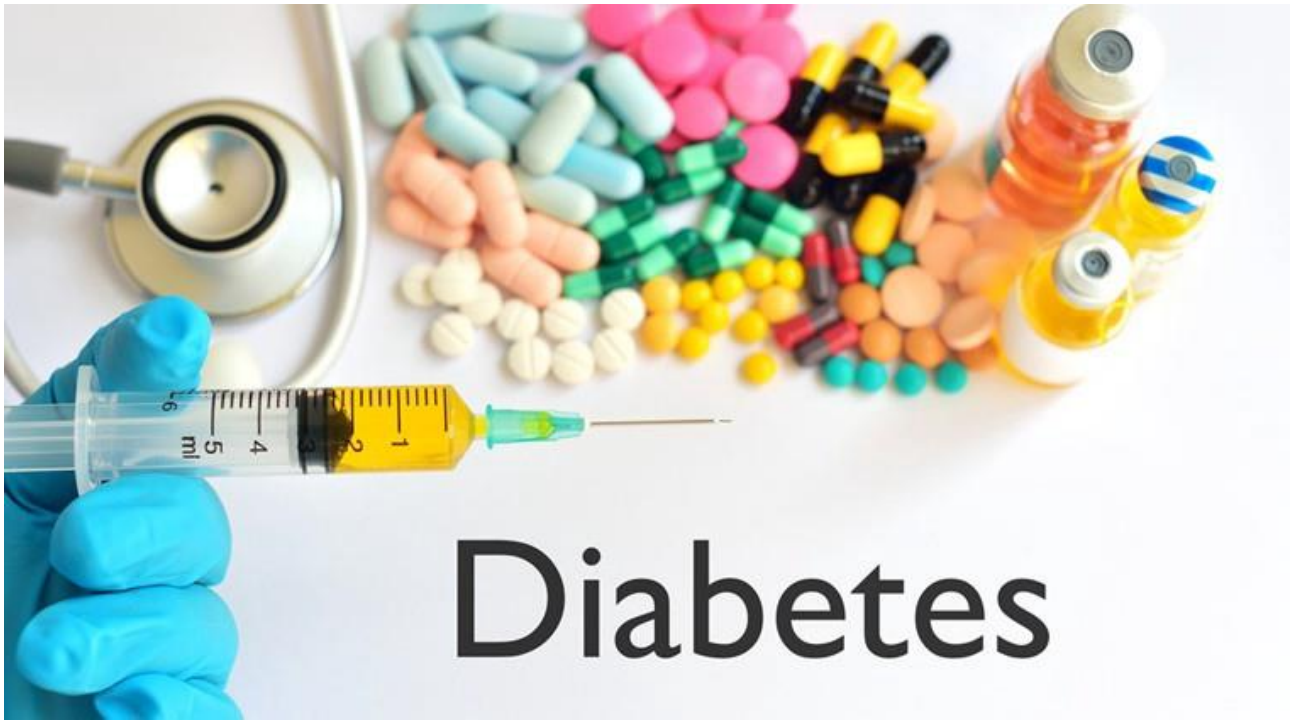
UNIVERSITÀ degli STUDI di CATANIA | DIPARTIMENTO di ECONOMIA e IMPRESA

Pima Indians Diabetes Database



Subject : Advanced Machine Learning

Year : 2021/22

Prof. : Vincenza Carchiolo

Students : Gianluigi Mazzaglia

Davide Saitta

Sommario

Introduction

Diabetes is a chronic disease that occurs when the pancreas is no longer able to make insulin, or when the body cannot make good use of the insulin it produces. Insulin is a hormone made by the pancreas, that acts like a key to let glucose from the food we eat pass from the blood stream into the cells in the body to produce energy. All carbohydrate foods are broken down into glucose in the blood. Insulin helps glucose get into the cells.

We will organize our project firstly making an **Univariate Analysis** and detecting the missing values and the Outlier and then we will do **Supervised Learing** where we are going to consider some models in order to determine which one better perform.

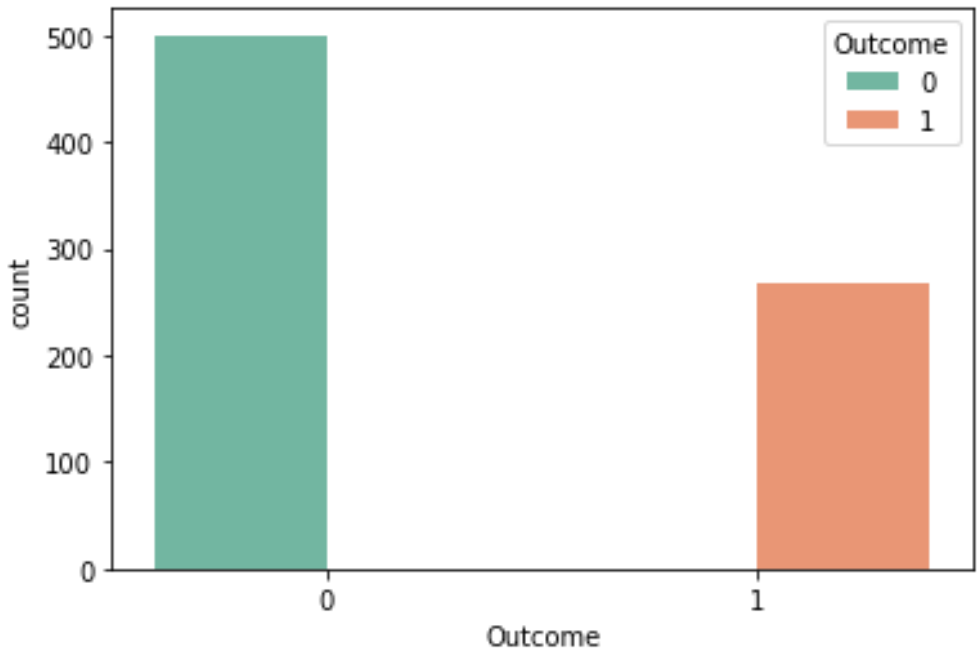
At the end we will do the **Unsupervised** part in order to understand the presence of Clusters and their meaning.

Dataset

The Pima Indians Diabetes Dataset is taken from the Kaggle web site in the following link https://www.kaggle.com/uciml/pima-indians-diabetes-database , It is originally from the National Institute od Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has a diabetes, based on certain diagnostic measurements included in the dataset. All patients are females, at least 21 year old of Pima Indian heritage. The dataset consists of several medical predictor variables and one target variable, **Outcome**. Predictor variables are:

- **Pregnancies**: Number of times pregnant
- **Glucose**: Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- **BloodPressure**: Diastolic blood pressure (mm Hg)
- **SkinThickness**: Triceps skin fold thickness (mm)
- **Insulin**: 2-hour serum insulin (U/ml)
- **BMI**: Body mass index (weight in kg / (height in m)^2)
- **DiabetesPedigreeFunction**
- **Age**

Here we have an extraction from the whole dataset that contains in total 768 observations and 9 columns with no null values and we can see that the target variable is unbalanced (with 0 means “Non-Diabetic” and 1 means “Diabetic”) :




```
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
```

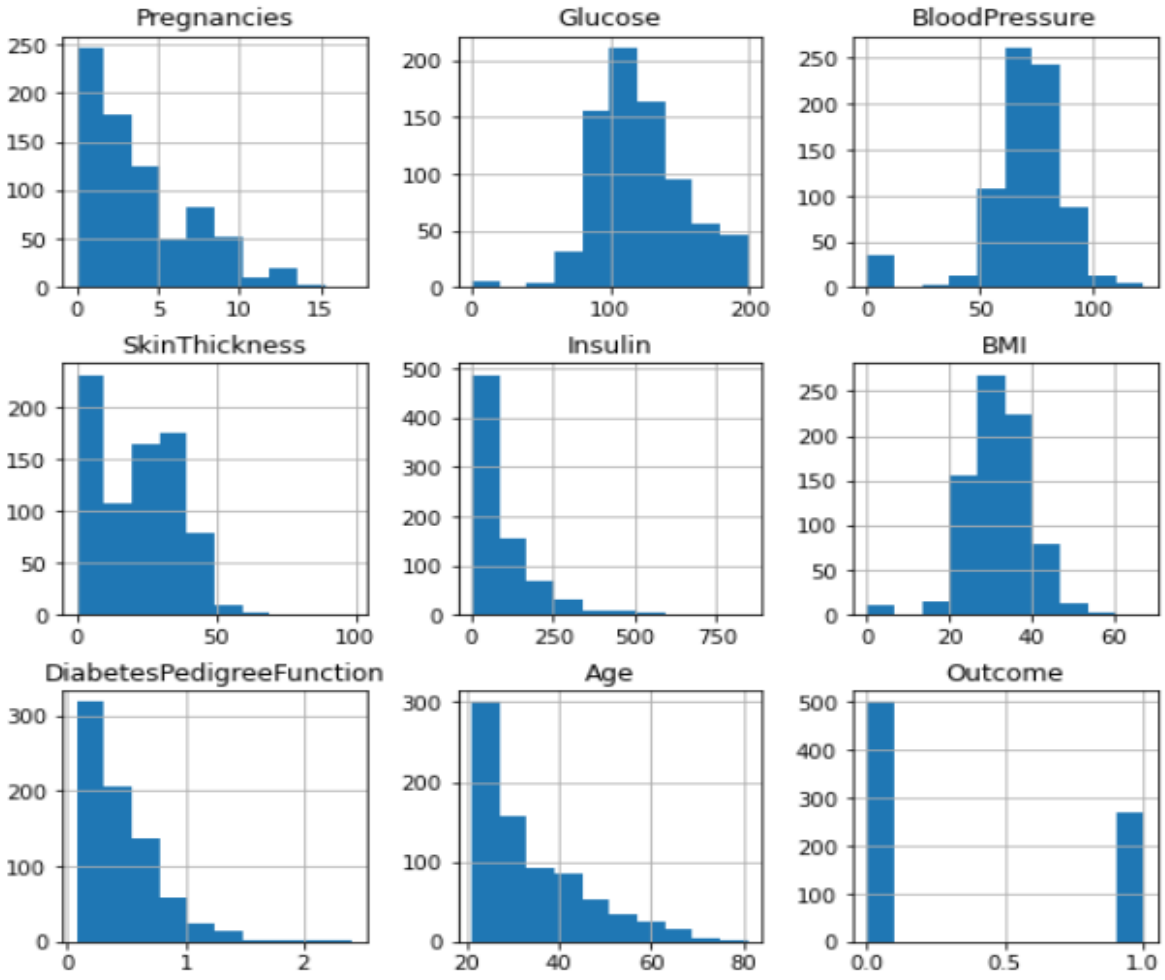
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

	count	mean	std	min	25%	50%	75%	max
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

We can see from the first plot that there are no missing values, but we can see from the informations that there are several variables with a min value equals to zero, we need to understand their meaning and we will now move on the Univariate Analysis.

Univariate Analysis

Here we can see a general overview about the distributions of all the variables:



From the histograms above we can see that there are some variables that need to be treated in order to understand if they can considered outliers.

Looking at the plots below we can see that:

- **Glucose** can't assume a value of zero in a living person, in our dataset we have in total 5 counts where the value is 0.
- **BloodPressure** can't assume a value of zero in a living person, we counted 35 value of 0.
- **SkinThickness** can't be in a normal person equals to 0 mm, we have 227 counts of 0 value.
- **Insulin** is rarely that assumes a value of 0, while here we counts 374 observations with 0 value.
- **BMI** that represents the body mass can't assume a value of zero, here we have 11 counts with 0 value.

```
In [51]: print("Total : ", df[df.Glucose == 0].shape[0]) #count the number of zero in Glucose
         print(df[df.Glucose == 0].groupby('Outcome')['Age'].count())

Total :  5
Outcome
0    3
1    2
Name: Age, dtype: int64
```

```
In [50]: print("Total : ", df[df.BloodPressure == 0].shape[0]) #count the number of zero in Blood Pressure
         print(df[df.BloodPressure == 0].groupby('Outcome')['Age'].count())

Total :  35
Outcome
0    19
1    16
Name: Age, dtype: int64
```

```
In [54]: print("Total : ", df[df.SkinThickness == 0].shape[0]) #count the number of zero in SkinThickness
         print(df[df.SkinThickness == 0].groupby('Outcome')['Age'].count())

Total :  227
Outcome
0    139
1     88
Name: Age, dtype: int64
```

```
In [56]: print("Total : ", df[df.Insulin == 0].shape[0]) #count the number of zero in Insulin
         print(df[df.Insulin == 0].groupby('Outcome')['Age'].count())

Total :  374
Outcome
0    236
1    138
Name: Age, dtype: int64
```

```
In [57]: print("Total : ", df[df.BMI == 0].shape[0]) #count the number of zero in BMI (body mass indicator)
         print(df[df.BMI == 0].groupby('Outcome')['Age'].count())

Total :  11
Outcome
0    9
1    2
Name: Age, dtype: int64
```

We can decide to handle this “invalid data” in several ways:

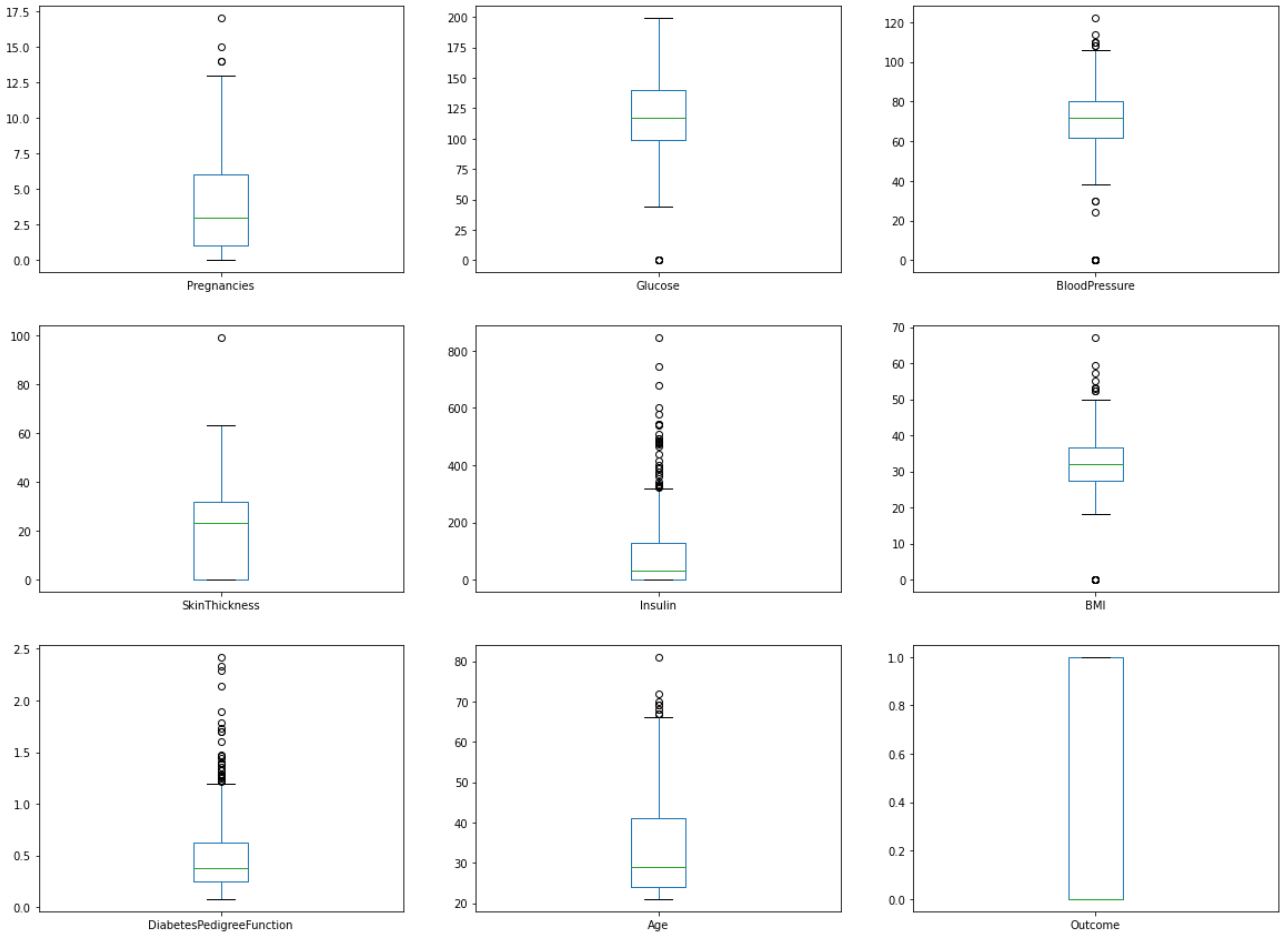
- Ignoring / removing them
- Considering the mean / median

