Assignment 4: Pipelines and Hyperparameter Tuning

- **Full Name** = Dominic Choi
- **UCID** = 30109955 ***

In this assignment, you will be putting together everything you have learned so far. You will need to do all the appropriate preprocessing, test different supervised learning models, and evaluate the results. More details for each step can be found below. You will also be asked to describe the process by which you came up with the code. More details can be found below. Please cite any websites or AI tools that you used to help you with this assignment.

For this assignment, in addition to your .ipynb file, please also attach a PDF file. To generate this PDF file, you can use the print function (located under the "File" within Jupyter Notebook). Name this file ENGG444_Assignment##__yourUCID.pdf (this name is similar to your main .ipynb file). We will evaluate your assignment based on the two files and you need to provide both.

Question	Point(s)
1. Preprocessing Tasks	
1.1	2
1.2	2
1.3	4
2. Pipeline and Modeling	
2.1	3
2.2	6
2.3	5
2.4	3
3. Bonus Question	**2**
Total	**25**

0. Dataset

This data is a subset of the **Heart Disease Dataset**, which contains information about patients with possible coronary artery disease. The data has **14 attributes** and **294 instances**. The attributes include demographic, clinical, and laboratory features, such as age, sex, chest pain type, blood pressure, cholesterol, and electrocardiogram results. The last attribute is the **diagnosis of heart disease**, which is a categorical variable with values from 0 (no presence) to 4 (high presence). The data can be used for **classification** tasks, such as predicting the presence or absence of heart disease based on the other attributes.

```
import pandas as pd

# Define the data source link
_link =
```

```
'https://archive.ics.uci.edu/ml/machine-learning-databases/heart-
disease/processed.hungarian.data'

# Read the CSV file into a Pandas DataFrame, considering '?' as
missing values
df = pd.read_csv(_link, na_values='?',
                 names=['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs',
                        'restecg', 'thalach', 'exang', 'oldpeak',
'slope',
                        'ca', 'thal', 'num'])

# Display the DataFrame
display(df)
```

```
     age  sex  cp  trestbps   chol  fbs  restecg  thalach  exang
oldpeak  \
0     28    1   2     130.0  132.0  0.0      2.0    185.0    0.0
0.0
1     29    1   2     120.0  243.0  0.0      0.0    160.0    0.0
0.0
2     29    1   2     140.0    NaN  0.0      0.0    170.0    0.0
0.0
3     30    0   1     170.0  237.0  0.0      1.0    170.0    0.0
0.0
4     31    0   2     100.0  219.0  0.0      1.0    150.0    0.0
0.0
..   ...  ...  ..       ...    ...  ...      ...      ...    ...
...
289   52    1   4     160.0  331.0  0.0      0.0     94.0    1.0
2.5
290   54    0   3     130.0  294.0  0.0      1.0    100.0    1.0
0.0
291   56    1   4     155.0  342.0  1.0      0.0    150.0    1.0
3.0
292   58    0   2     180.0  393.0  0.0      0.0    110.0    1.0
1.0
293   65    1   4     130.0  275.0  0.0      1.0    115.0    1.0
1.0

     slope  ca  thal  num
0      NaN NaN   NaN    0
1      NaN NaN   NaN    0
2      NaN NaN   NaN    0
3      NaN NaN   6.0    0
4      NaN NaN   NaN    0
..     ...  ..   ...  ...
289    NaN NaN   NaN    1
290    2.0 NaN   NaN    1
291    2.0 NaN   NaN    1
292    2.0 NaN   7.0    1
```

```
293    2.0 NaN   NaN    1

[294 rows x 14 columns]
```

1. Preprocessing Tasks

- **1.1** Find out which columns have more than 60% of their values missing and drop them from the data frame. Explain why this is a reasonable way to handle these columns. **(2 Points)**
- **1.2** For the remaining columns that have some missing values, choose an appropriate imputation method to fill them in. You can use the `SimpleImputer` class from `sklearn.impute` or any other method you prefer. Explain why you chose this method and how it affects the data. **(2 Points)**
- **1.3** Assign the `num` column to the variable `y` and the rest of the columns to the variable `X`. The `num` column indicates the presence or absence of heart disease based on the angiographic disease status of the patients. Create a `ColumnTransformer` object that applies different preprocessing steps to different subsets of features. Use `StandardScaler` for the numerical features, `OneHotEncoder` for the categorical features, and `passthrough` for the binary features. List the names of the features that belong to each group and explain why they need different transformations. You will use this `ColumnTransformer` in a pipeline in the next question. **(4 Points)**

Answer:

- **1.1** ..... This is a reasonable way to handle these columns, because columns with too many of missing values lack sufficient data for meaningful analysis or modeling. Keeping such columns may introduce noise and bias into the analysis.

```
# 1.1
# Calculate the percentage of missing values for each column
missing_percentage = df.isnull().sum() / len(df) * 100
print(missing_percentage)

# Get the column names with more than 60% missing values
columns_to_drop = missing_percentage[missing_percentage > 60].index

# Drop the columns from the DataFrame
df.drop(columns_to_drop, axis=1, inplace=True)

# Display the DataFrame
display(df)
```

```
age          0.000000
sex          0.000000
cp           0.000000
trestbps     0.340136
```

```
chol         7.823129
fbs          2.721088
restecg      0.340136
thalach      0.340136
exang        0.340136
oldpeak      0.000000
slope       64.625850
ca          98.979592
thal        90.476190
num          0.000000
dtype: float64
```

```
     age  sex  cp  trestbps   chol  fbs  restecg  thalach  exang
oldpeak  num
0     28    1   2     130.0  132.0  0.0      2.0    185.0    0.0
0.0    0
1     29    1   2     120.0  243.0  0.0      0.0    160.0    0.0
0.0    0
2     29    1   2     140.0    NaN  0.0      0.0    170.0    0.0
0.0    0
3     30    0   1     170.0  237.0  0.0      1.0    170.0    0.0
0.0    0
4     31    0   2     100.0  219.0  0.0      1.0    150.0    0.0
0.0    0
..   ...  ...  ..       ...    ...  ...      ...      ...    ...
...  ...
289   52    1   4     160.0  331.0  0.0      0.0     94.0    1.0
2.5    1
290   54    0   3     130.0  294.0  0.0      1.0    100.0    1.0
0.0    1
291   56    1   4     155.0  342.0  1.0      0.0    150.0    1.0
3.0    1
292   58    0   2     180.0  393.0  0.0      0.0    110.0    1.0
1.0    1
293   65    1   4     130.0  275.0  0.0      1.0    115.0    1.0
1.0    1

[294 rows x 11 columns]
```

Answer:

- **1.2** .....
  - I chose mean imputation, because it is straightforward to implement and often works well for numerical data. It's a good strat for handling missing values, especially the missingness is MCAR
  - Mean imputation can affect the data distribution, especially if there are outliers or skewed distributions in the original data. It can be suitable for numerical features with a relatively normal distribution and where the missing values are assumed to be MAR

```python
# 1.2
from sklearn.impute import SimpleImputer
# Create an instance of SimpleImputer with the mean strategy
imputer = SimpleImputer(strategy='mean')

# Fill in the missing values in the DataFrame
df_filled = pd.DataFrame(imputer.fit_transform(df),
columns=df.columns)

# Display the DataFrame
display(df_filled)
df_filled.isna().sum()
```

```
      age  sex   cp  trestbps        chol  fbs  restecg  thalach
exang  \
0    28.0  1.0  2.0     130.0  132.000000  0.0      2.0    185.0
0.0
1    29.0  1.0  2.0     120.0  243.000000  0.0      0.0    160.0
0.0
2    29.0  1.0  2.0     140.0  250.848708  0.0      0.0    170.0
0.0
3    30.0  0.0  1.0     170.0  237.000000  0.0      1.0    170.0
0.0
4    31.0  0.0  2.0     100.0  219.000000  0.0      1.0    150.0
0.0
..    ...  ...  ...       ...         ...  ...      ...      ...    ..
.
289  52.0  1.0  4.0     160.0  331.000000  0.0      0.0     94.0
1.0
290  54.0  0.0  3.0     130.0  294.000000  0.0      1.0    100.0
1.0
291  56.0  1.0  4.0     155.0  342.000000  1.0      0.0    150.0
1.0
292  58.0  0.0  2.0     180.0  393.000000  0.0      0.0    110.0
1.0
293  65.0  1.0  4.0     130.0  275.000000  0.0      1.0    115.0
1.0

     oldpeak  num
0        0.0  0.0
1        0.0  0.0
2        0.0  0.0
3        0.0  0.0
4        0.0  0.0
..       ...  ...
289      2.5  1.0
290      0.0  1.0
291      3.0  1.0
292      1.0  1.0
293      1.0  1.0
```

```
[294 rows x 11 columns]

age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
num         0
dtype: int64
```

Answer:

- **1.3** .....
  - **Numerical Features:** These features represent continuous variables. They require scaling with StandardScaler to ensure they are on the same scale
    - age
    - trestbps
    - chol
    - thalach
  - **Categorical Features:** Categorical features represent discrete variables. OneHotEncoder is used to convert these categorical features into binary vectors, making each category a separate binary feature.
    - cp
    - restecg
    - oldpeak
  - **Binary Features:** Since it's already binary, no further transformation is needed, and we'll keep it as-is using 'passthrough'.
    - sex
    - fbs
    - exang
    - num

```python
# 1.3
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split

# assing num to y
y = df_filled['num']
print("y: ")
print(y)
```

```
# assign the rest to X
X = df_filled.drop(columns=['num'])
print("X: ")
print(X)

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

# Identify numerical, categorical, and binary features
numerical_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
categorical_features = ['cp', 'restecg']
binary_features = ['sex', 'fbs', 'exang']

transformers= [
        ('numerical', StandardScaler(), numerical_features),
        ('categorical', OneHotEncoder(sparse_output=False,
handle_unknown='ignore'), categorical_features),
        ('binary', 'passthrough', binary_features)
    ]

# Create ColumnTransformer
preprocessor = ColumnTransformer(transformers=transformers)

y:
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
      ...
289    1.0
290    1.0
291    1.0
292    1.0
293    1.0
Name: num, Length: 294, dtype: float64
X:
      age  sex   cp  trestbps        chol  fbs  restecg  thalach
exang  \
0    28.0  1.0  2.0     130.0  132.000000  0.0      2.0    185.0
0.0
1    29.0  1.0  2.0     120.0  243.000000  0.0      0.0    160.0
0.0
2    29.0  1.0  2.0     140.0  250.848708  0.0      0.0    170.0
0.0
3    30.0  0.0  1.0     170.0  237.000000  0.0      1.0    170.0
0.0
4    31.0  0.0  2.0     100.0  219.000000  0.0      1.0    150.0
0.0
..    ...  ...  ...       ...         ...  ...      ...      ...    ..
```

```
.
289  52.0  1.0  4.0     160.0  331.000000  0.0      0.0     94.0
1.0
290  54.0  0.0  3.0     130.0  294.000000  0.0      1.0    100.0
1.0
291  56.0  1.0  4.0     155.0  342.000000  1.0      0.0    150.0
1.0
292  58.0  0.0  2.0     180.0  393.000000  0.0      0.0    110.0
1.0
293  65.0  1.0  4.0     130.0  275.000000  0.0      1.0    115.0
1.0

     oldpeak
0        0.0
1        0.0
2        0.0
3        0.0
4        0.0
..       ...
289      2.5
290      0.0
291      3.0
292      1.0
293      1.0

[294 rows x 10 columns]
```

2. Pipeline and Modeling

- **2.1** Create **three** `Pipeline` objects that take the column transformer from the previous question as the first step and add one or more models as the subsequent steps. You can use any models from `sklearn` or other libraries that are suitable for binary classification. For each pipeline, explain **why** you selected the model(s) and what are their **strengths and weaknesses** for this data set. **(3 Points)**

- **2.2** Use `GridSearchCV` to perform a grid search over the hyperparameters of each pipeline and find the best combination that maximizes the cross-validation score. Report the best parameters and the best score for each pipeline. Then, update the hyperparameters of each pipeline using the best parameters from the grid search. **(6 Points)**

- **2.3** Form a stacking classifier that uses the three pipelines from the previous question as the base estimators and a meta-model as the `final_estimator`. You can choose any model for the meta-model that is suitable for binary classification. Explain **why** you chose the meta-model and how it combines the predictions of the base estimators. Then, use `StratifiedKFold` to perform a cross-validation on the stacking classifier and present the accuracy scores and F1 scores for each fold.

