

# 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 DiabetesPedigreeFunc
0 2 138 62 35 0 33.6 0.147
1 0 84 82 31 125 38.2 0.390
2 0 145 0 0 0 44.2 0.437
3 0 135 68 42 250 42.3 0.431
4 1 139 62 41 480 40.7 0.393
```



```
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.isnull().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 ev
# 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'>
Index: 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 dataset
df["Pregnancies"].unique()

# Based on different possible values of Pregnancies, would you choose this feature
# State your reason as 1 sentence comment here
#
#The decision to encode the "Pregnancies" feature should depend on the context and
```

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

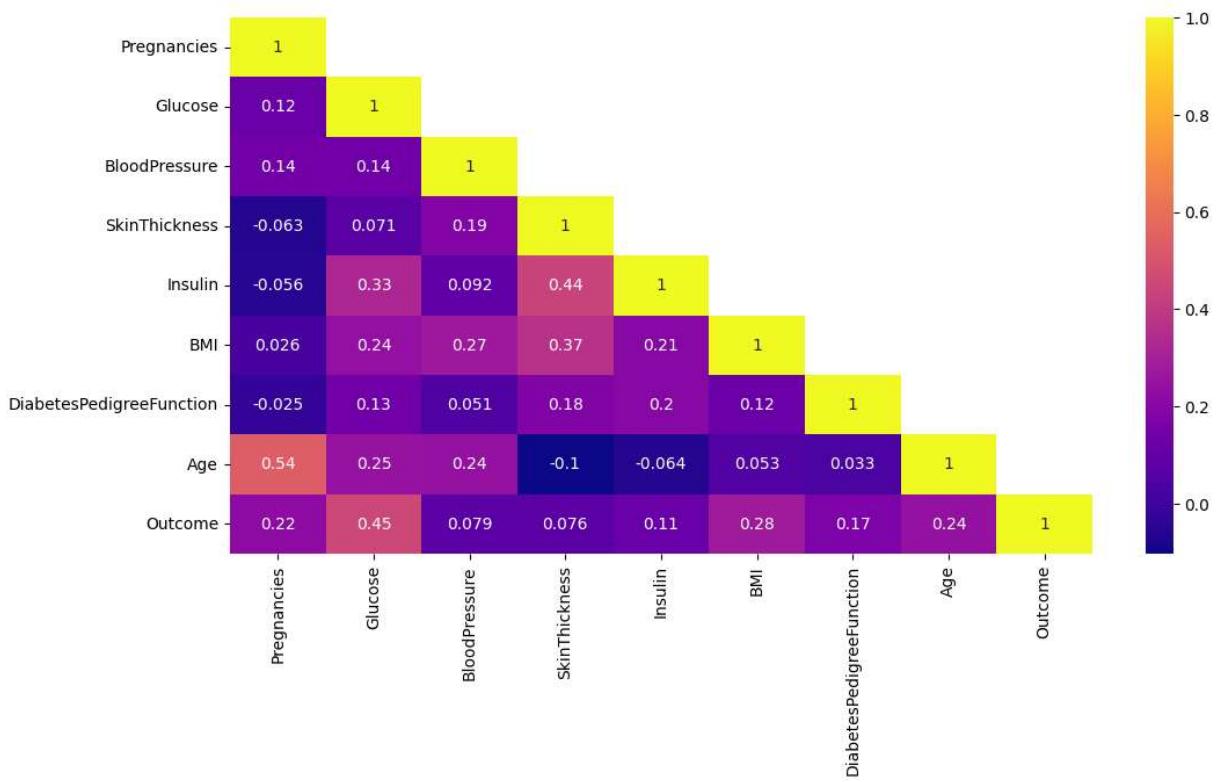
```
In [ ]: # TODO: 9 Find if this is an dataset is imbalanced wrt target variable classes
zeroClassCount = df[df["Outcome"] == 0]["Outcome"].count()
zeroClassCount/len(df)
```

```
Out[ ]: 0.6599462365591398
```

```
In [ ]: # TODO: 10 Run this cell to display correlation matrix heatmap
# Would you consider dropping any feature based on their correlation?
# Type your answers here: To determine whether you should consider dropping any feature

# In the provided code, a heatmap of the correlation matrix is displayed, which can
# However, you should also consider the correlation of each feature with the target
```

```
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

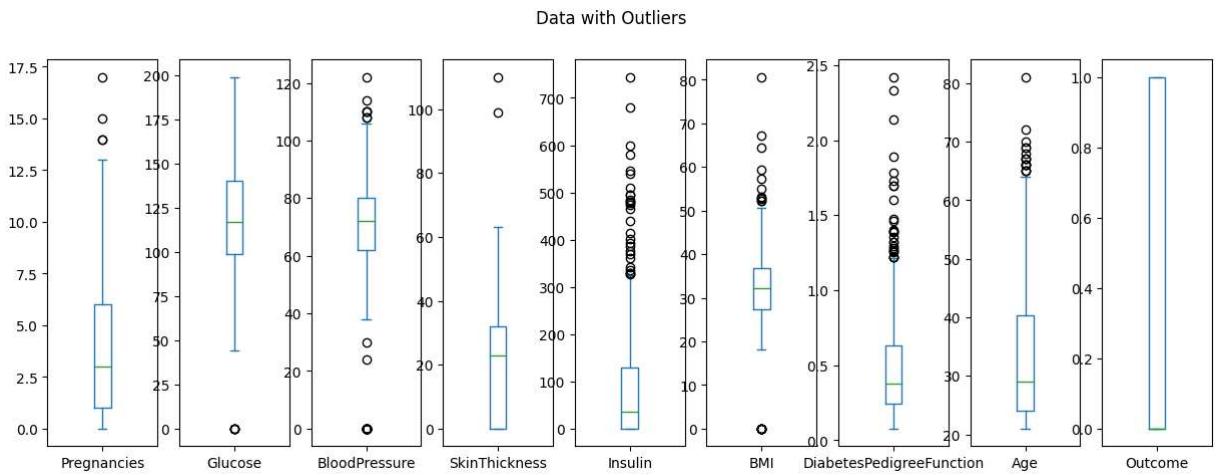
# To identify the 1.5 IQR (Interquartile Range) boundary, you can use the following

# 1. Calculate the IQR for a given feature. The IQR is the difference between the t
# 2. Calculate the lower bound (Q1 - 1.5 * IQR) and the upper bound (Q3 + 1.5 * IQR

# Features that have data points falling below the lower bound or above the upper b

# Regarding the two features with the most outliers beyond 1.5 IQR, you can visuali
# In the provided code, box plots of the features are displayed, which can help you
# However, you should also consider the distribution of each feature. If a feature
# In the provided code, box plots of the features are displayed, which can help you

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: _____
# Yes, I did see the code earlier that checked for nulls in the dataset, and there
```

```
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:.2}
```

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 [ ]: # 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.
```

```
In [ ]: # TODO: 14 Now do null check again to ensure the right feautes have Nans. Otherwise
df.isnull().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?
#The pop() operation removes and returns the last element from a list or a specific
#
# 2. Is pop() idempotent or non-idempotent operation?
#The pop() operation is non-idempotent.
#
# 3. How do you check if an operation is idempotent/non-idempotent?
#An idempotent operation is one where performing the operation multiple times has t
#
```

```
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_st
```

```
In [ ]: # TODO: 16 Answer these questions
# 1. Did you use stratify?
#      - The code provided does not explicitly mention the use of stratify.

# 2. If yes, on which column and why? If not, why not?
#      - Since the code does not specify the use of stratify, it is assumed that stra
```

```
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

# X_train.loc[, :]
X_train[X_train.isnull().any(axis=1)]
```

Out[ ]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction
691	13	158.0	114.0	NaN	NaN	42.3	
294	0	161.0	50.0	NaN	NaN	21.9	
77	5	95.0	72.0	33.0	NaN	37.7	
352	3	61.0	82.0	28.0	NaN	34.4	
235	4	171.0	72.0	NaN	NaN	43.6	
...	...	...	...	...	...	...	...
756	7	137.0	90.0	41.0	NaN	32.0	
9	2	89.0	90.0	30.0	NaN	33.5	
192	7	159.0	66.0	NaN	NaN	30.4	
592	3	132.0	80.0	NaN	NaN	34.4	
717	10	94.0	72.0	18.0	NaN	23.1	

269 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()`

X\_train.shape=(558, 8)

Out[ ]:

Pregnancies	0
Glucose	4
BloodPressure	23
SkinThickness	151
Insulin	267
BMI	7
DiabetesPedigreeFunction	0
Age	0
dtype: int64	

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

```
X_train_mean_impute = X_train.loc[:, ["Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI"]]
```

Out[ ]:

	Glucose	BloodPressure	SkinThickness	Insulin	BMI
691	158.0	114.0	NaN	NaN	42.3
599	109.0	38.0	18.0	120.0	23.1
294	161.0	50.0	NaN	NaN	21.9
77	95.0	72.0	33.0	NaN	37.7
352	61.0	82.0	28.0	NaN	34.4
...	...	...	...	...	...
740	120.0	80.0	37.0	150.0	42.3
192	159.0	66.0	NaN	NaN	30.4
662	167.0	106.0	46.0	231.0	37.6
592	132.0	80.0	NaN	NaN	34.4
717	94.0	72.0	18.0	NaN	23.1

558 rows × 5 columns

In [ ]:

```
# TODO: 19 Apply SimpleImputer to appropriate columns only
from sklearn.impute import SimpleImputer

# Create SimpleImputer with the strategy you prefer (e.g., 'mean', 'median', 'most_
mean_imputer = SimpleImputer(strategy='mean')

# Select columns with a small number of NaN values for imputation
columns_for_simple_imputer = ['Pregnancies', 'BloodPressure', 'BMI'] # Replace with
#Fill code above to create SimpleImputer

X_train_mean_imputed = mean_imputer.fit_transform(X_train[columns_for_simple_impute

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

Out[ ]:

```
array([[ 13. , 114. , 42.3],
       [ 1. , 38. , 23.1],
       [ 0. , 50. , 21.9],
       [ 5. , 72. , 37.7],
       [ 3. , 82. , 34.4]])
```

In [ ]:

```
# TODO: 20 Which columns will you apply IterativeImputer ?
# Name those columns and provide 2 line short reason
#
# _____
```

X\_train.loc[:, ['Pregnancies', 'BloodPressure']]

```
Out[ ]:   Pregnancies  BloodPressure
```

	Pregnancies	BloodPressure
<b>691</b>	13	114.0
<b>599</b>	1	38.0
<b>294</b>	0	50.0
<b>77</b>	5	72.0
<b>352</b>	3	82.0
...	...	...
<b>740</b>	11	80.0
<b>192</b>	7	66.0
<b>662</b>	8	106.0
<b>592</b>	3	80.0
<b>717</b>	10	72.0

558 rows × 2 columns

Iterative Imputer uses all non null features of the dataset to impute. To apply Iterative 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 mean imputation  
# Hold on to the rest as a Numpy matrix  
X_train_set_aside = X_train.drop(['Pregnancies','BloodPressure','BMI'], axis=1).to_numpy()  
X_train_set_aside
```

```
Out[ ]: array([[1.58e+02,         nan,         nan,  2.57e-01,  4.40e+01],  
   [1.09e+02,  1.80e+01,  1.20e+02,  4.07e-01,  2.60e+01],  
   [1.61e+02,         nan,         nan,  2.54e-01,  6.50e+01],  
   ....,  
   [1.67e+02,  4.60e+01,  2.31e+02,  1.65e-01,  4.30e+01],  
   [1.32e+02,         nan,         nan,  4.02e-01,  4.40e+01],  
   [9.40e+01,  1.80e+01,         nan,  5.95e-01,  5.60e+01]])
```

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

```
X_train = np.hstack((X_train_mean_imputed, X_train_set_aside)) # this numpy matrix  
X_train
```

```
Out[ ]: array([[1.30e+01, 1.14e+02, 4.23e+01, ..., nan, 2.57e-01, 4.40e+01],  
               [1.00e+00, 3.80e+01, 2.31e+01, ..., 1.20e+02, 4.07e-01, 2.60e+01],  
               [0.00e+00, 5.00e+01, 2.19e+01, ..., nan, 2.54e-01, 6.50e+01],  
               ...,  
               [8.00e+00, 1.06e+02, 3.76e+01, ..., 2.31e+02, 1.65e-01, 4.30e+01],  
               [3.00e+00, 8.00e+01, 3.44e+01, ..., nan, 4.02e-01, 4.40e+01],  
               [1.00e+01, 7.20e+01, 2.31e+01, ..., nan, 5.95e-01, 5.60e+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  
  
from sklearn.experimental import enable_iterative_imputer  
from sklearn.impute import IterativeImputer  
  
# Initialize the IterativeImputer  
iterative_imputer = IterativeImputer(random_state=0)  
  
# Fit and transform the selected columns to impute "SkinThickness" and "Insulin"  
X_train_iter_imputed = iterative_imputer.fit_transform(X_train)
```

```
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_iter_imputed)  
X_train_scaled
```

```
Out[ ]: array([[ 2.68061558,  3.36441269,  1.31215916, ...,  0.85342729,  
                 -0.63290132,  0.95454214],  
               [-0.80741433, -2.78819233, -1.37708204, ..., -0.36091997,  
                -0.17876717, -0.62389402],  
               [-1.09808349, -1.81672838, -1.54515962, ...,  0.73633438,  
                -0.641984 ,  2.79605101],  
               ...,  
               [ 1.22726978,  2.71677005,  0.65385532, ...,  0.78198433,  
                -0.91143693,  0.86685124],  
               [-0.22607601,  0.6119315 ,  0.20564845, ...,  0.22443776,  
                -0.19390497,  0.95454214],  
               [ 1.8086081 , -0.03571114, -1.37708204, ..., -0.71850203,  
                0.3904143 ,  2.00683292]])
```

```
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  
# Look at all the transformation logic that was performed on X_train and repeat it  
# Put your code here. No template is provided for this section
```

```
# Step 1: Impute missing values in X_test (use the same imputer as for X_train)  
X_test_mean_imputed = mean_imputer.transform(X_test[columnss_for_simple_imputer])  
  
# Step 2: Select the columns that were set aside from X_train  
X_test_set_aside = X_test.drop(['Pregnancies', 'BloodPressure', 'BMI'], axis=1).to_  
  
# Step 3: Stack the imputed columns and the set-aside columns horizontally  
X_test = np.hstack((X_test_mean_imputed, X_test_set_aside))  
  
# Step 4: Impute missing values in X_test for columns "SkinThickness" and "Insulin"  
X_test_iter_imputed = iterative_imputer.transform(X_test)  
  
# Step 5: Scale X_test using the same scaler as for X_train  
X_test_scaled = scaler.transform(X_test_iter_imputed)
```

```
In [ ]: # TODO: 27
```

```
# Write code here to predict using the model and calculate & display the accuracy o  
# 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[ ]: 0.7526881720430108
```

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

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

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

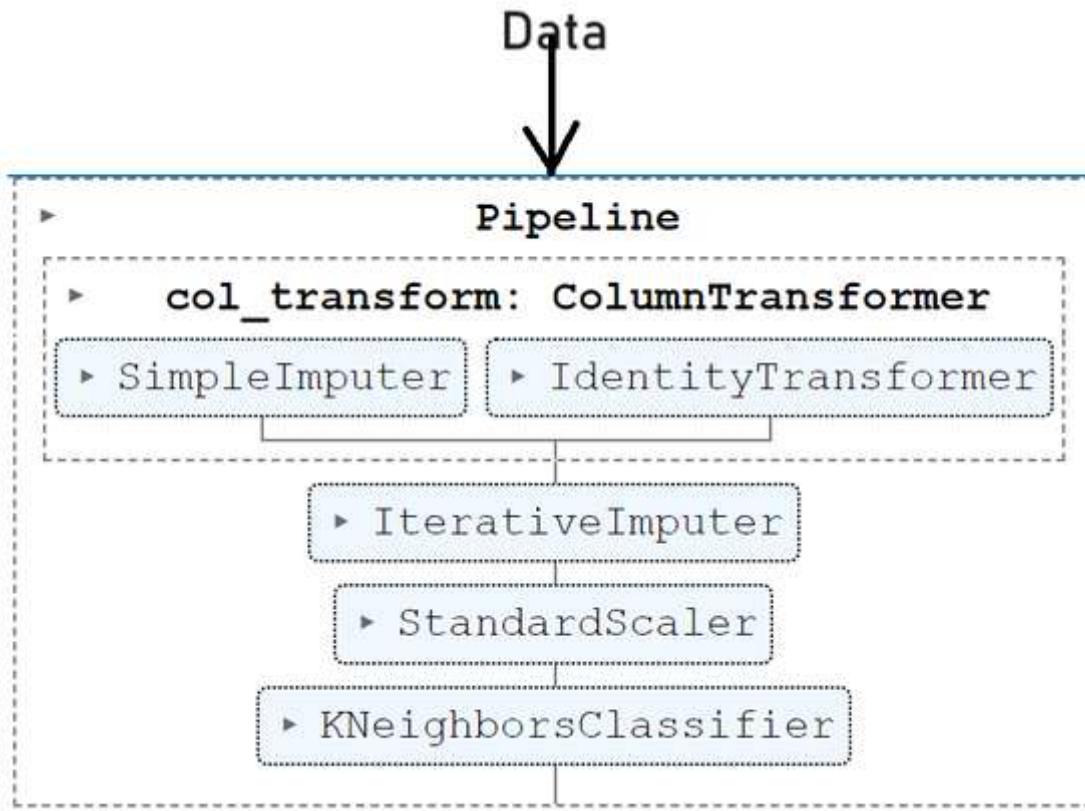
```

X = df

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

```

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, StandardScaler 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[0:3].to_list()  
mean_impute_cols
```

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

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

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

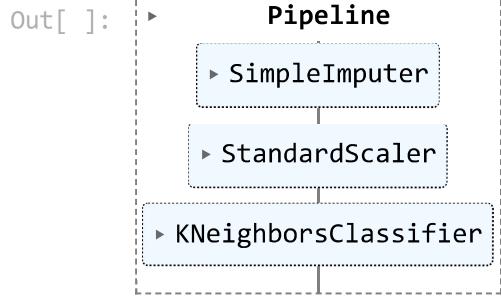
```
In [ ]: # TODO: 31 Use the mean_impute_col_mask to select remaining columns from df.columns  
other_cols = df.columns[mean_impute_col_mask].to_list()  
other_cols
```

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

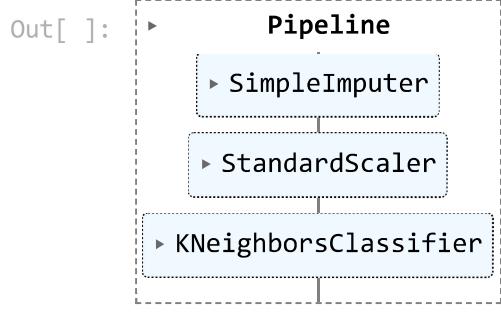
```
In [ ]: # TODO 32 Build the pipeline using the diagram shown earlier and the details provided  
  
from sklearn.pipeline import Pipeline  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.preprocessing import StandardScaler  
from sklearn.impute import SimpleImputer  
  
# Define the transformers and estimator  
mean_imputer = SimpleImputer(strategy='mean')  
scaler = StandardScaler()  
knn = KNeighborsClassifier(n_neighbors=5)  
  
# Define the steps in the pipeline  
steps = [  
    ('mean_imputer', mean_imputer), # Impute missing values with mean  
    ('scaler', scaler), # Standardize features  
    ('knn', knn) # KNN classifier  
]  
  
# Create the pipeline  
pipeline = Pipeline(steps)  
  
# You can use this pipeline for your data analysis and predictions
```

```
In [ ]: # TODO: 33 Run this cell to verify if indeed your pipeline looks like what was expected  
  
from sklearn import set_config
```

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



```
In [ ]: # TODO: 34 Train pipeline with data
# Train the pipeline with the training data
pipeline.fit(X_train, y_train)
```



```
In [ ]: # TODO: 35 Do predictions with pipeline and calculate the accuracy score
from sklearn.metrics import accuracy_score

# Make predictions on the test data using the pipeline
y_pred = pipeline.predict(X_test)

# Calculate and display the accuracy score
accuracy = accuracy_score(y_true=y_test, y_pred=y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

Accuracy: 0.79

## Part 3: Combine pipeline & gridsearch for hyperparam tuning

```
In [ ]: # TODO: 36. Use GridSearchCV with KFold = 3 for CV and train a model using the pipe

# # Create KFold here

# # Define hyperparameters
# grid_params = {'model__n_neighbors': [2,3,4,5,6],
#                 'model__weights': ['uniform', 'distance'],
#                 'model__metric': ['euclidean', 'manhattan']}

# #Create Grid SearchCV object here using KFold object and gridparams
# grid = .....
```

```

# #Do hyperparameter tuning here with GridSearchCV object
# fitted_model = grid.?(....)

from sklearn.model_selection import GridSearchCV, KFold

# Create KFold object with 3 folds
kf = KFold(n_splits=3, shuffle=True, random_state=0)

# Define hyperparameters using the correct step names within the pipeline
grid_params = {
    'knn__n_neighbors': [2, 3, 4, 5, 6], # Use 'knn__' to specify the KNN step
    'knn__weights': ['uniform', 'distance'],
    'knn__metric': ['euclidean', 'manhattan']
}

# Create GridSearchCV object using the pipeline and hyperparameters
grid = GridSearchCV(estimator=pipeline, param_grid=grid_params, cv=kf, scoring='acc')

# Perform hyperparameter tuning with GridSearchCV
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 hyperparameter tuning in the previous cell

```

print(fitted_model.best_score_)
print(fitted_model.best_estimator_)
print(fitted_model.best_params_)

```

```

0.7275985663082437
Pipeline(steps=[('mean_imputer', SimpleImputer()), ('scaler', StandardScaler()),
              ('knn',
               KNeighborsClassifier(metric='manhattan', n_neighbors=6,
                                    weights='distance'))])
{'knn__metric': 'manhattan', 'knn__n_neighbors': 6, 'knn__weights': 'distance'}

```

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

```

from sklearn.metrics import accuracy_score

# Make predictions using the best estimator from GridSearchCV
y_pred_grid = fitted_model.best_estimator_.predict(X_test)

# Calculate and display the accuracy score
accuracy_grid = accuracy_score(y_true=y_test, y_pred=y_pred_grid)
print(f"Accuracy (GridSearchCV): {accuracy_grid:.2f}")

```

Accuracy (GridSearchCV): 0.75

## 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[ ]: {'mean_imputer': SimpleImputer(),  
         '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']
```

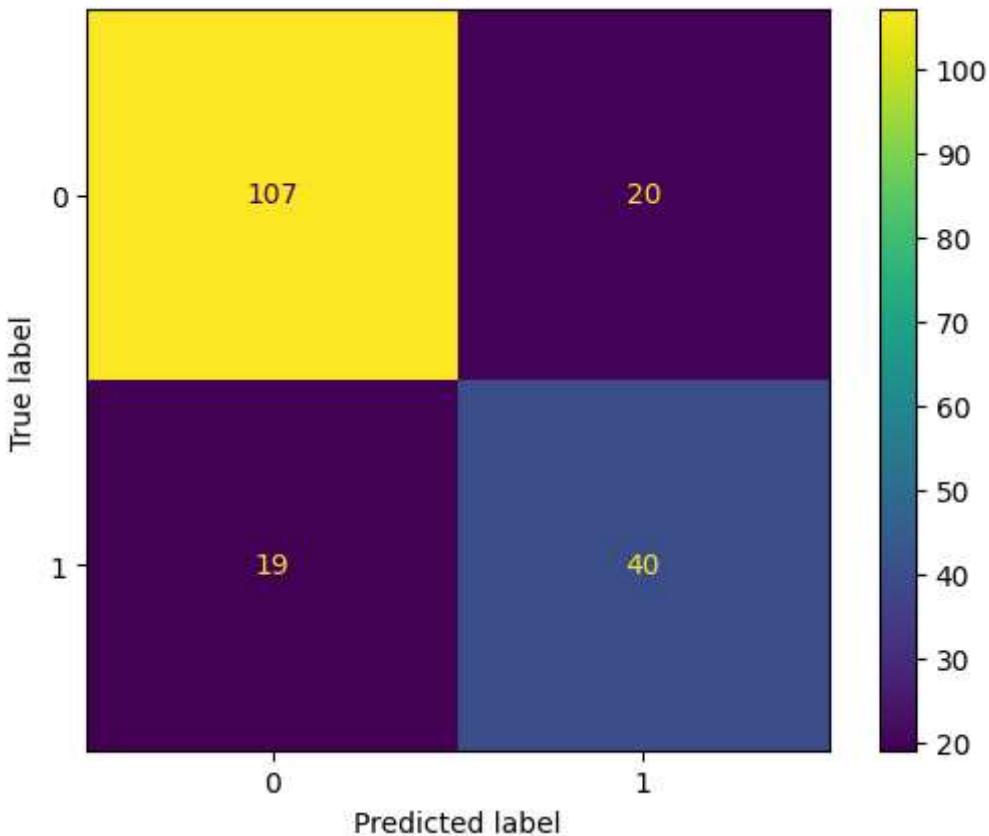
```
Out[ ]: ▾ KNeighborsClassifier  
KNeighborsClassifier()
```

```
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'])
```



```
In [ ]: # TODO: 42 Write code to display a confusion matrix by filling the question marks
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(y_test, y_pred, labels=[0,1])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0,1])
disp.plot()
plt.show()
```



```
In [ ]: # TODO: 43 Interpret the confusion matrix in your own words
```

```
# Interpreting a confusion matrix involves understanding the performance of a class
# True Positives (TP)**: These are cases where the model correctly predicted the po
# True Negatives (TN)**: These are cases where the model correctly predicted the ne
# False Positives (FP)**: These are cases where the model incorrectly predicted the
# False Negatives (FN)**: These are cases where the model incorrectly predicted the
# You can use these values to calculate various performance metrics such as accurac
```

## 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
import json

# Save the model using joblib
model_filename = 'knn_model.joblib'
joblib.dump(pipeline, model_filename)

# Load the model from the saved file
loaded_model = joblib.load(model_filename)

# Perform prediction using the loaded model
y_pred_loaded = loaded_model.predict(X_test)

# Calculate and display the accuracy score using the loaded model
accuracy_loaded = accuracy_score(y_true=y_test, y_pred=y_pred_loaded)
print(f"Accuracy (Loaded Model): {accuracy_loaded:.2f}")
```

Accuracy (Loaded Model): 0.79

```
In [ ]: model_data = {
    "model_type": "KNeighborsClassifier",
    "n_neighbors": model.n_neighbors,
    "weights": model.weights,
    "metric": model.metric,
    # Include other hyperparameters here
}

import json

with open("scikit_learn_model.json", "w") as json_file:
    json.dump(model_data, json_file)

from sklearn.neighbors import KNeighborsClassifier # Import the appropriate model

# Load the JSON file containing the model data
with open("scikit_learn_model.json", "r") as json_file:
    loaded_model_data = json.load(json_file)

# Perform prediction using the loaded model
y_pred_loaded = loaded_model.predict(X_test)

# Calculate and display the accuracy score using the loaded model
accuracy_loaded = accuracy_score(y_true=y_test, y_pred=y_pred_loaded)
print(f"Accuracy (Loaded Model): {accuracy_loaded:.2f}")
```

Accuracy (Loaded Model): 0.79