ensf444-assignment4

March 20, 2024

Assignment 4: Pipelines and Hyperparameter Tuning

- **Full Name** = David Rodriguez
- **UCID** = 30145288 ***

In this assignment, you will be putting together everything you have learned so far. You will need to find your own dataset, 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**

0.1 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.

[85]:
```
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 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** …… ## 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. As shown below the columns with more than 60% of its entries empty or NaN, are slope, ca, and thal. The reason this is a responsible way to handle these columns is because filling them in leads to high bias. Most metrics such as using the mean, most_frequent, and a constant are not suitable when it substitutes a significant portion of the data. Additionally, since we still have several other features that can be utilized to train and predict our model it is not irresponsible to drop the column.

Upon further examination the columns being dropped slope refers to the slop of the peak exercise ST segment on the patient's ECG during testing. 'ca' is the may stand for the number of major vessels colored by fluoroscopy or the coronary arteries. 'thal' refers to thalassemia, which is a genetic blood disorder that affects the production of hemoglobin. It is characterized by abnormal hemoglobin production. These are likely to be empty because they are harder to examine and are not often recorded. In future models, including them may be ideal.

[86]:
```
# 1.1
# Add necessary code here.
# print(len(df))
# print(df.isnull().sum())
missing_values = df.isnull().sum() # Count the number of missing values in each␣
 ↪column
missing_values = missing_values[missing_values > 0.6 * len(df)] # Select␣
 ↪columns with more than 60% missing values
print(missing_values) # Display the columns with more than 60% missing values
df = df.drop(missing_values.index, axis=1) # Drop the columns with more than␣
 ↪60% missing values
print(df.isnull().sum()) # Display the number of missing values in each column
```

```
# Inspect the nature of the data in each column to see if its binary,␣
 ↪numerical, or categorical
print('trestbps:', df['trestbps'].unique())
# print('chol:', df['chol'].unique())
print('fbs:', df['fbs'].unique())
print('restecg:', df['restecg'].unique())
print('thalach:', df['thalach'].unique())
print('exang:', df['exang'].unique())
print('oldpeak:', df['oldpeak'].unique())
```

```
slope    190
ca       291
thal     266
dtype: int64
age          0
sex          0
cp           0
trestbps     1
chol        23
fbs          8
restecg      1
thalach      1
exang        1
oldpeak      0
num          0
dtype: int64
trestbps: [130. 120. 140. 170. 100. 105. 110. 125. 150.  98. 112. 145. 190. 160.
 115. 142. 180. 132. 135.  nan 108. 124. 113. 122.  92. 118. 106. 200.
 138. 136. 128. 155.]
fbs: [ 0. nan  1.]
restecg: [ 2.  0.  1. nan]
thalach: [185. 160. 170. 150. 165. 184. 155. 190. 168. 180. 178. 172. 130. 142.
  98. 158. 129. 146. 145. 120. 106. 132. 140. 138. 167. 188. 144. 137.
 136. 152. 175. 176. 118. 154. 115. 135. 122. 110.  90. 116. 174. 125.
  nan 148. 100. 164. 139. 127. 162. 112. 134. 114. 128. 126. 124. 153.
 166. 103. 156.  87. 102.  92.  99. 121.  91. 108.  96.  82. 105. 143.
 119.  94.]
exang: [ 0.  1. nan]
oldpeak: [0.  1.  2.  1.5 0.5 3.  0.8 2.5 4.  5. ]
```

Answer:

- **1.2** ……

There are multiple ways of filling in null values, utilizing the feature's mean, the feature's most_frequent, and a constant. Other methods in basic models can use ffil, bfill and different interpolation methods, these are not able to be applied to our model due to the unorganized data, and high dimension disabling this possibility. The mean, most_frequent, and constant are good options here depending on the data type. The mean way is valid for numerical and continous data

since it still somewhat preserves the nature of the data. This will likely be an outlier in the model however, when data is scarce including it could prove beneficial. For categorical and binary data types, it is best to not use this method since it is possible to create a whole new unique data type which can undermine the function of the model. That's why I utilized most_frequent for these data types. In the examples: the binary and categorical features that had empty values were the, fbs, restecg, exang, this was evident upon inspecting the unique data types, and reading the description online. The numerical feature were chol, trestbps and thalch due to them being continous data types. Without investigating further at the causes of this missing data we can only try to fill it in to the best of our ability. Alternatively a good possibility could have been to preserve the ratio that currently exists to fill it however, I think mode is a good fit since it mimics this behaviour and otherwise the filling could be nondeterministic.

[87]:
```
# 1.2
# Add necessary code here.
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
# for x in df[df.isnull().sum().index]:
#     print(x)
#     print(df[x].unique())

Binary_And_Categorical = ['fbs','restecg','exang','sex','cp']
Numerical_Features = ['age','chol','trestbps','thalach','oldpeak']
imputerBinaryAndCategorical = SimpleImputer(strategy='most_frequent') # Create␣
 ↪an imputer object with a mean filling strategy
imputerNumerical = SimpleImputer(strategy='mean') # Create an imputer object␣
 ↪with a mean filling strategy

preprocessor = ColumnTransformer(
    transformers=[
        ('num', imputerNumerical, Numerical_Features),  # Impute numerical␣
 ↪features with mean
        ('cat', imputerBinaryAndCategorical, Binary_And_Categorical)  # Impute␣
 ↪categorical features with most frequent
    ],
    remainder='passthrough'  # Include all remaining columns in the output␣
 ↪DataFrame
)

other_columns = list(set(df.columns) - set(Numerical_Features) -␣
 ↪set(Binary_And_Categorical))
df_imputed = pd.DataFrame(preprocessor.fit_transform(df),␣
 ↪columns=Numerical_Features + Binary_And_Categorical + other_columns)

# print(df_imputed) # Display the number of missing values in each column
# print(df_imputed.isnull().sum()) # Display the number of missing values in␣
 ↪each column
```

```
print(df_imputed.isnull().sum()) # Display the number of missing values in each␣
 ↪column
# for x in df.columns:
#     # print(df[x], df[x].dtype)
```

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

Answer:

- **1.3** …… The numerical features are as follows: age, trestbps, chol, thalach, and oldpeak. The categorical features are cp and restecg. The binary features are sex,fbs, and exang.

This was all determined above by utilizing the uniques and investigating the nature of the data and what it represented. The binary categories were as followed: sex was 0,1 in the dataset representing male or female sex. Fbs is 0,1 or NaN (from empty data) which may mean fasting blood sugar is yes or no. Exang is 0,1 as well, likely meaning exercise-induced angina is either yes or no. Upon evaluating the others, there were lots of numerical data that was from measurements or did not pertain to a single category. Examples of this were chol(cholesterol levels, thalach, or thalach rate, trestbps, likely meaning resting blood pressure, and oldpeak, a magnitude of "ST depression induced by exercise relative to rest" according to papers from the NIH) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9931474/#:~:text=Oldpeak%20%3D%20ST%20depression%20c The categorical ones were determined by those that had classifications of 3 or more classes (non binary). Examples of this were the cp, meaning chest pain which is an ordinal category scaling based on the amount of pain. Anothe ris restecg which is likely, the resting EKG which is a nonordinal category since there are different classifications, 1 is normal, 2 is a ST-T wave anomaly, 3 is left ventricular hypertrophy, and 4 is any other abnormality.

These four different types of data require different transformations. The categories of them are as follows: Numerical Categorical Ordinal Categorical non-ordinal Binary

They require different transformations because the data they represent is different, numerical is continuous whereas categorical and binary is absolute so its a part of a bucket. Ordinal Categorical has scales which fluctuate. Thus keeping this is important for the data. Non-ordinal data doesn't care about the order and thus we can just use OneHotEncoder. We need to use StandardScaler for numerical because it handles outliers, it can also help models converge faster when there is this scaling applying. Whereas binary already has a yes or no so no need for anything is needed.

[88]:
```
# 1.3
# Add necessary code here.

# 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.
y = df_imputed['num']
X = df_imputed.drop(columns=['num'])


from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline

print(X.dtypes)
# num2 =list(df.select_dtypes(include=['float64']).columns)
numerical_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
categorical_features = ['cp','restecg',]
binary_features = ['sex', 'fbs', 'exang']



preprocessor = ColumnTransformer([
            ('Numerical', StandardScaler(), numerical_features),
            ('Categorical', OneHotEncoder(sparse_output = False,␣
 ↪handle_unknown='ignore'), categorical_features),
            ('Binary', 'passthrough', binary_features)
    ]
)
```

```
age         float64
chol        float64
trestbps    float64
thalach     float64
oldpeak     float64
fbs         float64
restecg     float64
exang       float64
sex         float64
cp          float64
```

```
dtype: object
```

2 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`. Interpret the results and compare them with the baseline scores from the previous assignment. **(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** ……

Selected Models:

The three models I chose to evaluate in the three pipelines were as follows: Logistic Regression, SVC pipeline, and gradient boosting. These three different models provide different benefits and drawbacks associated. Their applicability varies depending on the data type and its nature regarding how many dimension it has, how effective linear patterns will be, and also how interpretable we would like the model to be. Here are the strengths and weaknesses of each model.

2.1 Logistic Regression

Logistic regression models are very fast to train and predict. This allows them to be able to utilize large amounts of data without heavy hardware requirements. Additionally, since their calculations are typically calculated utilizing a weighted coefficient of features and then a threshold to then categorize it creating S-curve shapes, it could enable easy scalability with large dimensionality. Since it doesnt require that much computing. It is also extremely interpretable to other people. The hyperparameter that I really want to evaluate is the Logistic Regression solver method there are multiple ways to optimize the data to create the function and I would like to see this.

Drawbacks of this method are that it typically too basic for lots of data samples' correlations. This is caused by non-linear relationships with feature vectors in different segment values. Therefore,

its likely to underfit the dataset.

2.2 SVC

SVC models (Support Vector Classification) create boundary areas within a multidimensional space to classify various samples depending on the type of SVC hyperplanar created (linear, newton, saga, among others). These models can be incredibly strong at finding relationships for complex data.

They may not be as great for categorical feature types since this is usually used to create functions and areas for continuous data and thus may be overkill. However, they are computationally intensive and thus do not scale well on large feature dimensionality for creating the models. They also require careful scaling and preprocessing of the data to be able to generate a model. An immense pro is that it enables so many hyperparameter selection and specification that you can customize how underfit and overfit you would like it. Additionally Linear kernels solve your concerns of dimensionality scaling since they are less computation intensive. This model is great for pipelines since it gives us lots of hyperparameters to tune leading to great models.

2.3 Gradient Boosting

Gradient boosting is a model technique which is an ensemble of decision trees and combines weak learner trees to create stronger model. Additionally, the model is retrained and fine tuned using the learning rate to change how precise you want it to be. The benefits of this model are that it can handle complex data that is non-inear. This allows it to work for multiple types of datasets and samples. It provides feature importance which could be valuable in assessing high risk and low risk indicators for heart disease as shown in the data sample we are using. It has high predictive accuracy, this is achieved by combining ensembles of models and also relearning enabling very strong models. This could lead to overfitting depending on the learning rate but it is very strong. It can handle various data types such as numerical, categorical among others very nicely as well.

Downsides of this method are that it requires very careful hyperparameter tuning regarding the learning rate, n_learners among others. Additionally, the computation necessity of this is insanely high due to relearning over and over as well as combining methods. This leads it to be slow and harder to scale and train later on. It has hard interpretability compared to other models simply due to the ensemble making it harder for healthcare to use for our dataset. It could likely overfit too so we have to be very careful with parameters. This model is great for pipelining since there are several parameters to tune in combination making it a great candidate for this.

[89]:
```
# 2.1
# Add necessary code here.
# create three pipeline objects that take the column transformer from the␣
 ↪pervious question as the frist step and add one or more models as the␣
 ↪subsequeent steps. You can use any models from sklearn or other libraries.
#or other libraries that are suitable for binary classification. For each␣
 ↪pipeline, expalin why you selected the model(s) and what are tehir strengths␣
 ↪and weakensses for thsi data set.add
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import SVC
```

```
LogisticRegression_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('LogisticRegression', LogisticRegression())
])
SVC_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('SVC', SVC())
])
GradientBoosting_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('GradientBoosting', GradientBoostingClassifier())
])
```

Answer:

- **2.2** ……. In this code segment the reports were as follows: Logistic Regression Best parameters for LogisticRegression: {'LogisticRegression___C': 0.1, 'LogisticRegression___max_iter': 500, 'LogisticRegression___solver': 'liblinear'}

Best score for LogisticRegression on training data: 0.8803030303030303

Best accuracy score on testing data: 0.7923728813559322

SVC Best parameters for SVC: {'SVC___C': 1, 'SVC___gamma': 0.1, 'SVC___kernel': 'sigmoid'}

Best score for SVC on training on training data: 0.8969696969696969

Best accuracy score on testing data: 0.7923728813559322

GradientBoostingClassifier Best parameters for GradientBoosting: {'GradientBoosting___learning_rate': 0.1, 'GradientBoosting___max_depth': 3, 'GradientBoosting___n_estimators': 100} Best score for GradientBoosting on training data: 0.7954545454545455 Best accuracy score on testing data: 0.7838983050847458

Thus as seen here the best pipelines at first glance with the proper parameters found are SVC which had a training data accuracy of 0.8803030303030303 and a testing data accuracy of 0.7923728813559322.

[95]:
```
# 2.2
# Add necessary code here.
#  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.

from sklearn.model_selection import GridSearchCV, train_test_split
import warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.exceptions import FitFailedWarning
```

```
warnings.filterwarnings("ignore", category=UserWarning, message="Setting␣
 ↪penalty=None will ignore the C and l1_ratio parameters")
warnings.filterwarnings("ignore", category=FitFailedWarning)
warnings.filterwarnings("ignore", category=ConvergenceWarning)

param_grid_LogisticRegression = {
    'LogisticRegression__C': [0.1, 1, 10, 100],
    'LogisticRegression__solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag',␣
 ↪'saga'],
    'LogisticRegression__max_iter': [500, 750, 1000],
    'LogisticRegression__penalty': ['l2']

}

param_grid_SVC = {
    'SVC__C': [0.1, 1, 10, 100],
    'SVC__gamma': [1, 0.1, 0.01, 0.001],
    'SVC__kernel': ['rbf', 'poly', 'sigmoid','linear'],
    'SVC__probability': [True, False]
}
param_grd_GradientBoosting = {
    'GradientBoosting__n_estimators': [100, 200, 300],
    'GradientBoosting__learning_rate': [0.1, 0.01, 0.001],
    'GradientBoosting__max_depth': [3, 4, 5],
    'GradientBoosting__min_samples_split': [2, 3, 4],
    'GradientBoosting__subsample': [0.8, 0.9, 1.0]
}
# print(X,y)

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.2,␣
 ↪random_state=42)
# print(X_train, X_test, y_train, y_test)
grid_LogisticRegression =␣
 ↪GridSearchCV(estimator=LogisticRegression_pipeline,param_grid =␣
 ↪param_grid_LogisticRegression,scoring='accuracy', cv=5)
grid_LogisticRegression.fit(X_train, y_train)

grid_SVC = GridSearchCV(SVC_pipeline, param_grid_SVC, cv=5)
grid_SVC.fit(X_train, y_train)

grid_GradientBoosting = GridSearchCV(estimator=GradientBoosting_pipeline,␣
 ↪param_grid=param_grd_GradientBoosting,scoring='accuracy', cv=5)
grid_GradientBoosting.fit(X_train, y_train)

print("----------------------------------")
print("Logistic Regression")
```

```
print("Best parameters for LogisticRegression: ", grid_LogisticRegression.
 ↪best_params_)
print("Best score for LogisticRegression on training data: ",␣
 ↪grid_LogisticRegression.best_score_)
print("Best accuracy score on testing data: ", grid_LogisticRegression.
 ↪score(X_test, y_test))
print("-----------------------------------")
print("SVC")
print("Best parameters for SVC: ", grid_SVC.best_params_)
print("Best score for SVC on training on training data: ", grid_SVC.best_score_)
print("Best accuracy score on testing data: ", grid_SVC.score(X_test, y_test))

print("-----------------------------------")
print("GradientBoostingClassifier")
print("Best parameters for GradientBoosting: ", grid_GradientBoosting.
 ↪best_params_)
print("Best score for GradientBoosting on training data: ",␣
 ↪grid_GradientBoosting.best_score_)
print("Best accuracy score on testing data: ", grid_GradientBoosting.
 ↪score(X_test, y_test))

LogisticRegression_pipeline.set_params(**grid_LogisticRegression.best_params_)
SVC_pipeline.set_params(**grid_SVC.best_params_)
GradientBoosting_pipeline.set_params(**grid_GradientBoosting.best_params_)
```

```
-----------------------------------
Logistic Regression
Best parameters for LogisticRegression:  {'LogisticRegression__C': 0.1,
'LogisticRegression__max_iter': 500, 'LogisticRegression__penalty': 'l2',
'LogisticRegression__solver': 'liblinear'}
Best score for LogisticRegression on training data:  0.8969696969696971
Best accuracy score on testing data:  0.7838983050847458
-----------------------------------
SVC
Best parameters for SVC:  {'SVC__C': 1, 'SVC__gamma': 0.1, 'SVC__kernel':
'sigmoid', 'SVC__probability': True}
Best score for SVC on training on training data:  0.8969696969696969
Best accuracy score on testing data:  0.7923728813559322
-----------------------------------
GradientBoostingClassifier
Best parameters for GradientBoosting:  {'GradientBoosting__learning_rate': 0.1,
'GradientBoosting__max_depth': 3, 'GradientBoosting__min_samples_split': 4,
'GradientBoosting__n_estimators': 300, 'GradientBoosting__subsample': 1.0}
Best score for GradientBoosting on training data:  0.8287878787878789
Best accuracy score on testing data:  0.8008474576271186
```

[95]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('Numerical', StandardScaler(),
                                                  ['age', 'trestbps', 'chol',
                                                   'thalach', 'oldpeak']),
                                                 ('Categorical',
OneHotEncoder(handle_unknown='ignore',
sparse_output=False),
                                                  ['cp', 'restecg']),
                                                 ('Binary', 'passthrough',
                                                  ['sex', 'fbs', 'exang'])])),
                ('GradientBoosting',
                 GradientBoostingClassifier(min_samples_split=4,
                                            n_estimators=300))])
```

Answer:

- **2.3** ……

I chose Logistic Regression as my meta model because it is interpretable, scalable, yet it keeps up with regards to accuracy compared to the other models. Its simplicity and interpretability could provide good uses for people that will use this model such as health care practitioners who need to understand the models and their reasoning. Its applicability for larger features and feature addition in the future make it a good candidate.

The mean and standard deviation were good and they were as follows: 0.8064289888953828 AND 0,05364684966536485. The F1 Score was with a standard deviation of 0.7161119582172214 AND 0.08345023571320057. While investigating further we can find that these metrics are very consistent regardless of the counter and the different validation test the mean is the very close and the deviation is minor 0.05/0.80 is very little for the mean and then for the F1 Score 0.0834/0.71611 is pretty low as well. This means they perform well in different sets of data due to not overfitting or severly underfitting.

In my previous assignment the results I got were a respective training and validation accuracy of 0.70 and 0.66 respectively for SVC and then 0.97 and 0.89 for decision trees. The f1 score for the decision tree was 0.97 on average. The SVC's performance on the training and testing accuracy was pretty bad and low signifying excessive underfitting or poor model creation by the hyperparameters chosen. The decision trees' performance was fantastic with near perfect accuracies in training and testing as well as high F1 scores. (We didnt compute the average). When comparing these two models one might say the decision tree is simply better however, they were analyzing different datasets. Sometimes the default hyperparameters are better and other times they are more complex and need to be fine tuned more a with more complicated models like the ones chosen here. In this scenario the mean and standard deviation are respectable and the disparities could be accounted for by the different data and different parameters.

[ ]:
```
# 2.3
# Add necessary code here.
# 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`.
# Interpret the results and compare them with the baseline scores from the␣
 ↪previous assignment.
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
import numpy as np

meta_model= LogisticRegression()

stacking_classifier = StackingClassifier(
    estimators=[
        ('LogisticRegression', LogisticRegression_pipeline),
        ('SVC', SVC_pipeline),
        ('GradientBoosting', GradientBoosting_pipeline)
    ],
    final_estimator=meta_model
)

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
accuracy_scores = []
f1_scores = []
counter =0
for train_index, test_index in cv.split(X, y):
    X_train_fold, X_test_fold = X.iloc[train_index], X.iloc[test_index]
    y_train_fold, y_test_fold = y.iloc[train_index], y.iloc[test_index]
    stacking_classifier.fit(X_train_fold, y_train_fold)
    y_pred = stacking_classifier.predict(X_test_fold)
    accuracy_scores.append(accuracy_score(y_test_fold, y_pred))
    f1_scores.append(f1_score(y_test_fold, y_pred))
    print("accuracy for fold",counter,":", accuracy_score(y_test_fold, y_pred))
    print("f1 for fold",counter,":", f1_score(y_test_fold, y_pred))
    print()
    counter+=1
print()
print("Mean and Standard Deviation")
print(f"Accuracy (mean ± std): {sum(accuracy_scores) / len(accuracy_scores)} ±␣
 ↪{np.std(accuracy_scores)}")
```

```
print(f"F1 Score (mean ± std): {sum(f1_scores) / len(f1_scores)} ± {np.
  ↪std(f1_scores)}")
```

```
accuracy for fold 0 : 0.7966101694915254
f1 for fold 0 : 0.7142857142857143

accuracy for fold 1 : 0.8135593220338984
f1 for fold 1 : 0.717948717948718

accuracy for fold 2 : 0.7288135593220338
f1 for fold 2 : 0.5789473684210527

accuracy for fold 3 : 0.7966101694915254
f1 for fold 3 : 0.7272727272727273

accuracy for fold 4 : 0.896551724137931
f1 for fold 4 : 0.8421052631578947


Mean and Standard Deviation
Accuracy (mean ± std): 0.8064289888953828 ± 0.05364684966536485
F1 Score (mean ± std): 0.7161119582172214 ± 0.08345023571320057
```

Answer:

- **2.4** …… The final results of the stacking classifier seem promising with the mean and standard deviation being 0.8064289888953828 AND 0.05364684966536485. The F1 Score was with a standard deviation of 0.7161119582172214 AND 0.08345023571320057. They improved the results gathered in the individual models above which yielded the following results:

Logistic Regression Best parameters for LogisticRegression: {'LogisticRegression___C': 0.1, 'LogisticRegression___max_iter': 500, 'LogisticRegression___penalty': 'l2', 'LogisticRegression___solver': 'liblinear'} Best score for LogisticRegression on training data: 0.8969696969696971 Best accuracy score on testing data: 0.7838983050847458

SVC Best parameters for SVC: {'SVC___C': 1, 'SVC___gamma': 0.1, 'SVC___kernel': 'sigmoid', 'SVC___probability': True}

Best score for SVC on training on training data: 0.8969696969696969

Best accuracy score on testing data: 0.7923728813559322

GradientBoostingClassifier

Best parameters for GradientBoosting: {'GradientBoosting___learning_rate': 0.1, 'GradientBoosting___max_depth': 3, 'GradientBoosting___min_samples_split': 4, 'GradientBoosting___n_estimators': 300, 'GradientBoosting___subsample': 1.0}

Best score for GradientBoosting on training data: 0.8287878787878789

Best accuracy score on testing data: 0.8008474576271186

As seens they improved the testing data performance which is a better metric to evaluate the performance of the model. On average with different testing and training sets the model performed better as seen with the stacking classifier. Furthermore, this performance is consistent demonstrated by the low deviaiton in both F1 Score and Mean performance. The stacking classifer makes preidcitons of multiple base classifiers and combines them to make better classifications. Therefore, this improves the predictive perofrmance by combining predictions similar to the way ensembles solved them, and enables generalizations. This improved the base meta-model by minimizing existing error and bias within the model. This is shown as well byt he performance on the testing data being better on the new meta-model than on the old one where it was individual. The perofrmance could be improved by doing more thorough hyperparameter testing, unfortunately my computer could not do more features fast enough. Additionally, we could test more hyperparameters with more values to get a more thorough meta model.

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)**

The stacking classifiers performance can be improved is by testing more values for the hyperparameter. Unfortunately my computer had computational limitations and would take 5 minutes each time which is a lot of time. Furthermore, the more hyperparameters I utilized the worse the computastion was exponentially which meant I couldn't test all the different features such as teh ovr and ovo hyperparameters along with a several for gradient boosting regarding number of minimum samples per leaf and subsection etc. This would lead to improved meta-model and thus a better stacking classifier. This is because we would have a meta-model with more refined hyperparameters. Thus the main model in the classifier will be able to be better increasing performance significantly.

Another feasible way to create a stacking classifier would be to incorporate a wider variety of models such as different ensemble trees already as well as Ridge and Lasso Classficiaiton these are variations of the logistic regression with penalties however, when incorporating them in my model I got several warnings due to incompatible l1 and l2 for other hyperparameters I was using. This would prove even more computationally intensive though. This would improve the model overall by covering all weaknesses and assumptions the current models have and thus would incorporate more algorithms and feature importances similar to tree ensembles that helped reduce bias and assumptions.

Answer: