Pattern Recognition and Machine Learning assignment

January 2025
Group 40

Fotiadis Konstantinos
Bakoulas Epameinondas

Part A - A1

Our goal is to build a Maximum Likelihood Classifier (MLE) to estimate the θ parameters of the pdf function

$$p(x|\theta) = \frac{1}{\pi(1 + (x - \theta)^2)}$$

for each class ω1 (no stress) and ω2 (stress) given the samples D1 and D2.

Part A - A1

The likelihood is given as

$$p(D|\theta) = \prod p(x_n|\theta)$$

but since the logarithm is a monotonically increasing function, we will use the log-likelihood which is given as

$$l(\theta) = \log(p(D|\theta))$$

Part A - A1

In order to find the θ value that maximizes the log-likelihood (and the likelihood), we will pass a large range of theta values and return the one that maximizes the log-likelihood.

We follow this approach since calculating the gradient of the log-likelihood is computationally difficult.

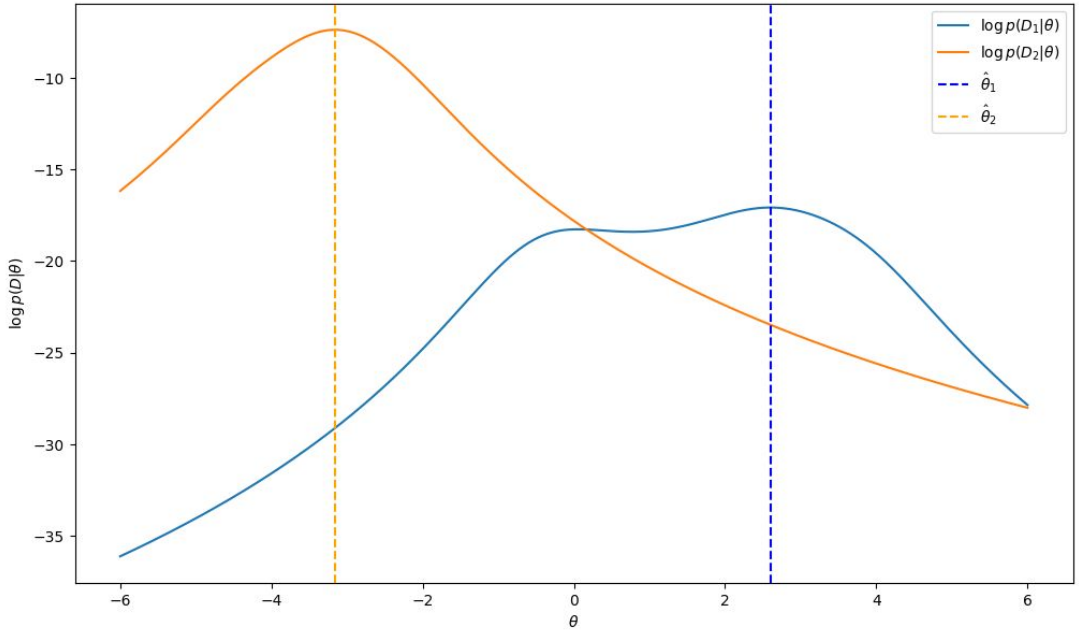
We need to select a range of theta values that will likely maximize the log-likelihood. Since our data (D1, D2) range from [-4.5, 4.1], we are going to select a slightly bigger range **[-6, 6]** to make sure that the theta value that maximizes the log-likelihood is included.

Part A - A1 Code

```python
def pdf (x, theta):
    return 1 / (np.pi * (1 + (x - theta)**2))

def log_likelihood (D, theta):
    return np.sum(np.log(pdf(D, theta)))

def fit (D, theta_values):
    likelihood_values = np.array([log_likelihood(D, theta) for theta in theta_values])
    return theta_values[np.argmax(likelihood_values)]

theta_values = np.linspace(-6, 6, 1000)
D1 = np.array([2.8, -0.4, -0.8, 2.3, -0.3, 3.6, 4.1])
D2 = np.array([-4.5, -3.4, -3.1, -3.0, -2.3])

theta_hat_1 = fit(D1, theta_values)
theta_hat_2 = fit(D2, theta_values)
```

Part A - A1

Now we can plot the log-likelihood for each class and for different θ values, as well as the θ1 and θ2 values that maximizes the log-likelihoods.



Part A - A2

Now we're tasked with using the linear discriminant function

$$g(x) = \log P(x|\hat{\theta}_1) - \log P(x|\hat{\theta}_2) + \log(P(\omega_1)) - \log(P(\omega_2))$$

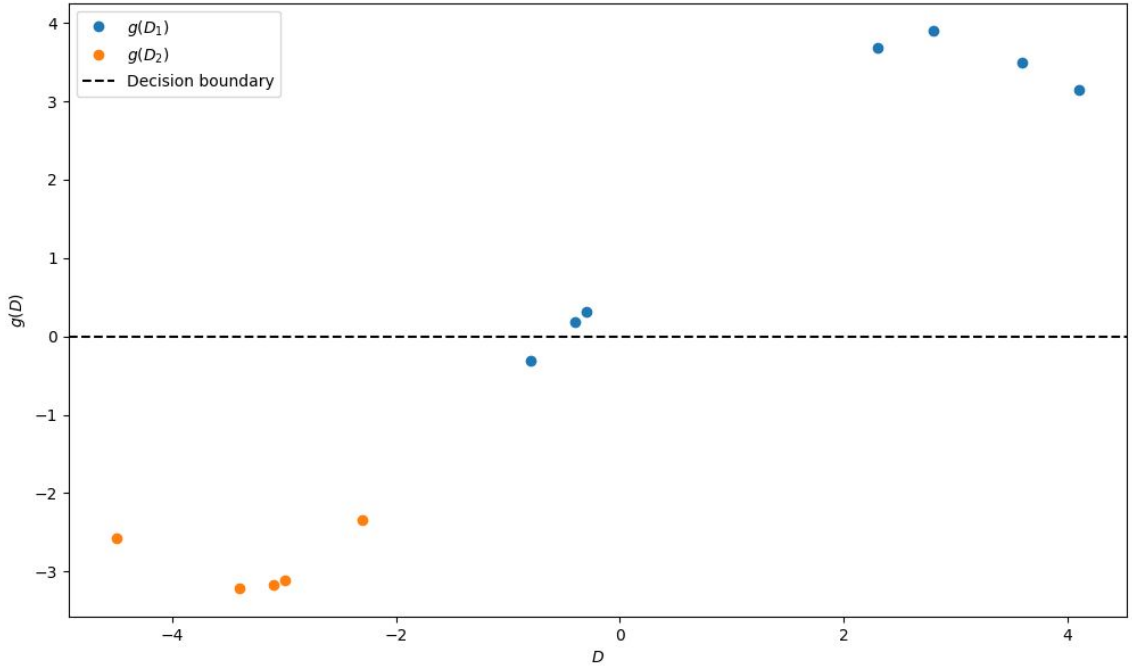
to classify the 2 datasets.

We will first calculate the a-priori probabilities for each class. We have 7 samples in class ω1 and 5 samples in class ω2 (12 total), so the a-priori probabilities are calculated as:

$$P(\omega_1) = \frac{\text{len}(D_1)}{\text{len}(D_1) + \text{len}(D_2)} \qquad P(\omega_2) = \frac{\text{len}(D_2)}{\text{len}(D_1) + \text{len}(D_2)}$$

Part A - A2

We can now calculate all of the g values and plot them. The decision boundary is g(x) = 0



Part A - A2

For the classification to be correct we need to:

- Assign x in class ω1 (no stress) if $g(x) > 0$

- Assign x in class ω2 (stress) if $g(x) < 0$

while the decision boundary is $g(x) = 0$.

We can clearly see that the classification is fairly accurate, since **11 out of 12 points are classified correctly.** Only one point from class ω1 is not classified correctly.

We could try to change the discriminant function so it achieves perfect classification of our samples, but this might lead to **overfitting** which is not desirable since the model might not generalize well and thus perform poorly on different sample data.

Part A - A2 Code

```
apriori_P1 = len(D1) / (len(D1) + len(D2))
apriori_P2 = len(D2) / (len(D1) + len(D2))

def predict (D, apriori_P1, apriori_P2, theta1, theta2):
    return np.log(pdf(D, theta1)) - np.log(pdf(D, theta2)) + np.log(apriori_P1) - np.log(apriori_P2)

g_values_D1 = predict(D1, apriori_P1, apriori_P2, theta_hat_1, theta_hat_2)
g_values_D2 = predict(D2, apriori_P1, apriori_P2, theta_hat_1, theta_hat_2)
```

Part B - B1

In this part we will make a Bayesian estimator. Firstly we calculate the posterior and the prior probabilities.

Prior

$$p(\theta) = \frac{1}{10\pi(1 + (\frac{\theta}{10})^2}$$

Posterior

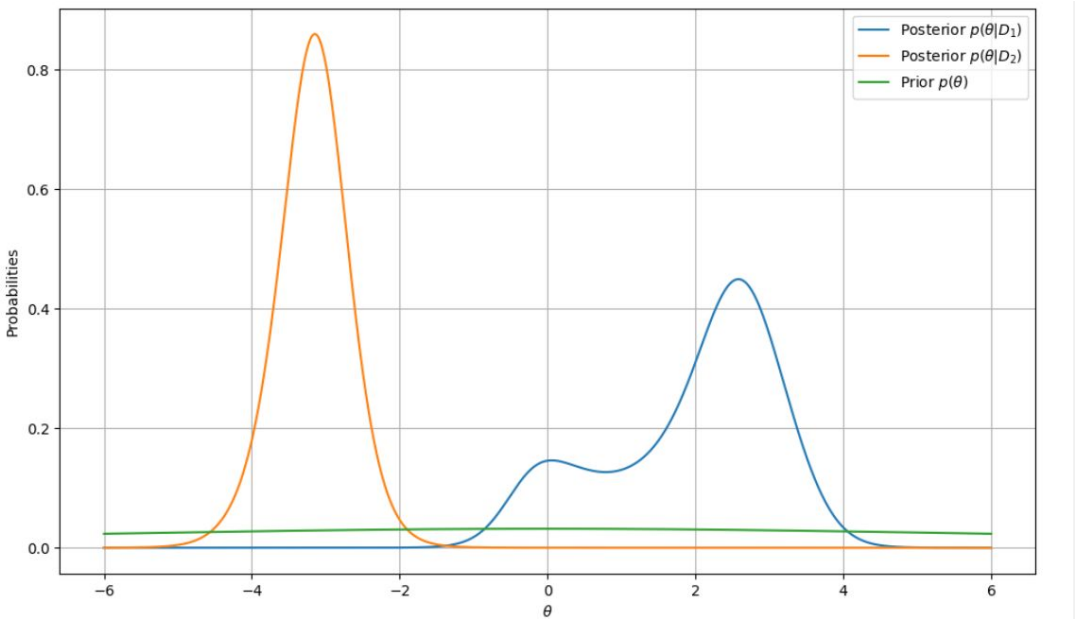
$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)d\theta}$$

Part B - B1 Code

```
def prior(theta):
    return 1 / (10 * np.pi * (1 + (theta / 10) ** 2))

def likelihood (D, theta):
    return np.prod(pdf(D, theta))

def posterior (D, theta_values):
    likelihood_values = np.array([likelihood(D, theta) for theta in theta_values])
    prior_values = prior(theta_values)
    posterior_unormalized = likelihood_values * prior_values
    return posterior_unormalized / np.trapezoid(posterior_unormalized, theta_values)
```

Part B - B1

Now we can plot the the prior and posterior probabilities.



Part B - B2

We will now implement the predict function which is given as:

$$h(x) = \log P(x|D_1) - \log P(x|D_2) + \log P(\omega_1) - \log P(\omega_2)$$

where $P(x|D)$ is calculated by the integral:

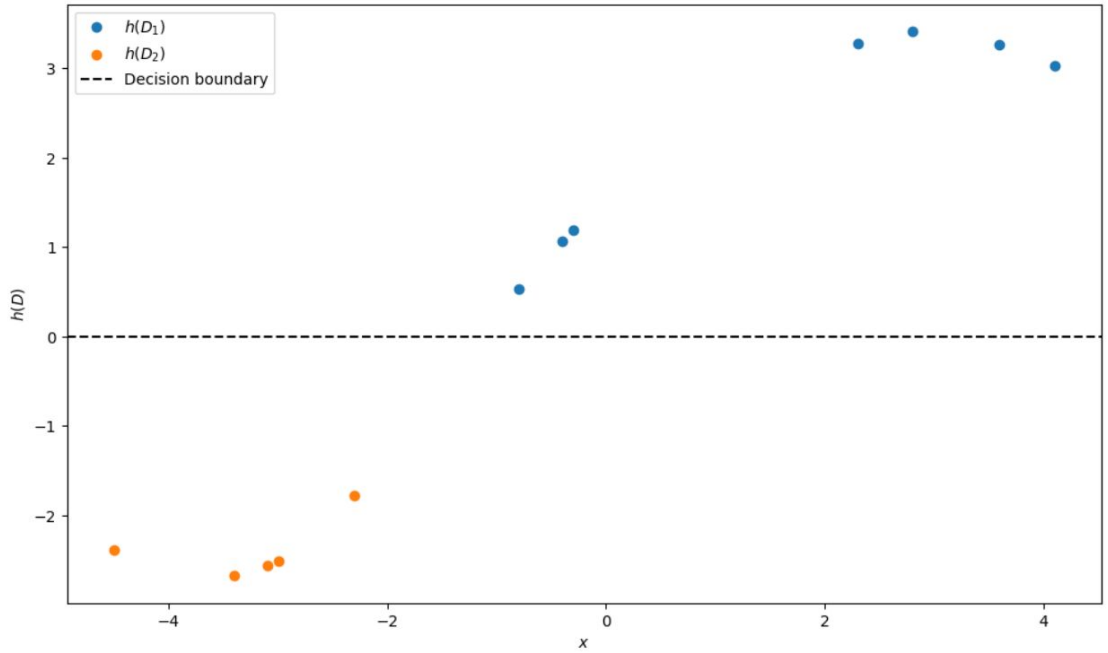
$$P(x|D) = \int p(x|\theta)p(\theta|D)\, d\theta$$

Part B - B2 Code

```python
def p_x_given_D (x, D, theta_values):
    pdf_values = pdf(x, theta_values)
    posterior_values = posterior(D, theta_values)
    integral = [pdf(x, theta) * posterior_values[i] for i, theta in enumerate(theta_values)]
    return np.trapezoid(integral, theta_values)

def predict (x, apriori_P1, apriori_P2, theta_values):
    p_x_given_D1 = p_x_given_D(x, D1, theta_values)
    p_x_given_D2 = p_x_given_D(x, D2, theta_values)
    return np.log(p_x_given_D1) - np.log(p_x_given_D2) + np.log(apriori_P1) - np.log(apriori_P2)
```

Part B - B2

We can now plot the predict function h(x). The decision boundary is again at h(x) = 0.



Part B - B2

- This time the classification achieves 100% accuracy, since for each point in the dataset D1 we have $h(x)>0$ and for the points in dataset D2 we have $h(x)<0$.

- The Bayesian Estimation seems like the better approach since it incorporates the prior knowledge about the parameter θ through the prior distribution $p(\theta)$, leading to better parameter estimations. In contrast, the MLE only relies on our dataset and tries to maximize the likelihood $p(D|\theta)$, without taking into account any information about the prior distribution of our parameter θ.

Part B - B2

- The BE accounts for all possible values of θ compared to MLE that estimates a single point $\hat{\theta}$ that maximizes the likelihood.

- It's important to note that the BE is a lot more computationally complex and expensive compared to MLE, so even though BE leads to better results, it might not be worth implementing in practice (especially for bigger multi-dimensional problems).

Part C - C1.1

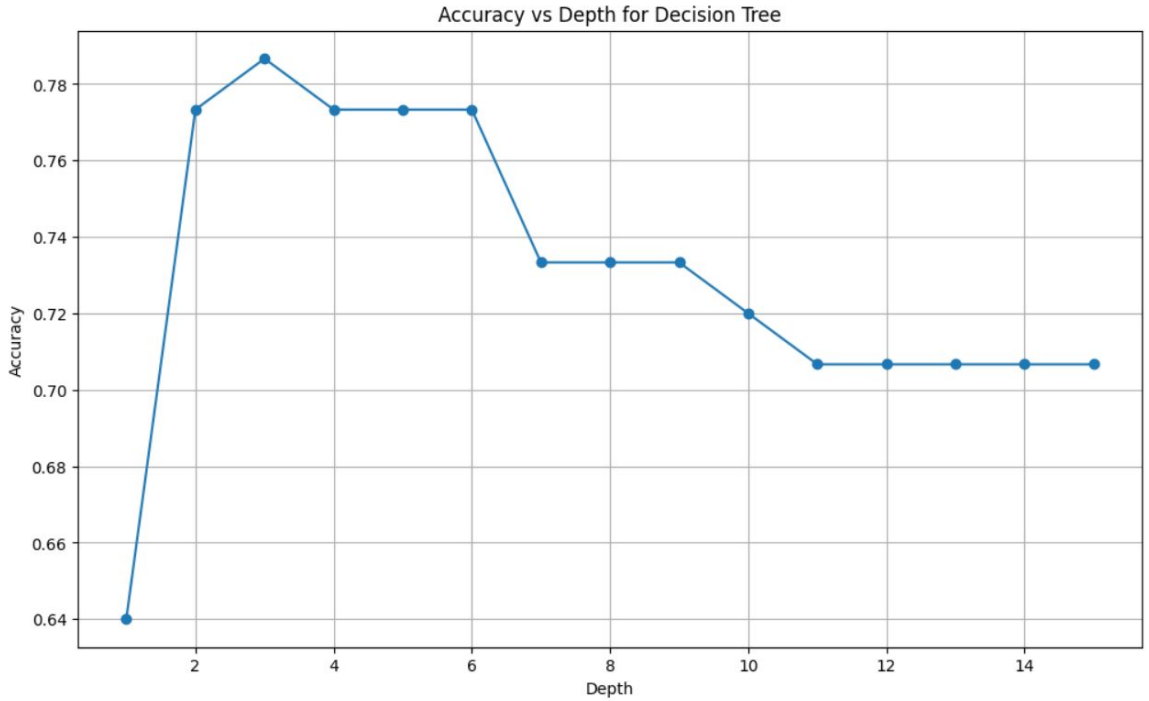
In this section we will train the Decision Tree Classifier algorithm with the first half of the first 2 features from the Iris dataset and then use it to classify the other half.

We will first iterate over the depth values **[1,15]** in order to find optimal depth that provides the best accuracy. The depth and accuracy values are saved in an array to plot them later on.

For each depth, we create a new DecisionTreeClassifier and we fit the training data. Then, the model predicts the labels for the test dataset and calculates the accuracy using the accuracy_score function. If the accuracy is higher than the previous best accuracy, we update best_accuracy and best_depth with the new values.

Part C - C1.1

Now we are going to plot the accuracy for each depth value.



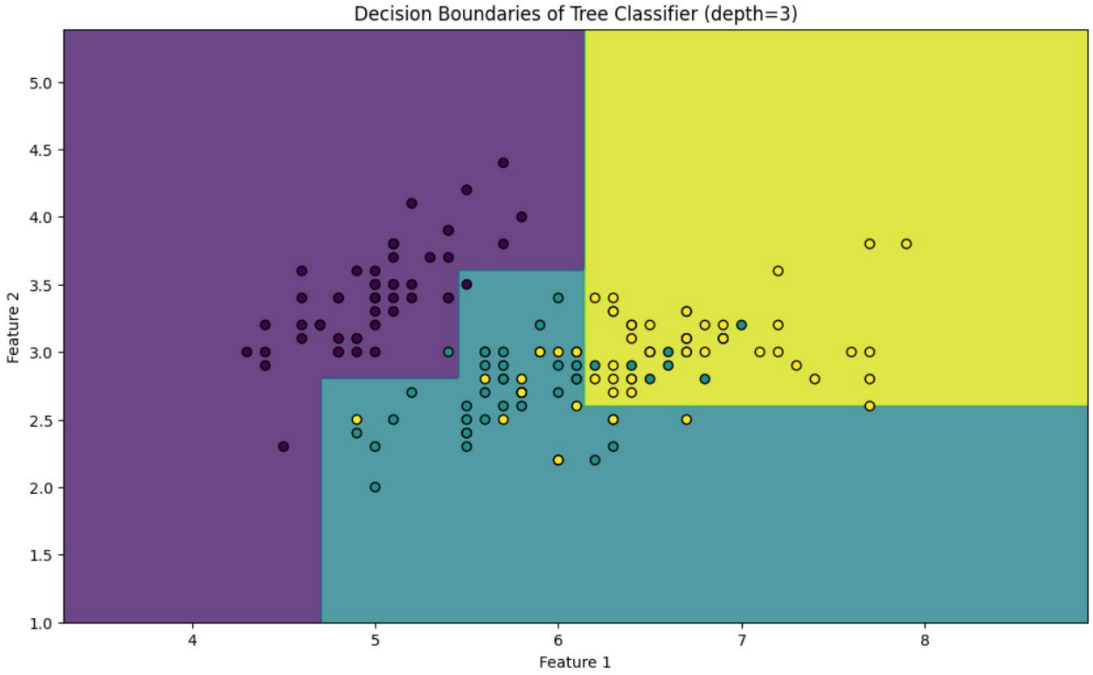
Part C - C1.1

We can clearly see that the highest accuracy is for **depth=3** with a value of **0.7866**. The accuracy has a significant drop at **depth=7**, indicating that there's **overfitting** and the model fails to generalize. Thus, it is unable to perform well on unseen data.

In our case the training set is relatively small (150 samples) so large depth values are not suitable. Keeping the depth to low values between 2–5 is great option if we want to avoid both overfitting and underfitting.

Part C - C1.2

Displaying the decision boundaries for our best depth (=3) and the points from both the training and test sets, we can see that the model is neither overfitting or underfitting.



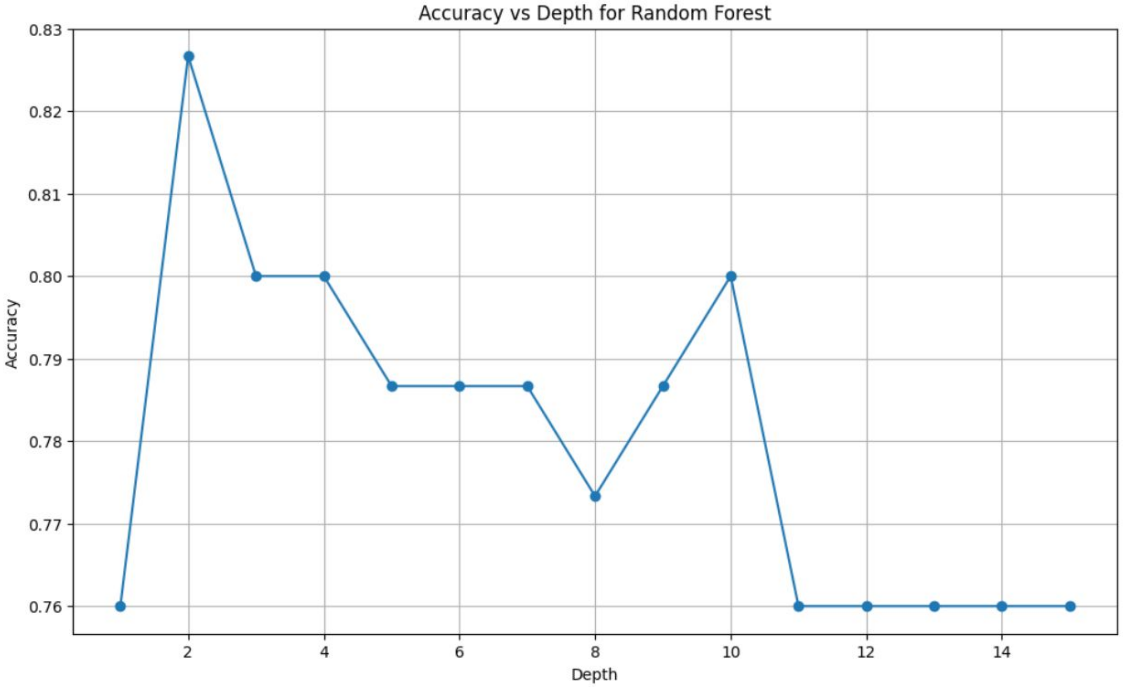
Part C - C2.1

In this section we are using Random Forest Classifier with bootstrap enabled, that creates 100 trees using 50% of the training points.

Just like before, we're going to iterate over the depth values **[1,15]** and for each one create a random forest with **γ=0.5**. Our aim is to find the depth with the highest accuracy.

Part C - C2.1

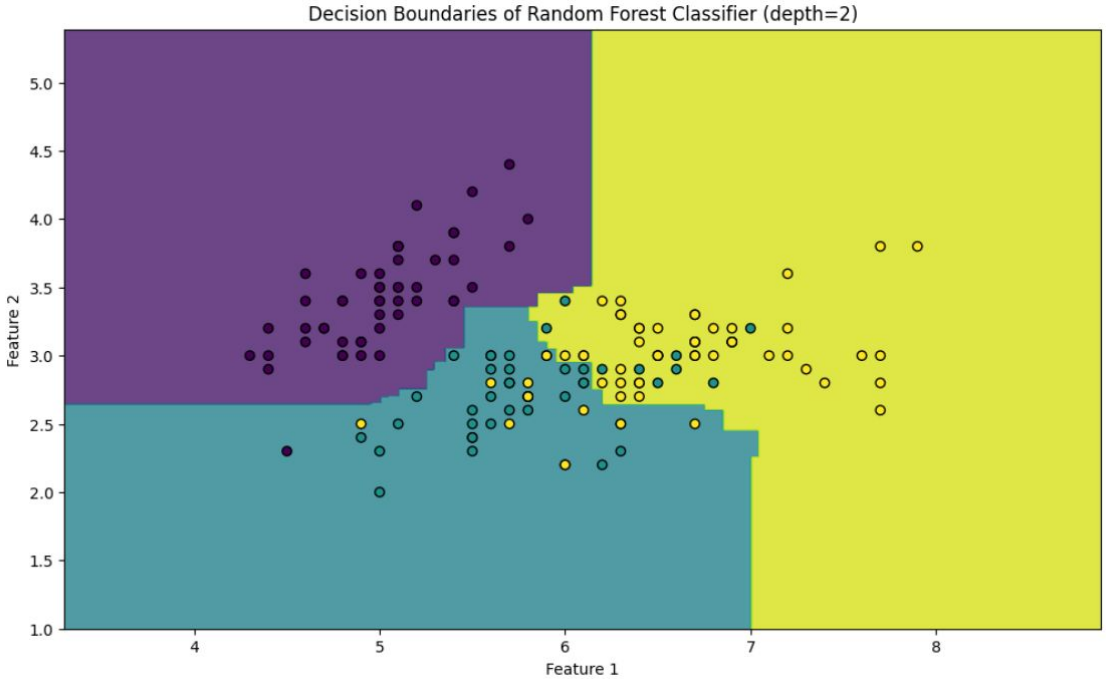
We plot again the accuracy for each depth value.



The highest accuracy achieved with the Random Forest Classifier is **0.8266** with **depth=2**. From the plot we can see for higher depth values the accuracy drops since we're overfitting. For depth=1 we have underfitting since the depth is too small.

Part C - C2.2

We display again the decision boundaries for the best depth (=2).



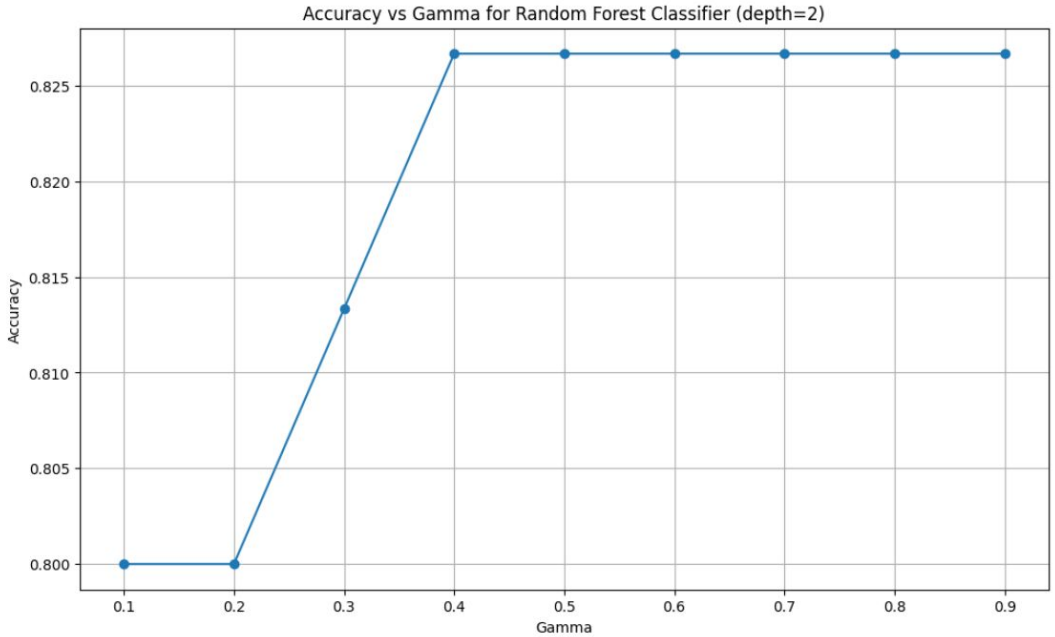
Part C - C2.3

Comparing the 2 decision boundaries we can conclude that the random forest classifier with γ = 0.5 achieves **higher accuracy** than the single decision tree classifier.

The decision tree produces more rectangular boundaries while the random forest, by combining multiple trees, generates **smoother and more flexible boundaries**, better capturing complex patterns in the data. Using γ=0.5 ensures a balance between diversity and accuracy by training each tree on 50% of the data. This configuration avoids overfitting while maintaining a sufficient amount of information for accurate predictions.

Part C - C2.3

Lastly, we're going to test the effect of γ for a specific depth value (e.g. 3) by printing the accuracy for various γ.



Part C - C2.3

In the Random Forest algorithm, the parameter γ represents the proportion of the training dataset used for each bootstrap sample. It directly influences the diversity and robustness of the model.

Our expected performance based on the γ value is the following:

Small γ (e.g., 20%)

- Pros: High diversity among trees due to smaller sample sizes. Reduces overfitting, especially for noisy datasets.
- Cons: May lead to underfitting since individual trees are trained on limited data.

Medium γ (e.g., 50%)

- Pros: Balanced approach between diversity and training data size. Provides sufficient information for each tree to learn patterns while retaining diversity.
- Cons: Might not capture highly complex patterns if data is too sparse.

Large γ (e.g., 80%)

- Pros: Higher accuracy in training, as each tree learns from a larger dataset. Suitable for large, clean datasets with low noise.
- Cons: Reduces diversity, leading to overfitting. Poor generalization to unseen data.

In our case the accuracy fluctuates between 0.8 and 0.8266, reaching its peak at **γ=40%**. The accuracy stays constant after that, indicating no real preference for a specific γ value. This could be because the dataset is relatively small and the model is not very complex.A typical default value of **γ = 0.5** often balances accuracy and generalization and leads to good results.

Part D

Our goal in this assignment is to train our own model in order to accurately predict the labels from a test set.

We first import the datasets into our project

```
train_data = pd.read_csv('datasetTV.csv', header=None)
test_data = pd.read_csv('datasetTest.csv', header=None)

X_train = train_data.iloc[:, :-1].values # All columns except the last one
y_train = train_data.iloc[:, -1].values # Last column is the target
X_test = test_data.values
```

Part D

After importing, we transform the values of the features using **StandardScaler** to ensure that the model is not biased towards any feature. The main idea behind it is to transform our data such that its distribution will have a mean value 0 and standard deviation of 1.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Part D

Before we start training our models, we will do some **hyperparameter tuning** to find the best hyperparameters for each model.

We will use **GridSearchCV** for this purpose. It performs an exhaustive search over specified parameter values for an estimator. The accuracy of the model is calculated using **5-fold cross-validation.**

Part D

```python
param_grids = {
    'k-NN': {
        'n_neighbors': [3, 5, 7, 9, 13, 17],
        'weights': ['uniform', 'distance'],
        'p': [1, 2]
    },
    'Perceptron': {
        'penalty': ['l1', 'l2', 'elasticnet'],
        'alpha': [1e-4, 1e-3, 1e-2],
        'max_iter': [1000, 2000]
    },
    'Logistic Regression': {
        'C': [0.01, 0.1, 1, 10],
        'penalty': ['l1', 'l2', 'elasticnet'],
        'solver': ['liblinear', 'saga']
    },
    'SVM': {
        'C': [0.1, 1, 10, 100, 1000],
        'kernel': ['poly', 'rbf', 'sigmoid'],
        'gamma': ['scale', 'auto']
    },
    'SVM Linear': {
        'C': [0.1, 1, 10, 100]
    },
    'Random Forests': {
        'n_estimators': [100, 200],
        'max_depth': [None, 10, 20],
        'max_features': ['sqrt', 'log2']
    },
    'AdaBoost': {
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.1, 1.0]
    },
    'Decision Tree': {
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },
    'Neural Networks': {
        'hidden_layer_sizes': [(50,), (100,), (100, 50)],
        'activation': ['relu', 'tanh', 'logistic'],
        'alpha': [1e-5, 1e-4],
        'learning_rate_init': [1e-3, 1e-4]
    }
}

naive_bayes_model = GaussianNB()
nb_scores = cross_val_score(naive_bayes_model, X_train, y_train, cv=5, scoring='accuracy')
results['Naive Bayes'] = {
    'Best Parameters': None,
    'Best Accuracy': np.mean(nb_scores)
}
```

Part D

```
# Perform Grid Search for each model
results = {}

for model_name, param_grid in param_grids.items():
    print(f"Tuning {model_name}...")
    model = {
        'k-NN': KNeighborsClassifier(),
        'Perceptron': Perceptron(),
        'Logistic Regression': LogisticRegression(max_iter=500),
        'SVM': SVC(),
        'SVM Linear': LinearSVC(),
        'Random Forests': RandomForestClassifier(random_state=42),
        'AdaBoost': AdaBoostClassifier(random_state=42),
        'Decision Tree': DecisionTreeClassifier(random_state=42),
        'Neural Networks': MLPClassifier(max_iter=500, random_state=42)
    }.get(model_name)

    grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy', verbose=2)
    grid_search.fit(X_train, y_train)
    results[model_name] = {
        'Best Parameters': grid_search.best_params_,
        'Best Accuracy': grid_search.best_score_
    }
```

Part D

The hyperparameter tuning took a total 1.5 hours, which is worth noting. Now we can print the training results

```
for model_name, result in results.items():
    print(f"{model_name}: Best Parameters = {result['Best Parameters']}")
    print(f"{model_name}: Best Accuracy = {result['Best Accuracy']:.4f}")
```

Part D - Hyperparameter Tuning results

Naive Bayes:

- Best Parameters: None
- Best Accuracy: 0.7026

k-NN:

- Best Parameters:
  - Number of Neighbors: 9,
  - Distance Metric Parameter: 2
  - Weight Function: distance
- Best Accuracy: 0.8220

Perceptron:

- Best Parameters:
  - Regularization Parameter: 0.0001
  - Maximum Iterations: 1000
  - Penalty Type: l1
- Best Accuracy: 0.7057

Part D - Hyperparameter Tuning results

Logistic Regression:

- Best Parameters:
  - Inverse Regularization Strength (C): 0.01
  - Penalty Type: l2
  - Optimization Algorithm: saga
- Best Accuracy: 0.7811

SVM:

- Best Parameters:
  - Regularization Parameter (C): 10
  - Kernel Coefficient (gamma): auto
  - Kernel Type: rbf
- Best Accuracy: 0.8534

SVM Linear:

- Best Parameters:
  - Regularization Parameter (C): 0.1
- Best Accuracy: 0.7634

Part D - Hyperparameter Tuning results

Random Forests:

- Best Parameters:
  - Maximum Depth: 20
  - Number of Features Considered for Splits: sqrt
  - Number of Trees: 200
- Best Accuracy = 0.8140

AdaBoost:

- Best Parameters:
  - Learning Rate: 1.0
  - Number of Estimators: 200
- Best Accuracy = 0.6609

Decision Tree:

- Best Parameters:
  - Maximum Depth: 10
  - Minimum Samples per Leaf: 4
  - Minimum Samples for Splits: 10
- Best Accuracy = 0.6390

Neural Networks:

- Best Parameters:
  - Activation Function: relu
  - Regularization Parameter (alpha): 1e-05
  - Hidden Layer Sizes: (100,)
  - Initial Learning Rate: 0.001
- Best Accuracy = 0.8279

Part D

We will select the model with the best accuracy, using the optimal hyperparameters

```
best_model = max(results, key=lambda key: results[key]['Best Accuracy'])
best_accuracy = results[best_model]['Best Accuracy']
print(f"Best Model: {best_model} with accuracy {best_accuracy:.4f}")
print(f"Hyperparameters of the best model: {results[best_model]['Best Parameters']}")
```

Best Model: SVM with accuracy 0.8534

Hyperparameters of the best model:

- Regularization Parameter (C): 10
- Kernel Coefficient (gamma): auto
- Kernel Type: rbf

Part D - Best model

The model with the highest accuracy is **SVM (Support Vector Machine)** with hyperparameters shown below. We will use this model to make predictions on the test dataset

```
clf_best_model = SVC(kernel='rbf', C=10, gamma='auto', random_state=42)
```

Part D

Now we're going to train the model on the entire dataset, and then make predictions on the test dataset. We will then save the predictions to a numpy file.

```
clf_best_model.fit(X_train, y_train)
y_test = clf_best_model.predict(X_test)

np.save('labels40.npy', y_test)
```

Thank you for your time