In our case considering the average will be misleading in our prediction Analysis, and if on the one side we can’t remove 374 observations out 764 that represents almost 50% of our data and so we will leave the zeroes in Insulin and SkinThickness. On the other hand we can decide to remove the zeroes that appears in Glucose, BMI and BloodPressure that are in total 50 observations and at the end we would have 724 observations considered as a valid in our Analysis :

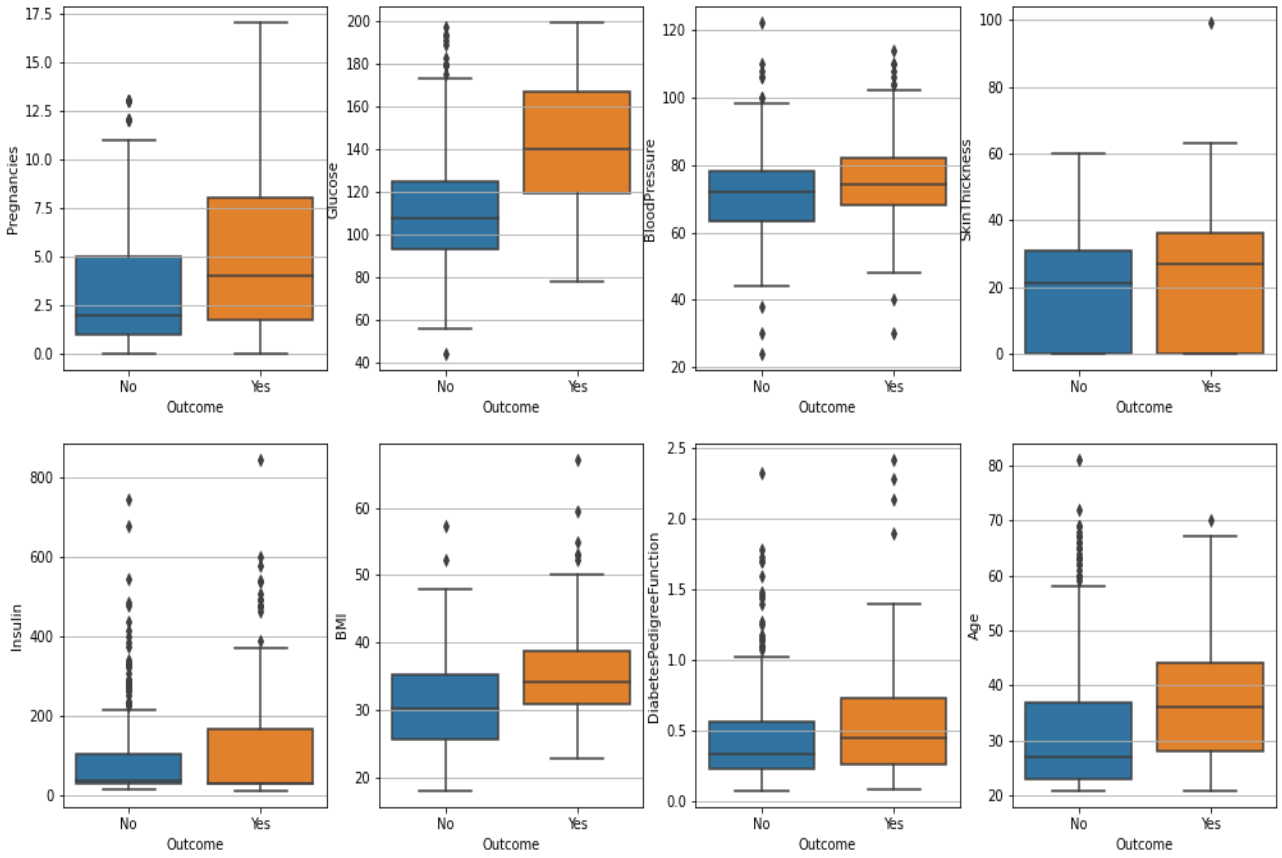
```
diabetes = df[(df.BloodPressure != 0) & (df.BMI != 0) & (df.Glucose != 0)] #removing 0 values in BloodPressure,BMI and Glucose
diabetes.shape[0]
```

```
724
```

Here we have the box plot representation of all the variables and what is evidence is that “Insulin” has an important amount of outliers as the “PedigreeFunction”, the representation of “SkinThickness” is distorted by the high presence of 0 value, it has a positive skew with a long tail on the right. Also the median of “Insuline” is very close to 0 for the same reason, in fact 364 observations out 765 are 0.

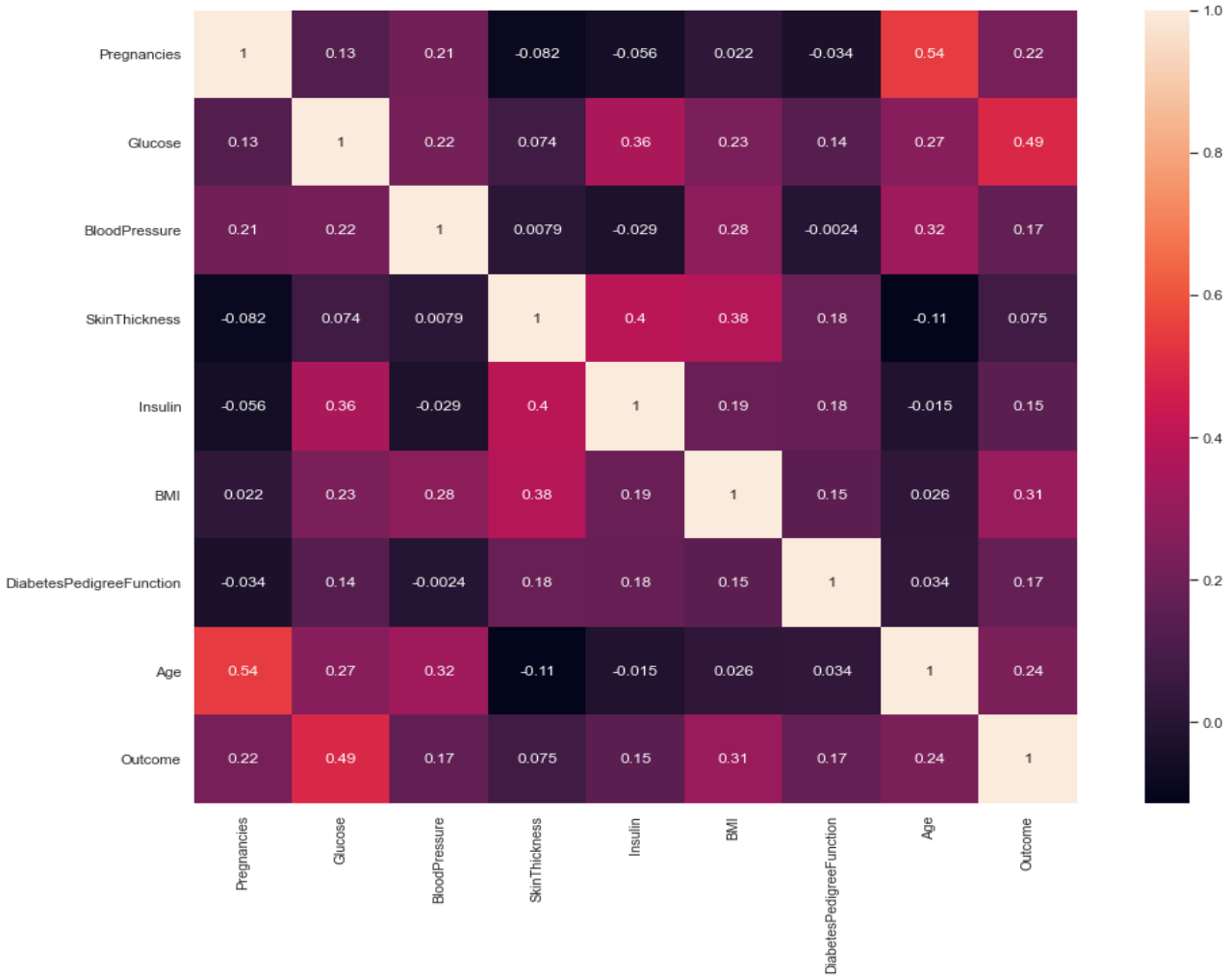


We also want to see the box plot relation and comparing diabetic with non diabetic persons, we can see that people with an high value of Glucose tend to be Diabetic and this is confirmed also in BMI and Insulin.



Before to proceed with the final decision of how to manage the zeroes values we will check other analysis like correlation, the bivariate analysis between the target variable and all the features and we will try some modeling to understand how our accuracy will improve or not.

Correlation.

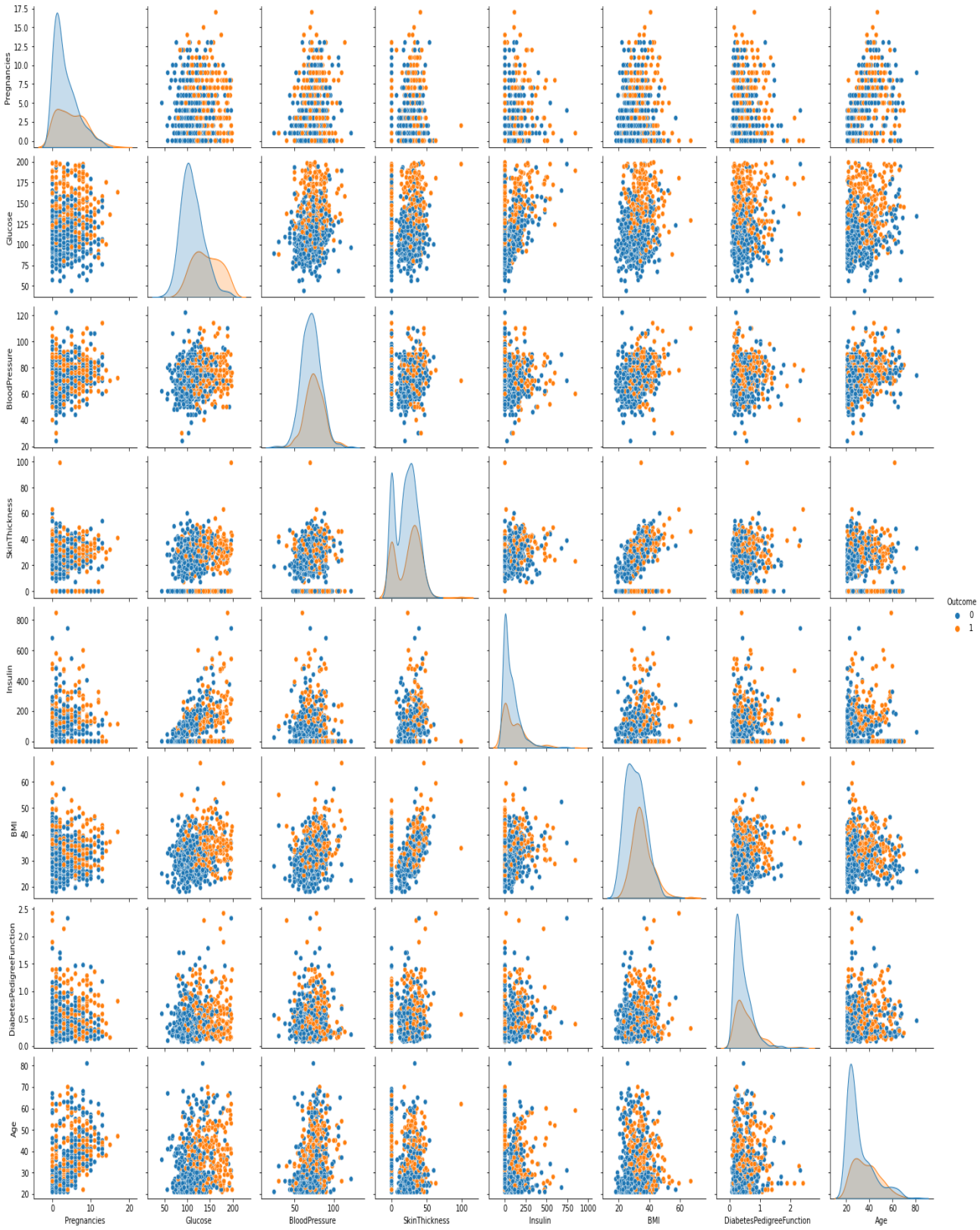


We can see that there are no strong correlation beetween the features , the strongest ones are between :

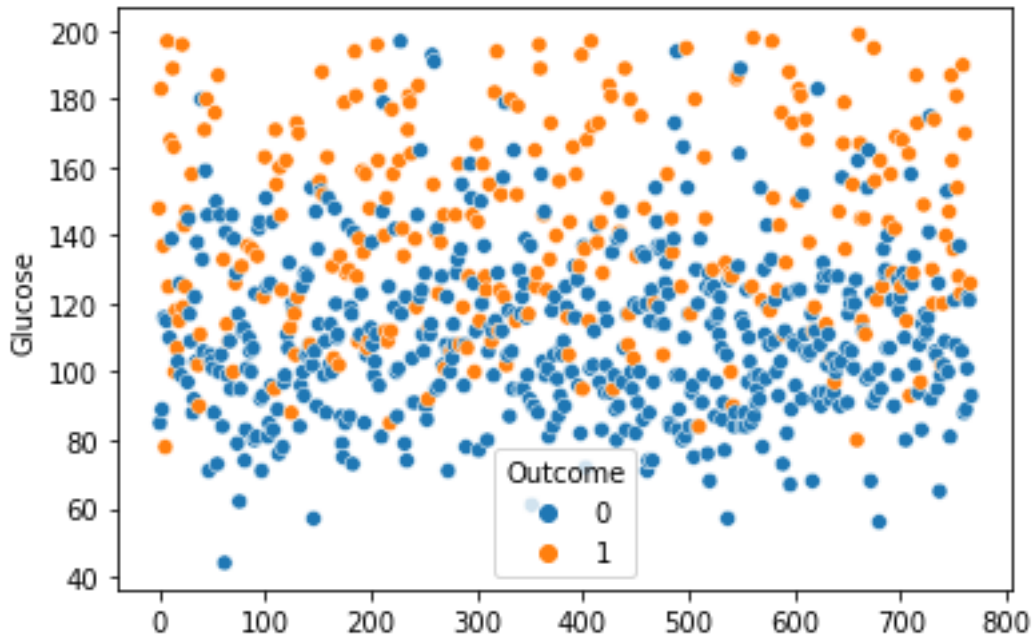
- Age / Pregnancies (0,54) : older women tend to have higher number of pregnancies
- Glucose / Outcome (0,49) : women with high level of Glucose tend to be diabetic
- There is also a signs of correlation between Insulin and Glucose (0.36), the level of these 2 features tend to increase proportionally
- We can finally underline as was likely that if there is an increasing of SkinThickness there is an increasing proportionally of BMI and Insuline (0,38 and 0,40).

In any case there is no presence of **multicollinearity**, so no correlations above 0,70.

In the plot below we see the relation between all the features and Outcome, in particular we can distinguish two distinct group in Glucose within diabetic and Non diabetic with women with an high level of Glucose tend to be Diabetic.



Relation Glucose - Outcome



Supervised Learning

Before to proceed with the usage of the different models we are going to split our Dataset into two parts: a **training set** (70% of the total dataset) to train the model and the **testing set** (30% of the total) to test the model and evaluate the accuracy of the model.

After some trials we have decided as a final choice to replace the zeroes values with the median of the column, in fact the difference in terms of **Accuracy**, **Sensitivity** and **Specificity** was too much dropping without these 51 features, at the end we decided to maintain all 768 features in order to have a better performance.

We will start with the the Bayesian Classification and then we will do Logistic Regression, Support Vector Machine, K- Nearest Neighbor, Decision Tree, and some Ensemble methods like Random forests and then we will compare the results.

Bayesian Classification

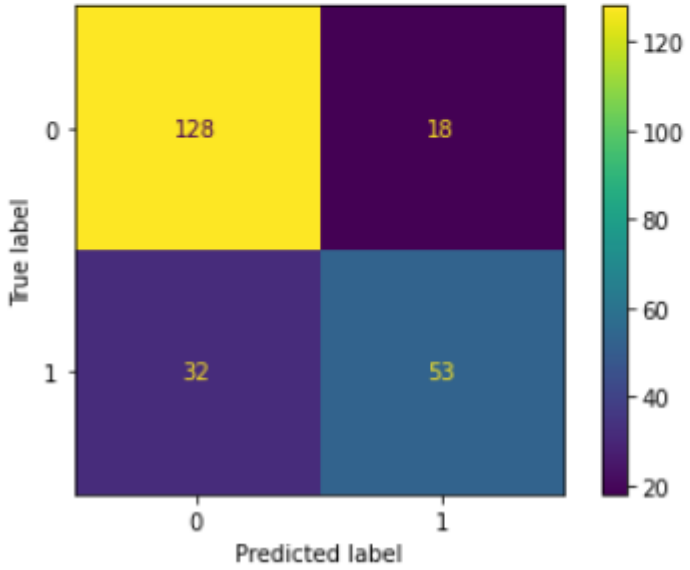
Naive Bayes is a statistical classification technique based on Bayes Theorem which uses probability for prediction of unknown class. Naive Bayes classifier assumes that the effect of a particular feature in a class is indipendent from other features.

```
#Create a Gaussian Classifier
gnb = GaussianNB()

#Train the model using the training sets
gnb.fit(X_train, y_train)
#Predict the response for test dataset
y_pred = gnb.predict(X_test)
```

To evaluate the model we will evaluate the accuracy using actual and predicted values. Notes that the accuracy to predict person with diabetes or not is not the only evaluator we need to consider, we need to identify the correctness in terms of Sensitivity and Specificity, and this will be relevant in the final choice of the best model we are going to choose.
Now we will have a look to the confusion matrix and the following report:

	precision	recall	f1-score	support
0	0.80	0.88	0.84	146
1	0.75	0.62	0.68	85
accuracy			0.78	231
macro avg	0.77	0.75	0.76	231
weighted avg	0.78	0.78	0.78	231

```
Accuracy: 0.7835497835497836
Sensitivity : 0.6235294117647059
Specificity : 0.8767123287671232
```



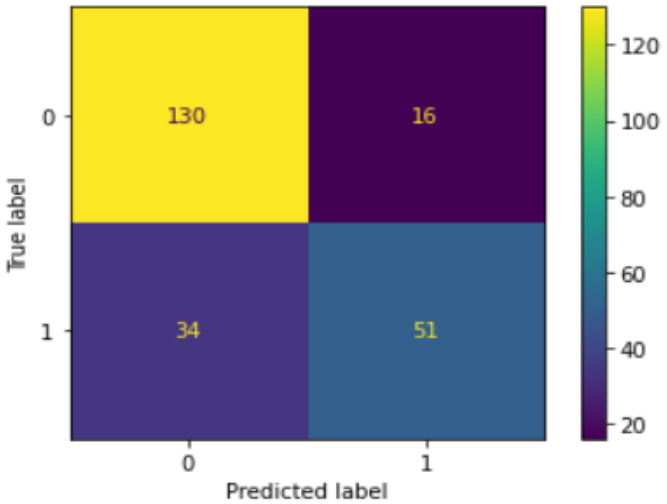
From the report above we can see that considering 231 observations, and the confusion matrix in the rows consider the true label, so the model predicts correctly 128 true negative out 146, it correctly classify 53 true positive out 85 items.
We can consider an accuracy of 0,783 that can be considered as a satisfactory level, with the **precision** value of 0,80 for the negative value (128 over 160) that represents the correctly classified, and 0,75 considering the positive values.
The accuracy alone is not enough to determine the performance of a model but under the column **recall** we are considering the value of **Specificity** equals to 0.878 (round 0.88) that represents the amount of total 0 value classified over the amount of negative values (in this case 128 over 146), while the other value of **Sensitivity** that consider the amount of true value classified over the total amount of 1 values, in this case this value equals to 0,62 can't be considered as a satisfactory and so we have to consider it in the final choice.

Logistic Regression

Now we are going to use Logistic Regression, that is supposed to prerforms in a better way when we face target binary classification problem. It is used when there are no multicollinearity among variables, when data is linearly separable. We are going to see the result and if there are some improvements to respect to the previous model.

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(max_iter=100, C = 100)
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)
```

```
                precision    recall  f1-score   support

           0       0.79      0.89      0.84       146
           1       0.76      0.60      0.67        85

    accuracy                           0.78       231
   macro avg       0.78      0.75      0.75       231
weighted avg       0.78      0.78      0.78       231

Accuracy of logistic regression classifier on test set: 0.784
Sensitivity : 0.6
Specificity : 0.8904109589041096
```



What we can see from the confusion matrix above is that the score is very similar to that one used in the Bayesian method, the accuracy still around 78,4 % with a Specificity value that increase from 0,87 to 0,89 regardless the Sensitivity that

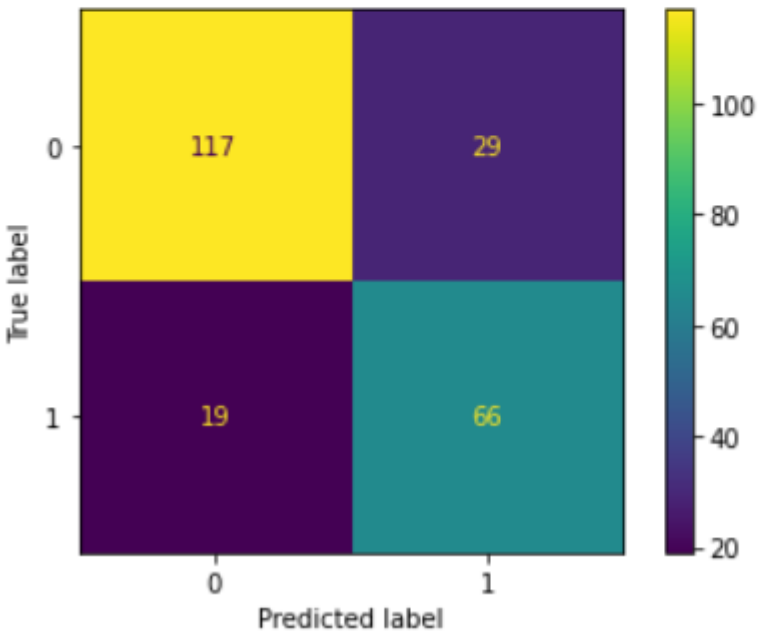
decreace from 0,62 to 0,60. We can definetely say that there is no improvement. What we will do next is to re-run the Logistic Regression modifying the parameters, in particular here we have been considered “max_iter=100” that consider 100 as the maximum number of iteration taken for the solvers to converge, “C=100” is referred to the concept of regularization, smaller values specify stronger regularization (as default C=1.0).

Now we are going to add the parameter “**class_weight : balanced**”, this can be useful when our data are unbalanced and it works adjusting weights inversely proportional to class frequencies in the input data, we will run the algorithm looking at the result:

```
logreg = LogisticRegression(max_iter=100, C = 100, class_weight= 'balanced')
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)
```

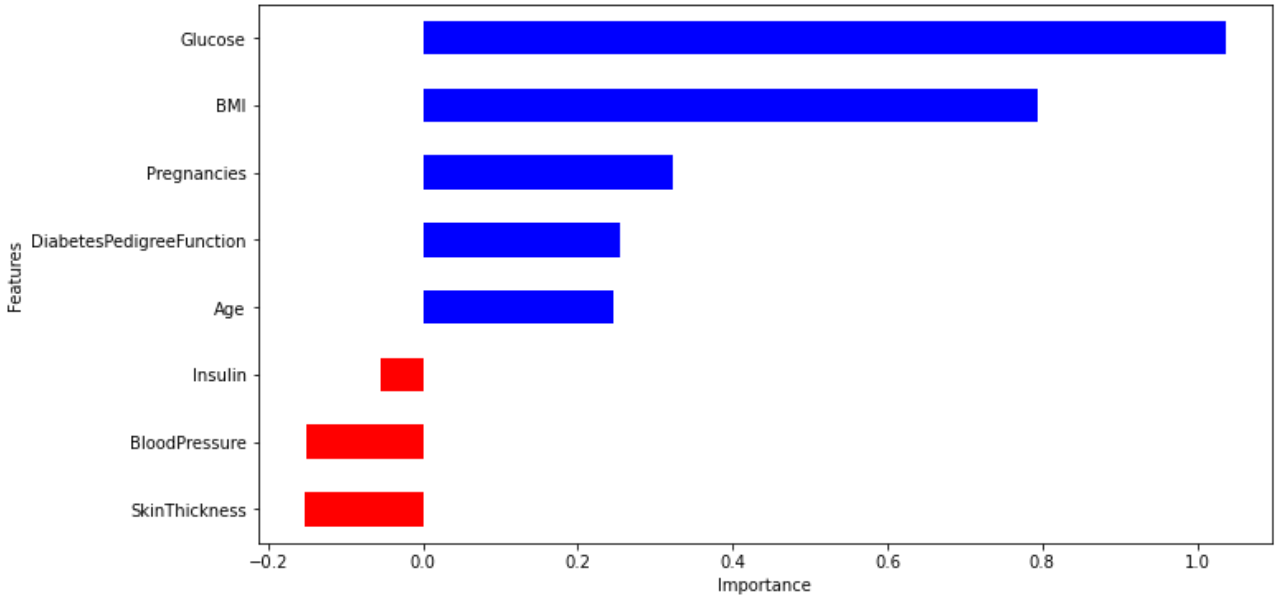
	precision	recall	f1-score	support
0	0.86	0.80	0.83	146
1	0.69	0.78	0.73	85
accuracy			0.79	231
macro avg	0.78	0.79	0.78	231
weighted avg	0.80	0.79	0.79	231

```
Accuracy of logistic regression classifier on test set: 0.792
Sensitivity : 0.7764705882352941
Specificity : 0.8013698630136986
```



Looking at the confusion matrix there are two aspects that we can highlight, in particular the positive value correctly predicted increase from 51 to 66 with a very deeply improvement in the Sensitivity from 60 % to 77,6 % that can be considered as a good value. Of course on the other hand there is a loss in terms of Specificity from 89 % to 80 % but this value maintain a good score and we can say that thia method is much better than the previous one.

In the plot below we can see what is the contribution of the features, as we could image Glucose that represents the variable higher correlated with the Outcome is the most important variable used in the logistic model.



Support vector Machine

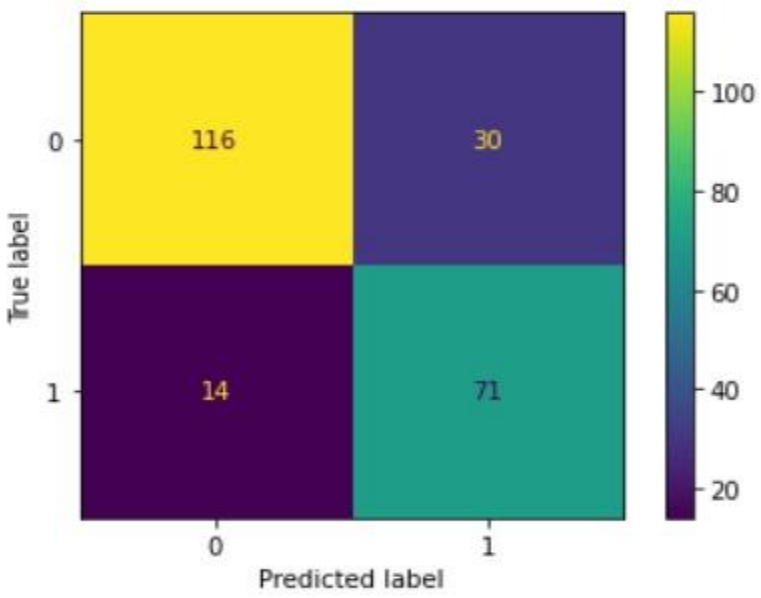
Support vector Machines are discriminative classification models, rather than modeling each class, they simply find a curve (in two dimensions) or a manifold (in multiple dimensions) that divides the classes from each other. It's versatile, in fact according to the different function is possible to define different kernel (rbf, linear, poly and sigmoid). To make the best choice we run a loop and getting the best accuracy within the kernels, in our case "rbf" ws that one with the best accuracy. Rbf (**Radial Basis Function**) are the most generalized form of kernelization and is one of

the most widely used kernels (it is also used as a Default kernel) due to its similarity to the Gaussian distribution. We are going to run the model and looking at the results:

```
#SUPPORT VECTOR MACHINE
model_svm = svm.SVC(kernel='rbf' , class_weight='balanced')
model_svm.fit(X_train, y_train)
y_pred = model_svm.predict(X_test)
```

	precision	recall	f1-score	support
0	0.89	0.79	0.84	146
1	0.70	0.84	0.76	85
accuracy			0.81	231
macro avg	0.80	0.81	0.80	231
weighted avg	0.82	0.81	0.81	231

```
Accuracy of Support vector machine  classifier on test set: 0.810
Sensitivity : 0.8352941176470589
Specificity : 0.7945205479452054
```



As we can see from the confusion matrix above here we have a very good result in terms of Sensitivity that increase to more than 83 %, also the value of Specificity still high with an 80% of correctly zero classified. The level of Accuracy is 81% , considering the class_weight = balanced allow us to have a very good prediction.

K-Nearest Neighbor

Neighbors based classification is a type of instance-based learning or non-generalizing learning.

Scikit-learn implements two different nearest neighbors' classifiers:

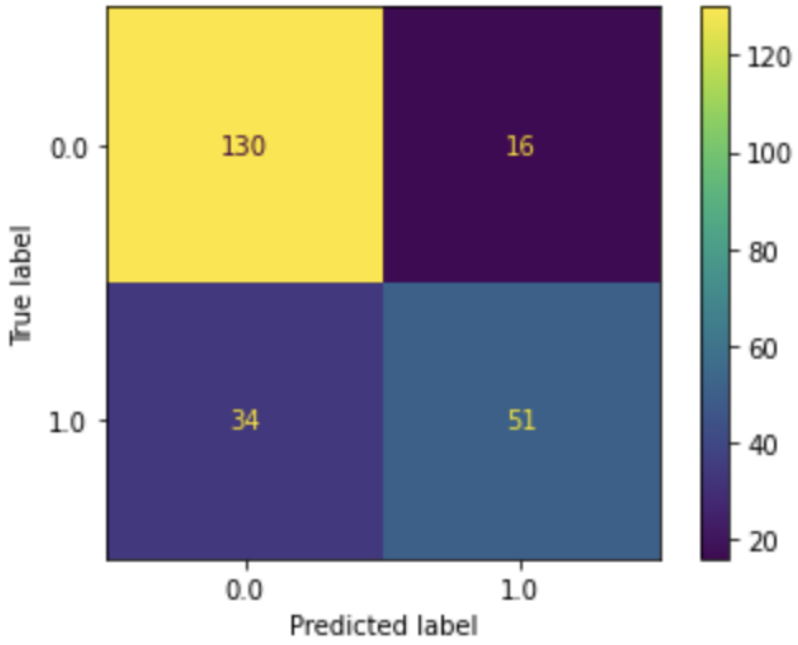
- **KNeighborsClassifier**

  KNeighborsClassifier implements learning based on the k nearest neighbors of each query point.
  The optimal choice of the value k is highly data dependent. In our case, the value of the parameter that highlight the best result in terms of accuracy are the following:

```
KNN = KNeighborsClassifier(n_neighbors=7)
```

```
KNN = KNeighborsClassifier(n_neighbors=9, weights="distance")
```

  In term of accuracy, we obtain practically the same result for both of 78.36%.



Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 130 over 146 (89.04%), while the number of “diabetic” people that have been correctly classified are 51 over 85 (60.00%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 34, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 16.

Is evident that this model has a higher accuracy in predict ‘non-diabetic’ people then ‘diabetic’ people. This is related to the fact that the data is not balanced. The accuracy on predicts ‘diabetic’ people of 60.00% is just slightly better than random guessing.

In order to solve this issue, we can perform undersamplig, and make the data more balanced.

To do that, we have built the following functions:

```
def sample_together(n, X, y):
    rows = random.sample(np.arange(0,len(X.index)).tolist(),n)
    return X.iloc[rows,], y.iloc[rows,]
```

```python
def undersample(X, y, under = True):
    y_min = y[y == under]
    y_max = y[y != under]
    X_min = X.filter(y_min.index,axis = 0)
    X_max = X.filter(y_max.index,axis = 0)

    X_under, y_under = sample_together(len(y_min.index), X_max, y_max)

    X = pd.concat([X_under, X_min])
    y = pd.concat([y_under, y_min])
    return X, y
```

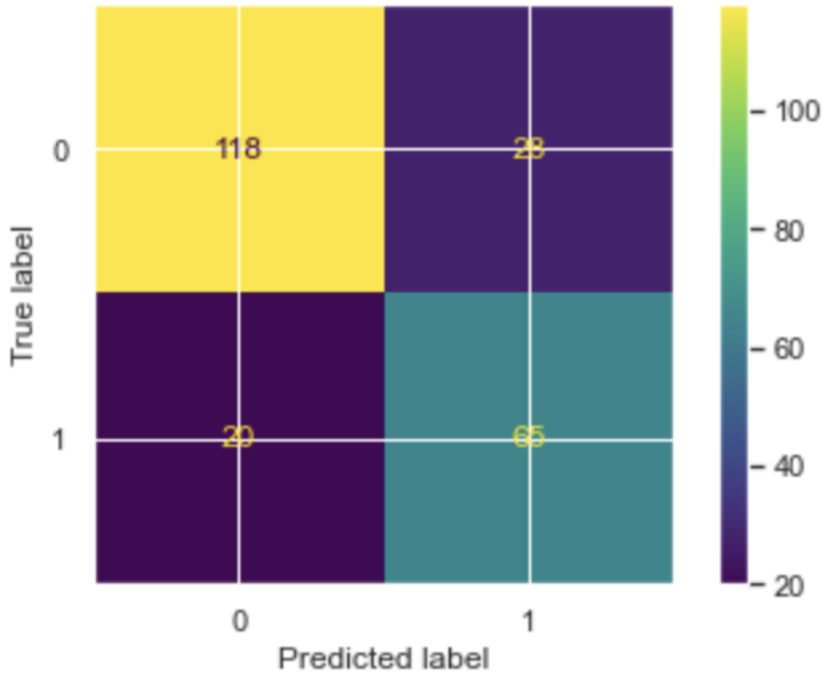
```python
X_train_u, y_train_u = undersample(X_train, y_train)
```

In order to reduce the loss information related to the under sampling, we apply it only to the training data.

Using the new balanced data, we reach the best result using the following parameters:

```python
KNN = KNeighborsClassifier(n_neighbors=15, weights="distance")
```

This model we obtain an accuracy of 77.49%.



Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 118 over 146 (80.82%), while the number of

“diabetic” people that have been correctly classified are 65 over 85 (76.47%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 20, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 28.

The improvement using the balanced version of the data is evident. The accuracy improves from 78.36% to 79.22% (0.86%). It may seem not such improvement, but if we give a look to the confusion matrix, we can compare the values of sensitivity and specificity.

Now we have a sensitivity of 76.47% versus 60.00% of the previous result, so we have an improvement of 16.47%.

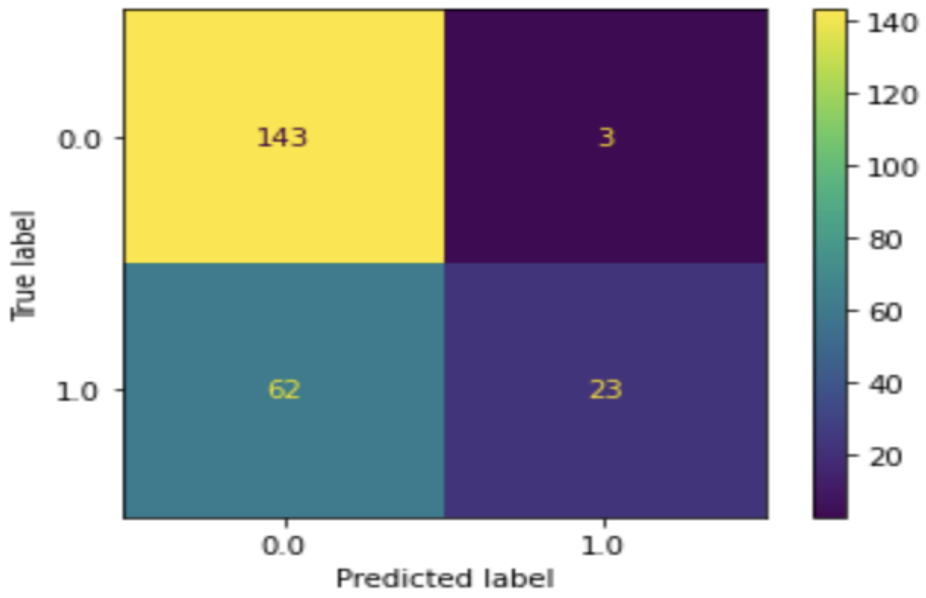
On the other hand, we have a reduction of the specificity from 89.04% to 80.82% (8.22%).

- **RadiusNeighborsClassifier**

  RadiusNeighborsClassifier implements learning based on the number of neighbors within a fixed radius r of each training point.
  Using below parameters, we obtain an accuracy of 71.86%.

```
rnc = RadiusNeighborsClassifier(weights='distance',radius=0.689)
```

  Is not just an accuracy problem, looking to the confusion matrix, is evident the fact that our model has a really low sensitivity.



The number of “non-diabetic” people that have been correctly classified are 143 over 146 (97.95%), while the number of “diabetic” people that have been correctly classified are 23 over 85 (27.06%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 62, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 3.

This model is not acceptable, as its ability to predict ‘diabetic’ people is much worse than random guessing. But this result was expected as is known the fact that this method due to the curse of dimensionality, is not suggested for high dimensional data.

Decision Trees (DTs)

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

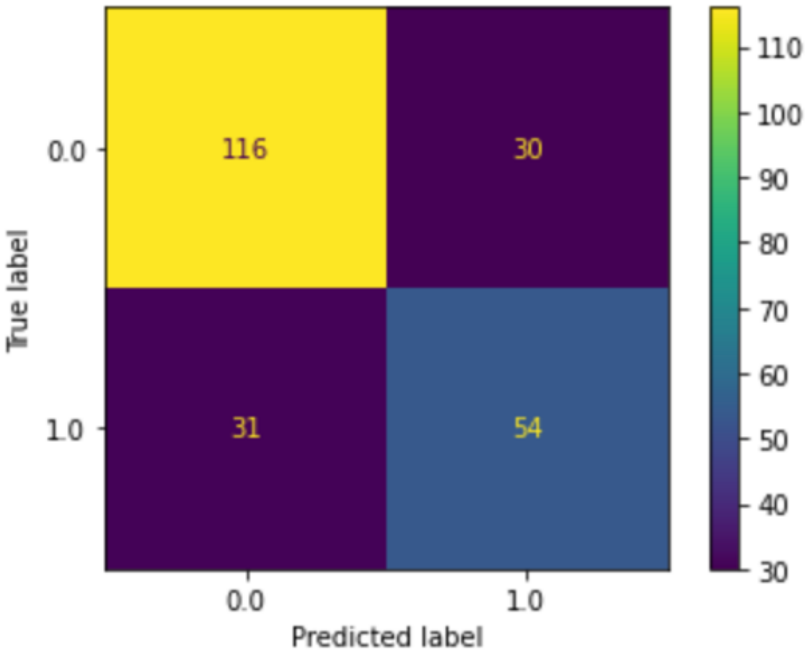
We performed different decision trees.

```
clf = DecisionTreeClassifier()
```

Using the default attributes, is obtained an accuracy of 70.56%.

```
clf = DecisionTreeClassifier(criterion="entropy")
```

Changing only the criterion from the default one (that is gini) to entropy, which is a function that measure the quality of a split we obtain an accuracy of 74.46%.

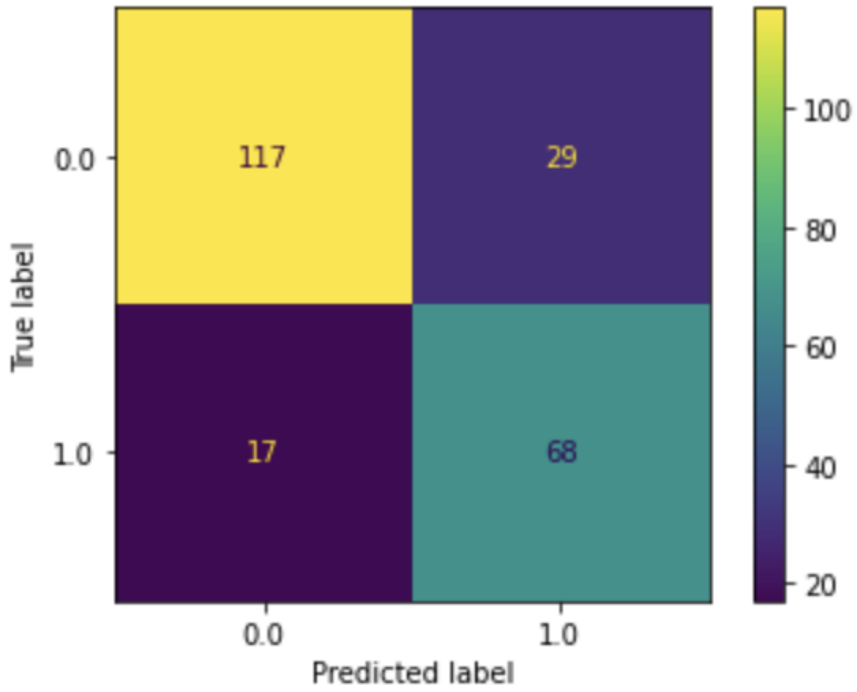


The problem with those basic models, is the fact that they tend to overfit the data. To optimize the model, we can perform pre-pruning using max_depth, in this way the model results more generalized.

```
clf = DecisionTreeClassifier(criterion="entropy", max_depth=7)
```

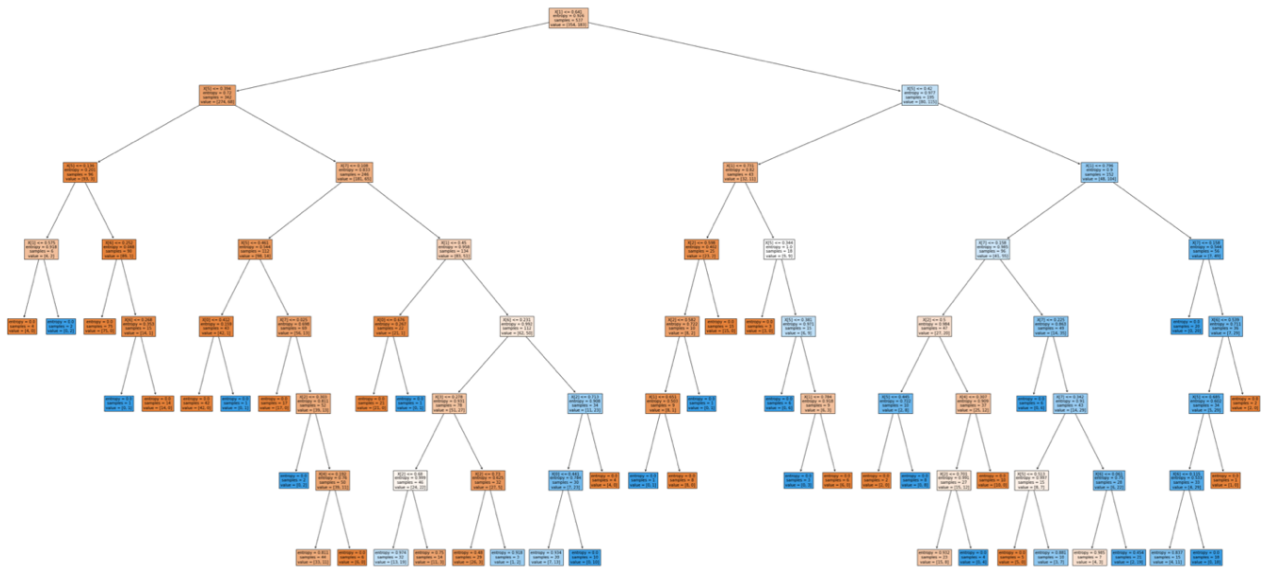
The configuration that exploits the best value of accuracy is the one that use the entropy as criterion to determine the purity of the splits, and with depth = 7.

This model produces an accuracy of 80.09%.



Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 117 over 146 (80.14%), while the number of “diabetic” people that have been correctly classified are 68 over 85 (80.00%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 17, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 29.

One of the advantages of decision trees is that tree can be visualized.



Another way to perform pruning is using the cost complexity pruning, which, is parameterized by ccp_alpha.

In order to get the effective values of alpha we do as follows:

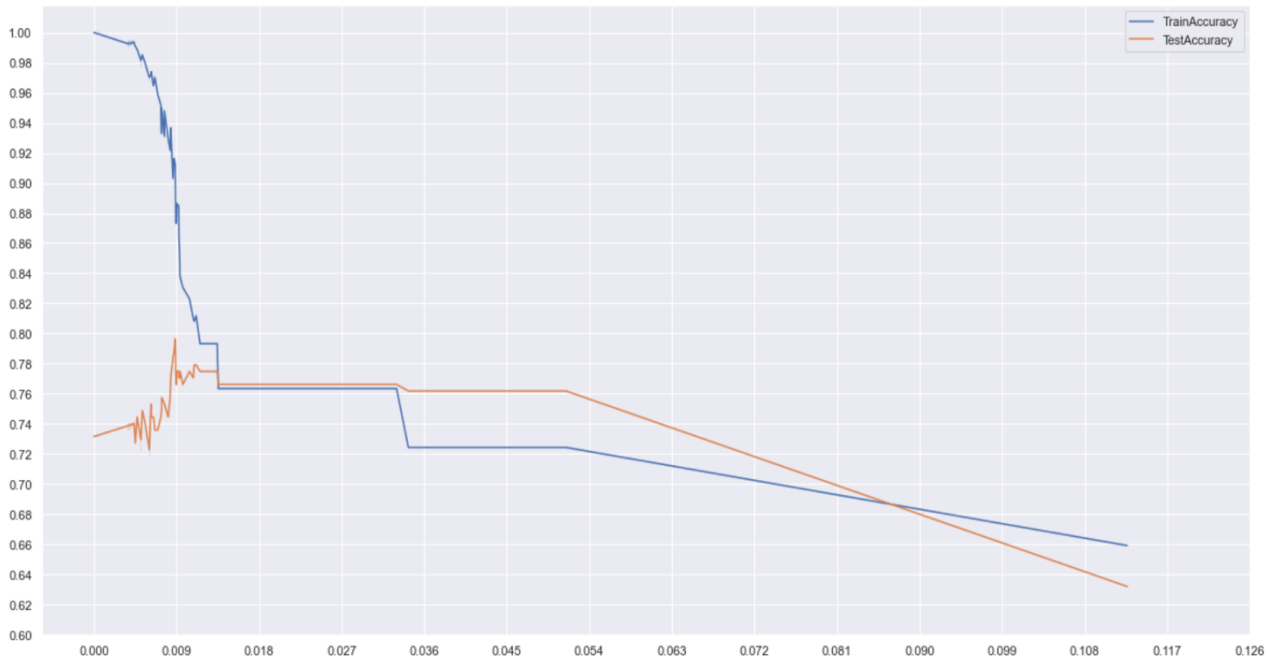
```
path=clf.cost_complexity_pruning_path(X_train,y_train)

alphas=path.ccp_alphas
alphas
```

```
array([0.        , 0.00176419, 0.00184173, 0.00194453, 0.0027933 ,
       0.0027933 , 0.00283763, 0.00293048, 0.00300455, 0.00306603,
       0.00351955, 0.0035551 , 0.00499542, 0.00578282, 0.00583738,
       0.00682078, 0.00933972, 0.01112616, 0.01180365, 0.01631776,
       0.01752658, 0.02118767, 0.07069409])
```

Then we visualize how change the accuracy varying alpha for both training and test dataset.

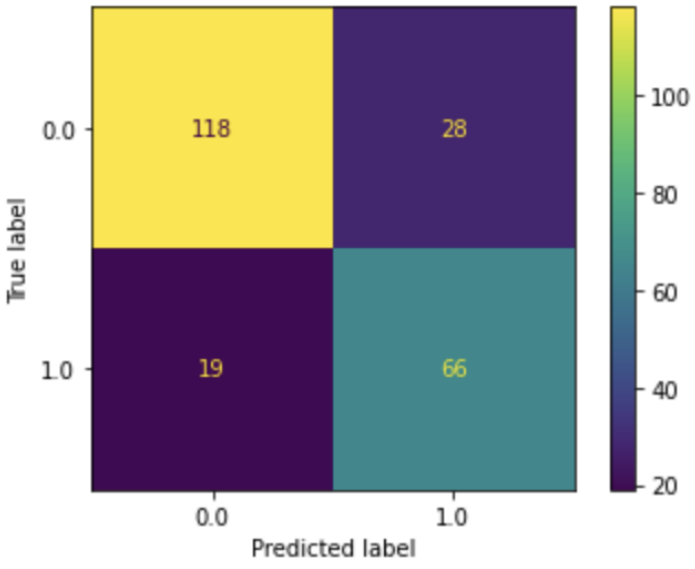
```
accuracy_train,accuracy_test=[],[]
for i in alphas:
    clf = DecisionTreeClassifier(criterion= 'entropy', ccp_alpha=i)
    clf.fit(X_train,y_train)
    y_train_pred=clf.predict(X_train)
    y_test_pred=clf.predict(X_test)
    accuracy_train.append(accuracy_score(y_train,y_train_pred))
    accuracy_test.append(accuracy_score(y_test,y_test_pred))

sn.set()
plt.figure(figsize = (20, 10))
sn.lineplot(x=alphas, y=accuracy_train, label="TrainAccuracy")
sn.lineplot(x=alphas, y=accuracy_test, label="TestAccuracy")
plt.xticks(ticks=np.arange(0.00,0.08,0.011))
plt.yticks(ticks=np.arange(0.70,1.0,0.02))
plt. show()
```

Setting as criterion entropy, and keeping the default parameters, we determine the values of accuracy of all alphas.

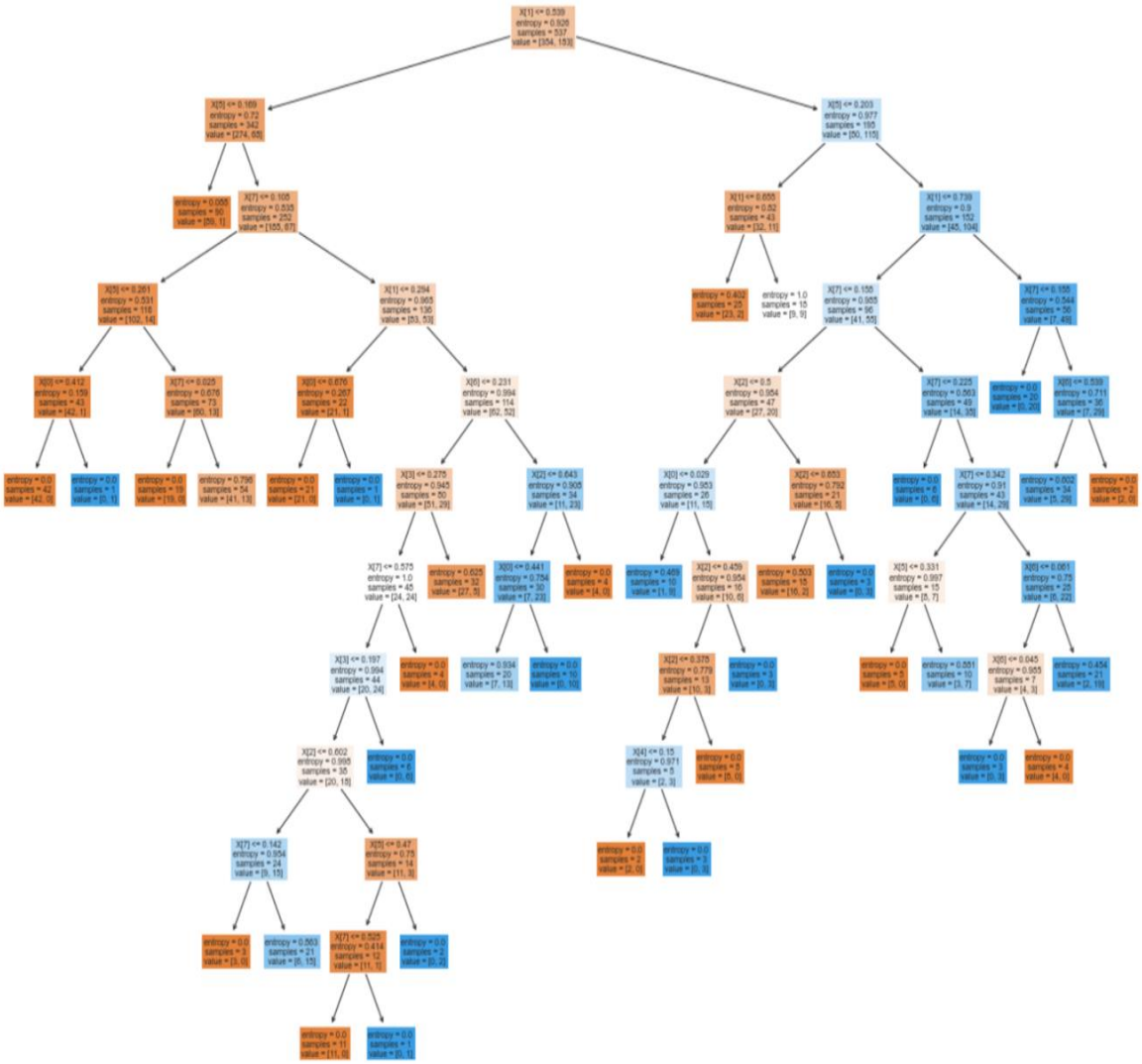


When ccp_alpha is zero, the tree overfits, leading to 100% of training accuracy, and 73.16% of testing accuracy. Setting ccp_alpha = 0.00887155, reduce the training accuracy to 91.25% but in the other hand maximizes the testing accuracy to 79.65%.

```
clf = DecisionTreeClassifier(criterion= 'entropy', ccp_alpha=0.00887155)
```



Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 118 over 146 (80.82%), while the number of “diabetic” people that have been correctly classified are 66 over 85 (77.65%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 19, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 28.



Ensamble Methods

The goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve robustness over a single estimator (generalizability).

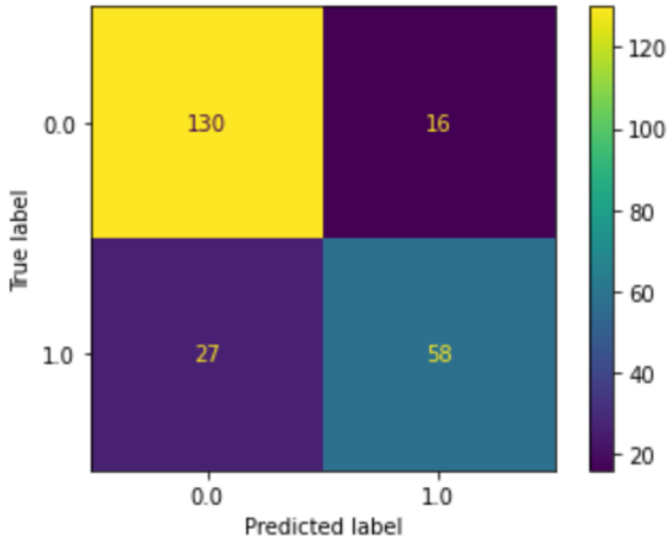
Two families of ensemble methods are usually distinguished: Averaging and boosting methods.

Random Forests

In random Forests, each tree in the ensemble is built from a sample drawn with replacement from the training set.

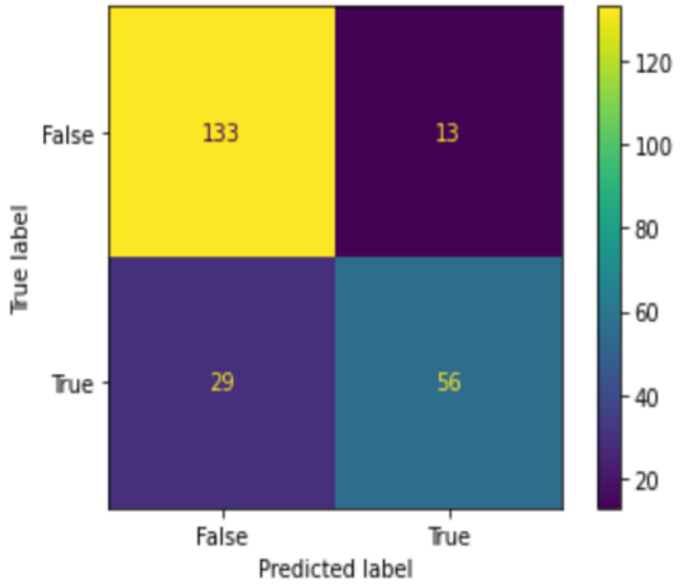
We have built two models with similar accuracy one that use as criterion gini and the other that use entropy. Let's compare the confusion matrix of both.

```
model = RandomForestClassifier(n_estimators=300, criterion="gini", max_features= 3)
```



The first model has an accuracy of 81.39%. specifically, looking at the Confusion matrix we see that, the number of "non-diabetic" people that have been correctly classified are 130 over 146 (89.04%), while the number of "diabetic" people that have been correctly classified are 58 over 85 (68.24%).

```
model = RandomForestClassifier(n_estimators=300, criterion="entropy", max_features= 3)
```



On the other hand, the second model has an accuracy of 81.82% which is slightly higher but looking at the confusion matrix we can see the fact that, the number of "non-diabetic" people that have been correctly classified are 133 over 146 (91.10%), while the number of "diabetic" people that have been correctly classified are 56 over 85 (65.88%).

Even if the second model has a higher accuracy, the first one has a higher ability of predict 'diabetic' people.

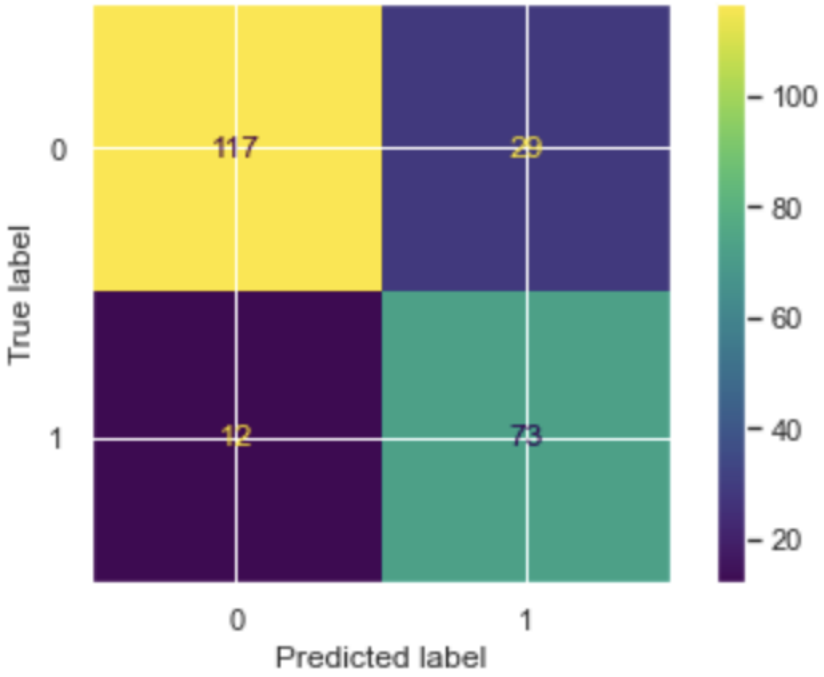
Bagging

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction.

Bagging methods work best with strong and complex models.

Using the Support Vector Machine as estimator, we obtain an accuracy of 82.25%.

It might seem in line with the previous obtained, but the interesting part is highlight on the confusion matrix.



Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 117 over 146 (80.14%), while the number of “diabetic” people that have been correctly classified are 73 over 85 (85.88%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 12, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 29.

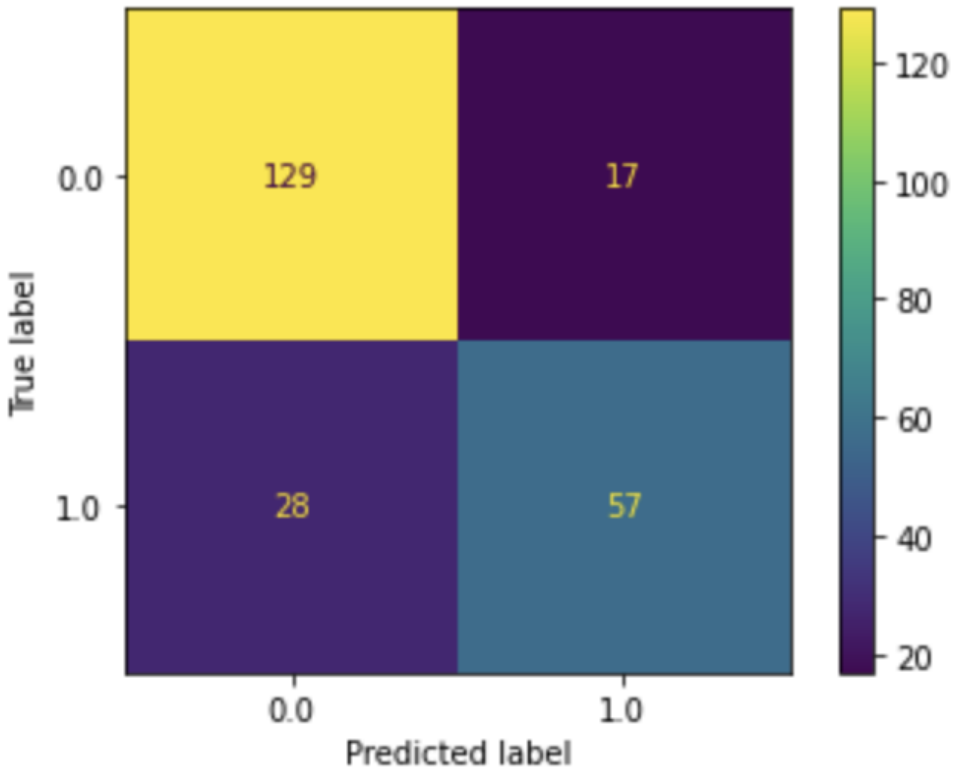
This is the best results that we have obtained as the ability of predict both ‘diabetic’ and ‘non-diabetic’, is over 80%.

AdaBoost

The core principle of AdaBoost is to fit a sequence of weak learners on repeatedly modified versions of the data.

We are going to use as estimator a simple Decision Tree with max_depth = 1.

```
adabdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), algorithm="SAMME.R", n_estimators=200)
```

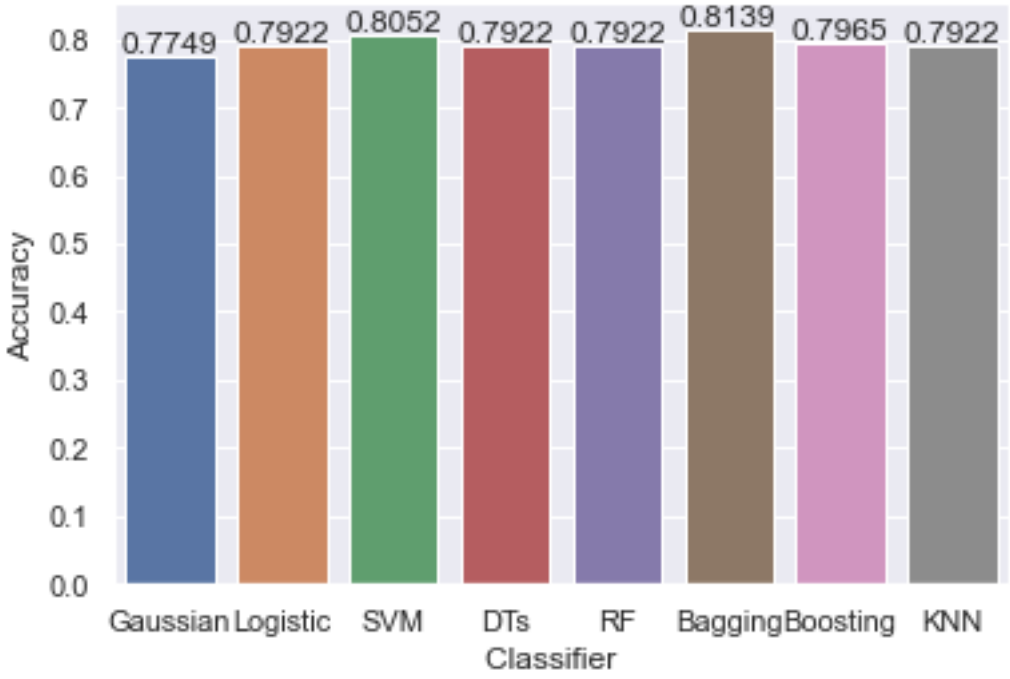


Looking at the Confusion matrix we see that, the number of “non-diabetic” people that have been correctly classified are 129 over 146 (88.36%), while the number of “diabetic” people that have been correctly classified are 56 over 85 (67.06%). On the other hand, the number of people classified as “non-diabetic” but who were diabetic is equals to 28, while the number of people classified as “diabetic” but who were “non-diabetic” is equals to 17.

Comparing the models

As we can see from the barplot below there are just two models with an accuracy higher than 80 %. Bagging is the method that performs in the best way also considering the values of Sensitivity and Specificity.

	Name	Score
0	Gaussian	0.774892
1	Logistic	0.792208
2	SVM	0.805195
3	DTs	0.792208
4	RF	0.792208
5	Bagging	0.813853
6	Boosting	0.796537
7	KNN	0.792208



Unsupervised Learning

Unsupervised learning uses machine learning algorithms to analyze and group unlabeled datasets, the goal of the algorithms is to discover hidden patterns or data groupings.

We categorize unsupervised learning models in two main tasks clustering and dimensionality reduction.

Clustering

Clustering is the search for structures and patterns within a dataset. Clustering is done whenever we want to find groups (or clusters) in the data.

We can distinguish various clustering approaches; some of which are listed below.

KMeans

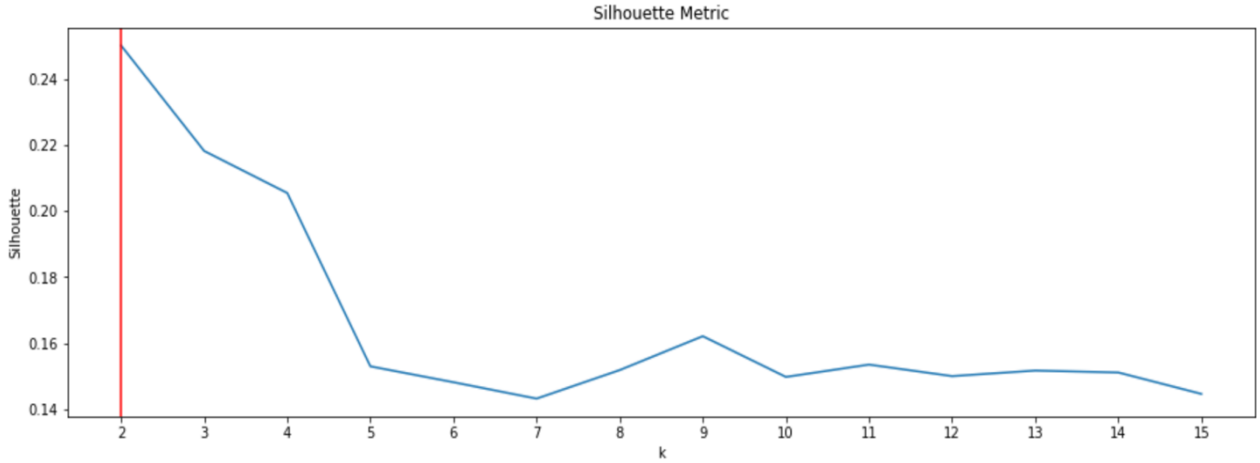
The **KMeans** algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the *inertia* or within-cluster sum-of-squares.

This algorithm requires the number of clusters to be specified.

Using the silhouette method, is possible to determine the number of clusters with better defined clusters.

```
k_to_test = range(2, 16, 1)
silhouette_scores = {}

for k in k_to_test:
    kmeans_k = KMeans( n_clusters =k, algorithm= "full")
    kmeans_k.fit(X.drop("cluster", axis = 1))
    labels_k = kmeans_k.labels_
    score_k = metrics.silhouette_score(X.drop("cluster", axis = 1), labels_k)
    silhouette_scores[k] = score_k
    print("Kmeans with k = %d\tSS: %5.4f" % (k, score_k))
    print("\n")
```

The number of clusters with the higher silhouette score is the one with higher cohesion within elements of the same cluster and lower cohesion of elements that does not belong to the same cluster.



As, using the silhouette score we obtain that the suggested number of clusters is k=2, let's perform it.

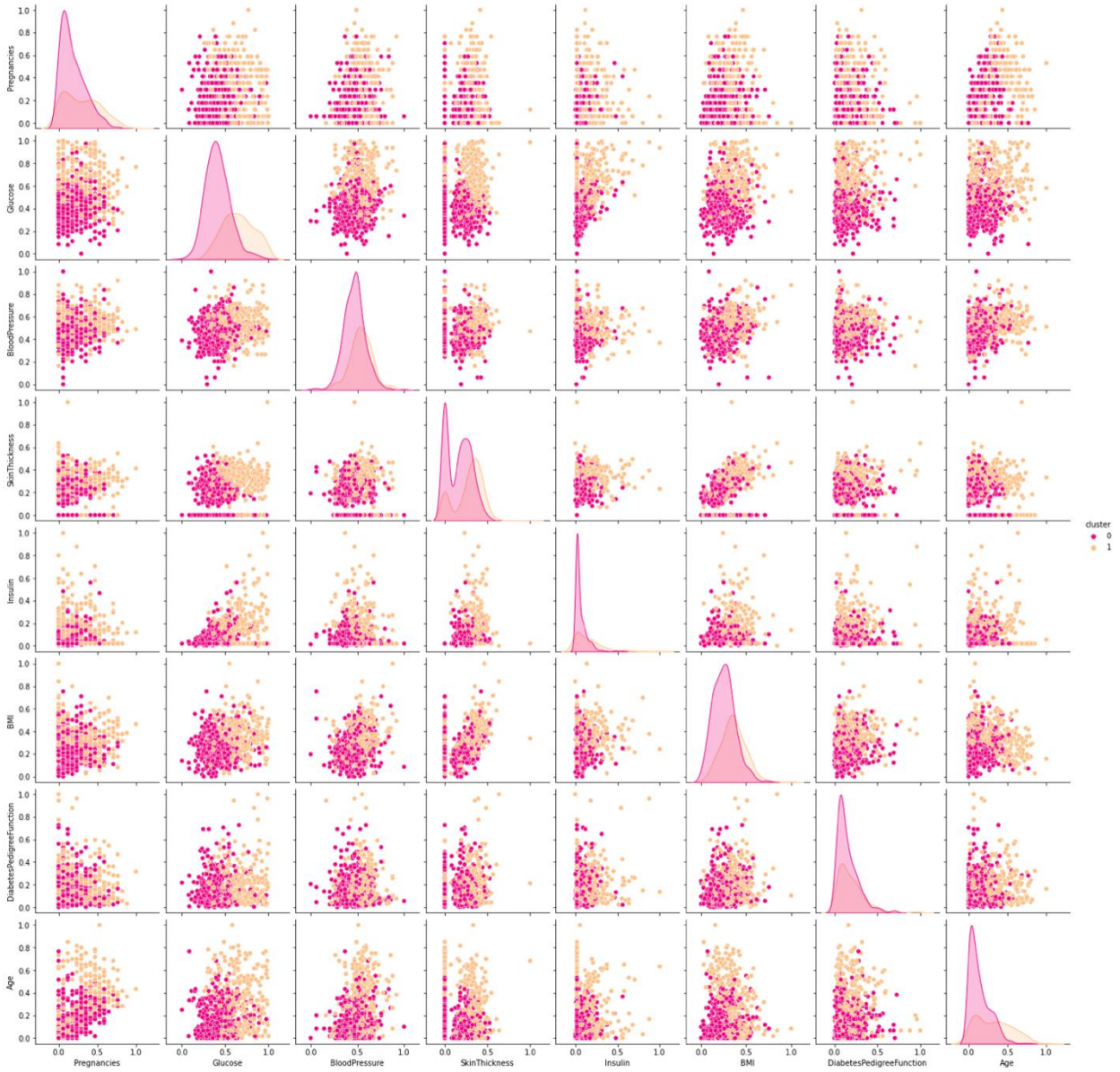
```
kmeans = KMeans(n_clusters=2, n_init=30)
```

Ones we have instantiate the model, we can fit it with our data.

```
kmeans = kmeans.fit(X)
```

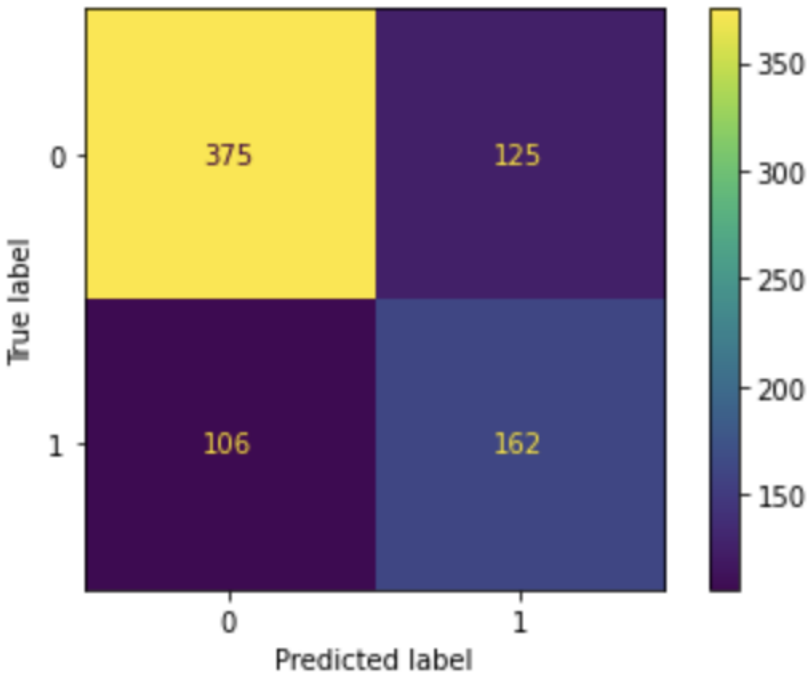
Now let's see to which cluster our observations have been assigned to. In order to do that, we add a new columns named "cluster" with the label of the class assignment and then plot it.

```
X["cluster"] = kmeans.labels_
```



As we can see, we can notice that the two cluster are not well separated, so we cannot expect that the evaluation of the clusters will be strictly related to the original labels.

Comparing the result of the clusterization and the true labels, we obtain the following confusion matrix.



Using the KMeans algorithm, the 69.92% of the data is correctly clustered with respect to the target variable.

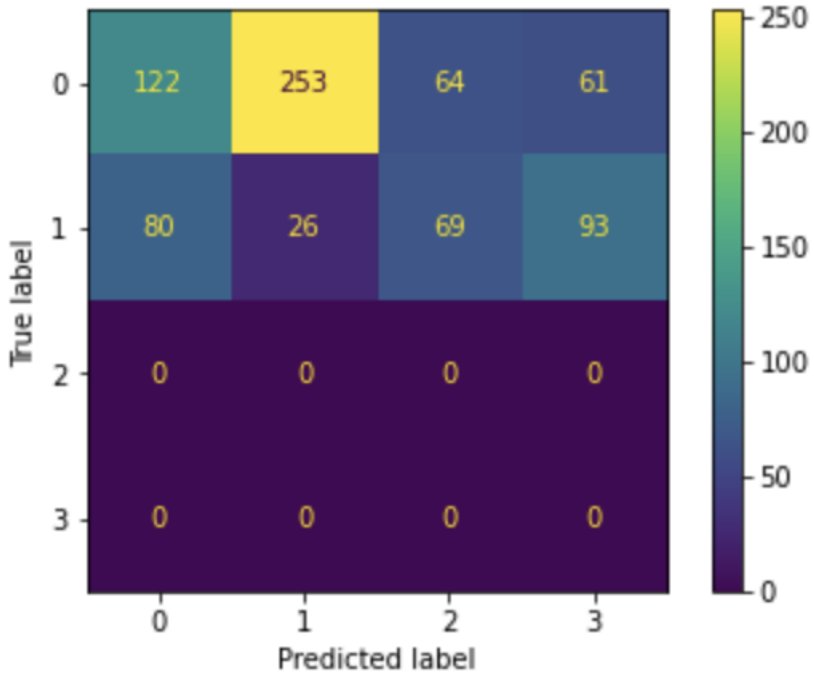
Affinity Propagation

- Affinity Propagation creates clusters by sending messages between pairs of samples until convergence. This algorithm can be interesting as it chooses the number of clusters based on the data provided.

```
af = AffinityPropagation(max_iter = 250, preference=-10)
af = af.fit(X)

labels = af.labels_

X["cluster"] = labels

sn.pairplot(data = X, hue = "cluster", palette = "Accent_r")
plt.show()
```

Once we performed the algorithm and fit with our data, we can take the labels and create a new column called cluster, in order to visualize and analyze it.


Pregnancies
Glucose
BloodPressure
SkinThickness
Insulin
BMI
DiabetesPedigreeFunction
Age
cluster
0
1
2
3



From this confusion matrix we can see that the last two rows of the matrix are zeros.

This is in line with the true labels of our data that has as target values only 0 and 1. On the other hand, using affinity propagation the algorithm divides the observations in 4 clusters.

We can see that there is a tendency to locate target values 0 more in clusters 0 and 1, together they have 375 zeros; On the other hand, clusters 2 and 3 tends to have more values of the target variable 1, together they have 162 values of 1. Whit these considerations, is quite evident that if we merge clusters 0 and 1 and cluster 2 and 3, we reach the same result as for the KMeans.

The affinity propagation algorithm chose as number of clusters 4 suggesting us that there is some path in the data that is better explained with this choice.

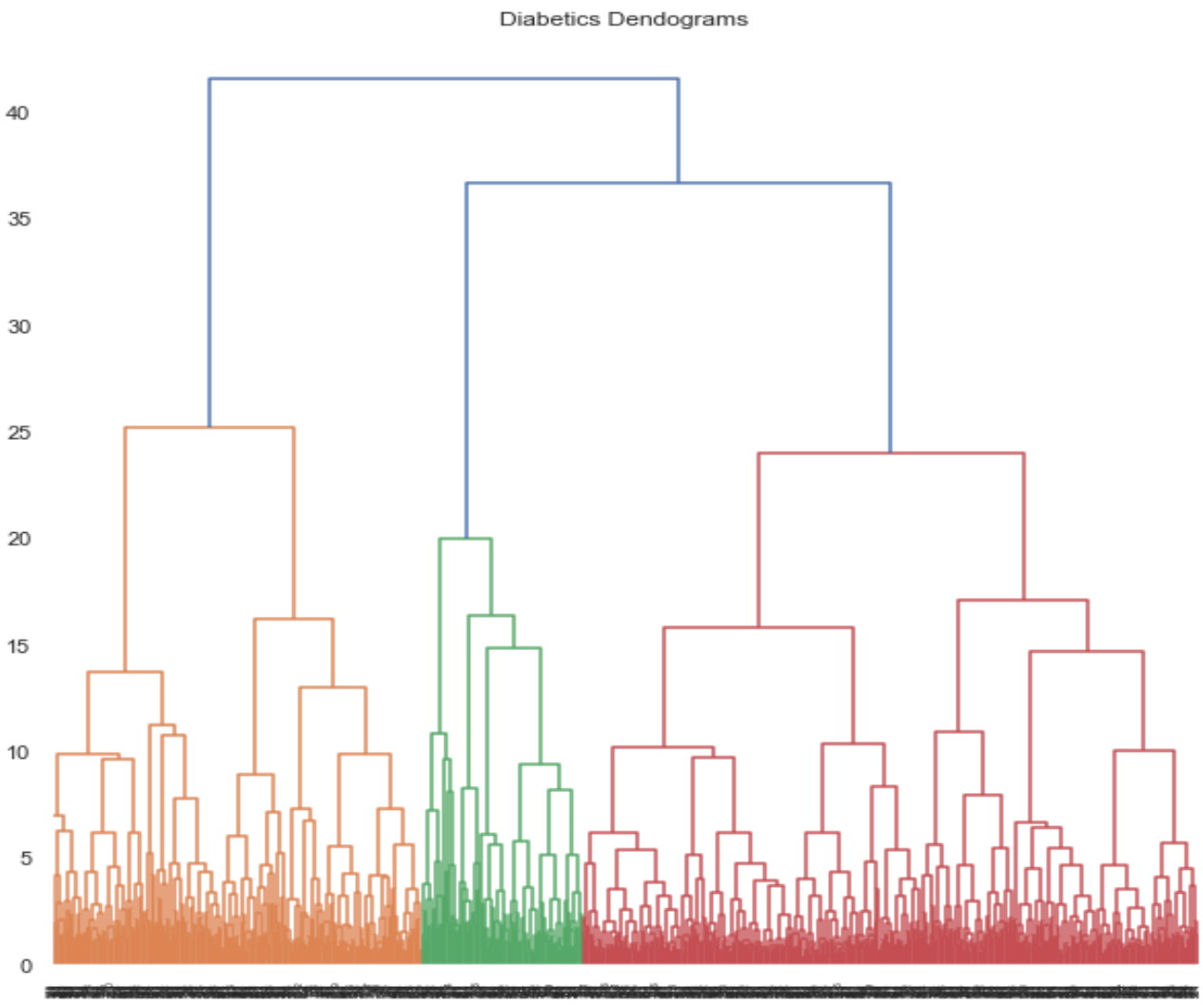
Hierarchical Clustering

Hierarchical clustering is a method of CA which seeks to build a hierarchy of clusters, it doesn't require the number of cluster K as an input. The strategy we will use is the "Agglomerative", that is a bottom-up approach where each observation starts in its own cluster (leaf), and pairs if clusters are merged as one moves up the hierarchy. This process goes on until there is just one single big cluster (root). In this

case we have 768 units where each one represents one leaf, the algorithm merges pairs of clusters with the lowest dissimilarity distance according to the chosen linkage method and it builds a cluster hierarchy that is commonly displayed as a tree diagram called dendogram. The linkage method we used is the “ward” that instead of measuring the distance directly, it analyses the variance of clusters.

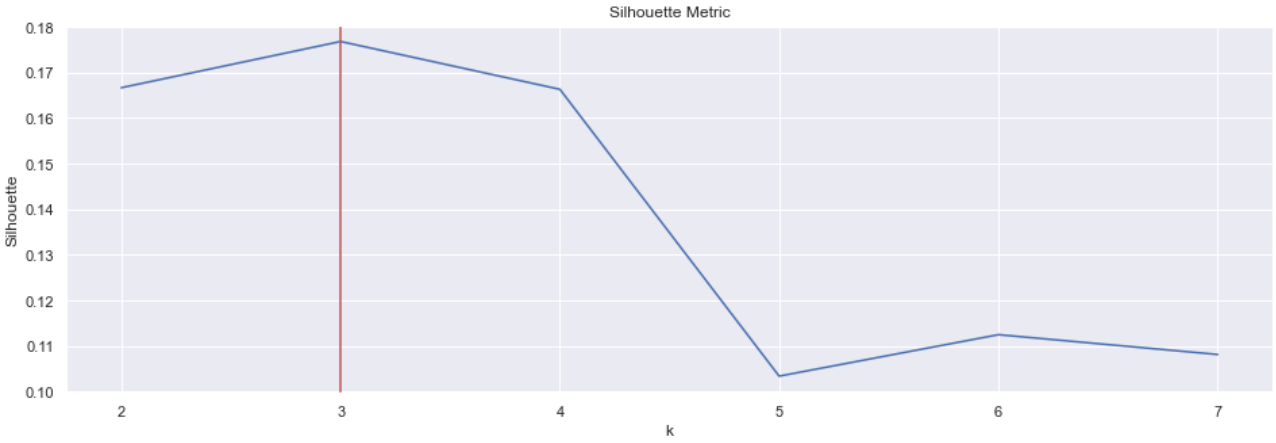
From the Dendogram below we can see that the plot seems to suggest a number of cluster equals to 2, 3 or 5. The number of clusters depends from the height where we decide to cut our dendogram.

```
import scipy.cluster.hierarchy as shc

plt.figure(figsize=(10, 10))
plt.title("Diabetics Dendograms")
dend = shc.dendrogram(shc.linkage(X, method='ward'))
```



Choice of number of clusters

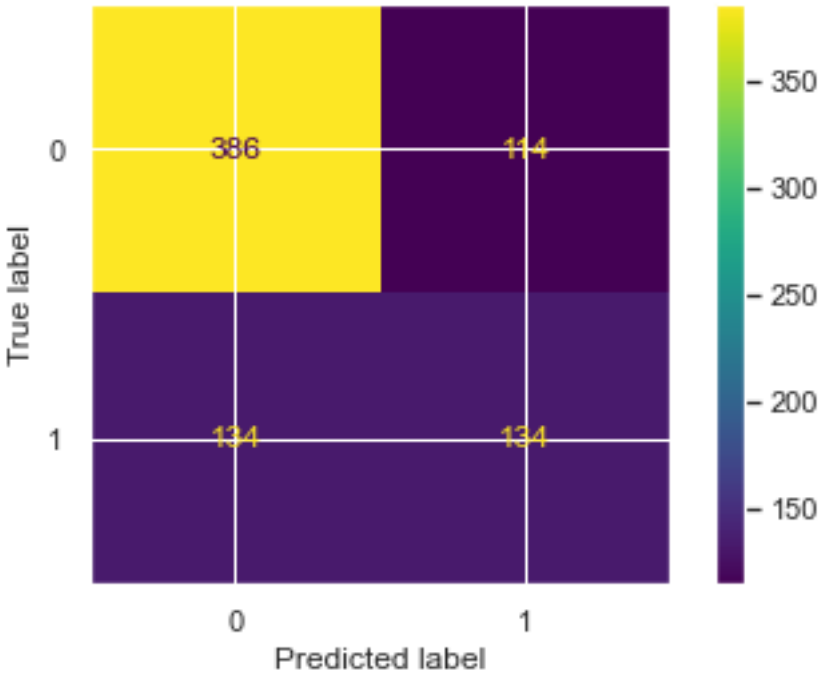
Looking a the silhouette we can see that there is an high value in correspondance of 3 clusters that is the suggested value, but there is an high value also in correspondance of 2, we are going to try which one give us a good result.



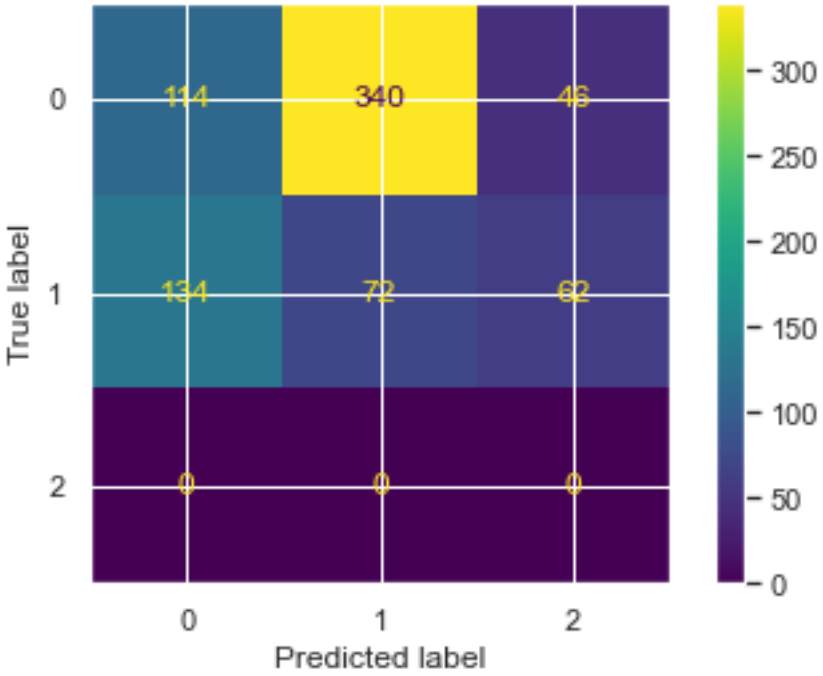
We decided firstly to cut the Dendogram in order to have just 2 clusters :

```
from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
out=cluster.fit_predict(X)
```

We compute the confusion Matrix with the target variable, but how we can see from the image below there isn't a clear distinction between the 2 groups, of course the cluster 0 contains an higher amount of zero, but this is not confirmed from the other cells of the matrix and considering that the value of 0 is much higher to respect to the class 1, there is no a real path during the clusterization compared with the target variable.



We will now have a look with 3 clusters and we are going to discover if here we have new relations:



Considering 3 clusters here cluster 1 seems to clustering the target variable 0, people non-Diabetic , but also here still no clear evidence about the comparison between the target variable and the clusters because the other two clusters split more or less in an equals way the true labels 0-1.

Principal Component Analysis

Principal Component Analysi PCA is a dimensionality reduction method that has as a main goal that one to reduce the dimensionality of the dataset transforming a set of variables into a smaller one that still contains most of the information. The first step we did it was to standardize the data, then looking at the correlation matrix we don't have strong correlation within the variables, so this can let move us to think we don't have redundancy in our data or case of multicollinearity. Firstly we used the StandardScaler() option to ensure that all the features have a standard deviation of 1 and a mean of 0. In this example we specified to create 8 principle components using Skit learn decomposition.PCA library. Here we can see the 8 principal components and 768 observations:

```
pca = PCA(n_components=8)
Principal_components=pca.fit_transform(XX)
pca_df = pd.DataFrame(data = Principal_components,
                      columns = ['PC 1', 'PC 2', 'PC 3', 'PC 4', 'PC 5', 'PC 6','PC7','PC8'])
print(pca_df)
```

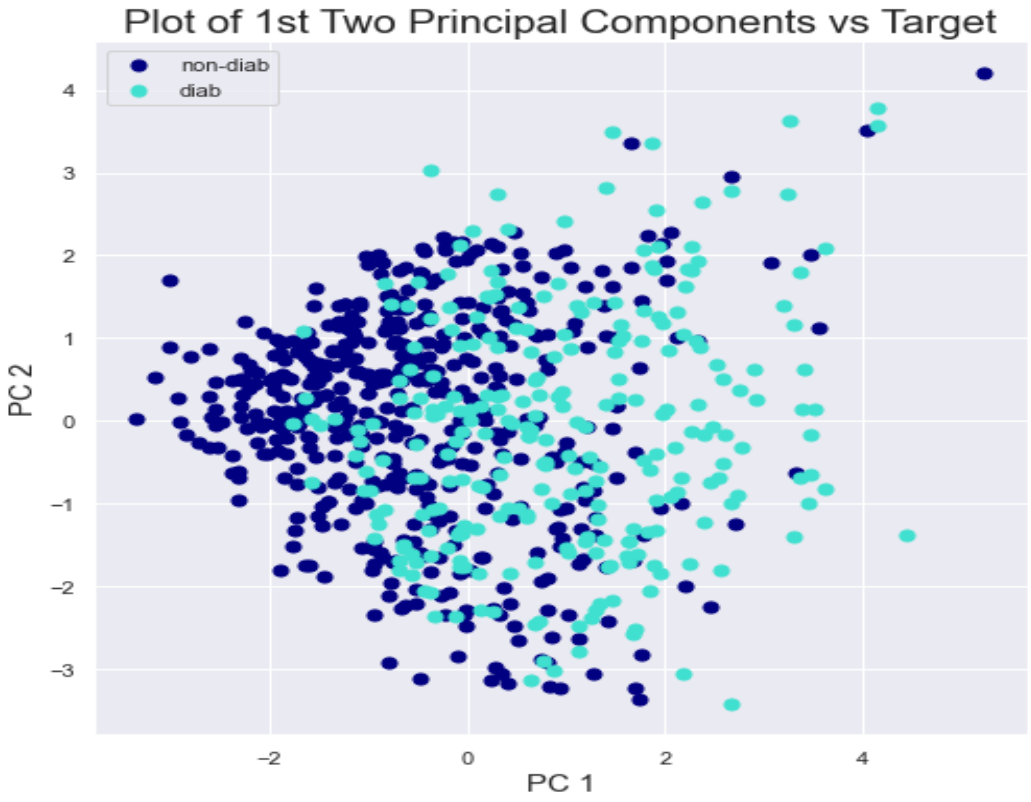
```
         PC 1      PC 2      PC 3      PC 4      PC 5      PC 6       PC7  \
0    1.295132 -0.723744 -0.308086  0.535685  0.443034  0.385007  1.305837
1   -1.543666  0.439956  0.248175  0.200739  0.579120 -0.596095  0.742527
2   -0.066408 -1.348256 -2.028268 -0.021322 -1.033353  0.992484  0.136716
3   -1.714696  0.842474  0.354911 -0.600905  0.408396 -0.441238 -0.061436
4    1.468775  3.488429 -2.416858  4.249291 -0.784960  1.664609  0.259668
..        ...       ...       ...       ...       ...       ...       ...
763  1.785365 -1.371812 -0.076787 -0.327301  2.811258 -0.816173  0.603463
764 -0.314728  0.597545  0.847352 -0.160210 -0.043535  0.574815  0.418506
765 -0.469286 -0.232018 -0.319261 -0.679489  0.429586 -0.465455  0.035790
766 -0.978002 -0.849627 -0.712782 -0.198331 -0.648450  0.712912  0.661564
767 -1.288865  0.836447  0.816939 -0.001764  0.342788 -0.351684  0.462631

          PC8
0   -0.042248
1    0.266936
2   -1.497418
3    0.028097
4    0.922262
```

We then concatenated the observations with the target variable in order that each row can be associated to the correspondent target variable.

```
for_visual = pd.concat([pca_df, df['Outcome']], axis = 1)
print(for_visual)
```

```
          PC 1      PC 2      PC 3      PC 4      PC 5      PC 6       PC7
0     1.295132 -0.723744 -0.308086  0.535685  0.443034  0.385007  1.305837
1    -1.543666  0.439956  0.248175  0.200739  0.579120 -0.596095  0.742527
2    -0.066408 -1.348256 -2.028268 -0.021322 -1.033353  0.992484  0.136716
3    -1.714696  0.842474  0.354911 -0.600905  0.408396 -0.441238 -0.061436
4     1.468775  3.488429 -2.416858  4.249291 -0.784960  1.664609  0.259668
..         ...       ...       ...       ...       ...       ...       ...
763   1.785365 -1.371812 -0.076787 -0.327301  2.811258 -0.816173  0.603463
764  -0.314728  0.597545  0.847352 -0.160210 -0.043535  0.574815  0.418506
765  -0.469286 -0.232018 -0.319261 -0.679489  0.429586 -0.465455  0.035790
766  -0.978002 -0.849627 -0.712782 -0.198331 -0.648450  0.712912  0.661564
767  -1.288865  0.836447  0.816939 -0.001764  0.342788 -0.351684  0.462631

          PC8  Outcome
0   -0.042248        1
1    0.266936        0
2   -1.497418        1
3    0.028097        0
4    0.922262        1
..        ...      ...
763  0.799810        0
764 -0.087429        0
765 -0.550542        0
766  1.358963        1
767 -0.174518        0
```

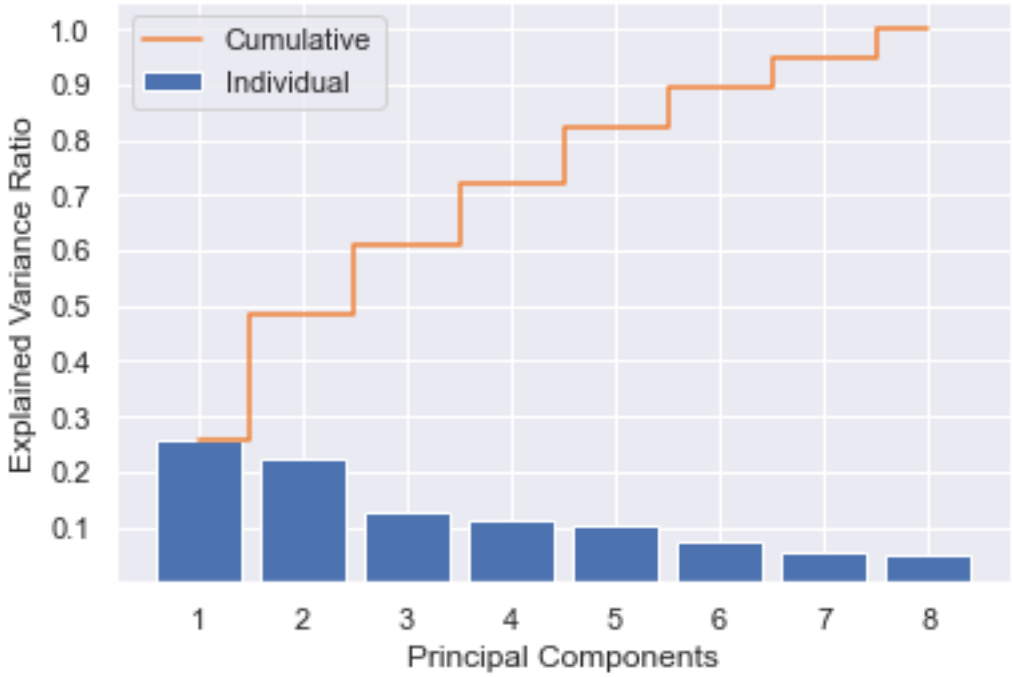
We can plot the bivariate between each pair of principal component. In the plot below we can simply see the first 2 Principal components versus the Target variable, and this means just printing the original data in a low dimensional space.



Select the Number of Principal Components

There are several tecnhique that allow us to determine the more appropriate number of Principal Components. The first is the **Scree Plot** in which the goal is to identify a possible presence of elbow:

We can see that in correspondance of 2, 3 or 5 clusters there is a relevant elbow that can suggest us to consider as the right number of principal components one of these. Considering that in order to reduce the number of features we must maintain at least 70% of the total variance it's a good practice having a look to the following plot:




```
Proportion of Variance Explained :  [0.25660594 0.22533412 0.12638702 0.11068762 0.10324803 0.07210951
 0.05478258 0.05084519]
Cumulative Prop. Variance Explained:  [0.25660594 0.48194005 0.60832708 0.71901469 0.82226272 0.89437223
 0.94915481 1.        ]
```

The Cumulative Variance Explained show us that in order to have a value of total variance explained we should consider at least 4 principal components even if would be better to choose also the fifth principal component to have a better result.

We can also see that following the **Kaiser's rule** it suggest to keep the eigenvalues with a value greater than 1. The eigenvalues are the sum of squares of the distance between the projected data points and the original along the eigenvector associated with a principal component:

```
print(pca.explained_variance_)
```

```
[2.05552396 1.80502323 1.01241443 0.88665543 0.82706113 0.57762822
 0.43883201 0.40729183]
```

This method show the the first 3 components are greter than 1 even if they explain just around 60 % of the total variance that can't be considered as a satisfactory level.

As a final solution we can see the loading matrix that show what is the contribute that each feature give to each principal component. For example PC1 has a strong

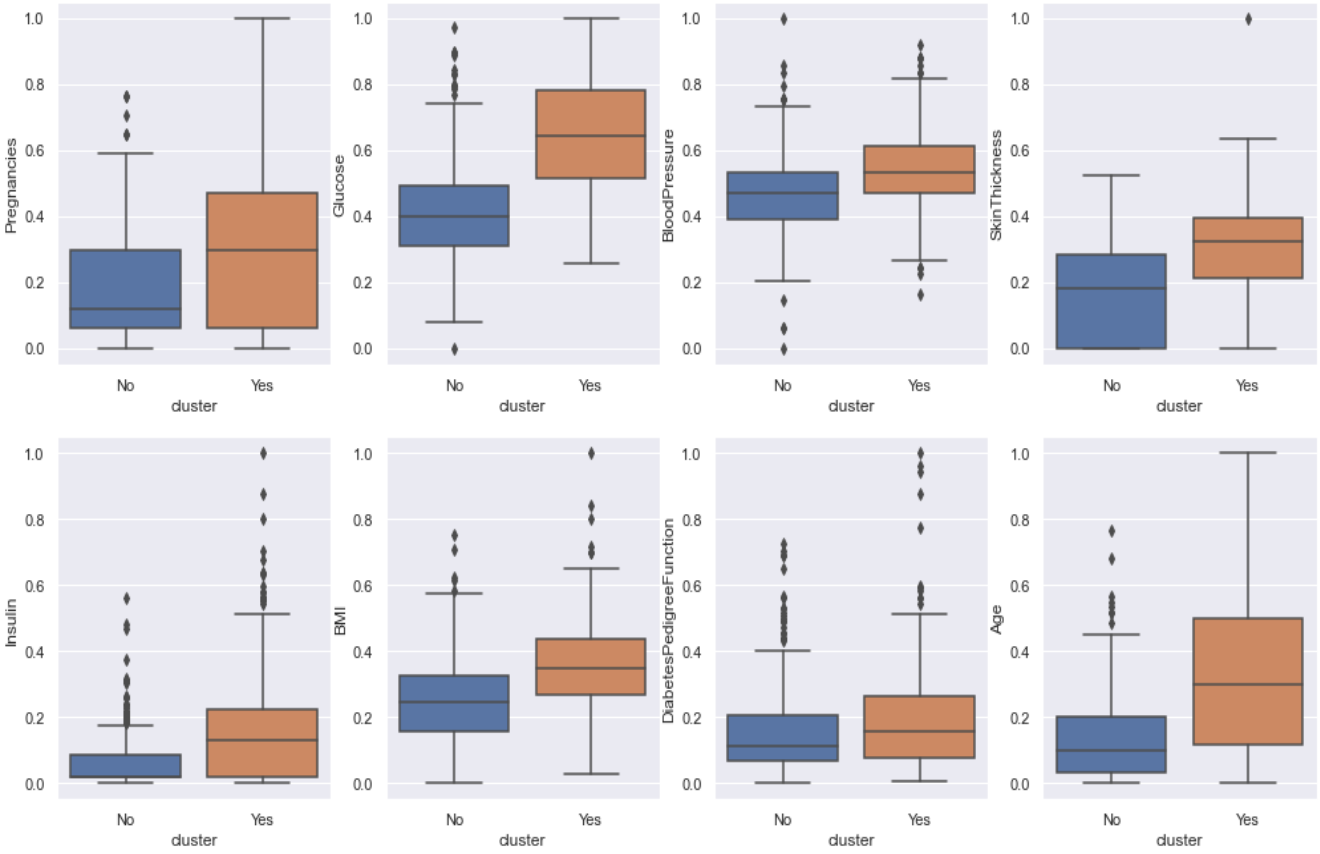
loading from Glucose (0.46), also BMI with a value of 0.42. for the PC2 there is a negative contribution of the feature Pregnancies (-0.49) and Age (-0.50).

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
Pregnancies	0.263802	-0.492512	-0.143693	0.111114	0.516432	0.181753	-0.367987	-0.468717
Glucose	0.464289	-0.020337	-0.312557	-0.365356	-0.470060	0.334920	0.317157	-0.345151
BloodPressure	0.360465	-0.274598	0.520996	-0.013945	-0.339818	-0.599330	-0.119539	-0.184575
SkinThickness	0.310657	0.447506	0.167055	0.035514	0.537323	-0.209177	0.531014	-0.244161
Insulin	0.362888	0.364150	-0.399240	-0.359775	0.120232	-0.334172	-0.504380	0.257586
BMI	0.424883	0.209155	0.536830	0.078515	0.012457	0.573352	-0.246432	0.302921
DiabetesPedigreeFunction	0.239905	0.220034	-0.316734	0.842515	-0.265430	-0.067571	-0.069640	-0.063332
Age	0.344431	-0.506544	-0.185519	0.085321	0.151938	-0.082909	0.384388	0.636687

Clustering Considerations

Understanding how the clustering works and what is the meaning of any cluster is not an easy challenge, what we have tried to explore was to compare the target variable with the cluster'results. **K-Means** seems to follow a path which appears to represents the binary classification that we have in our original dataset, and this is confirmed looking at the confusion matrix that shows an accuracy around 70.00 %.

What can be intersting is having a look to the boxplot of all the features differentiated by the clusters that the k-means method found using a k=2 that was the suggested number of clusters according to the Silhouette method :



Looking at the boxplot above we can see that, in each of the features we are considering, the value of the median of the cluster "Yes" is higher to respect to the median of the cluster "No". For ALL the variables there is a clear distinction and this means that comparing a cluster with higher value of Glucose, Insulin, Age, BMI, BloodPressure seems to be evident that we are facing two different clusters that corresponds to persons with a kind of disease, in particular:

- **Cluster "No"** : persons with NO disease and this means "healthy people"

- **Cluster "Yes"** : persons with an high level of Glucose, Insulin, BMI, SkinThikness, Advanced Age that induce us to think about persons with a kind of disease like Diabetes (that corresponds to the real scope of our dataset).