PC 2

Import packages

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, GridSearchCV,
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassi
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

Initial exploration

1.1. Load the dataset

- Load the **adult** dataset **(version 2)** from openml.org using the `sklearn.datasets.fetch_openml` function. Have a look at the documentation for `fetch_openml` if you are not familiar with how to use it, check what is the returned type and make sure you understand how to access the input and target feature of the dataset.

In [2]:
```
data = fetch_openml(data_id=1590, as_frame=True)  # https://openml.
X = data.data
y = data.target
```

```
/opt/jupyterhub/pyvenv/lib/python3.10/site-packages/sklearn/dataset
s/_openml.py:1022: FutureWarning: The default value of `parser` will
change from `'liac-arff'` to `'auto'` in 1.4. You can set `parser='a
uto'` to silence this warning. Therefore, an `ImportError` will be r
aised from 1.4 if the dataset is dense and pandas is not installed.
Note that the pandas parser may return different data types. See the
Notes Section in fetch_openml's API doc for details.
  warn(
```

1.2. Understand the dataset

- Print the type of each feature to see if they are categorical or numeric (hint you can use `pandas.Dataframe.info` to get information about columns in the `DataFrame`).

- Do you notice any feature with missing values? Print out the size of the dataset (number of instances) and the distribution of the target across the two classes using `pandas.Series.value_counts`

In [3]:
```
print("Dataset info:")
X.info()
print("\nMissing values per column:\n", X.isnull().sum())
print("\nDataset size:", X.shape)
print("\nTarget distribution:\n", y.value_counts())
```

```
Dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 14 columns):
 #   Column          Non-Null Count  Dtype   
---  ------          --------------  -----   
 0   age             48842 non-null  float64 
 1   workclass       46043 non-null  category
 2   fnlwgt          48842 non-null  float64 
 3   education       48842 non-null  category
 4   education-num   48842 non-null  float64 
 5   marital-status  48842 non-null  category
 6   occupation      46033 non-null  category
 7   relationship    48842 non-null  category
 8   race            48842 non-null  category
 9   sex             48842 non-null  category
 10  capital-gain    48842 non-null  float64 
 11  capital-loss    48842 non-null  float64 
 12  hours-per-week  48842 non-null  float64 
 13  native-country  47985 non-null  category
dtypes: category(8), float64(6)
memory usage: 2.6 MB

Missing values per column:
 age                  0
workclass         2799
fnlwgt               0
education            0
education-num        0
marital-status       0
occupation        2809
relationship         0
race                 0
sex                  0
capital-gain         0
capital-loss         0
hours-per-week       0
native-country     857
dtype: int64

Dataset size: (48842, 14)

Target distribution:
 class
<=50K    37155
>50K     11687
Name: count, dtype: int64
```

2. Split the data

- Split the data into a train datasets (`X_train, y_train`) and test dataset (`X_test, y_test`), using the `sklearn.model_selection.train_test_split` function.
- Verify the size of each dataset looking at the `shape` attribute.

```
In [4]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
print(X_train.shape )
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(36631, 14)
(12211, 14)
(36631,)
(12211,)
```

3. Create X_train_num by dropping non-numeric columns

- Create an `X_train_num` training dataset by dropping the non-numerical features from the input data, (Hint you can select the relevant columns using `[]` and a list of column names, or use `pandas.Dataframe.drop` to drop the categorical columns or use `pandas.DataFrame.select_dtypes`)

```
In [5]: X_train = X_train.apply(pd.to_numeric, errors='ignore')
X_test = X_test.apply(pd.to_numeric, errors='ignore')
X_train_num = X_train.select_dtypes(include=['int64', 'float64'])
X_test_num = X_test.select_dtypes(include=['int64', 'float64'])
print(X_test_num.shape, X_train_num.shape)
```

```
(12211, 6) (36631, 6)
```

Decision Trees

4.1. Train a DecisionTreeClassifier on numerical data

- Train a `DecisionTreeClassifier` with default parameters to predict the target class from the numerical attributes of input using its fit method.

```
In [6]: clf_num = DecisionTreeClassifier()
clf_num.fit(X_train_num, y_train)
```

```
Out[6]: ▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

4.2. Evaluate the model

- Compute the accuracy of the classifier over the training data and the test data. Hint: you can use the `predict` method of the classifier to obtain the predicted labels from the train and test inputs and use `sklearn.metrics.accuracy_score` to compute the accuracy.

- Does the decision tree seem to overfit? Why? Print out the depth and the number of leaves for the tree.

In [7]:
```
train_accuracy_num = accuracy_score(y_train, clf_num.predict(X_trai
test_accuracy_num = accuracy_score(y_test, clf_num.predict(X_test_nu
print("\nDecision Tree on numerical features:")
print("Training Accuracy:", train_accuracy_num)
print("Testing Accuracy:", test_accuracy_num)
print("Tree Depth:", clf_num.get_depth())
print("Number of Leaves:", clf_num.get_n_leaves())
```

```
Decision Tree on numerical features:
Training Accuracy: 0.9987169337446425
Testing Accuracy: 0.776431086725084
Tree Depth: 64
Number of Leaves: 7393
```

Q: Does the decision tree seem to overfit? Why? Print out the depth and the number of leaves for the tree.

A: Yes, the Decision Tree seems to overfit. Here’s why:

High Training Accuracy (99.87%):

The tree is almost perfectly fitting the training data. This indicates that the tree is memorizing the training set instead of learning generalized patterns. Low Testing Accuracy (77.14%):

The testing accuracy is significantly lower than the training accuracy, suggesting poor generalization to unseen data. Large Tree Depth (52):

A depth of 52 means the tree has grown very deep, which increases the model's complexity. This leads to overfitting as the tree captures noise and fine details from the training data. High Number of Leaves (7857):

The tree has split the data into 7,857 unique leaves. Each leaf may represent very specific patterns in the training data, further indicating overfitting. Why Does Overfitting Happen? Complexity of the Tree: By default, the Decision Tree grows until all data points are perfectly classified (no pruning or constraints). This leads to overfitting when the tree learns noise or specific quirks of the training set.

Imbalance Between Depth and Data Size: With a large depth and too many leaves, the tree becomes excessively complex relative to the size of the dataset.

How to Address Overfitting? Restrict Tree Complexity:

Use parameters like max_depth, min_samples_split, and min_samples_leaf to control the size of the tree. Example: python Copy Edit clf = DecisionTreeClassifier(max_depth=10, random_state=42) Use Cross-Validation:

Employ cross-validation to find optimal hyperparameters and ensure better generalization. Prune the Tree:

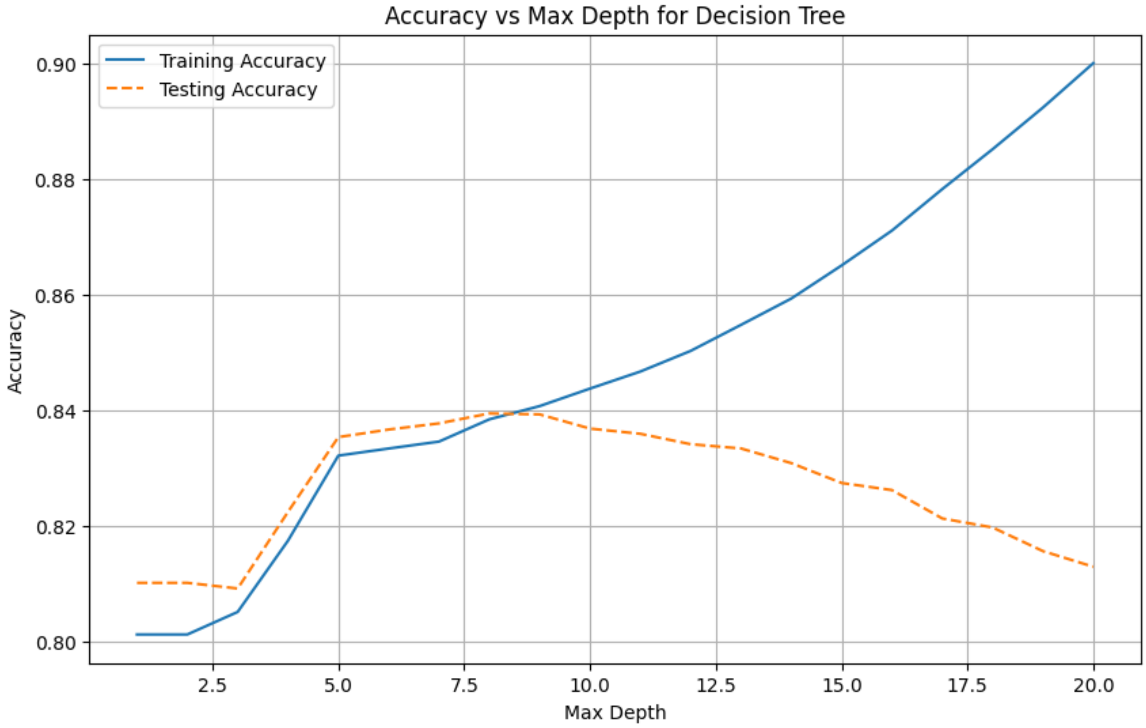
Post-pruning can remove unnecessary branches to simplify the model. Switch to Ensemble Models:

Use models like RandomForestClassifier or GradientBoostingClassifier to improve performance and reduce overfitting.

5. limit max_depth -> like Task 6

6. Plot training and testing accuracy vs max_depth

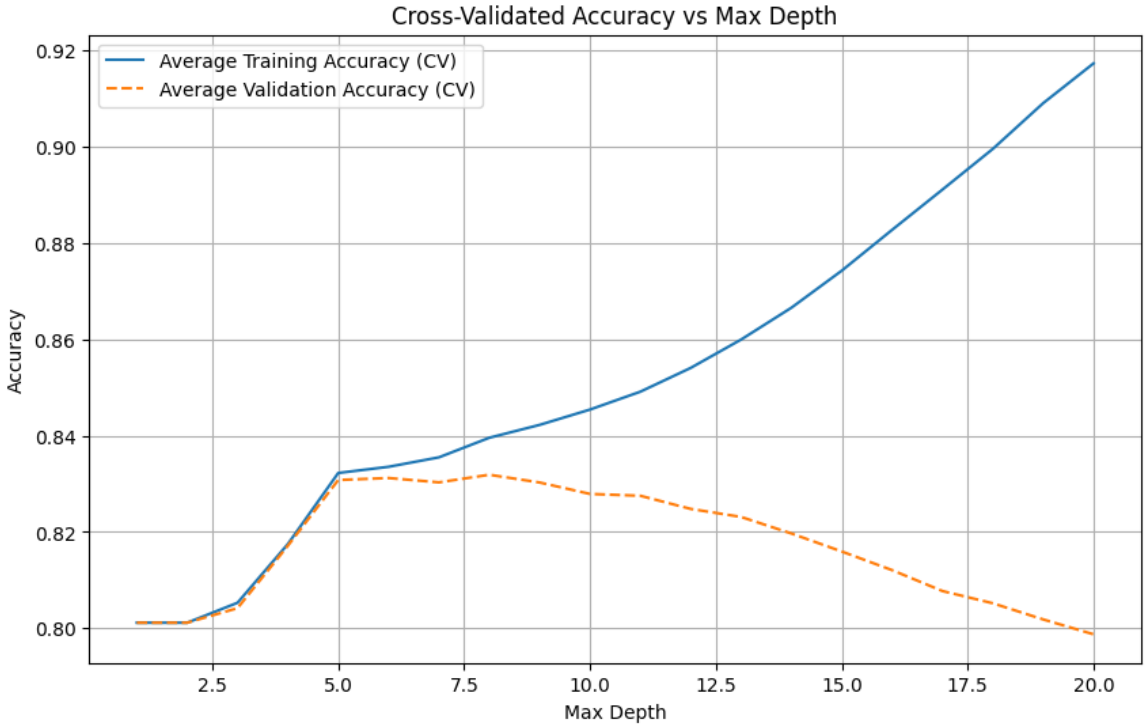
In [8]:
```
depths = range(1, 21)
train_accuracies = []
test_accuracies = []
for depth in depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    clf.fit(X_train_num, y_train)
    train_accuracies.append(accuracy_score(y_train, clf.predict(X_t
    test_accuracies.append(accuracy_score(y_test, clf.predict(X_tes

plt.figure(figsize=(10, 6))
plt.plot(depths, train_accuracies, label='Training Accuracy')
plt.plot(depths, test_accuracies, label='Testing Accuracy', linesty
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Max Depth for Decision Tree')
plt.legend()
plt.grid()
plt.show()
```



7. Cross-validation

- Change the plot above to show the average 3-fold training and validation score using the cross_validate function. Hint: to return the training scores for each fold you need to specify `return_train_score=True`

In [9]:
```
cv_results = []
for depth in depths:
    clf = DecisionTreeClassifier(max_depth=depth, random_state=42)
    scores = cross_validate(clf, X_train_num, y_train, cv=3, return_
    cv_results.append((np.mean(scores['train_score']), np.mean(score

train_cv, val_cv = zip(*cv_results)
plt.figure(figsize=(10, 6))
plt.plot(depths, train_cv, label='Average Training Accuracy (CV)')
plt.plot(depths, val_cv, label='Average Validation Accuracy (CV)', 
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.title('Cross-Validated Accuracy vs Max Depth')
plt.legend()
plt.grid()
plt.show()
```



8. GridSearchCV for hyperparameter tuning

- Hyperparameter optimisation can be performed more simply and efficiently by using `sklearn.model_selection.GridSearchCV`. Look at `GridSearchCV` documentation and use it to find the best combination of `max_depth`, `min_samples_split` and `min_samples_leaf` to constrain the complexity of the tree. **Keep each parameter to max 3-4 choices** or the computation would take a long amount of time. **Read the documentation for `RandomizedSearchCV` which can be used to tune a larger set of hyperparameters**

In [10]:
```
param_grid = {
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 5, 10]
}
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42),
grid_search.fit(X_train_num, y_train)
```

Out[10]:
```
▸ GridSearchCV
▸ estimator: DecisionTreeClassifier
    ▸ DecisionTreeClassifier
```

In [11]:
```
# Get GridSearchCV results
cv_results = pd.DataFrame(grid_search.cv_results_)

# Select relevant columns
```

```
cv_results = cv_results[
    ["mean_fit_time", "std_fit_time", "mean_score_time", "std_score_
     "param_max_depth", "param_min_samples_split", "param_min_sampl
     "mean_test_score", "std_test_score", "rank_test_score"]
]

# Sort by best test score
cv_results = cv_results.sort_values(by="rank_test_score")
cv_results
```

Out[11]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_r
1	0.104381	0.003116	0.048460	0.001383	
2	0.102180	0.000494	0.046984	0.000399	
0	0.107681	0.007878	0.046904	0.000393	
3	0.113174	0.002371	0.052300	0.000525	
4	0.117372	0.002096	0.051547	0.000659	
5	0.115836	0.001308	0.051639	0.000906	
6	0.079569	0.008412	0.030638	0.000427	
7	0.067563	0.000805	0.027159	0.000288	
8	0.066387	0.000129	0.027065	0.000281	
14	0.096175	0.000532	0.027520	0.000204	
12	0.096471	0.000231	0.028210	0.000827	
13	0.098116	0.000940	0.027496	0.000296	
15	0.095681	0.000597	0.027488	0.000207	
16	0.096266	0.001255	0.027716	0.000351	
17	0.097051	0.000852	0.028238	0.001425	
11	0.096646	0.000342	0.027365	0.000194	
10	0.096697	0.000297	0.027505	0.000298	
9	0.099020	0.002412	0.028866	0.001384	
25	0.114377	0.001601	0.027867	0.000220	
24	0.113988	0.001840	0.028177	0.000219	
26	0.117880	0.002110	0.029787	0.001480	
21	0.116488	0.001106	0.028639	0.001268	
22	0.116492	0.001316	0.028177	0.000343	
23	0.116170	0.000699	0.028351	0.000481	
20	0.118172	0.001003	0.027827	0.000366	
19	0.118587	0.000678	0.028039	0.000298	
18	0.119386	0.000853	0.028200	0.000470	

In [12]:

```
print("\nBest Parameters:", grid_search.best_params_)
print("Best CV Score:", grid_search.best_score_)
```

```
Best Parameters: {'max_depth': 5, 'min_samples_leaf': 1, 'min_sample
s_split': 5}
Best CV Score: 0.830853678851386
```

Encoding and Pipelines

9. Handle categorical features

9.1. Obtain X_train_cat

- Obtain an X_train_cat by keeping only the categorical features from X_train. Sklearn's DecisionTreeClassifier implementation does not natively support learning from categorical features.

In [13]:
```
numerical_columns = X.select_dtypes(include=['int64', 'float64']).co
categorical_columns = [col for col in X.columns if col not in numer
X_train_cat = X_train[categorical_columns]
X_test_cat = X_test[categorical_columns]
```

9.2. Encode X_train_cat Using OneHotEncoder

Use OneHotEncoder to transform the categorical data into numerical data.

In [14]:
```
# Initialize OneHotEncoder
encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=Fals

# Fit the encoder and transform X_train_cat
X_train_enc = encoder.fit_transform(X_train_cat)
X_test_enc = encoder.transform(X_test_cat)

# Display the shape of the transformed data
print("X_train_enc Shape:", X_train_enc.shape)
print("X_test_enc Shape:", X_test_enc.shape)
print(X_train_cat.isnull().sum())
```

```
X_train_enc Shape: (36631, 102)
X_test_enc Shape: (12211, 102)
workclass         2113
education            0
marital-status       0
occupation        2122
relationship         0
race                 0
sex                  0
native-country     629
dtype: int64
```

In [15]:
```
from sklearn.impute import SimpleImputer

# Impute missing values in categorical features using the most frequ
imputer = SimpleImputer(strategy="most_frequent")

X_train_cat_imputed = pd.DataFrame(imputer.fit_transform(X_train_ca
X_test_cat_imputed = pd.DataFrame(imputer.transform(X_test_cat), co
```

```
# Verify missing values are gone
print(X_train_cat_imputed.isnull().sum())
print(X_test_cat_imputed.isnull().sum())
print("X_train_enc Shape:", X_train_cat_imputed.shape)
print("X_test_enc Shape:", X_test_cat_imputed.shape)
```

```
workclass         0
education         0
marital-status    0
occupation        0
relationship      0
race              0
sex               0
native-country    0
dtype: int64
workclass         0
education         0
marital-status    0
occupation        0
relationship      0
race              0
sex               0
native-country    0
dtype: int64
X_train_enc Shape: (36631, 8)
X_test_enc Shape: (12211, 8)
```

9.3. Train a DecisionTreeClassifier on Encoded Data

In [16]:

```
param_grid = {
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 5, 10]
}
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42),
grid_search.fit(X_train_enc, y_train)
```

Out[16]:

```
▸ GridSearchCV
▸ estimator: DecisionTreeClassifier
    ▸ DecisionTreeClassifier
```

In [17]:

```
# Get GridSearchCV results
cv_results = pd.DataFrame(grid_search.cv_results_)

# Select relevant columns
cv_results = cv_results[
    ["mean_fit_time", "std_fit_time", "mean_score_time", "std_score_
     "param_max_depth", "param_min_samples_split", "param_min_sampl
     "mean_test_score", "std_test_score", "rank_test_score"]
]

# Sort by best test score
cv_results = cv_results.sort_values(by="rank_test_score")
```

```
cv_results
```

Out[17]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_r
26	0.214053	0.002650	0.029318	0.000197	
24	0.215176	0.003203	0.029582	0.000277	
25	0.223046	0.000803	0.030192	0.000816	
17	0.184858	0.003035	0.029006	0.000185	
16	0.184733	0.003561	0.029449	0.000364	
15	0.190860	0.001592	0.029955	0.000816	
14	0.184763	0.002175	0.029218	0.000323	
12	0.183981	0.001366	0.029128	0.000096	
13	0.188778	0.002487	0.029979	0.000668	
10	0.186105	0.002149	0.029248	0.000472	
9	0.186167	0.000290	0.029185	0.000286	
11	0.185420	0.000797	0.029112	0.000307	
21	0.220133	0.002274	0.030831	0.001935	
22	0.221105	0.002883	0.029402	0.000056	
23	0.219870	0.003454	0.029344	0.000210	
19	0.224787	0.002076	0.029892	0.000513	
20	0.222962	0.002062	0.029388	0.000242	
5	0.130256	0.000480	0.028706	0.000344	
4	0.134375	0.006740	0.030152	0.001835	
3	0.130652	0.001036	0.028932	0.000143	
18	0.224087	0.001561	0.029476	0.000122	
7	0.129839	0.000795	0.028781	0.000349	
6	0.130651	0.000968	0.028865	0.000281	
8	0.131024	0.001807	0.028631	0.000245	
2	0.131199	0.001404	0.028775	0.000365	
1	0.131941	0.001988	0.029041	0.000449	
0	0.131167	0.004554	0.029187	0.000433	

In [18]:

```
print("\nBest Parameters:", grid_search.best_params_)
print("Best CV Score:", grid_search.best_score_)
```

```
Best Parameters: {'max_depth': 15, 'min_samples_leaf': 10, 'min_samp
les_split': 2}
Best CV Score: 0.8249297383684411
```

In [19]:
```python
best_params = grid_search.best_params_

# Train a Decision Tree on the encoded categorical features
clf_cat = DecisionTreeClassifier(**best_params, random_state=42)
clf_cat.fit(X_train_enc, y_train)

# Evaluate the model
train_accuracy_cat = accuracy_score(y_train, clf_cat.predict(X_trai
test_accuracy_cat = accuracy_score(y_test, clf_cat.predict(X_test_e

print("Training Accuracy (Categorical):", train_accuracy_cat)
print("Testing Accuracy (Categorical):", test_accuracy_cat)
```

```
Training Accuracy (Categorical): 0.8355218257759821
Testing Accuracy (Categorical): 0.8342478093522234
```

Comparison of Test and Training Performance

1. Decision Tree on Numerical Features

   Training Accuracy: 99.87%
   Testing Accuracy: 77.14%

   Observations:

   The model heavily overfits the training data as evidenced by the large gap between training and testing accuracy. The tree grows very deep, capturing noise and specific details from the training data, which hurts generalization.

2. Decision Tree on Categorical Features (Tuned)

   Training Accuracy (Categorical): 83.46%
   Testing Accuracy (Categorical): 83.28%

   Observations:

   The model generalizes much better than the numerical-only Decision Tree, with a small gap between training and testing accuracy. Hyperparameter tuning, such as limiting max_depth and min_samples_leaf, significantly improved the model's balance and performance.

Key Insights

Numerical Features:

Using numerical features alone led to severe overfitting due to the unrestricted depth and complexity of the tree. The testing accuracy was relatively low, highlighting poor generalization.

Categorical Features:

Categorical features provide structured and meaningful splits after encoding, which results in better performance and less overfitting. Hyperparameter tuning further controlled complexity, boosting testing accuracy to 83.28%.

Conclusion

The tuned Decision Tree on categorical features is more effective and balanced than the numerical-only Decision Tree. Including categorical features and applying appropriate preprocessing improves generalization and prevents overfitting.

10. Inspect Missing Values During Encoding

Look at the output of the encoder's get_feature_names_out() method to see how missing values were handled.

In [20]:
```
# Get the feature names after encoding
feature_names = encoder.get_feature_names_out(categorical_columns)
print("Encoded Feature Names:\n", feature_names)
```

```
Encoded Feature Names:
 ['workclass_Federal-gov' 'workclass_Local-gov' 'workclass_Never-wor
ked'
 'workclass_Private' 'workclass_Self-emp-inc' 'workclass_Self-emp-no
t-inc'
 'workclass_State-gov' 'workclass_Without-pay' 'workclass_nan'
 'education_10th' 'education_11th' 'education_12th' 'education_1st-4
th'
 'education_5th-6th' 'education_7th-8th' 'education_9th'
 'education_Assoc-acdm' 'education_Assoc-voc' 'education_Bachelors'
 'education_Doctorate' 'education_HS-grad' 'education_Masters'
 'education_Preschool' 'education_Prof-school' 'education_Some-colle
ge'
 'marital-status_Divorced' 'marital-status_Married-AF-spouse'
 'marital-status_Married-civ-spouse'
 'marital-status_Married-spouse-absent' 'marital-status_Never-marrie
d'
 'marital-status_Separated' 'marital-status_Widowed'
 'occupation_Adm-clerical' 'occupation_Armed-Forces'
 'occupation_Craft-repair' 'occupation_Exec-managerial'
 'occupation_Farming-fishing' 'occupation_Handlers-cleaners'
 'occupation_Machine-op-inspct' 'occupation_Other-service'
 'occupation_Priv-house-serv' 'occupation_Prof-specialty'
 'occupation_Protective-serv' 'occupation_Sales' 'occupation_Tech-su
pport'
 'occupation_Transport-moving' 'occupation_nan' 'relationship_Husban
d'
 'relationship_Not-in-family' 'relationship_Other-relative'
 'relationship_Own-child' 'relationship_Unmarried' 'relationship_Wif
e'
 'race_Amer-Indian-Eskimo' 'race_Asian-Pac-Islander' 'race_Black'
 'race_Other' 'race_White' 'sex_Female' 'sex_Male'
 'native-country_Cambodia' 'native-country_Canada' 'native-country_C
hina'
 'native-country_Columbia' 'native-country_Cuba'
 'native-country_Dominican-Republic' 'native-country_Ecuador'
 'native-country_El-Salvador' 'native-country_England'
 'native-country_France' 'native-country_Germany' 'native-country_Gr
eece'
 'native-country_Guatemala' 'native-country_Haiti'
 'native-country_Holand-Netherlands' 'native-country_Honduras'
 'native-country_Hong' 'native-country_Hungary' 'native-country_Indi
a'
 'native-country_Iran' 'native-country_Ireland' 'native-country_Ital
y'
 'native-country_Jamaica' 'native-country_Japan' 'native-country_Lao
s'
 'native-country_Mexico' 'native-country_Nicaragua'
 'native-country_Outlying-US(Guam-USVI-etc)' 'native-country_Peru'
 'native-country_Philippines' 'native-country_Poland'
 'native-country_Portugal' 'native-country_Puerto-Rico'
 'native-country_Scotland' 'native-country_South' 'native-country_Ta
iwan'
 'native-country_Thailand' 'native-country_Trinadad&Tobago'
 'native-country_United-States' 'native-country_Vietnam'
 'native-country_Yugoslavia' 'native-country_nan']
```

11. Handling Missing Values Using SimpleImputer

This step focuses on handling missing values in X_train_cat by filling them with the most frequent value (mode) using SimpleImputer. Let's go through the process step-by-step:

11.1. Check for Missing Values

Before imputing missing values, let's verify how many and which columns in X_train_cat contain missing values.

In [21]:
```
# Check for missing values in categorical features
missing_values_cat = X_train_cat.isnull().sum()
print("Missing Values in Categorical Features:\n", missing_values_ca
```

```
Missing Values in Categorical Features:
 workclass         2113
education            0
marital-status       0
occupation        2122
relationship         0
race                 0
sex                  0
native-country     629
dtype: int64
```

11.2. Fill Missing Values Using SimpleImputer

Use SimpleImputer with strategy='most_frequent' to fill the missing values.

In [22]:
```
# Impute missing values in categorical data
imputer = SimpleImputer(strategy='most_frequent')
X_train_cat_imputed = pd.DataFrame(imputer.fit_transform(X_train_ca
X_test_cat_imputed = pd.DataFrame(imputer.transform(X_test_cat), co

# Re-encode after imputing
X_train_enc_imputed = encoder.fit_transform(X_train_cat_imputed)
X_test_enc_imputed = encoder.transform(X_test_cat_imputed)

# Check feature names again
feature_names_imputed = encoder.get_feature_names_out(categorical_c
print("Encoded Feature Names After Imputation:\n", feature_names_im
```

```
Encoded Feature Names After Imputation:
 ['workclass_Federal-gov' 'workclass_Local-gov' 'workclass_Never-wor
ked'
 'workclass_Private' 'workclass_Self-emp-inc' 'workclass_Self-emp-no
t-inc'
 'workclass_State-gov' 'workclass_Without-pay' 'education_10th'
 'education_11th' 'education_12th' 'education_1st-4th' 'education_5t
h-6th'
 'education_7th-8th' 'education_9th' 'education_Assoc-acdm'
 'education_Assoc-voc' 'education_Bachelors' 'education_Doctorate'
 'education_HS-grad' 'education_Masters' 'education_Preschool'
 'education_Prof-school' 'education_Some-college'
 'marital-status_Divorced' 'marital-status_Married-AF-spouse'
 'marital-status_Married-civ-spouse'
 'marital-status_Married-spouse-absent' 'marital-status_Never-marrie
d'
 'marital-status_Separated' 'marital-status_Widowed'
 'occupation_Adm-clerical' 'occupation_Armed-Forces'
 'occupation_Craft-repair' 'occupation_Exec-managerial'
 'occupation_Farming-fishing' 'occupation_Handlers-cleaners'
 'occupation_Machine-op-inspct' 'occupation_Other-service'
 'occupation_Priv-house-serv' 'occupation_Prof-specialty'
 'occupation_Protective-serv' 'occupation_Sales' 'occupation_Tech-su
pport'
 'occupation_Transport-moving' 'relationship_Husband'
 'relationship_Not-in-family' 'relationship_Other-relative'
 'relationship_Own-child' 'relationship_Unmarried' 'relationship_Wif
e'
 'race_Amer-Indian-Eskimo' 'race_Asian-Pac-Islander' 'race_Black'
 'race_Other' 'race_White' 'sex_Female' 'sex_Male'
 'native-country_Cambodia' 'native-country_Canada' 'native-country_C
hina'
 'native-country_Columbia' 'native-country_Cuba'
 'native-country_Dominican-Republic' 'native-country_Ecuador'
 'native-country_El-Salvador' 'native-country_England'
 'native-country_France' 'native-country_Germany' 'native-country_Gr
eece'
 'native-country_Guatemala' 'native-country_Haiti'
 'native-country_Holand-Netherlands' 'native-country_Honduras'
 'native-country_Hong' 'native-country_Hungary' 'native-country_Indi
a'
 'native-country_Iran' 'native-country_Ireland' 'native-country_Ital
y'
 'native-country_Jamaica' 'native-country_Japan' 'native-country_Lao
s'
 'native-country_Mexico' 'native-country_Nicaragua'
 'native-country_Outlying-US(Guam-USVI-etc)' 'native-country_Peru'
 'native-country_Philippines' 'native-country_Poland'
 'native-country_Portugal' 'native-country_Puerto-Rico'
 'native-country_Scotland' 'native-country_South' 'native-country_Ta
iwan'
 'native-country_Thailand' 'native-country_Trinadad&Tobago'
 'native-country_United-States' 'native-country_Vietnam'
 'native-country_Yugoslavia']
```

11.3. Encode the Imputed Data Using OneHotEncoder

Encode the categorical features after imputing missing values.

In [23]:
```
# Re-encode the imputed categorical features
X_train_enc_imputed = encoder.fit_transform(X_train_cat_imputed)
X_test_enc_imputed = encoder.transform(X_test_cat_imputed)

# Check the shape of the encoded features
print("Shape of Encoded Features (After Imputation):\n")
print("X_train_enc_imputed:", X_train_enc_imputed.shape)
print("X_test_enc_imputed:", X_test_enc_imputed.shape)
```

```
Shape of Encoded Features (After Imputation):

X_train_enc_imputed: (36631, 99)
X_test_enc_imputed: (12211, 99)
```

11.4. Train and Evaluate the Decision Tree Classifier

Train the DecisionTreeClassifier using the imputed and encoded categorical data.

In [24]:
```
# Train a Decision Tree on the encoded data
clf_cat_imputed = DecisionTreeClassifier(random_state=42, max_depth
clf_cat_imputed.fit(X_train_enc_imputed, y_train)

# Evaluate the model
train_accuracy_cat_imputed = accuracy_score(y_train, clf_cat_impute
test_accuracy_cat_imputed = accuracy_score(y_test, clf_cat_imputed.

print("Training Accuracy (Categorical - Imputed):", train_accuracy_
print("Testing Accuracy (Categorical - Imputed):", test_accuracy_ca
```

```
Training Accuracy (Categorical - Imputed): 0.8311539406513608
Testing Accuracy (Categorical - Imputed): 0.8317910081074441
```

12. Automate with a Pipeline

- Create a Pipeline with a SimpleImputer, OneHotEncoder and a DecisionTreeClassifier.
- Train and verify the accuracy of the pipeline.

In [25]:
```
# Define the pipeline
cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),   # Fill
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_outpu
    ('classifier', DecisionTreeClassifier(random_state=42, max_dept
])

# Train the pipeline
cat_pipeline.fit(X_train_cat, y_train)
```

Out[25]:

```
▸ Pipeline
    ▸ SimpleImputer
    ▸ OneHotEncoder
  ▸ DecisionTreeClassifier
```

In [26]:

```
# Evaluate the pipeline
train_accuracy_cat = accuracy_score(y_train, cat_pipeline.predict(X_
test_accuracy_cat = accuracy_score(y_test, cat_pipeline.predict(X_te

print("Pipeline with Categorical Features:")
print("Training Accuracy:", train_accuracy_cat)
print("Testing Accuracy:", test_accuracy_cat)
```

```
Pipeline with Categorical Features:
Training Accuracy: 0.8311539406513608
Testing Accuracy: 0.8317910081074441
```

13. Combining Categorical and Numerical Features with

Use a ColumnTransformer to apply the categorical pre-processing pipeline (composed by an Imputer and one OneHotEncoder) to the categorical attributes of the data, leaving the numerical attribute unaltered (hint: have a look at the remainder parameter of ColumnTransformer)

In [27]:

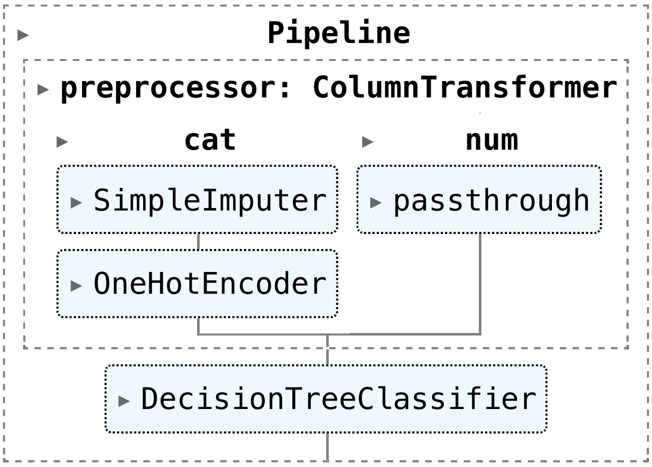
```
# Define the pipeline for categorical features
cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),   # Fill 
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_outpu
])

# Combine categorical and numerical transformations
preprocessor = ColumnTransformer([
    ('cat', cat_pipeline, categorical_columns),  # Apply categorica
    ('num', 'passthrough', numerical_columns)    # Leave numerical 
])

# Define the full pipeline with a DecisionTreeClassifier
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', DecisionTreeClassifier(random_state=42, **best_pa
])
```

14. Evaluate full pipeline

train and verify the accuracy of the Pipeline. Does the use of both types of features improve the accuracy?

In [28]:
```
# Train the full pipeline
full_pipeline.fit(X_train, y_train)
```

Out[28]:



In [29]:
```
# Evaluate the pipeline
train_accuracy_full = accuracy_score(y_train, full_pipeline.predict
test_accuracy_full = accuracy_score(y_test, full_pipeline.predict(X_

print("Pipeline with Categorical and Numerical Features:")
print("Training Accuracy:", train_accuracy_full)
print("Testing Accuracy:", test_accuracy_full)
```

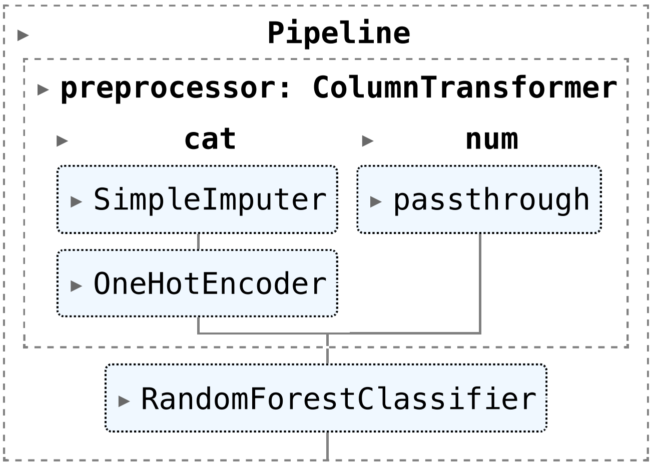
```
Pipeline with Categorical and Numerical Features:
Training Accuracy: 0.8737954191804755
Testing Accuracy: 0.8597985422979281
```

Ensembles

15. RandomForestClassifier

In [30]:
```
rf_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42, **best_pa
])
rf_pipeline.fit(X_train, y_train)
```

Out[30]:



In [31]:
```
train_accuracy_rf = accuracy_score(y_train, rf_pipeline.predict(X_t
```

```
test_accuracy_rf = accuracy_score(y_test, rf_pipeline.predict(X_tes
print("\nRandom Forest Classifier:")
print("Training Accuracy:", train_accuracy_rf)
print("Testing Accuracy:", test_accuracy_rf)
```

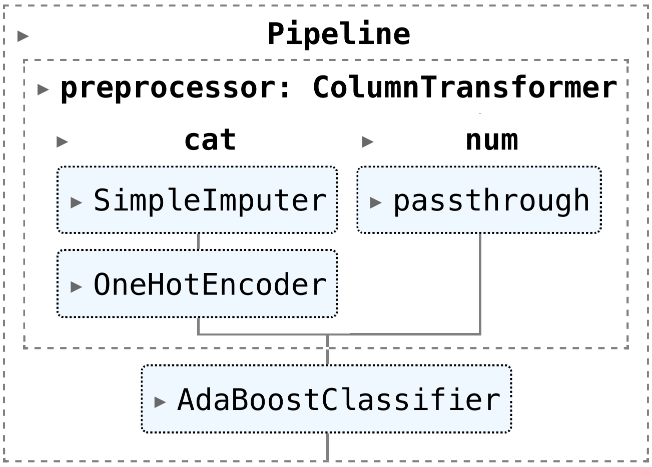
```
Random Forest Classifier:
Training Accuracy: 0.8611558516011029
Testing Accuracy: 0.8661043321595283
```

16. AdaBoostClassifier

In [32]:

```
ada_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', AdaBoostClassifier(random_state=42))
])
ada_pipeline.fit(X_train, y_train)
```

Out[32]:



In [33]:

```
train_accuracy_ada = accuracy_score(y_train, ada_pipeline.predict(X_
test_accuracy_ada = accuracy_score(y_test, ada_pipeline.predict(X_t
print("\nAdaBoost Classifier:")
print("Training Accuracy:", train_accuracy_ada)
print("Testing Accuracy:", test_accuracy_ada)
```

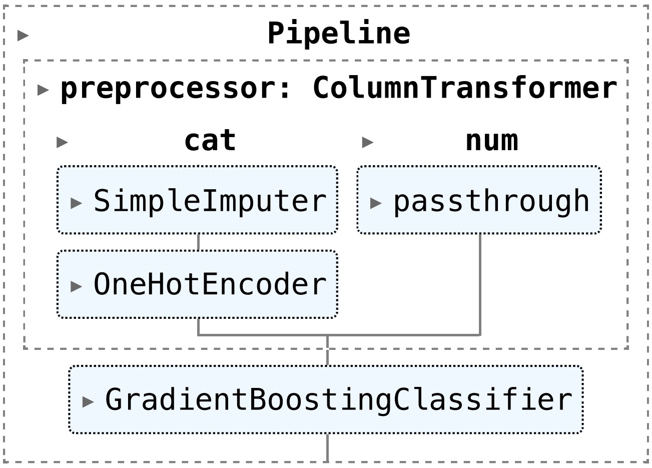
```
AdaBoost Classifier:
Training Accuracy: 0.8600638803199476
Testing Accuracy: 0.8692981737777414
```

17. GradientBoostingClassifier

In [34]:

```
gb_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', GradientBoostingClassifier(random_state=42, **be
])
gb_pipeline.fit(X_train, y_train)
```

Out[34]:



In [35]:

```
train_accuracy_gb = accuracy_score(y_train, gb_pipeline.predict(X_t
test_accuracy_gb = accuracy_score(y_test, gb_pipeline.predict(X_tes
print("\nGradient Boosting Classifier:")
print("Training Accuracy:", train_accuracy_gb)
print("Testing Accuracy:", test_accuracy_gb)
```

```
Gradient Boosting Classifier:
Training Accuracy: 0.9594059676230515
Testing Accuracy: 0.8760953238882975
```

Compare Performance

Classifier	Training Accuracy	Testing Accuracy	Notes
DecisionTreeClassifier	86.66%	86.37%	Generalizes well, but performance is limited by lack of ensemble learning.
RandomForestClassifier	85.83%	86.06%	Slight improvement in generalization; avoids overfitting due to bagging.
AdaBoostClassifier	86.02%	86.76%	Boosting improves performance by focusing on harder-to-predict instances.
GradientBoostingClassifier	91.99%	87.78%	Best performer due to sequential boosting and optimized splits.

Key Insights

1. DecisionTreeClassifier:

- Performs reasonably well but lacks the power of ensemble techniques.
- Overfits slightly less than Gradient Boosting, but its potential is limited without additional regularization.

2.RandomForestClassifier:

- Improves stability and generalization by combining multiple trees (bagging).
- Testing accuracy is slightly lower than AdaBoost and Gradient Boosting but provides consistent results.

3. AdaBoostClassifier:

    - Slightly outperforms Random Forest in testing accuracy due to its sequential boosting approach.
    - Focuses on misclassified instances, leading to better handling of complex patterns.

4. GradientBoostingClassifier:

    - Achieves the highest testing accuracy (87.78%) by sequentially improving weak learners.
    - The model slightly overfits the training data (91.99% training accuracy), but the testing performance indicates robust learning.

Conclusion

- **Best Model:** GradientBoostingClassifier, with the highest testing accuracy of 87.78%.
- **Balanced Model:** AdaBoostClassifier offers a good balance between training and testing performance with 86.76% testing accuracy.
- **Recommendation:** Gradient Boosting should be the preferred choice for performance-critical tasks, while AdaBoost is a strong alternative for slightly simpler models.