Assignment 4: Pipelines and Hyperparameter Tuning

- **Full Name** = Hiu Sum Yuen
- **UCID** = 30162577

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.

```
!pip install pandas
```

```
    Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (1.5.3)
    Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
    Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.4)
    Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.25.2)
    Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas) (1.
```

```
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	slope	c
0	28	1	2	130.0	132.0	0.0	2.0	185.0	0.0	0.0	NaN	Nal
1	29	1	2	120.0	243.0	0.0	0.0	160.0	0.0	0.0	NaN	Nal
2	29	1	2	140.0	NaN	0.0	0.0	170.0	0.0	0.0	NaN	Nal
3	30	0	1	170.0	237.0	0.0	1.0	170.0	0.0	0.0	NaN	Nal
4	31	0	2	100.0	219.0	0.0	1.0	150.0	0.0	0.0	NaN	Nal
...	...	...	...	...	...	...	...	...	...	...	...	.
289	52	1	4	160.0	331.0	0.0	0.0	94.0	1.0	2.5	NaN	Nal
290	54	0	3	130.0	294.0	0.0	1.0	100.0	1.0	0.0	2.0	Nal
291	56	1	4	155.0	342.0	1.0	0.0	150.0	1.0	3.0	2.0	Nal
292	58	0	2	180.0	393.0	0.0	0.0	110.0	1.0	1.0	2.0	Nal
293	65	1	4	130.0	275.0	0.0	1.0	115.0	1.0	1.0	2.0	Nal

294 rows × 14 columns

Next steps: Generate code with df View recommended plots

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** I found the percentage of NaN values per column, then dropped the columns with 60% or higher NaN occurences using the pandas built in drop function, using '.index' to identify the columns to be dropped, as a list for the parameter 'columns'.

```
# 1.1
isNaPercentage = df.isna().mean()
columnsToDrop = isNaPercentage[isNaPercentage > 0.6].index
columnsToDrop
```

```
    Index(['slope', 'ca', 'thal'], dtype='object')
```

```
df.drop(columns = columnsToDrop, inplace = True) # default: inplace = False
df
```

	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 × 11 columns

Next steps: Generate code with `df` View recommended plots

Answer:

- **1.2** Through interpretation of which features are numerical and categorical respectively, impute numerical features with mean strategy and categorical features with most frequent strategy; Categorical features should not have decimals as the integer assigned to it represents ordinal encoding.

```
df.isna().sum()
```

```
age          0
sex          0
cp           0
trestbps     1
chol        23
fbs          8
restecg      1
thalach      1
exang        1
oldpeak      0
num          0
dtype: int64
```

```
# 1.2
from sklearn.impute import SimpleImputer

numericalFeatures = ["age", "trestbps", "chol", "thalach", "oldpeak"]
categoricalFeatures = ["sex", "cp", "fbs", "restecg", "exang"]

numImputer = SimpleImputer(strategy = "mean")
catImputer = SimpleImputer(strategy = "most_frequent")

df[numericalFeatures] = numImputer.fit_transform(df[numericalFeatures])
df[categoricalFeatures] = catImputer.fit_transform(df[categoricalFeatures])

df.isna().sum()
```

```
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: "age", "trestbps", "chol", "thalach", "oldpeak"
    - Standard Scaler is normalization meant for numerical data.
  - categorical features: "cp", "restecg"
    - One Hot Encoding expresses categorical data as binary.
  - binary features: "sex", "fbs", "exang"
    - Scaling not neccessary for binary data as it is neither continuous nor categorical.

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

numericalFeatures = ["age", "trestbps", "chol", "thalach", "oldpeak"]
categoricalFeatures = ["cp", "restecg"]
binaryFeatures = ["sex", "fbs", "exang"]

preprocessor = ColumnTransformer(
    transformers = [
        ("num", StandardScaler(), numericalFeatures),
        ("cat", OneHotEncoder(), categoricalFeatures),
        ("bin", "passthrough", binaryFeatures)
    ],
    remainder = "drop"
)

X = df.drop(columns = "num")
y = df["num"]
```

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**
  - logistic regression:
    - For simplicity.
    - Strengths: Simple to implement, easy to interpret.
    - Weaknesses: Sensitivity to outliers, poor handling of non-linear relationships.
  - gradient boosting:
    - More complexity when fitting.
    - Strengths: Works well with mixture of binary and continuous features, good scaling.
    - Weaknesses: Sensitive to hyperparameters. Poor performance with high-dimensional sparse data.
  - SVC:
    - Performs well on a large variety of datasets.

- Strengths: Works well in both low and high dimensional data.
- Weaknesses: Don't scale with number of samples.

```
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import SVC
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

```
# 2.1
lrModel = LogisticRegression()
gbModel = GradientBoostingClassifier()
svcModel = SVC()

# lrPipe = make_pipeline(preprocessor, lrModel)
# gbPipe = make_pipeline(preprocessor, gbModel)
# svcPipe = make_pipeline(preprocessor, svcModel)

lrPipe = Pipeline([
    ("preprocessing", preprocessor),
    ("classifier", lrModel)
    ])

gbPipe = Pipeline([
    ("preprocessing", preprocessor),
    ("classifier", gbModel)
    ])

svcPipe = Pipeline([
    ("preprocessing", preprocessor),
    ("classifier", svcModel)
    ])
```

Answer:

- **2.2**
- The best hyper parameters for:
  - LogisticRegression:
    - C = 1
  - GradientBoostingClassifier:
    - learning rate = 1
    - n_estimators = 150
  - SVC:
    - C = 0.01
    - gamma = 0.001
    - kernel = linear

```
# 2.2
from sklearn.model_selection import GridSearchCV

lrParamGrid = {'classifier__C': [0.001, 0.01, 0.1, 1, 10]}
gbParamGrid = {'classifier__n_estimators': [50, 100, 150], 'classifier__learning_rate': [0.01, 0.1, 1]}
svcParamGrid = {'classifier__C': [0.001, 0.01, 0.1, 1, 10], 'classifier__gamma': [0.001, 0.01, 0.1, 1], 'classifier__kernel': ['

paramGrids = [lrParamGrid, gbParamGrid, svcParamGrid]
pipes = [lrPipe, gbPipe, svcPipe]
models = [lrModel, gbModel, svcModel]
best_params = []

for paramGrid, pipe, model in zip(paramGrids, pipes, models):
    gridSearch = GridSearchCV(pipe, paramGrid, cv = 5)
    gridSearch.fit(X_train, y_train)
    print(f"Model:\t{model}")
    print(f"\tBest parameters: {gridSearch.best_params_}")
    print(f"\tBest score: {gridSearch.best_score_}")
    best_params.append(gridSearch.best_params_)
    print("\n")

lrParams = best_params[0]
gbParams = best_params[1]
svcParams = best_params[2]

lrModel = LogisticRegression(C = lrParams["classifier__C"])
gbModel = GradientBoostingClassifier(n_estimators = gbParams["classifier__n_estimators"], learning_rate = gbParams["classifier__
svcModel = SVC(C = svcParams["classifier__C"], gamma = svcParams["classifier__gamma"], kernel = svcParams["classifier__kernel"])
```

```
Model:  LogisticRegression()
        Best parameters: {'classifier__C': 1}
        Best score: 0.8170212765957446


Model:  GradientBoostingClassifier()
        Best parameters: {'classifier__learning_rate': 1, 'classifier__n_estimators': 150}
        Best score: 0.7914893617021276


Model:  SVC()
        Best parameters: {'classifier__C': 0.01, 'classifier__gamma': 0.001, 'classifier__kernel': 'linear'}
        Best score: 0.8382978723404255
```

Answer:

- **2.3** I chose Logistic Regression for the final estimator meta-model as it is simple to implement and interpret. Predictions of the base estimators are treated as features of the final model training.

```
# 2.3
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
import numpy as np

classifierStack =  StackingClassifier(
    estimators = [
        ('lr', lrModel),
        ('gb', gbModel),
        ('svc', svcModel)
    ],
    final_estimator = LogisticRegression()
)

skf = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 0)

trainAccuracyList = []
trainF1List = []

testAccuracyList = []
testF1List = []

for fold, (train_idx, test_idx) in enumerate(skf.split(X, y), 1):
    XTrainFold, XTestFold = X.iloc[train_idx], X.iloc[test_idx]
    yTrainFold, yTestFold = y.iloc[train_idx], y.iloc[test_idx]
```

```
        classifierStack.fit(XTrainFold, yTrainFold)

        yTrainPred = classifierStack.predict(XTrainFold)
        yTestPred = classifierStack.predict(XTestFold)

        trainAccuracy = accuracy_score(yTrainFold, yTrainPred)
        testAccuracy = accuracy_score(yTestFold, yTestPred)
        trainF1 = f1_score(yTrainFold, yTrainPred, average = "weighted")
        testF1 = f1_score(yTestFold, yTestPred, average = "weighted")

        trainAccuracyList.append(trainAccuracy)
        trainF1List.append(trainF1)
        testAccuracyList.append(testAccuracy)
        testF1List.append(testF1)

    trainaccuracyMean = np.mean(trainAccuracyList)
    trainaccuracyStd = np.std(trainAccuracyList)
    trainf1Mean = np.mean(trainF1List)
    trainf1Std = np.std(trainF1List)

    testAccuracyMean = np.mean(testAccuracyList)
    testAccuracyStd = np.std(testAccuracyList)
    testF1Mean = np.mean(testF1List)
    testF1Std = np.std(testF1List)


    print(f"Train Accuracy: {trainaccuracyMean:.4f} +/- {trainaccuracyStd:.4f}")
    print(f"Test Accuracy: {testAccuracyMean:.4f} +/- {testAccuracyStd:.4f}")
    print(f"Train F1: {trainf1Mean:.4f} +/- {trainf1Std:.4f}")
    print(f"Test F1: {testF1Mean:.4f} +/- {testF1Std:.4f}")
```

```
Train Accuracy: 0.8461 +/- 0.0120
Test Accuracy: 0.8333 +/- 0.0333
Train F1: 0.8429 +/- 0.0121
Test F1: 0.8292 +/- 0.0337
```

Answer:

- **2.4**
    - Stacking classifier results:
        - accuracy = 0.8366
        - f1 = 0.8324
    - Logistic Regression:
        - accuracy = 0.8644
        - f1 = 0.8637
        - Logistic regression shows a flat out improvement throughout, demonstrating that the dataset is likely linearly relational.
    - Gradient Boosting:
        - accuracy = 0.8475
        - f1 = 0.8461
        - Gradient boosting has shown better validation results. However, this model is undesirable as it vastly overfits the training data, with a score of 1.0 for both training accuracy and training f1 score.
    - SVC:
        - accuracy = 0.8136
        - f1 = 0.8052
        - SVC has worse validation scores than the stacking classifier. SVC is sensitive to data preprocessing and hyperparameter settings, so there could have been other hyperparameter settings that could result in a better model.

```
models = [lrModel, gbModel, svcModel]
modelNames = ["Logistic Regression", "Gradient Boosting", "SVC"]

for model, name in zip(models, modelNames):
    model.fit(X_train, y_train)
    yTrainPred = model.predict(X_train)
    yTestpred = model.predict(X_test)

    trainAccuracy = accuracy_score(y_train, yTrainPred)
    testAccuracy = accuracy_score(y_test, yTestpred)
    trainF1 = f1_score(y_train, yTrainPred, average = "weighted")
    testF1 = f1_score(y_test, yTestpred, average = "weighted")

    print(f"{name}:")
    print(f"  train accuracy: {trainAccuracy:.4f}")
    print(f"  test accuracy: {testAccuracy:.4f}")
    print(f"  train f1: {trainF1:.4f}")
    print(f"  test f1: {testF1:.4f}\n")
```

```
Logistic Regression:
  train accuracy: 0.8468
  test accuracy: 0.8475
  train f1: 0.8421
  test f1: 0.8471

/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
Gradient Boosting:
  train accuracy: 1.0000
```

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)**

```
SVC
```

Answer:

```
  train f1: 0.8249
```

1. Use a different meta-model. Logistic Regression is a good choice due to implementation simplicity, but there can be other models that result in better accuracy.
2. More careful tuning of hyperparameters of the base and meta-models can increase performance of the stacking classifier.

Start coding or generate with AI.