Assignments IF3170 - Artificial Intelligence

Web Application for classifying hearth disease from data clinic

Phase A - Find Best Model

Group 3 - Unexpected


```
- Rizki Alif Salman Alfarisy / 13516005
- Jonathan Alvaro / 13516023
- Joseph Salimin / 13516037
- Kevin Leonardo Limitius / 13516049
- Kevin Basuki / 13516071
```


First of All.. Import All Library Needed

In [1]:

```
import numpy as np
import pandas as pd
import pickle
import itertools

from sklearn.model_selection import train_test_split, StratifiedKFold, train_test_split, KFold
from sklearn.model_selection import cross_validate
from sklearn import metrics
from sklearn import tree
from sklearn.impute import SimpleImputer

from sklearn import svm, datasets
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, classification_report, accuracy_score,
confusion_matrix

from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

from sklearn.externals import joblib

from pprint import pprint
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

Read Data From CSV

In [2]:

```
heart_train = pd.read_csv('tubes2_HeartDisease_train.csv')
heart_test = pd.read_csv('tubes2_HeartDisease_test.csv')
```

Rename Column Names

In [3]:

```python
from copy import deepcopy

test_columns = {
    'Column1': 'age',
    'Column2': 'sex',
    'Column3': 'chest_pain_type',
    'Column4': 'resting_blood_pressure',
    'Column5': 'serum_cholesterol',
    'Column6': 'fasting_blood_sugar',
    'Column7': 'resting_ecg',
    'Column8': 'max_heart_rate_achieved',
    'Column9': 'exercise_induced_angina',
    'Column10': 'st_depression',
    'Column11': 'peak_exercise_st_segment',
    'Column12': 'num_major_flourosopy',
    'Column13': 'thal'
}
# Create train columns
train_columns = test_columns.copy()
train_columns['Column14'] = 'heart_disease_diagnosis'

# Rename columns
heart_train = heart_train.rename(columns=train_columns)
heart_test = heart_test.rename(columns=test_columns)
```

Data Analysis

In [4]:

```python
print('Column data heart train')
pprint(heart_train.dtypes)
print()
print('Show heart train head')
pprint(heart_train.head())
print()
print('Find sum value undefined in each column')
heart_train.isna().sum()
```

```
Column data heart train
age                          int64
sex                          int64
chest_pain_type              int64
resting_blood_pressure      object
serum_cholesterol           object
fasting_blood_sugar         object
resting_ecg                 object
max_heart_rate_achieved     object
exercise_induced_angina     object
st_depression               object
peak_exercise_st_segment    object
num_major_flourosopy        object
thal                        object
heart_disease_diagnosis      int64
dtype: object

Show heart train head
   age  sex  chest_pain_type resting_blood_pressure serum_cholesterol  \
0   54    1                4                    125               216
1   55    1                4                    158               217
2   54    0                3                    135               304
3   48    0                3                    120               195
4   50    1                4                    120                 0

  fasting_blood_sugar resting_ecg max_heart_rate_achieved  \
0                   0           0                     140
1                   0           0                     110
2                   1           0                     170
3                   0           0                     125
4                   0           1                     156

  exercise_induced_angina st_depression peak_exercise_st_segment  \
0                       0             0                        ?
1                       1           2.5                        2
2                       0             0                        1
```

```
3                        0             0                        ?
4                        1             0                        1

  num_major_flourosopy thal  heart_disease_diagnosis
0                    ?    ?                        1
1                    ?    ?                        1
2                    0    3                        0
3                    ?    ?                        0
4                    ?    6                        3

Find sum value undefined in each column
```

Out[4]:

```
age                         0
sex                         0
chest_pain_type             0
resting_blood_pressure      0
serum_cholesterol           0
fasting_blood_sugar         0
resting_ecg                 1
max_heart_rate_achieved     0
exercise_induced_angina     0
st_depression               0
peak_exercise_st_segment    0
num_major_flourosopy        0
thal                        0
heart_disease_diagnosis     0
dtype: int64
```

From above results, we can conclude that data given from CSV:

1. Not all data is numeric
2. There are some datas which value is '?'
3. There are also some datas which value is undefined (NaN)

Our conclusion:

Results above can disturb the modeling process. Because of that, we need to do some pre-processing to ensure that there are little to no noise in the model.

Dataframe Conversion to Numeric

In [5]:

```python
# Convert string to numeric
heart_train = heart_train.apply(pd.to_numeric, errors = 'coerce')
# Print data
print('Data type of columns after conversion')
print(heart_train.dtypes)
print()
# Show NaN count
print('Total value NaN after heart_train converted to numeric value')
print(heart_train.isna().sum())
```

```
Data type of columns after conversion
age                           int64
sex                           int64
chest_pain_type               int64
resting_blood_pressure      float64
serum_cholesterol           float64
fasting_blood_sugar         float64
resting_ecg                 float64
max_heart_rate_achieved     float64
exercise_induced_angina     float64
st_depression               float64
peak_exercise_st_segment    float64
num_major_flourosopy        float64
thal                        float64
heart_disease_diagnosis       int64
dtype: object

Total value NaN after heart_train converted to numeric value
```

```
Total value NaN after heart_train converted to numeric value
age                          0
sex                          0
chest_pain_type              0
resting_blood_pressure      47
serum_cholesterol           24
fasting_blood_sugar         78
resting_ecg                  2
max_heart_rate_achieved     44
exercise_induced_angina     44
st_depression               49
peak_exercise_st_segment   262
num_major_flourosopy       514
thal                       408
heart_disease_diagnosis      0
dtype: int64
```

Removing NaN Values

In the process above, first, we convert the data from object (string) to numeric value. We succeed in removing object value from dataframe. But there are also a problem, value which can not be convered to numeric data type will be converted to NaN and that problem can make dataframe hard to be processed.

First of all, to reduce noise in data, we try to remove row which has more than 3 NaNs in its attribute columns.

One of the easiest way to remove NaN value is to remove row which contains NaN value in it. But that way is not really good and not feasible, since Column 12 has 514 rows which value is NaN and that means removing 514 rows which will reduce many data trainings. Another way is to replace NaN value with median. We choose median value rather than mean because median value is much more stable thant mean for irregular data (outliers).

For categorical data, for example in Column 3 and Column 7, it is better to replace NaN value with mode, since replacing the valie with median will result in creating meaningless value (not in that category). That's why, for Column 3, 6, 7, 11, and 13, we replace the value with mode.

During the imputation process, in order to ensure that the missing data is not filled in by overgeneralized values, we first separate the training data into 5 different clusters, one for each possible output class. Then, we impute the missing values separately for each cluster before finally the clusters are once again merged.

In [6]:

```python
# Drop row with too many NaNs
heart_train = heart_train.dropna(thresh=10)

# Count NaN value
print('Total NaN Value')
print(heart_train.isna().sum())
```

```
Total NaN Value
age                          0
sex                          0
chest_pain_type              0
resting_blood_pressure       2
serum_cholesterol           21
fasting_blood_sugar         77
resting_ecg                  2
max_heart_rate_achieved      0
exercise_induced_angina      0
st_depression                5
peak_exercise_st_segment   218
num_major_flourosopy       469
thal                       366
heart_disease_diagnosis      0
dtype: int64
```

In [7]:

```python
# Cluster dataset
heart_train_0 = heart_train[heart_train["heart_disease_diagnosis"] == 0].copy()
heart_train_1 = heart_train[heart_train["heart_disease_diagnosis"] == 1].copy()
heart_train_2 = heart_train[heart_train["heart_disease_diagnosis"] == 2].copy()
heart_train_3 = heart_train[heart_train["heart_disease_diagnosis"] == 3].copy()
heart_train_4 = heart_train[heart_train["heart_disease_diagnosis"] == 4].copy()
```

In [8]:

```
def fill_nan_with_value(train):
    train['peak_exercise_st_segment'].fillna(train['peak_exercise_st_segment'].mode()[0], inplace=T
rue)
    train['chest_pain_type'].fillna(train['chest_pain_type'].mode()[0], inplace=True)
    train['resting_ecg'].fillna(train['resting_ecg'].mode()[0], inplace=True)
    train['fasting_blood_sugar'].fillna(train['fasting_blood_sugar'].mode()[0], inplace=True)
    train['thal'].fillna(train['thal'].mode()[0], inplace=True)
    imp = SimpleImputer(missing_values=np.nan, strategy='median')
    c = train.columns
    train = pd.DataFrame(imp.fit_transform(train))
    train.columns = c
    return train
```

In [9]:

```
heart_train_0 = fill_nan_with_value(heart_train_0)
heart_train_1 = fill_nan_with_value(heart_train_1)
heart_train_2 = fill_nan_with_value(heart_train_2)
heart_train_3 = fill_nan_with_value(heart_train_3)
heart_train_4 = fill_nan_with_value(heart_train_4)

heart_train = pd.concat([heart_train_0, heart_train_1, heart_train_2, heart_train_3, heart_train_4]
)

heart_train = heart_train.reset_index()
del heart_train['index']

# Count NaN value
print('Total NaN Value')
print(heart_train.isna().sum())
```

```
Total NaN Value
age                         0
sex                         0
chest_pain_type             0
resting_blood_pressure      0
serum_cholesterol           0
fasting_blood_sugar         0
resting_ecg                 0
max_heart_rate_achieved     0
exercise_induced_angina     0
st_depression               0
peak_exercise_st_segment    0
num_major_flourosopy        0
thal                        0
heart_disease_diagnosis     0
dtype: int64
```

Check Correlation and Remove Unnecessery Columns

Then, we want to check the correlation of the column and the result. The results is shown below.

In [10]:

```
heart_train['age'].corr(heart_train['heart_disease_diagnosis'])
```

Out[10]:

```
0.35731799317243274
```

In [11]:

```
heart_train['sex'].corr(heart_train['heart_disease_diagnosis'])
```

Out[11]:

```
0.2557223264055121
```

In [12]:

```
heart_train['chest_pain_type'].corr(heart_train['heart_disease_diagnosis'])
```

Out[12]:

```
0.39488988172819645
```

In [13]:

```
heart_train['resting_blood_pressure'].corr(heart_train['heart_disease_diagnosis'])
```

Out[13]:

```
0.11710812250912425
```

In [14]:

```
heart_train['serum_cholesterol'].corr(heart_train['heart_disease_diagnosis'])
```

Out[14]:

```
-0.25320518865574626
```

In [15]:

```
heart_train['fasting_blood_sugar'].corr(heart_train['heart_disease_diagnosis'])
```

Out[15]:

```
0.10352085498426744
```

In [16]:

```
heart_train['resting_ecg'].corr(heart_train['heart_disease_diagnosis'])
```

Out[16]:

```
0.14898347335673548
```

In [17]:

```
heart_train['max_heart_rate_achieved'].corr(heart_train['heart_disease_diagnosis'])
```

Out[17]:

```
-0.3662003932557409
```

In [18]:

```
heart_train['exercise_induced_angina'].corr(heart_train['heart_disease_diagnosis'])
```

Out[18]:

```
0.3924863582518776
```

In [19]:

```
heart_train['st_depression'].corr(heart_train['heart_disease_diagnosis'])
```

Out[19]:

```
0.23000199179131123
```

In [20]:

```
heart_train['peak_exercise_st_segment'].corr(heart_train['heart_disease_diagnosis'])
```

Out[20]:

```
0.46570768383508016
```

In [21]:

```
heart_train['thal'].corr(heart_train['heart_disease_diagnosis'])
```

Out[21]:

```
0.5765972104207246
```

Delete Unneccesary Columns

From results above, then we want to delete columns which are not really necessary. The column which has correlation between -0.25 < x < 0.25 will be removed by us.

In [22]:

```
del heart_train['resting_blood_pressure']
del heart_train['fasting_blood_sugar']
del heart_train['resting_ecg']
del heart_train['st_depression']
```

Splitting Data

In [23]:

```
# Split data train
# heart_train_copy = heart_train.copy()
Y = heart_train['heart_disease_diagnosis']
X = heart_train.drop('heart_disease_diagnosis', axis = 1).values

# Best so far
scaler = MinMaxScaler(feature_range=(0,1))
X = scaler.fit_transform(X)
```

In [24]:

```
KF = StratifiedKFold(10, shuffle=True)
```

Find Best Model

From dataset we have pre processed above, then we create model and find which model can give best performance (not just accuracy, but also the behaviour of the model too).

Gaussian Naive-Bayes

In [25]:

```
sum_acc, sum_prec, sum_rec = 0, 0, 0
list_gnb_model = []

for trainidx, testidx in KF.split(X, Y):
    # Create GNB Model and test it
    gnb = GaussianNB()
    X_train, X_test = X[trainidx], X[testidx]
    Y_train, Y_test = Y[trainidx], Y[testidx]
    gnb.fit(X_train,Y_train)
    # Calculate accuraction, precision, and recall value
    accuracy = metrics.accuracy_score(Y_test, gnb.predict(X_test))
    precision = metrics.precision_score(Y_test, gnb.predict(X_test), average="macro")
    recall = metrics.recall_score(Y_test, gnb.predict(X_test), average="macro")
```

```
    # Append it to list
    list_gnb_model.append((gnb, accuracy, precision, recall))
    sum_acc += accuracy
    sum_prec += precision
    sum_rec += recall

# Find best model based on accuracy
best_model_index = 0
for i in range(1, len(list_gnb_model)):
    if list_gnb_model[i][1] > list_gnb_model[best_model_index][1]:
        best_model_index = i
gnb = list_gnb_model[best_model_index][0]

print("Average Accuracy : {0:.4f}".format(sum_acc/10))
print("Average Precision : {0:.4f}".format(sum_prec/10))
print("Average Recall : {0:.4f}".format(sum_rec/10))
```

```
Average Accuracy : 0.7327
Average Precision : 0.4940
Average Recall : 0.5090
```

Decision Tree

In [26]:

```
sum_acc, sum_prec, sum_rec = 0, 0, 0
list_dt_model = []

for trainidx, testidx in KF.split(X, Y):
    # Create Tree Model and test it
    dt = tree.DecisionTreeClassifier()
    X_train, X_test = X[trainidx], X[testidx]
    Y_train, Y_test = Y[trainidx], Y[testidx]
    dt.fit(X_train,Y_train)
    # Calculate accuraction, precision, and recall value
    accuracy = metrics.accuracy_score(Y_test, dt.predict(X_test))
    precision = metrics.precision_score(Y_test, dt.predict(X_test), average="macro")
    recall = metrics.recall_score(Y_test, dt.predict(X_test), average="macro")
    # Append it to list
    list_dt_model.append((dt, accuracy, precision, recall))
    sum_acc += accuracy
    sum_prec += precision
    sum_rec += recall

# Find best model based on accuracy
best_model_index = 0
for i in range(1, len(list_dt_model)):
    if list_dt_model[i][1] > list_dt_model[best_model_index][1]:
        best_model_index = i
dt = list_dt_model[best_model_index][0]

print("Average Accuracy : {0:.4f}".format(sum_acc/10))
print("Average Precision : {0:.4f}".format(sum_prec/10))
print("Average Recall : {0:.4f}".format(sum_rec/10))
```

```
Average Accuracy : 0.6991
Average Precision : 0.5105
Average Recall : 0.4950
```

KNN

In [27]:

```
sum_acc, sum_prec, sum_rec = 0, 0, 0
list_knn_model = []

for trainidx, testidx in KF.split(X, Y):
    # Create KNN Model and test it
    knn = KNeighborsClassifier(n_neighbors=44, weights='distance')
    X_train, X_test = X[trainidx], X[testidx]
    Y_train, Y_test = Y[trainidx], Y[testidx]
```

```python
    knn.fit(X_train,Y_train)
    # Calculate accuraction, precision, and recall value
    accuracy = metrics.accuracy_score(Y_test, knn.predict(X_test))
    precision = metrics.precision_score(Y_test, knn.predict(X_test), average="macro")
    recall = metrics.recall_score(Y_test, knn.predict(X_test), average="macro")
    # Append it to list
    list_knn_model.append((knn, accuracy, precision, recall))
    sum_acc += accuracy
    sum_prec += precision
    sum_rec += recall

# Find best model based on accuracy
best_model_index = 0
for i in range(1, len(list_knn_model)):
    if list_knn_model[i][1] > list_knn_model[best_model_index][1]:
        best_model_index = i
knn = list_knn_model[best_model_index][0]

print("Average Accuracy : {0:.4f}".format(sum_acc/10))
print("Average Precision : {0:.4f}".format(sum_prec/10))
print("Average Recall : {0:.4f}".format(sum_rec/10))
```

```
Average Accuracy : 0.7092
Average Precision : 0.5168
Average Recall : 0.4339
```

MLP Classifier

In [28]:

```python
sum_acc, sum_prec, sum_rec = 0, 0, 0
list_mlp_model = []

for trainidx, testidx in KF.split(X, Y):
    # Create MLP Model and test it
    mlp = MLPClassifier(hidden_layer_sizes=(7), solver='sgd',
                        max_iter=1000, learning_rate_init=0.1, learning_rate='adaptive',
                        activation='identity')
    X_train, X_test = X[trainidx], X[testidx]
    Y_train, Y_test = Y[trainidx], Y[testidx]
    mlp.fit(X_train,Y_train)
    # Calculate accuraction, precision, and recall value
    accuracy = metrics.accuracy_score(Y_test, mlp.predict(X_test))
    precision = metrics.precision_score(Y_test, mlp.predict(X_test), average="macro")
    recall = metrics.recall_score(Y_test, mlp.predict(X_test), average="macro")
    # Append it to list
    list_mlp_model.append((mlp, accuracy, precision, recall))
    sum_acc += accuracy
    sum_prec += precision
    sum_rec += recall

# Find best model based on accuracy
best_model_index = 0
for i in range(1, len(list_mlp_model)):
    if list_mlp_model[i][1] > list_mlp_model[best_model_index][1]:
        best_model_index = i
mlp = list_mlp_model[best_model_index][0]

print("Average Accuracy : {0:.4f}".format(sum_acc/10))
print("Average Precision : {0:.4f}".format(sum_prec/10))
print("Average Recall : {0:.4f}".format(sum_rec/10))
```

```
Average Accuracy : 0.7358
Average Precision : 0.5082
Average Recall : 0.4906
```

Analysis

Out of the 4 models that have been trained above, we decided to use the MLP model for several reasons. First, the simplest reason is that it has the highest accuracy. Then, there is also the fact that some attributes of the data have continuous values which is more suitable to be processed by the MLP model instead of the other three.

Save model

In [29]:

```
joblib.dump(mlp, 'mlp.pkl')
```

Out[29]:

```
['mlp.pkl']
```

Load model

In [30]:

```
mlp = joblib.load('mlp.pkl')
mlp
```

Out[30]:

```
MLPClassifier(activation='identity', alpha=0.0001, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=7, learning_rate='adaptive',
       learning_rate_init=0.1, max_iter=1000, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=None, shuffle=True, solver='sgd', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
```

Pre process Data Test

After saving model, then we also pre process data test with same steps as data train, the results are shown below here.

In [31]:

```
# Convert data string in heart_test to numeric
heart_test = heart_test.apply(pd.to_numeric, errors = 'coerce')
# Print data
print('Data type of columns after conversion')
print(heart_test.dtypes)
print()
# Show NaN count
print('Total value NaN after heart_train converted to numeric value')
print(heart_test.isna().sum())
```

```
Data type of columns after conversion
age                            int64
sex                            int64
chest_pain_type                int64
resting_blood_pressure       float64
serum_cholesterol            float64
fasting_blood_sugar          float64
resting_ecg                    int64
max_heart_rate_achieved      float64
exercise_induced_angina      float64
st_depression                float64
peak_exercise_st_segment     float64
num_major_flourosopy         float64
thal                         float64
dtype: object

Total value NaN after heart_train converted to numeric value
age                           0
sex                           0
chest_pain_type               0
resting_blood_pressure       12
serum_cholesterol             6
fasting_blood_sugar          12
resting_ecg                   0
max_heart_rate_achieved      11
```

```
max_heart_rate_achieved      11
exercise_induced_angina      11
st_depression                13
peak_exercise_st_segment     47
num_major_flourosopy         97
thal                         78
dtype: int64
```

In [32]:

```
# Fill NaN Value for test is same for data train
# Mode
heart_test['peak_exercise_st_segment'].fillna(heart_test['peak_exercise_st_segment'].mode()[0],
inplace=True)
heart_test['chest_pain_type'].fillna(heart_test['chest_pain_type'].mode()[0], inplace=True)
heart_test['resting_ecg'].fillna(heart_test['resting_ecg'].mode()[0], inplace=True)
heart_test['fasting_blood_sugar'].fillna(heart_test['fasting_blood_sugar'].mode()[0], inplace=True
)
heart_test['thal'].fillna(heart_test['thal'].mode()[0], inplace=True)
# Median
imp = SimpleImputer(missing_values=np.nan, strategy='median')
c = heart_test.columns
heart_test = pd.DataFrame(imp.fit_transform(heart_test))
heart_test.columns = c
# Remove unnecessary column
del heart_test['resting_blood_pressure']
del heart_test['fasting_blood_sugar']
del heart_test['resting_ecg']
del heart_test['st_depression']
# Scaler
# Best so far
scaler_test = MinMaxScaler(feature_range=(0,1))
heart_test = scaler_test.fit_transform(heart_test)
```

Predict

Then we try to predict data test with our chosen model.

In [33]:

```
mlp.predict(heart_test)
```

Out[33]:

```
array([0., 1., 1., 1., 0., 0., 3., 1., 1., 0., 1., 1., 0., 1., 0., 3., 1.,
       1., 1., 1., 0., 2., 0., 1., 1., 0., 1., 1., 0., 0., 0., 1., 2., 0.,
       1., 0., 0., 0., 0., 0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 0., 1., 1., 1., 0., 1., 0.,
       0., 1., 1., 0., 1., 1., 0., 1., 1., 0., 0., 1., 0., 3., 0., 1., 0.,
       1., 0., 0., 1., 1., 3., 0., 0., 0., 0., 1., 0., 1., 1., 0., 2., 0.,
       0., 0., 1., 0., 0., 1., 1., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0.,
       0., 0., 1., 0., 1., 1., 0., 3., 1., 1., 0., 0., 1., 1., 0., 1., 3.,
       0., 1., 0., 1., 0.])
```

The End

Thank You!!