Bangladesh Army International University of Science and Technology

Department Of Computer Science & Engineering

Assignment

Assignment No : 02

CourseTitle :Machine Learning Sessional

Course Code : CSE 412

SubmittedTo

Mousumi Hasan Mukti

Assistant Professor

Dept.of CSE, BAIUST

SubmittedBy

Susmita Paul

ID:0822210205101017

Level/Term:4/1

Date Of Submission :23-05-2025

Introduction:

A Decision Tree is a popular supervised machine learning algorithm used for both classification and regression tasks. It works by splitting the data into subsets based on the value of input features, forming a tree-like structure of decisions.

Each internal node represents a feature (or attribute), each branch represents a decision rule, and each leaf node represents an outcome or class label.

Objective:

The main objective of a Decision Tree algorithm is to create a model that can accurately predict the value of a target variable based on several input features. It achieves this by learning decision rules from the training data and organizing them in a tree-like structure. Each internal node of the tree represents a decision based on the value of an attribute, while each leaf node represents an outcome or class label.

In classification tasks, the objective is to split the dataset into subsets that contain instances of a single class as much as possible. For regression, the goal is to minimize the error between predicted and actual values. The algorithm selects the best feature and condition to split the data at each node using metrics such as Gini impurity, entropy (information gain), or mean squared error.

A secondary, but equally important, objective is to ensure the model is interpretable and easy to understand. Decision trees provide a visual and logical flow of decisions, making them suitable for domains where explainability is critical.

Moreover, the model should generalize well to unseen data. To achieve this, it may need to be regularized by limiting tree depth or applying pruning techniques to avoid overfitting and improve performance.

Theory

A Decision Tree is a flowchart-like structure used for classification and regression tasks. It works by recursively splitting the dataset based on feature values that best separate the target classes or values. Each internal node represents a decision based on a feature, each branch a possible outcome, and each leaf node a final prediction. The algorithm chooses splits using metrics like Gini impurity, entropy (information gain), or mean squared error to maximize data purity at each step. The process continues until a stopping criterion is met, such as maximum depth or pure nodes, resulting in a predictive and interpretable model.

Code description:

```
]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt
   import seaborn as sns
   %matplotlib inline

]: data = pd.read_csv(r'C:\Users\usrer\ML-task\datasets\bank+marketing\bank\bank.csv',sep=';')
```

This code imports libraries (pandas, numpy, matplotlib, seaborn) and loads bank marketing data from a CSV file into a DataFrame.

```
data.info()
```

```
data.head()
```

```
data.describe()
```

The code shows data statistics (describe()), first rows (head()), and structure (info()). The image likely displays a DataFrame summary with counts, means, and distributions of numerical columns.

```
sns.set_style('whitegrid')
plt.rcParams['figure.figsize'] = (10, 6)

plt.figure(figsize=(8, 5))
sns.countplot(x='y', data=data)
plt.title('Distribution of Term Deposit Subscription')
plt.xlabel('Subscribed to Term Deposit?')
plt.ylabel('Count')
plt.show()
```

This code visualizes subscription distribution ('y' column) using a countplot. It sets a whitegrid style, 8x5 figure size, and adds title/labels. The plot shows how many customers subscribed ('yes') vs didn't('no').

```
num_cols = ['age', 'balance', 'duration', 'campaign']


for col in num_cols:
    plt.figure(figsize=(10, 6))
    sns.histplot(data=data, x=col, hue='y', element='step', stat='density', common_norm=False)
    plt.title(f'Distribution of {col} by Subscription')
    plt.xlabel(col)
    plt.ylabel('Density')
    plt.show()
```

This code creates density histplots for numerical columns ('age', 'balance', etc.), comparing distributions between subscribers ('y=yes') and non-subscribers ('y=no'). Each plot shows overlapping density curves with a legend.

```
cat_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'poutcome']

for col in cat_cols:
    plt.figure(figsize=(12, 6))
    sns.countplot(x=col, hue='y', data=data)
    plt.title(f'Subscription by {col}')
    plt.xticks(rotation=45)
    plt.legend(title='Subscribed', loc='upper right')
    plt.show()
```

This code generates countplots for categorical features (job, marital status, etc.), showing subscription rates ('y') for each category. Plots are rotated 45° for readability, with legends indicating subscription status.

```
sns.jointplot(x='age', y='balance', data=data, color='purple', kind='reg')
plt.suptitle('Age vs Balance Relationship', y=1.05)
plt.show()
```

This code creates a joint regression plot comparing 'age' and 'balance' with a purple regression line. It shows the relationship between customers' ages and account balances, with a title positioned above the plot.

```
plt.figure(figsize=(10,6))

data[data['y']=='yes']['duration'].hist(
    alpha=0.5, color='purple', bins=30,
    label='Subscribed (y=yes)'
)

data[data['y']=='no']['duration'].hist(
    alpha=0.5, color='brown', bins=30,
    label='Not Subscribed (y=no)'
)

plt.legend()
plt.xlabel('Call Duration (seconds)')
plt.ylabel('Count')
plt.title('Call Duration Distribution by Subscription Status')
plt.xlim(0, 1500)
plt.show()
```

This code creates overlapping histograms comparing call duration for subscribers (purple) vs non-subscribers (brown). It visualizes how call length relates to subscription outcomes, with a 1500-second limit for clearer comparison.

```
plt.figure(figsize=(11,7))
sns.lmplot(
    x='age',
    y='balance',
    data=data[data['balance'] >= 0],
    hue='y',
    palette='Set1',
    height=6,
    aspect=1.5
)
plt.title('Relationship Between Age and Balance by Subscription Status')
plt.xlabel('Age')
plt.ylabel('Account Balance')
plt.show()
```

This code creates a scatter plot with regression lines showing the relationship between age and account balance, differentiated by subscription status (yes/no). It filters negative balances and uses a 'Set1' color palette.

```
cat_feats = ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'poutcome']

final_data = pd.get_dummies(data, columns=cat_feats, drop_first=True)

final_data.info()
```

This code converts categorical features (job, marital status, etc.) into dummy variables using one-hot encoding (with drop_first=True to avoid multicollinearity), then displays the transformed dataframe's structure using info().

```
final_data['y'] = final_data['y'].map({'yes': 1, 'no': 0})

print("\nNew columns after one-hot encoding:")
print(final_data.columns.tolist())

print(f"\nOriginal shape: {data.shape}")
print(f"Transformed shape: {final_data.shape}")
```

This code converts the target variable 'y' to binary (1 for 'yes', 0 for 'no'), then prints the new column names after one-hot encoding and compares original vs transformed dataframe dimensions.

```
top_jobs = data['job'].value_counts().nlargest(5).index
data['job'] = data['job'].where(data['job'].isin(top_jobs), 'other')

from sklearn.model_selection import train_test_split

X = final_data.drop('y', axis=1)
y = final_data['y']


X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.30,
    random_state=42,
    stratify=y
)
```

This code:

1) Consolidates less frequent jobs into 'other' category
2) Splits data into train/test sets (70/30)
3) Maintains class distribution (stratify=y)
4) Uses random seed 42 for reproducibility
5) Separates features (X) from target (y)

```
print(f"Original dataset shape: {final_data.shape}")
print(f"Training set shape: {X_train.shape}")
print(f"Testing set shape: {X_test.shape}")

print("\nClass distribution in full dataset:")
print(y.value_counts(normalize=True))

print("\nClass distribution in training set:")
print(y_train.value_counts(normalize=True))

print("\nClass distribution in testing set:")
print(y_test.value_counts(normalize=True))
```

This code prints dataset shapes (original, train, test) and class distributions (percentage of 'yes'/'no' subscriptions) for:

1) Full dataset
2) Training set
3) Test set

Verifying stratified split maintained original proportions.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE()
X_res, y_res = smote.fit_resample(X_train, y_train)
```

This code uses SMOTE (Synthetic Minority Oversampling Technique) to balance the training data by generating synthetic samples of the minority class ('y=1'), creating equal class distribution in X_res/y_res.

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn import tree
import matplotlib.pyplot as plt

label_encoders = {}
categorical_cols = ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'poutcome', 'y']

for col in categorical_cols:
    le = LabelEncoder()
    data[col] = le.fit_transform(data[col])
    label_encoders[col] = le

X = data.drop('y', axis=1)
y = data['y']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

dt_classifier = DecisionTreeClassifier(random_state=42)
dt_classifier.fit(X_train, y_train)

y_pred = dt_classifier.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

plt.figure(figsize=(20,10))
tree.plot_tree(dt_classifier, filled=True, feature_names=X.columns, class_names=['No', 'Yes'], max_depth=3)
plt.show()
```

This code:

1) Encodes categorical variables using LabelEncoder

2) Splits data into 70% train, 30% test

3) Trains a Decision Tree classifier

4) Evaluates performance (accuracy, classification report, confusion matrix)

5) Visualizes the tree (first 3 levels)

Key steps in building and evaluating a Decision Tree model.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42),
                           param_grid,
                           cv=5,
                           scoring='accuracy')
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print("Best Parameters:", best_params)

best_dt = grid_search.best_estimator_
```

This code performs hyperparameter tuning for a Decision Tree using GridSearchCV:

1) Tests combinations of max_depth, min_samples_split, and min_samples_leaf

2) Uses 5-fold cross-validation

3) Finds optimal parameters for accuracy

4) Stores best performing model in best_dt

```
importances = best_dt.feature_importances_

feature_importance = pd.DataFrame({'Feature': X.columns, 'Importance': importances})
feature_importance = feature_importance.sort_values('Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature', data=feature_importance)
plt.title('Feature Importance')
plt.show()
```

This code visualizes feature importance from the optimized Decision Tree model:

1) Extracts importance scores for each feature

2) Creates a sorted DataFrame

3) Displays horizontal bar plot

4) Shows most influential predictors for subscription decisions

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix

basic_dtree = DecisionTreeClassifier()
basic_dtree.fit(X_train, y_train)
basic_predictions = basic_dtree.predict(X_test)

print("Basic Decision Tree Performance:")
print(classification_report(y_test, basic_predictions))
print(confusion_matrix(y_test, basic_predictions))


param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy'],
    'class_weight': [None, 'balanced']
```

```
tuned_dtree = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=5, scoring='f1')
tuned_dtree.fit(X_train, y_train)

print("\nBest Parameters:", tuned_dtree.best_params_)
best_dtree = tuned_dtree.best_estimator_


tuned_predictions = best_dtree.predict(X_test)

print("\nTuned Decision Tree Performance:")
print(classification_report(y_test, tuned_predictions))
print(confusion_matrix(y_test, tuned_predictions))


importances = best_dtree.feature_importances_
features = X_train.columns
feature_importance = pd.DataFrame({'feature': features, 'importance': importances})
print("\nFeature Importance:")
print(feature_importance.sort_values('importance', ascending=False))
```

This code compares basic vs optimized Decision Tree models:

1) First trains a basic tree and evaluates performance

2) Then performs grid search with 5-fold CV to find optimal hyperparameters

3) Evaluates tuned model's performance

4) Finally analyzes and displays feature importance

5) Uses classification metrics (precision/recall/F1) for evaluation

Conclusion:

The code demonstrates a comprehensive approach to building and optimizing a Decision Tree classifier for predicting customer subscriptions. It begins with a basic model, then systematically improves it through hyperparameter tuning using GridSearchCV, testing various combinations of tree depth, split criteria, and class weights. The evaluation metrics (classification report and confusion matrix) provide a clear comparison between the basic and tuned models. The feature importance analysis reveals which

customer attributes most influence subscription decisions. This end-to-end process showcases best practices in machine learning: starting with a simple model, methodically optimizing it, rigorously evaluating performance, and interpreting results. The implementation handles both model training and explainability, making it valuable for both prediction and business insight generation.

Referances:

1. Data set from: https://archive.ics.uci.edu/dataset/222/bank+marketing