AML Lab Sessional 1 Jupyter Notebook

1. VS Code is a MUST

1. You should code in VS Code. Google Colab not allowed

2. Pre-requisites for lab in VS Code

1. Install PDF export support https://saturncloud.io/blog/how-to-export-jupyter-notebook-by-vscode-in-pdf-format/

3. At the end, export your notebook as html and ipynb

1. Open VS Code command palette Shift + Ctrl + P
2. Type "Export Jupyter Notebook" in the search bar and select "Export Jupyter Notebook to html"
3. Upload your html AND ipynb here: https://tinyurl.com/y9ptbej2 (In the prompt, put your name correctly else your submission will be rejected)

4. Lab Sessional Summary

1. A code template is given to you in lab sesional in this jupyter notebook
2. The template will follow a linear sequence of TODOs appropriately labelled with question marks
3. You will have to fill the TODO question marks to compile those notebook cells and proceed to next cells
4. You can think of the linear sequence of TODOs as a guided thought process.
5. Sessional is open book. Google, github, browse product documentation ChatGPT - Do anything you want, except copying others. No discussing among yourselves (Sending questions to your seniors and seeking answers is prohibited). If you are caught carrying out these illegal activities, you will be reported for immediate action.
6. You CANNOT replace the TODO code template with some other code copied from stack overflow, ChatGPT etc. All your browsing and search should give you insights into finally how you can fit that into the framework I provide for the thought process of solving the problem. You will have to mandatorily fill the question marks and proceed with the lab problem.

In [ ]:
```
import numpy as np
import pandas as pd
import sklearn as sk

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
import warnings
warnings.filterwarnings("ignore")
```

In [ ]:
```
sk.__version__
```

Out[ ]:
```
'1.3.0'
```

Problem Definition

You are provided with diabetes dataset. Your task is:

1. Part 1: To perform EDA and prediction without pipeline
2. Part 2: To move EDA and kNN prediction into a pipeline
3. Part 3: To combine the pipeline and gridsearch to get best K and weights hyperparameters
4. Part 4: Save the model as json file, load & use it to do prediction. This part is optional and will be used to accumulate bonus points as a buffer for the semester lab exam if there is any shortfall

Part 1 - Perform EDA and prediction without pipeline

In [ ]:
```
# TODO: 1 Load the csv file
df = pd.read_csv('d2.csv')
df.head()
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	2	138	62	35	0	33.6	0.127	47
1	0	84	82	31	125	38.2	0.233	23
2	0	145	0	0	0	44.2	0.630	31
3	0	135	68	42	250	42.3	0.365	24
4	1	139	62	41	480	40.7	0.536	21

In [ ]:
```
# TODO: 2 Display information about dataframe
df.info(verbose=True, show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               2000 non-null   int64
 1   Glucose                   2000 non-null   int64
 2   BloodPressure             2000 non-null   int64
 3   SkinThickness             2000 non-null   int64
 4   Insulin                   2000 non-null   int64
 5   BMI                       2000 non-null   float64
 6   DiabetesPedigreeFunction  2000 non-null   float64
 7   Age                       2000 non-null   int64
 8   Outcome                   2000 non-null   int64
dtypes: float64(2), int64(7)
memory usage: 140.8 KB
```

In [ ]:
```
# TODO: 3 Check if there are nulls and display their total for each feature
df.isna().sum()
```

Out[ ]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

In [ ]:
```
# TODO: 4 Check for duplicates
dupsSeries = df.duplicated() # Should return a pandas Series with True False for every

# Print the number of duplicates
print(f"Number of duplicates = {dupsSeries.sum()}")
```

```
Number of duplicates = 1256
```

In [ ]:
```
# TODO: 5 Drop duplicates inplace
df.drop_duplicates(inplace=True)
```

In [ ]:
```
# TODO: 6 Display how many rows exist in dataframe after dropping duplicates
df.info(verbose=True, show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 744 entries, 0 to 1568
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               744 non-null    int64
 1   Glucose                   744 non-null    int64
 2   BloodPressure             744 non-null    int64
 3   SkinThickness             744 non-null    int64
 4   Insulin                   744 non-null    int64
 5   BMI                       744 non-null    float64
 6   DiabetesPedigreeFunction  744 non-null    float64
 7   Age                       744 non-null    int64
 8   Outcome                   744 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 58.1 KB
```

In [ ]:
```
# TODO: 7 Identify candidate features for encoding by seeing how many distinct values
df.nunique()
```

Out[ ]:
```
Pregnancies                 17
Glucose                    136
BloodPressure               47
SkinThickness               53
Insulin                    182
BMI                        247
DiabetesPedigreeFunction   505
Age                         52
Outcome                      2
dtype: int64
```

In [ ]:
```
# TODO: 8
# Display all possible values that the column named "Pregnancies" has in this datafram
df["Pregnancies"].unique()

# Based on different possible values of Pregnancies, would you choose this feature to
# State your reason as 1 sentence comment here
# Since the values are already integer values from 0-17, I would not encode it.
```

Out[ ]:
```
array([ 2,  0,  1,  4,  8,  3,  6,  5,  7, 10, 12,  9, 11, 13, 15, 17, 14],
      dtype=int64)
```

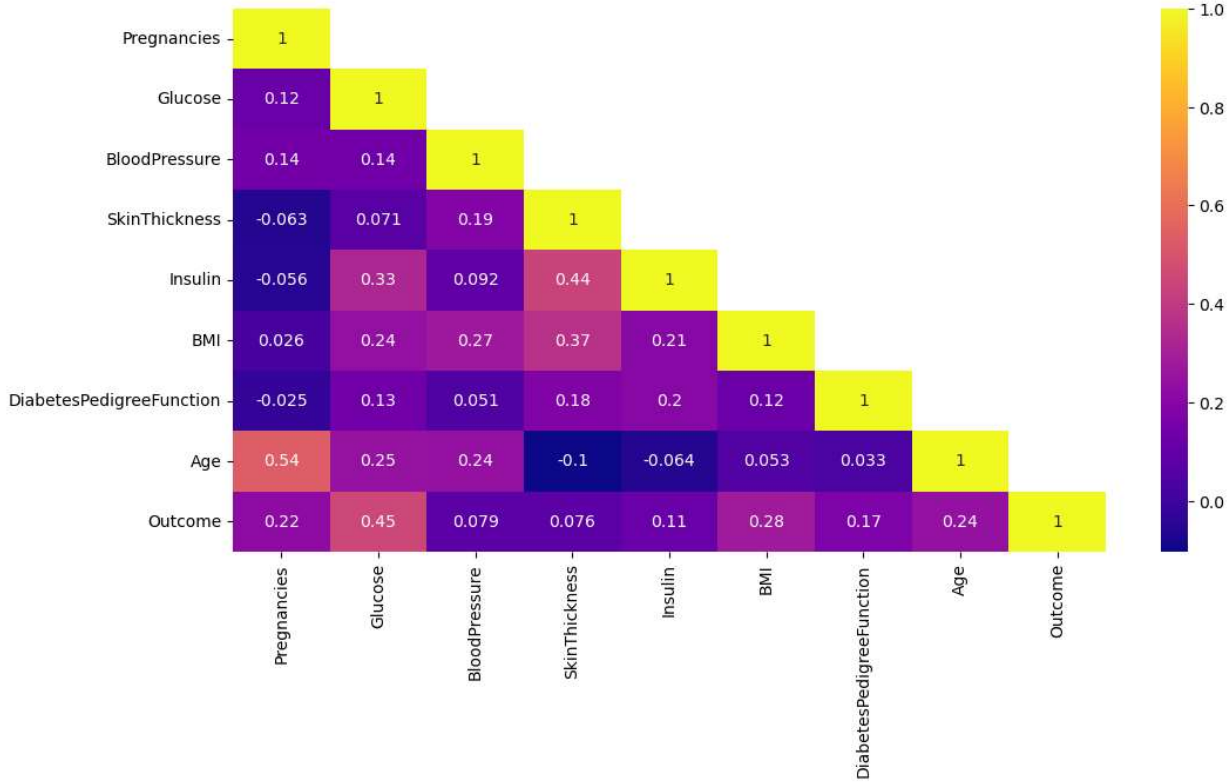
In [ ]:
```
# TODO: 9 Find if this is an dataset is imbalanced wrt target variable classes
zeroClassCount = df[df["Outcome"] == 0]["Outcome"].count()
zeroClassCount/len(df)
```

Out[ ]:
```
0.6599462365591398
```

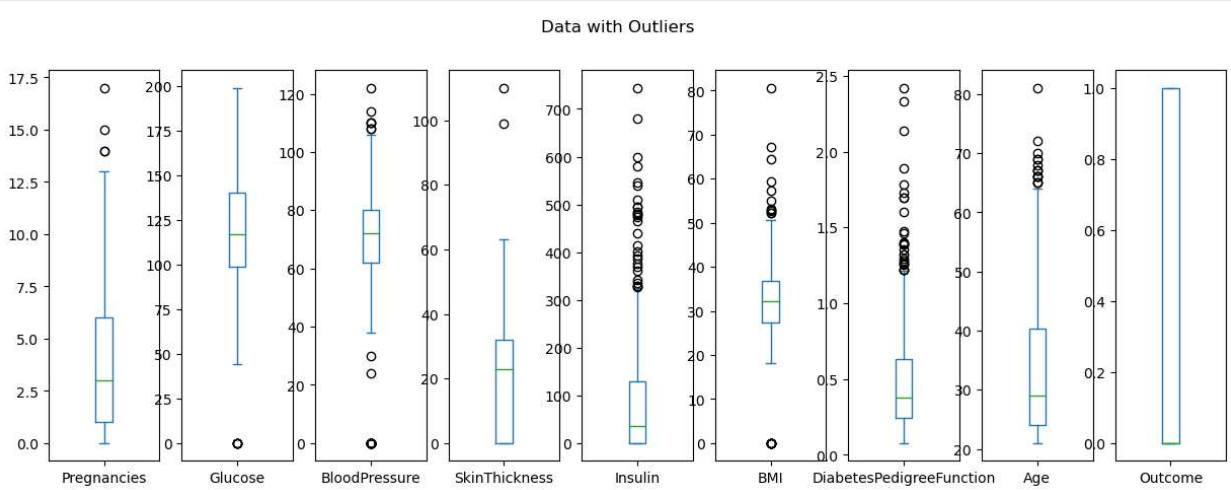
In [ ]:
```
zeroClassCount = df[df["Outcome"] == 1]["Outcome"].count()
zeroClassCount/len(df)
```

Out[ ]:
```
0.3400537634408602
```

In [ ]:
```
# TODO: 10 Run this cell to display correlation matrix heat map
# Would you consider dropping any feature based on their correlation?
# Type your answers here: features that have low correlation with target variable can
plt.figure(figsize=(12,6))
sns.heatmap(df.corr(),cmap='plasma',fmt='.2g',annot=True,mask=np.triu(df.corr(),+1))
plt.show()
```



In [ ]:
```
# TODO: 11 Run this cell to see box plots.
# Visually examine the box plot and answer this question
# How do you identify the 1.5 IQR boundary?
# Which two features have most outliers beyond 1.5 IQR?
# (Increase the figure size if you want to see larger image)
# Put your answer for both questions here
# The horizontal line at the top calculated as 1.5 * IQR, points falling beyond these
# Lower Bound: (Q1 - 1.5 * IQR)
# Upper Bound: (Q3 + 1.5 * IQR)
# Insulin and DiabetesPedigreeFunction has the most outliers beyond the IQR range

df.plot(kind="box",subplots=True,figsize=(15,5),title="Data with Outliers");
```



In [ ]:
```
# TODO: 12 Did you see any nulls in the dataset?
# Ans: no
```

In [ ]:
```
# But look at how many features have zeros in them
# Display zeroes in each column as percentage
```

```
for col in df.columns:
    count = (df[col] == 0).sum()
    percentage = (count * 100)/df.shape[0]
    print(f'Count of zeros in Column {col} : {count}, percentage 0s: {percentage:.2f}%
```

```
Count of zeros in Column Pregnancies : 112, percentage 0s: 15.05%
Count of zeros in Column Glucose : 5, percentage 0s: 0.67%
Count of zeros in Column BloodPressure : 34, percentage 0s: 4.57%
Count of zeros in Column SkinThickness : 215, percentage 0s: 28.90%
Count of zeros in Column Insulin : 359, percentage 0s: 48.25%
Count of zeros in Column BMI : 10, percentage 0s: 1.34%
Count of zeros in Column DiabetesPedigreeFunction : 0, percentage 0s: 0.00%
Count of zeros in Column Age : 0, percentage 0s: 0.00%
Count of zeros in Column Outcome : 491, percentage 0s: 65.99%
```

1. After executing the above cell, you found many cells contain 0.
2. Some cells should never contain 0. For e.g. Glucose
3. Identify the cells that should never contain 0 and replace those 0 with Nan

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [ ]:
```
# TODO: 13 Replace 0 with Nan for features that should never contain 0
# Choose 5 columns you want to replace 0 with Nan
df[['Glucose','BloodPressure','SkinThickness','Insulin','BMI']] = \
    df[['Glucose','BloodPressure','SkinThickness','Insulin','BMI']].replace(0, np.NaN)
```

In [ ]:
```
# TODO: 14  Now do null check again to ensure the right featues have Nans. Otherwise g

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

Out[ ]:
```
Pregnancies                   0
Glucose                       5
BloodPressure                34
SkinThickness               215
Insulin                     359
BMI                          10
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

In [ ]:
```
y = df.pop("Outcome") #Setup target variable
type(y)
```

Out[ ]:
```
pandas.core.series.Series
```

In [ ]:
```
# TODO: 15

# Answer these questions with short one liner right in this cell as a comment
#
# 1. What does the pop operation do?
# It deletes the column that is provided as argument and returns it
#
# 2. Is pop() idempotent or non-idempotent operation?
```

```
#    non-idempotent operation
#
# 3. How do you check if an operation is idempotent/non-idempotent?
#    if applying same operation multiple times has same effect then it is idempotent e
#
```

In [ ]:
```
X = df # Setup independent variables
```

In [ ]:
```
from sklearn.model_selection import train_test_split

# Fill this out
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state
```

In [ ]:
```
# TODO: 16 Answer these questions
# 1. Did you use stratify?
#    yes

# 2. If yes, on which column and why? If not, why not?
#    column:y, this makes sure there is equal representation of datapoints of both cla
#    in train and test (the % of points belong to each class will be same in train and
#
```

In [ ]:
```
# TODO: 17 Display all records that have at least one Nan column value
# If you cannot get this, you can leave this cell execution and proceed
# Subsequent cells do not depend on this. If you cannot solve this, you can proceed to

nan_mask = X_train.isna().any(axis=1)
X_train.loc[nan_mask, :]
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Ag
616	6	117.0	96.0	NaN	NaN	28.7	0.157	3
48	7	103.0	66.0	32.0	NaN	39.1	0.344	3
200	0	113.0	80.0	16.0	NaN	31.0	0.874	2
101	1	151.0	60.0	NaN	NaN	26.1	0.179	2
615	3	106.0	72.0	NaN	NaN	25.8	0.207	2
...	...	...	...	...	...	...	...	
276	7	106.0	60.0	24.0	NaN	26.5	0.296	2
687	1	107.0	50.0	19.0	NaN	28.3	0.181	2
60	2	84.0	NaN	NaN	NaN	NaN	0.304	2
404	5	168.0	64.0	NaN	NaN	32.9	0.135	4
636	5	104.0	74.0	NaN	NaN	28.8	0.153	4

270 rows × 8 columns

Deciding on Imputation

1. Now you should decide which imputation to use - whether SimpleImputer or IterativeImputer
2. Instead of following MCAR/MAR path to decide on SimpleImputer or IterativeImputer, you will take a simple alternative
3. Use IterativeImputer if a large number of rows have Nan for a column
4. Use SimpleImputer if the number of Nans is very small

In [ ]:
```
print(f"X_train.shape={X_train.shape}")
X_train.isna().sum()
```

Out[ ]:
```
X_train.shape=(558, 8)
Pregnancies                   0
Glucose                       2
BloodPressure                26
SkinThickness               159
Insulin                     267
BMI                          10
DiabetesPedigreeFunction      0
Age                           0
dtype: int64
```

In [ ]:
```
# TODO 18 Based on the results of previous cell which columns will you apply SimpleImp
# Select only those columns here

X_train_mean_impute = X_train.loc[:,['Glucose','BloodPressure','BMI']]
X_train_mean_impute
```

Out[ ]:

	Glucose	BloodPressure	BMI
616	117.0	96.0	28.7
318	115.0	66.0	38.1
25	130.0	60.0	28.6
328	102.0	86.0	45.5
760	88.0	58.0	28.4
...	...	...	...
462	74.0	70.0	35.3
390	100.0	66.0	32.0
307	137.0	68.0	24.8
636	104.0	74.0	28.8
765	121.0	72.0	26.2

558 rows × 3 columns

In [ ]:
```
# TODO: 19 Apply SimpleImputer to appropriate columns only

from sklearn.impute import SimpleImputer
mean_imputer = SimpleImputer(strategy='mean')

#Fill code above to create SimpleImputer
```

```
X_train_mean_imputed = mean_imputer.fit_transform(X_train_mean_impute)

# Display first few
X_train_mean_imputed[0:5,:]
```

Out[ ]:
```
array([[117. ,  96. ,  28.7],
       [115. ,  66. ,  38.1],
       [130. ,  60. ,  28.6],
       [102. ,  86. ,  45.5],
       [ 88. ,  58. ,  28.4]])
```

In [ ]:
```
# TODO: 20 Which columns will you apply IterativeImputer ?
# Name those columns and provide 2 line short reason
# Skinthickness and insulin columns are imputed using IterativeImputer as they have lo
# or when there is a complex relationship between missing values across multiple colum

X_train.loc[:,['SkinThickness','Insulin']]
```

Out[ ]:

	SkinThickness	Insulin
616	NaN	NaN
318	39.0	140.0
25	23.0	170.0
328	36.0	120.0
760	26.0	16.0
...	...	...
462	40.0	49.0
390	29.0	196.0
307	14.0	148.0
636	NaN	NaN
765	23.0	112.0

558 rows × 2 columns

Iterative Imputer uses all non null features of the dataset to impute. To apply Iteratative Imputer on the two columns identified above, you should have all other features combined into the dataframe or numpy nmatrix first.

You should do the following for that

1. Remove the columns of the dataframe that were subjected to mean imputation
2. Convert the remaining dataframe into numpy matrix
3. Combine the above numpy matrix with mean imputed columns earlier. The "combining" two matrices is achieved by concatenating those two numpy matrices - by using an appropriate numpy stacking function
4. The stacked numpy matrix is then used for iterative imputation

In [ ]:
```
# TODO: 21 Remove the columns of X_train dataframe that were already subjected to mear
# Hold on to the rest as a Numpy matrix
X_train_set_aside = X_train.drop(['Glucose','BloodPressure','BMI'], axis=1).to_numpy()
X_train_set_aside
```

Out[ ]:
```
array([[6.00e+00,      nan,      nan, 1.57e-01, 3.00e+01],
       [3.00e+00, 3.90e+01, 1.40e+02, 1.50e-01, 2.80e+01],
       [1.00e+00, 2.30e+01, 1.70e+02, 6.92e-01, 2.10e+01],
       ...,
       [0.00e+00, 1.40e+01, 1.48e+02, 1.43e-01, 2.10e+01],
       [5.00e+00,      nan,      nan, 1.53e-01, 4.80e+01],
       [5.00e+00, 2.30e+01, 1.12e+02, 2.45e-01, 3.00e+01]])
```

In [ ]:
```
# TODO: 22 Concatenate the two numpy matrices by using an appropriate numpy stacking f
# Identify which are those two numpy matrices first and use them

X_train = np.hstack([X_train_mean_imputed, X_train_iter_imputed]) # this numpy matrix
X_train
```

Out[ ]:
```
array([[1.17000000e+02, 9.60000000e+01, 2.87000000e+01, ...,
        1.49392823e+02, 1.57000000e-01, 3.00000000e+01],
       [1.15000000e+02, 6.60000000e+01, 3.81000000e+01, ...,
        1.40000000e+02, 1.50000000e-01, 2.80000000e+01],
       [1.30000000e+02, 6.00000000e+01, 2.86000000e+01, ...,
        1.70000000e+02, 6.92000000e-01, 2.10000000e+01],
       ...,
       [1.37000000e+02, 6.80000000e+01, 2.48000000e+01, ...,
        1.48000000e+02, 1.43000000e-01, 2.10000000e+01],
       [1.04000000e+02, 7.40000000e+01, 2.88000000e+01, ...,
        1.81687731e+02, 1.53000000e-01, 4.80000000e+01],
       [1.21000000e+02, 7.20000000e+01, 2.62000000e+01, ...,
        1.12000000e+02, 2.45000000e-01, 3.00000000e+01]])
```

In [ ]:
```
# TODO: 23 Perform Iterative Imputation on two columns with Nan data
# At this point only two columns will have Nan - SkinThickness and Insulin
# Those 2 will be imputed using all other columns
# But to make life easy, this is equivalent to performing Iterative Imputation using a
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
imputer = IterativeImputer(max_iter=10, random_state=0)
imputer.fit(X_train_set_aside)

# Transform the data to impute missing values
X_train_iter_imputed = imputer.transform(X_train_set_aside)
X_train_iter_imputed # this variable should hold your imputer values
```

Out[ ]:
```
array([[6.00000000e+00, 2.91498069e+01, 1.49392823e+02, 1.57000000e-01,
        3.00000000e+01],
       [3.00000000e+00, 3.90000000e+01, 1.40000000e+02, 1.50000000e-01,
        2.80000000e+01],
       [1.00000000e+00, 2.30000000e+01, 1.70000000e+02, 6.92000000e-01,
        2.10000000e+01],
       ...,
       [0.00000000e+00, 1.40000000e+01, 1.48000000e+02, 1.43000000e-01,
        2.10000000e+01],
       [5.00000000e+00, 3.10892705e+01, 1.81687731e+02, 1.53000000e-01,
        4.80000000e+01],
       [5.00000000e+00, 2.30000000e+01, 1.12000000e+02, 2.45000000e-01,
        3.00000000e+01]])
```

In [ ]:
```
# TODO: 24 Fill the question marks to do z transform

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled
```

Out[ ]:
```
array([[-0.12039277,  2.00900571, -0.57304878, ..., -0.06431185,
        -0.96883242, -0.2482146 ],
       [-0.18563486, -0.52707996,  0.68698831, ..., -0.18007374,
        -0.99082191, -0.42442801],
       [ 0.3036808 , -1.03429709, -0.58645343, ...,  0.1896614 ,
         0.71179295, -1.04117496],
       ...,
       [ 0.5320281 , -0.35800758, -1.09583013, ..., -0.08147771,
        -1.0128114 , -1.04117496],
       [-0.54446634,  0.14920955, -0.55964413, ...,  0.33370689,
        -0.98139784,  1.33770614],
       [ 0.0100914 , -0.01986283, -0.90816503, ..., -0.52515988,
        -0.69239311, -0.2482146 ]])
```

In [ ]:
```
# TODO: 25 Train the KNN model in this cell

from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train_scaled, y_train)
```

Out[ ]:
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

In [ ]:
```
# TODO: 26  Perform all necessary transformation on X_test in this cell to prepare it
# look at all the transformation logic that was performed on X_train and repeat it for
# Put your code here. No template is provided for this section

print(X_test.isna().sum())
X_test_mean_impute = X_test.loc[:,['Glucose','BloodPressure']]
X_test_mean_imputed = mean_imputer.fit_transform(X_test_mean_impute)
X_test_set_aside = X_test.drop(['Glucose','BloodPressure'], axis=1).to_numpy()
imputer.fit(X_test_set_aside)
X_test_iter_imputed = imputer.transform(X_test_set_aside)
X_test = np.hstack([X_test_mean_imputed, X_test_iter_imputed])
X_test_scaled = scaler.fit_transform(X_test)
```

```
Pregnancies                 0
Glucose                     3
BloodPressure               8
SkinThickness              56
Insulin                    92
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
dtype: int64
```

In [ ]:
```
X_train.shape
```

Out[ ]:
```
(558, 8)
```

In [ ]:
```
# TODO: 27

# Write code here to predict using the model and calculate & display the accuracy of y
# You should get an accuracy of around 75%

from sklearn.metrics import accuracy_score

# Do prediction using knn here
y_pred = model.predict(X_test_scaled)

accuracy_score(y_pred=y_pred, y_true=y_test)
```

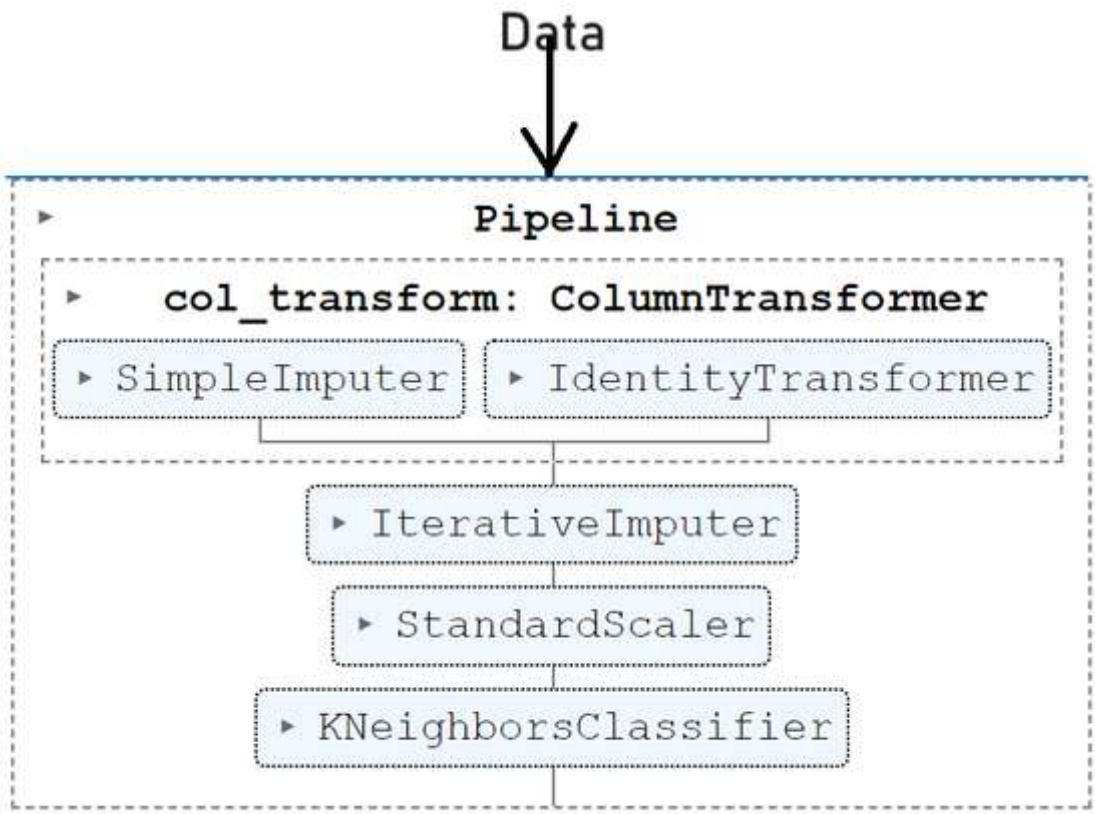
Out[ ]:
```
0.7311827956989247
```

Part 2: Data processing and prediction using a pipeline

In [ ]:
```
# TODO: 28 Repeat activities that need to be done outside pipeline
# Reading csv, dropping duplicates & other steps that you might have performed - but r

df = pd.read_csv('d2.csv')
df.drop_duplicates(inplace=True)
#Add other code for dropping duplicates etc
```

In [ ]:
```
# TODO: 29 Split the data into X and y. Then do train test split here

y = df.pop("Outcome")
X = df

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state
```

Design a pipeline that looks like following:



1. SimpleImputer performs mean imputation only on the relevant columns
2. IdentityTransformer (whose code is given below) is a custom sklearn transformer that performs no operation on the input data. Input data is passed through as is
3. The two are wrapped within a ColumnTransformer
4. This is followed by sequential execution of IterativeImputer, StandaScaler and the KNN based clasifier

In [ ]:
```
# This is a No-op transformer. Use this as is
from sklearn.base import BaseEstimator, TransformerMixin

class IdentityTransformer(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X*1
```

Look at the columns of the dataframe and decide which ones need to go to SimpleImputer and which ones should go to IdentityTransformer

In [ ]:
```
df.columns
```

Out[ ]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age'],
      dtype='object')
```

In [ ]:
```
# TODO: 30 Select the columns for mean imputation
# Do not type the names directly. Use indices
mean_impute_cols = df.columns[[1,2,5]].to_list()
mean_impute_cols
```

Out[ ]:
```
['Glucose', 'BloodPressure', 'BMI']
```

In [ ]:
```
mean_impute_col_mask = df.columns.isin(mean_impute_cols)
mean_impute_col_mask
```

Out[ ]:
```
array([False,  True,  True, False, False,  True, False, False])
```

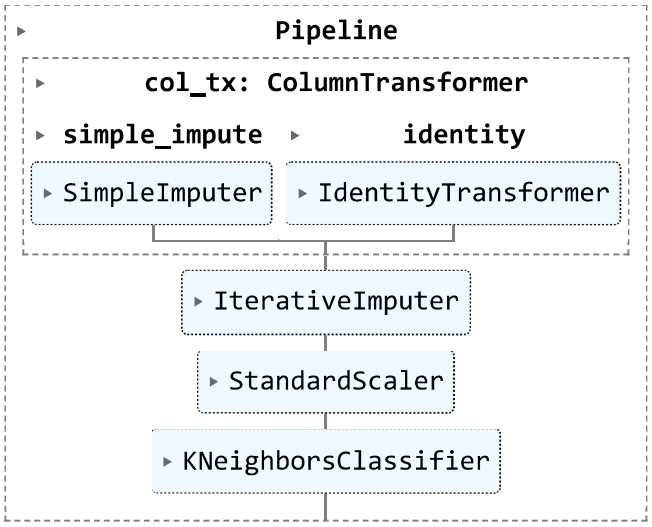
In [ ]:
```
# TODO: 31 Use the mean_impute_col_mask to select remaining columns from df.columns to
other_cols = df.columns[mean_impute_col_mask].to_list()
other_cols = ['Pregnancies','SkinThickness','Insulin','DiabetesPedigreeFunction','Age'
other_cols
```

Out[ ]:
```
['Pregnancies', 'SkinThickness', 'Insulin', 'DiabetesPedigreeFunction', 'Age']
```

In [ ]:
```
# TODO 32 Build the pipeline using the diagram shown earlier and the details provided

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer



preprocessing_pipeline = ColumnTransformer([
    ("simple_impute", SimpleImputer(strategy="mean"), mean_impute_cols),
    ("identity", IdentityTransformer(), other_cols),
])

pipeline = Pipeline([
    ('col_tx', preprocessing_pipeline),
    ('iterativeimputer', IterativeImputer()),
    ('standard_scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])
```

In [ ]:
```
# TODO: 33 Run this cell to verify if indeed your pipeline looks like what was expecte

from sklearn import set_config
set_config(display='diagram')
pipeline  # Put your pipeline variable name here to display pipeline as diagram
```

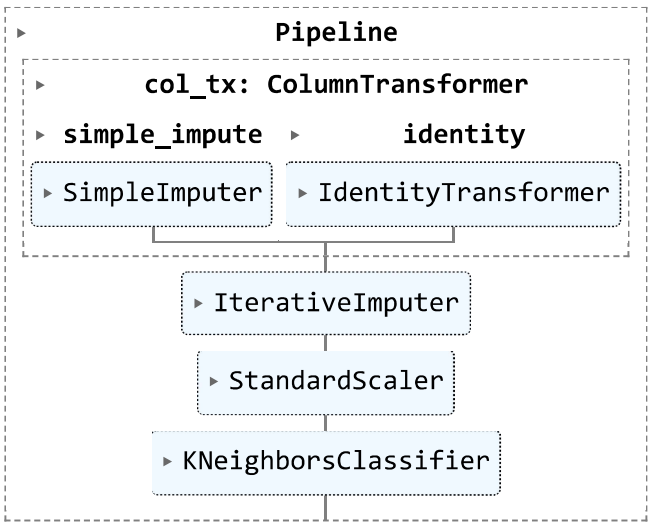
Out[ ]:



In [ ]:

```
# TODO: 34 Train pipeline with data
pipeline.fit(X_train, y_train)
```

Out[ ]:



In [ ]:

```
# TODO: 35 Do predictions with pipeline and calculate the accuracy score
pipeline.score(X_test, y_test)
```

Out[ ]:

```
0.6774193548387096
```

Part 3: Combine pipeline & gridsearch for hyperparam tuning

In [ ]:

```
# Define KFold for cross-validation

# Define hyperparameters and their possible values

# Create a GridSearchCV object


# Fit the GridSearchCV object to your data
grid_search.fit(X_train, y_train)
```

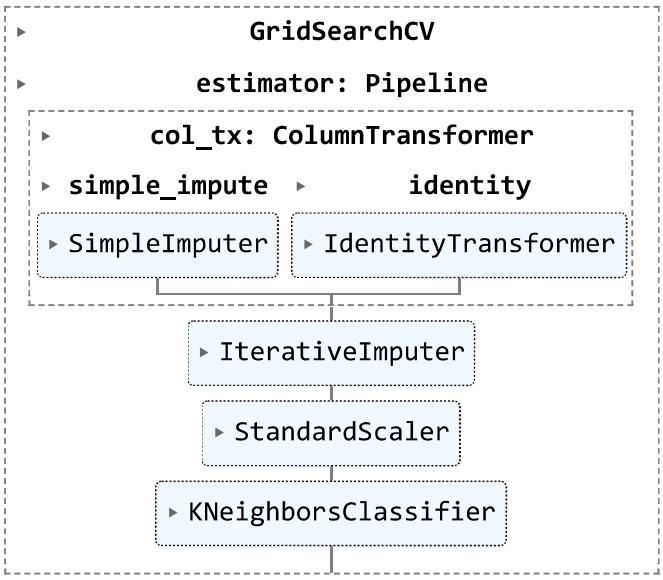
```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=4, knn__weights=distance; total time
```

```
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=euclidean, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=2, knn__weights=distance; total time
```

```
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=4, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance; total time
```

```
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=uniform; total time=
0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
[CV] END knn__metric=manhattan, knn__n_neighbors=6, knn__weights=distance; total time
=   0.0s
```

Out[ ]:



In [ ]:

```python
# TODO: 36. Use GridSearchCV with KFold = 3 for CV and train a model using the pipelir

# Create KFold here
from sklearn.model_selection import GridSearchCV, KFold

kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Define hyperparameters
grid_params = {
    'knn__n_neighbors': [2, 3, 4, 5, 6],
    'knn__weights': ['uniform', 'distance'],
    'knn__metric': ['euclidean', 'manhattan']
}


#Create Grid SearchCV object here using KFold object and gridparams
grid = GridSearchCV(estimator=pipeline,param_grid=grid_params,  cv=kfold)
```

```
#Do hyperparameter tuning here with GridSearchCV object
fitted_model = grid.fit(X_train, y_train)
```

In [ ]:
```
# TODO: 37 Display the results for best score, best estimator and best params selected
# fitted_model is the result of hyperpaarameter tuning in the previous cell
print(fitted_model.best_score_)
print(fitted_model.best_estimator_)
print(fitted_model.best_params_)
```

```
0.7616312741312742
Pipeline(steps=[('col_tx',
                 ColumnTransformer(transformers=[('simple_impute',
                                                  SimpleImputer(),
                                                  ['Glucose', 'BloodPressure',
                                                   'BMI']),
                                                 ('identity',
                                                  IdentityTransformer(),
                                                  ['Pregnancies',
                                                   'SkinThickness', 'Insulin',
                                                   'DiabetesPedigreeFunction',
                                                   'Age'])])),
                ('iterativeimputer', IterativeImputer()),
                ('standard_scaler', StandardScaler()),
                ('knn',
                 KNeighborsClassifier(metric='euclidean', n_neighbors=6,
                                      weights='distance'))])
{'knn__metric': 'euclidean', 'knn__n_neighbors': 6, 'knn__weights': 'distance'}
```

In [ ]:
```
# TODO: 38 Do predictions with gridsearch and calculate the accuracy score

best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy Score:", accuracy)
```

```
Accuracy Score: 0.6881720430107527
```

Confusion matrix calculation

The next 3 cells are meant for providing hints to calculating the confusion matrix

In [ ]:
```
# TODO: 39 Put your pipeline name in place of question mark
pipeline.named_steps
```

Out[ ]:
```
{'col_tx': ColumnTransformer(transformers=[('simple_impute', SimpleImputer(),
                                            ['Glucose', 'BloodPressure', 'BMI']),
                                           ('identity', IdentityTransformer(),
                                            ['Pregnancies', 'SkinThickness', 'Insulin',
                                             'DiabetesPedigreeFunction', 'Age'])]),
 'iterativeimputer': IterativeImputer(),
 'standard_scaler': StandardScaler(),
 'knn': KNeighborsClassifier()}
```

In [ ]:
```
# TODO: 40 Put your pipeline name in place of first question mark
# and appropriate value for second question mark to display the model
pipeline.named_steps['knn']
```

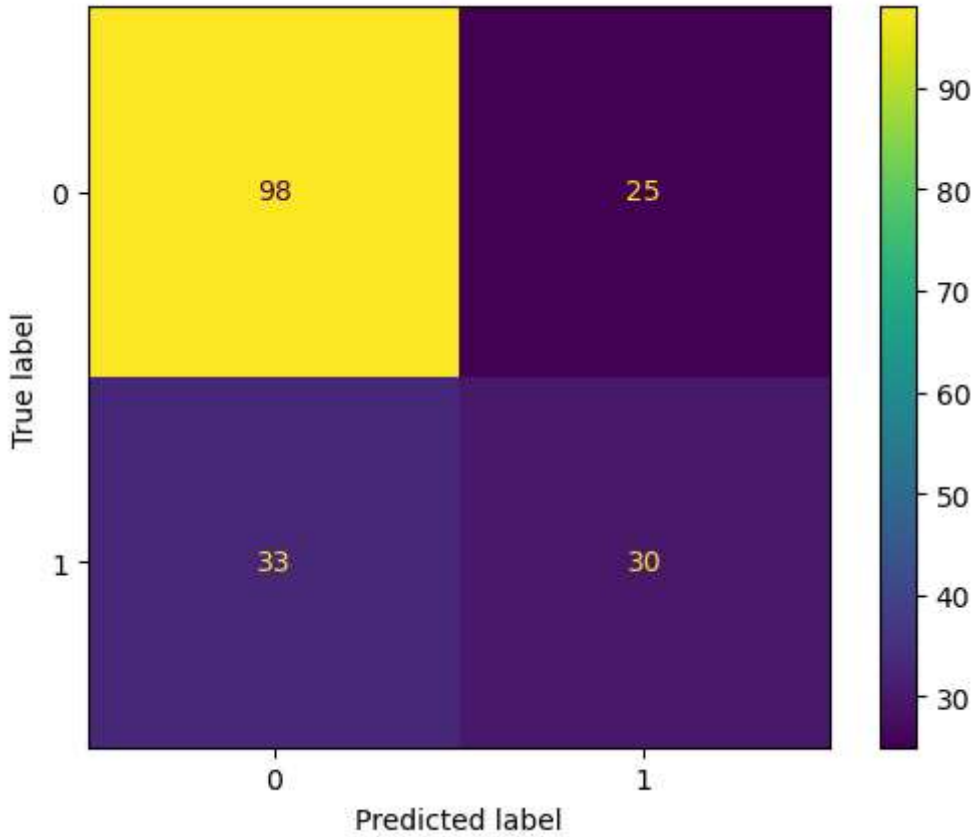
Out[ ]:
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

In [ ]:
```
# TODO: 41 Put your pipeline name in place of first question mark
# and appropriate value for second question mark to display the model
vars(pipeline.named_steps['knn'])
```

Out[ ]:
```
{'n_neighbors': 5,
 'radius': None,
 'algorithm': 'auto',
 'leaf_size': 30,
 'metric': 'minkowski',
 'metric_params': None,
 'p': 2,
 'n_jobs': None,
 'weights': 'uniform',
 'n_features_in_': 8,
 'outputs_2d_': False,
 'classes_': array([0, 1], dtype=int64),
 '_y': array([0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
        1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0,
        0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0,
        1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
        0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
        0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1,
        0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1,
        0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0,
        0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
        1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1,
        0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,
        0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
        0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1,
        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0,
        1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1,
        1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0,
        0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,
        1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
        0, 0, 1, 0, 0, 0, 0, 0], dtype=int64),
 'effective_metric_params_': {},
 'effective_metric_': 'euclidean',
 '_fit_method': 'kd_tree',
 '_fit_X': array([[-0.10345647,  1.40717811, -0.42599312, ..., -0.71800328,
         -0.96883242, -0.2482146 ],
        [-0.1669644 , -0.1488138 ,  0.6609452 , ...,  0.56067306,
         -0.99082191, -0.42442801],
        [ 0.30934509, -0.46001218, -0.43755629, ...,  0.83467513,
          0.71179295, -1.04117496],
        ...,
        [ 0.53162285, -0.04508101, -0.87695689, ...,  0.63374028,
         -1.0128114 , -1.04117496],
        [-0.51625803,  0.26611737, -0.41442995, ..., -0.71800328,
         -0.98139784,  1.33770614],
        [ 0.02355939,  0.16238458, -0.71507246, ...,  0.30493779,
         -0.69239311, -0.2482146 ]]),
 'n_samples_fit_': 558,
 '_tree': <sklearn.neighbors._kd_tree.KDTree at 0x12d4e44e040>}
```

In [ ]:
```python
# TODO: 42 Write code to display a confusion matrix by filling the question marks
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
```

```
cm = confusion_matrix(y_test, y_pred, labels=best_model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_model.classes_)
disp.plot()
plt.show()
```



In [ ]:
```
# TODO: 43 Interpret the confusion matrix in your own words
# True Postives = 30
# False Positives = 25
# True Negatives = 98
# False Negatives = 33
```

Part 4: Save the model as json file, load & use it to do prediction

This is optional and will be used to accumulate bonus points as a buffer for the semester lab exam if there is any shortfall

In [ ]:
```
import joblib

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

Out[ ]:
```
['best_model.pkl']
```

In [ ]:
```
loaded_model = joblib.load('best_model.pkl')
y_pred = loaded_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy Score:", accuracy)
```

```
Accuracy Score: 0.6881720430107527
```