1. Build a classifier, compare its performance with an ensemble technique like random forest.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

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

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Decision Tree Classifier
decision_tree_clf = DecisionTreeClassifier(random_state=42)
decision_tree_clf.fit(X_train, y_train)
y_pred_tree = decision_tree_clf.predict(X_test)
accuracy_tree = accuracy_score(y_test, y_pred_tree)
print("Decision Tree accuracy:", accuracy_tree)

# Random Forest Classifier
random_forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest_clf.fit(X_train, y_train)
y_pred_forest = random_forest_clf.predict(X_test)
accuracy_forest = accuracy_score(y_test, y_pred_forest)
print("Random Forest accuracy:", accuracy_forest)
```

```
Decision Tree accuracy: 1.0
Random Forest accuracy: 1.0
```

2. Evaluate various classification algorithms performance on a dataset using various measures like True Positive rate, False Positive rate, precision, recall etc.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.preprocessing import label_binarize
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

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

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Binarize the output (required for some metrics like ROC)
y_bin = label_binarize(y_test, classes=[0, 1, 2])

# Initialize classifiers
classifiers = {
    "Decision Tree": DecisionTreeClassifier(random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    "Logistic Regression": LogisticRegression(max_iter=200, random_state=42),
    "SVM": SVC(probability=True, random_state=42)
}

# Train classifiers
for name, clf in classifiers.items():
    clf.fit(X_train, y_train)

# Function to calculate TPR and FPR
def calculate_tpr_fpr(conf_matrix):
    TP = conf_matrix[1, 1]
    FN = conf_matrix[1, 0]
    FP = conf_matrix[0, 1]
    TN = conf_matrix[0, 0]
    TPR = TP / (TP + FN)
    FPR = FP / (FP + TN)
    return TPR, FPR

# Evaluate classifiers
results = []
for name, clf in classifiers.items():
    y_pred = clf.predict(X_test)
    y_proba = clf.predict_proba(X_test) if hasattr(clf, "predict_proba") else None
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='macro')
    recall = recall_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')
```

```python
    # Only calculate ROC AUC if y_proba is available
    if y_proba is not None:
    # Use "ovr" (one-vs-rest) strategy for multi-class ROC AUC
        roc_auc = roc_auc_score(y_bin, y_proba, multi_class="ovo", average="macro")
    else:
        roc_auc = "N/A"
    conf_matrix = confusion_matrix(y_test, y_pred)
    tpr, fpr = calculate_tpr_fpr(conf_matrix)
    results.append({
        "Classifier": name,
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1-Score": f1,
        "ROC AUC": roc_auc,
        "TPR": tpr,
        "FPR": fpr })

# Convert results to a DataFrame for better visualization
results_df = pd.DataFrame(results)
pd.set_option('display.max_columns', None)
print(results_df)
```

```
            Classifier  Accuracy  Precision  Recall  F1-Score  ROC AUC  TPR  \
0        Decision Tree       1.0        1.0     1.0       1.0      1.0  1.0
1        Random Forest       1.0        1.0     1.0       1.0      1.0  1.0
2  Logistic Regression       1.0        1.0     1.0       1.0      1.0  1.0
3                  SVM       1.0        1.0     1.0       1.0      1.0  1.0

   FPR
0  0.0
1  0.0
2  0.0
3  0.0
```

3. Demonstrate GA for optimization(minimization or maximization problem).

```
import numpy as np

# Define the Rastrigin function
def rastrigin(X):
    A = 10
    return A * len(X) + sum([(x**2 - A * np.cos(2 * np.pi * x)) for x in X])

# Initialize population
def initialize_population(pop_size, dimensions, bounds):
    population = []
    for _ in range(pop_size):
        individual = np.random.uniform(bounds[0], bounds[1], dimensions)
        population.append(individual)
    return population

# Evaluate fitness
def evaluate_population(population):
    return [rastrigin(individual) for individual in population]

# Select parents (roulette wheel selection)
def select_parents(population, fitness, num_parents):
    fitness = np.array(fitness)
    probs = fitness / fitness.sum()
    selected_indices = np.random.choice(len(population), size=num_parents, p=probs)
    return [population[i] for i in selected_indices]

# Crossover (single-point crossover)
def crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        if i+1 < len(parents):
            crossover_point = np.random.randint(1, len(parents[i]))
            child1 = np.concatenate((parents[i][:crossover_point], parents[i+1][crossover_point:]))
            child2 = np.concatenate((parents[i+1][:crossover_point], parents[i][crossover_point:]))
            offspring.append(child1)
            offspring.append(child2)
    return offspring
```

```python
# Genetic Algorithm
def genetic_algorithm(pop_size, dimensions, bounds, generations, mutation_rate):
    population = initialize_population(pop_size, dimensions, bounds)
    for generation in range(generations):
        fitness = evaluate_population(population)
        parents = select_parents(population, fitness, pop_size // 2)
        offspring = crossover(parents)
        offspring = mutate(offspring, mutation_rate, bounds)
        population = parents + offspring
        print(f"Generation {generation}: Best fitness = {min(fitness)}")
    best_individual = population[np.argmin(evaluate_population(population))]
    return best_individual

# Parameters
pop_size = 50
dimensions = 5
bounds = [-5.12, 5.12]
generations = 100
mutation_rate = 0.1
# Run GA
best_solution = genetic_algorithm(pop_size, dimensions, bounds, generations, mutation_rate)
print(f"Best solution: {best_solution}, Best fitness: {rastrigin(best_solution)}")
```

```
Generation 0: Best fitness = 36.68820117821527
Generation 1: Best fitness = 36.68820117821527
Generation 2: Best fitness = 53.66591205777277
Generation 3: Best fitness = 55.75031085423464
Generation 4: Best fitness = 55.75031085423464
Generation 5: Best fitness = 55.75031085423464
Generation 6: Best fitness = 59.23308376744469
Generation 7: Best fitness = 63.69733547178184
Generation 8: Best fitness = 59.23308376744469
Generation 9: Best fitness = 69.50323493503399
Generation 10: Best fitness = 86.51591836081982
Generation 11: Best fitness = 90.89891593970793
Generation 12: Best fitness = 90.89891593970793
Generation 13: Best fitness = 90.89891593970793
Generation 14: Best fitness = 99.08262747044364
Generation 15: Best fitness = 92.89832318626239
Generation 16: Best fitness = 103.84564428851657
Generation 17: Best fitness = 98.21810549156811
Generation 18: Best fitness = 102.5762148786591
Generation 19: Best fitness = 106.57774237664455
Generation 20: Best fitness = 110.87696739337306
Generation 21: Best fitness = 101.75882797127363
Generation 22: Best fitness = 101.75882797127363
Generation 23: Best fitness = 101.75882797127363
Generation 24: Best fitness = 115.25791437353864
Generation 25: Best fitness = 118.80977429518988
Generation 26: Best fitness = 119.9300295458205
Generation 27: Best fitness = 110.19758979133972
Generation 28: Best fitness = 117.85802600037718
Generation 29: Best fitness = 117.85802600037718
Generation 30: Best fitness = 117.92454959363927
Generation 31: Best fitness = 122.03337833708194
Generation 32: Best fitness = 122.03337833708194
Generation 33: Best fitness = 127.51035976187565
Generation 34: Best fitness = 125.64541846150048
```

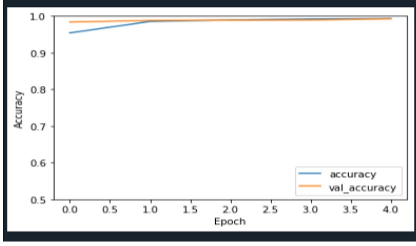
```
Generation 36: Best fitness = 127.51035976187565
Generation 37: Best fitness = 127.51035976187565
Generation 38: Best fitness = 127.51035976187565
Generation 39: Best fitness = 120.44413550155387
Generation 40: Best fitness = 127.51035976187565
Generation 41: Best fitness = 111.28335790086271
Generation 42: Best fitness = 127.51035976187565
Generation 43: Best fitness = 127.51035976187565
Generation 44: Best fitness = 111.53992836052318
Generation 45: Best fitness = 111.53992836052318
Generation 46: Best fitness = 118.79727681225174
Generation 47: Best fitness = 118.79727681225174
Generation 48: Best fitness = 118.79727681225174
Generation 49: Best fitness = 118.79727681225174
Generation 50: Best fitness = 118.79727681225174
Generation 51: Best fitness = 112.21419506283313
Generation 52: Best fitness = 118.79727681225174
Generation 53: Best fitness = 112.89732367546388
Generation 54: Best fitness = 123.5071796032803
Generation 55: Best fitness = 119.52531793344735
Generation 56: Best fitness = 123.5071796032803
Generation 57: Best fitness = 123.5071796032803
Generation 58: Best fitness = 123.5071796032803
Generation 59: Best fitness = 123.5071796032803
Generation 60: Best fitness = 94.26488818352477
Generation 61: Best fitness = 123.5071796032803
Generation 62: Best fitness = 121.82660383617088
Generation 63: Best fitness = 122.67715666537508
Generation 64: Best fitness = 122.15180337785114
Generation 65: Best fitness = 122.15180337785114
Generation 66: Best fitness = 129.52135732588985
Generation 67: Best fitness = 129.52135732588985
Generation 68: Best fitness = 129.52135732588985
Generation 69: Best fitness = 129.52135732588985
Generation 70: Best fitness = 123.11509639232845
Generation 71: Best fitness = 117.34085855390211
```

```
Generation 72: Best fitness = 132.9964273396452
Generation 73: Best fitness = 119.92728410998734
Generation 74: Best fitness = 120.46399283380627
Generation 75: Best fitness = 120.46399283380627
Generation 76: Best fitness = 120.28722699666513
Generation 77: Best fitness = 122.14374075181475
Generation 78: Best fitness = 129.35425198623267
Generation 79: Best fitness = 127.34786054732868
Generation 80: Best fitness = 119.84242121514842
Generation 81: Best fitness = 122.81146708841618
Generation 82: Best fitness = 119.75820565169887
Generation 83: Best fitness = 114.54050156870298
Generation 84: Best fitness = 127.34786054732868
Generation 85: Best fitness = 127.34786054732868
Generation 86: Best fitness = 127.34786054732868
Generation 87: Best fitness = 127.34786054732868
Generation 88: Best fitness = 127.98811423808823
Generation 89: Best fitness = 127.98811423808823
Generation 90: Best fitness = 125.3784964337156
Generation 91: Best fitness = 127.98811423808823
Generation 92: Best fitness = 114.52605365857733
Generation 93: Best fitness = 115.36738715061395
Generation 94: Best fitness = 108.10505390168468
Generation 95: Best fitness = 120.20536394139933
Generation 96: Best fitness = 115.57323269667874
Generation 97: Best fitness = 115.46717006246723
Generation 98: Best fitness = 123.59507435346053
Generation 99: Best fitness = 121.1426372211482
Best solution: [ 1.6480014  -1.81215877 -1.48397149  0.85150772  3.5056891 ], Best fitness:
87.37759646796857
```

4. Case study on supervised/unsupervised learning algorithm: Hand written digits classification using CNN.

```python
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape the data to include a channel dimension
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))

#Build the CNN Model:
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Add Dense layers on top
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

#Compile the Model
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])

#Train the Model
history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

#Evaluate the Model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')

#Plot Training History
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 17s 8ms/step - accuracy: 0.8930 - loss: 0.3446 - val_accuracy:
0.9836 - val_loss: 0.0497
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - accuracy: 0.9845 - loss: 0.0494 - val_accuracy:
0.9880 - val_loss: 0.0394
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 9ms/step - accuracy: 0.9892 - loss: 0.0329 - val_accuracy:
0.9888 - val_loss: 0.0345
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9919 - loss: 0.0252 - val_accuracy:
0.9893 - val_loss: 0.0370
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9937 - loss: 0.0196 - val_accuracy:
0.9927 - val_loss: 0.0268
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9890 - loss: 0.0369
Test accuracy: 0.9926999807357788
```



5. Case study on supervised/unsupervised learning algorithm: Text classification using

a) Scikit-learn

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report

# Load the dataset
df = pd.read_csv('Corona_NLP_test.csv')

X = df['OriginalTweet'].values  # Extracting the 'text' column as input features
y = df['Sentiment'].values  # Extracting the 'label' column as target labels


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
vectorizer = TfidfVectorizer(max_features=1000)  # Adjust max_features as needed
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

#Model Training and Evaluation:**
#Choose a classifier (e.g., SVM) and train it on the extracted features.
clf = SVC()
clf.fit(X_train_tfidf, y_train)
y_pred = clf.predict(X_test_tfidf)
print(classification_report(y_test, y_pred))
```

```
                   precision    recall  f1-score   support

Extremely Negative       0.68      0.23      0.35       184
Extremely Positive       0.74      0.18      0.29       192
          Negative       0.34      0.63      0.44       299
           Neutral       0.62      0.25      0.36       193
          Positive       0.31      0.45      0.37       272

          accuracy                           0.38      1140
         macro avg       0.54      0.35      0.36      1140
      weighted avg       0.50      0.38      0.37      1140
```

b) Tensorflow/Keras

```python
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

#Load and Preprocess the Data
df = pd.read_csv('IMDB.csv')

X = df['review'].values
y = df['sentiment'].values

# Convert string labels to numeric labels using LabelEncoder
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

#Tokenization and Padding
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(X_train)
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
X_train_pad = pad_sequences(X_train_seq, maxlen=100)
X_test_pad = pad_sequences(X_test_seq, maxlen=100)

#Model Building:
model = Sequential([
       Embedding(10000, 32),    #removed input length for warning
       LSTM(64),
       Dense(1, activation='sigmoid')
   ])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

#Model Training:
model.fit(X_train_pad, y_train, epochs=5, validation_data=(X_test_pad, y_test))

#Model Evaluation
loss, accuracy = model.evaluate(X_test_pad, y_test)
print(f'Test Accuracy: {accuracy}')
```

```
Epoch 1/5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 58s 51ms/step - accuracy: 0.7451 - loss: 0.5001 - val_accuracy:
0.8393 - val_loss: 0.3603
Epoch 2/5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 34s 31ms/step - accuracy: 0.8447 - loss: 0.3553 - val_accuracy:
0.8424 - val_loss: 0.3529
Epoch 3/5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 34s 31ms/step - accuracy: 0.8533 - loss: 0.3327 - val_accuracy:
0.8424 - val_loss: 0.3508
Epoch 4/5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 34s 31ms/step - accuracy: 0.8697 - loss: 0.3052 - val_accuracy:
0.8559 - val_loss: 0.3262
Epoch 5/5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 35s 32ms/step - accuracy: 0.8769 - loss: 0.2845 - val_accuracy:
0.8597 - val_loss: 0.3289
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - accuracy: 0.8588 - loss: 0.3266
Test Accuracy: 0.8597333431243896
```