DAT341 / DIT867 Applied machine learning
Assignment 2

Chunqiu Xia Chunqiu@student.chalmers.se
Yicheng Li yicheng@student.chalmers.se
Wenjun Tian wenjunt@chalmers.se

February 2, 2025

Task 1: Working with a dataset with categorical features

Step 1: Reading the data

Firstly, we should read the data from both training dataset and testing dataset.

[4]:
```
import pandas as pd

train_data = pd.read_csv('adult_train.csv')
test_data = pd.read_csv('adult_test.csv')
X_train = train_data.drop(columns=['target'])
y_train = train_data['target']

X_test = test_data.drop(columns=['target'])
y_test = test_data['target']

print("X_train:")
print(X_train.head())
print("\n y_train:")
print(y_train.head())
print("\n X_test:")
print(X_test.head())
print("\n y_test:")
print(y_test.head())
```

```
X_train:
   age workclass     education  education-num      marital-status  \
0   27   Private  Some-college             10            Divorced
1   27   Private     Bachelors             13       Never-married
2   25   Private    Assoc-acdm             12  Married-civ-spouse
3   46   Private       5th-6th              3  Married-civ-spouse
4   45   Private          11th              7            Divorced

        occupation   relationship                race     sex  capital-gain  \
0     Adm-clerical      Unmarried               White  Female             0
1   Prof-specialty  Not-in-family               White  Female             0
2            Sales        Husband               White    Male             0
```

```
3  Transport-moving        Husband  Amer-Indian-Eskimo    Male             0
4  Transport-moving  Not-in-family               White    Male             0

   capital-loss  hours-per-week native-country
0             0              44  United-States
1             0              40  United-States
2             0              40  United-States
3          1902              40  United-States
4          2824              76  United-States

 y_train:
0    <=50K
1    <=50K
2    <=50K
3    <=50K
4     >50K
Name: target, dtype: object

 X_test:
   age  workclass     education  education-num      marital-status  \
0   25    Private          11th              7       Never-married
1   38    Private       HS-grad              9  Married-civ-spouse
2   28  Local-gov    Assoc-acdm             12  Married-civ-spouse
3   44    Private  Some-college             10  Married-civ-spouse
4   18          ?  Some-college             10       Never-married

          occupation relationship   race     sex  capital-gain  capital-loss  \
0  Machine-op-inspct    Own-child  Black    Male             0             0
1    Farming-fishing      Husband  White    Male             0             0
2    Protective-serv      Husband  White    Male             0             0
3  Machine-op-inspct      Husband  Black    Male          7688             0
4                  ?    Own-child  White  Female             0             0

   hours-per-week native-country
0              40  United-States
1              50  United-States
2              40  United-States
3              40  United-States
4              30  United-States

 y_test:
0    <=50K
1    <=50K
2     >50K
3     >50K
4    <=50K
Name: target, dtype: object
```

Step 2: Encoding the features as numbers.

[7]:
```
import pandas as pd
from sklearn.feature_extraction import DictVectorizer

dicts_for_train = X_train.to_dict('records')
dicts_for_test = X_test.to_dict('records')

dv = DictVectorizer()
X_train_encoded = dv.fit_transform(dicts_for_train)
X_test_encoded = dv.transform(dicts_for_test)

print(f"Training: {X_train_encoded.shape}")
print(f"Testing: {X_test_encoded.shape}")
```

```
Training: (32561, 107)
Testing: (16281, 107)
```

Step 3: Combining the steps

Then we could define the step to simply the process.

[10]:
```
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction import DictVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

pipeline = make_pipeline(
    DictVectorizer(),
    DecisionTreeClassifier()
)

pipeline.fit(dicts_for_train, y_train)
y_pre = pipeline.predict(dicts_for_test)
accuracy = accuracy_score(y_test, y_pre)
print(f"The accuracy of this model is: {accuracy:.4f}")
```

```
The accuracy of this model is: 0.8160
```

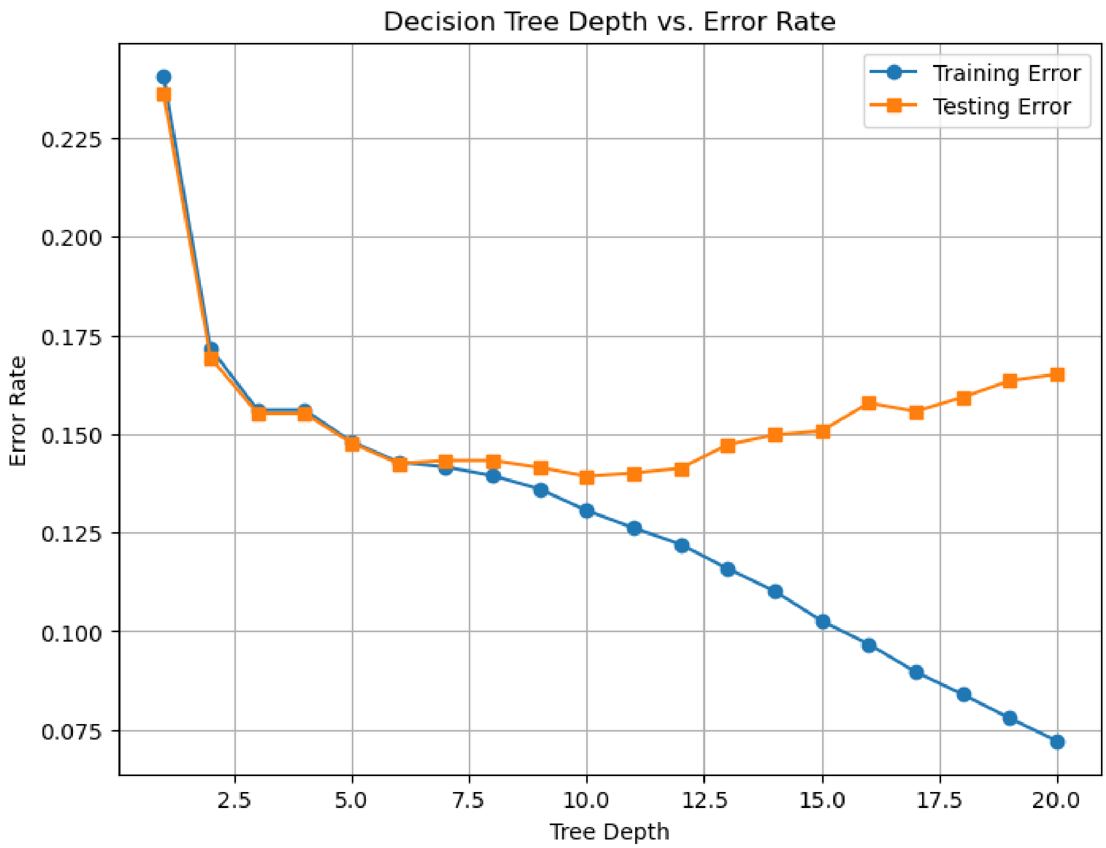
Task 2: Decision trees and random forests

2.1 Underfitting and overfitting in decision tree classifiers.

In this part, we will train `DecisionTreeClassifier` on the given dataset. The evaluation metric we pick is the `Gini index`

[13]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
```

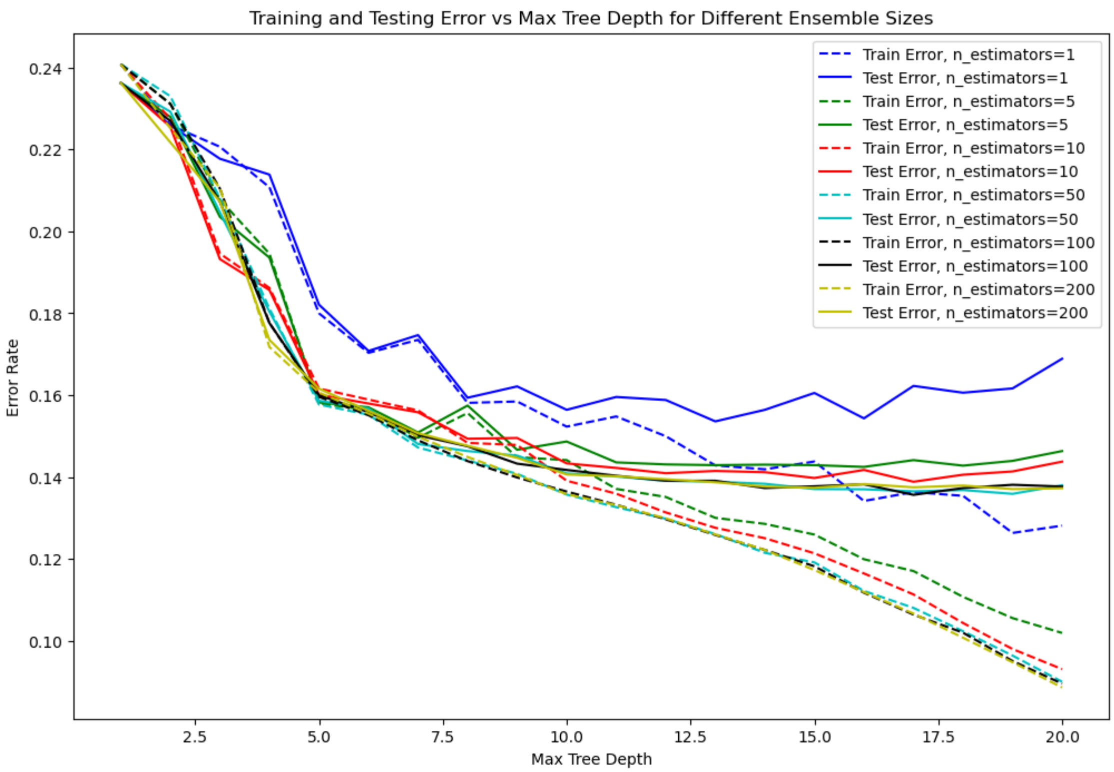
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score


# record train/test errors given depth of trees
train_errors = []
test_errors = []
depths = range(1, 21)

for depth in depths:
    clf = DecisionTreeClassifier(criterion='gini', max_depth=depth,␣
 ↪random_state=114514)
    clf.fit(X_train_encoded, y_train)

    # E_train
    train_pred = clf.predict(X_train_encoded)
    # E_test
    test_pred = clf.predict(X_test_encoded)

    train_errors.append(1 - accuracy_score(y_train, train_pred))
    test_errors.append(1 - accuracy_score(y_test, test_pred))

# plot error-depth relation
plt.figure(figsize=(8, 6))
plt.plot(depths, train_errors, label='Training Error', marker='o')
plt.plot(depths, test_errors, label='Testing Error', marker='s')
plt.xlabel('Tree Depth')
plt.ylabel('Error Rate')
plt.title('Decision Tree Depth vs. Error Rate')
plt.legend()
plt.grid()
plt.show()
```



2.2 Underfitting and overfitting in random forest classifiers.

[16]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_train, X_test = X_train_encoded, X_test_encoded

tree_depths = range(1, 21)
# ensemble size
n_estimators_list = [1, 5, 10, 50, 100, 200]
colors = ['b', 'g', 'r', 'c', 'k', 'y']

plt.figure(figsize=(12, 8))

# record errors
for (n_estimators, color) in zip(n_estimators_list, colors):
```

```
    train_errors = []
    test_errors = []

    for depth in tree_depths:
        # train
        clf = RandomForestClassifier(n_estimators=n_estimators, max_depth=depth,␣
↪criterion='gini', random_state=1919810, n_jobs=-1)
        clf.fit(X_train, y_train)

        # calculate errors
        y_train_pred = clf.predict(X_train)
        y_test_pred = clf.predict(X_test)
        train_error = 1 - accuracy_score(y_train, y_train_pred)
        test_error = 1 - accuracy_score(y_test, y_test_pred)

        train_errors.append(train_error)
        test_errors.append(test_error)

    # plot error-depth relationshjp for given ensemble size
    plt.plot(tree_depths, train_errors, label=f'Train Error,␣
↪n_estimators={n_estimators}', linestyle='dashed', color=color)
    plt.plot(tree_depths, test_errors, label=f'Test Error,␣
↪n_estimators={n_estimators}', color=color)

plt.xlabel('Max Tree Depth')
plt.ylabel('Error Rate')
plt.title('Training and Testing Error vs Max Tree Depth for Different Ensemble␣
↪Sizes')
plt.legend()
plt.show()
```



2.3 Discussion

What's the difference between the curve for a decision tree and for a random forest with an ensemble size of 1, and why do we see this difference? The error curves of random forest with ensemble size 1 have bigger bias (E_train ˆ 1/2) then those of decision tree.

This is caused by bagging (both data and feature bagging). In random forest, the real training data is randomly sampled from the trainning set WITH REPLACEMENT, meaning that there could be some data points missing in the real training data. Moreover, the selected features are also subsets of all the features, thus when n is small, some features may not be trained adequately. Therefore, the acurracy of predicting on the training data is much lower. This leads to higher bias (or to say E_train)

What happens with the curve for random forests as the ensemble size grows? As the ensemble size grows, the variance is decreasing, given the same max depth of trees. This manifest that bagging is useful for lowering variance by adding randomness and averaging.

When the ensemble size is small (like 1 and 5 in the graph), as the ensemble size grows, the bias is also decreasing. The reason is stated in the previos question. However, as the ensemble size continues to grow, the bias remains relatively low, since the sampling is enough to cover all the training data/features.

What happens with the best observed test set accuracy as the ensemble size grows? As the ensemble size grows, the best observed test set accuracy generally appears at bigger max depth of trees, indicating that bagging benefits the generalization ability of complex models.

What happens with the training time as the ensemble size grows? As the ensemble size grows, the training time is growing propotionally, since every single tree has the same algorithm and same amount of data to train.

Task 3: Feature importances in random forest classifiers

[20]:
```
import pandas as pd
import numpy as np

pipeline = make_pipeline(
        DictVectorizer(),
        RandomForestClassifier(n_estimators=200, max_depth=20,␣
 ↪random_state=361433, n_jobs=-1)
        )

pipeline.fit(dicts_for_train, y_train)
```

[20]:
```
Pipeline(steps=[('dictvectorizer', DictVectorizer()),
                ('randomforestclassifier',
                 RandomForestClassifier(max_depth=20, n_estimators=200,
                                        n_jobs=-1, random_state=361433))])
```

[22]:
```
name = pipeline.steps[0][1].feature_names_
importance = pipeline.steps[1][1].feature_importances_

df = list(zip(name, importance))
df.sort(reverse=True, key=lambda pair: pair[1])

for name, importance in df[:5]:
    print(f"Feature {name} has importance {importance}")
```

```
Feature capital-gain has importance 0.142423692940853
Feature age has importance 0.10996594168311488
Feature marital-status=Married-civ-spouse has importance 0.10204825789539673
Feature education-num has importance 0.08924149489715372
Feature hours-per-week has importance 0.07135769170666657
```

Why this result?

In a random forest model, feature importance is determined by information gain, which measures how much useful information a feature provides when splitting the data in decision trees. The higher the information gain, the more effective the feature is at distinguishing between high and low-income individuals, thus ranking higher.

Capital gain has the highest importance because most people do not have capital gains, while high-income individuals tend to have higher capital gains, making it a strong feature for distinguishing between income levels. Age influences career development and salary growth, with middle-aged individuals often at their peak income, so it effectively differentiates income levels. Marital status (married-civ-spouse) usually indicates more stable career and financial situations, associated with higher-income groups. Education level (education-num) affects career choices and salary potential; while income varies across industries, higher education typically correlates with higher-paying jobs. Hours worked per week influences income, with full-time workers generally earning more, though long hours don't always lead to higher income, still providing some distinction.

Alternative ways

1. Permutation improtance
Permutation importance evaluates a feature's impact on model performance by disrupting its relationship with the target variable. The mechanism involves three steps: First, compute the baseline performance (e.g., accuracy or R^2) of the model on the original dataset. Next, randomly shuffle the values of a single feature while keeping other features intact, effectively breaking its association with the target, and recalculate the model's performance on this perturbed dataset. The importance of the feature is quantified as the difference between the baseline performance and the permuted performance. Unlike the default Mean Decrease in Impurity (MDI) method—which measures feature importance based on the reduction in node impurity during tree splits—permutation importance directly reflects a feature's contribution to the model's predictive power. This makes it more reliable, especially for high-cardinality or correlated features, as MDI tends to overestimate the importance of such features. However, permutation importance may underestimate the importance of correlated features, as shuffling one feature leaves related features intact, allowing the model to partially recover the lost information. While computationally more expensive than MDI (due to repeated predictions), it avoids the need for retraining the model and provides a more holistic view of feature relevance.

2. Drop-column improtance
Drop-column importance measures a feature's necessity by completely removing it and retraining the model. The process begins by establishing a baseline performance using the full feature set. The target feature is then excluded, the model is retrained from scratch on the remaining data, and the performance is reevaluated. The importance is derived from the drop in performance between the baseline and the reduced model. Regarded as the "gold standard" for feature importance, this method eliminates biases inherent in MDI (e.g., overvaluing high-cardinality features) and addresses limitations of permutation importance (e.g., incomplete disruption of correlated features). By fully erasing a feature's information, it provides the most accurate assessment of its true contribution to the model. However, this accuracy comes at a prohibitive computational cost, as retraining the model for every feature is resource-intensive, especially with large datasets or complex models. Compared to permutation importance, drop-column importance is less practical for routine use but serves as a critical benchmark for validating results from faster methods. While both strategies may yield different absolute importance values, their rankings of features should align if the model's dependencies are well-structured.