Lab Sheet-8
Evaluating Classification Algorithms: A Model Selection Approach

---

INTRODUCTION:

In this lab, we will explore various classification algorithms applied to the Adult Income dataset to evaluate their performance and identify the best-fitting model. Through experimentation with different techniques, we aim to understand how well each model generalizes and determine the optimal approach for this dataset. This process mirrors real-world model selection, where testing and tuning guide us to the most effective solution.

ABOUT THE DATASET:

The **Adult Income Dataset**, available on Kaggle, is a well-known dataset used for **binary classification** tasks, specifically to predict whether a person's income exceeds $50,000 per year based on various demographic attributes.

Key Features:

The dataset contains **14 attributes** (also known as features) with a mix of continuous and categorical data types. The goal is to classify whether an individual's annual income is **above** or **below $50,000**, which is represented by the target variable (income). The attributes are as follows:

1. **Age**: Continuous. Represents the individual's age.
2. **Workclass**: Categorical. Defines the type of employment (e.g., Private, Self-emp, Government).
3. **Fnlwgt**: Continuous. A census-specific feature (final weight), though less important for analysis.
4. **Education**: Categorical. Reflects the highest education level attained (e.g., Bachelors, HS-grad).
5. **Education-Num**: Continuous. Numeric encoding of education level.
6. **Marital Status**: Categorical. Marital status (e.g., Married, Single, Divorced).
7. **Occupation**: Categorical. The type of work or occupation (e.g., Tech-support, Sales, Exec-managerial).
8. **Relationship**: Categorical. Relationship to the household (e.g., Husband, Wife, Unmarried).
9. **Race**: Categorical. Race (e.g., White, Asian-Pac-Islander, Black).
10. **Sex**: Categorical. Gender (Male or Female).
11. **Capital Gain**: Continuous. Investment income from capital gains.
12. **Capital Loss**: Continuous. Investment losses from capital.
13. **Hours per Week**: Continuous. Number of hours worked per week.
14. **Native Country**: Categorical. Country of origin (e.g., United States, Mexico, Canada).

Target Variable: **Income**-Binary classification label indicating whether an individual's income is **<=50K or >50K**.

Dataset Statistics:

- **Training set**: 32,561 instances.
- **Test set**: 16,281 instances.

GETTING AND PRE-PROCESSING OF DATA:

We obtain the data from an online repository here, however you could use the shared train and test dataset in order to obtain the data.

```
url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data"

columns = ["age", "workclass", "fnlwgt", "education", "education-num",
"marital-status",

           "occupation", "relationship", "race", "sex", "capital-gain",
"capital-loss",

           "hours-per-week", "native-country", "income"]

data = pd.read_csv(url, names=columns, sep=',\s+', na_values='?')
```

1. **NULL Values:**

   Simple Imputer function simply replaces null values present in the dataset with the most frequent values in that column.

   ```
   imputer = SimpleImputer(strategy='most_frequent')

   data_imputed = pd.DataFrame(imputer.fit_transform(data),
   columns=columns)
   ```

2. **Encoding Categorical Variables:**

   Convert Categorical Variables into Numbers using the Label Encoder method.

   ```
   label_encoders = {}

   for column in data_imputed.select_dtypes(include='object').columns:
   ```

```
    le = LabelEncoder()

    data_imputed[column] = le.fit_transform(data_imputed[column])

    label_encoders[column] = le
```

3. **Seperate the features and targets:**

```
X = data_imputed.drop('income', axis=1)

y = data_imputed['income']
```

4. **Normalization of Data:**

```
scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

5. **Mean, Variance and Standard Deviation**

```
    import numpy as np
```

```
mean = np.mean(data)

variance = np.var(data)

std_dev = np.std(data
```

EXERCISE: Try to plot the correlation matrix for the dataset(features+target) and identify which feature is correlated the most with the target. This helps us have a better idea of which feature is the most important.

Feel free to refer to the documentation of scikit-learn to learn more about these algorithms. You can try to tune[vary] the important hyperparameters for each algorithm in order to observe changes in accuracy(if any).(An example is hidden somewhere in the doc)

LOGISTIC REGRESSION:

class sklearn.linear_model.LogisticRegression(*penalty='l2'*, ***, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='lbfgs'*, *max_iter=100*, *multi_class='deprecated'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*)

IMPORTANT HYPERPARAMETERS: penalty and C

```
from sklearn.linear_model import LogisticRegression



# Logistic Regression model with custom hyperparameters

model = LogisticRegression(

    penalty='l2',         # L2 regularization

    C=0.1,                # Stronger regularization (small C)

    solver='saga',        # Solver supporting elasticnet

    max_iter=500,         # Increase max iterations

    tol=1e-5,             # Smaller tolerance for convergence

    class_weight='balanced',  # Handle class imbalance

    multi_class='multinomial' # Multinomial logistic regression

)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

from sklearn.metrics import accuracy_score

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

MULTI LAYER PERCEPTRON:

class sklearn.linear_model.Perceptron(*, *penalty=None*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=None*, *random_state=0*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*)

IMPORTANT HYPERPARAMETERS:**activation, alpha, learning rate**

```
from sklearn.neural_network import MLPClassifier

# Define the MLP Classifier with hyperparameters

mlp = MLPClassifier(hidden_layer_sizes=(100, 50),   # Two hidden layers
with 100 and 50 neurons

    activation='relu',              # Activation function

    solver='adam',                  # Optimizer

    alpha=0.0001,                   # L2 regularization

    learning_rate='adaptive',       # Adaptive learning rate

    learning_rate_init=0.001,       # Initial learning rate

    max_iter=500,                   # Increased number of iterations

    early_stopping=True,            # Use early stopping

    validation_fraction=0.1         # 10% validation split

)

mlp.fit(X_train, y_train)

y_pred = mlp.predict(X_test)

# Print accuracy

from sklearn.metrics import accuracy_score

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

NAIVE BAYES CLASSIFIER:

class sklearn.naive_bayes.GaussianNB(*, *priors=None*, *var_smoothing=1e-09*)

IMPORTANT HYPERPARAMETERS:var_smoothing

```
from sklearn.naive_bayes import GaussianNB



# Initialize the model with custom var_smoothing

gnb_model = GaussianNB(var_smoothing=1e-8)  # Slightly less smoothing



gnb_model.fit(X_train, y_train)

y_pred = gnb_model.predict(X_test)



from sklearn.metrics import accuracy_score

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

DECISION TREES:

class sklearn.tree.DecisionTreeClassifier(*, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *class_weight=None*, *ccp_alpha=0.0*, *monotonic_cst=None*)

IMPORTANT HYPERPARAMETERS: max_depth, min_samples_split

```
from sklearn.tree import DecisionTreeClassifier

# Initialize the Decision Tree model with custom hyperparameters

tree_model = DecisionTreeClassifier(

    max_depth=5,              # Limit depth of tree

    min_samples_split=10      # Require at least 10 samples to split
```

```
)

# Train the model

tree_model.fit(X_train, y_train)

# Make predictions

y_pred = tree_model.predict(X_test)

from sklearn.metrics import accuracy_score

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

LINEAR SVM:

class sklearn.svm.LinearSVC(*penalty='l2'*, *loss='squared_hinge'*, ***, *dual='auto'*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

IMPORTANT HYPERPARAMETERS: C(Regularization) for linear SVM

```
from sklearn.svm import LinearSVC

# Initialize the SVM model with custom hyperparameters

svm_model = LinearSVC(

    C=1.0,                  # Regularization parameter

)

# Train the model

svm_model.fit(X_train, y_train)

# Make predictions

y_pred = svm_model.predict(X_test)

from sklearn.metrics import accuracy_score

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

KERNEL SVM:

class sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ties=False, random_state=None)

IMPORTANT HYPERPARAMETERS: C(Regularization), kernel type, degree

```
from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

# Initialize the kernel SVM model with custom hyperparameters

svm_model = SVC(

    C=1.0,                # Regularization parameter

    kernel='rbf',         # Kernel type: 'linear', 'poly', 'rbf', 'sigmoid'

    gamma='scale'         # Kernel coefficient ('scale' is the default for
RBF)

)

svm_model.fit(X_train, y_train)

y_pred = svm_model.predict(X_test)

# Evaluate the accuracy

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

```
Accuracy: 0.8526
```

Linear Discriminant Analysis:

Linear Discriminant Analysis (LDA) is a popular supervised classification algorithm that projects the data onto a lower-dimensional space while maximizing the class separability. It assumes that the different classes have identical covariances and are normally distributed. LDA is widely used for dimensionality reduction and classification tasks when the classes are linearly separable.

class sklearn.discriminant_analysis.LinearDiscriminantAnalysis(*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*, *covariance_estimator=None*)

IMPORTANT HYPERPARAMETERS: solver, n_components(Setting n_components controls how many linear discriminants are retained. The number of components must be less than or equal to min(n_classes - 1, n_features). If None, LDA will keep as many components as possible.)

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.metrics import accuracy_score

# Initialize the LDA model with default hyperparameters

lda_model = LinearDiscriminantAnalysis()

lda_model.fit(X_train, y_train)

y_pred = lda_model.predict(X_test)

# Evaluate the accuracy

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Quadratic Discriminant Analysis:

Quadratic Discriminant Analysis (QDA) is a classification algorithm that models each class using a multivariate normal distribution with a distinct covariance matrix for each class. This makes QDA suitable for situations where class distributions have different shapes (i.e., different covariance structures), as opposed to Linear Discriminant Analysis (LDA), which assumes identical covariance matrices for all classes.

IMPORTANT HYPERPARAMETERS: priors(If None, the model will compute the priors based on the class distribution in the training set (i.e., proportional to the number of samples in each class)), reg_param(regularization)

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

from sklearn.metrics import accuracy_score

# Initialize the QDA model with default hyperparameters

qda_model = QuadraticDiscriminantAnalysis()

# Train the model

qda_model.fit(X_train, y_train)

# Make predictions

y_pred = qda_model.predict(X_test)

# Evaluate the accuracy

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

Passive Aggressive Classifier:

The Passive-Aggressive Classifier is an online learning algorithm that is particularly useful for large-scale and streaming data. It is designed for binary and multi-class classification tasks, and it operates by adjusting itself only when it makes incorrect predictions (passive when correct, aggressive when wrong). The classifier is well-suited for scenarios where fast updates and low memory usage are important.

IMPORTANT HYPERPARAMETERS: C, max_iter, tol

As an example, the hyperparameter tuning has been shown in the below code for the above mentioned hyperparameters. You can use this as reference for other algorithms.

```
from sklearn.linear_model import PassiveAggressiveClassifier

from sklearn.metrics import accuracy_score

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import PassiveAggressiveClassifier

from sklearn.metrics import accuracy_score

from sklearn.model_selection import GridSearchCV
```

```
# Feature scaling

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)

# Hyperparameter tuning using GridSearchCV

param_grid = {

    'C': [0.01, 0.1, 0.5, 1.0, 2.0],

    'max_iter': [500, 1000, 2000],

    'tol': [1e-4, 1e-3, 1e-2]

}

# Grid search

grid_search = GridSearchCV(PassiveAggressiveClassifier(random_state=42),
param_grid, cv=5, scoring='accuracy')

grid_search.fit(X_train_scaled, y_train)



# Get the best model

best_pac_model = grid_search.best_estimator_



# Make predictions

y_pred = best_pac_model.predict(X_test_scaled)



# Evaluate the accuracy

print(f"Best Accuracy: {accuracy_score(y_test, y_pred):.4f}")

print(f"Best Parameters: {grid_search.best_params_}")
```

```
Best Accuracy: 0.8081
```

```
Best Parameters: {'C': 0.01, 'max_iter': 500, 'tol': 0.01}
```

Ridge Classifier

The RidgeClassifier in scikit-learn is a linear classifier that applies **L2 regularization** (Ridge) to prevent overfitting. It's especially useful for datasets with highly correlated features. It works similarly to logistic regression but uses the squared loss function (ridge regression) for classification problems.

IMPORTANT HYPERPARAMETERS: alpha(regularization parameter), solver('auto': Chooses the solver automatically.'svd': Uses Singular Value Decomposition; works well when n_samples > n_features.)

```
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)

ridge_model = RidgeClassifier(

    alpha=1.0,           # Regularization strength (default is 1.0)

    tol=1e-3,            # Tolerance for stopping criteria

    random_state=42      # Random state for reproducibility

)

# Train the model

ridge_model.fit(X_train_scaled, y_train)

# Make predictions

y_pred = ridge_model.predict(X_test_scaled)

# Evaluate the accuracy

print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

TASK: Which model do you think performed the best? Check the precision, recall and accuracy for each model and decide which one is good for the given dataset.