COMP8260 - AI Systems Class 2

Full Jupyter Notebook Solution (Tasks 1-17)

This notebook provides structured solutions to **all tasks (1-17)** from the class document. Each task includes explanations, implementation, and table comparisons where needed.

Task 1: Load and Explore the Dataset

We load the **Adult Dataset** from `openml.org` using `fetch_openml`. The dataset contains **demographic and employment information**, and we aim to predict whether a person earns `<=50K` or `>50K` per year.

In [1]:

```
from sklearn.datasets import fetch_openml
import pandas as pd

# Load dataset
dataset = fetch_openml(data_id=1590, as_frame=True)

# Extract features and target
X = dataset.data
y = dataset.target

# Display basic info
print("Dataset Info:")
print(X.info())

# Check missing values
print("\nMissing values per column:")
print(X.isnull().sum())

# Check dataset size
print("\nDataset Shape:", X.shape)

# Target class distribution
print("\nTarget Distribution:")
print(y.value_counts())
```

```
/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(
```

```
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
None

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 Shape: (48842, 14)

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

Task 2: Split Data into Training and Testing Sets

We split the dataset into **80% training** and **20% testing** using `train_test_split`.

In [2]:
```python
from sklearn.model_selection import train_test_split

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=

# Display shapes
print("Training Set Shape:", X_train.shape)
print("Testing Set Shape:", X_test.shape)
```

```
Training Set Shape: (39073, 14)
Testing Set Shape: (9769, 14)
```

Task 3: Extract Numerical Features

We select only the **numerical columns** for our first Decision Tree model.

In [3]:
```python
# Select numerical features
X_train_num = X_train.select_dtypes(include=['int64', 'float64'])
X_test_num = X_test.select_dtypes(include=['int64', 'float64'])

# Display shapes
print("X_train_num Shape:", X_train_num.shape)
print("X_test_num Shape:", X_test_num.shape)
```

```
X_train_num Shape: (39073, 6)
X_test_num Shape: (9769, 6)
```

Task 4: Train a Decision Tree Classifier on Numerical Data

We train a `DecisionTreeClassifier` using only numerical features and evaluate it.

In [4]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Train Decision Tree
clf_num = DecisionTreeClassifier(random_state=42)
clf_num.fit(X_train_num, y_train)

# Evaluate
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 results
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())
```

```
Training Accuracy: 0.9986691577304021
Testing Accuracy: 0.7714197973180469
Tree Depth: 52
Number of Leaves: 7857
```

Tasks 5-8: Encode Categorical Data & Train on Encoded Features

We apply **OneHotEncoding** to categorical variables, retrain the Decision Tree, and optimize hyperparameters using `GridSearchCV`.

In [5]:
```
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Select categorical features
categorical_features = X_train.select_dtypes(include=['category', '

# Define preprocessing pipeline
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_outpu
])

# Apply encoding
preprocessor = ColumnTransformer([
    ('cat', categorical_transformer, categorical_features)
], remainder='passthrough')

X_train_enc = preprocessor.fit_transform(X_train)
X_test_enc = preprocessor.transform(X_test)

# Check encoded shape
print("X_train_enc Shape:", X_train_enc.shape)
print("X_test_enc Shape:", X_test_enc.shape)

# Train Decision Tree on Encoded Data
clf_cat = DecisionTreeClassifier(random_state=42)
clf_cat.fit(X_train_enc, y_train)

# Evaluate
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 results
print("Training Accuracy (Categorical):", train_accuracy_cat)
print("Testing Accuracy (Categorical):", test_accuracy_cat)
print("Tree Depth:", clf_cat.get_depth())
print("Number of Leaves:", clf_cat.get_n_leaves())
```

```
X_train_enc Shape: (39073, 105)
X_test_enc Shape: (9769, 105)
Training Accuracy (Categorical): 0.999872034397154
Testing Accuracy (Categorical): 0.8220902855972976
Tree Depth: 53
Number of Leaves: 5752
```

Tasks 9-12: Hyperparameter Optimization

We use `GridSearchCV` to find the best hyperparameters.

In [6]:
```
from sklearn.model_selection import GridSearchCV

# Define hyperparameter grid
param_grid = {
    'max_depth': [10, 20, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5]
}

# Perform Grid Search
grid_search = GridSearchCV(DecisionTreeClassifier(), param_grid, cv
grid_search.fit(X_train_enc, y_train)

# Display best parameters
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)
```

```
Best Parameters: {'max_depth': 10, 'min_samples_leaf': 5, 'min_sampl
es_split': 10}
Best Score: 0.8544519178924169
```

Tasks 13-17: Final Model Evaluation & Table Comparisons

We train the final Decision Tree using **optimal hyperparameters** and compare the results with `class2-output-only.ipynb`.

In [7]:
```
# Train best model using best parameters
clf_best = DecisionTreeClassifier(**grid_search.best_params_, rando
clf_best.fit(X_train_enc, y_train)

# Evaluate
train_accuracy_best = accuracy_score(y_train, clf_best.predict(X_tr
test_accuracy_best = accuracy_score(y_test, clf_best.predict(X_test

# Print results
print("Optimized Training Accuracy:", train_accuracy_best)
print("Optimized Testing Accuracy:", test_accuracy_best)
print("Optimized Tree Depth:", clf_best.get_depth())
print("Optimized Number of Leaves:", clf_best.get_n_leaves())
```

```
# Compare with class2-output-only.ipynb
comparison_data = {
    "Metric": ["Training Accuracy", "Testing Accuracy", "Tree Depth'
    "Your Model": [train_accuracy_best, test_accuracy_best, clf_bes
    "Reference Output": [0.8629, 0.8251, 64, 7405]
}

df_comparison = pd.DataFrame(comparison_data)
df_comparison
```

```
Optimized Training Accuracy: 0.8665830624728074
Optimized Testing Accuracy: 0.8636503224485618
Optimized Tree Depth: 10
Optimized Number of Leaves: 250
```

Out[7]:

	Metric	Your Model	Reference Output
0	Training Accuracy	0.866583	0.8629
1	Testing Accuracy	0.863650	0.8251
2	Tree Depth	10.000000	64.0000
3	Number of Leaves	250.000000	7405.0000

In [8]:
```
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassi
```

Task 15: Train a Random Forest Classifier

We now train a **RandomForestClassifier** and evaluate its performance.

In [9]:
```
# Train Random Forest
clf_rf = RandomForestClassifier(n_estimators=100, random_state=42)
clf_rf.fit(X_train_enc, y_train)

# Evaluate
train_accuracy_rf = accuracy_score(y_train, clf_rf.predict(X_train_
test_accuracy_rf = accuracy_score(y_test, clf_rf.predict(X_test_enc

# Print results
print("Random Forest Training Accuracy:", train_accuracy_rf)
print("Random Forest Testing Accuracy:", test_accuracy_rf)
print("Random Forest Number of Trees:", len(clf_rf.estimators_))
```

```
Random Forest Training Accuracy: 0.9998464412765848
Random Forest Testing Accuracy: 0.8584297266864571
Random Forest Number of Trees: 100
```

Task 16: Train an AdaBoost Classifier

We now train an **AdaBoostClassifier**, which is a boosting technique that improves weak learners.

In [10]:
```
# Train AdaBoost
```

```
clf_ada = AdaBoostClassifier(n_estimators=100, random_state=42)
clf_ada.fit(X_train_enc, y_train)

# Evaluate
train_accuracy_ada = accuracy_score(y_train, clf_ada.predict(X_trai
test_accuracy_ada = accuracy_score(y_test, clf_ada.predict(X_test_en

# Print results
print("AdaBoost Training Accuracy:", train_accuracy_ada)
print("AdaBoost Testing Accuracy:", test_accuracy_ada)
```

```
AdaBoost Training Accuracy: 0.8641005297775958
AdaBoost Testing Accuracy: 0.8732725969904801
```

Task 17: Train a Gradient Boosting Classifier

We now train a **GradientBoostingClassifier**, another boosting technique that reduces bias.

In [11]:
```
# Train Gradient Boosting
clf_gb = GradientBoostingClassifier(n_estimators=100, random_state=
clf_gb.fit(X_train_enc, y_train)

# Evaluate
train_accuracy_gb = accuracy_score(y_train, clf_gb.predict(X_train_
test_accuracy_gb = accuracy_score(y_test, clf_gb.predict(X_test_enc

# Print results
print("Gradient Boosting Training Accuracy:", train_accuracy_gb)
print("Gradient Boosting Testing Accuracy:", test_accuracy_gb)
```

```
Gradient Boosting Training Accuracy: 0.8670693317636219
Gradient Boosting Testing Accuracy: 0.8722489507626164
```

Model Comparison

We compare the performance of **Decision Tree, Random Forest, AdaBoost, and Gradient Boosting**.

In [12]:
```
# Create a comparison table
comparison_data = {
    "Model": ["Decision Tree", "Random Forest", "AdaBoost", "Gradie
    "Training Accuracy": [train_accuracy_best, train_accuracy_rf, t
    "Testing Accuracy": [test_accuracy_best, test_accuracy_rf, test_
}

df_comparison = pd.DataFrame(comparison_data)

df_comparison
```

Out[12]:

	Model	Training Accuracy	Testing Accuracy
0	Decision Tree	0.866583	0.863650
1	Random Forest	0.999846	0.858430
2	AdaBoost	0.864101	0.873273
3	Gradient Boosting	0.867069	0.872249

✅ AdaBoost

Best testing accuracy → 0.8733 (highest among all models)

Balanced training accuracy → 0.8641 (not overfitting too much)

Boosting method helps reduce bias and generalizes well

In [ ]: