ΕΛΛΗΝΙΚΟ
ΑΝΟΙΚΤΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ

Panagiotis Paltsokas
std163861@ac.eap.gr
#ΗΛΕ-48

DAMA 61 : Written Assignment 3

PROBLEM 1

This problem concentrates on training models for the MNIST dataset using Decision Trees, Ensemble Models and Dimensionality reduction: Please fix the random state to 42 where required.

1.1

Open a Jupyter-notebook load the the MNIST dataset and split it to 80% training and 20% test parts using stratisfied splitting with a fixed random state. Retain only the 10000 first entries of the resulting training set and the first 2000 from the test set. Convert labels from array of strings to array of 64-bit integers.

In [1]:
```
## Collection of imports for Problem 1
# import numpy as np
# import pandas as pd
# import matplotlib.pyplot as plt
# import time
# from sklearn.datasets import fetch_openml
# from sklearn.model_selection import train_test_split, GridSearchCV
# from sklearn.metrics import accuracy_score, f1_score
# from sklearn.tree import DecisionTreeClassifier
# from sklearn.ensemble import GradientBoostingClassifier
# from sklearn.cluster import KMeans
# from sklearn.pipeline import make_pipeline, Pipeline
# from sklearn.preprocessing import StandardScaler
# from sklearn.decomposition import PCA
```

Load MNIST dataset

In [2]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
```

In [3]:
```
X_full, y_full = mnist["data"], mnist["target"]
```

In [4]:
```
import numpy as np

y_full = y_full.astype(np.int64) # Convert labels to array of 64-bit integers
```

Split dataset into 80% training and 20% test while stratifying on labels. Stratification ensures each label is proportionally represented in train and test sets.

In [5]:
```
from sklearn.model_selection import train_test_split

X_train_full, X_test_full, y_train_full, y_test_full = train_test_split(X_full, y_full, test_size=0.2, stratify=y_full, random_state=42)
```

Retain only 10,000 training and 2,000 test samples

In [6]:
```
X_train, y_train = X_train_full[:10000], y_train_full[:10000]
X_test, y_test = X_test_full[:2000], y_test_full[:2000]

print(f"Training set shape: {X_train.shape}, Labels shape: {y_train.shape}")
print(f"Test set shape: {X_test.shape}, Labels shape: {y_test.shape}")
print(f"Label dtype: {y_train.dtype}")
```

```
Training set shape: (10000, 784), Labels shape: (10000,)
Test set shape: (2000, 784), Labels shape: (2000,)
Label dtype: int64
```

1.2

Perform a Grid Search with 5-fold cross validation, maximum features taking the values [100, 150, 200], and the maximum depth the values [2, 4, 5], for a Decision Tree Classifier, using the Entropy criterion and fixed random state. Print the accuracy score, the F1 - score (with average parameter set to "macro"), with respect to test data, and the values of the best parameters for the best estimator. Print all scores for all combinations and discuss the results.

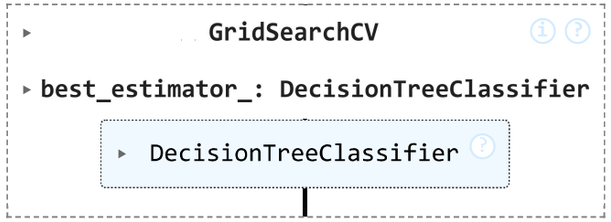
Set the Grid Search parameters

In [7]:
```
gs_parameters = {'max_features' : [100, 150, 200], 'max_depth' : [2, 4, 5]}
```

Perform Grid Search with 5-fold cross-validation

In [8]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, f1_score

clf = DecisionTreeClassifier(criterion='entropy', random_state=42)

grid_search = GridSearchCV(estimator=clf,
                           param_grid=gs_parameters,
                           cv=5,
                           scoring='accuracy',
                           n_jobs=-1) # use n_jobs=-1 to use all CPU cores

grid_search.fit(X_train, y_train)
```

Out[8]:



Retrieve the best estimator from the Grid Search and use it to make predictions on the test data

In [9]:
```
# Best estimator
best_clf = grid_search.best_estimator_
```

```
# Predictions on test data
y_pred = best_clf.predict(X_test)
```

Fit the best estimator from the Grid Search, to time it for question 1.6.

In [10]:
```
import time
start_time=time.time()
best_clf.fit(X_train, y_train)
end_time=time.time()
dt_training_time=end_time - start_time
```

Evaluate accuracy and F1-score on the test set

In [11]:
```
accuracy = accuracy_score(y_test, y_pred)
f1_macro = f1_score(y_test, y_pred, average='macro')

print(f"Decision Tree Best Parameters: {grid_search.best_params_}")
print(f"Decision Tree Accuracy on test set: {accuracy:.4f}")
print(f"Decision Tree F1 Score (macro) on test set: {f1_macro:.4f}")
print(f"Decision Tree Training Time (s): {dt_training_time:.4f}")
```

```
Decision Tree Best Parameters: {'max_depth': 5, 'max_features': 150}
Decision Tree Accuracy on test set: 0.6760
Decision Tree F1 Score (macro) on test set: 0.6639
Decision Tree Training Time (s): 0.1850
```

Print all parameter combinations and their scores

In [12]:
```
import pandas as pd

pd.DataFrame(grid_search.cv_results_).head(5)
```

Out[12]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_max_features	params	split0_test_score	split1_test_score	split2_test_score	split3_test_scor
0	0.235600	0.012224	0.016000	0.004050	2	100	{'max_depth': 2, 'max_features': 100}	0.3365	0.3050	0.3140	0.32
1	0.265999	0.014028	0.013200	0.001938	2	150	{'max_depth': 2, 'max_features': 150}	0.3090	0.3075	0.3245	0.31
2	0.325000	0.029435	0.016799	0.004021	2	200	{'max_depth': 2, 'max_features': 200}	0.3195	0.3220	0.3265	0.32
3	0.325999	0.018953	0.021000	0.005621	4	100	{'max_depth': 4, 'max_features': 100}	0.5720	0.5625	0.5945	0.57
4	0.388799	0.030974	0.012600	0.002417	4	150	{'max_depth': 4, 'max_features': 150}	0.5785	0.5735	0.5940	0.59

In [13]:

```
results = pd.DataFrame(grid_search.cv_results_)
results[['param_max_features', 'param_max_depth', 'mean_test_score']]
```

Out[13]:

	param_max_features	param_max_depth	mean_test_score
0	100	2	0.3162
1	150	2	0.3162
2	200	2	0.3230
3	100	4	0.5626
4	150	4	0.5775
5	200	4	0.5841
6	100	5	0.6506
7	150	5	0.6605
8	200	5	0.6546

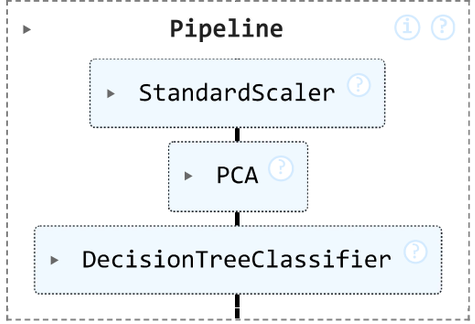
We observe that a shallow tree depth of **max_depth**=2 results in underfitting, providing significantly lower test scores (around 32%). Increasing the depth to 4 leads to a notable improvement in performance, and further increasing it to 5 results in the best scores, confirming that deeper trees help the model learn more meaningful patterns. Regarding **max_features**, the relationship is not strictly linear. While higher values (200) tend to achieve the best results when max_depth=5 (65.46%), the improvement across different values is not consistent across all depths. Generally, lower max_features can help reduce variance, but too low values may increase bias. Given these results, decision trees alone provide reasonable performance but are limited. To further enhance generalization and performance, we should explore ensemble methods, which can mitigate overfitting while leveraging the benefits of multiple trees.

1.3

Create a pipeline performing PCA with fixed random state, retaining 90% of the variance of the initial features included in training dataset, to reduce dimensionality of the dataset and train a Decision Tree with the depth parameter discovered in the previous question. Compute accuracy and F1 - score with respect to test data and compare with the previous results. Please discuss the advantages or disadvantages of using PCA.

In [14]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline

scaler = StandardScaler()
pca = PCA(n_components=0.90, random_state=42)
dt_pca = DecisionTreeClassifier(criterion='entropy', max_depth=5, random_state=42)

pca_pipeline = make_pipeline(scaler, pca, dt_pca)

pca_pipeline.fit(X_train, y_train)
```

Out[14]:



Calculate the time need to fit the Decision Tree after scaling the data and performing PCA.

In [15]:
```
X_train_scaled = scaler.fit_transform(X_train)

X_train_pca_time = pca.fit_transform(X_train_scaled)

start_time = time.time()
best_clf.fit(X_train_pca_time, y_train)
pca_dt_training_time = time.time() - start_time
```

Make predictions on the test set

In [16]:
```
y_pred_pca = pca_pipeline.predict(X_test)
```

Evaluate performance on the test set

In [17]:
```
from sklearn.metrics import accuracy_score, f1_score

accuracy_pca = accuracy_score(y_test, y_pred_pca)
f1_macro_pca = f1_score(y_test, y_pred_pca, average='macro')

print(f"Decision Tree Accuracy after PCA : {accuracy_pca:.4f}")
print(f"Decision Tree F1 Score (macro) after PCA : {f1_macro_pca:.4f}")
print(f"Decision Tree Training Time (s) after PCA: {pca_dt_training_time:.4f}")
```

```
Decision Tree Accuracy after PCA : 0.6720
Decision Tree F1 Score (macro) after PCA : 0.6682
Decision Tree Training Time (s) after PCA: 2.3130
```

Using PCA before training a Decision Tree in Problem 1.3 resulted in a minor accuracy drop (0.676 -> 0.672) but a slight F1-score improvement (0.6639 -> 0.6682), suggesting better class balance. However, training time increased significantly (0.187s -> 2.197s) due to the additional preprocessing step. While PCA is beneficial for reducing dimensionality, removing noise, and improving efficiency in high-dimensional datasets, it is particularly useful for linear models that rely on decorrelated features. Additionally, it can help mitigate overfitting in cases where redundant or highly correlated features are present.

Despite these benefits, PCA can discard valuable information by transforming data into new components, which may explain the slight accuracy decline. Decision Trees naturally select important features and handle high-dimensional data well, making PCA redundant in most cases. Here, the added computational cost did not translate into meaningful performance gains, highlighting that PCA is unnecessary for Decision Trees unless dealing with extreme feature spaces (e.g. 10,000+ features)

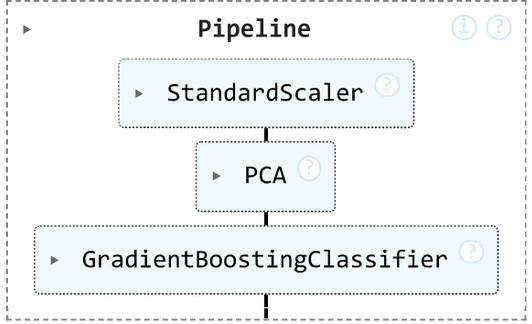
1.4

Perform PCA with the number of components dictated in the previous question, compress the training data and train a Gradient Boosting Classifier (GBC). The model should have a maximum depth of 2, 6 estimators and a learning rate equal to 1.0. The random state should be fixed where required. Discuss the results with respect to previously used Tree Classifier. How is it possible for GBC with shallow trees to outperform a deeper Tree Classifier?

Create a pipeline which standardizes the features to ensure all features contribute equally, applies Principal Component Analysis to reduce dimensionality, retaining 90% of the variance, and trains a Gradient Boosting Classifier, with shallow trees.

In [18]:
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, f1_score


gbc_pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Standardize features before PCA to ensure all features have equal importance
    ('pca', PCA(n_components=0.90, random_state=42)),  # Retain 90% variance
    ('gbc', GradientBoostingClassifier(
        max_depth=2,  # Use shallow trees, which helps reduce overfitting
        n_estimators=6,
        learning_rate=1.0,
        random_state=42
    ))
])


gbc_pipeline.fit(X_train, y_train)
```

Out[18]:



Calculate the time need to fit the Gradient Boosting Classifier after scaling the data and performing PCA.

In [19]:
```
gbc = GradientBoostingClassifier(max_depth=2, n_estimators=6, learning_rate=1.0, random_state=42)
start_time=time.time()
gbc.fit(X_train_pca_time, y_train)
end_time=time.time()
gbc_training_time = end_time - start_time
```

Make predictions on the test set

In [20]:
```
y_pred_gbc = gbc_pipeline.predict(X_test)
```

Evaluate performance on the test set

In [21]:
```
accuracy_gbc = accuracy_score(y_test, y_pred_gbc)
f1_macro_gbc = f1_score(y_test, y_pred_gbc, average='macro')

print(f"Accuracy with GBC: {accuracy_gbc:.4f}")
print(f"F1 Score (macro) with GBC: {f1_macro_gbc:.4f}")
print(f"Training Time (s): {gbc_training_time:.4f}")
```

```
Accuracy with GBC: 0.7940
F1 Score (macro) with GBC: 0.7903
Training Time (s): 28.0613
```

By combining PCA with a Gradient Boosting ensemble of shallow trees, we achieve the best results so far (~79%). Boosting iteratively refines weak learners, explaining the significant performance gain. This highlights the value of ensemble methods.

1.5

Reconstruct the first five images (digits) by using the output of PCA, of the previous question, and plot them along with their corresponding originals in the same figure and discuss. Perform KMeans Clustering, with 20 clusters, using the PCA transformed

training data and plot the most representative digits in a Figure with 2 rows and 10 columns. The most representative digit of each cluster is the one that is closer to its corresponding centroid.

Extract the PCA object from the pipeline

In [22]:
```
gbc_pipeline.named_steps['pca']
```

Out[22]:
```
PCA(n_components=0.9, random_state=42)
```

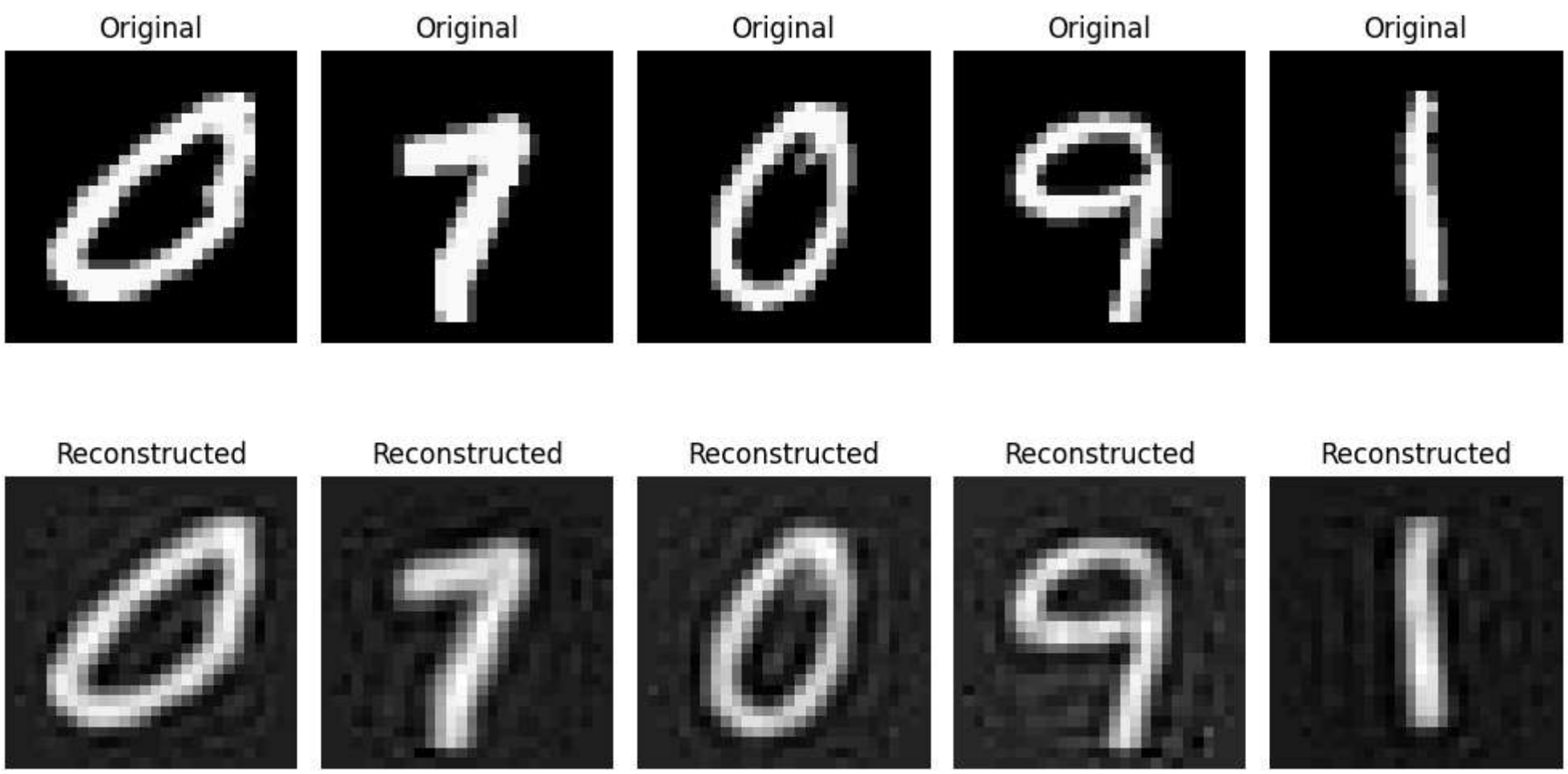
In [23]:
```
pca = gbc_pipeline.named_steps['pca']
```

Transform and inverse transform the first 5 images

In [24]:
```
X_train_pca = pca.transform(X_train[:5])  #Compress: Reduce the dimensionality while retaining 90% of the variance
X_reconstructed = pca.inverse_transform(X_train_pca)  #Reconstruct: Reverse the PCA transformation. We expect to get lower quality images on pixel level.
```

Plot original and reconstructed images

In [25]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 5, figsize=(10, 6)) # Create a 2x5 grid of subplots
for i in range(5):
    # First row will be the Original images
    axes[0, i].imshow(X_train[i].reshape(28, 28), cmap='gray')  # display the image from X_train, reshaping to 28x28
    axes[0, i].set_title("Original")
    axes[0, i].axis('off') # remove axis information for clearer visualization
    # Second row will be the Reconstructed images
    axes[1, i].imshow(X_reconstructed[i].reshape(28, 28), cmap='gray')
    axes[1, i].set_title("Reconstructed")
    axes[1, i].axis('off')

plt.tight_layout()
plt.show()
```



The reconstructed digits are visibly blurrier, but their overall form is intact. PCA at 90% variance discards some fine details while preserving the main shapes. This confirms PCA's ability to compress data while retaining the digit's essential structure

Retrieve scaler and PCA from the pipeline and apply them to the data

In [26]:
```
scaler = gbc_pipeline.named_steps['scaler']
pca = gbc_pipeline.named_steps['pca']

X_train_scaled = scaler.transform(X_train)
X_train_pca = pca.transform(X_train_scaled)
```

Fit KMeans on the transformed data

Train KMeans with 20 clusters

In [27]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=20, random_state=42)
start_time=time.time()
kmeans.fit(X_train_pca)
```

Out[27]:

KMeans

KMeans(n_clusters=20, random_state=42)

In [28]:
```
end_time=time.time()
kmeans_training_time=end_time-start_time
```

Retrieve the clusters' centers

In [29]:
```
centroids = kmeans.cluster_centers_
print(f"Centroids shape:",centroids.shape)
print(f"Centroids:\n",centroids)
```

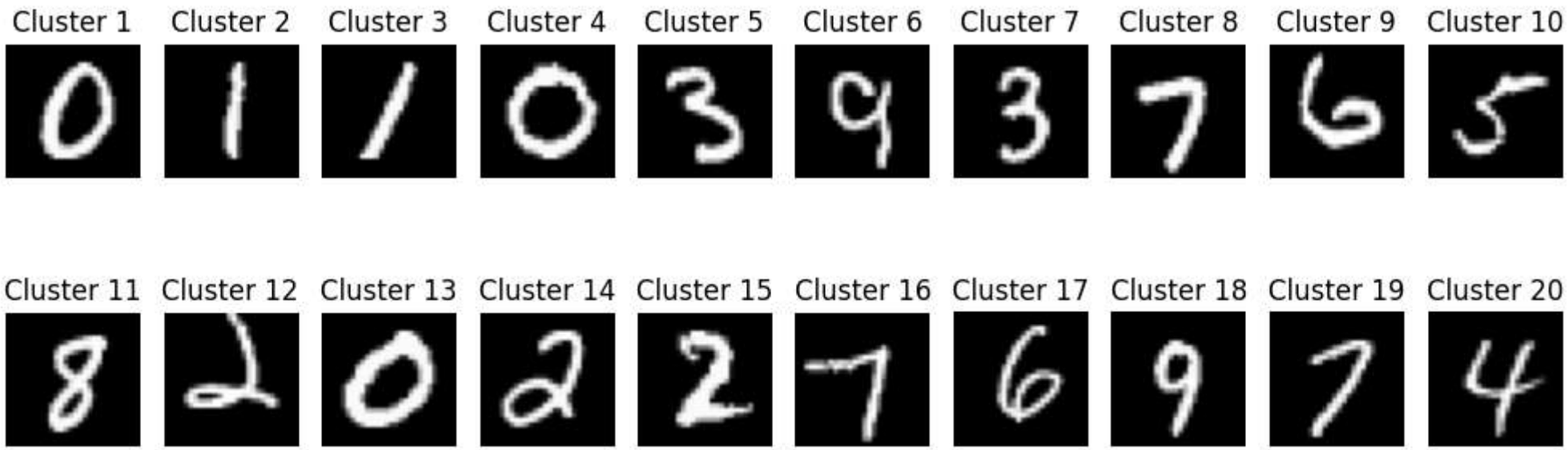
```
Centroids shape: (20, 202)
Centroids:
 [[ 7.76025358e+00 -3.08492192e+00 -1.31698545e+00 ... -2.80128097e-02
   1.85147229e-02 -1.68453622e-02]
 [-6.79232309e+00 -8.60008687e-01  2.55982378e+00 ...  9.65279677e-03
  -1.86900877e-02 -1.71743474e-02]
 [-6.17234799e+00 -3.26791005e+00  4.28607490e+00 ... -3.70340451e-03
  -1.52001662e-02  2.51446045e-02]
 ...
 [-4.70315542e+00  2.50640618e+00 -3.41116835e+00 ...  7.23727166e-02
  -9.72970314e-03 -8.84886856e-03]
 [-3.85704467e+00 -1.55096573e+00 -6.19069117e+00 ... -8.30950751e-02
   6.30512522e-02 -1.36439826e-02]
 [ 1.39201774e-01 -4.31901452e-01 -4.55670331e+00 ...  1.44115973e-02
   8.06821835e-02  6.37723206e-03]]
```

Find the most representative image per cluster

In [30]:
```
closest_samples = [] # Initialize an empty list to store the distances

for center in centroids: # Iterate over each centroid in the list of centroids
    distances = np.linalg.norm(X_train_pca - center, axis=1) # calculate the Euclidean distance of every point in X_train_pca from the center
    closest_samples.append(np.argmin(distances)) # append the index of the point in X_train_pca, which is closest to the center, to closest_samples list
```

Plot the most representative images

In [31]:
```
fig, axes = plt.subplots(2, 10, figsize=(10, 4))
for i, idx in enumerate(closest_samples): # enumerate captures both the index i of the loop, and the index idx of the element from closest_samples
    row, col = divmod(i, 10) # divomod will devide the index i of the loop by 10, returning the quotient to the row, and the remainder to the column
    axes[row, col].imshow(X_train[idx].reshape(28, 28), cmap='gray') # display the image from X_train, reshaping to 28x28
    axes[row, col].axis('off') # remove axis information for clearer visualization
    axes[row, col].set_title(f'Cluster {i+1}')

plt.tight_layout()
plt.show()
```



1.6

An important part of unsupervised learning is labelling. Manually label the printed images of question (5) and store them in a vector. Then, assign the test data to the 20 clusters and assign the corresponding label to each test instance in the clusters (label propagation). By treating these assignments as predictions compute the accuracy and F1 score, with respect to test set labels, and compare to previous approaches with respect also to training times.

Manually label the printed images of 1.5 and store them in a vector

In [32]:
```
cluster_labels = np.array([0, 1, 1, 0, 3, 9, 3, 7, 6, 5, 8, 2, 0, 2, 2, 7, 6, 9, 7, 4])  # Manually labeled based on printed digits
```

Assign labels to test data based on their cluster assignments

Transform the test data using the same scaler and PCA as training

In [33]:
```
X_test_pca = pca.transform(scaler.transform(X_test))
```

Predict clusters for the test data

In [34]:
```
test_clusters = kmeans.predict(X_test_pca)
```

Assign labels to test data, based on their cluster assignment

In [35]:
```
test_labels_pred = [] # Initialize an empty list to store the predicted labels of the test data
for cluster in test_clusters:
    test_labels_pred.append(cluster_labels[cluster]) # For each predicted cluster index in the test set, append the previously assigned label to the list
```

Convert the Python list of predicted labels into a NumPy array

In [36]:
```
test_labels_pred = np.array(test_labels_pred)
```

Evaluate the results

In [37]:
```
accuracy_label_propagation = accuracy_score(y_test, test_labels_pred)
f1_macro_label_propagation = f1_score(y_test, test_labels_pred, average='macro')

print(f"Accuracy with Label Propagation: {accuracy_label_propagation:.4f}")
print(f"F1 Score (macro) with Label Propagation: {f1_macro_label_propagation:.4f}")
print(f"KMeans Training Time (s): {kmeans_training_time:.4f}")
```

```
Accuracy with Label Propagation: 0.6645
F1 Score (macro) with Label Propagation: 0.6516
KMeans Training Time (s): 0.2810
```

In [38]:
```
all_results = {'Algorithm': ['Decision Tree', 'PCA + Decision Tree', 'Gradient Boosting (PCA)', 'Label Propagation (KMeans + PCA)'],
               'Accuracy': [accuracy, accuracy_pca, accuracy_gbc, accuracy_label_propagation],
               'F1 Score (macro)': [f1_macro, f1_macro_pca, f1_macro_gbc, f1_macro_label_propagation],
               'Training Time (s)': [dt_training_time, pca_dt_training_time, gbc_training_time, kmeans_training_time]}

df_all_results = pd.DataFrame(all_results)
df_all_results['Training Time (s)'] = df_all_results['Training Time (s)'].round(1)
df_all_results
```

Out[38]:

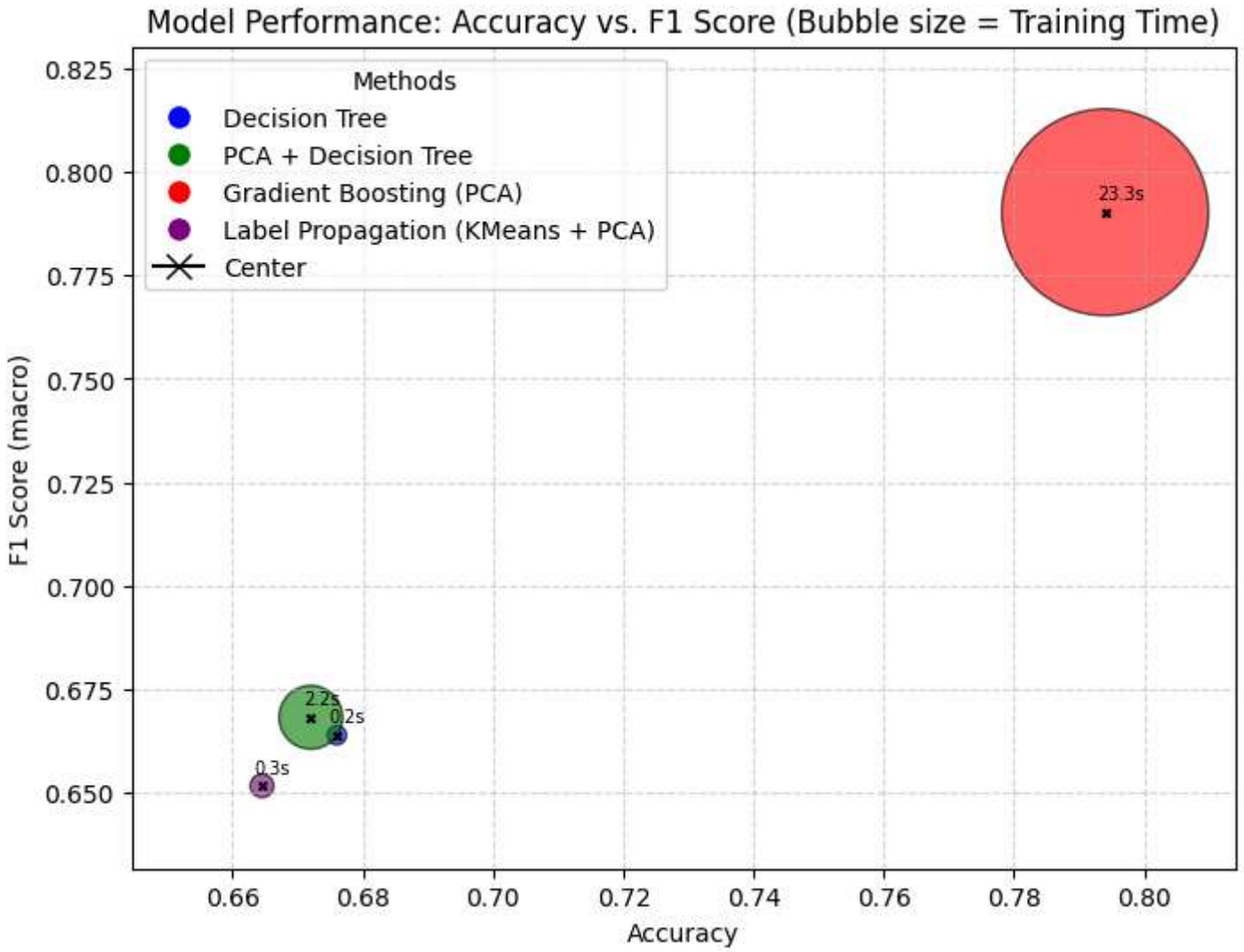
	Algorithm	Accuracy	F1 Score (macro)	Training Time (s)
0	Decision Tree	0.6760	0.663865	0.2
1	PCA + Decision Tree	0.6720	0.668193	2.3
2	Gradient Boosting (PCA)	0.7940	0.790264	28.1
3	Label Propagation (KMeans + PCA)	0.6645	0.651647	0.3

The **Decision Tree**, even though a fast option, appears to be struggling with complex patterns in the data, as indicated by its accuracy and F1 scores.
The use of **PCA** does not appear to improve the Decision Tree's performance, suggesting that the dimensionality reduction might have led to the loss of valuable features or that the tree-based model does not benefit significantly from PCA-transformed data.
Gradient Boosting significantly outperforms the Decision Tree models, achieving the best accuracy and F1 score, demonstrating the power of ensemble methods in improving generalization. However, this comes at a cost of a much longer training time.
Finally, event though **Label Propagation with KMeans and PCA** is computationally efficient, it is the worst performing algorithm, probably because clustering methods do not take into account label-specific information when forming groups, leading to clusters that may not align well with the actual digit classes. Additionally, the dimensionality reduction from PCA might have removed key features needed for better differentiation

Extra code: Visualization

In [39]:
```
# EXTRA CODE #

# Define the results
algorithms = [
    "Decision Tree",
    "PCA + Decision Tree",
    "Gradient Boosting (PCA)",
    "Label Propagation (KMeans + PCA)"
]

accuracy = [0.6760, 0.6720, 0.7940, 0.6645]
f1_score_macro = [0.663865, 0.668193, 0.790264, 0.651647]
training_time = [0.2, 2.2, 23.3, 0.3]

# Define colors for each algorithm
colors = ["blue", "green", "red", "purple"]

# Define sizes for training times
size_factor = 300  # Scaling factor for better visualization
sizes = [t * size_factor for t in training_time]

plt.figure(figsize=(8, 6))
```

```
scatter = plt.scatter(accuracy, f1_score_macro, s=sizes, alpha=0.6, c=colors, edgecolors="black", label=algorithms)

# Add time labels
for i, time in enumerate(training_time):
    plt.annotate(f"{time:.1f}s", (accuracy[i], f1_score_macro[i]), fontsize=7, xytext=(-3, 5), textcoords="offset points")

# Use X markers for the points, where each point’s coordinates are the corresponding accuracy and F1 score pair.
plt.scatter(accuracy, f1_score_macro, color="black", marker="x", s=10, label="Center")

# Expand axis limits
plt.xlim(min(accuracy) - 0.02, max(accuracy) + 0.02)
plt.ylim(min(f1_score_macro) - 0.02, max(f1_score_macro) + 0.04)

# Add Labels and title
plt.xlabel("Accuracy")
plt.ylabel("F1 Score (macro)")
plt.title("Model Performance: Accuracy vs. F1 Score (Bubble size = Training Time)")

# Add legend
legend_patches = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor=colors[i], markersize=10, label=algorithms[i]) for i in range(len(algorithms))]
legend_patches.append(plt.Line2D([0], [0], marker='x', color='black', markersize=10, label="Center"))
plt.legend(handles=legend_patches, title="Methods")

# Show plot
plt.grid(True, linestyle="--", alpha=0.5)
plt.show()
```



The visualization confirms the previous results by clearly illustrating the trade-offs between accuracy, F1 score, and training time for each model. Gradient Boosting (PCA) achieves the highest accuracy and F1 score but has a significantly longer training time. The Decision Tree and PCA + Decision Tree exhibit similar performance, with PCA providing no meaningful improvement while slightly increasing training time. Finally, Label Propagation (KMeans + PCA) is the fastest but delivers the worst performance.

Problem 2

This problem focuses on supervised learning concepts using Ensemble Models, utilizing popular learning frameworks. Additionally, the final question benchmarks the performance of a semi-supervised learning approach using the scikit-learn library. Please fix the random state to 42 where required.

In [40]:
```
## Collection of imports for Problem 2
# import pandas as pd
```

```
# import numpy as np
# from sklearn.model_selection import train_test_split, StratifiedShuffleSplit
# from sklearn.tree import DecisionTreeClassifier
# from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
# from sklearn.svm import SVC
# from sklearn.ensemble import BaggingClassifier
# from sklearn.metrics import classification_report, confusion_matrix
# from sklearn.semi_supervised import SelfTrainingClassifier
```

2.1

Load the provided dataset, pima.csv, print its shape and display the distribution of the target variable.

In [41]:
```
import pandas as pd
# Downloaded the dataset from Kaggle and renamed from diabetes.csv to pima.csv, in order to allign with the request.
data = pd.read_csv('pima.csv')
data_og=pd.read_csv('pima.csv') # Keep a copy of the original dataset
```

Print the first ten lines to take a peek in the data

In [42]:
```
data.head(10)
```

Out[42]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

Print the shape of the datase

In [43]:
```
print("Dataset shape:", data.shape)
```

```
Dataset shape: (768, 9)
```

Extract the target variable

In [44]:
```
target=data['Outcome']
```

Print the target variable distribution

In [45]:
```
print("Target Variable Distribution:")
print(target.value_counts())
```

```
Target Variable Distribution:
Outcome
0    500
1    268
Name: count, dtype: int64
```

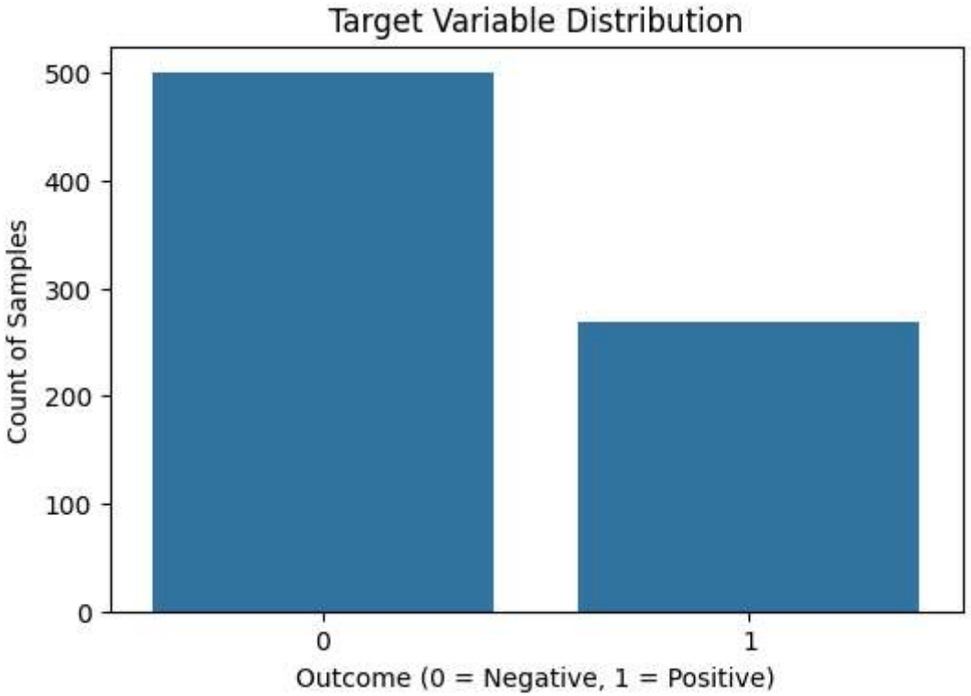
Print the percentage distribution of the target variable

In [46]:
```
print((target.value_counts()/data.shape[0] * 100).round(2).astype(str) + '%')
```

```
Outcome
0    65.1%
1    34.9%
Name: count, dtype: object
```

Plot the distribution of the target variable

In [47]:
```
from matplotlib import pyplot as plt
import seaborn as sns

plt.figure(figsize=(6,4))
sns.countplot(x=target, data=data)
plt.title("Target Variable Distribution")
plt.xlabel("Outcome (0 = Negative, 1 = Positive)")
plt.ylabel("Count of Samples")
plt.show()
```



2.2

Display the main statistical characteristics of each column in the data in a tabular format. Identify columns where the minimum value takes an unreasonable value equal to zero. Replace these zero values with the median of the corresponding column, as they represent missing values based on the dataset's creation protocol. Print the updated dataset's characteristics in a tabular format and list the columns affected by this step.

Display the summary statistics of the dataset.

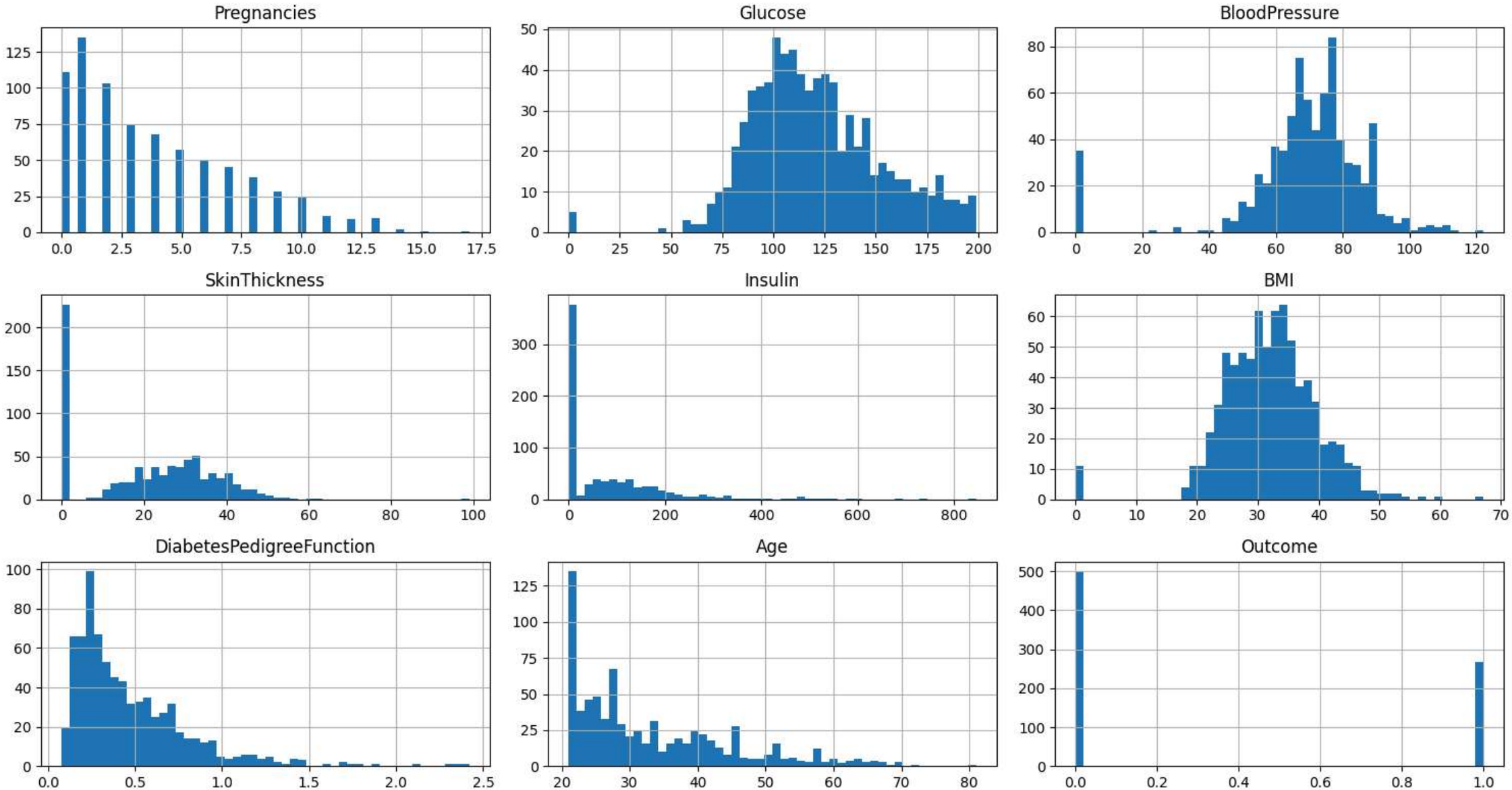
In [48]:
```
data.describe()
```

Out[48]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Plot histograms of all features to visually identify any abnormalities.

In [49]:
```
data.hist(bins=50, figsize=(15,8)) # Create the histograms, with 50 bins
plt.tight_layout() # Use tight_layout so that the histogram title does not collide with the x-axis ticks/values.
plt.show()
```



We wouldn't expect the attributes 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin' and 'BMI', to have zero values. These zeroes represent missing values.

Define the columns that should not contain zero values, and count the number of zeroes in each column.

In [50]:
```
unreasonable_columns = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
print("Number of zero values in unreasonable columns:")
print((data[unreasonable_columns] == 0).sum())
```

```
Number of zero values in unreasonable columns:
Glucose            5
BloodPressure     35
SkinThickness    227
Insulin          374
BMI               11
dtype: int64
```

Replace all zero values in the unreasonable columns with the median of the non-zero values (In order to calculate the median of the actual values we remove the zeroes from the calculation, because not doing so, would alter the values' distribution). Then we replace those zeroes with the median.

In [51]:
```
for col in unreasonable_columns:
    median_value = data[col][data[col] != 0].median()
    data[col] = data[col].replace(0, median_value)
```

Confirm there are no zero values remaining in the unreasonable columns.

In [52]:
```
(data[unreasonable_columns]==0).any()
```

Out[52]:
```
Glucose          False
BloodPressure    False
SkinThickness    False
Insulin          False
BMI              False
dtype: bool
```

Print the updated dataset's statistics

In [53]:
```
data.describe()
```

Out[53]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.656250	72.386719	29.108073	140.671875	32.455208	0.471876	33.240885	0.348958
std	3.369578	30.438286	12.096642	8.791221	86.383060	6.875177	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	25.000000	121.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	29.000000	125.000000	32.300000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Identify and print the affected columns.

In [54]:
```
affected_columns = []
for col in unreasonable_columns:
    if not data_og[col].equals(data[col]):
        affected_columns.append(col)

# Print affected columns
print("Affected columns:")
print(affected_columns)
```

```
Affected columns:
['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
```

2.3

Perform a stratified split of the dataset, allocating 700 instances to the training set and the remaining samples as the test set. Ensure that the target variable is the Outcome column, while the rest of the columns constitute the feature space for the classification task.

In [55]:
```
X = data.drop(columns=['Outcome'])
y = data['Outcome']
```

In [56]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    train_size =700,
```

```
                                                stratify = y,
                                                random_state=42)
```

In [57]:
```
print("Train set size:",len(X_train))
print("Test set size:", len(X_test))
```

```
Train set size: 700
Test set size: 68
```

2.4

Train the following models:

- A simple Decision Tree classifier using the default values of its parameters.

- A Random Forest classifier using the default values of its parameters.

- A Bagging classifier with an SVM classifier (linear kernel) and 10 estimators.

- An AdaBoost classifier with a decision tree classifier, using 100 estimators and a learning rate of 0.25

For each model, display a classification report presenting the standard classification metrics, recision, recall, and f1-score and the confusion matrix as they were calculated on the test set.

Training

Decision Tree classifier

In [58]:
```
from sklearn.tree import DecisionTreeClassifier

dt_clf = DecisionTreeClassifier(random_state=42)
dt_clf.fit(X_train, y_train)
y_pred_dt = dt_clf.predict(X_test)
```

Random Forest classifier

In [59]:
```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(random_state=42)
```

```
rf_clf.fit(X_train, y_train)
y_pred_rf = rf_clf.predict(X_test)
```

Bagging classifier (SVM)

In [60]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

bag_svm_pipeline = make_pipeline(StandardScaler(),
                                 BaggingClassifier(SVC(kernel='linear', random_state=42),
                                                   n_estimators=10,
                                                   random_state=42))

bag_svm_pipeline.fit(X_train, y_train)
y_pred_bag_svm = bag_svm_pipeline.predict(X_test)
```

AdaBoost classifier (Decision Tree)

In [61]:
```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(random_state=42),
    n_estimators=100,
    learning_rate=0.25,
    algorithm="SAMME",
    random_state=42)
ada_clf.fit(X_train, y_train)
y_pred_ada = ada_clf.predict(X_test)
```

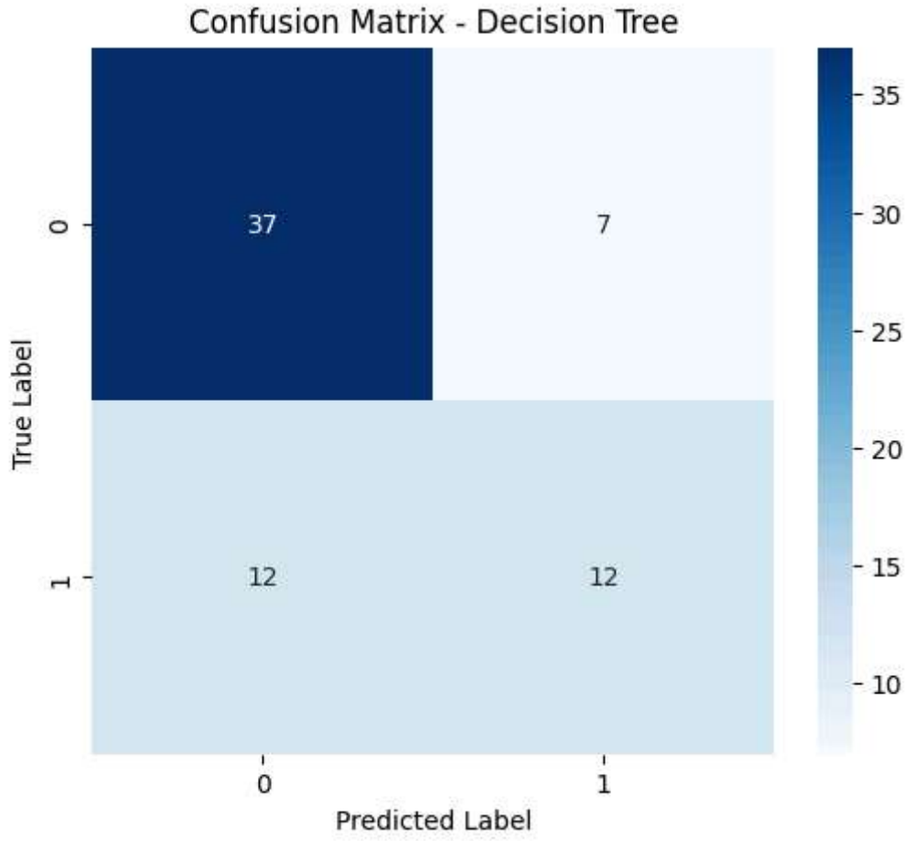
Evaluation

In [62]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix

def model_evaluation_and_conf_mat(model, X_test, y_test, model_name):
    """Evaluates the model on test data, and prints classification metrics along with a Seaborn heatmap confusion matrix."""
    print(f"\n {model_name} Classification metrics")
    y_pred = model.predict(X_test)

    print(classification_report(y_test, y_pred))
```

```
    # Compute confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Plot confusion matrix using Seaborn
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()
```

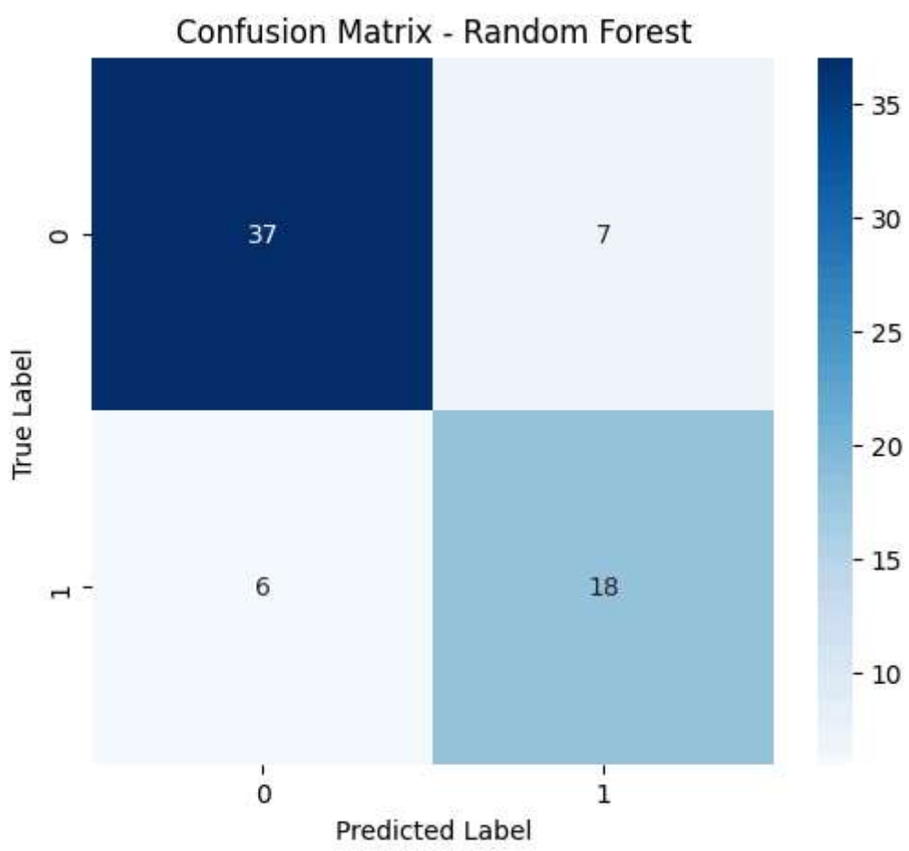
In [63]:
```
print("\nDecision Tree Classification Report:")
model_evaluation_and_conf_mat(dt_clf, X_test, y_test, "Decision Tree")
```

```
Decision Tree Classification Report:

 Decision Tree Classification metrics
              precision    recall  f1-score   support

           0       0.76      0.84      0.80        44
           1       0.63      0.50      0.56        24

    accuracy                           0.72        68
   macro avg       0.69      0.67      0.68        68
weighted avg       0.71      0.72      0.71        68
```



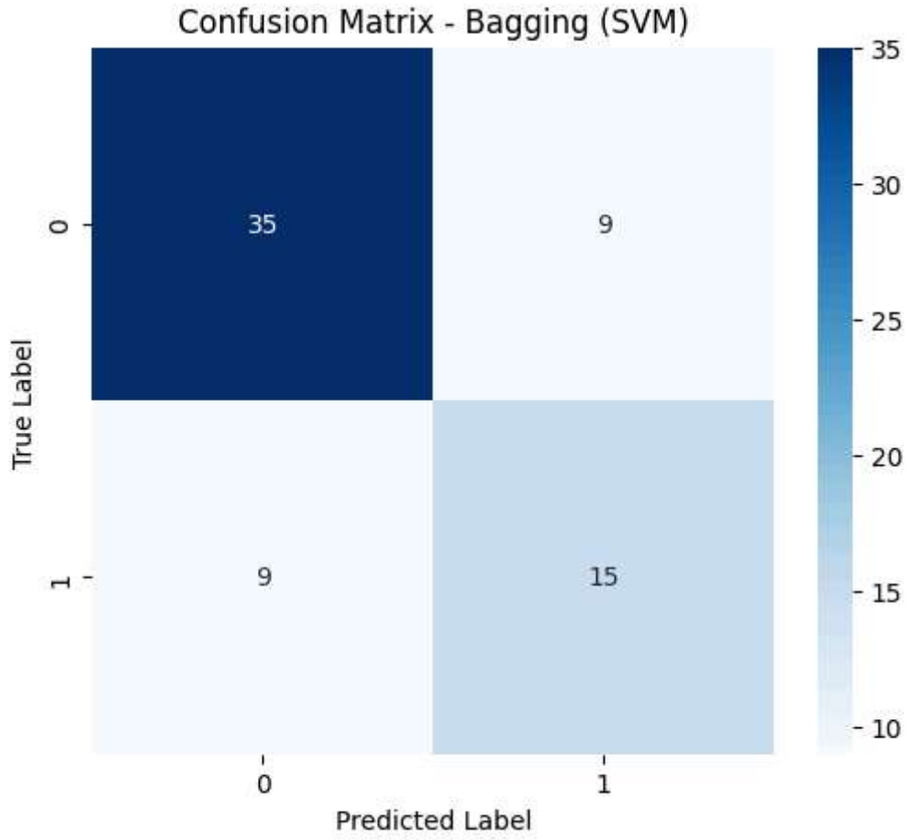
In [64]:
```
print("\nRandom Forest Classification Report:")
model_evaluation_and_conf_mat(rf_clf, X_test, y_test, "Random Forest")
```

```
Random Forest Classification Report:

 Random Forest Classification metrics
              precision    recall  f1-score   support

           0       0.86      0.84      0.85        44
           1       0.72      0.75      0.73        24

    accuracy                           0.81        68
   macro avg       0.79      0.80      0.79        68
weighted avg       0.81      0.81      0.81        68
```



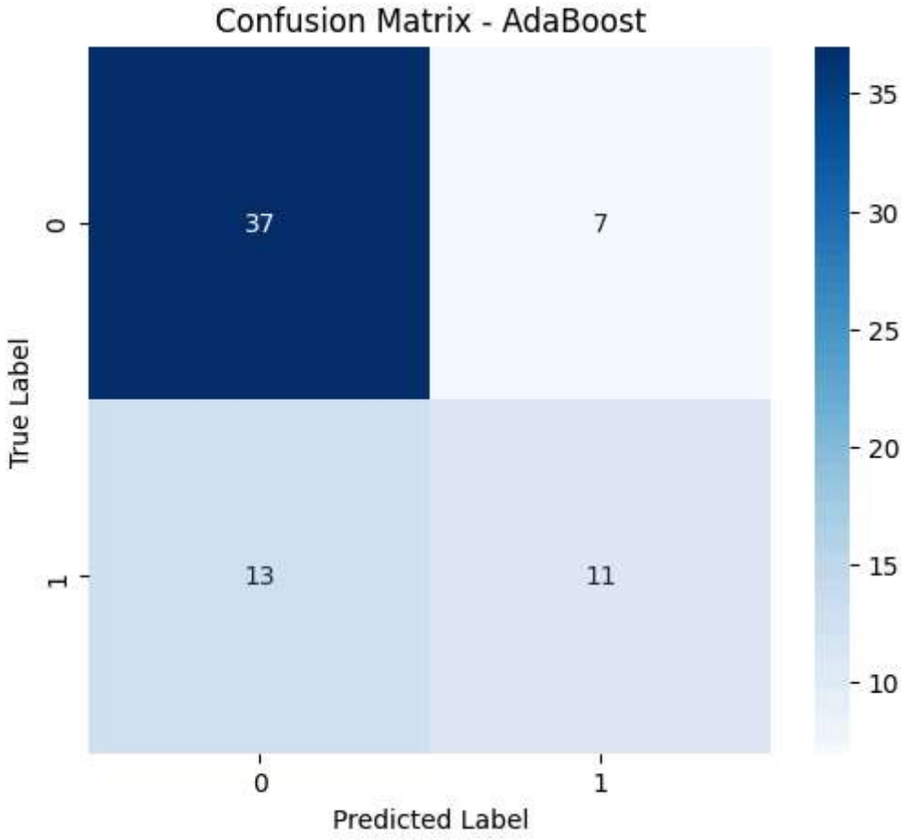
In [65]:
```
print("\nBagging with SVM Classification Report:")
model_evaluation_and_conf_mat(bag_svm_pipeline, X_test, y_test, "Bagging (SVM)")
```

```
Bagging with SVM Classification Report:

 Bagging (SVM) Classification metrics
              precision    recall  f1-score   support

           0       0.80      0.80      0.80        44
           1       0.62      0.62      0.62        24

    accuracy                           0.74        68
   macro avg       0.71      0.71      0.71        68
weighted avg       0.74      0.74      0.74        68
```



In [66]:
```
print("\nAdaBoost Classification Report:")
model_evaluation_and_conf_mat(ada_clf, X_test, y_test, "AdaBoost")
```

```
AdaBoost Classification Report:

 AdaBoost Classification metrics
              precision    recall  f1-score   support

           0       0.74      0.84      0.79        44
           1       0.61      0.46      0.52        24

    accuracy                           0.71        68
   macro avg       0.68      0.65      0.66        68
weighted avg       0.69      0.71      0.69        68
```



2.5

Compare the performance of the previous four models. Comment on the observed performance differences.

In [67]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

def model_metrics(model, X_test, y_test):
    y_pred = model.predict(X_test)
    return {"Accuracy": accuracy_score(y_test, y_pred),
    "Precision": precision_score(y_test, y_pred),
    "Recall": recall_score(y_test, y_pred),
    "F1-score": f1_score(y_test, y_pred)}
```

In [68]:
```
metrics_df = pd.DataFrame([
    model_metrics(dt_clf, X_test, y_test),
    model_metrics(rf_clf, X_test, y_test),
```

```
    model_metrics(bag_svm_pipeline, X_test, y_test),
    model_metrics(ada_clf, X_test, y_test)
], index=["Decision Tree", "Random Forest", "Bagging (SVM)", "AdaBoost"]).round(2).astype(str)

metrics_df
```

Out[68]:

	Accuracy	Precision	Recall	F1-score
Decision Tree	0.72	0.63	0.5	0.56
Random Forest	0.81	0.72	0.75	0.73
Bagging (SVM)	0.74	0.62	0.62	0.62
AdaBoost	0.71	0.61	0.46	0.52

Looking at the results, **Random Forest** clearly stands out as the best model, achieving the highest accuracy (0.81) and the best balance between precision, recall, and F1-score. This shows how powerful ensemble methods can be in improving predictions and handling complex patterns in the data.
Bagging with SVM does a decent job too, with an accuracy of 0.74 and an F1-score of 0.62. It offers some improvements over a standalone model but doesn't quite reach the level of Random Forest.
On the other hand, the **Decision Tree** struggles a bit more. While its accuracy (0.72) isn't too far behind, its recall is quite low (0.50), meaning it misses a lot of positive cases. This suggests it's not as reliable when it comes to generalizing well.
AdaBoost, despite being another ensemble method, surprisingly performs the worst. Its recall (0.46) is the lowest, and its F1-score (0.52) reflects that it's having a hard time maintaining a good balance between precision and recall.
Overall, Random Forest is the clear winner, showing why it's such a popular choice.

2.6

Select the best-performing classifier from Question 4 as the base model. Wrap it in an instance of the SelfTrainingClassifier class with the arguments criterion='threshold' and threshold=0.99. Randomly select 200 instances from the initial training set as labeled data, marking the rest as unlabeled. Train the semi-supervised model on the combined labeled and unlabeled data and train the supervised model on only the labeled data. Evaluate both models on the same test set, display the classification reports, and add your comments.

In [69]:
```
from sklearn.ensemble import RandomForestClassifier


base_model = RandomForestClassifier(random_state=42)
```

In [70]:
```
from sklearn.semi_supervised import SelfTrainingClassifier
self_training_clf = SelfTrainingClassifier(base_model, criterion="threshold", threshold=0.99)
```

In [71]:
```
train_idx = X_train.index.to_numpy()

# Randomly select 200 indices from X_train's index
np.random.seed(42)
labeled_idx = np.random.choice(train_idx, 200, replace=False)
labeled_idx[:10] # Check the first 10 indeces
```

Out[71]:
```
array([748, 686, 408, 282, 423,  62, 331, 506, 284, 306], dtype=int64)
```

In [72]:
```
X_labeled = X_train.loc[labeled_idx]
y_labeled = y_train.loc[labeled_idx]
```

In [73]:
```
X_unlabeled = X_train.drop(index=labeled_idx) # Create a set keeping only the unlabeled data from X_train,
                                              # by droping the indeces of the labeled data
y_unlabeled = np.full(len(X_unlabeled),-1) # Set the unlabeled dataset's target to -1 (no class)
```

In [74]:
```
# Check correct distribution of labeled/unlabeled data
(len(X_unlabeled) == 500) and (len(y_unlabeled) == 500)
```

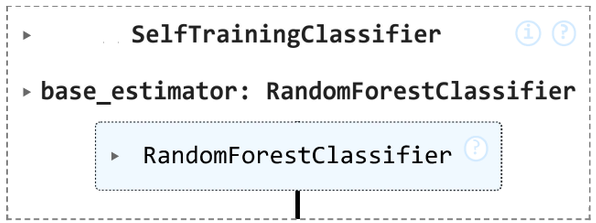
Out[74]:
```
True
```

In [75]:
```
X_combined = pd.concat([X_labeled, X_unlabeled]) # pd.DataFrame
y_combined = np.concatenate([y_labeled, y_unlabeled]) # np.array
```

Train the SelfTrainingClassifier on the combined labeled and unlabeled data

In [76]:
```
self_training_clf.fit(X_combined, y_combined)
```

Out[76]:



Train a fully supervised model only on the labeled data

In [77]:
```
supervised_clf = RandomForestClassifier(random_state=42)
supervised_clf.fit(X_labeled, y_labeled)
```

Out[77]:

RandomForestClassifier

RandomForestClassifier(random_state=42)

Evaluate both models on the same test set

In [78]:
```
y_pred_self_training = self_training_clf.predict(X_test)
y_pred_supervised = supervised_clf.predict(X_test)
```

In [79]:
```
print("\nSelf-Training Classifier Performance:")
print(classification_report(y_test, y_pred_self_training))

print("\nFully Supervised Classifier Performance:")
print(classification_report(y_test, y_pred_supervised))
```

```
Self-Training Classifier Performance:
              precision    recall  f1-score   support

           0       0.81      0.77      0.79        44
           1       0.62      0.67      0.64        24

    accuracy                           0.74        68
   macro avg       0.71      0.72      0.72        68
weighted avg       0.74      0.74      0.74        68


Fully Supervised Classifier Performance:
              precision    recall  f1-score   support

           0       0.81      0.80      0.80        44
           1       0.64      0.67      0.65        24

    accuracy                           0.75        68
   macro avg       0.73      0.73      0.73        68
weighted avg       0.75      0.75      0.75        68
```

In [80]:
```
# Convert classification reports into DataFrames
report_self_training = classification_report(y_test, y_pred_self_training, output_dict=True)
report_supervised = classification_report(y_test, y_pred_supervised, output_dict=True)

# Convert to Pandas DataFrame for better visualization
df_self_training = pd.DataFrame(report_self_training).transpose()
df_supervised = pd.DataFrame(report_supervised).transpose()
```

In [81]:
```
df_self_training
```

Out[81]:

	precision	recall	f1-score	support
0	0.809524	0.772727	0.790698	44.000000
1	0.615385	0.666667	0.640000	24.000000
accuracy	0.735294	0.735294	0.735294	0.735294
macro avg	0.712454	0.719697	0.715349	68.000000
weighted avg	0.741004	0.735294	0.737510	68.000000

In [82]:
```
df_supervised
```

Out[82]:

	precision	recall	f1-score	support
0	0.813953	0.795455	0.804598	44.00
1	0.640000	0.666667	0.653061	24.00
accuracy	0.750000	0.750000	0.750000	0.75
macro avg	0.726977	0.731061	0.728829	68.00
weighted avg	0.752558	0.750000	0.751114	68.00

The fully supervised model has slightly better performance across all metrics. It is better at identifying non-diabetic patients (Class 0), making fewer false negatives, and also makes slightly fewer false positives for diabetic patients (Class 1). However, self-training achieves nearly the same performance as full supervision, despite using only 200 labeled samples. This highlights its ability to perform competitively with minimal labeled data, making it a promising approach when labeled data is limited.