Report the mean and the standard deviation of each score in the format of `mean ± std`. For example, `0.85 ± 0.05`. **(5 Points)**

- **2.4**: Interpret the final results of the stacking classifier and compare its performance with the individual models. Explain how stacking classifier has improved or deteriorated the prediction accuracy and F1 score, and what are the possible reasons for that. **(3 Points)**

Answer:

- **2.1** .....
  - **Pipeline 1: Logistic Regression:** I chose it, because it is simple, interpretable, and efficient, making it a good baseline model for classification.
    - Pros:
      - Simple and interpretable.
      - Efficient to train, especially with large datasets.
      - Provides probabilities for predictions.
    - Cons:
      - Assumes a linear relationship between features, and may not capture complexity
      - Sensitive to outliers
  - **Pipeline 2: Random Forest Classifier:** I chose it, because it's robust, highly flexible, and often performs well across a variety of datasets.
    - Pros:
      - Can handle non-linear relationships and interactions between features
      - Robust to outliers and noisy data.
      - Less prone to overfitting
    - Cons:
      - Less interpretable compared to simpler models like Logistic Regression
      - Requires more computation and may be slower to train compared to simpler models.
  - **Pipeline 3: Support Vector Classifier (SVC):** I chose it, because it works well in high-dimensional spaces and is effective in cases where the number of dimensions is greater than the number of samples
    - Pros:
      - Effective in high-dimensional spaces
      - Can capture complex relationships
      - Robust to overfitting
      - Versatile due to the flexibility in choosing different kernel functions
    - Cons:
      - Sensitive to the choice of kernel and hyperparameters
      - Computationally intensive

- Less interpretable compared to simpler models

```
# 2.1
from sklearn.pipeline import make_pipeline

# Pipeline 1
from sklearn.linear_model import LogisticRegression
pipeline_lr = make_pipeline(preprocessor,
LogisticRegression(max_iter=1000))

# Pipeline 2
from sklearn.ensemble import RandomForestClassifier
pipeline_rf = make_pipeline(preprocessor, RandomForestClassifier())

# Pipeline 3
from sklearn.svm import SVC
pipeline_svc = make_pipeline(preprocessor, SVC())
```

Answer:

- **2.2** .....
  - Best parameters for Logistic Regression: {'logisticregression__C': 1, 'logisticregression__penalty': 'l2', 'logisticregression__solver': 'saga'}
  - Best parameters for Random Forest: {'randomforestclassifier__max_depth': 5, 'randomforestclassifier__min_samples_split': 5, 'randomforestclassifier__n_estimators': 300}
  - Best parameters for SVC: {'svc__C': 0.1, 'svc__gamma': 'scale', 'svc__kernel': 'linear'}
  - Best score for Logistic Regression: 0.8140070921985816
  - Best score for Random Forest: 0.8219858156028369
  - Best score for SVC: 0.8179078014184397

```
# 2.2
from sklearn.model_selection import GridSearchCV

scoring = ['accuracy', 'f1']

# Define parameter grids for each pipeline
param_grid_lr = {
    'logisticregression__C': [0.001, 0.1, 1, 10],
    'logisticregression__penalty': ['l1', 'l2'],
    'logisticregression__solver': ['liblinear', 'saga']
}

param_grid_rf = {
    'randomforestclassifier__n_estimators': [100, 200, 300],
    'randomforestclassifier__max_depth': [5, 10, 15],
    'randomforestclassifier__min_samples_split': [2, 5, 10],
}
```

```
param_grid_svc = {
    'svc__C': [0.1, 1, 10],
    'svc__kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'svc__gamma': ['scale', 'auto']
}

# Perform grid search for each pipeline
grid_search_lr = GridSearchCV(pipeline_lr, param_grid_lr, cv=5,
scoring=scoring, refit='accuracy')
grid_search_rf = GridSearchCV(pipeline_rf, param_grid_rf, cv=5,
scoring=scoring, refit='accuracy')
grid_search_svc = GridSearchCV(pipeline_svc, param_grid_svc, cv=5,
scoring=scoring, refit='accuracy')

grid_search_lr.fit(X_train, y_train)
grid_search_rf.fit(X_train, y_train)
grid_search_svc.fit(X_train, y_train)

print("Best parameters for Logistic Regression: ",
grid_search_lr.best_params_)
print("Best parameters for Random Forest: ",
grid_search_rf.best_params_)
print("Best parameters for SVC: ", grid_search_svc.best_params_)
print("")
print("Best score for Logistic Regression: ",
grid_search_lr.best_score_)
print("Best score for Random Forest: ", grid_search_rf.best_score_)
print("Best score for SVC: ", grid_search_svc.best_score_)
```

```
Best parameters for Logistic Regression:  {'logisticregression__C':
0.1, 'logisticregression__penalty': 'l2',
'logisticregression__solver': 'saga'}
Best parameters for Random Forest:
{'randomforestclassifier__max_depth': 5,
'randomforestclassifier__min_samples_split': 2,
'randomforestclassifier__n_estimators': 300}
Best parameters for SVC:  {'svc__C': 1, 'svc__gamma': 'scale',
'svc__kernel': 'rbf'}

Best score for Logistic Regression:  0.8140070921985816
Best score for Random Forest:  0.8219858156028369
Best score for SVC:  0.8179078014184397
```

```
# Update Logistic Regression pipeline
pipeline_lr.set_params(**grid_search_lr.best_params_)

# Update Random Forest pipeline
pipeline_rf.set_params(**grid_search_rf.best_params_)
```

```
# Update SVC pipeline
pipeline_svc.set_params(**grid_search_svc.best_params_)

Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('numerical',
StandardScaler(),
                                                  ['age', 'trestbps',
'chol',
                                                   'thalach',
'oldpeak']),
                                                 ('categorical',

OneHotEncoder(handle_unknown='ignore',

sparse_output=False),
                                                  ['cp', 'restecg']),
                                                 ('binary',
'passthrough',
                                                  ['sex', 'fbs',
'exang'])])),
                ('svc', SVC(C=1))])
```

Answer:

- **2.3** .....
  - Best score for Logistic Regression: 0.8140070921985816
  - Best score for Random Forest: 0.8219858156028369
  - Best score for SVC: 0.8179078014184397
  - Stacked
    - Accuracy: 0.80 ± 0.04
    - F1 Score: 0.70 ± 0.09

```
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
# 2.3
# Define the stacking classifier
stacking_classifier = StackingClassifier(
    estimators=[
        ('lr', pipeline_lr),
        ('rf', pipeline_rf),
        ('svc', pipeline_svc)
    ],
    final_estimator=LogisticRegression(max_iter=1000)
)

# Perform cross-validation
skf = StratifiedKFold(n_splits=5)
accuracy_scores = []
```

```
f1_scores = []

for train_index, test_index in skf.split(X, y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    stacking_classifier.fit(X_train, y_train)
    y_pred = stacking_classifier.predict(X_test)

    accuracy_scores.append(accuracy_score(y_test, y_pred))
    f1_scores.append(f1_score(y_test, y_pred))

# Calculate mean and standard deviation of scores
import numpy as np
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)
mean_f1 = np.mean(f1_scores)
std_f1 = np.std(f1_scores)

# Print the results
print(f"Accuracy: {mean_accuracy:.2f} ± {std_accuracy:.2f}")
print(f"F1 Score: {mean_f1:.2f} ± {std_f1:.2f}")
```

```
Accuracy: 0.80 ± 0.04
F1 Score: 0.70 ± 0.09
```

Answer:

- **2.4** .....
  - Logistic Regression:
    - The stacking classifier's performance is slightly lower than the best score achieved by Logistic Regression alone. Stacking might have provided stability to predictions, lower margin of error
  - Random Forest:
    - The stacking classifier's performance is similar to Random Forest alone. In this case, stacking did not improve performance. Possibly provided robustness by combining predictions from other models
  - SVC:
    - The stacking classifier's performance is similar to SVC alone. Possibly have provided more consistent predictions.
  - Possible reasons for the stacking classifier not deteriorated prediction accuracy:
    - Lack of diversity among the base models
    - Complexity of the problem may not benefit significantly from stacking
    - Suboptimal choice of hyperparameters in the stacking classifier

Bonus Question: The stacking classifier has achieved a high accuracy and F1 score, but there may be still room for improvement. Suggest **two** possible ways to improve the modeling using the stacking classifier, and explain **how** and **why** they could improve the performance. **(2 points)**

Answer:

1. Using more models:
   - HOW: Include a broader range of base models, possibly from different families of algorithms or with different parameter settings.
   - WHY: A wider variety of models allow the ensemble to capture a broader range of patterns and reduce risk of overfitting on specific features. Adding models with different strengths can result in a more robust overall prediction
2. Outlier Detection and Handling:
   - HOW: Implement techniques to detect and handle outliers in the dataset before training the stacking classifier. We can use stats methods such as Z-Score to filter outliers
   - WHY: Outliers can negatively impact the performance of machine learning models by skewing the learned relationships and introducing noise