

1. Program to show the edge detection using python programming

1. Prewitt Edge Detection

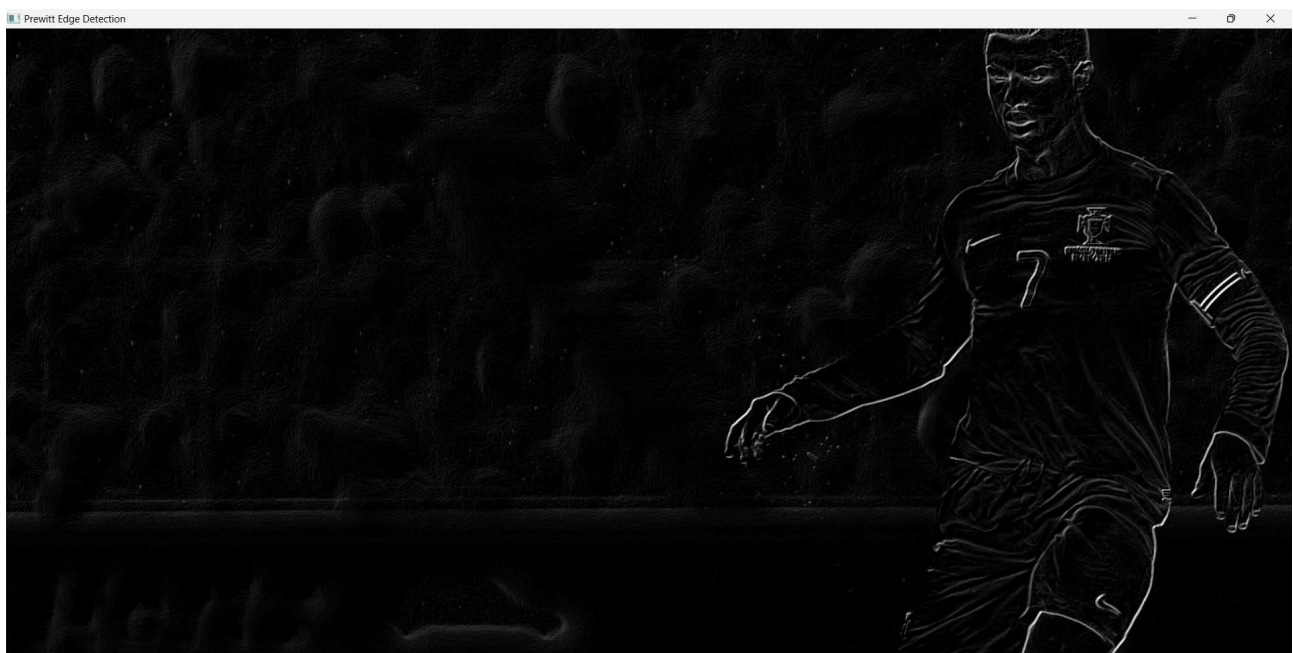
```
import cv2
import numpy as np

img = cv2.imread('cr7.jpg', 0)

prewittx = cv2.filter2D(img, -1, np.array([[ -1, 0, 1], [ -1, 0, 1], [ -1, 0, 1]]))
prewitty = cv2.filter2D(img, -1, np.array([[ -1, -1, -1], [ 0, 0, 0], [ 1, 1, 1]]))
prewitt = cv2.addWeighted(prewittx, 0.5, prewitty, 0.5, 0)

cv2.imshow('Prewitt Edge Detection', prewitt)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



2.Laplacian Edge Detection

```
import cv2
import numpy as np

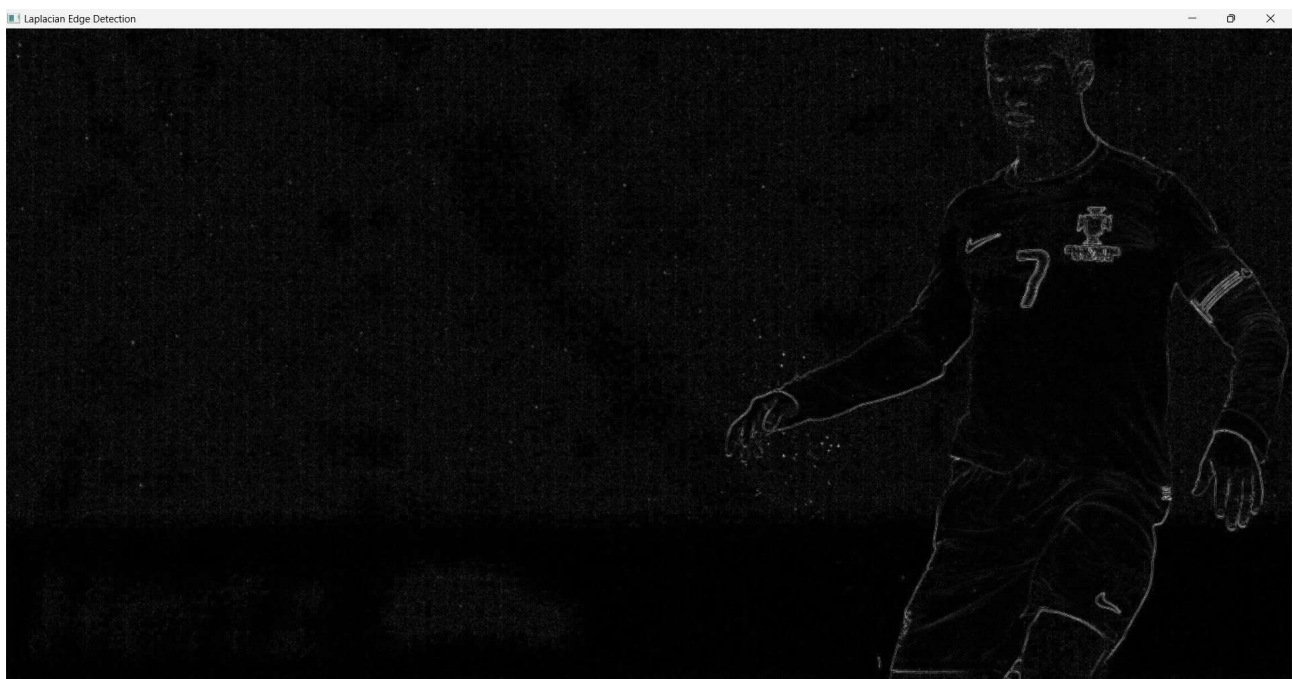
img = cv2.imread('cr7.jpg', 0)

laplacian = cv2.Laplacian(img, cv2.CV_64F)

laplacian = np.uint8(np.absolute(laplacian))

cv2.imshow('Laplacian Edge Detection', laplacian)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



3.Canny Edge Program

```
import cv2

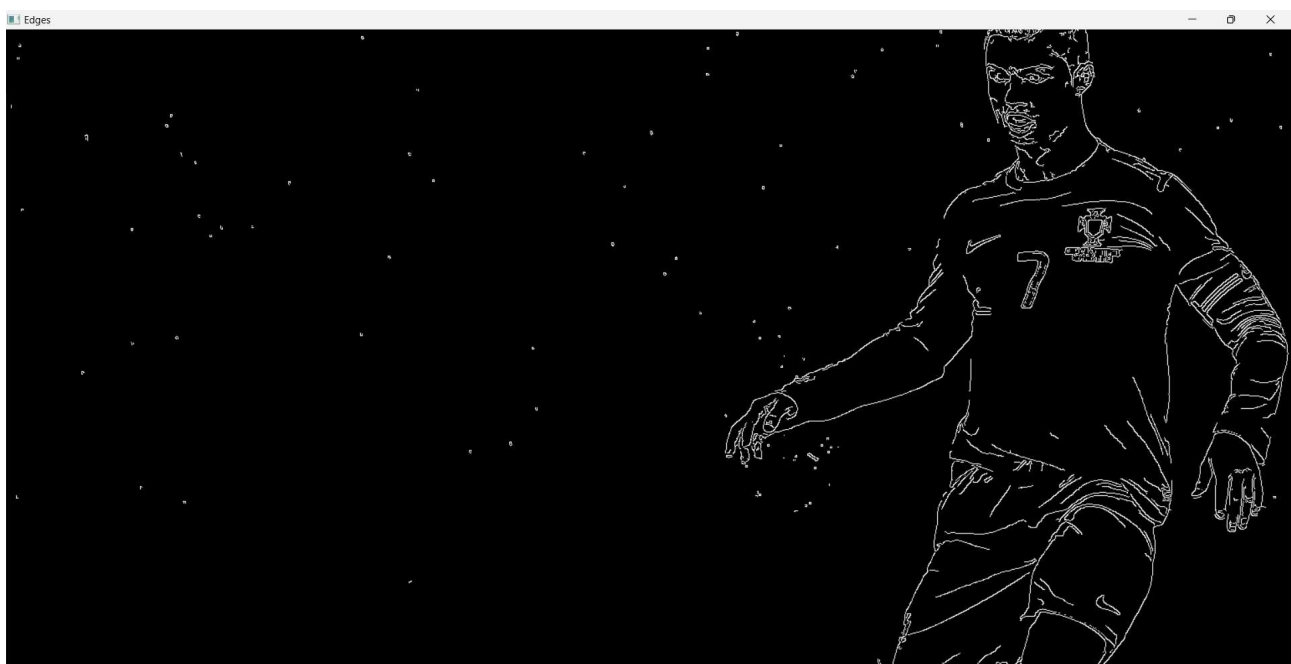
img = cv2.imread('cr7.jpg', 0)

edges = cv2.Canny(img, 100, 200)

cv2.imshow('Original', img)
cv2.imshow('Edges', edges)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



2. Program to implement decision tree algorithm.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.metrics import accuracy_score

iris = load_iris()

data = pd.DataFrame(data = iris.data, columns = iris.feature_names)

data['Species'] = iris.target

target = np.unique(iris.target)
target_n = np.unique(iris.target_names)
target_dict = dict(zip(target, target_n))
data['Species'] = data['Species'].replace(target_dict)

x = data.drop(columns = "Species")
y = data["Species"]
names_features = x.columns
target_labels = y.unique()

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 93)

from sklearn.tree import DecisionTreeClassifier

dtc = DecisionTreeClassifier(max_depth = 3, random_state = 93)

dtc.fit(x_train, y_train)

plt.figure(figsize = (30, 10), facecolor = 'b')
Tree = tree.plot_tree(dtc, feature_names = names_features, class_names = target_labels, rounded =
True, filled = True, fontsize = 14)
plt.show()
y_pred = dtc.predict(x_test)
```

```

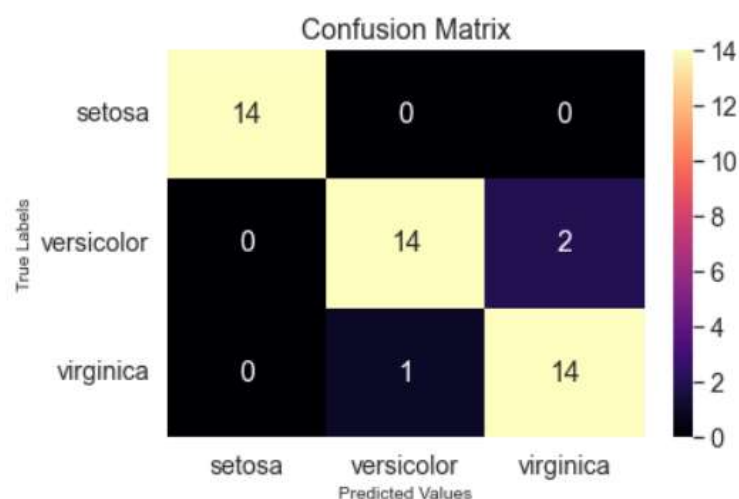
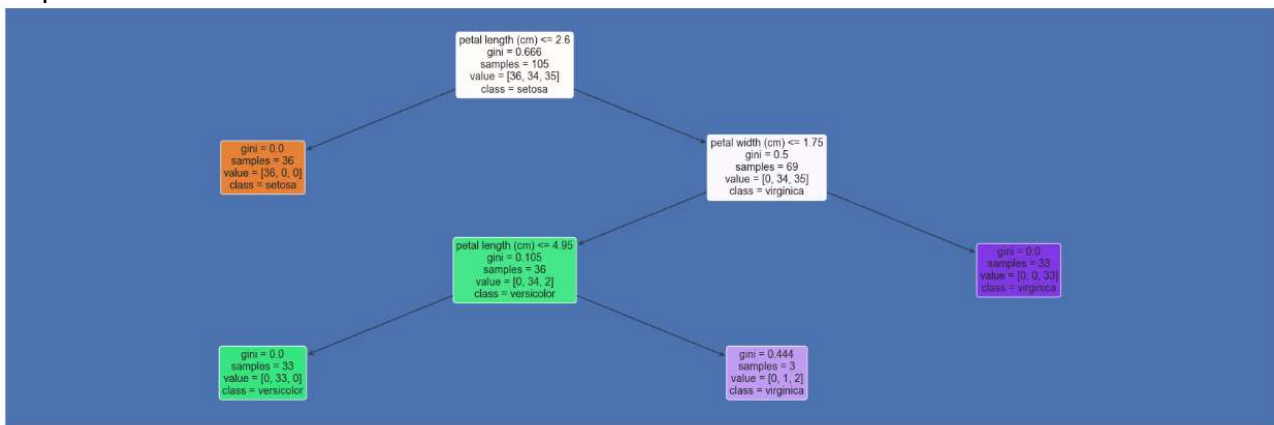
confusion_matrix = metrics.confusion_matrix(y_test, y_pred)
matrix = pd.DataFrame(confusion_matrix)
axis = plt.axes()
sns.set(font_scale = 1.3)
plt.figure(figsize = (10,7))

sns.heatmap(matrix, annot = True, fmt = "g", ax = axis, cmap = "magma")
axis.set_title('Confusion Matrix')
axis.set_xlabel("Predicted Values", fontsize = 10)
axis.set_xticklabels([""] + target_labels)
axis.set_ylabel("True Labels", fontsize = 10)
axis.set_yticklabels(list(target_labels), rotation = 0)
plt.show()

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

```

Output:



<Figure size 720x504 with 0 Axes>

Accuracy: 0.9333333333333333

3.Program to implement Find S algorithm.

```
import numpy as np
import pandas as pd

data = pd.read_csv('enjoysport.csv')
concepts = np.array(data.iloc[:,0:-1])
print("\nInstances are:\n",concepts)

target = np.array(data.iloc[:,-1])
print("\nTarget Values are: ",target)

def learn(concepts, target):
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and generic_h")
    print("\nSpecific Boundary: ", specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print("\nGeneric Boundary: ",general_h)

    for i, h in enumerate(concepts):
        print("\nInstance", i+1 , "is ", h)
        if target[i] == "yes":
            print("Instance is Positive ")
            for x in range(len(specific_h)):
                if h[x]!= specific_h[x]:
                    specific_h[x] ='?'
                    general_h[x][x] ='?'
        if target[i] == "no":
            print("Instance is Negative ")
            for x in range(len(specific_h)):
                if h[x]!= specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

    print("Specific Boundary after ", i+1, "Instance is ", specific_h)
    print("Generic Boundary after ", i+1, "Instance is ", general_h)
    print("\n")

    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    return specific_h, general_h

s_final, g_final = learn(concepts, target)
print("Final Specific_h: ", s_final, sep="\n")
print("Final General_h: ", g_final, sep="\n")
```

Instances are:

```
[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']  
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']  
['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']  
['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]
```

Target Values are: [1 1 0 1]

Initialization of specific_h and generic_h

Specific Boundary: ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Generic Boundary: [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 1 is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Specific Boundary after 1 Instance is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Generic Boundary after 1 Instance is [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 2 is ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']

Specific Boundary after 2 Instance is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Generic Boundary after 2 Instance is []

Instance 3 is ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']

Specific Boundary after 3 Instance is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Generic Boundary after 3 Instance is []

Instance 4 is ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']

Specific Boundary after 4 Instance is ['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Generic Boundary after 4 Instance is []

Final Specific_h:

['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Final General_h:

[]

4.Program to implement K-means clustering.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

X = np.random.rand(100, 2)
wcss = []

for i in range(1, 11):

    kmeans = KMeans(n_clusters=i)

    kmeans.fit(X)

    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()

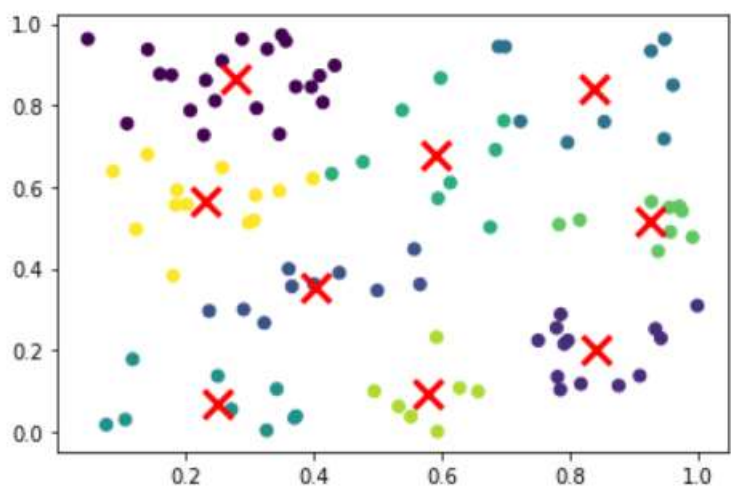
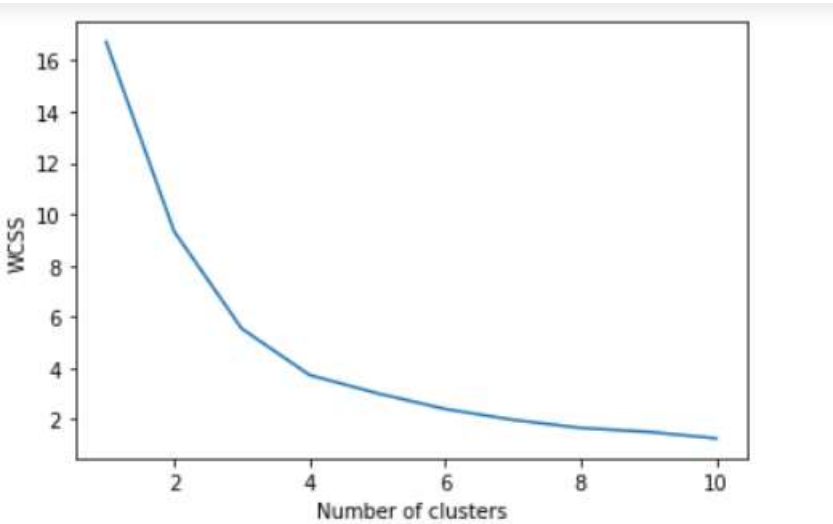
k = np.argmin(np.diff(wcss, 2)) + 2
kmeans = KMeans(n_clusters=k)

kmeans.fit(X)

centers = kmeans.cluster_centers_
labels = kmeans.labels_

plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(centers[:, 0], centers[:, 1], marker='x', s=200, linewidths=3, color='r')
plt.show()
```


Output:



5.Program to implement Independent Component Analysis.

```
import numpy as np
np.random.seed(0)
from scipy import signal
from scipy.io import wavfile
from matplotlib import pyplot as plt
import seaborn as sns
sns.set(rc={'figure.figsize':(11.7,8.27)})
def g(x):
    return np.tanh(x)
def g_der(x):
    return 1 - g(x) * g(x)
def center(X):
    X = np.array(X)
    mean = X.mean(axis=1, keepdims=True)
    return X- mean
def whitening(X):
    cov = np.cov(X)
    d, E = np.linalg.eigh(cov)
    D = np.diag(d)
    D_inv = np.sqrt(np.linalg.inv(D))
    X_whiten = np.dot(E, np.dot(D_inv, np.dot(E.T, X)))
    return X_whiten
def calculate_new_w(w, X):
    w_new = (X * g(np.dot(w.T, X))).mean(axis=1) - g_der(np.dot(w.T, X)).mean() * w
    w_new /= np.sqrt((w_new ** 2).sum())
    return w_new
def ica(X, iterations, tolerance=1e-5):
    X = center(X)
    X = whitening(X)
    components_nr = X.shape[0]
    W = np.zeros((components_nr, components_nr), dtype=X.dtype)
    for i in range(components_nr):
        w = np.random.rand(components_nr)
        for j in range(iterations):
            w_new = calculate_new_w(w, X)
            if i >= 1:
                w_new -= np.dot(np.dot(w_new, W[:i].T), W[:i])
                distance = np.abs(np.abs((w * w_new).sum()) - 1)
                w = w_new
            if distance < tolerance:
                break
        W[i, :] = w
    S = np.dot(W, X)
    return S
def plot_mixture_sources_predictions(X, original_sources, S):
    fig = plt.figure()
    plt.subplot(3, 1, 1)
    for x in X:
        plt.plot(x)
```

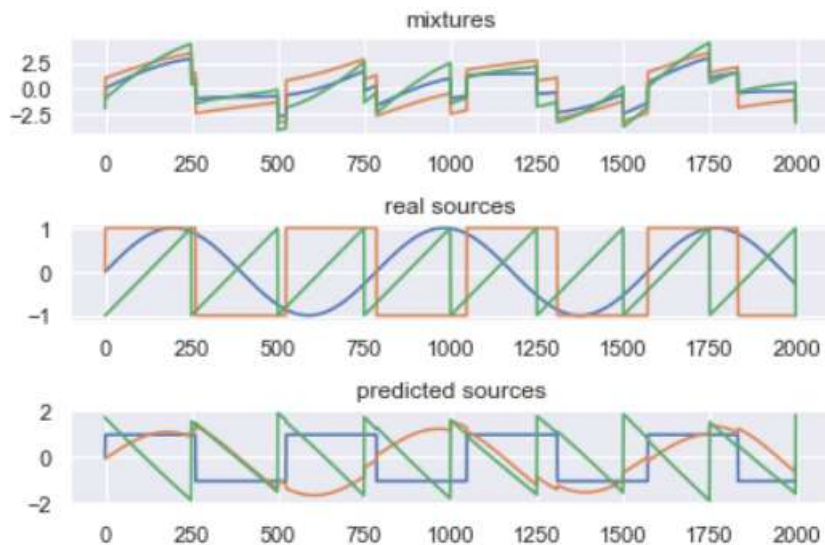
```

plt.title("mixtures")
plt.subplot(3, 1, 2)
for s in original_sources:
    plt.plot(s)
plt.title("real sources")
plt.subplot(3,1,3)
for s in S:
    plt.plot(s)
plt.title("predicted sources")
fig.tight_layout()
plt.show()
def mix_sources(mixtures, apply_noise=False):
    for i in range(len(mixtures)):
        max_val = np.max(mixtures[i])
        if max_val > 1 or np.min(mixtures[i]) < -1:
            mixtures[i] = mixtures[i] / (max_val / 2) - 0.5
    X = np.c_[[mix for mix in mixtures]]
    if apply_noise:
        X += 0.02 * np.random.normal(size=X.shape)
    return X
n_samples = 2000
time = np.linspace(0, 8, n_samples)
s1 = np.sin(2 * time) # sinusoidal
s2 = np.sign(np.sin(3 * time)) # square signal
s3 = signal.sawtooth(2 * np.pi * time) # saw tooth signal

X = np.c_[s1, s2, s3]
A = np.array([1, 1, 1], [0.5, 2, 1.0], [1.5, 1.0, 2.0]))
X = np.dot(X, A.T)
X = X.T
S = ica(X, iterations=1000)
plot_mixture_sources_predictions(X, [s1, s2, s3], S)

```

Output:



6. Program to implement Special Clustering.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import SpectralClustering
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score

X = pd.read_csv('CC GENERAL.csv')

X = X.drop('CUST_ID', axis = 1)

X.fillna(method = 'ffill', inplace = True)

X.head()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_normalized = normalize(X_scaled)

X_normalized = pd.DataFrame(X_normalized)

pca = PCA(n_components = 2)
X_principal = pca.fit_transform(X_normalized)
X_principal = pd.DataFrame(X_principal)
X_principal.columns = ['P1', 'P2']
X_principal.head()

spectral_model_rbf = SpectralClustering(n_clusters = 2, affinity = 'rbf')

labels_rbf = spectral_model_rbf.fit_predict(X_principal)

colours = {}
colours[0] = 'b'
colours[1] = 'y'

cvec = [colours[label] for label in labels_rbf]

b = plt.scatter(X_principal['P1'], X_principal['P2'], color = 'b');
y = plt.scatter(X_principal['P1'], X_principal['P2'], color = 'y');
```

```
plt.figure(figsize =(9, 9))
plt.scatter(X_principal['P1'], X_principal['P2'], c = cvec)
plt.legend((b, y), ('Label 0', 'Label 1'))
plt.show()
```

```
spectral_model_nn = SpectralClustering(n_clusters = 2, affinity ='nearest_neighbors')
```

```
labels_nn = spectral_model_nn.fit_predict(X_principal)
# List of different values of affinity
affinity = ['rbf', 'nearest-neighbours']
```

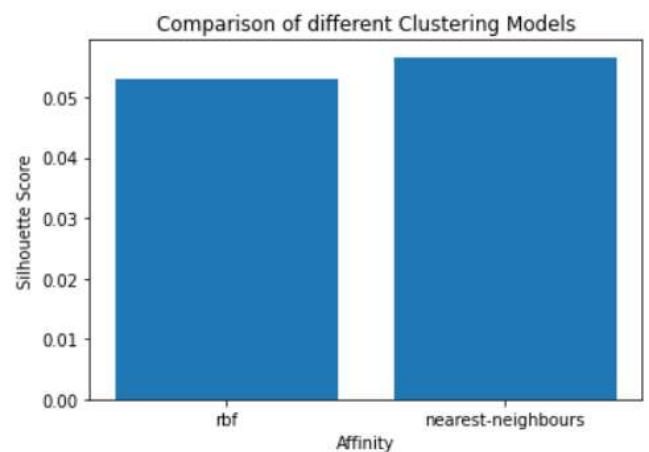
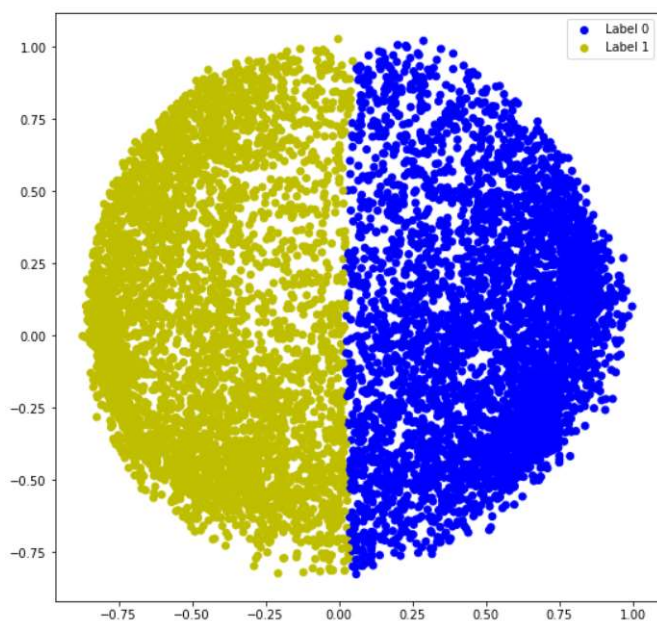
```
s_scores = []
```

```
s_scores.append(silhouette_score(X, labels_rbf))
s_scores.append(silhouette_score(X, labels_nn))
```

```
print(s_scores)
```

```
plt.bar(affinity, s_scores)
plt.xlabel('Affinity')
plt.ylabel('Silhouette Score')
plt.title('Comparison of different Clustering Models')
plt.show()
```

Output:



```
[0.05300611480757429, 0.05667039590382262]
```

7. Program to implement confusion matrix.

```
from sklearn.metrics import confusion_matrix, plot_confusion_matrix
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
X, y = make_classification(random_state=0)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

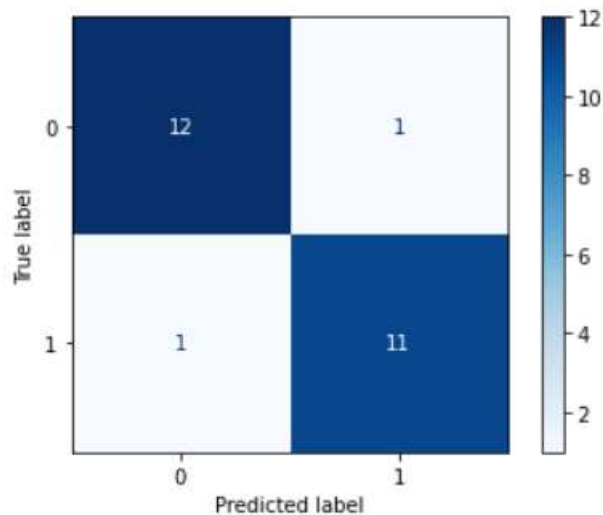
```
clf = LogisticRegression(random_state=0)
clf.fit(X_train, y_train)
```

```
y_pred = clf.predict(X_test)
```

```
conf_matrix = confusion_matrix(y_test, y_pred)
```

```
plot_confusion_matrix(clf, X_test, y_test, cmap=plt.cm.Blues)
plt.show()
```

Output:



8.Program to implement Expectation-Maximization Algorithm(EM)

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.special import logsumexp
from sklearn.mixture import GaussianMixture
from matplotlib import pyplot as plt

def GMM_sklearn(x, weights=None, means=None, covariances=None):
    model = GaussianMixture(n_components=2,
                             covariance_type='full',
                             tol=0.01,
                             max_iter=1000,
                             weights_init=weights,
                             means_init=means,
                             precisions_init=covariances)
    model.fit(x)
    print("\nscikit learn:\n\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1: %s"
          % (model.weights_[1], model.means_[0, :], model.means_[1, :], model.covariances_[0, :],
            model.covariances_[1, :]))
    return model.predict(x), model.predict_proba(x)[: ,1]

def get_random_psd(n):
    x = np.random.normal(0, 1, size=(n, n))
    return np.dot(x, x.transpose())

def initialize_random_params():
    params = {'phi': np.random.uniform(0, 1),
              'mu0': np.random.normal(0, 1, size=(2,)),
              'mu1': np.random.normal(0, 1, size=(2,)),
              'sigma0': get_random_psd(2),
              'sigma1': get_random_psd(2)}
    return params

def learn_params(x_labeled, y_labeled):
    n = x_labeled.shape[0]
    phi = x_labeled[y_labeled == 1].shape[0] / n
    mu0 = np.sum(x_labeled[y_labeled == 0], axis=0) / x_labeled[y_labeled == 0].shape[0]
    mu1 = np.sum(x_labeled[y_labeled == 1], axis=0) / x_labeled[y_labeled == 1].shape[0]
    sigma0 = np.cov(x_labeled[y_labeled == 0].T, bias= True)
    sigma1 = np.cov(x_labeled[y_labeled == 1].T, bias=True)
    return {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}

def e_step(x, params):
    np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x),
```

```

        stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)])
log_p_y_x = np.log([1-params["phi"], params["phi"]])[np.newaxis, ...] + \
        np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(x),
        stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)].T
log_p_y_x_norm = logsumexp(log_p_y_x, axis=1)
return log_p_y_x_norm, np.exp(log_p_y_x - log_p_y_x_norm[..., np.newaxis])

```

```

def m_step(x, params):
    total_count = x.shape[0]
    _, heuristics = e_step(x, params)
    heuristic0 = heuristics[:, 0]
    heuristic1 = heuristics[:, 1]
    sum_heuristic1 = np.sum(heuristic1)
    sum_heuristic0 = np.sum(heuristic0)
    phi = (sum_heuristic1/total_count)
    mu0 = (heuristic0[..., np.newaxis].T.dot(x)/sum_heuristic0).flatten()
    mu1 = (heuristic1[..., np.newaxis].T.dot(x)/sum_heuristic1).flatten()
    diff0 = x - mu0
    sigma0 = diff0.T.dot(diff0 * heuristic0[..., np.newaxis]) / sum_heuristic0
    diff1 = x - mu1
    sigma1 = diff1.T.dot(diff1 * heuristic1[..., np.newaxis]) / sum_heuristic1
    params = {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma1}
    return params

```

```

def get_avg_log_likelihood(x, params):
    loglikelihood, _ = e_step(x, params)
    return np.mean(loglikelihood)

```

```

def run_em(x, params):
    avg_loglikelihoods = []
    while True:
        avg_loglikelihood = get_avg_log_likelihood(x, params)
        avg_loglikelihoods.append(avg_loglikelihood)
        if len(avg_loglikelihoods) > 2 and abs(avg_loglikelihoods[-1] - avg_loglikelihoods[-2]) <
0.0001:
            break
        params = m_step(x_unlabeled, params)
    print("\tphi: %s\n\tmu_0: %s\n\tmu_1: %s\n\tsigma_0: %s\n\tsigma_1: %s"
        % (params['phi'], params['mu0'], params['mu1'], params['sigma0'], params['sigma1']))
    _, posterior = e_step(x_unlabeled, params)
    forecasts = np.argmax(posterior, axis=1)
    return forecasts, posterior, avg_loglikelihoods

```



```

if __name__ == '__main__':
    data_unlabeled = pd.read_csv('unlabeled.csv')
    x_unlabeled = data_unlabeled[["x1", "x2"]].values

    print("unsupervised: ")
    random_params = initialize_random_params()
    unsupervised_forecasts, unsupervised_posterior, unsupervised_loglikelihoods =
run_em(x_unlabeled, random_params)
    print("total steps: ", len(unsupervised_loglikelihoods))
    plt.plot(unsupervised_loglikelihoods)
    plt.title("unsupervised log likelihoods")
    plt.savefig("unsupervised.png")
    plt.close()

    print("\nsemi-supervised: ")
    data_labeled = pd.read_csv('labeled.csv')
    x_labeled = data_labeled[["x1", "x2"]].values
    y_labeled = data_labeled["y"].values
    learned_params = learn_params(x_labeled, y_labeled)
    semisupervised_forecasts, semisupervised_posterior, semisupervised_loglikelihoods =
run_em(x_unlabeled, learned_params)
    print("total steps: ", len(semisupervised_loglikelihoods))
    plt.plot(semisupervised_loglikelihoods)
    plt.title("semi-supervised log likelihoods")
    plt.savefig("semi-supervised.png")

    learned_params = learn_params(x_labeled, y_labeled)
    weights = [1 - learned_params["phi"], learned_params["phi"]]
    means = [learned_params["mu0"], learned_params["mu1"]]
    covariances = [learned_params["sigma0"], learned_params["sigma1"]]
    sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, covariances)

    output_df = pd.DataFrame({'semisupervised_forecasts': semisupervised_forecasts,
                             'semisupervised_posterior': semisupervised_posterior[:, 1],
                             'sklearn_forecasts': sklearn_forecasts,
                             'posterior_sklearn': posterior_sklearn})

    print("\n%s%% of forecasts matched." % (output_df[output_df["semisupervised_forecasts"] ==
output_df["sklearn_forecasts"]].shape[0] / output_df.shape[0] * 100))

```

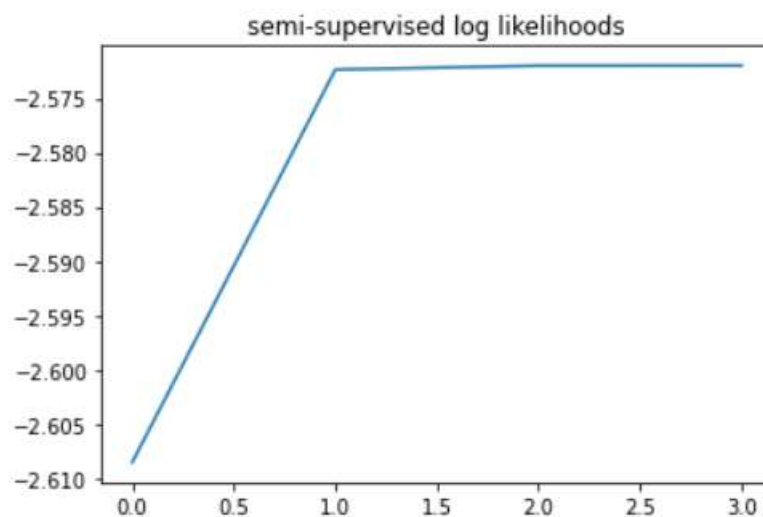
Output:

```
unsupervised:
  phi: 0.5985711658340387
  mu_0: [-1.06906401 -1.06927 ]
  mu_1: [0.96194764 0.98361038]
  sigma_0: [[0.34296075 0.28252623]
 [0.28252623 0.70705511]]
  sigma_1: [[0.74290629 0.15800099]
 [0.15800099 0.31664651]]
total steps: 24
```

```
semi-supervised:
  phi: 0.5863498817945461
  mu_0: [-1.04546727 -1.02704636]
  mu_1: [0.98763329 0.99661118]
  sigma_0: [[0.36018609 0.30853357]
 [0.30853357 0.75384027]]
  sigma_1: [[0.7196797 0.1437903 ]
 [0.1437903 0.30853791]]
total steps: 4
```

```
scikit learn:
  phi: 0.59647894226803
  mu_0: [-1.06169376 -1.0563389 ]
  mu_1: [0.96408565 0.98206315]
  sigma_0: [[0.35027155 0.29629092]
 [0.29629092 0.73083581]]
  sigma_1: [[0.74510804 0.16156928]
 [0.16156928 0.32021029]]
```

99.4% of forecasts matched.



9. Program to implement Q-learning

```
import numpy as np
```

```
R = np.array([[-1, -1, -1, -1, 0, -1],
              [-1, -1, -1, 0, -1, 100],
              [-1, -1, -1, 0, -1, -1],
              [-1, 0, 0, -1, 0, -1],
              [0, -1, -1, 0, -1, 100],
              [-1, 0, -1, -1, 0, 100]])
```

```
Q = np.zeros_like(R)
```

```
alpha = 0.8
gamma = 0.95
n_episodes = 1000
```

```
for episode in range(n_episodes):
    state = np.random.randint(0, 6)
    while state != 5:
        action = np.random.choice(np.where(R[state, :] != -1)[0])
        next_state = action
        Q[state, action] = Q[state, action] + alpha * (R[state, action] + gamma *
np.max(Q[next_state, :]) - Q[state, action])
        state = next_state
```

```
print(Q)
```

Output:

```
[[ 0  0  0  0 93  0]
 [ 0  0  0 88  0 99]
 [ 0  0  0 88  0  0]
 [ 0 93 83  0 93  0]
 [88  0  0 88  0 99]
 [ 0  0  0  0  0  0]]
```

10. Program to implement Genetic algorithm.

```
import numpy
import ga

equation_inputs = [4,-2,3.5,5,-11,-4.7]

num_weights = len(equation_inputs)

sol_per_pop = 8
num_parents_mating = 4

pop_size = (sol_per_pop,num_weights)
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)

best_outputs = []
num_generations = 1000
for generation in range(num_generations):
    print("Generation : ", generation)

    fitness = ga.cal_pop_fitness(equation_inputs, new_population)
    print("Fitness")
    print(fitness)

    best_outputs.append(numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))

    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))

    # Selecting the best parents in the population for mating.
    parents = ga.select_mating_pool(new_population, fitness,
                                    num_parents_mating)
    print("Parents")
    print(parents)

    offspring_crossover = ga.crossover(parents,
                                       offspring_size=(pop_size[0]-parents.shape[0], num_weights))
    print("Crossover")
    print(offspring_crossover)

    # Adding some variations to the offspring using mutation.
    offspring_mutation = ga.mutation(offspring_crossover, num_mutations=2)
    print("Mutation")
    print(offspring_mutation)
```

```

# Creating the new population based on the parents and offspring.
new_population[0:parents.shape[0], :] = parents
new_population[parents.shape[0]:, :] = offspring_mutation

# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final generation.
fitness = ga.cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = numpy.where(fitness == numpy.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])

import matplotlib.pyplot
matplotlib.pyplot.plot(best_outputs)
matplotlib.pyplot.xlabel("Iteration")
matplotlib.pyplot.ylabel("Fitness")
matplotlib.pyplot.show()

```

Output:

```

[[ 2.20606732e+00 -2.40439876e+00 -2.13461012e+00  7.87018176e-02
 -5.47611080e-04 -3.94073159e+00]
 [ 3.02632039e+00 -2.80481380e-01 -2.87786378e+00  1.02778725e-01
  3.68002726e+00 -3.91712492e+00]
 [ 1.32374716e+00  4.27028930e-01  1.08097405e+00 -2.49556024e+00
 -6.81563339e-01 -3.37254889e+00]
 [ 1.38357290e-01  1.21535147e+00 -3.29653686e+00  4.13246975e-01
  1.14109082e+00 -2.50787896e-01]
 [-3.84485966e+00  1.65903291e+00 -2.41064586e-01  2.54967863e+00
  9.85377835e-01 -3.28143143e+00]
 [-4.14190657e-01 -3.87303751e+00 -3.16455714e+00  2.80399734e-01
  3.05122130e+00 -1.26511874e+00]
 [-3.75961734e+00  1.92618362e+00 -1.36575199e+00 -3.39035543e+00
 -3.36468095e+00 -1.39327094e+00]
 [-2.96224890e+00  3.05263204e+00  3.84911109e-01  3.05708819e+00
 -3.23054962e-01  8.23976445e-01]]
Generation :  0

```