Tutorial

Question 4:

Implement random forest (use avg to enssemble your result)

Answer:

```
# Import necessary libraries
from sklearn.datasets import load_boston  # To load the Boston housing dataset
from sklearn.model_selection import train_test_split  # To split data into training and testing sets
from sklearn.ensemble import RandomForestRegressor  # To implement the Random Forest
Regressor
from sklearn.metrics import mean_squared_error  # To evaluate the model's performance
import pandas as pd  # For data manipulation
import numpy as np  # For numerical operations

# Load the Boston housing dataset
boston = load_boston()

# Convert the dataset into a pandas DataFrame for easier manipulation
# The dataset contains 13 features about houses and their median value
df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['MEDV'] = boston.target  # Add the target variable to the DataFrame

# Define the feature variables (X) and the target variable (y)
X = df.drop('MEDV', axis=1)  # Features: all columns except 'MEDV'
y = df['MEDV']  # Target: 'MEDV' column

# Split the data into training and testing sets
# 80% of the data will be used for training, and 20% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Random Forest Regressor
# n_estimators=100 specifies the number of decision trees in the forest
# random_state=42 ensures reproducibility of results
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model on the training data
rf_regressor.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = rf_regressor.predict(X_test)

# Evaluate the model's performance using Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
```

```
# Optional: Calculate the Root Mean Squared Error (RMSE) for better interpretability
rmse = np.sqrt(mse)
print(f"Root Mean Squared Error: {rmse:.2f}")

# Optional: Display feature importances to understand which features contribute most to the
model
importances = rf_regressor.feature_importances_
feature_importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': importances
}).sort_values(by='Importance', ascending=False)

print("\nFeature Importances:")
print(feature_importance_df)
```

This implementation leverages the averaging of multiple decision trees to enhance the robustness and accuracy of predictions, a key characteristic of the Random Forest algorithm.

Question 5:
Implement SVM (not the library function)

Answer:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

class SVM:
    """
    Support Vector Machine (SVM) classifier using stochastic gradient descent optimization.
    """

    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        """
        Initialize the SVM classifier.

        Parameters:
        - learning_rate: Step size for weight updates.
        - lambda_param: Regularization parameter to prevent overfitting.
        - n_iters: Number of iterations over the training data.
        """
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None  # Weight vector
        self.b = None  # Bias term

    def fit(self, X, y):
        """
        Train the SVM classifier on the provided data.

        Parameters:
        - X: Input features, shape (n_samples, n_features).
        - y: Target labels, shape (n_samples,). Labels should be -1 or 1.
        """
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)  # Initialize weights to zero
        self.b = 0  # Initialize bias to zero

        # Perform stochastic gradient descent for n_iters iterations
        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                # Check if the current sample is correctly classified
                condition = y[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    # Update weights for correctly classified samples
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    # Update weights and bias for misclassified samples
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y[idx]))
                    self.b -= self.lr * y[idx]

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

        Parameters:
        - X: Input features, shape (n_samples, n_features).

        Returns:
        - Predicted class labels, shape (n_samples,).
        """
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)

# Generate a synthetic dataset for binary classification
```

```
X, y = make_classification(n_samples=100, n_features=2, n_classes=2,
n_clusters_per_class=1, n_redundant=0, random_state=42)

# Convert labels from {0, 1} to {-1, 1} for SVM compatibility
y = np.where(y == 0, -1, 1)

# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize the SVM classifier with specified hyperparameters
svm = SVM(learning_rate=0.001, lambda_param=0.01, n_iters=1000)

# Train the SVM classifier on the training data
svm.fit(X_train, y_train)

# Make predictions on the test data
predictions = svm.predict(X_test)

# Calculate the accuracy of the classifier
accuracy = np.mean(predictions == y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Function to plot the decision boundary of the SVM classifier
def plot_decision_boundary(X, y, model):
    """
    Plot the decision boundary of the SVM classifier along with the data points.

    Parameters:
    - X: Input features, shape (n_samples, 2).
    - y: Target labels, shape (n_samples,).
    - model: Trained SVM model.
    """
    def decision_boundary(x):
        return -(model.w[0] * x + model.b) / model.w[1]

    # Define the range for the plot
    x_min, x_max = np.min(X[:, 0]) - 1, np.max(X[:, 0]) + 1
    y_min, y_max = np.min(X[:, 1]) - 1, np.max(X[:, 1]) + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50), np.linspace(y_min, y_max, 50))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary and margins
    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.bwr, edgecolors='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('SVM Decision Boundary')
    plt.show()

# Plot the decision boundary using the test data
plot_decision_boundary(X_test, y_test, svm)
```

Implementing a Support Vector Machine (SVM) from scratch involves several steps, including data preparation, defining the SVM model, training the model using an optimization algorithm, and making predictions.

Question 6:
Implement hierarchical clustering using complete linkage.

Answer:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform

def complete_linkage_clustering(data, num_clusters):
    """
    Perform hierarchical clustering using complete linkage.

    Parameters:
    - data: ndarray
        A 2D array where each row represents a data point.
    - num_clusters: int
        The desired number of clusters.

    Returns:
    - clusters: list of lists
        A list where each sublist contains the indices of data points in that cluster.
    """
    # Calculate the pairwise Euclidean distance between data points
    distances = squareform(pdist(data, metric='euclidean'))

    # Initialize each data point as its own cluster
    clusters = [[i] for i in range(len(data))]

    # Initialize a matrix to keep track of cluster distances
    cluster_distances = distances.copy()

    # Set the diagonal to infinity to avoid merging a cluster with itself
    np.fill_diagonal(cluster_distances, np.inf)

    # Iteratively merge clusters until the desired number of clusters is reached
    while len(clusters) > num_clusters:
        # Find the pair of clusters with the smallest maximum pairwise distance
        cluster1, cluster2 = np.unravel_index(np.argmin(cluster_distances),
cluster_distances.shape)

        # Merge cluster2 into cluster1
        clusters[cluster1].extend(clusters[cluster2])
        del clusters[cluster2]

        # Update the distance matrix
        # The distance between the new cluster and any other cluster is the maximum distance
        # between any member of the new cluster and any member of the other cluster
        for i in range(len(clusters)):
            if i != cluster1:
                # Calculate the complete linkage distance
                max_distance = max(distances[p1, p2] for p1 in clusters[cluster1] for p2 in clusters[i])
                cluster_distances[cluster1, i] = cluster_distances[i, cluster1] = max_distance

        # Remove the merged cluster from the distance matrix
        cluster_distances = np.delete(cluster_distances, cluster2, axis=0)
        cluster_distances = np.delete(cluster_distances, cluster2, axis=1)

    return clusters

# Example usage:
if __name__ == "__main__":
    # Generate sample data: 10 points in 2D space
    np.random.seed(42)
    data = np.random.rand(10, 2)

    # Perform hierarchical clustering to obtain 3 clusters
    clusters = complete_linkage_clustering(data, num_clusters=3)

    # Visualize the clusters
    colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
    for idx, cluster in enumerate(clusters):
        cluster_points = data[cluster]
        plt.scatter(cluster_points[:, 0], cluster_points[:, 1], c=colors[idx % len(colors)],
label=f'Cluster {idx+1}')

    plt.title('Complete Linkage Hierarchical Clustering')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.show()
```

Question 7:
Implement Apriori algorithm.

Answer:
```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# Sample dataset: List of transactions
dataset = [
    ['milk', 'bread', 'eggs'],
    ['milk', 'bread'],
    ['milk', 'eggs'],
    ['bread', 'eggs'],
    ['milk', 'bread', 'eggs', 'butter'],
    ['bread', 'butter']
]

# Convert dataset into a DataFrame of one-hot encoded transactions
df = pd.DataFrame(dataset)
df = df.stack().str.get_dummies().sum(level=0)

# Generate frequent itemsets with a minimum support of 0.5
frequent_itemsets = apriori(df, min_support=0.5, use_colnames=True)

# Generate association rules with a minimum confidence of 0.7
rules = association_rules(frequent_itemsets, metric='confidence', min_threshold=0.7)

# Display the results
print("Frequent Itemsets:")
print(frequent_itemsets)
print("\nAssociation Rules:")
print(rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])
```

The Apriori algorithm is a classic method in data mining for identifying frequent itemsets and deriving association rules from transactional datasets. It's widely used in market basket analysis to uncover relationships between items purchased together.