NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY
NEW DELHI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Pattern Processing using Artificial Intelligence
COCSE60

Practical File

Shivam Tyagi
2019UCO1725

INDEX

Write a python program to implement a simple Chatbot.

Code:

```
import nltk
from nltk.chat.util import Chat, reflections

# Define patterns and responses for the chatbot
patterns = [
    (r"hi|Hi|hello|Hello|Hey|hey", ["Hello!", "Hi there!", "Hey!"]),
    (r"What is your name?|what is your name?", ["My name is Chatbot.", "You can call me Chatbot."]),
    (r"How are you?|how are you?", ["I'm doing well, thank you!", "I'm fine, thank you."]),
    (r"bye|goodbye", ["Goodbye!", "See you later.", "Bye!"]),
    (r"(.*)", ["I'm sorry, I don't understand.", "Can you please rephrase that?", "I'm not sure."])
]

# Create the chatbot
chatbot = Chat(patterns, reflections)

# Start the conversation with the chatbot
print("Welcome to Chatbot!")
print("You can start chatting with the chatbot. Type 'bye' to exit.")
while True:
    user_input = input("You: ")
    if user_input.lower() == 'bye':
        print("Chatbot: Goodbye!")
        break
    else:
        response = chatbot.respond(user_input)
        print("Chatbot: " + response)
```

We can also use premade LLMs (Large Language Models).

Code2:

```
import openai

# Set up your OpenAI API credentials
openai.api_key = "__PRIVATE KEY__"

# Function to generate response from GPT-3
def generate_response(prompt):
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=prompt,
```

```
        max_tokens=64
    )
    return response.choices[0].text.strip()

# Start the conversation with the chatbot
print("Welcome to Chatbot!")
print("You can start chatting with the chatbot. Type 'bye' to exit.")
while True:
    user_input = input("You: ")
    if user_input.lower() == 'bye':
        print("Chatbot: Goodbye!")
        break
    else:
        prompt = f"You: {user_input}\nChatbot:"
        response = generate_response(prompt)
        print("Chatbot: " + response)
```

Screenshot:

```
Welcome to Chatbot!
You can start chatting with the chatbot. Type 'bye' to exit.
You:  Hello
Chatbot: Hi there!
You:  who might you be
Chatbot: Can you please rephrase that?
You:  what is your name?
Chatbot: You can call me Chatbot.
You:  how are you doing
Chatbot: I'm fine, thank you.
You:  bye now
Chatbot: Bye!
You:  bye
Chatbot: Goodbye!
```

Write a program to implement k-means clustering from scratch.

```python
import numpy as np

class KMeans:
    def __init__(self, n_clusters=None, max_iters=100, t=1e-4):
        """
        Initialize KMeans clustering algorithm.

        Parameters:
        ----------
        n_clusters: int, optional (default=None)
            The number of clusters to form

        max_iters: int, optional (default=100)
            The maximum number of iterations for convergence

        tol: float, optional (default=1e-4)
            The tolerance

        """
        if not isinstance(n_clusters, int) or n_clusters <= 0:
            raise ValueError("n_clusters must be a positive integer greater than 0")
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.tol = t

    def fit(self, X):
        """
        Fit the KMeans model to the input data.

        Parameters:
        ----------
        X: array-like, shape (n_samples, n_features)
            The input data.

        """
        self.X = X
        self.n_samples, self.n_features = X.shape

        # Randomly initialize centroids
        self.centroids = self.X[np.random.choice(self.n_samples, size=self.n_clusters, replace=False)]
```

```
    # Iteratively update centroids until convergence
    for _ in range(self.max_iters):
        # Assign each data point to the nearest centroid
        distances = np.linalg.norm(self.X[:, np.newaxis] - self.centroids, axis=2)
        self.labels = np.argmin(distances, axis=1)

        # Update centroids
        new_centroids = np.array([self.X[self.labels == i].mean(axis=0) for i in range(self.n_clusters)])

        # Check for convergence
        if np.linalg.norm(new_centroids - self.centroids) < self.tol:
            break

        self.centroids = new_centroids

def predict(self, X):
    """
    Predict cluster labels for input data.

    Parameters:
    ----------
    X: array-like, shape (n_samples, n_features)
        The input data.

    Returns:
    ----------
    labels: array, shape (n_samples,)
        The predicted cluster labels for each data point.

    """
    distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
    labels = np.argmin(distances, axis=1)
    return labels
```

In [3]:
```
# Generate synthetic data for clustering
np.random.seed(0)
X = np.random.rand(100, 2) * 10

# Initialize KMeans with the desired number of clusters
n_clusters = 3
kmeans = KMeans(n_clusters=n_clusters)

# Fit the KMeans model to the data
kmeans.fit(X)

# Get the cluster labels for each data point
labels = kmeans.labels

# Get the cluster centroids
centroids = kmeans.centroids

# Print the cluster labels and centroids
print("Cluster Labels:")
print(labels)
print("Cluster Centroids:")
print(centroids)
```

```
Cluster Labels:
[2 0 2 2 0 0 2 1 2 2 0 2 2 2 0 2 2 2 0 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 0 0
 0 1 1 1 2 0 1 2 2 2 1 1 0 0 2 2 0 2 2 0 0 0 0 0 0 2 1 0 0 1 0 2 0 0 2 0 2
 2 2 2 2 0 1 0 2 0 1 0 1 1 0 2 0 2 0 0 2 0 1 2 0 1 1]
Cluster Centroids:
[[7.61673384 4.07653637]
 [2.72237147 2.10979969]
 [3.52178633 7.84887336]]
```

Generating samples of Gaussian (normal) distributions and plotting them for visualization

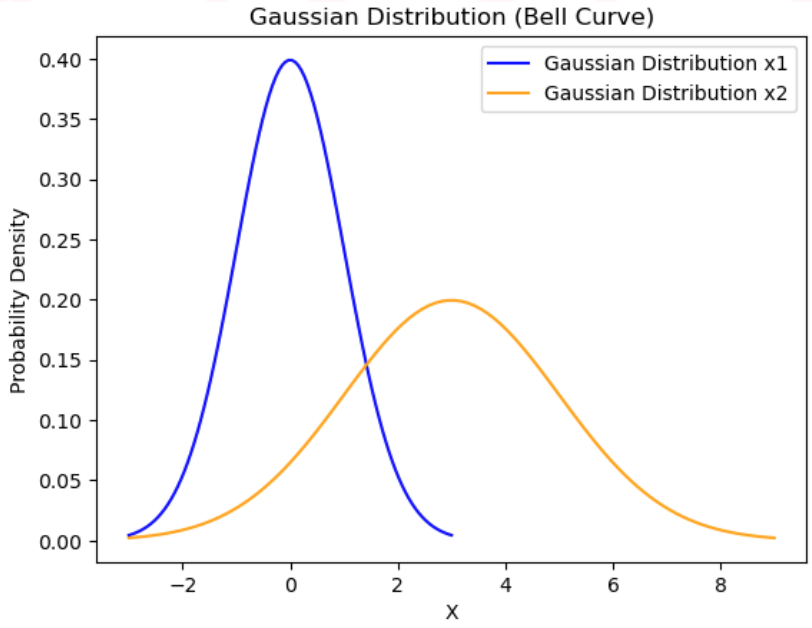
```python
import numpy as np
import matplotlib.pyplot as plt

# Mean and standard deviation of the Gaussian distribution
mu = 0
sigma = 1
mu2 = 3
sigma2= 2

# Generate x values for the bell curve
x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
x2 = np.linspace(mu2 - 3*sigma2, mu2 + 3*sigma2, 100)

# Compute PDF
pdf = (1/(sigma * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mu)/sigma)**2)
pdf2 = (1/(sigma2 * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x2 - mu2)/sigma2)**2)

# Plot the Gaussian distribution as a bell curve
plt.plot(x, pdf, color='blue', label='Gaussian Distribution x1')
plt.plot(x2, pdf2, color='orange', label='Gaussian Distribution x2')
plt.xlabel('X')
plt.ylabel('Probability Density')
plt.title('Gaussian Distribution (Bell Curve)')
plt.legend()
plt.show()
```



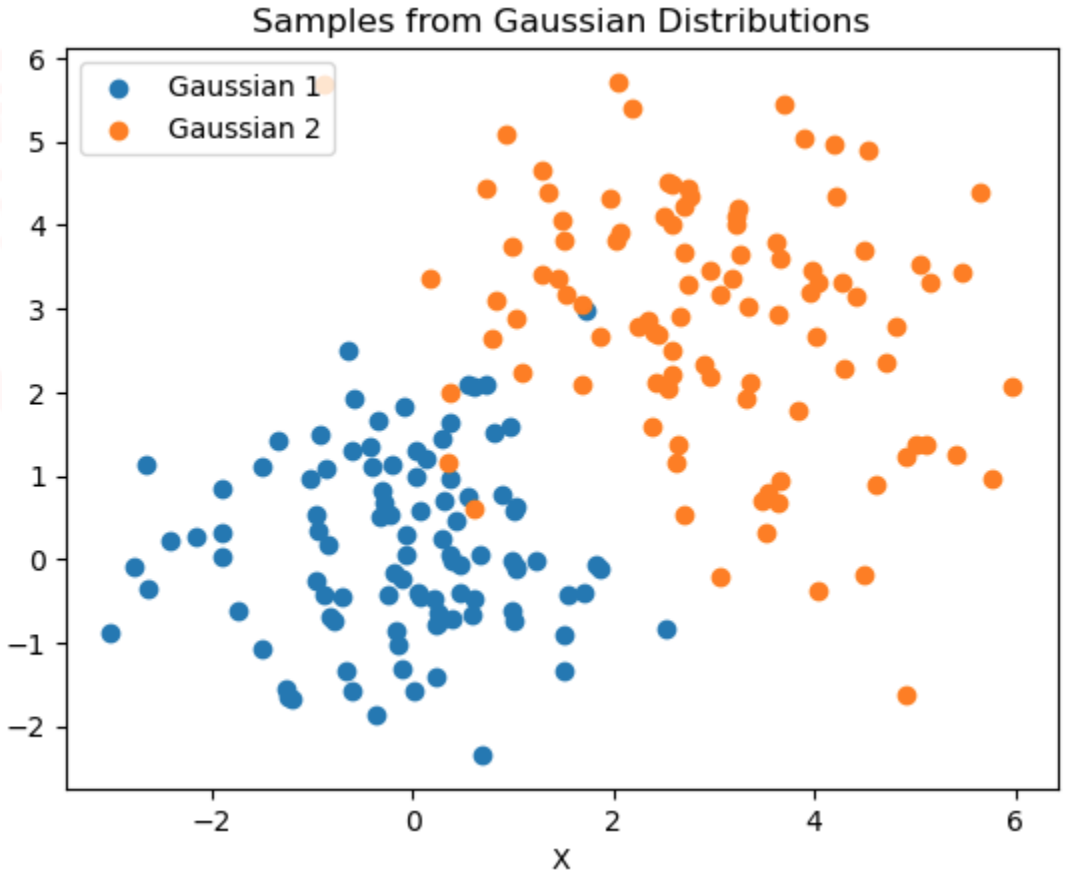
```python
import matplotlib.pyplot as plt

# Number of samples for each Gaussian distribution
num_samples = 100

# Mean and standard deviation for each Gaussian distribution
mu1 = [0, 0]
sigma1 = [[1, 0], [0, 1]]
mu2 = [3, 3]
sigma2 = [[2, 0], [0, 2]]

# Generate samples from Gaussian distributions
samples1 = np.random.multivariate_normal(mu1, sigma1, num_samples)
samples2 = np.random.multivariate_normal(mu2, sigma2, num_samples)

# Plot the generated samples
plt.scatter(samples1[:, 0], samples1[:, 1], label='Gaussian 1')
plt.scatter(samples2[:, 0], samples2[:, 1], label='Gaussian 2')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Samples from Gaussian Distributions')
plt.legend()
plt.show()
```



Implement Decision Tree algorithms.

DT

Creating the class and methods...

i dont really want to do this please use prebuilt packages

```python
class DecisionTree:
    def __init__(self, max_depth=None):
        """
        Initialize Decision Tree classifier.

        Parameters:
        ----------
        max_depth: int, optional (default=None)
            The maximum depth of the tree. If None, the tree will be grown to its maximum extent.

        """
        self.max_depth = max_depth

    def _calculate_gini(self, y):
        # Yes using gini index and not ID3
        """
        Calculate the Gini impurity of a target variable.

        Parameters:
        ----------
        y: array-like, shape (n_samples,)
            The target variable.

        Returns:
        -------
        gini: float
            The Gini impurity of the target variable.

        """
        classes, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        gini = 1 - np.sum(probabilities**2)
        return gini

    def _split_dataset(self, X, y, feature_idx, threshold):
        """
        Split the dataset based on a feature and threshold.

        Parameters:
        ----------
        X: array-like, shape (n_samples, n_features)
```

```
        X: array-like, shape (n_samples, n_features)
            The input features.

        y: array-like, shape (n_samples,)
            The target variable.

        feature_idx: int
            The index of the feature to split on.

        threshold: float
            The threshold value to split on.

        Returns:
        -------
        left_X, left_y, right_X, right_y: arrays
            The split dataset on the left and right branches of the tree.

        """
        left_mask = X[:, feature_idx] <= threshold
        right_mask = ~left_mask
        left_X, left_y = X[left_mask], y[left_mask]
        right_X, right_y = X[right_mask], y[right_mask]
        return left_X, left_y, right_X, right_y

    def _find_best_split(self, X, y):
        """
        Find the best feature and threshold to split the dataset based on Gini impurity.

        Parameters:
        ----------
        X: array-like, shape (n_samples, n_features)
            The input features.

        y: array-like, shape (n_samples,)
            The target variable.
```

```
    Returns:
    -------
    best_feature_idx: int
        The index of the best feature to split on.

    best_threshold: float
        The threshold value for the best feature to split on.

    """
    best_gini = np.inf
    best_feature_idx = None
    best_threshold = None

    for feature_idx in range(X.shape[1]):
        unique_values = np.unique(X[:, feature_idx])
        thresholds = (unique_values[:-1] + unique_values[1:]) / 2

        for threshold in thresholds:
            left_X, left_y, right_X, right_y = self._split_dataset(X, y, feature_idx, threshold)

            if len(left_y) == 0 or len(right_y) == 0:
                # Skip if the split results in empty branches
                continue

            gini = (len(left_y) / len(y)) * self._calculate_gini(left_y) + \
                   (len(right_y) / len(y)) * self._calculate_gini(right_y)

            if gini < best_gini:
                best_gini = gini
                best_feature_idx = feature_idx
                best_threshold = threshold

    return best_feature_idx, best_threshold

def _build_tree(self, X, y, depth=0):
    """
    Recursively build the decision tree.

    Parameters:
    ----------
           X: array-like, shape (n_samples, n_features)
        The input features.

    y: array-like, shape (n_samples,)
        The target variable.

    depth: int, optional (default=0)
```

```
    The current depth of the tree.

Returns:
-------
node: dict
    The constructed decision tree node as a dictionary.

"""
if depth == self.max_depth or np.unique(y).size == 1:
    # Leaf node: maximum depth reached or pure leaf node
    leaf_value = np.argmax(np.bincount(y))
    return {"leaf": True, "value": leaf_value}

best_feature_idx, best_threshold = self._find_best_split(X, y)

if best_feature_idx is None:
    # No split found, create a leaf node
    leaf_value = np.argmax(np.bincount(y))
    return {"leaf": True, "value": leaf_value}

# Split the dataset based on the best feature and threshold
left_X, left_y, right_X, right_y = self._split_dataset(X, y, best_feature_idx, best_threshold)

# Recursively build the left and right branches
left_node = self._build_tree(left_X, left_y, depth + 1)
right_node = self._build_tree(right_X, right_y, depth + 1)

# Create a decision node with the best feature and threshold
node = {
    "leaf": False,
    "feature_idx": best_feature_idx,
    "threshold": best_threshold,
    "left": left_node,
    "right": right_node
}
```

```
    return node

def fit(self, X, y):
    """
    Fit the decision tree to the training data.

    Parameters:
    ----------
    X: array-like, shape (n_samples, n_features)
        The input features.

    y: array-like, shape (n_samples,)
        The target variable.

    Returns:
    -------
    None

    """
    self.root = self._build_tree(X, y)

def _predict_single(self, x, node):
    """
    Predict the class label for a single input sample.

    Parameters:
    ----------
    x: array-like, shape (n_features,)
        The input sample.

    node: dict
        The current decision tree node.

    Returns:
    -------
    prediction: int
        The predicted class label for the input sample.

    """
    if node["leaf"]:
```

```
        # Leaf node, return the predicted class label
        return node["value"]

    # Recursively follow the decision tree based on feature and threshold
    if x[node["feature_idx"]] <= node["threshold"]:
        return self._predict_single(x, node["left"])
    else:
        return self._predict_single(x, node["right"])

def predict(self, X):
    """
    Predict the class labels for multiple input samples.

    Parameters:
    ----------
    X: array-like, shape (n_samples, n_features)
        The input features.

    Returns:
    -------
    predictions: array-like, shape (n_samples,)
        The predicted class labels for the input samples.

    """
    predictions = np.array([self._predict_single(x, self.root) for x in X])
    return predictions
```

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate synthetic dataset
np.random.seed(123)
X, y = make_classification(n_samples=100, n_features=5, n_informative=3, n_classes=2, random_state=1725)

# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1725)

# Initialize the DecisionTreeClassifier with max_depth=3
clf = DecisionTree(max_depth=3)

# Fit the model to the training data
clf.fit(X_train, y_train)

# Predict on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.3f}".format(accuracy))
```

```
Accuracy: 0.950
```

Implement SVM

```python
class SVM:
    def __init__(self, learning_rate=0.01, max_iterations=1000, C=1.0):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.C = C
        self.w = None
        self.b = None

    def fit(self, X, y):
        # Input parameter validation
        if not isinstance(X, np.ndarray) or not isinstance(y, np.ndarray):
            raise ValueError("Input data and labels must be numpy arrays.")
        if X.ndim != 2:
            raise ValueError("Input data must be a 2D array.")
        if y.ndim != 1:
            raise ValueError("Input labels must be a 1D array.")
        if X.shape[0] != y.shape[0]:
            raise ValueError("Number of samples in data and labels must match.")

        n_samples, n_features = X.shape

        # Initialize weights and bias
        self.w = np.zeros(n_features)
        self.b = 0

        # Perform gradient descent
        for _ in range(self.max_iterations):
            # Compute decision function
            scores = np.dot(X, self.w) + self.b

            # Compute hinge loss
            margins = y * scores
            loss = np.maximum(0, 1 - margins)

            # Compute gradient of hinge loss
            dW = np.zeros(n_features)
            db = 0
            mask = margins > 0
            dW += np.dot(X[mask].T, -y[mask])
            db += -np.sum(y[mask])

            # Add regularization term to gradient
            dW += self.C * self.w

            # Update weights and bias
            self.w -= self.learning_rate * dW
            self.b -= self.learning_rate * db
```

```
    def predict(self, X):
        # Input parameter validation
        if not isinstance(X, np.ndarray):
            raise ValueError("Input data must be a numpy array.")
        if X.ndim != 2:
            raise ValueError("Input data must be a 2D array.")
        if self.w is None or self.b is None:
            raise ValueError("Model has not been trained yet. Please call fit() first.")

        # Compute decision function
        scores = np.dot(X, self.w) + self.b

        # Predict labels
        y_pred = np.sign(scores)

        return y_pred
```

```
# Instantiate SVM
clf = SVM(learning_rate=0.01, max_iterations=1000, C=1.0)

# Train the classifier
clf.fit(X_train, y_train)

# Predict labels on test set
y_pred = clf.predict(X_test)

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

```
Accuracy: 0.4
```

Thats terrible... implementing kernel trick to increase performance

```
class SVM:
    def __init__(self, learning_rate=0.01, max_iterations=1000, C=1.0, kernel='linear', degree=3, gamma=0.1):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.C = C
        self.kernel = kernel
        self.degree = degree
        self.gamma = gamma

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Add bias term to the feature matrix
        X = np.hstack((np.ones((n_samples, 1)), X))

        # Initialize the weight vector and bias term
        self.W = np.zeros(n_features + 1)
        self.b = 0

        # Define the kernel function
        if self.kernel == 'linear':
            kernel_func = self.linear_kernel
        elif self.kernel == 'poly':
            kernel_func = self.polynomial_kernel
        elif self.kernel == 'rbf':
            kernel_func = self.rbf_kernel
        else:
            raise ValueError("Invalid kernel type. Supported kernels: 'linear', 'poly', 'rbf'")

        # Start gradient descent
        for i in range(self.max_iterations):
            # Compute the hinge loss and gradients
            loss, dW, db = self.compute_gradient(X, y, kernel_func)

            # Update the weight vector and bias term
            self.W -= self.learning_rate * dW
            self.b -= self.learning_rate * db

    def predict(self, X):
        # Add bias term to the feature matrix
        X = np.hstack((np.ones((X.shape[0], 1)), X))

        # Compute the predicted labels
        scores = np.dot(X, self.W) + self.b
        y_pred = np.sign(scores)

        return y_pred
```

```python
    def compute_gradient(self, X, y, kernel_func):
        n_samples = X.shape[0]

        # Compute the scores
        scores = np.dot(X, self.W) + self.b

        # Compute the hinge loss
        margin = y * scores
        hinge_loss = np.maximum(0, 1 - margin)
        loss = np.mean(hinge_loss) + 0.5 * self.C * np.sum(self.W ** 2)

        # Compute the gradients
        dW = np.zeros_like(self.W)
        db = 0

        # Only update gradients for samples that violate the margin
        violation = margin < 1
        dW += self.C * np.dot(X[violation].T, -y[violation])
        db += -np.sum(y[violation])

        # Add regularization to gradients
        dW += self.C * self.W

        # Update gradients with respect to kernel
        if self.kernel == 'linear':
            pass  # No additional updates needed for linear kernel
        elif self.kernel == 'poly':
            # Update gradients with respect to polynomial kernel
            dW, db = self.update_gradients_poly(X, y, dW, db, kernel_func)
        elif self.kernel == 'rbf':
            # Update gradients with respect to RBF kernel
            dW, db = self.update_gradients_rbf(X, y, dW, db, kernel_func)

        # Normalize gradients by number of samples
        dW /= n_samples
        db /= n_samples

        return loss, dW, db

    def linear_kernel(self, X):
        return X

    def polynomial_kernel(self, X):
        return (1 + np.dot(X, self.gamma)) ** self.degree
```

```python
def update_gradients_poly(self, X, y, dW, db, kernel_func):
    # Compute kernel matrix
    K = kernel_func(X)

    # Compute gradient with respect to weights
    dW += self.C * np.dot(K.T, -y)

    # Compute gradient with respect to bias
    db += -np.sum(y)

    return dW, db
```

```python
# Initialize
linear, poly, rbf = SVM(kernel='linear', degree=3, gamma=0.1), SVM(kernel='poly', degree=3, gamma=0.1),

# Fit on data
linear.fit(X_train, y_train)
poly.fit(X_train, y_train)
# rbf.fit(X_train, y_train)

# Predict labels for new samples
y_pred_lin = linear.predict(X_test)
y_pred_poly = linear.predict(X_test)
# y_pred_rbf = linear.predict(X_test)

# Evaluate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Linear Accuracy: {accuracy:.2f}")
accuracy = np.mean(y_pred_poly == y_test)
print(f"Poly Accuracy: {accuracy:.2f}")
# accuracy = np.mean(y_pred_rbf == y_test)
# print(f"RBF Accuracy: {accuracy:.2f}")
```

```
Linear Accuracy: 0.40
Poly Accuracy: 0.60
```

Implement Principal component analysis and use it for unsupervised learning

PCA

```
class PCA:
    def __init__(self, n_components=None):
        self.n_components = n_components

    def fit(self, X):
        # Compute the covariance matrix
        cov_matrix = np.cov(X.T)

        # Compute the eigenvalues and eigenvectors of the covariance matrix
        eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

        # Sort the eigenvalues and eigenvectors in descending order
        idx = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[idx]
        eigenvectors = eigenvectors[:, idx]

        # Select the top n_components eigenvectors if specified
        if self.n_components is not None:
            eigenvectors = eigenvectors[:, :self.n_components]

        self.eigenvectors = eigenvectors

    def transform(self, X):
        # Project the data onto the selected eigenvectors
        return np.dot(X, self.eigenvectors)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

In [93]:
```
# Create a synthetic dataset
X = np.random.rand(20, 10)

# Initialize PCA with n_components=3
pca = PCA(n_components=3)

# Fit and transform the dataset
X_pca = pca.fit_transform(X)

# Print the transformed data
print("Transformed data:")
print(X_pca)
```

```
Transformed data:
[[ 1.59648586e+00 -4.14116611e-02 -7.97150119e-02]
 [ 7.77231956e-01  1.83299293e-01 -4.63531548e-01]
 [ 4.60890525e-01 -7.07151808e-01 -1.80863071e-01]
 [ 5.26114291e-01  3.58750974e-01 -4.81415136e-01]
 [ 2.90814792e-01  4.13221281e-01 -7.24108138e-01]
 [ 4.37100061e-01 -2.67465112e-01  1.23879819e-01]
 [ 3.73754603e-01  3.90225020e-01 -1.30067681e-03]
 [ 1.08195542e+00  2.22844551e-01 -5.72371596e-01]
 [ 8.59561905e-01  5.70371306e-01  3.53103588e-01]
 [ 9.49067225e-01  7.75475840e-01  2.62688808e-02]
 [ 6.33885331e-01  2.22311540e-01 -7.41963844e-01]
 [ 2.61590645e-01 -6.55904189e-03 -2.41716711e-01]
 [ 4.78294538e-01  2.77895005e-01  2.29292760e-01]
 [ 2.72869378e-01  9.35407690e-01 -2.81273786e-01]
 [ 5.96409354e-01 -8.46201375e-03 -5.53128047e-01]
 [ 9.36935085e-01  8.44905520e-01 -1.99598257e-01]
 [ 7.53435415e-02 -1.60902842e-01  5.99014676e-01]
 [ 3.64055575e-01  1.00036487e-02 -8.09996590e-01]
 [ 3.89684370e-01  7.42828457e-01  6.80284445e-02]
 [ 1.76663526e+00 -2.59778507e-02 -6.12475438e-02]]
```

Implement Maximum-Likelihood estimation

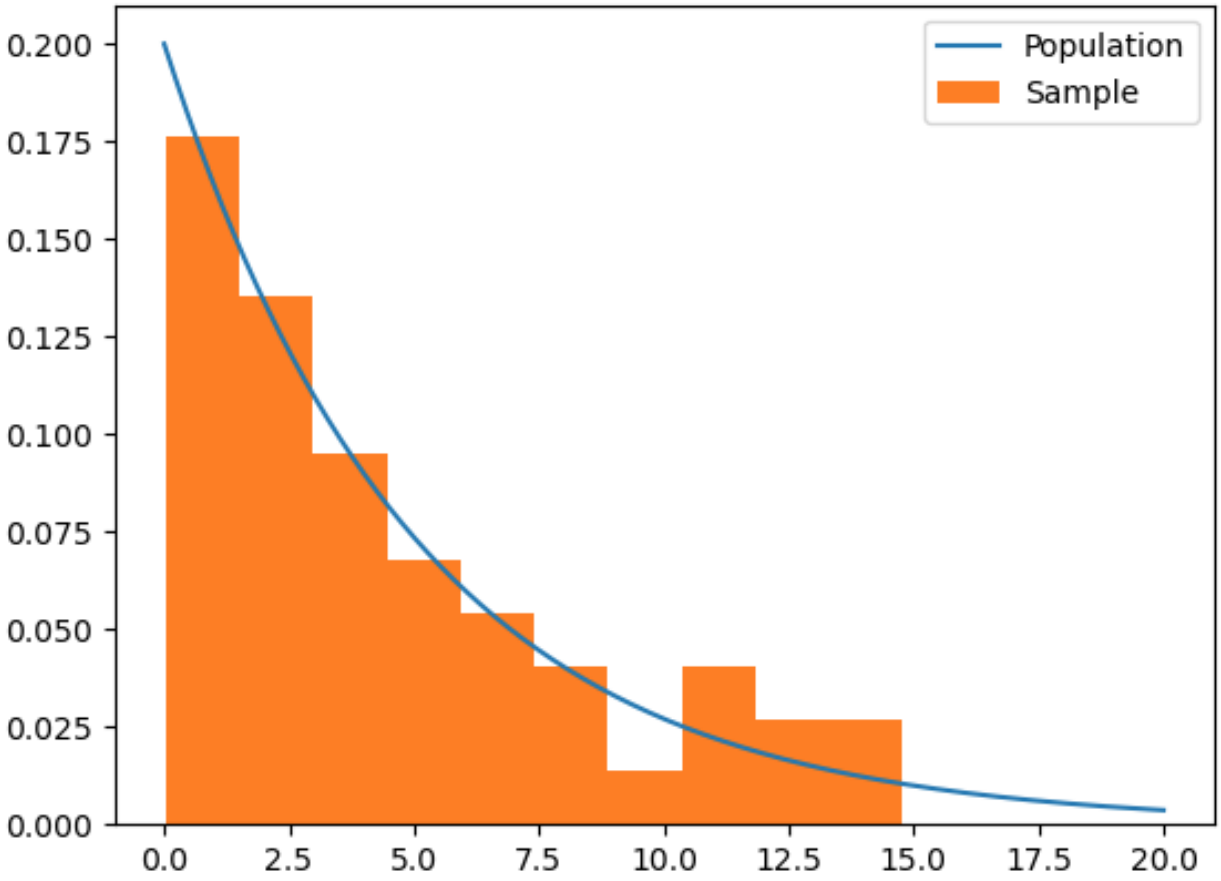
```
# import libraries
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.stats import expon
```

```
# define population and sample size
sample_size = 1e8
pop_rate = 5

# create function to get samples
get_sample = lambda x: np.random.exponential(pop_rate, x)

# generate data
X = np.arange(0, 20, 0.001)
y = expon.pdf(X, scale=pop_rate)

# Plot the samples
plt.plot(X, y, label='Population')
sample = get_sample(50)
plt.hist(sample, label='Sample', density = True)
plt.legend()
plt.show()
```

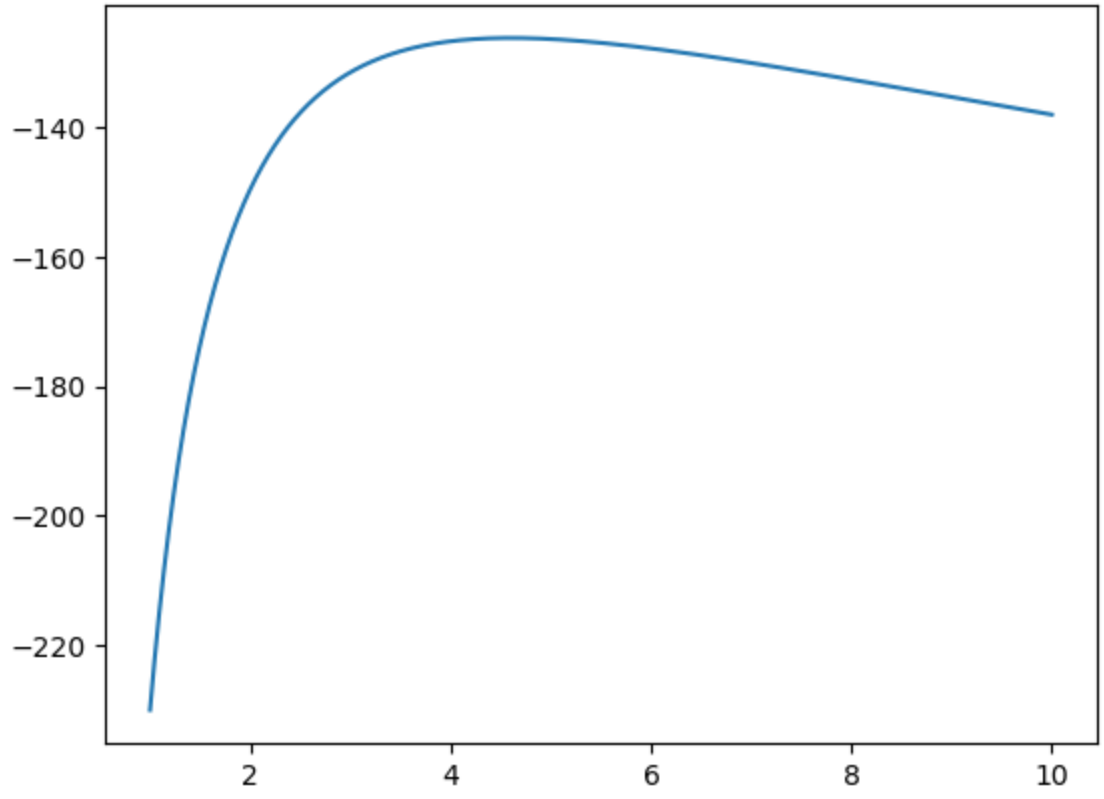

Population
Sample
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

```
# Prepare log-liklihood for estimation
log_lik = lambda x: sum(np.log((1/x)*(np.e**(-i/x))) for i in sample)

# prepare estimates
rate = np.arange(1,10,0.001)
estimate = np.asarray([log_lik(_) for _ in rate])

# plot the extimation
plt.plot(rate, estimate)
plt.show()

# Comparing estimated parameter max and mean of sampled data
print(f"Sample mean {sample.mean():.4f} \t Estimate {rate[np.argmax(estimate)]:.4f}")
print('Absolute difference between sample and estimate:')
print(np.abs(sample.mean() - rate[np.argmax(estimate)]))
```




```
Sample mean 4.5996       Estimate 4.6000
Absolute difference between sample and estimate:
0.0003941618202363273
```

Implement agglomerative Hierarchical clustering

Hierarchical Clustering

```
from scipy.spatial.distance import cdist

class AHC:
    def __init__(self, n_clusters=2, linkage='single', distance_metric='euclidean'):
        """
        Agglomerative Hierarchical Clustering

        Parameters:
        -----------
        n_clusters : int, optional (default=2)
            The number of clusters to form.
        linkage : str, optional (default='single')
            The linkage criterion to determine distance between clusters.
            Possible values: {'single', 'complete', 'average'}
        distance_metric : str, optional (default='euclidean')
            The distance metric to use when calculating pairwise distances between samples.
            Possible values: {'euclidean', 'manhattan', 'cosine'}
        """
        self.n_clusters = n_clusters
        self.linkage = linkage
        self.distance_metric = distance_metric
        self.labels_ = None
        self.distances_ = None

    def fit(self, X):
        """
        Fit the Agglomerative Hierarchical Clustering model to the input data.

        Parameters:
        -----------
        X : array-like, shape (n_samples, n_features)
            The input data.
        """
        n_samples = X.shape[0]
        self.labels_ = np.arange(n_samples)
        dist_matrix = np.zeros((n_samples, n_samples))

        for i in range(n_samples):
            for j in range(i+1, n_samples):
                dist_matrix[i, j] = np.linalg.norm(X[i] - X[j])
                dist_matrix[j, i] = dist_matrix[i, j]

        while len(np.unique(self.labels_)) > self.n_clusters:
            min_dist = np.inf
            merge_idx = None
```

```
        for i in range(n_samples):
            for j in range(i+1, n_samples):
                if self.labels_[i] == self.labels_[j]:
                    continue

                if self.linkage == 'single':
                    dist = dist_matrix[i, j]
                elif self.linkage == 'complete':
                    dist = np.max([dist_matrix[k, l] for k in range(n_samples) for l in range(n_samples)
                                   if self.labels_[k] == self.labels_[i] and self.labels_[l] != self.labels_[i]])
                else:
                    raise ValueError("Invalid linkage type. Choose 'single' or 'complete'.")

                if dist < min_dist:
                    min_dist = dist
                    merge_idx = (i, j)

        if merge_idx is not None:
            i, j = merge_idx
            self.labels_[self.labels_ == self.labels_[j]] = self.labels_[i]

    return self

def predict(self, X):
    """
    Predict the cluster labels for new data.

    Parameters:
    -----------
    X : array-like, shape (n_samples, n_features)
        The input data.

    Returns:
    --------
    labels : array, shape (n_samples,)
        The predicted cluster labels.
    """
    if self.labels_ is None:
        raise ValueError("fit() method must be called before predict()")

    distances = np.zeros((X.shape[0], len(np.unique(self.labels_))))

    # Compute pairwise distances between new data and cluster centroids
    if self.distance_metric == 'euclidean':
        distances = cdist(X, self.cluster_centers_, metric='euclidean')
    elif self.distance_metric == 'manhattan':
            distances = cdist(X, self.cluster_centers_, metric='cityblock')
        elif self.distance_metric == 'cosine':
            distances = cdist(X, self.cluster_centers_, metric='cosine')
        else:
            raise ValueError("Invalid distance metric. Possible values: {'euclidean', 'manhattan', 'cosine'}")

        # Assign samples to closest cluster
        labels = np.argmin(distances, axis=1)

        return labels
```

```python
# Importing libraries and dataset
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate synthetic data
X, y = make_blobs(n_samples=200, centers=3, random_state=1725)

# Instantiate AgglomerativeClustering with single linkage criterion
clf = AHC(n_clusters=3, linkage='single')

# Fit and predict clusters
clf.fit(X)
labels = clf.labels_

# Plot the original data points and their cluster assignments
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.title('Agglomerative Hierarchical Clustering (Single Linkage)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```