Capstone Project: Health Care - NIDDK Dataset

Problem Statement:

1) NIDDK (National Institute of Diabetes and Digestive and Kidney Diseases) research creates knowledge about and treatments for the most chronic, costly, and consequential diseases.

2) The dataset used in this project is originally from NIDDK. The objective is to predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset.

3) Build a model to accurately predict whether the patients in the dataset have diabetes or not.

In [ ]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import classification_report

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

In [ ]:

```
# Import the required data file in pandas dataframe

diabetes_data = pd.read_csv(
    "D:\\Data Science\\Capstone Project\\NIDDK Project\\health care diabetes.csv"
)
diabetes_data.head()
```

In [ ]:

```
#checking the shape of the dataframe

diabetes_data.shape
```

In [ ]:

```
# checking data type of the dataframe and Finding out the null values in data

diabetes_data.info()
```

In [ ]:

```
diabetes_data.columns
```

In [ ]:

```
# Changing 0 values in Insulin column as NaN value

zero_to_null = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
diabetes_data[zero_to_null] = diabetes_data[zero_to_null].replace(0, np.nan)
diabetes_data.head()
```

In [ ]:
```
# Display information about the DataFrame

diabetes_data.info()
```

In [ ]:
```
# Count the number of missing values in each column of the diabetes_data DataFrame

diabetes_data.isna().sum()
```

In [ ]:
```
# Generate descriptive statistics for the diabetes_data DataFrame and transpose the

diabetes_data.describe().transpose()
```

In [ ]:
```
# Plot histograms for the selected columns: 'Age', 'Insulin', 'Glucose', 'BloodPress


diabetes_data[[
    'Age', 'Insulin', 'Glucose', 'BloodPressure', 'SkinThickness', 'BMI'
]].hist(figsize=(10, 10))
plt.tight_layout()
plt.show()
```

Insulin data showed high left skewed, and Insulin values also depends on Age group. Hence, NaN values in Insulin is filled based on age group.

In [ ]:
```
# Create a new column indicating the age group for each row based on the 'Age' colum
bins = [20, 30, 40, 50, 60, float('inf')]
labels = ['21-30', '31-40', '41-50', '51-60', 'above 60']
diabetes_data['Age Group'] = pd.cut(diabetes_data['Age'],
                                    bins=bins,
                                    labels=labels,
                                    include_lowest=True)

# Group the data by age group and calculate the median insulin value for each group
insulin_median_by_age_group = diabetes_data.groupby(
    'Age Group')['Insulin'].median()

# Print the results
print(insulin_median_by_age_group)
```

In [ ]:
```
# Define a dictionary with average insulin values based on age groups

insulin_values = {
    '21-30': 105,
    '31-40': 140,
    '41-50': 131,
    '51-60': 207,
    'above 60': 180
}

# Fill NaN values in the 'Insulin' column based on the age group
diabetes_data['Insulin'] = diabetes_data.apply(
    lambda x: insulin_values[x['Age Group']]
    if pd.isna(x['Insulin']) else x['Insulin'],
```

```
    axis=1)
diabetes_data.head()
```

In [ ]:
```
# Remaining variables have balanced dataset,
# Hence NaN values in those variables can be replaced by mean values of respective d

fillna_mean = ['Glucose', 'BloodPressure', 'SkinThickness', 'BMI']
diabetes_data[fillna_mean] = diabetes_data[fillna_mean].fillna(
    diabetes_data[fillna_mean].mean())
diabetes_data.isna().sum(
)  # No NaN values after replacing NaN values with mean.
```

In [ ]:
```
#Checking the distribution of Target variable in the data

diabetes_data['Outcome'].value_counts().plot(kind='bar')
```

As outcome data is not evenly distributed, we will create new samples using SMOTE method for outcome class '1'. This method will generate new samples using extrapolation and will not duplicate any available data.

In [ ]:
```
# Exported the data to prepare a Tableau Dashboard

diabetes_data.to_excel('NIDDK_Updated Data.xlsx', sheet_name = 'NIDDK_Data')
```

In [ ]:
```
# Install the imbalanced-learn library using pip
!pip install imbalanced-learn
```

In [ ]:
```
# Import the SMOTE class from the imblearn.over_sampling module

from imblearn.over_sampling import SMOTE
```

In [ ]:
```
# Extract the feature columns and target column by dropping the 'Outcome' and 'Age G

data_X = diabetes_data.drop(['Outcome','Age Group'], axis=1)
data_y = diabetes_data['Outcome']
```

In [ ]:
```
# Apply SMOTE oversampling technique to balance the classes by creating synthetic sa

X_resampled, y_resampled = SMOTE(random_state=100).fit_resample(data_X, data_y)
print(X_resampled.shape, y_resampled.shape)
```

In [ ]:
```
# Plot a bar chart to visualize the class distribution after oversampling
y_resampled.value_counts().plot(kind='bar')
```

In [ ]:
```
# Concatenate X_resampled and y_resampled along axis 1 to create the resampled data

data_resampled = pd.concat([X_resampled, y_resampled],axis=1)
data_resampled.shape
```

In [ ]:
```
# Create a scatter plot of 'BMI' vs 'Glucose' using the resampled data, with 'Outcom
```

```
sns.scatterplot(x="BMI", y="Glucose", data=data_resampled, hue="Outcome");
```

In [ ]:
```
# Create a heatmap of the correlation matrix of the resampled data

sns.heatmap(data_resampled.corr(),annot=True, cmap='YlGnBu', linewidths=0.1)
fig=plt.gcf()
fig.set_size_inches(20,20)
plt.show()
```

In [ ]:
```
data_resampled.columns
```

In [ ]:
```
# 'data' variale is already defined for only numerical continous features in above c
select_data=data_resampled.loc[:,['Glucose','BloodPressure','Insulin','BMI']]
sns.pairplot(select_data)
```

A baseline model to predict the risk of diabetes using a various machine learning models

In [ ]:
```
# Import the StandardScaler from sklearn.preprocessing

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# Get the column names of the resampled data (excluding the target column)

columns = data_resampled.columns[:-1]
scaled_data = sc.fit_transform(data_resampled[columns])
diabetes_data_sc = pd.DataFrame(scaled_data, columns= columns)
diabetes_data_sc.head()
```

In [ ]:
```
# Create empty lists to store models and evaluation metrics

models = []
model_accuracy = []
model_f1_score = []
model_auc_score = []
```

1) Logistic Regression

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Assigning the feature data to X
X = diabetes_data_sc

# Assigning the target variable to y
y = data_resampled['Outcome']

# Splitting the data into training and testing sets
# Splitting the data into training and testing sets using train_test_split function

X_train, X_test, y_train, y_test = train_test_split (X, y, test_size = 0.2, random_s
```

In [ ]:
```
# Logistic regression
model_lr = LogisticRegression(random_state=100)  # Create a logistic regression mode
model_lr.fit(X_train, y_train)  # Fit the model to the training data
y_pred = model_lr.predict(X_test)  # Predict the target variable for the test data
accuracy_lr = accuracy_score(y_test, y_pred)  # Calculate the accuracy of the model
print('Accuracy of Logistic Regression= %.3f' % accuracy_lr)  # Print the accuracy o
```

In [ ]:
```
from sklearn.model_selection import GridSearchCV, cross_val_score
parameters = {'C': np.logspace(0, 5, 50)}  # Define the parameter grid for grid sear
```

In [ ]:
```
gs_lr = GridSearchCV(model_lr, param_grid=parameters, cv=5, verbose=0)  # Perform gr
gs_lr.fit(X_train, y_train)  # Fit the grid search model to the training data
```

In [ ]:
```
lr_best_param = gs_lr.best_params_  # Get the best parameters found by grid search
lr_best_param
```

In [ ]:
```
# Logistic regression
model_lr_1 = LogisticRegression(C=2.02, random_state=100)  # Create a logistic regre
model_lr_1.fit(X_train, y_train)  # Fit the model to the training data with best par
y_pred_lr = model_lr_1.predict(X_test)  # Predict the target variable for the test a
accuracy_lr = accuracy_score(y_test, y_pred_lr)  # Calculate the accuracy of the upa
print('Accuracy of Logistic Regression= %.3f' % accuracy_lr)  # Print the accuracy o
```

In [ ]:
```
from sklearn.metrics import roc_auc_score, roc_curve

