0:4]
y_train = example1.iloc[0:len(iris),4]
```

In [15]:
```
#fitting the DT model with depth one
clf_gini = DecisionTreeClassifier(criterion = "gini", random_state = 100, max_depth=1)
clf = clf_gini.fit(X_train, y_train)
```

In [16]:
```
#plotting tree for round 1 boosting
tree.plot_tree(clf)
```

Out[16]:
```
[Text(167.4, 163.07999999999998, 'X[2] <= 4.85\ngini = 0.48\nsamples = 100\nvalue = [60, 40]'),
 Text(83.7, 54.360000000000014, 'gini = 0.201\nsamples = 44\nvalue = [5, 39]'),
 Text(251.10000000000002, 54.360000000000014, 'gini = 0.035\nsamples = 56\nvalue = [55, 1]')]
```



In [17]:
```
#prediction
y_pred = clf_gini.predict(example.iloc[0:len(iris),0:4])
y_pred
```

Out[17]:
```
array([ 1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1, -1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1, -1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,
       -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
      dtype=int64)
```

In [18]:
```
#adding a column pred1 after the first round of boosting
example['pred1'] = y_pred
```

In [19]:
```
example
```

Out[19]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1
50	7.0	3.2	4.7	1.4	1	0.01	1
51	6.4	3.2	4.5	1.5	1	0.01	1
52	6.9	3.1	4.9	1.5	1	0.01	-1
53	5.5	2.3	4.0	1.3	1	0.01	1
54	6.5	2.8	4.6	1.5	1	0.01	1
55	5.7	2.8	4.5	1.3	1	0.01	1
56	6.3	3.3	4.7	1.6	1	0.01	1
57	4.9	2.4	3.3	1.0	1	0.01	1
58	6.6	2.9	4.6	1.3	1	0.01	1
59	5.2	2.7	3.9	1.4	1	0.01	1
60	5.0	2.0	3.5	1.0	1	0.01	1
61	5.9	3.0	4.2	1.5	1	0.01	1

In [20]:
```
#misclassified = 0 if the label and prediction are same
example.loc[example.Label != example.pred1, 'misclassified'] = 1
example.loc[example.Label == example.pred1, 'misclassified'] = 0
```

In [21]:
```
#error calculation
e1 = sum(example['misclassified'] * example['probR1'])
```

In [22]:
```
e1
```

Out[22]: 0.07

In [23]:
```
#calculation of alpha (performance)
alpha1 = 0.5*log((1-e1)/e1)
```

In [24]:
```
#update weight
new_weight = example['probR1']*np.exp(-1*alpha1*example['Label']*example['pred1'])
```

In [25]:
```
#normalized weight
z = sum(new_weight)
normalized_weight = new_weight/sum(new_weight)
```

In [26]:
```
example['prob2'] = round(normalized_weight,4)
```

In [27]:
```
example
```

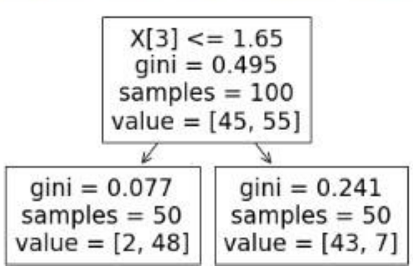
Out[27]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1	misclassified	prob2
50	7.0	3.2	4.7	1.4	1	0.01	1	0.0	0.0054
51	6.4	3.2	4.5	1.5	1	0.01	1	0.0	0.0054
52	6.9	3.1	4.9	1.5	1	0.01	-1	1.0	0.0714
53	5.5	2.3	4.0	1.3	1	0.01	1	0.0	0.0054
54	6.5	2.8	4.6	1.5	1	0.01	1	0.0	0.0054
55	5.7	2.8	4.5	1.3	1	0.01	1	0.0	0.0054
56	6.3	3.3	4.7	1.6	1	0.01	1	0.0	0.0054
57	4.9	2.4	3.3	1.0	1	0.01	1	0.0	0.0054
58	6.6	2.9	4.6	1.3	1	0.01	1	0.0	0.0054
59	5.2	2.7	3.9	1.4	1	0.01	1	0.0	0.0054
60	5.0	2.0	3.5	1.0	1	0.01	1	0.0	0.0054
61	5.9	3.0	4.2	1.5	1	0.01	1	0.0	0.0054

In [28]:
```
#round 2
random.seed(20)
example2 = example.sample(len(example), replace = True, weights = example['prob2'])
example2 = example2.iloc[:,0:5]
X_train = example2.iloc[0:len(iris),0:4]
y_train = example2.iloc[0:len(iris),4]

clf_gini = DecisionTreeClassifier(criterion = "gini", random_state = 100, max_depth=1)
clf = clf_gini.fit(X_train, y_train)

y_pred = clf_gini.predict(example.iloc[0:len(iris),0:4])
#adding a column pred2 after the second round of boosting
example['pred2'] = y_pred
```

In [29]:
```
#plotting tree for round 2 boosting
tree.plot_tree(clf)
```

Out[29]:
```
[Text(167.4, 163.07999999999998, 'X[3] <= 1.65\ngini = 0.495\nsamples = 100\nvalue = [45, 55]'),
 Text(83.7, 54.360000000000014, 'gini = 0.077\nsamples = 50\nvalue = [2, 48]'),
 Text(251.10000000000002, 54.360000000000014, 'gini = 0.241\nsamples = 50\nvalue = [43, 7]')]
```



In [30]:
```
example
```
Out[30]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1	misclassified	prob2	pred2
50	7.0	3.2	4.7	1.4	1	0.01	1	0.0	0.0054	1
51	6.4	3.2	4.5	1.5	1	0.01	1	0.0	0.0054	1
52	6.9	3.1	4.9	1.5	1	0.01	-1	1.0	0.0714	1
53	5.5	2.3	4.0	1.3	1	0.01	1	0.0	0.0054	1
54	6.5	2.8	4.6	1.5	1	0.01	1	0.0	0.0054	1
55	5.7	2.8	4.5	1.3	1	0.01	1	0.0	0.0054	1
56	6.3	3.3	4.7	1.6	1	0.01	1	0.0	0.0054	1
57	4.9	2.4	3.3	1.0	1	0.01	1	0.0	0.0054	1
58	6.6	2.9	4.6	1.3	1	0.01	1	0.0	0.0054	1
59	5.2	2.7	3.9	1.4	1	0.01	1	0.0	0.0054	1
60	5.0	2.0	3.5	1.0	1	0.01	1	0.0	0.0054	1
61	5.9	3.0	4.2	1.5	1	0.01	1	0.0	0.0054	1

In [31]:
```
#adding a field misclassified2
example.loc[example.Label != example.pred2, 'misclassified2'] = 1
example.loc[example.Label == example.pred2, 'misclassified2'] = 0
```
In [32]:
```
# calculation of error
e2 = sum(example['misclassified2'] * example['prob2'])
e2
```
Out[32]: `0.09840000000000002`

In [33]:
```
#calculation of alpha
alpha2 = 0.5*log((1-e2)/e2)
alpha2
```
Out[33]: `1.1075650793336793`

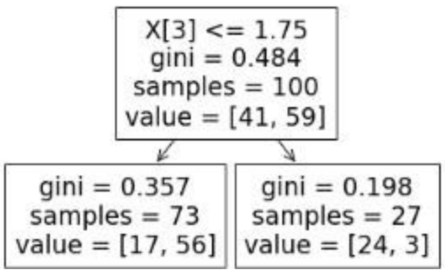
In [34]:
```
#update weight
new_weight = example['prob2']*np.exp(-1*alpha2*example['Label']*example['pred2'])
z = sum(new_weight)
normalized_weight = new_weight/sum(new_weight)
```
In [35]:
```
example['prob3'] = round(normalized_weight,4)
```
In [36]:
```
example
```
Out[36]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1	misclassified	prob2	pred2	misclassified2	prob3
50	7.0	3.2	4.7	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030
51	6.4	3.2	4.5	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030
52	6.9	3.1	4.9	1.5	1	0.01	-1	1.0	0.0714	1	0.0	0.0396
53	5.5	2.3	4.0	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030
54	6.5	2.8	4.6	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030
55	5.7	2.8	4.5	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030
56	6.3	3.3	4.7	1.6	1	0.01	1	0.0	0.0054	1	0.0	0.0030
57	4.9	2.4	3.3	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030
58	6.6	2.9	4.6	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030
59	5.2	2.7	3.9	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030
60	5.0	2.0	3.5	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030
61	5.9	3.0	4.2	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030

In [37]:
```
#round 3
random.seed(30)
example3 = example.sample(len(example), replace = True, weights = example['prob3'])
example3 = example3.iloc[:,0:5]
X_train = example3.iloc[0:len(iris),0:4]
y_train = example3.iloc[0:len(iris),4]

clf_gini = DecisionTreeClassifier(criterion = "gini", random_state = 100, max_depth=1)
clf = clf_gini.fit(X_train, y_train)

#adding a column pred3 after the third round of boosting
y_pred = clf_gini.predict(example.iloc[0:len(iris),0:4])
example['pred3'] = y_pred
```

In [38]:
```
#plotting tree for round 3 boosting
tree.plot_tree(clf)
```

Out[38]:
```
[Text(167.4, 163.07999999999998, 'X[3] <= 1.75\ngini = 0.484\nsamples = 100\nvalue = [41, 59]'),
 Text(83.7, 54.360000000000014, 'gini = 0.357\nsamples = 73\nvalue = [17, 56]'),
 Text(251.10000000000002, 54.360000000000014, 'gini = 0.198\nsamples = 27\nvalue = [24, 3]')]
```



In [39]:
```
#adding a field misclassified3
example.loc[example.Label != example.pred3, 'misclassified3'] = 1
example.loc[example.Label == example.pred3, 'misclassified3'] = 0
```

In [41]:
```
#weighted error calculation
e3 = sum(example['misclassified3'] * example['prob3']) #/len(example)
e3
```

Out[41]: 0.17660000000000003

In [42]:
```
#calculation of performance(alpha)
alpha3 = 0.5*log((1-e3)/e3)
```

In [43]:
```
#update weight
new_weight = example['prob3']*np.exp(-1*alpha3*example['Label']*example['pred3'])
z = sum(new_weight)
normalized_weight = new_weight/sum(new_weight)
```

In [44]:
```
example['prob4'] = round(normalized_weight,4)
```

In [45]:
```
example
```

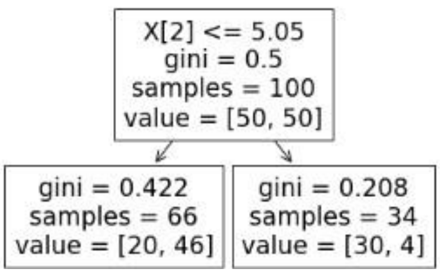
Out[45]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1	misclassified	prob2	pred2	misclassified2	prob3	pred3	misclassified3
50	7.0	3.2	4.7	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
51	6.4	3.2	4.5	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
52	6.9	3.1	4.9	1.5	1	0.01	-1	1.0	0.0714	1	0.0	0.0396	1	0.0
53	5.5	2.3	4.0	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
54	6.5	2.8	4.6	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
55	5.7	2.8	4.5	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
56	6.3	3.3	4.7	1.6	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
57	4.9	2.4	3.3	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
58	6.6	2.9	4.6	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
59	5.2	2.7	3.9	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
60	5.0	2.0	3.5	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0

```
In [46]: #Round 4
         random.seed(40)
         example4 = example.sample(len(example), replace = True, weights = example['prob4'])
         example4 = example4.iloc[:,0:5]
         X_train = example4.iloc[0:len(iris),0:4]
         y_train = example4.iloc[0:len(iris),4]

         clf_gini = DecisionTreeClassifier(criterion = "gini", random_state = 100, max_depth=1)
         clf = clf_gini.fit(X_train, y_train)

         #adding a column pred4 after the fourth round of boosting
         y_pred = clf_gini.predict(example.iloc[0:len(iris),0:4])
         example['pred4'] = y_pred
```

```
In [47]: #plotting tree for round 4 boosting
         tree.plot_tree(clf)
```

```
Out[47]: [Text(167.4, 163.07999999999998, 'X[2] <= 5.05\ngini = 0.5\nsamples = 100\nvalue = [50, 50]'),
          Text(83.7, 54.360000000000014, 'gini = 0.422\nsamples = 66\nvalue = [20, 46]'),
          Text(251.10000000000002, 54.360000000000014, 'gini = 0.208\nsamples = 34\nvalue = [30, 4]')]
```




```
In [48]: #adding a field misclassified4
         example.loc[example.Label != example.pred4, 'misclassified4'] = 1
         example.loc[example.Label == example.pred4, 'misclassified4'] = 0
```

```
In [49]: #error calculation
         e4 = sum(example['misclassified4'] * example['prob4'])
         e4

Out[49]: 0.2705
```

```
In [50]: # calculation of performance (alpha)
         alpha4 = 0.5*log((1-e4)/e4)
```

```
In [51]: #printing the alpha value which is used in each round of boosting
         print(alpha1)
         print(alpha2)
         print(alpha3)
         print(alpha4)

         1.2933446720489712
         1.1075650793336793
         0.7697774105829721
         0.4960436348381521
```

```
In [52]: #final prediction
         t = alpha1 * example['pred1'] + alpha2 * example['pred2'] + alpha3 * example['pred3'] + alpha4 * example['pred4']
```

```
In [53]: #sign of the final prediction
         np.sign(list(t))

Out[53]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                 1.,  1.,  1.,  1.,  1.,  1.,  1., -1.,  1.,  1.,  1.,  1.,  1.,
                 1., -1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1., -1., -1.,
                -1., -1., -1., -1.,  1., -1., -1., -1., -1., -1., -1., -1., -1.,
                -1., -1., -1., -1.,  1., -1., -1., -1., -1., -1., -1., -1., -1.,
                -1.,  1., -1., -1., -1.,  1.,  1., -1., -1., -1., -1., -1., -1.,
                -1., -1., -1., -1., -1., -1., -1., -1., -1.])
```

In [54]:
```
example['final_pred'] = np.sign(list(t))
```

In [55]:
```
example
```

Out[55]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Label	probR1	pred1	misclassified	prob2	pred2	misclassified2	prob3	pred3	misclassified3
50	7.0	3.2	4.7	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
51	6.4	3.2	4.5	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
52	6.9	3.1	4.9	1.5	1	0.01	-1	1.0	0.0714	1	0.0	0.0396	1	0.0
53	5.5	2.3	4.0	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
54	6.5	2.8	4.6	1.5	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
55	5.7	2.8	4.5	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
56	6.3	3.3	4.7	1.6	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
57	4.9	2.4	3.3	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
58	6.6	2.9	4.6	1.3	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
59	5.2	2.7	3.9	1.4	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0
60	5.0	2.0	3.5	1.0	1	0.01	1	0.0	0.0054	1	0.0	0.0030	1	0.0

In [56]:
```
#Confusion matrix
c=confusion_matrix(example['Label'], example['final_pred'])
c
```

Out[56]:
```
array([[45,  5],
       [ 2, 48]], dtype=int64)
```

In [57]:
```
#Overall Accuracy
(c[0,0]+c[1,1])/np.sum(c)*100
```

Out[57]:
```
93.0
```

In [58]:
```
#Fitting the model using the adaboost classifier library
```

In [59]:
```
from sklearn.ensemble import AdaBoostClassifier
```

In [60]:
```
iris = pd.read_csv("iris.csv")
iris = iris.drop('Unnamed: 0', axis=1)
iris = iris[(iris['Species'] == 'versicolor') | (iris['Species'] == 'virginica')]
```

In [61]:
```
#X_train and Y_train split
X_train = iris.iloc[0:len(iris),0:4]
y_train = iris.iloc[0:len(iris),4]
```

In [62]:
```
clf = AdaBoostClassifier(n_estimators=4, random_state=0)
clf.fit(X_train, y_train)
```

Out[62]:
```
AdaBoostClassifier(n_estimators=4, random_state=0)
```

In [63]:
```
clf.predict([[5.5, 2.5, 4.0, 1.3]])
```

Out[63]:
```
array(['versicolor'], dtype=object)
```

In [64]:
```
clf.score(X_train, y_train)
```

Out[64]:
```
0.96
```

1) Soft voting:

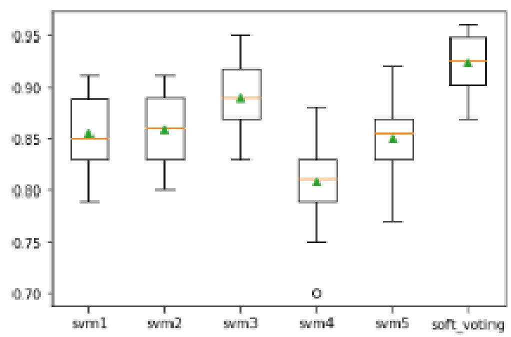
In [1]:
```
# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))
    models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))
    models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))
    models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))
    models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))
    # define the voting ensemble
    ensemble = VotingClassifier(estimators=models, voting='soft')
    return ensemble
```

In [2]:
```
# get a list of models to evaluate
def get_models():
    models = dict()
    models['svm1'] = SVC(probability=True, kernel='poly', degree=1)
    models['svm2'] = SVC(probability=True, kernel='poly', degree=2)
    models['svm3'] = SVC(probability=True, kernel='poly', degree=3)
    models['svm4'] = SVC(probability=True, kernel='poly', degree=4)
    models['svm5'] = SVC(probability=True, kernel='poly', degree=5)
    models['soft_voting'] = get_voting()
    return models
```

In [3]:
```
# compare soft voting ensemble to standalone classifiers
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
    X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=2)
    return X, y

# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))
    models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))
    models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))
    models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))
    models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))
    # define the voting ensemble
    ensemble = VotingClassifier(estimators=models, voting='soft')
    return ensemble

# get a list of models to evaluate
def get_models():
    models = dict()
    models['svm1'] = SVC(probability=True, kernel='poly', degree=1)
    models['svm2'] = SVC(probability=True, kernel='poly', degree=2)
    models['svm3'] = SVC(probability=True, kernel='poly', degree=3)
    models['svm4'] = SVC(probability=True, kernel='poly', degree=4)
    models['svm5'] = SVC(probability=True, kernel='poly', degree=5)
    models['soft_voting'] = get_voting()
    return models
```

```
# evaluate a give model using cross-validation
def evaluate_model(model, X, y):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1, error_score='raise')
    return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model, X, y)
    results.append(scores)
    names.append(name)
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>svm1 0.855 (0.035)
>svm2 0.859 (0.034)
>svm3 0.890 (0.035)
>svm4 0.808 (0.037)
>svm5 0.850 (0.037)
>soft_voting 0.923 (0.027)
```



In [4]:
```
# make a prediction with a soft voting ensemble
from sklearn.datasets import make_classification
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=2)
# define the base models
models = list()
models.append(('svm1', SVC(probability=True, kernel='poly', degree=1)))
models.append(('svm2', SVC(probability=True, kernel='poly', degree=2)))
models.append(('svm3', SVC(probability=True, kernel='poly', degree=3)))
models.append(('svm4', SVC(probability=True, kernel='poly', degree=4)))
models.append(('svm5', SVC(probability=True, kernel='poly', degree=5)))
# define the soft voting ensemble
ensemble = VotingClassifier(estimators=models, voting='soft')
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
data = [[5.88891819,2.64867662,-0.42728226,-1.24988856,-0.00822,-3.57895574,2.87938412,-1.55614691,-0.3816878
yhat = ensemble.predict(data)
print('Predicted Class: %d' % (yhat))
```

```
Predicted Class: 1
```

1) Hard Voting:

In [1]:
```
# test classification dataset
from sklearn.datasets import make_classification
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=2)
# summarize the dataset
print(X.shape, y.shape)
```

```
(1000, 20) (1000,)
```

Voting Ensemble for Classification Hard Voting Ensemble for Classification

In [2]:
```
# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('knn1', KNeighborsClassifier(n_neighbors=1)))
    models.append(('knn3', KNeighborsClassifier(n_neighbors=3)))
    models.append(('knn5', KNeighborsClassifier(n_neighbors=5)))
    models.append(('knn7', KNeighborsClassifier(n_neighbors=7)))
    models.append(('knn9', KNeighborsClassifier(n_neighbors=9)))
    # define the voting ensemble
    ensemble = VotingClassifier(estimators=models, voting='hard')
    return ensemble
```

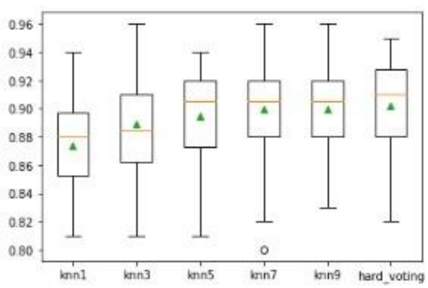
In [3]:
```
# get a list of models to evaluate
def get_models():
    models = dict()
    models['knn1'] = KNeighborsClassifier(n_neighbors=1)
    models['knn3'] = KNeighborsClassifier(n_neighbors=3)
    models['knn5'] = KNeighborsClassifier(n_neighbors=5)
    models['knn7'] = KNeighborsClassifier(n_neighbors=7)
    models['knn9'] = KNeighborsClassifier(n_neighbors=9)
    models['hard_voting'] = get_voting()
    return models
```

In [4]:
```
# evaluate a give model using cross-validation
def evaluate_model(model, X, y):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1, error_score='raise')
    return scores
```

In [5]:
```
# compare hard voting to standalone classifiers
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
    X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=2)
    return X, y

# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('knn1', KNeighborsClassifier(n_neighbors=1)))
    models.append(('knn3', KNeighborsClassifier(n_neighbors=3)))
    models.append(('knn5', KNeighborsClassifier(n_neighbors=5)))
    models.append(('knn7', KNeighborsClassifier(n_neighbors=7)))
    models.append(('knn9', KNeighborsClassifier(n_neighbors=9)))
    # define the voting ensemble
    ensemble = VotingClassifier(estimators=models, voting='hard')
    return ensemble
```

```
# get a list of models to evaluate
def get_models():
    models = dict()
    models['knn1'] = KNeighborsClassifier(n_neighbors=1)
    models['knn3'] = KNeighborsClassifier(n_neighbors=3)
    models['knn5'] = KNeighborsClassifier(n_neighbors=5)
    models['knn7'] = KNeighborsClassifier(n_neighbors=7)
    models['knn9'] = KNeighborsClassifier(n_neighbors=9)
    models['hard_voting'] = get_voting()
    return models

# evaluate a give model using cross-validation
def evaluate_model(model, X, y):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1, error_score='raise')
    return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model, X, y)
    results.append(scores)
    names.append(name)
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>knn1 0.873 (0.030)
>knn3 0.889 (0.038)
>knn5 0.895 (0.031)
>knn7 0.899 (0.035)
>knn9 0.900 (0.033)
>hard_voting 0.902 (0.034)
```



In [7]:

```
# make a prediction with a hard voting ensemble
from sklearn.datasets import make_classification
from sklearn.ensemble import VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
# define dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redundant=5, random_state=2)
# define the base models
models = list()
models.append(('knn1', KNeighborsClassifier(n_neighbors=1)))
models.append(('knn3', KNeighborsClassifier(n_neighbors=3)))
models.append(('knn5', KNeighborsClassifier(n_neighbors=5)))
models.append(('knn7', KNeighborsClassifier(n_neighbors=7)))
models.append(('knn9', KNeighborsClassifier(n_neighbors=9)))
# define the hard voting ensemble
ensemble = VotingClassifier(estimators=models, voting='hard')
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
data = [[5.88891819,2.64867662,-0.42728226,-1.24988856,-0.00822,-3.57895574,2.87938412,-1.55614691,-0.38168784,7.50285659,-1.
yhat = ensemble.predict(data)
print('Predicted Class: %d' % (yhat))
```

```
Predicted Class: 1
```

Voting Regression:

In [1]:
```
# test regression dataset
from sklearn.datasets import make_regression
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1, random_state=1)
# summarize the dataset
print(X.shape, y.shape)
```

```
(1000, 20) (1000,)
```

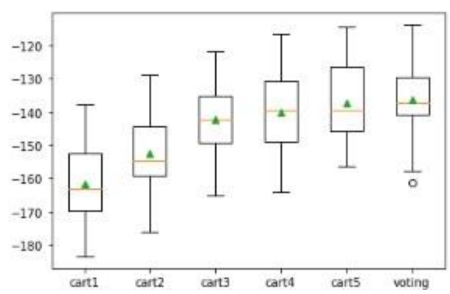
In [2]:
```
# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('cart1', DecisionTreeRegressor(max_depth=1)))
    models.append(('cart2', DecisionTreeRegressor(max_depth=2)))
    models.append(('cart3', DecisionTreeRegressor(max_depth=3)))
    models.append(('cart4', DecisionTreeRegressor(max_depth=4)))
    models.append(('cart5', DecisionTreeRegressor(max_depth=5)))
    # define the voting ensemble
    ensemble = VotingRegressor(estimators=models)
    return ensemble
```

In [3]:
```
# get a list of models to evaluate
def get_models():
    models = dict()
    models['cart1'] = DecisionTreeRegressor(max_depth=1)
    models['cart2'] = DecisionTreeRegressor(max_depth=2)
    models['cart3'] = DecisionTreeRegressor(max_depth=3)
    models['cart4'] = DecisionTreeRegressor(max_depth=4)
    models['cart5'] = DecisionTreeRegressor(max_depth=5)
    models['voting'] = get_voting()
    return models
```

```
In [4]: # compare voting ensemble to each standalone models for regression
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
from matplotlib import pyplot

# get the dataset
def get_dataset():
    X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1, random_state=1)
    return X, y

# get a voting ensemble of models
def get_voting():
    # define the base models
    models = list()
    models.append(('cart1', DecisionTreeRegressor(max_depth=1)))
    models.append(('cart2', DecisionTreeRegressor(max_depth=2)))
    models.append(('cart3', DecisionTreeRegressor(max_depth=3)))
    models.append(('cart4', DecisionTreeRegressor(max_depth=4)))
    models.append(('cart5', DecisionTreeRegressor(max_depth=5)))
    # define the voting ensemble
    ensemble = VotingRegressor(estimators=models)
    return ensemble

# get a list of models to evaluate
def get_models():
    models = dict()
    models['cart1'] = DecisionTreeRegressor(max_depth=1)
    models['cart2'] = DecisionTreeRegressor(max_depth=2)
    models['cart3'] = DecisionTreeRegressor(max_depth=3)
    models['cart4'] = DecisionTreeRegressor(max_depth=4)
    models['cart5'] = DecisionTreeRegressor(max_depth=5)
    models['voting'] = get_voting()
    return models
```

```
# evaluate a give model using cross-validation
def evaluate_model(model, X, y):
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error', cv=cv, n_jobs=-1, error_score='raise')
    return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model, X, y)
    results.append(scores)
    names.append(name)
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>cart1 -161.519 (11.414)
>cart2 -152.596 (11.271)
>cart3 -142.378 (10.900)
>cart4 -140.086 (12.469)
>cart5 -137.145 (12.222)
>voting -136.347 (11.231)
```



```
In [5]: # make a prediction with a voting ensemble
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import VotingRegressor
# define dataset
X, y = make_regression(n_samples=1000, n_features=20, n_informative=15, noise=0.1, random_state=1)
# define the base models
models = list()
models.append(('cart1', DecisionTreeRegressor(max_depth=1)))
models.append(('cart2', DecisionTreeRegressor(max_depth=2)))
models.append(('cart3', DecisionTreeRegressor(max_depth=3)))
models.append(('cart4', DecisionTreeRegressor(max_depth=4)))
models.append(('cart5', DecisionTreeRegressor(max_depth=5)))
# define the voting ensemble
ensemble = VotingRegressor(estimators=models)
# fit the model on all available data
ensemble.fit(X, y)
# make a prediction for one example
data = [[0.59332206,-0.56637507,1.34808718,-0.57054047,-0.72480487,1.05648449,0.77744852,0.07361796,0.88398267,2.02843157,1.0190
yhat = ensemble.predict(data)
print('Predicted Value: %.3f' % (yhat))

Predicted Value: 141.319
```

1) Gradient Boosting:

```
In [1]: def gradient_descent(gradient, start, learn_rate, n_iter):
            vector = start
            for _ in range(n_iter):
                diff = -learn_rate * gradient(vector)
                vector += diff
            return vector
```

```
In [2]: import numpy as np

        def gradient_descent(
            gradient, start, learn_rate, n_iter=50, tolerance=1e-06
        ):
            vector = start
            for _ in range(n_iter):
                diff = -learn_rate * gradient(vector)
                if np.all(np.abs(diff) <= tolerance):
                    break
                vector += diff
            return vector
```

```
In [3]:  gradient_descent(
        ...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.2
        ... )

Out[3]: 2.210739197207331e-06
```

```
In [4]: gradient_descent(
        ...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.8
        ... )

Out[4]: -4.77519666596786e-07
```

```
In [5]: gradient_descent(
        ...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005
        ... )

Out[5]: 6.050060671375367
```

In [6]:
```
gradient_descent(
...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005,
...      n_iter=100
... )
3.660323412732294
>>> gradient_descent(
...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005,
...      n_iter=1000
... )
0.0004317124741065828
>>> gradient_descent(
...      gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005,
...      n_iter=2000
... )
```

Out[6]: 9.952518849647663e-05

In [7]:
```
gradient_descent(
...      gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,
...      learn_rate=0.2
... )
```

Out[7]: -1.4207567437458342

In [8]:
```
gradient_descent(
...      gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,
...      learn_rate=0.1
... )
```

Out[8]: 1.285401330315467

Artificial Intelligence & Machine LearningExperiment No. 12

Deployment of Machine Learning Models

PRACTICAL NO 12

Aim: Deployment of Machine Learning Models.

Objective: To learn Deployment of Machine Learning Models.

Software Requirement:

- **Spyder (Anaconda3):** Spyder, the Scientific Python Development Environment, is **a free integrated development environment (IDE) that is included with Anaconda**. It includes editing, interactive testing, debugging, and introspection features.
  It features a unique combination of the advanced editing, analysis, debugging and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection and beautiful visualization capabilities of a scientific package.

1) Deployment:

app.py:

```
import numpy as np
from flask import Flask, request, jsonify, render_template
import pickle

app = Flask(__name__)
model = pickle.load(open('model.pkl', 'rb'))

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/predict',methods=['POST'])
def predict():
    '''
    For rendering results on HTML GUI
    '''
```

```
    int_features = [int(x) for x in request.form.values()]
    final_features = [np.array(int_features)]
    prediction = model.predict(final_features)

    output = round(prediction[0], 2)

    return render_template('index.html', prediction_text='Employee Salary
should be $ {}'.format(output))

@app.route('/predict_api',methods=['POST'])
def predict_api():
    '''
    For direct API calls trought request
    '''
    data = request.get_json(force=True)
    prediction = model.predict([np.array(list(data.values()))])

    output = prediction[0]
    return jsonify(output)

if __name___== "_main_":
    app.run(debug=True)
```

model.py

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pickle

dataset = pd.read_csv('hiring.csv')

dataset['experience'].fillna(0, inplace=True)

dataset['test_score'].fillna(dataset['test_score'].mean(), inplace=True)

X = dataset.iloc[:, :3]

#Converting words to integer values
def convert_to_int(word):
```

```
    word_dict = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5, 'six':6, 'seven':7, 'eight':8,
                'nine':9, 'ten':10, 'eleven':11, 'twelve':12, 'zero':0, 0: 0}
    return word_dict[word]

X['experience'] = X['experience'].apply(lambda x : convert_to_int(x))

y = dataset.iloc[:, -1]

#Splitting Training and Test Set
#Since we have a very small dataset, we will train our model with all availabe data.

from sklearn.linear_model import LinearRegression
regressor = LinearRegression()

#Fitting model with trainig data
regressor.fit(X, y)

# Saving model  to  disk
pickle.dump(regressor, open('model.pkl','wb'))

# Loading model to compare the results
model = pickle.load(open('model.pkl','rb'))
print(model.predict([[2, 9, 6]]))
```

request.py

```
import requests


url = 'http://localhost:5000/predict_api'

r = requests.post(url,json={'experience':2, 'test_score':9, 'interview_score':6})


print(r.json())
```

index.html

```
<!DOCTYPE html>
<html >
<!--From https://codepen.io/frytyler/pen/EGdtg-->
<head>
  <meta charset="UTF-8">
  <title>ML API</title>
  <link href='https://fonts.googleapis.com/css?family=Pacifico' rel='stylesheet'
type='text/css'>
<link href='https://fonts.googleapis.com/css?family=Arimo' rel='stylesheet'
type='text/css'>
<link href='https://fonts.googleapis.com/css?family=Hind:300' rel='stylesheet'
type='text/css'>
<link
href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300'
rel='stylesheet' type='text/css'>
<link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">

</head>

<body>
<div class="login">
<h1>Predict Salary Analysis</h1>

     <!-- Main Input For Receiving Query to our ML -->
    <form action="{{ url_for('predict')}}"method="post">
       <input type="text" name="experience" placeholder="Experience"
required="required" />
```

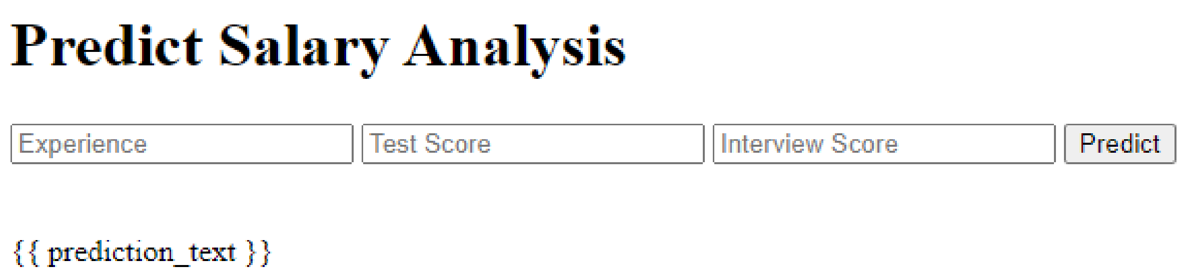
```
        <input type="text" name="test_score" placeholder="Test Score"
required="required" />
       <input type="text" name="interview_score" placeholder="Interview Score"
required="required" />

        <button type="submit" class="btn btn-primary btn-block btn-
large">Predict</button>
    </form>

   <br>
   <br>
   {{ prediction_text }}

 </div>
</body>
</html>
```

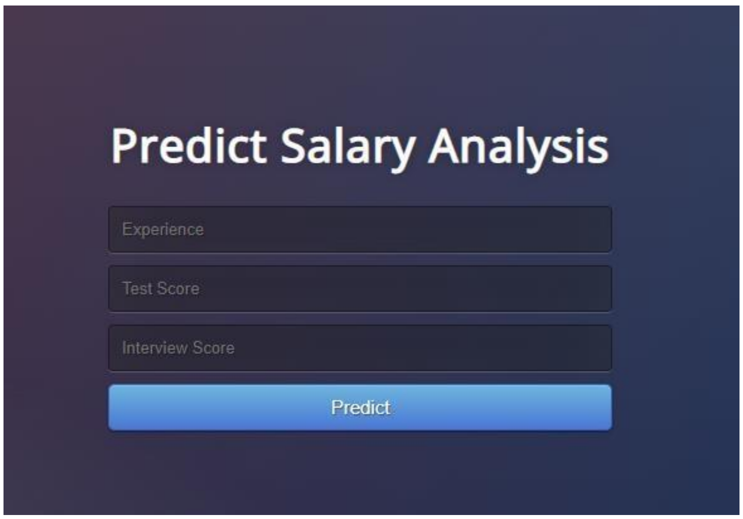
OUTPUT:

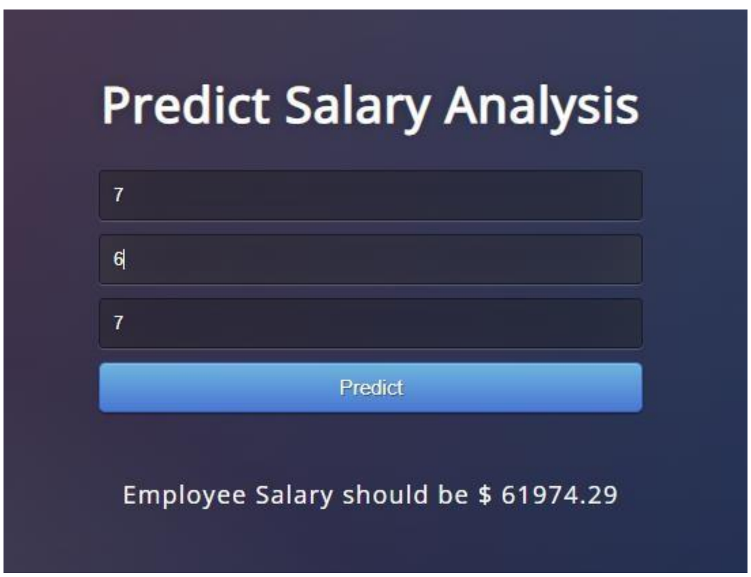
On Anaconda Prompt:

```
(base) C:\Users\admin\Desktop\40_AIML_Pract\New folder\PRACTICAL-9\Deployment\Deployment>python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 170-982-676
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Index.html


Predict Salary Analysis
Experience
Test Score
Interview Score
Predict
{{ prediction_text }}


Predict Salary Analysis
Experience
Test Score
Interview Score
Predict


Predict Salary Analysis
7
6
7
Predict
Employee Salary should be $ 61974.29

2)IR_Project:

app.py

```
import numpy as np
from flask import Flask, request, jsonify, render_template
import pickle
model = pickle.load(open('model.pkl', 'rb'))
app = Flask(_name_)
@app.route('/')
def home():
    return render_template('index.html')
@app.route('/predict',methods=['POST'])
def predict():
  '''
  For rendering results on HTML GUI
  '''
  int_features = [float(x) for x in request.form.values()]

  final_features = [np.array(int_features)]
  prediction = model.predict(final_features)

  output =prediction[0]
  return render_template('index.html', prediction_text='The Flower is
{}'.format(output))
if __name___== "_main_":
  app.run(debug=True)
```

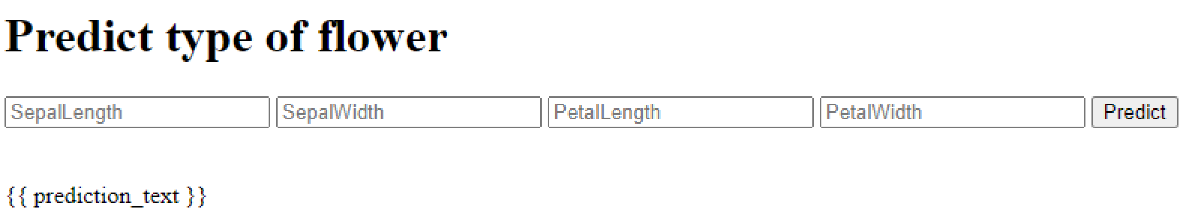
model.py

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import pickle
data=pd.read_csv('iris.csv')
# X = feature values, all the columns except the last column
X = data.iloc[:, :-1]
# y = target values, last column of the data frame
y = data.iloc[:, -1]
#Split the data into 80% training and 20% testing
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
#Train the model
model = LogisticRegression()
model.fit(x_train, y_train) #Training the model
#Test the model
predictions = model.predict(x_test)
print( classification_report(y_test, predictions) )
print( accuracy_score(y_test, predictions))
pickle.dump(model,open('model.pkl','wb'))
p=model.predict([[5.1,3.5,1.4,0.2]])print(p[0])
```

index.html

```
<!DOCTYPE html>
<html >
<head>
<meta charset="UTF-8">
<title>ML API</title>
</head>
<body>
<div class="login">
<h1>Predict type of flower</h1>
<!-- Main Input For Receiving Query to our ML -->
<form action="{{ url_for('predict')}}"method="post">
<input type="text" name="SepalLength" placeholder="SepalLength" required="required" />
<input type="text" name="SepalWidth" placeholder="SepalWidth" required="required" />
<input type="text" name="PetalLength" placeholder="PetalLength" required="required" />
<input type="text" name="PetalWidth" placeholder="PetalWidth" required="required" />
<button type="submit" class="btn btn-primary btn-block btn-large">Predict</button>
</form>
<br>
<br>
{{ prediction_text }}
</div>
</body>
</html>
```

Index.html



On Anaconda Prompt:

```
(base) C:\Users\admin\Desktop\40_AIML_Pract\New folder\PRACTICAL-9\IR_PROJECT>python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 170-982-676
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

OUTPUT: