1.Increasing Training Set Size Experiment: Consider the iris dataset for multiclass classification and perform the following steps.

1. Divide the data into 80% training and 20% testing.
2. From the training set only take 5% of the data and train the supervised learning models (Logistic Regression, Decision Trees, Random Forest, and Naive Bayes) and test it on the test set created in the previous step.
3. Repeat the training again with now 10% of the data and keep on adding the 5% until you use the whole training set.
4. In every training test on the 20% of the test set and report the accuracy and f1-score of the model.
5. Plot the sample graph for accuracy and f1-score as provided below:

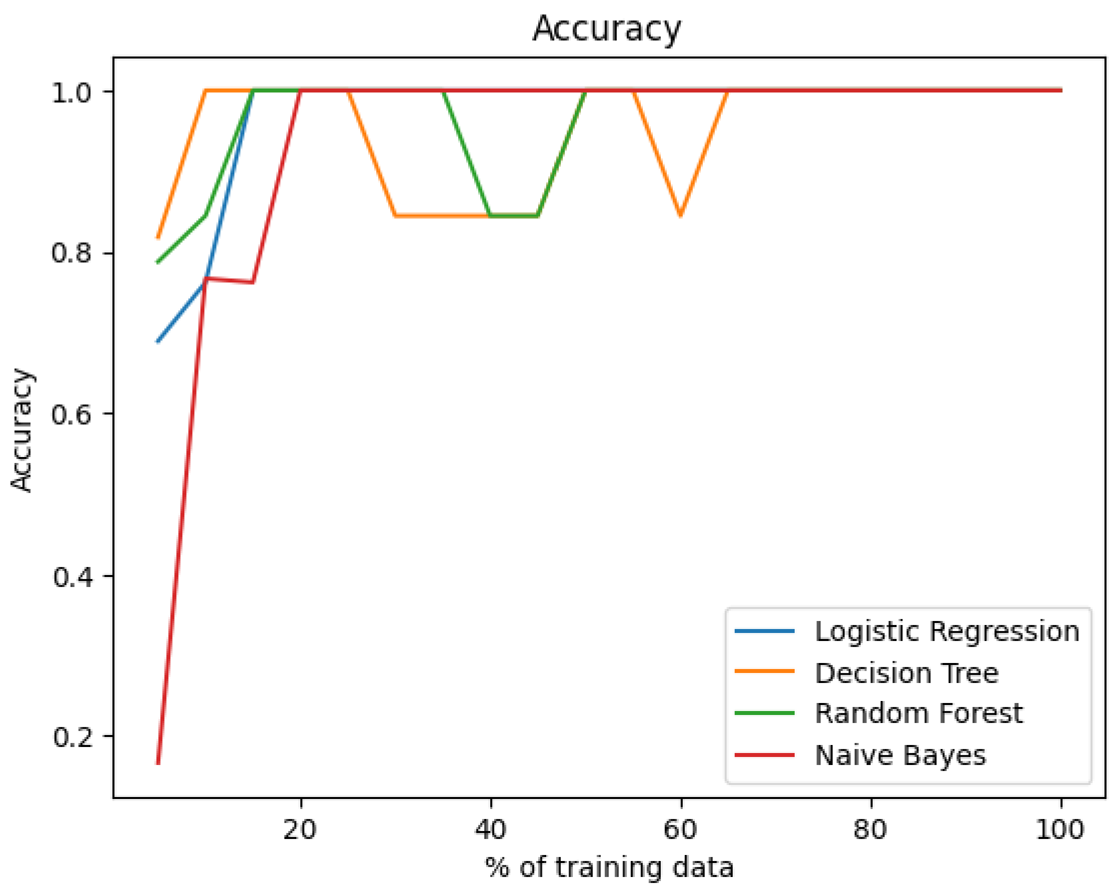
```
# Importing necessary libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, f1_score
import matplotlib.pyplot as plt

# Loading the iris dataset
iris = load_iris()
X = iris.data
y = iris.target


# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Defining the models to be trained
models = [LogisticRegression(), DecisionTreeClassifier(), RandomForestClassifier(), GaussianN


# Initializing the lists to store the scores
acc_scores = []
f1_scores = []

# Looping through different percentages of the training data
for i in range(5, 105, 5):
    # Selecting i% of the training data
    n_samples = int(X_train.shape[0] * i / 100)
```

```
    X_train_subset = X_train[:n_samples, :]
    y_train_subset = y_train[:n_samples]
    print(f'Training with {i}% of the data')
    # Training the models on the subset of the training data
    if i == 5:
        for model in models:
            model.fit(X_train_subset, y_train_subset)

            # Testing the models on the test set
            y_pred = model.predict(X_test)
            acc = accuracy_score(y_test, y_pred)
            f1 = f1_score(y_test, y_pred, average='weighted')

            # Storing the scores
            acc_scores.append(acc)
            f1_scores.append(f1)
            print(f'{type(model).__name__} accuracy: {acc:.2f}')
    else:
        for model in models:
            model.fit(X_train_subset, y_train_subset)

            # Testing the models on the 20% of the test set
            n_test_samples = int(X_test.shape[0] * 0.2)
            X_test_subset = X_test[:n_test_samples, :]
            y_test_subset = y_test[:n_test_samples]
            y_pred = model.predict(X_test_subset)
            acc = accuracy_score(y_test_subset, y_pred)
            f1 = f1_score(y_test_subset, y_pred, average='weighted')

            # Storing the scores
            acc_scores.append(acc)
            f1_scores.append(f1)
            print(f'{type(model).__name__} accuracy: {acc:.2f}, f1-score: {f1:.2f}')
```

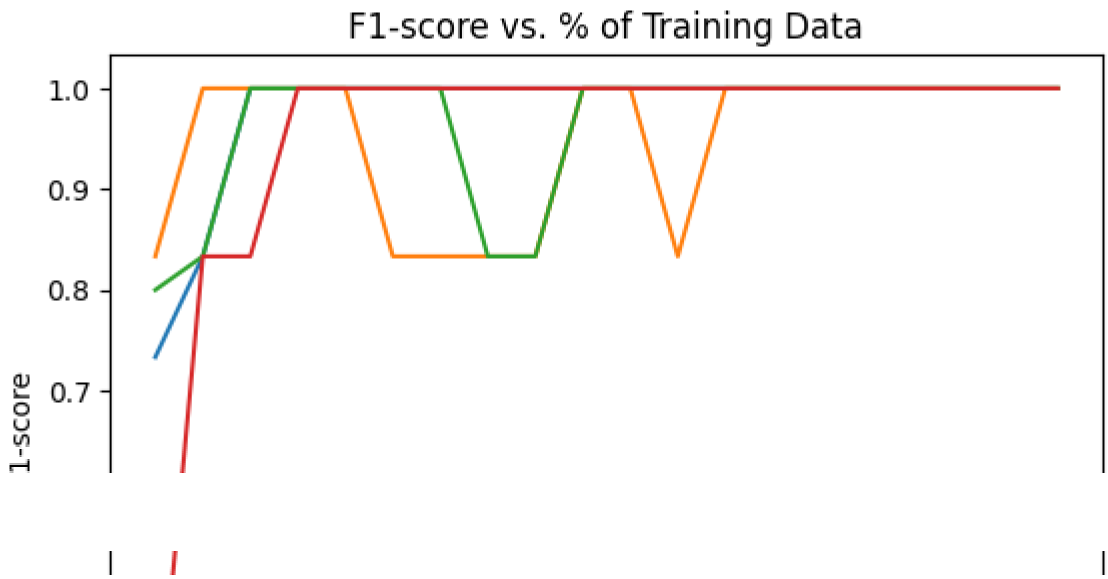
```
RandomForestClassifier accuracy: 1.00, f1-score: 1.00
GaussianNB accuracy: 1.00, f1-score: 1.00
Training with 85% of the data
LogisticRegression accuracy: 1.00, f1-score: 1.00
DecisionTreeClassifier accuracy: 1.00, f1-score: 1.00
RandomForestClassifier accuracy: 1.00, f1-score: 1.00
GaussianNB accuracy: 1.00, f1-score: 1.00
Training with 90% of the data
LogisticRegression accuracy: 1.00, f1-score: 1.00
DecisionTreeClassifier accuracy: 1.00, f1-score: 1.00
RandomForestClassifier accuracy: 1.00, f1-score: 1.00
GaussianNB accuracy: 1.00, f1-score: 1.00
Training with 95% of the data
/usr/local/lib/python3.9/dist-packages/sklearn/linear_model/_logistic.py:458: Converg
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
LogisticRegression accuracy: 1.00, f1-score: 1.00
DecisionTreeClassifier accuracy: 1.00, f1-score: 1.00
RandomForestClassifier accuracy: 1.00, f1-score: 1.00
GaussianNB accuracy: 1.00, f1-score: 1.00
Training with 100% of the data
LogisticRegression accuracy: 1.00, f1-score: 1.00
DecisionTreeClassifier accuracy: 1.00, f1-score: 1.00
RandomForestClassifier accuracy: 1.00, f1-score: 1.00
GaussianNB accuracy: 1.00, f1-score: 1.00
/usr/local/lib/python3.9/dist-packages/sklearn/linear_model/_logistic.py:458: Converg
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
```

```
# Creating the x-axis for the plot
x = np.arange(5, 105, 5)

# Reshaping the scores into a 2D array for plotting
acc_scores = np.array(acc_scores).reshape(-1, 4)
f1_scores = np.array(f1_scores).reshape(-1, 4)

# Plotting the accuracy scores
plt.plot(x, f1_scores[:, 0], label='Logistic Regression')
plt.plot(x, f1_scores[:, 1], label='Decision Tree')
plt.plot(x, f1_scores[:, 2], label='Random Forest')
plt.plot(x, f1_scores[:, 3], label='Naive Bayes')
plt.xlabel('% of training data')
plt.ylabel('Accuracy')
plt.title('Accuracy')
plt.legend()
```

```
plt.show()
```




```
# Plotting the accuracy scores
plt.plot(x, acc_scores[:, 0], label='Logistic Regression')
plt.plot(x, acc_scores[:, 1], label='Decision Tree')
plt.plot(x, acc_scores[:, 2], label='Random Forest')
plt.plot(x, acc_scores[:, 3], label='Naive Bayes')
plt.xlabel('% of training data')
plt.ylabel('F1-score')
plt.title('F1-score vs. % of Training Data')
plt.legend()
plt.show()
```



2. (30 Points) Linear Regression: Consider the following N data points (N=10)

   X = [−1.4 , −1.6 , −1.3 , 0 . 2 , 2 . 0 , −1.1 , 0 . 0 , 0 . 3 , −0.9 , −1.8]

   r = [6 . 9 , 7 . 8 , 8 . 0 , 5 . 8 , 1 . 9 , 7 . 3 , 5 . 8 , 5 . 8 , 8 . 2 , 9 . 6]

   Note that these data points are ordered, so that (X1,r1) = (−1.3,6.9) and (X10,r10) = (−1.8,9.6). For the above data points, fit a linear regression model, f(x) = w0 +w1x, by estimating the values of w0 and w1, so that ri = f(Xi) +εi .

Hint: w0 = r¯−w1X¯, w1 = ∑ N i=1 Xiri −X¯rN¯ ∑ N i=1 (Xi) 2 −N(X¯) 2 where r¯ = 1 N N ∑ i=1 ri , X¯ = 1 N N ∑ i=1 Xi

Reference: Chatgpt

```
import numpy as np

# Define the training data set
X = np.array([-1.4, -1.6, -1.3, 0.2, 2.0, -1.1, 0.0, 0.3, -0.9, -1.8])
r = np.array([6.9, 7.8, 8.0, 5.8, 1.9, 7.3, 5.8, 5.8, 8.2, 9.6])


# Define the function to compute w0 and w1
def compute_w(X, r):
    # Compute the number of data points
    N = len(X)
    # Compute the mean of the response variable
    r_mean = np.mean(r)
    # Compute the mean of the predictor variable
    X_mean = np.mean(X)
    # Compute the numerator of the slope
    numerator = np.sum(X * r) - N * X_mean * r_mean
    # Compute the denominator of the slope
```

```
    denominator = np.sum(X**2) - N * X_mean**2
    # Compute the slope
    w1 = numerator / denominator
    # Compute the intercept
    w0 = r_mean - w1 * X_mean
    return w0, w1


# Compute w0 and w1
w0, w1 = compute_w(X, r)

# Define the linear regression model
f = lambda x: w0 + w1 * x

# Compute RMSE and MAE for the training data set
ei = f(X) - r
RMSE = np.sqrt(np.mean(ei**2))
MAE = np.median(np.abs(ei))


# Print the results
print("Linear Regression Model:")
print("w0 =", w0)
print("w1 =", w1)
print("RMSE for the training data set =", RMSE)
print("MAE for the training data set =", MAE)
```

```
Linear Regression Model:
w0 = 5.768549422336327
w1 = -1.6811617458279855
RMSE for the training data set = 0.648223478278935
MAE for the training data set = 0.5210125160462125
```

```
# Define the test data set
Z = np.array([-0.6, 1.8, -0.1, 1.1, -1.7])

# Compute the predictions for the test data set
predictions = f(Z)

# Define the true labels for the test data set
u = np.array([5.1, -0.2, 6.5, 2.2, 8.3])



# Compute RMSE and MAE for the test data set
ei = predictions - u
RMSE = np.sqrt(np.mean(ei**2))
MAE = np.median(np.abs(ei))

# Print the results
print("Predictions for the test data set:", predictions)
```

```
Predictions for the test data set: [6.77724647 2.74245828 5.9366656  3.9192715  8.62652
```

```
print("RMSE for the test data set =", RMSE)
print("MAE for the test data set =", MAE)
```

```
RMSE for the test data set = 1.7234311575258012
MAE for the test data set = 1.6772464698331184
```

Double-click (or enter) to edit

3. Consider the 30 data points and their corresponding class labels stored in a dictionary named “data_dict”.

d a t a_dic t = { (2 . 0 , 3. 4 3 , 4 . 3 7) : 2 , (2 . 4 9 , 4. 2 8 , 4 . 8 3) : 2 , (2 . 5 8 , 4. 3 6 , 4 . 4 8) : 2 , (2 . 6 6 , 4. 4 5 , 5 . 9 5): 2 , (2 . 8 2 , 3. 6 6 , 4 . 5 1) : 2 , (3 . 0 3 , 4. 3 7 , 5 . 0 7) : 2 , (3 . 2 7 , 4. 5 4 , 4 . 5 7) : 2 , (3 . 4 1 , 3. 9 4 , 5 . 3 5) : 2 , (3 . 5 3 , 4. 3 2 , 5 . 4 1) : 2 , (3 . 5 3 , 4 . 6 , 6 . 8) : 1 , (3 . 6 1 , 4. 2 5 , 5 . 2 1) : 1 , (3 . 6 1 , 4. 7 8 , 5 . 4 7) : 1 , (3 . 7 2 , 5. 4 4 , 5 . 8 8) : 1 , (3 . 8 7 , 4. 9 6 , 4 . 5 2) : 2 , (4 . 1 3 , 5. 2 9 , 6 . 6) : 1 , (4 . 2 5 , 5. 9 7 , 5 . 4 8) : 1 , (4 . 6 1 , 4 . 9 , 5 . 1 1) : 1 , (4 . 7 3 , 4 . 4 , 6 . 7 8) : 1 , (4 . 9 7 , 4. 2 5 , 5 . 0) : 1 , (4 . 9 8 , 5. 2 7 , 6 . 7 9) : 1 , (5 . 0 8 , 3. 5 1 , 4 . 6 9) : 3 , (5 . 1 5 , 3. 5 8 , 4 . 2) : 3 , (5 . 6 7 , 2. 2 7 , 4 . 6 5) : 3 , (5 . 6 7 , 3. 8 1 , 5 . 7 5) : 3 , (5 . 9 4 , 2. 3 4 , 4 . 1 2) : 3 , (6 . 0 6 , 3. 1 6 , 4 . 3 6) : 3 , (6 . 0 9 , 3. 1 9 , 4 . 0 2) : 3 , (6 . 4 3 , 3. 4 2 , 4 . 1 8) : 3 , (6 . 5 6 , 2 . 7 , 4 . 0 3) : 3 , (6 . 7 9 , 3. 4 6 , 4 . 8 1) : 3}

For instance, the first point has coordinates (x1, x2, x3) = (2.0,3.43,4.37)and belongs to class 2. In total we have threeclasses: 1, 2, and 3.

As a discriminant function, consider a distance function based on below center coordinates (encoded as a dictionary of values) for each class labels

centers_dict={} centers_dict [(3,4,5)] = 1 # center coordinates for class 1 , i.e . , c1 =4, c2 =5, c3=6

centers_dict [(4,5,6)] = 2 # center coordinates for class 2 , i.e . , c1 =3, c2 =4, c3=5

centers_dict [(6,3,5)] = 3 # center coordinates for class 3 ,i.e . , c1 =6, c2 =3, c3=5

Note that a discriminant function based on Minkowski distance can be written as g(x) = (n ∑ i=1 |ci −xi | p) 1/p where xi = (x1, x2, x3), ci = (c1, c2, c3)

Based on above discriminant functions, perform a K-Means Clustering task over 30 points in data_dict and then compare it with true labels. What is the number of correctly classified instances for each value of p in distance measure?

Double-click (or enter) to edit

```
data_dict = {(2.0, 3.43, 4.37): 2, (2.49, 4.28, 4.83): 2, (2.58, 4.36, 4.48): 2, (2.66, 4.45,
(2.82, 3.66, 4.51): 2, (3.03, 4.37, 5.07): 2, (3.27, 4.54, 4.57): 2, (3.41, 3.94, 5.35): 2,
(3.53, 4.32, 5.41): 2, (3.53, 4.6, 6.8): 1, (3.61, 4.25, 5.21): 1, (3.61, 4.78, 5.47): 1,
(3.72, 5.44, 5.88): 1, (3.87, 4.96, 4.52): 2, (4.13, 5.29, 6.6): 1, (4.25, 5.97, 5.48): 1,
(4.61, 4.9, 5.11): 1, (4.73, 4.4, 6.78): 1, (4.97, 4.25, 5.0): 1, (4.98, 5.27, 6.79): 1,
(5.08, 3.51, 4.69): 3, (5.15, 3.58, 4.2): 3, (5.67, 2.27, 4.65): 3, (5.67, 3.81, 5.75): 3,
(5.94, 2.34, 4.12): 3, (6.06, 3.16, 4.36): 3, (6.09, 3.19, 4.02): 3, (6.43, 3.42, 4.18): 3,
(6.56, 2.7, 4.03): 3, (6.79, 3.46, 4.81): 3}
```

```
import numpy as np

# initialize cluster centers
centers_dict = {(3, 4, 5): 1, (4, 5, 6): 2, (6, 3, 5): 3}
centers = [np.array(coord) for coord in centers_dict.keys()]

# assign each point to its nearest centroid
def assign_clusters(data_dict, centers, p):
    clusters = {}
    for point, label in data_dict.items():
        point_arr = np.array(point)
        distances = [np.sum(np.abs(center - point_arr) ** p) ** (1/p) for center in centers]
        nearest_cluster = np.argmin(distances) + 1
        clusters[point] = nearest_cluster
    return clusters

# update centroids to be the mean of the points assigned to them
def update_centers(data_dict, clusters):
    new_centers = []
    for i in range(1, 4):
        cluster_points = [np.array(point) for point, label in clusters.items() if label == i]
        if cluster_points:
            new_center = np.mean(cluster_points, axis=0)
        else:
            new_center = np.zeros(3) # if no points in cluster, set center to (0,0,0)
        new_centers.append(new_center)
    return new_centers

# perform K-Means Clustering for a given value of p
def kmeans(data_dict, centers_dict, p):
    centers = [np.array(coord) for coord in centers_dict.keys()]
    while True:
        clusters = assign_clusters(data_dict, centers, p)
        new_centers = update_centers(data_dict, clusters)
        if np.allclose(centers, new_centers):
            break
        centers = new_centers
    return clusters

# test with p=1
```

```
clusters_p1 = kmeans(data_dict, centers_dict, 1)

# count number of correctly classified instances
correct_p1 = sum([label == true_label for (point, label), true_label in zip(clusters_p1.items
print("Number of correctly classified instances for p=1:", correct_p1)

# test with p=2
clusters_p2 = kmeans(data_dict, centers_dict, 2)

# count number of correctly classified instances
correct_p2 = sum([label == true_label for (point, label), true_label in zip(clusters_p2.items
print("Number of correctly classified instances for p=2:", correct_p2)

# test with p=3
clusters_p3 = kmeans(data_dict, centers_dict, 3)

# count number of correctly classified instances
correct_p3 = sum([label == true_label for (point, label), true_label in zip(clusters_p3.items
print("Number of correctly classified instances for p=3:", correct_p3)
```

```
Number of correctly classified instances for p=1: 13
Number of correctly classified instances for p=2: 11
Number of correctly classified instances for p=3: 11
```

✓ 0s completed at 11:44 AM