probs = model_lr.predict_proba(X_test)  # Get predicted probabilities for the test a
probs = probs[:, 1]  # Extract probabilities of the positive class
auc_lr = roc_auc_score(y_test, probs)  # Calculate the AUC-ROC score
print('AUC:', auc_lr)  # Print the AUC-ROC score
```

In [ ]:
```
fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve metrics
plt.plot(fpr, tpr, marker='.')  # Plot ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - Logistic Regression')  # Set title
```

In [ ]:
```
#Append model name, model accuracy and AUC.

models.append('LR')
model_accuracy.append(accuracy_lr)
model_auc_score.append(auc_lr)
```

2) Decision Tree:

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
model_dt = DecisionTreeClassifier(random_state=100)
```

In [ ]:
```
# Define the parameters for grid search
parameters = {
```

```
    'max_depth': [1, 2, 3, 4, 5, 6, None]
}
# Create a GridSearchCV object with DecisionTreeClassifier and parameters
gs_dt = GridSearchCV(model_dt, param_grid=parameters, cv=5, verbose=0)

gs_dt.fit(X_train, y_train)  # Fit the GridSearchCV object to the training data

gs_dt.best_params_  # Get the best parameters found by grid search
```

In [ ]:

```
# Get the best score found by grid search
gs_dt.best_score_
```

In [ ]:

```
model_dt = DecisionTreeClassifier(max_depth = 3)
model_dt.fit(X_train, y_train)
accuracy_dt = model_dt.score(X_test, y_test)
print('Accuracy of Decision Tree= %.3f' %accuracy_dt)
```

In [ ]:

```
model_dt.feature_importances_
```

In [ ]:

```
plt.figure(figsize=(8,3))  # Create a figure with a specific size
columns = X_train.columns  # Get the column names of X_train
sns.barplot(y=columns, x=model_dt.feature_importances_)  # Create a bar plot of feat
plt.title("Feature Importance in Model")  # Set the title of the plot
```

In [ ]:

```
probs = model_dt.predict_proba(X_test)  # Get the predicted probabilities from the m
probs = probs[:,1]  # Extract the probabilities for the positive class
auc_dt = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC:', auc_dt)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0,1], [0,1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')  # Set the x-axis label
plt.ylabel('True Positive Rate')  # Set the y-axis label
plt.title('ROC curve - Decision Tree')  # Set the title of the plot
```

In [ ]:

```
models.append('DT')  # Add the model name to the list of models
model_accuracy.append(accuracy_dt)  # Add the model accuracy to the list of accuraci
model_auc_score.append(auc_dt)  # Add the AUC score to the list of AUC scores
```

3) RandomForest Classifier:

In [ ]:

```
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(random_state=100)  # Create a Random Forest classifier
```

In [ ]:

```
parameters = {
    'n_estimators' : [10,50,100,150],  # Define the number of trees in the forest
    'max_depth' : [None,1,3,5,7,9],  # Define the maximum depth of the tree
    'min_samples_leaf' : [1,3,5,7,9],  # Define the minimum number of samples requir
    'min_samples_split': [1,2,3,4,5]  # Define the minimum number of samples require
}
```

In [ ]:
```
gs_rf = GridSearchCV(estimator=rf,param_grid=parameters,cv=5,verbose=0)  # Perform g
gs_rf.fit(X_train, y_train)  # Fit the model with training data
```

In [ ]:
```
gs_rf.best_score_  # Print the best score achieved during grid search
```

In [ ]:
```
gs_rf.best_params_  # Print the best hyperparameters found during grid search
```

In [ ]:
```
model_rf = RandomForestClassifier(n_estimators=100,max_depth=None,min_samples_leaf=1
model_rf.fit(X_train, y_train)  # Fit the model with training data
accuracy_rf = model_rf.score(X_test, y_test)  # Calculate the accuracy of the model
print('Accuracy of Random Forest= %.3f' %accuracy_rf)  # Print the accuracy
```

In [ ]:
```
plt.figure(figsize=(8,3))
sns.barplot(y=columns, x=model_rf.feature_importances_)  # Plot the feature importan
plt.title("Feature Importance in Model")
```

In [ ]:
```
probs = model_rf.predict_proba(X_test)  # Get predicted probabilities from the model
probs = probs [:,1]  # Extract the probabilities for the positive class
auc_rf = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC:', auc_rf)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr,tpr,marker='.')
plt.plot([0,1],[0,1],linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - Random Forest');
```

In [ ]:
```
models.append('RF')  # Add the model name to a list of models
model_accuracy.append(accuracy_rf)  # Add the model accuracy to a list
model_auc_score.append(auc_rf)  # Add the model AUC score to a list
```

4) K-Nearest Neighbour (KNN):

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
model_knn = KNeighborsClassifier()  # Create KNN classifier
```

In [ ]:
```
knn_neighbors = [i for i in range(2,20)]  # List of neighbors to test
parameters = {
    'n_neighbors': knn_neighbors
}
```

In [ ]:
```
gs_knn = GridSearchCV(estimator=model_knn,param_grid=parameters,cv=5,verbose=0)  # P
gs_knn.fit(X_train, y_train)  # Fit the model with training data
```

In [ ]:
```
gs_knn.best_params_  # Print the best parameters found by grid search
```

In [ ]:
```
gs_knn.best_score_  # Print the best score achieved by the model
```

In [ ]:
```
model_knn = KNeighborsClassifier(n_neighbors=3, p=2)  # Create KNN model with specif
model_knn.fit(X_train,y_train)  # Fit the model with training data
model_knn.score(X_train,y_train)  # Calculate the accuracy score on training data
```

In [ ]:
```
accuracy_knn = model_knn.score(X_test, y_test)  # Calculate the accuracy score on te
print('Accuracy of KNN= %.3f' %accuracy_knn)
```

In [ ]:
```
pred_y_knn = model_knn.predict(X_test)  # Make predictions on test data
accuracy_score(y_test,pred_y_knn)  # Calculate accuracy score using predicted and tr
```

In [ ]:
```
probs = model_knn.predict_proba(X_test)  # Get class probabilities for test data
probs = probs [:,1]  # Extract probabilities for positive class
auc_knn = roc_auc_score(y_test, probs)  # Calculate AUC score
print('AUC:', auc_knn)

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve
plt.plot(fpr,tpr,marker='.')  # Plot ROC curve
plt.plot([0,1],[0,1],linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - KNN');
```

In [ ]:
```
models.append('KNN')  # Add model name to list
model_accuracy.append(accuracy_knn)  # Add model accuracy to list
model_auc_score.append(auc_knn)  # Add model AUC score to list
```

In [ ]:
```
gs_knn.cv_results_['mean_test_score']  # Print mean test scores for different parame
```

In [ ]:
```
plt.figure(figsize=(6,4))
sns.barplot(x=knn_neighbors, y=gs_knn.cv_results_['mean_test_score'])  # Plot bar ch
plt.xlabel("N_Neighbors")
plt.ylabel("Test Accuracy")
plt.title("Test Accuracy vs. N_Neighbors");
```

5) Support Vector Machine (SVM):

In [ ]:
```
from sklearn.svm import SVC
model_svm = SVC(kernel='rbf', random_state=100, verbose=0)  # Create an SVM model wi
```

In [ ]:
```
parameters = {
    'C': [1, 5, 10, 15, 20, 25]  # Define a grid of C values for hyperparameter tuni
}
```

In [ ]:
```
gs_svm = GridSearchCV(estimator=model_svm, param_grid=parameters, cv=5, verbose=5)
gs_svm.fit(X, y)  # Fit the model to the training data
```

In [ ]:
```
gs_svm.best_score_
```

In [ ]:
```
gs_svm.best_params_
```

In [ ]:
```
gs_svm.best_estimator_
```

In [ ]:
```
model_svm_1 = SVC(probability=True, C=5, kernel='rbf', random_state=100, verbose=0)
```

In [ ]:
```
model_svm_1.fit(X_train,y_train)
```

In [ ]:
```
model_svm_1.score(X_train,y_train)
```

In [ ]:
```
accuracy_svm = model_svm_1.score(X_test, y_test)  # Calculate the accuracy of the SV
print('Accuracy of SVM = %.3f' % accuracy_svm)
```

In [ ]:
```
probs = model_svm_1.predict_proba(X_test)  # Get the predicted probabilities from th
probs = probs[:, 1]  # Select the probabilities for the positive class
auc_svm = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC: %.3f' % auc_svm)

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve values
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - SVM');
```

In [ ]:
```
models.append('KNN')
model_accuracy.append(accuracy_svm)
model_auc_score.append(auc_svm)
print(accuracy_svm, '%.3f' % auc_svm)  # Print the accuracy and AUC score
```

6) Naive Bayes Algorithm:

In [ ]:
```
from sklearn.naive_bayes import GaussianNB
model_gnb = GaussianNB()  # Create Gaussian Naive Bayes model
```

In [ ]:
```
model_gnb.fit(X_train, y_train)  # Train the model
```

In [ ]:
```
accuracy_gnb = model_gnb.score(X_test, y_test)  # Calculate accuracy score
accuracy_gnb
```

In [ ]:
```
probs = model_gnb.predict_proba(X_test)  # Get predicted probabilities
probs = probs[:, 1]  # Select probabilities for positive class
auc_gnb = roc_auc_score(y_test, probs)  # Calculate AUC score
print('AUC: %.3f' % auc_gnb)
```

```
fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Add diagonal reference line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - GNB');
```

In [ ]:
```
models.append('GNB')  # Add model name to list
model_accuracy.append(accuracy_gnb)  # Add accuracy score to list
model_auc_score.append(auc_gnb)  # Add AUC score to list
print(accuracy_gnb, '%.3f' % auc_gnb)  # Print accuracy score and AUC score
```

7) Ensembler Learning --> Adaptive Boosting

In [ ]:
```
from sklearn.ensemble import AdaBoostClassifier
model_ada = AdaBoostClassifier(random_state=100)  # Initialize AdaBoost classifier
```

In [ ]:
```
parameters = {
    'n_estimators': [10,100,500,1000]  # Set parameter grid for grid search
}
gs_ada = GridSearchCV(model_ada,param_grid=parameters,cv=5,verbose=0)  # Perform gri
gs_ada.fit(X,y)  # Fit grid search to data
```

In [ ]:
```
gs_ada.best_params_  # Print the best parameters found by grid search
```

In [ ]:
```
gs_ada.best_score_  # Print the best score found by grid search
```

In [ ]:
```
model_ada = AdaBoostClassifier(n_estimators=100,random_state=100)  # Initialize AdaB
model_ada.fit(X_train,y_train)  # Fit the model to the training data
accuracy_ada = model_ada.score(X_test,y_test)  # Calculate accuracy on the test data
accuracy_ada  # Print the accuracy
```

In [ ]:
```
probs = model_ada.predict_proba(X_test)  # Get predicted probabilities
probs = probs [:,1]  # Extract probabilities for positive class
auc_ada = roc_auc_score(y_test, probs)  # Calculate AUC score
print('AUC: %.3f' %auc_ada)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve values
plt.plot(fpr,tpr,marker='.')  # Plot ROC curve
plt.plot([0,1],[0,1],linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - ADA');  # Set title for the plot
```

In [ ]:
```
models.append('ADA')  # Append model name to a list
model_accuracy.append(accuracy_ada)  # Append accuracy to a list
model_auc_score.append(auc_ada)  # Append AUC score to a list
print(accuracy_ada, '%.3f' % auc_ada)  # Print accuracy and AUC score
```

8) Ensembler Learning --> Gradient Boosting

In [ ]:
```
!pip install xgboost  # Install XGBoost library
from xgboost import XGBClassifier  # Import XGBoost classifier
xgb = XGBClassifier()  # Initialize XGBoost classifier
```

In [ ]:
```
parameters = {
    'n_estimators': range(2, 10, 1),  # Define range of values for number of estimat
    'max_depth': range(10, 250, 50),  # Define range of values for maximum depth
    'learning_rate': [0.1, 0.01, 0.05]  # Define learning rates to be tested
}

gs_xgb = GridSearchCV(xgb, param_grid=parameters, cv=5, verbose=0)  # Perform grid s
gs_xgb.fit(X, y)  # Fit the model with the best parameters
```

In [ ]:
```
gs_xgb.best_params_  # Display the best parameters found by grid search
```

In [ ]:
```
gs_xgb.best_score_  # Display the best score obtained by grid search
```

In [ ]:
```
model_xgb = XGBClassifier(n_estimators=8, learning_rate=0.1, max_depth=10)  # Create
model_xgb.fit(X_train, y_train)  # Fit the XGBoost model to the training data
accuracy_xgb = model_xgb.score(X_test, y_test)  # Calculate the accuracy of the mode
accuracy_xgb  # Display the accuracy of the model on the test data
```

In [ ]:
```
model_xgb.score(X_train, y_train)  # Calculate the accuracy of the model on the trai
```

In [ ]:
```
probs = model_xgb.predict_proba(X_test)  # Calculate the predicted probabilities for
probs = probs[:, 1]  # Keep the probabilities of the positive class
auc_xgb = roc_auc_score(y_test, probs)  # Calculate the AUC score using the predicte
print('AUC: %.3f' % auc_xgb)  # Display the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - XGBoost');  # Set title for the plot
```

In [ ]:
```
plt.figure(figsize=(8, 3))  # Create a new figure with specified size
sns.barplot(y=columns, x=model_xgb.feature_importances_)  # Create a bar plot for fe
plt.title("Feature Importance in Model");  # Set title for the plot
```

In [ ]:
```
models.append('XGBoost')  # Add model name to the list of models
model_accuracy.append(accuracy_xgb)  # Add model accuracy to the list
model_auc_score.append(auc_xgb)  # Add AUC score to the list
print(accuracy_xgb, '%.3f' % auc_xgb)  # Display accuracy and AUC score
```

In [ ]:
```
# Creating a dataframe to summarize model performance
model_summary = pd.DataFrame(zip(models,model_accuracy,model_auc_score),columns= ['M
model_summary = model_summary.set_index('Model')
```

```
# Displaying the model summary table
model_summary
```

In [ ]:
```
# Plotting a bar chart to compare different classification models
model_summary.plot(figsize=(10,7),kind='bar')
plt.xlabel('Different classification models')
plt.yticks(np.arange(0, 1.2, step=0.2))
plt.title ("Comparison of different classification Algorithms");
```

As Random Forest Model showed highest accuracy in our data, we will set Random Forest as our Final Model

Data Modeling:

Creating a Classification report for Random Forest model

In [ ]:
```
# Initializing the best model with specific hyperparameters
best_model = RandomForestClassifier(n_estimators=100,max_depth=None,min_samples_leaf
```

In [ ]:
```
# Fitting the best model on the training data
best_model.fit(X_train,y_train)

# Predicting the target variable using the best model on the test data
y_predict_rf = best_model.predict(X_test)

# Generating the classification report
report_RF = classification_report(y_test, y_predict_rf)
print(report_RF)
```

In [ ]:
```

```

In [ ]:
```
# Generating the confusion matrix
CF_matrix = confusion_matrix(y_test,y_predict_rf)
print('Confusion Matrix:\n',CF_matrix)
```

In [ ]:
```
# Creating a heatmap of the confusion matrix
sns.heatmap(CF_matrix/np.sum(CF_matrix),annot=True,fmt='.2%')
```

In [ ]:
```
model_score = best_model.score(X_test, y_test)
print ('Accuracy of Random Forest: %.3f' % model_score)
```

With NIDDK dataset, Random Forest method gave best accuracy (84%) in prediction of diabetes compared to other machine learning methods.

Data Visualization:

A tableau dashboard is created to visualize the data with the following objectives:

```
a. Pie chart to describe the diabetic or non-diabetic population
b. Scatter charts between relevant variables to analyze the
relationships
c. Histogram or frequency charts to analyze the distribution of the
data
d. Heatmap of correlation analysis among the relevant variables
e. Create bins of these age values: 20-25, 25-30, 30-35, etc. Analyze
different
variables for these age brackets using a bubble chart.
```

Please find the tableau project on below link:

https://public.tableau.com/app/profile/dharmesh1254/viz/Diabetes

In [ ]: