Solution 1 (a)

Step 1: Linear Classifiers

A single hyperplane can only separate data that is linearly separable. It cannot capture non-linear relationships or create complex, disjoint decision regions.

$\therefore$ Linear classifiers are not as expressive as decision trees and cannot represent any arbitrary decision boundary.

Solution 1 (b)

Step 1: Support Vector Machines with a Quadratic Kernel

A quadratic boundary is more flexible than a linear one and can capture more complex relationships. However, it is still restricted to a second-degree polynomial. It cannot represent all possible decision boundaries and especially those with highly irregular shapes.

$\therefore$ A SVM with a quadradic kernel is not as expressive as decision trees and cannot represent any arbitrary decision boundary.

Solution 1 (c)

Step 1:Nearest Neighbor Classifiers

As the number of training samples grows ($N \to \infty$), a nearest neighbor classifier can approximate any arbitrarily complex decision boundary and can form intricate and highly non-linear shapes.

$\therefore$ Nearest neighbor classifiers (with a sufficient number of diverse training samples) are similar in expressive power to decision trees.

Solution 1 (d)

Step 1: Gaussian Generative Models

Lets consider the decision boundary for the following Gaussian generative models:

- **Linear Models:** The decision boundary is linear.
- **Quadratic Models:** The decision boundary can be up to a second degree polynomial.
- **Mixture Models:** The decision boundary can be highly non linear.

$\therefore$ Gaussian Generative Models with a linear or quadradic decision boundary not as expressive as decision trees and cannot represent any arbitrary decision boundary, while Guasian Mixture Models are as expressive.

Solution 2

Step 1: Number of Ways to Choose a Feature

Let, S be a set consisting of n data points in a d-dimensional space. If we assume an even distribution of n points across all dimensions d of the set S, then the probability of selecting a single point n_i(feature) such that it belongs to a single axis of d is $\frac{1}{d}$. Hence, we have d potential ways to choose a feature.

Step 2: Effective Number of Splits for a Chosen Feature

If we assume a feature is chosen, and sort the n points. It follows that there exist at most $n-1$ unique midpoints or split points. Hence, there are roughly $n-1$ possible splits given one feature has been selected.

Step 3: Total Number of Possible Splits to Try

From **Step 1** we know there is d features to choose from and from **Step 2** we know the maximum number of splits is $n-1$. It follows that there exists, $d \times (n-1)$ possibilities to try.

$\therefore$ When starting at the top node of a decision tree, we have $d \times (n-1)$ possibilities to try.

Solution 3

Step 1

The Gini impurity index (GII) is defined as the following where p is the probability of one label and $1 - p$ is the probability of the second label:

$$GII = 2p(1 - p)$$

By the commutative property of multiplication it doe not matter what value we choose for p since the result will be the same.

Step 2: Calculate the Gini Impurity Index

Let $p = 0.2$ and subsitute into the Gini impurity formula:

$$GII = 2p(1 - p)$$

$$GII = 2 \cdot (0.2) \cdot (1 - 0.2)$$

$$GII = 0.32$$

$\therefore$ The Gini impurity index for a node in which 20% of the points have one label and 80% have the other label is 0.32.

Solution 4 (a)

Step 1

To incorporate weights λ_i into decision trees, we modify how we calculate the proportion of points belonging to each class at a given node. It follows that, incorprating weighted data would only impact the impurity calculation.

Let S be a set that contains all data points at a particular node. The weighted proportion p_k for class k at this node is calculated as the sum of weights of points in class k divided by the total sum of weights of all points at the node:

$$p_k = \frac{\sum_{i \in S_k} \lambda_i}{\sum_{j \in S} \lambda_j}$$

$\therefore$These weighted proportions $\sum_k p_k = 1$ are then used in the impurity measures.

Solution 4 (b)

Step 1

To incorporate weights λ_i into a Gaussian generative model, we must adjust the calculations of the class prior, weighted mean, and covariance.

Step 2: Weighted Class Priors (π_j)

In a Gaussian generative model, the class prior $\pi_j = P(C_j)$ is the fraction of training samples belonging to class j. With weighted data, let N be the total number of data points. Let I_j be the set of indices of data points belonging to class j. The weighted class prior π_j is the sum of weights of samples in class j divided by the total sum of weights of all samples:

$$\pi_j = \frac{\sum_{i \in I_j} \lambda_i}{\sum_{m=1}^{N} \lambda_m}$$

Step 3: Weighted Mean ($\boldsymbol{\mu}_j$)

The mean $\boldsymbol{\mu}_j$ for class j is the average of the feature vectors $\mathbf{x}^{(i)}$ belonging to class j. For weighted data, this becomes a weighted average:

$$\boldsymbol{\mu}_j = \frac{\sum_{i \in I_j} \lambda_i \mathbf{x}^{(i)}}{\sum_{i \in I_j} \lambda_i}$$

Each data point $\mathbf{x}^{(i)}$ in class j contributes to the mean proportionally to its weight λ_i.

Step 4: Weighted Covariance Matrix ($\boldsymbol{\Sigma}_j$)

The covariance matrix $\boldsymbol{\Sigma}_j$ for class j measures the spread and correlation of features for that class. For weighted data, it is estimated as:

$$\boldsymbol{\Sigma}_j = \frac{\sum_{i \in I_j} \lambda_i (\mathbf{x}^{(i)} - \boldsymbol{\mu}_j)(\mathbf{x}^{(i)} - \boldsymbol{\mu}_j)^T}{\sum_{i \in I_j} \lambda_i}$$

$\therefore$ For Gaussian generative models, sample weights λ_i are incorporated by using weighted sums to estimate the class priors π_j, class means $\boldsymbol{\mu}_j$, and class covariance matrices $\boldsymbol{\Sigma}_j$

Solution 4 (c)

Step 1

The objective function for a soft-margin linear SVM is defined as:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i$$

such that:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \text{for } i = 1, \ldots, N$$

Where C is the regularization parameter that balances margin maximization and misclassification penalty, and ξ_i are slack variables allowing for misclassifications.

Step 2

To incorporate sample weights into the objective function for a soft-margin linear SVM, we must modify the penalty term for the slack variables. Points with higher weights will incur a larger penalty if they are misclassified or fall within the margin. The objective function becomes:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \lambda_i\xi_i$$

such that:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \text{for } i = 1, \ldots, N$$

$\therefore$ For Support Vector Machines, sample weights λ_i are incorporated by modifying the objective function to penalize errors on high-weight samples more heavily.

Solution 5 (a)

Step 1

Boosting algorithms, are designed to minimize an objective function related to the training error. The condition that the weak learner's error is at most $\frac{1}{2} - \epsilon$ ensures that boosting can effectively reduce training error, but it doesn't control for the inherent generalization gap between training and test performance.

$\therefore$ The statement ”Boosting converges to a final classifier with zero test error” is **False**.

Solution 5 (b)

Step 1

The condition that each weak learner h_t has a weighted error $\text{err}_t \leq \frac{1}{2} - \epsilon$ (where $\epsilon > 0$) ensures that each weak learner performs better than random guessing on the current distribution of weights. It follows that the training error would converge to zero over enough itterations.

$\therefore$ The statement "Boosting converges to a final classifier with zero training error" is **True** (under the given conditions and assuming enough iterations).

Solution 5 (c)

Step 1

The class $\mathcal{H}$ represents the set of possible *weak* classifiers and the final classifier produced by boosting, $H(\mathbf{x})$, is a weighted linear combination of these weak classifiers. While each weak learner is an element of $\mathcal{H}$, the subsquent sum of weak learners and boosting can form a complex non-linear decision boundary. Hence, the final classifier would not nessicarily be an element of $\mathcal{H}$.

$\therefore$ The statement "Boosting's final classifier belongs to class $\mathcal{H}$" is **False**.

Solution 6 (a)

Step 1

In a Random Forest, each decision tree is trained independently of the others. Each tree is built using a different bootstrap sample of the training data and considers a random subset of features at each split. Since the construction of one tree does not depend on the outcome or structure of any other tree, their training processes can be executed in parallel.

$\therefore$ The statement: *The trees can be trained in parallel.* is **True**

Solution 6 (b)

Step 1

Individual trees in a Random Forest are typically grown to a large depth. They are trained to fit their respective bootstrap samples of the data as well as possible. Hence, each tree is *optimized* to capture the patterns in the subset of data it sees, and is a reason why random forests have an issue of overfitting the training data.

∴ The statement: *Each individual tree is more highly optimized.* is **True**.

Solution 6 (c)

Step 1

From **Solution 6 (b)**, we know that individual trees in a Random Forest are grown to a large depth and each individual tree is trained trained to fit their respective bootstrap samples of the data as well as possible. Conversely, bossted decision trees on require slightly better accuracy on the training data than random guessing.

$\therefore$ The statement: *Each individual tree has better accuracy.* is **True**.

Solution 7 (a)

Sneak Preview at mini-data.txt

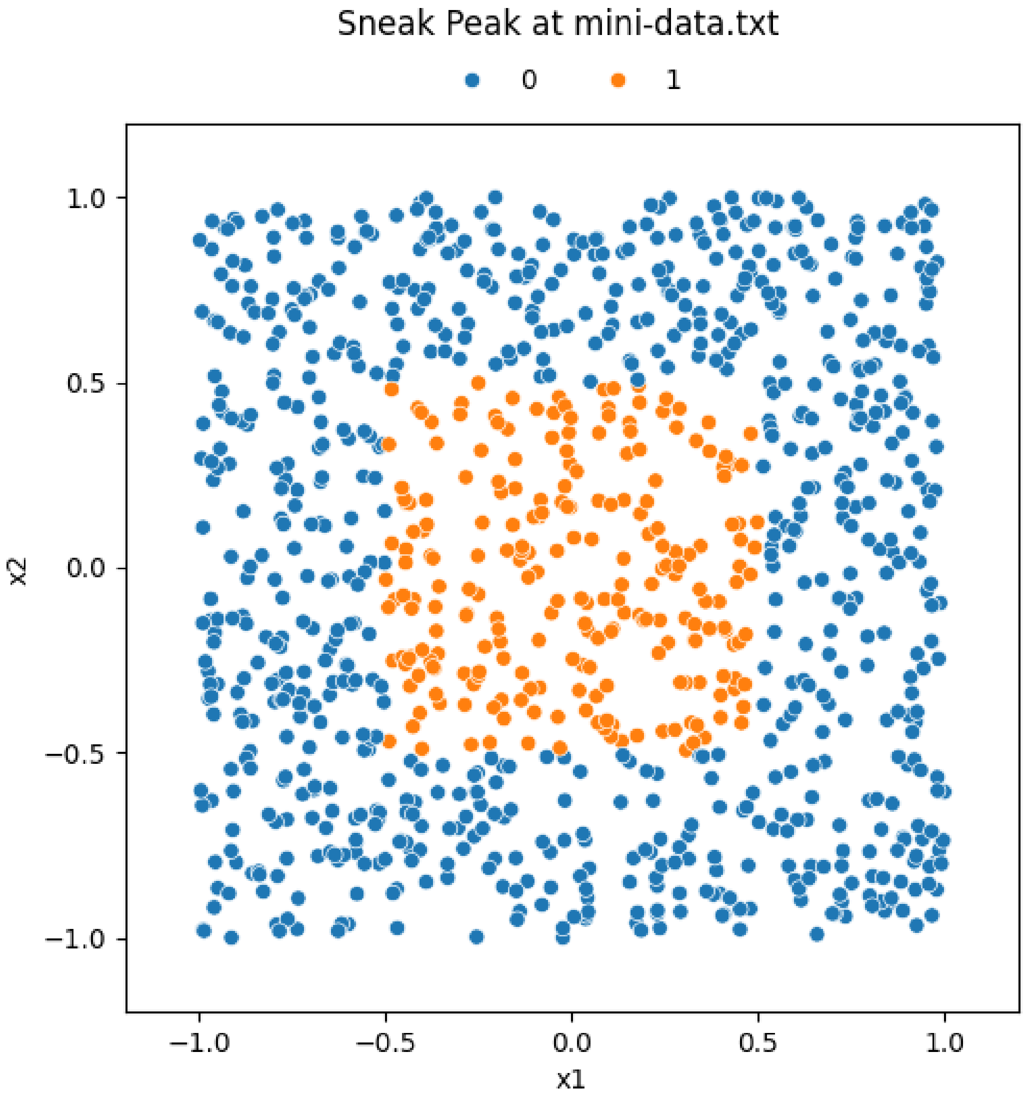

Figure 1: Sneak Peak of mini-data.txt, reveals a square doughnut shape.

Solution 7 (b)

DecisionTreeClassifier: Stop condition

Max depth	Training accuracy	Test accuracy
1	0.50	0.49
2	0.76	0.75
3	0.88	0.90
4	1.00	1.00

Table 1: The stop condition selected was max_depth=4

Python Code

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_data,
                                                    y_data,
                                                    test_size=0.2,
                                                    random_state=42)

def dtc_q7(x_train, x_test, y_train, y_test,n):
    dtc = DecisionTreeClassifier(criterion='log_loss',
                                 max_depth=n,
                                 max_leaf_nodes=n+1,
                                 class_weight='balanced',
                                 min_samples_leaf=int(len(x_train)*0.015),
                                 random_state=42)
    dtc.fit(x_train,y_train)
    print(f"train score(max_depth={n}): {dtc.score(x_train,y_train)}")
    print(f"test score(max_depth={n}): {dtc.score(x_test,y_test)}")

for n in range(1,5):
    dtc_q7(x_train, x_test, y_train, y_test,n)
```

Solution 7 (c)

Plot

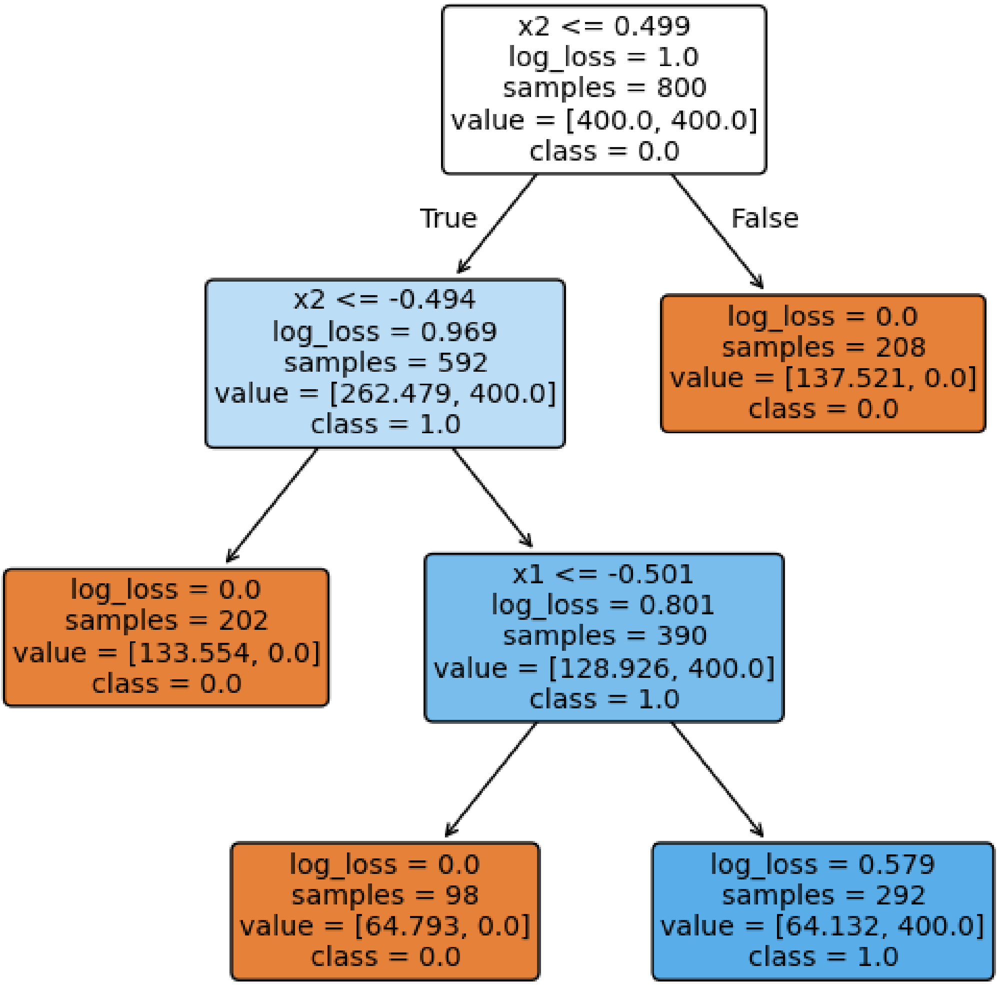

Figure 2: Tree plot from the decision tree classifier.

Solution 7 (d)

Stump Plots

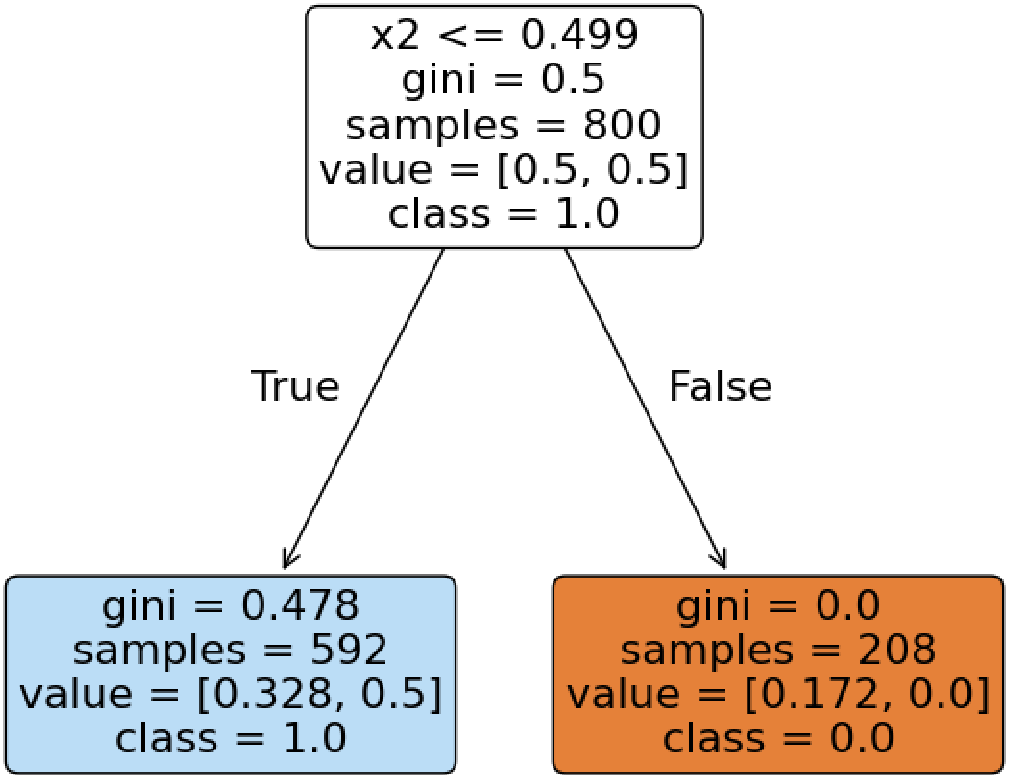

Figure 3: Tree plot from the ada boost with one stump

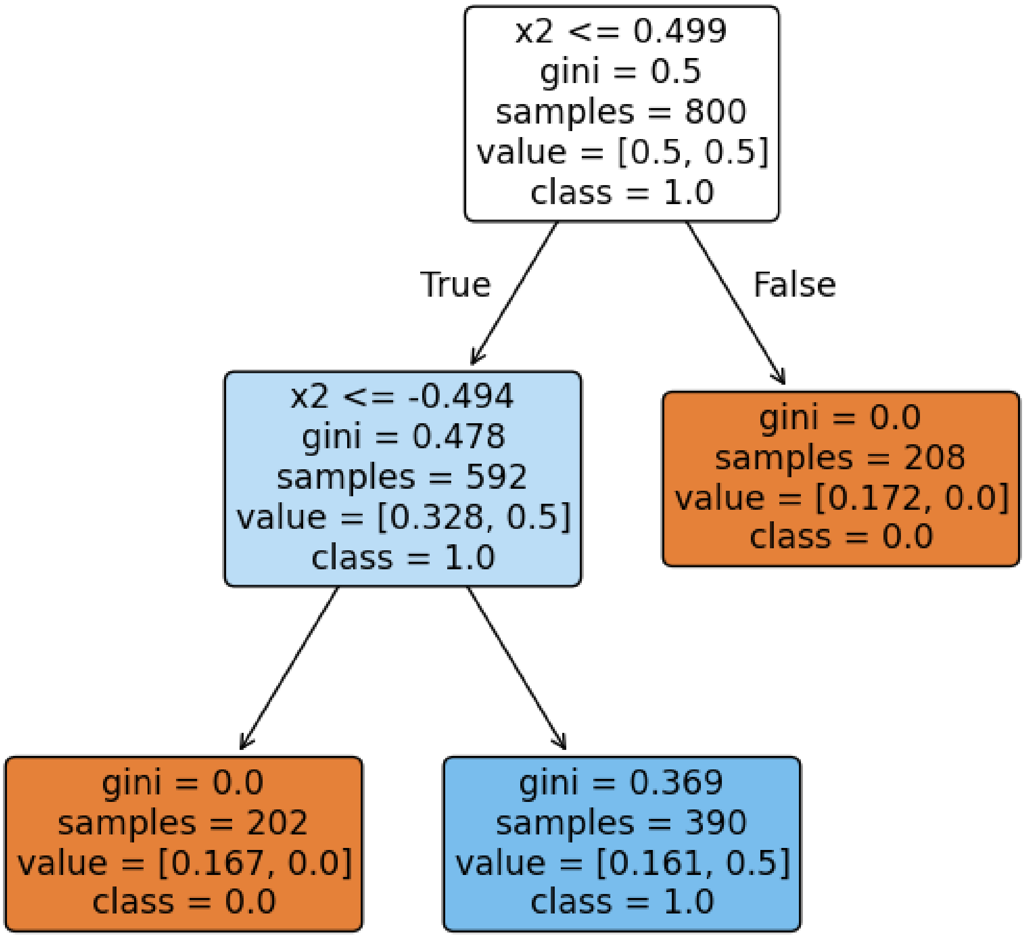

Figure 4: Tree plot from the ada boost with two stumps

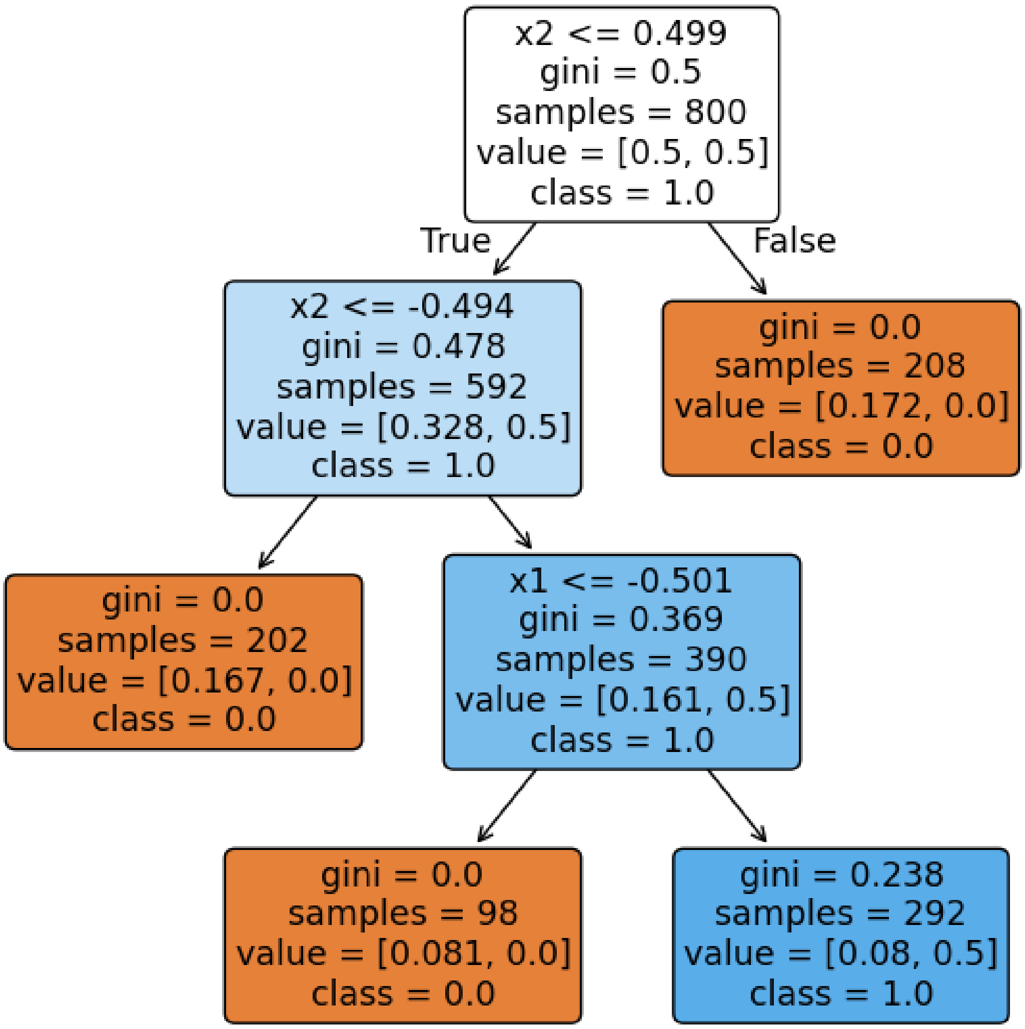

Figure 5: Tree plot from the ada boost with three stumps

Solution 7 (e)

Ada Boost Training Accuracy

Stumps	Training accuracy
1	0.50
2	0.73
3	1.00

Table 2: The training accuracy of the ada boost model increased as the number of stumps increased.

Python Code

```python
from sklearn.ensemble import AdaBoostClassifier

def ada_q7(x_train, x_test, y_train, y_test,n):
    ## initalize the first decision tree stump for the ada booster
    stump = DecisionTreeClassifier(max_depth=n,
                                   class_weight='balanced',
                                   random_state=42)

    ## initalize ada booster model with n stumps
    ada = AdaBoostClassifier(estimator=stump,
                             n_estimators=n,
                             learning_rate=1.0,
                             algorithm='SAMME',
                             random_state=42)

    ada.fit(x_train,y_train)
    print(f"train score(n_estimators={n}): {ada.score(x_train,y_train)}")
    print(f"test score(n_estimators={n}): {ada.score(x_test,y_test)}")

    fig = plt.figure(figsize=(6,6))
    tree.plot_tree(ada.estimators_[0], feature_names=["x1", "x2"],
        class_names=["0.0","1.0"],filled=True, rounded=True)
    plt.tight_layout()
    plt.savefig(f"hw9_q7d_ada_stumps_{n}.png")

stumps = [1,2,3]
for n in stumps:
    ada_q7(x_train, x_test, y_train, y_test,n)
```

Solution 8 (a)

Running the following commands tells us the distribution of fradulent and legitimate in *creditcard.csv*

- legitimate (class:0): 284315
  - `cat creditcard.csv | rev | cut -d ',' -f 1 | rev | grep 0 | wc -l`
- fradulent (class:1): 492
  - `cat creditcard.csv | rev | cut -d ',' -f 1 | rev | grep 1 | wc -l`

Solution 8 (b)

Python Code: Downsample

```python
import pandas as pd
from sklearn.model_selection import train_test_split

## read creditcard.csv
cc = pd.read_csv('creditcard.csv')

## split data by class, balance to 50/50, and combine into one dataframe
fraud = cc[cc['Class']==1.0]
legit = cc[cc['Class']==0.0].sample(n=len(fraud),random_state=42)
cc_bal = pd.concat([fraud,legit]).sample(frac=1, random_state=42)

## split balanced dataframe to x,y , convert to numpy arrays and then training and test data sets
x = cc_bal.drop(columns=['Class'],inplace=False).to_numpy()
y = cc_bal['Class'].astype(int).to_numpy()

## split data for models
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2, random_state=42)
```

Solution 8 (c)

Python Code: Fit Models

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier
from sklearn.model_selection import cross_val_predict, StratifiedKFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def fit_dt(x_train, x_test, y_train, y_test ):
    dt = DecisionTreeClassifier(criterion='gini', max_depth=4, class_weight='balanced',
        min_samples_leaf=20, random_state=42)
    dt.fit(x_train,y_train)
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    y_pred = cross_val_predict(dt, x_test, y_test, cv=cv)
    cm = confusion_matrix(y_test,y_pred,labels=[0,1])
    cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["legit","fraud"])
    cmd.plot(cmap="Blues")
    score = dt.score(x_test,y_test)
    print(f"accuracy of decision tree:(train): {dt.score(x_train,y_train):.3f}")
    print(f"accuracy of decision tree:(test): {score:.3f}")
    cmd.ax_.set_title(f"Decision Tree Classifier\n Test Accuracy: {score:.3f}")
    plt.savefig("dt_classifier.png")

def fit_ada(x_train, x_test, y_train, y_test):
    ada = AdaBoostClassifier(n_estimators=6, algorithm='SAMME',random_state=42)
    ada.fit(x_train,y_train)
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    y_pred = cross_val_predict(ada,x_test,y_test,cv=cv)
    cm = confusion_matrix(y_test,y_pred,labels=[0,1])
    cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["legit","fraud"])
    cmd.plot(cmap="Greens")
    score = ada.score(x_test,y_test)
    print(f"accuracy of ada boost(train): {ada.score(x_train,y_train):.3f}")
    print(f"accuracy of ada boost(test): {score:.3f}")
    cmd.ax_.set_title(f"Ada Boost Classifier\n Test Accuracy: {score:.3f}")
    plt.savefig("boost_classifier.png")


def fit_rf(x_train, x_test, y_train, y_test):
    rf = RandomForestClassifier(n_estimators=100,
        criterion='gini',class_weight='balanced',random_state=42)
    rf.fit(x_train,y_train)
    cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=42)
    y_pred =cross_val_predict(rf,x_test,y_test,cv=cv)
    cm = confusion_matrix(y_test,y_pred,labels=[0,1])
    cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["legit","fraud"])
    cmd.plot(cmap="Reds")
    score = rf.score(x_test,y_test)
    print(f"accuracy of random forest(train): {rf.score(x_train,y_train):.3f}")
    print(f"accuracy of random forest(test): {score:.3f}")
    cmd.ax_.set_title(f"Random Forest Classifier\n Test Accuracy: {score:.3f}")
    plt.savefig("rf_classifier.png")


fit_dt(x_train, x_test, y_train, y_test)
fit_ada(x_train, x_test, y_train, y_test)
fit_rf(x_train, x_test, y_train, y_test)
```

Plots: Slick Confusion Matrices

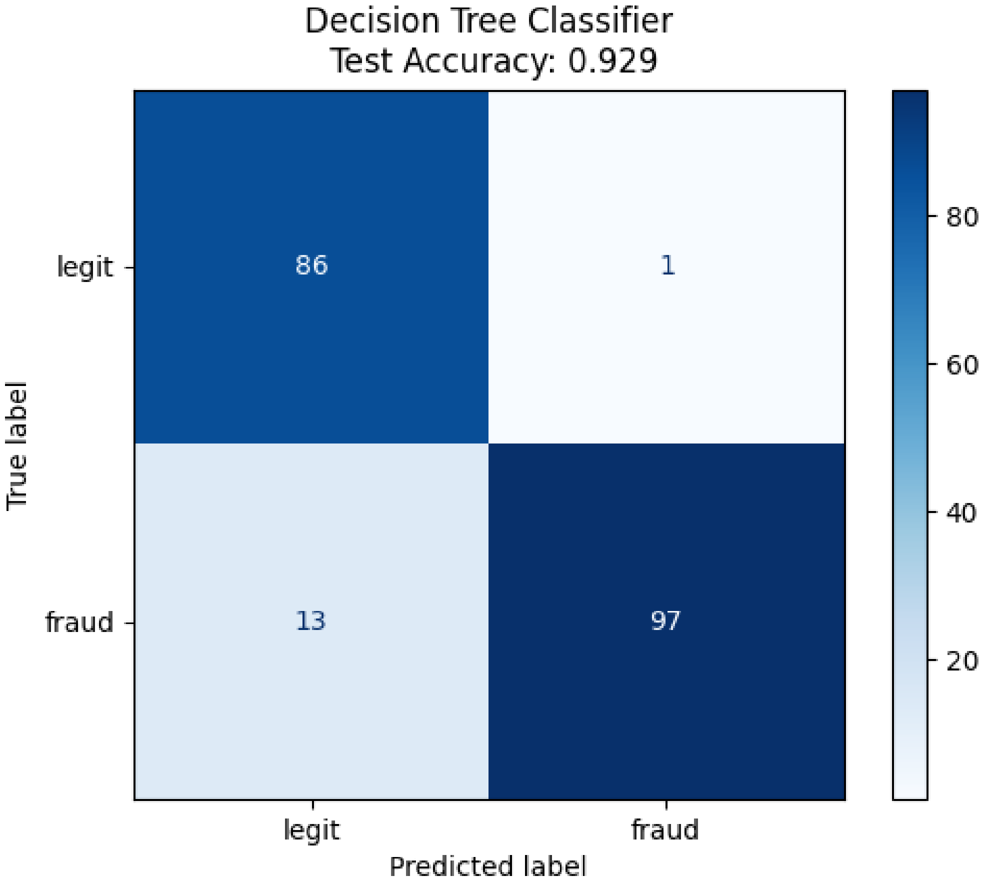

Figure 6: Confusion matrix for decision tree classifier.

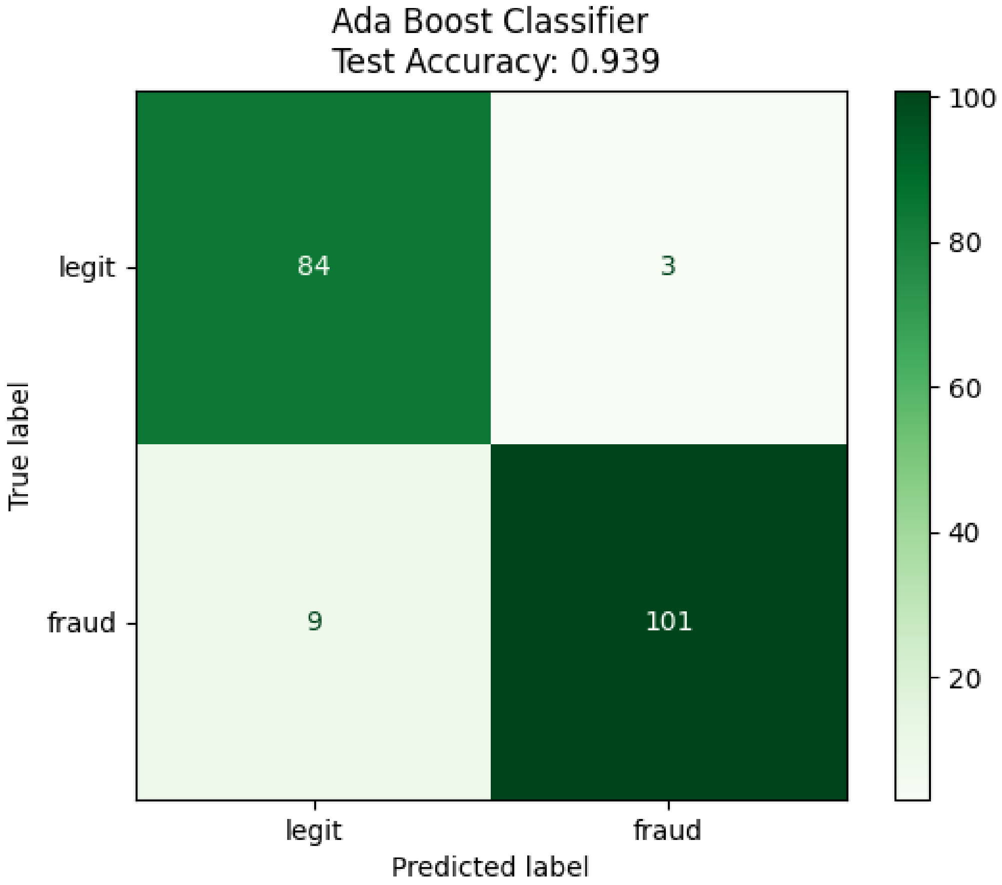

Figure 7: Confusion matrix for ada boost classifier.

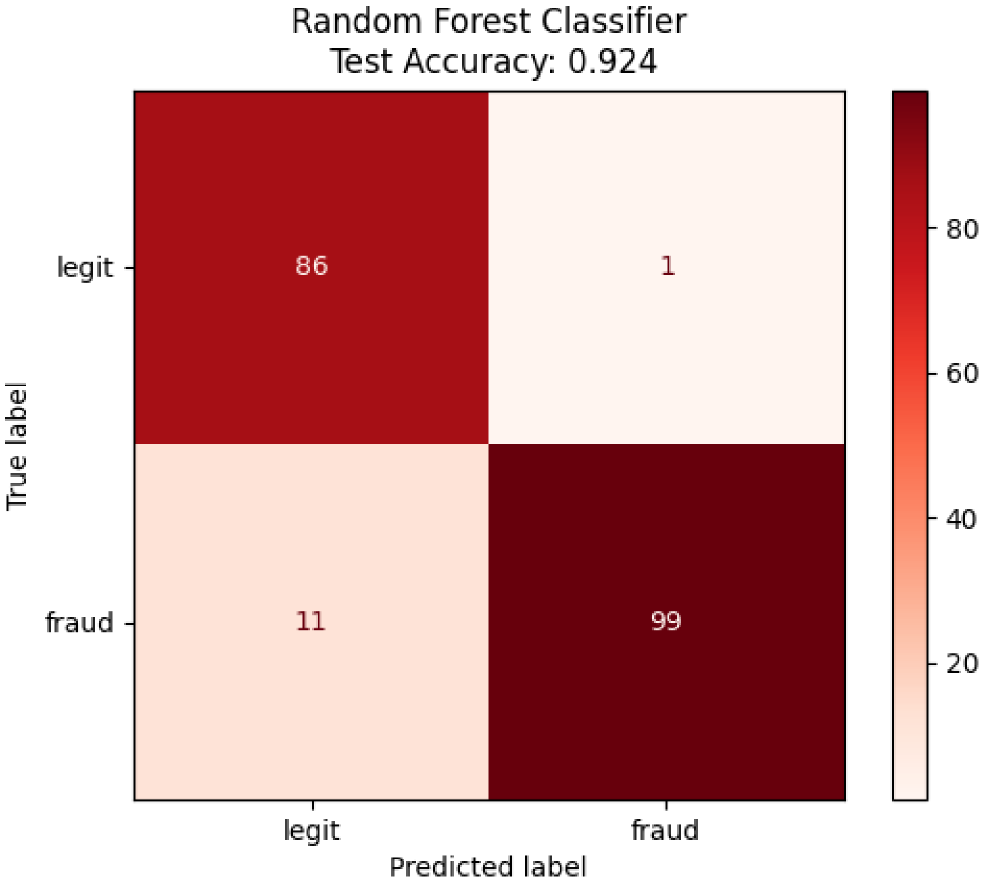

Figure 8: Confusion matrix for random forest classifier.