Lab Sheet-10
Ensemble Methods
Machine Learning
BITS F464
I Semester 2024-25

INTRODUCTION:

In this labsheet, we would be learning about some ensemble methods used in machine learning and just like the previous labs, we would be using one dataset and try to test out various algorithms that you might need to know. Typically these are considered to be compute intensive while training the model and hence we take a dataset which is typically small.

ABOUT THE DATASET:

The **Heart Disease dataset** from the UCI Machine Learning Repository is commonly used for binary classification tasks in healthcare research. It aims to predict the presence of heart disease based on several health and demographic factors. The dataset originally includes data from four different locations, but the Cleveland dataset, containing 303 samples, is most frequently used due to its cleaner data and straightforward labels.

Key Characteristics:

1. **Objective**: Predict whether a patient has heart disease (binary outcome, where `1` indicates presence and `0` absence).
2. **Features**: 13 attributes related to patients' health and demographics:
   - **Age**: Patient's age in years.
   - **Sex**: Patient's sex (1 = male, 0 = female).
   - **cp (Chest Pain Type)**: Chest pain type (0–3), ranging from asymptomatic to severe angina.
   - **trestbps (Resting Blood Pressure)**: Resting blood pressure in mm Hg.
   - **chol (Serum Cholesterol)**: Serum cholesterol in mg/dl.
   - **fbs (Fasting Blood Sugar)**: Indicator for fasting blood sugar > 120 mg/dl (1 = true, 0 = false).
   - **restecg (Resting ECG)**: Results of electrocardiographic tests (values 0, 1, or 2).
   - **thalach (Maximum Heart Rate Achieved)**: Maximum heart rate achieved during the test.
   - **exang (Exercise-Induced Angina)**: Angina induced by exercise (1 = yes, 0 = no).
   - **oldpeak**: ST depression induced by exercise relative to rest.
   - **slope (Slope of the Peak Exercise ST Segment)**: Slope of the ST segment during peak exercise (0–2).
   - **ca (Number of Major Vessels Colored by Fluoroscopy)**: Number of major vessels (0–3).

  - **thal**: Thallium stress test result (values 3, 6, or 7, indicating normal, fixed defect, or reversible defect).
3. **Target Variable**: `target` indicates the presence of heart disease:
  - Values >0 represent different levels of disease severity, but the dataset is often binarized where `1` represents the presence and `0` the absence of disease.

GENERAL OVERVIEW:

Ensemble methods are machine learning techniques that combine predictions from multiple individual models (often called "weak learners") to produce a more accurate and robust prediction than any single model alone. The main idea behind ensemble methods is that by aggregating diverse predictions, they can reduce model variance, increase accuracy, and improve generalisation on unseen data.

We will discuss the following algorithms here:

1. Random Forest
2. Extra Random Trees
3. XGBoost
4. NGBoost
5. AdaBoost
6. LightGBM

We follow the typical ML training pipeline:

1. Data pre-processing
2. Choose a model and their hyperparameters.
3. Choose typical loss functions for the problem and compare
4. Hopefully get to know an intuition of which algorithm to use during which dataset.

```
!pip install ngboost # if you dont find something pip install stuff..

import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.ensemble import RandomForestClassifier,
ExtraTreesClassifier
```

```
from xgboost import XGBClassifier

from ngboost import NGBClassifier

from sklearn.metrics import accuracy_score, roc_auc_score,
precision_score, f1_score

from sklearn.impute import SimpleImputer

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-
disease/processed.cleveland.data"

columns = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg',
'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target']

data = pd.read_csv(url, names=columns)
```

```
data.replace('?', np.nan, inplace=True)  # Replace "?" with NaN

data.dropna(inplace=True)
```

```
data['target'] = data['target'].apply(lambda x: 1 if x > 0 else 0)
```

What do you think Pipeline does?(Hint:Just serializes the operations you want to perform)

```
# Splitting features and target

X = data.drop(columns='target')

y = data['target']

# Separate categorical and numeric columns

numeric_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']

categorical_features = ['sex', 'cp', 'fbs', 'restecg', 'exang',
'slope', 'ca', 'thal']

# Preprocessing pipelines for numerical and categorical data

numeric_transformer = Pipeline(steps=[

    ('imputer', SimpleImputer(strategy='median')),        # Impute
missing values for numeric columns
```

```
    ('scaler', StandardScaler())                          # Standardize numerical features
])


categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')), # Impute missing values for categorical columns
    ('onehot', OneHotEncoder(handle_unknown='ignore'))    # One-hot encode categorical features
])
# Combine preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

You have the data ready now!

Random Forest:

GENERAL IDEA:

Random Forest is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and stability. It trains each tree on a randomly sampled subset of the data, with replacement (bagging), and each tree uses a random subset of features for splits. By averaging or voting on the predictions from all trees, Random Forest reduces overfitting and increases robustness, especially for classification and regression tasks. It is efficient, interpretable, and versatile, handling both numerical and categorical data, and it works well even with imbalanced datasets, making it a popular choice in machine learning.

IMPORTANT HYPERPARAMETERS:
n_estimators: The number of trees in the forest. More trees generally improve performance but increase training time.
max_depth: The maximum depth of each tree. Limiting the depth helps prevent overfitting, especially on smaller datasets.
min_samples_split: The minimum number of samples required to split a node. Higher values prevent overfitting by ensuring nodes have more samples before splitting.

```
rf_model = RandomForestClassifier(
    n_estimators=100,             # Number of trees
    max_depth=10,                 # Maximum depth of each tree
    min_samples_split=5,          # Minimum samples to split a node
    min_samples_leaf=2,           # Minimum samples in a leaf node
    max_features='sqrt',          # Number of features to consider for each split
    bootstrap=True,               # Use bootstrapping
    random_state=42               # Set seed for reproducibility
)

# Create pipeline with preprocessing and model
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('classifier', rf_model)])

# Train the model
```

```
pipeline.fit(X_train, y_train)

# Make predictions
y_pred = pipeline.predict(X_test)
y_pred_proba = pipeline.predict_proba(X_test)[:, 1]

# Evaluate model performance
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

Extra Random Trees:

GENERAL IDEA:
Extra Random Forest, or Extremely Randomized Trees, is similar to Random Forest but introduces more randomness during training. Instead of selecting the best split for each feature, Extra Trees chooses random split points, which can lead to faster training and less overfitting on complex datasets. This added randomness often improves generalization by creating more diverse trees. Extra Trees is effective for large datasets and is commonly used in both classification and regression tasks, where it can offer a performance boost over Random Forest in certain cases by reducing the variance of the model.

IMPORTANT HYPERPARAMETERS:

n_estimators:

- The number of trees in the forest. Higher values often improve stability and performance, but at a cost of increased computation. Common values range from 100 to 500, depending on data size and complexity.

max_depth:

- Maximum depth of each tree. Limiting depth can reduce overfitting, especially on noisy datasets. `None` allows the trees to grow fully until each leaf is pure or has fewer samples than `min_samples_split`.

min_samples_split:

- Minimum number of samples required to split an internal node. Higher values reduce model complexity, helping to prevent overfitting by controlling how deep the tree can grow. Default is `2`, but for noisy data, values between `10` and `20` may work better.

min_samples_leaf:

- Minimum number of samples required to be at a leaf node. Setting a higher value creates a smoother decision boundary and prevents the model from learning idiosyncratic patterns in small regions of the feature space.

max_features:

- Number of features considered for each split. Options include `"auto"` (sqrt of total features for classification), `"sqrt"` (similar to `"auto"`), `"log2"`, or a fraction of total features. Lower values add randomness, which can help with generalization and reduce overfitting.

bootstrap:

- Determines if sampling is done with replacement. By default, Extra Trees does not use bootstrapping (i.e., `bootstrap=False`). Setting `True` enables bootstrapping, similar to Random Forests, which can improve robustness.

random_state:

- Sets the random seed for reproducibility. This ensures that the results are consistent across different runs, particularly important for randomized splits in Extra Trees.

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_auc_score,
precision_score, f1_score

# Define Extra Trees model
extra_trees_model = ExtraTreesClassifier(
    n_estimators=100,         # Number of trees in the forest
    max_depth=None,           # Maximum depth of each tree; None allows
full depth
    min_samples_split=2,      # Minimum samples required to split a node
    min_samples_leaf=1,       # Minimum samples required in a leaf node
    #max_features= ,      # Number of features to consider for best
split
    bootstrap=False,          # No replacement sampling by default
    random_state=42           # Seed for reproducibility
)

# Create pipeline with preprocessing and model
pipeline_etr = Pipeline(steps=[('preprocessor', preprocessor),
('classifier', extra_trees_model)])

# Train the model
pipeline_etr.fit(X_train, y_train)
```

```
# Make predictions
y_pred_etr = pipeline_etr.predict(X_test)
y_pred_proba_etr = pipeline_etr.predict_proba(X_test)[:, 1]

# Evaluate model performance
accuracy_etr = accuracy_score(y_test, y_pred_etr)
auc_roc_etr = roc_auc_score(y_test, y_pred_proba_etr)
precision_etr = precision_score(y_test, y_pred_etr)
f1_etr = f1_score(y_test, y_pred_etr)

print(f"Extra Trees - Accuracy: {accuracy_etr}, AUC-ROC: {auc_roc_etr}, Precision: {precision_etr}, F1 Score: {f1_etr}")
```

XGBoost:

GENERAL IDEA:

XGBoost (Extreme Gradient Boosting) is an advanced boosting algorithm that optimizes both computational efficiency and predictive accuracy. It builds trees sequentially, with each new tree learning from the errors of previous trees, and it employs regularization to reduce overfitting. Known for its speed and scalability, XGBoost uses techniques like tree pruning, parallelization, and hardware optimization to handle large datasets effectively. XGBoost has become popular for structured data tasks, especially in machine learning competitions, due to its high accuracy, flexibility, and the ability to handle missing values natively.

IMPORTANT HYPERPARAMETERS:

1. **n_estimators**: Number of trees (or rounds of boosting). Higher values can lead to better performance but may increase computation.
2. **learning_rate**: Step size shrinkage used to prevent overfitting. Smaller values are often better but require more trees.
3. **max_depth**: Maximum depth of each tree. Higher values allow more complex trees but can lead to overfitting.
4. **subsample**: Fraction of samples used per tree. Lower values can help reduce overfitting.

```
xgb_model = XGBClassifier(

    n_estimators=100,                # Number of boosting rounds

    learning_rate=0.1,               # Step size shrinkage

    max_depth=6,                     # Maximum tree depth
```

```python
    subsample=0.8,                  # Fraction of samples used per tree
    colsample_bytree=0.8,           # Fraction of features used per tree
    gamma=1,                        # Minimum loss reduction to make a split
    use_label_encoder=False,        # Disable the use of the label encoder for warnings
    eval_metric='logloss',          # Evaluation metric for binary classification
    random_state=42                 # Set seed for reproducibility
)
# Create pipeline with preprocessing and model
pipeline = Pipeline(steps=[('preprocessor', preprocessor), ('classifier', xgb_model)])
# Train the model
pipeline.fit(X_train, y_train)


# Make predictions
y_pred = pipeline.predict(X_test)
y_pred_proba = pipeline.predict_proba(X_test)[:, 1]
# Evaluate model performance
accuracy = accuracy_score(y_test, y_pred)
auc_roc = roc_auc_score(y_test, y_pred_proba)
precision = precision_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

NGBoost:

GENERAL IDEA:

NGBoost (Natural Gradient Boosting) is a gradient boosting framework that incorporates probabilistic predictions, offering not only point estimates but also uncertainties. Unlike other boosting algorithms, NGBoost uses natural gradients to improve stability and convergence, and it can be applied to a range of probability distributions, making it versatile for classification, regression, and probabilistic forecasting tasks. NGBoost is particularly valuable in scenarios where understanding predictive confidence or uncertainty is essential, such as in healthcare and financial modeling. It supports custom distributions and scoring functions, making it highly adaptable to various complex applications.

IMPORTANT HYPERPARAMETERS:

n_estimators: Number of boosting iterations or trees. Increasing this often improves performance but increases computation.

learning_rate: Controls the step size for each boosting iteration. Smaller values lead to slower convergence but can improve accuracy.

natural_gradient: Boolean to enable natural gradients, which can improve stability and performance.

minibatch_frac: Fraction of samples to use per tree, controlling subsampling to reduce overfitting.

```
from ngboost.distns import Bernoulli  # Distribution for binary classification

from sklearn.tree import DecisionTreeRegressor
from ngboost.scores import LogScore  # Scoring function


ngb_model = NGBClassifier(

    Dist=Bernoulli,                      # Distribution for binary
classification

    Score=LogScore,                       # Scoring function

    Base=DecisionTreeRegressor(max_depth=3),  # Base learner

    n_estimators=100,                     # Number of boosting
iterations

    learning_rate=0.01,                   # Learning rate

    minibatch_frac=0.8,                   # Fraction of samples per
boosting round

    natural_gradient=True,                # Use natural gradients

    random_state=42                       # Set seed for reproducibility

)


# Create pipeline with preprocessing and model

pipeline = Pipeline(steps=[('preprocessor', preprocessor),
('classifier', ngb_model)])


# Train the model

pipeline.fit(X_train, y_train)


# Make predictions

y_pred = pipeline.predict(X_test)

y_pred_proba = pipeline.predict_proba(X_test)[:, 1]
```

```
# Evaluate model performance

accuracy = accuracy_score(y_test, y_pred)

auc_roc = roc_auc_score(y_test, y_pred_proba)

precision = precision_score(y_test, y_pred)

f1 = f1_score(y_test, y_pred)
```

Adaboost:

GENERAL IDEA:

AdaBoost (Adaptive Boosting) is a boosting algorithm that builds an ensemble of weak learners, often simple decision stumps, to create a strong classifier. Each learner is trained sequentially, with greater emphasis on misclassified instances from previous rounds. AdaBoost combines the outputs of all learners by assigning weights based on accuracy, effectively focusing on difficult cases. Although it is prone to overfitting on noisy data, AdaBoost is simple, interpretable, and works well with binary and multiclass classification tasks. It is widely used for improving accuracy in models that may not perform well independently.

IMPORTANT HYPERPARAMETERS:

n_estimators:

- Number of boosting rounds, or the total number of weak learners (typically decision stumps).
- Increasing this can improve performance by reducing bias but may lead to overfitting if too high.

learning_rate:

- Controls the contribution of each weak learner to the final model. Lower values often yield better results, but you may need more iterations to converge.
- Smaller values can improve accuracy and stability by reducing the model’s sensitivity to each new weak learner.

base_estimator:

- The model used as the weak learner. The most common choice is a shallow `DecisionTreeClassifier`.
- You can adjust the complexity of the base estimator (e.g., tree depth) to manage bias-variance trade-offs.

random_state:

- Sets the seed for reproducibility. Ensures that results are consistent across runs by fixing the randomness in sampling and model initialization.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_auc_score,
precision_score, f1_score

# Define AdaBoost model with a decision tree as the base estimator
ada_model = AdaBoostClassifier(
    #base_estimator=DecisionTreeClassifier(max_depth=3),  # Base
learner
    n_estimators=100,                                    # Number of
boosting iterations
    learning_rate=0.01,                                  # Learning
rate
    random_state=42                                      # Set seed for
reproducibility
)

# Create pipeline with preprocessing and model
pipeline_ada = Pipeline(steps=[('preprocessor', preprocessor),
('classifier', ada_model)])

# Train the model
pipeline_ada.fit(X_train, y_train)

# Make predictions
y_pred_ada = pipeline_ada.predict(X_test)
y_pred_proba_ada = pipeline_ada.predict_proba(X_test)[:, 1]

# Evaluate model performance
accuracy_ada = accuracy_score(y_test, y_pred_ada)
auc_roc_ada = roc_auc_score(y_test, y_pred_proba_ada)
precision_ada = precision_score(y_test, y_pred_ada)
```

```
f1_ada = f1_score(y_test, y_pred_ada)

print(f"AdaBoost - Accuracy: {accuracy_ada}, AUC-ROC: {auc_roc_ada}, Precision: {precision_ada}, F1 Score: {f1_ada}")
```

LightGBM:

GENERAL IDEA:

LightGBM (Light Gradient Boosting Machine) is a highly efficient gradient boosting framework optimised for speed and scalability. It uses a leaf-wise tree growth strategy, which tends to improve accuracy by reducing loss more effectively than traditional level-wise methods. LightGBM supports large datasets and complex features due to its efficient handling of categorical features and sparse data. It is well-suited for high-dimensional data, offering fast training and low memory usage, making it an ideal choice in scenarios where both high accuracy and computational efficiency are crucial, such as in financial and e-commerce applications.

IMPORTANT HYPERPARAMETERS:

n_estimators:

- Number of trees to grow, or boosting rounds. Higher values can improve accuracy but also increase computation time and risk overfitting.

learning_rate:

- Controls the shrinkage of each boosting step. Lower values can increase accuracy but may require more trees to reach convergence.

- Common practice is to find a good trade-off between n_estimators and learning_rate.

max_depth:

- Maximum depth of each tree, which controls the complexity. Lower values can help reduce overfitting, while higher values allow for more intricate decision boundaries.

min_data_in_leaf:

- Minimum number of samples required in a leaf node, which serves as a regularization term to prevent overly specific splits.
- Helps prevent overfitting on small datasets by setting a lower bound on leaf samples.

subsample:

- Fraction of the data sampled per boosting iteration (also called bagging_fraction in LightGBM). This introduces randomness and can reduce overfitting.
- Typical values are around 0.8, meaning 80% of the data is used in each iteration.

colsample_bytree:

- Fraction of features sampled per tree, which adds randomness and helps with generalization.
- Values close to 1 use nearly all features, while smaller values (e.g., 0.8) can help reduce overfitting.

reg_alpha and reg_lambda:

- L1 and L2 regularization terms, respectively, which penalize large coefficients and prevent overfitting.
- Adjust these to control the complexity of the model, especially when dealing with highly complex data.

random_state:

- Seed to ensure reproducibility of results. Useful for consistent results across multiple runs of the model.

```
from lightgbm import LGBMClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, roc_auc_score,
precision_score, f1_score

# Define LightGBM model
lgbm_model = LGBMClassifier(
    n_estimators=100,              # Number of boosting iterations
    learning_rate=0.01,            # Learning rate
    subsample=0.8,                 # Fraction of samples per boosting
round
    colsample_bytree=0.8,          # Fraction of features per boosting
round
    random_state=42                # Set seed for reproducibility
)

# Create pipeline with preprocessing and model
pipeline_lgbm = Pipeline(steps=[('preprocessor', preprocessor),
('classifier', lgbm_model)])

# Train the model
pipeline_lgbm.fit(X_train, y_train)

# Make predictions
y_pred_lgbm = pipeline_lgbm.predict(X_test)
y_pred_proba_lgbm = pipeline_lgbm.predict_proba(X_test)[:, 1]

# Evaluate model performance
accuracy_lgbm = accuracy_score(y_test, y_pred_lgbm)
auc_roc_lgbm = roc_auc_score(y_test, y_pred_proba_lgbm)
precision_lgbm = precision_score(y_test, y_pred_lgbm)
f1_lgbm = f1_score(y_test, y_pred_lgbm)

print(f"LightGBM - Accuracy: {accuracy_lgbm}, AUC-ROC: {auc_roc_lgbm},
Precision: {precision_lgbm}, F1 Score: {f1_lgbm}")
```