Diabetes Prediction

with machine learning classification algorithms

Git-hub URL:
https://github.com/Alikhalili56/ML_Diabetes-Prediction.git

Pima Indians Diabetes Database

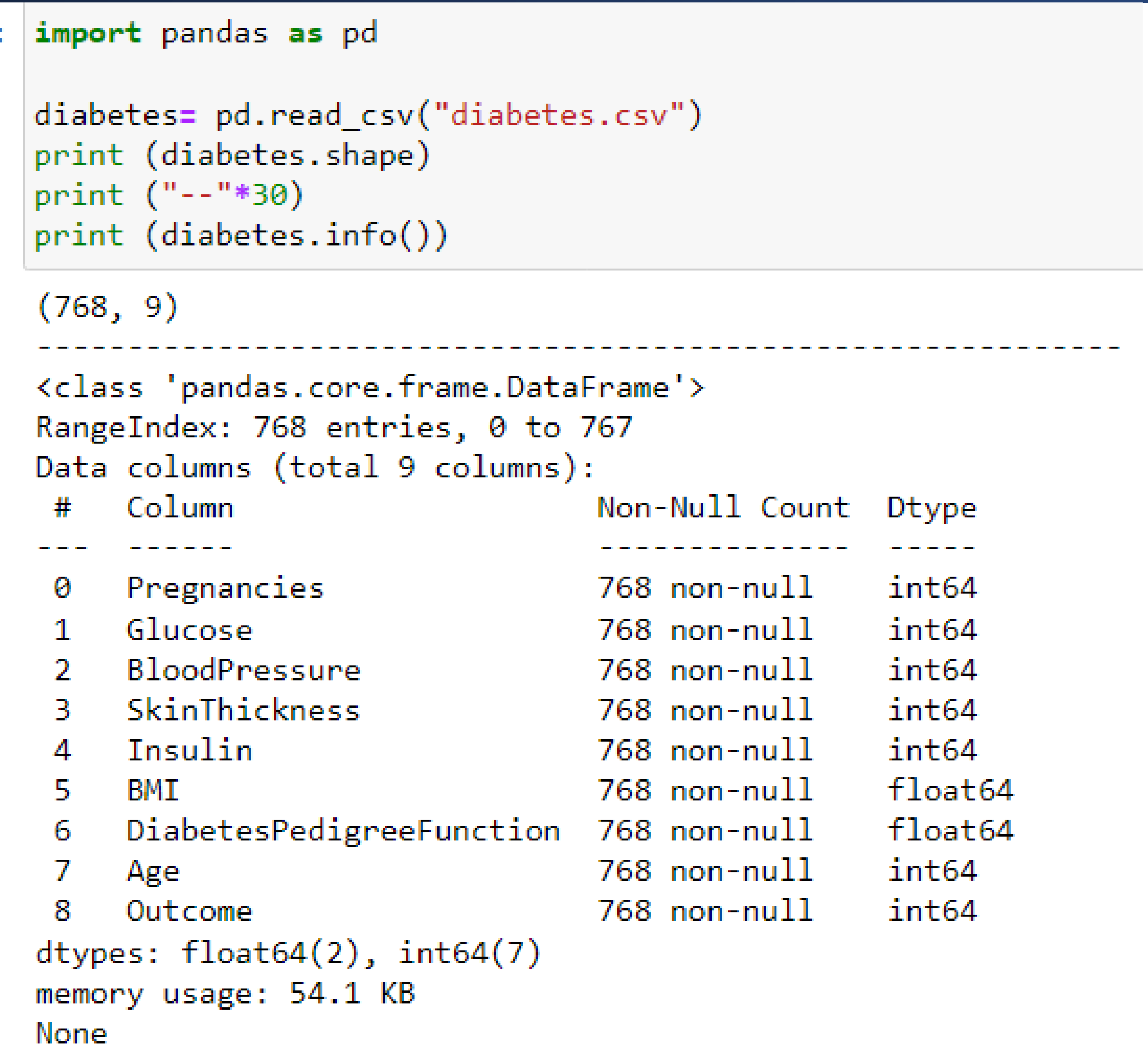
- **Context**
- This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.
- **Content**
- The datasets consists of several medical predictor variables and one target variable, `Outcome`. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

Data overview

https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

We save our data in a csv file named : diabetes.csv

Data overview

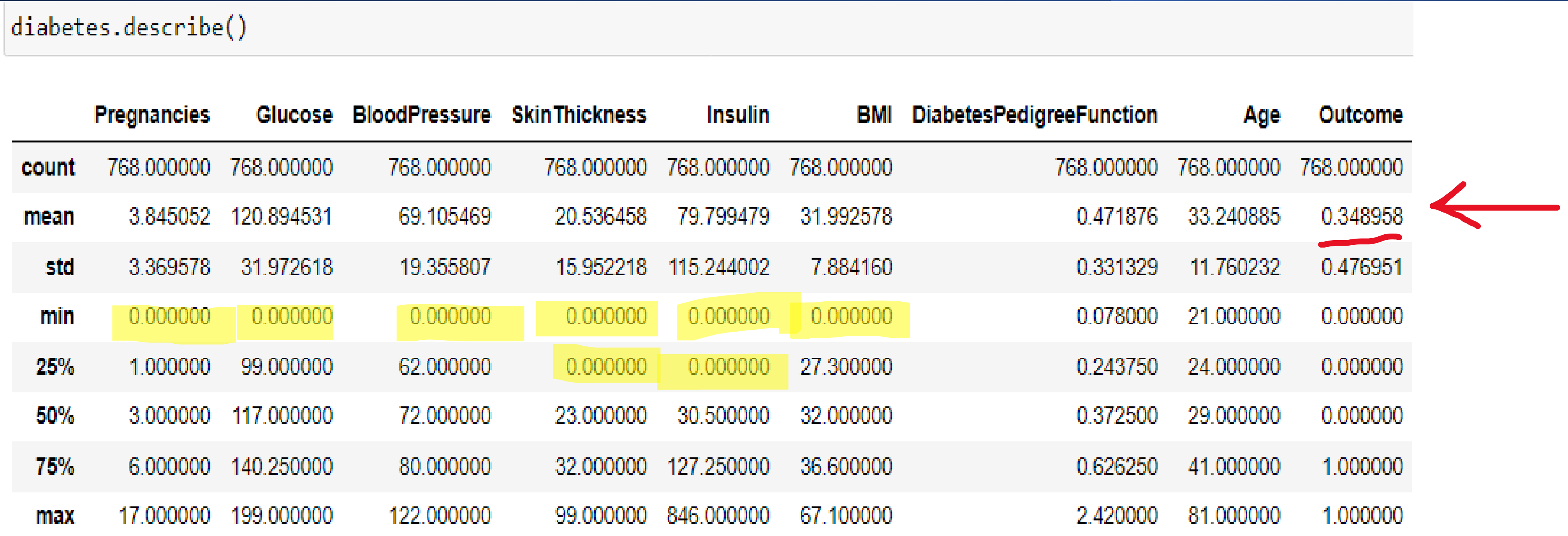


```
import pandas as pd

diabetes= pd.read_csv("diabetes.csv")
print (diabetes.shape)
print ("--"*30)
print (diabetes.info())
```

```
(768, 9)
------------------------------------------------------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
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  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
None
```

The output shows that there are no missing values in the dataset and the data types of the columns are either integers or floats.

Data Quality Issues



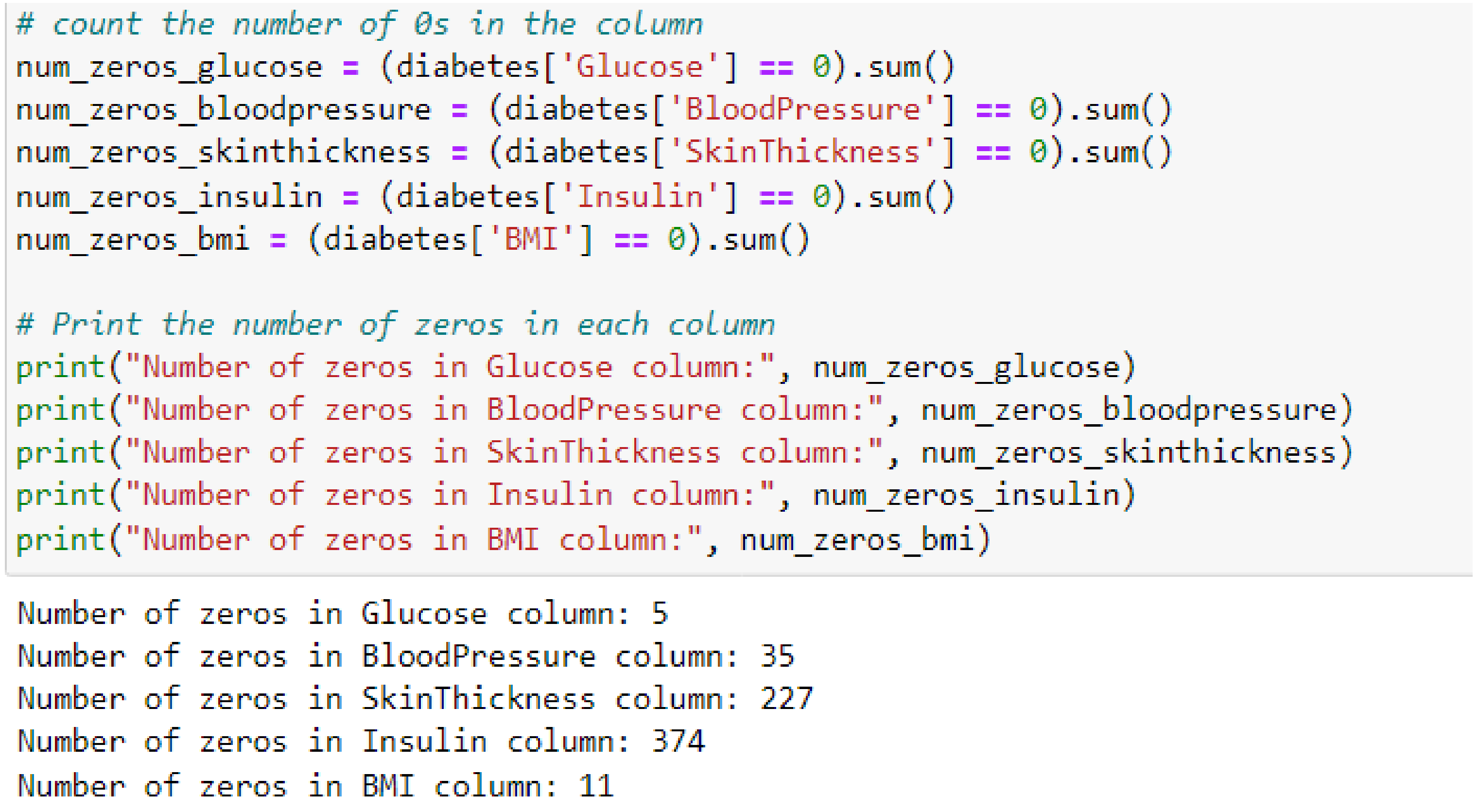
```
diabetes.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

The **`describe()`** method provides a statistical summary of the numerical columns of the "diabetes" dataset.

However, it is important to note that some columns have a minimum value of zero, such as "Glucose", "Blood Pressure", "Skin Thickness", "Insulin", and "BMI". These values are unlikely to be accurate and may indicate missing or incomplete data that needs to be addressed before using the dataset for machine learning.

Data Quality Issues



```
# count the number of 0s in the column
num_zeros_glucose = (diabetes['Glucose'] == 0).sum()
num_zeros_bloodpressure = (diabetes['BloodPressure'] == 0).sum()
num_zeros_skinthickness = (diabetes['SkinThickness'] == 0).sum()
num_zeros_insulin = (diabetes['Insulin'] == 0).sum()
num_zeros_bmi = (diabetes['BMI'] == 0).sum()

# Print the number of zeros in each column
print("Number of zeros in Glucose column:", num_zeros_glucose)
print("Number of zeros in BloodPressure column:", num_zeros_bloodpressure)
print("Number of zeros in SkinThickness column:", num_zeros_skinthickness)
print("Number of zeros in Insulin column:", num_zeros_insulin)
print("Number of zeros in BMI column:", num_zeros_bmi)
```

```
Number of zeros in Glucose column: 5
Number of zeros in BloodPressure column: 35
Number of zeros in SkinThickness column: 227
Number of zeros in Insulin column: 374
Number of zeros in BMI column: 11
```

Now, we can see the problem. I am not a doctor but I imagine that having a Blood Pressure of 0 is equivalent to be dead. Also, a Skin Thickness, insulin, glucose or BMI of 0 do not correlate good with the fact of being a living being.
Because of that, the data is NOT clean, and we need to take action.

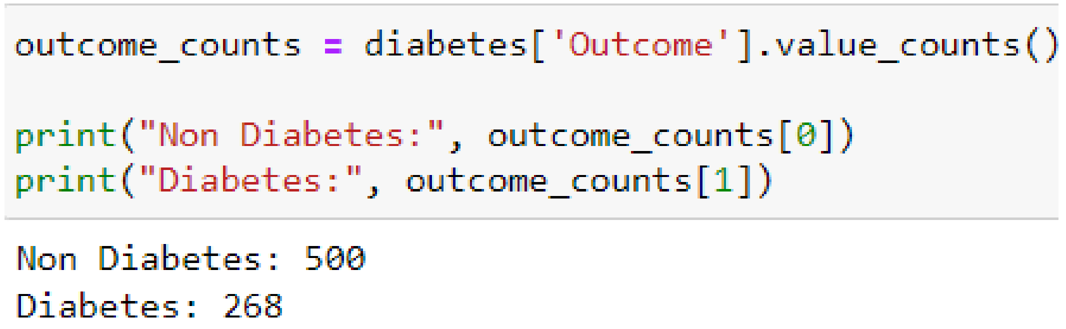
Cleaning Data

- Before building our model we should clean the data:
  I tried 2 different methods:
- 1- Deleting some rows consisting 0 values in Glucose, Blood Pressure or BMI using EXCEL.
  2- Replacing rows with 0 values in Glucose, Blood Pressure, BMI, Skin Thickness, Insulin with the median values.

- But, first we are going to train our Random Forest model with uncleaned data then compare the model that trained with cleaned data.

Data outcome distribution

- Is our dataset balanced?



```
outcome_counts = diabetes['Outcome'].value_counts()

print("Non Diabetes:", outcome_counts[0])
print("Diabetes:", outcome_counts[1])
```

```
Non Diabetes: 500
Diabetes: 268
```

This is an example of imbalanced data, where the Diabetes class has significantly fewer samples than the Non-Diabetes class and the Non-Diabetes class is dominant. Specifically, the Diabetes class represents only around 35% of the total dataset, while the Non-Diabetes class represents around 65%.

This imbalance can pose challenges for machine learning algorithms because they may have difficulty identifying the minority class, in this case, the Diabetes class.

Possible Approaches

We consider 2 Approaches:

Training algorithms that might also work well with imbalanced data such as:
Random Forest, Support Vector Machine (SVM), KNN, Gradient Boosting, Nural Networks, ...

In addition to these algorithms, various preprocessing techniques such as oversampling, under sampling, and data augmentation can also be used to handle imbalanced data.

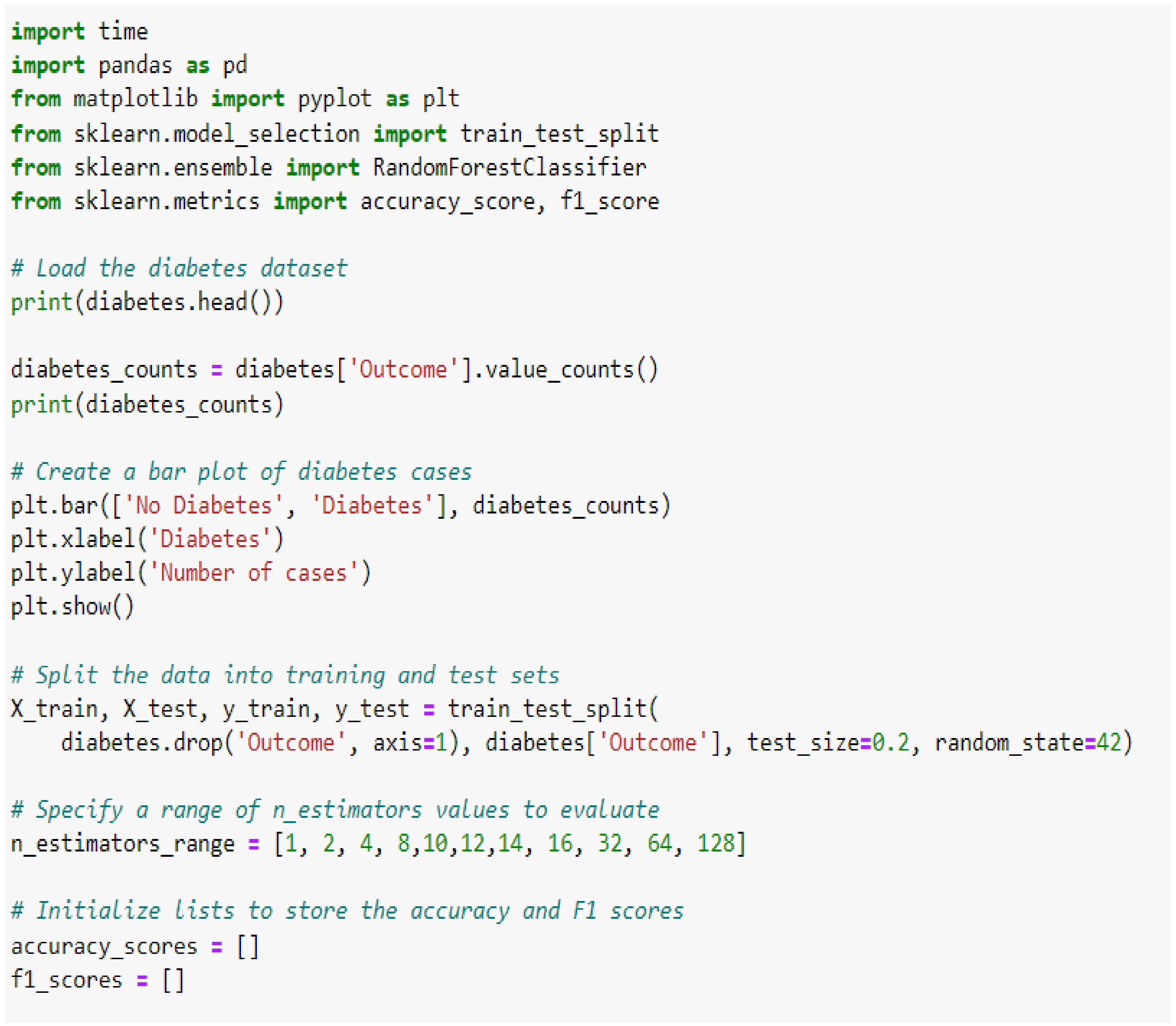
The algorithms under consideration for training in our study

Random Forest : Random Forest is an ensemble learning algorithm that combines multiple decision trees to create a powerful model for classification or regression. In this algorithm, each decision tree is constructed using a random subset of the features and the samples, and the final prediction is based on the average or majority vote of the predictions of the individual trees.

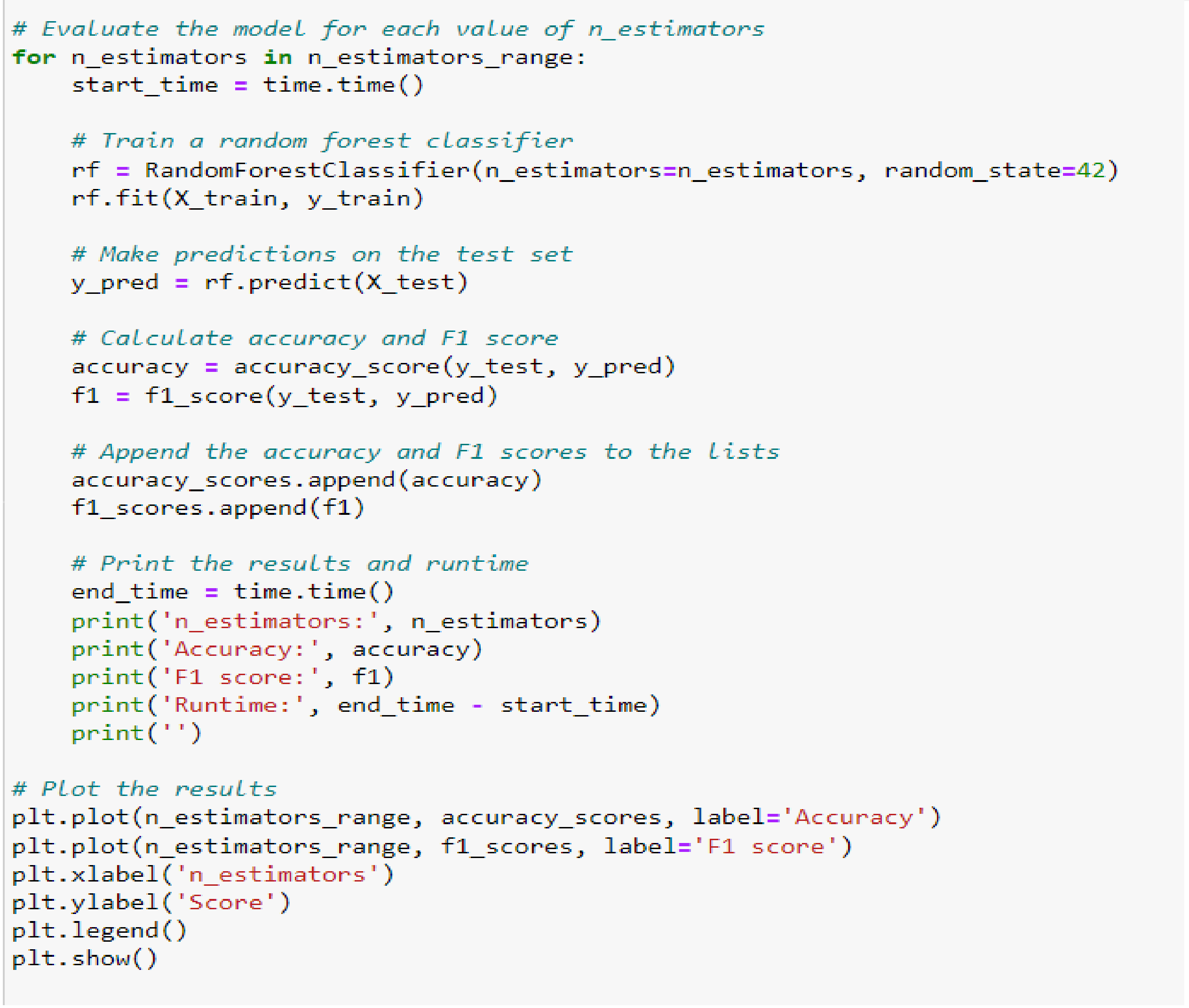
Support Vector Classifier (SVC): Support Vector Classifier (SVC) is a type of Support Vector Machine (SVM) algorithm that is used for classification tasks. SVC is a supervised learning algorithm that works by finding the best possible hyperplane in a high-dimensional space that separates the different classes.

K-Nearest Neighbors (KNN): K-Nearest Neighbors (KNN) is a popular supervised machine learning algorithm used for both classification and regression tasks. It is a non-parametric, lazy learning algorithm that works based on the concept of similarity between data points.

Training Random Forest classifier with Uncleaned Diabetes Data

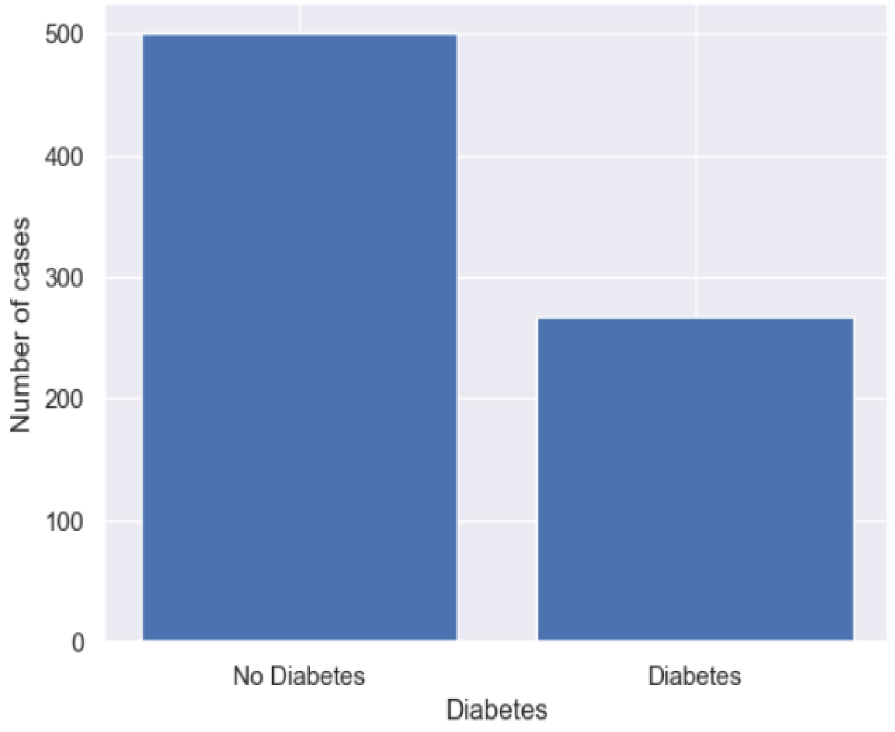


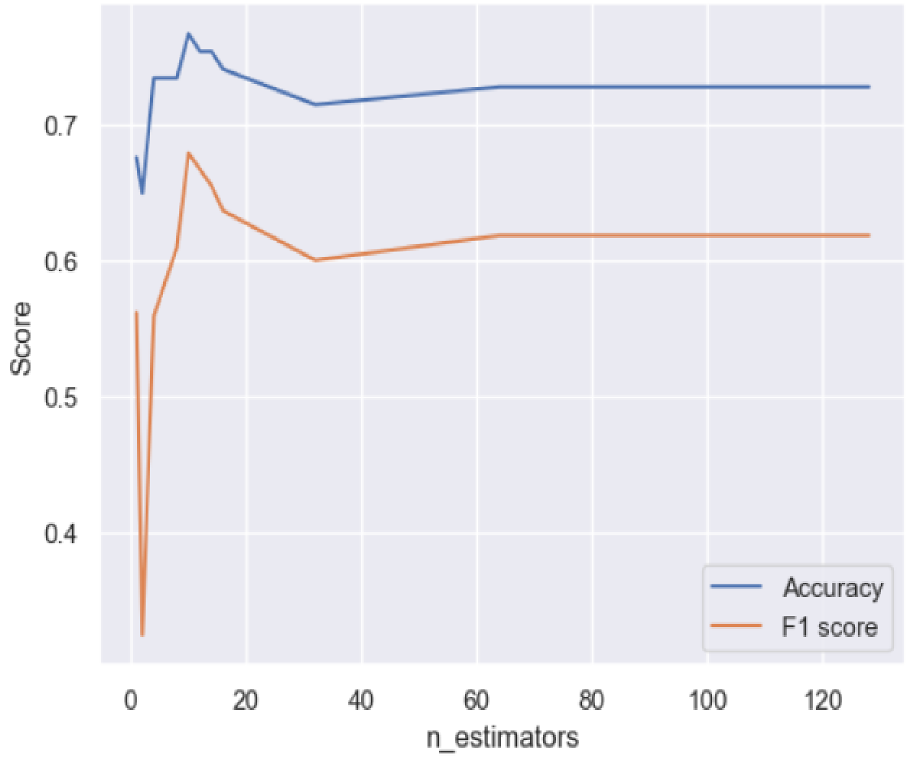
```python
import time
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score

# Load the diabetes dataset
print(diabetes.head())

diabetes_counts = diabetes['Outcome'].value_counts()
print(diabetes_counts)

# Create a bar plot of diabetes cases
plt.bar(['No Diabetes', 'Diabetes'], diabetes_counts)
plt.xlabel('Diabetes')
plt.ylabel('Number of cases')
plt.show()

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    diabetes.drop('Outcome', axis=1), diabetes['Outcome'], test_size=0.2, random_state=42)

# Specify a range of n_estimators values to evaluate
n_estimators_range = [1, 2, 4, 8,10,12,14, 16, 32, 64, 128]

# Initialize lists to store the accuracy and F1 scores
accuracy_scores = []
f1_scores = []
```



```python
# Evaluate the model for each value of n_estimators
for n_estimators in n_estimators_range:
    start_time = time.time()

    # Train a random forest classifier
    rf = RandomForestClassifier(n_estimators=n_estimators, random_state=42)
    rf.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = rf.predict(X_test)

    # Calculate accuracy and F1 score
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    # Append the accuracy and F1 scores to the lists
    accuracy_scores.append(accuracy)
    f1_scores.append(f1)

    # Print the results and runtime
    end_time = time.time()
    print('n_estimators:', n_estimators)
    print('Accuracy:', accuracy)
    print('F1 score:', f1)
    print('Runtime:', end_time - start_time)
    print('')

# Plot the results
plt.plot(n_estimators_range, accuracy_scores, label='Accuracy')
plt.plot(n_estimators_range, f1_scores, label='F1 score')
plt.xlabel('n_estimators')
plt.ylabel('Score')
plt.legend()
plt.show()
```

Output





```
n_estimators: 8
Accuracy: 0.7337662337662337
F1 score: 0.6095238095238096
Runtime: 0.018946647644042969

n_estimators: 10
Accuracy: 0.7662337662337663
F1 score: 0.6785714285714286
Runtime: 0.022152662277221680

n_estimators: 12
Accuracy: 0.7532467532467533
F1 score: 0.6666666666666665
Runtime: 0.028591632843017578
```

Comparing Different evaluation metrics

Accuracy, Precision, recall, and F1-score are metrics used in binary classification problems to evaluate the performance of a model. They are based on the confusion matrix, which is a table showing the number of true positives, false positives, true negatives, and false negatives.

Precision = True Positives / (True Positives + False Positives)

Recall = True Positives / (True Positives + False Negatives)

F1-score = 2 * (Precision * Recall) / (Precision + Recall)

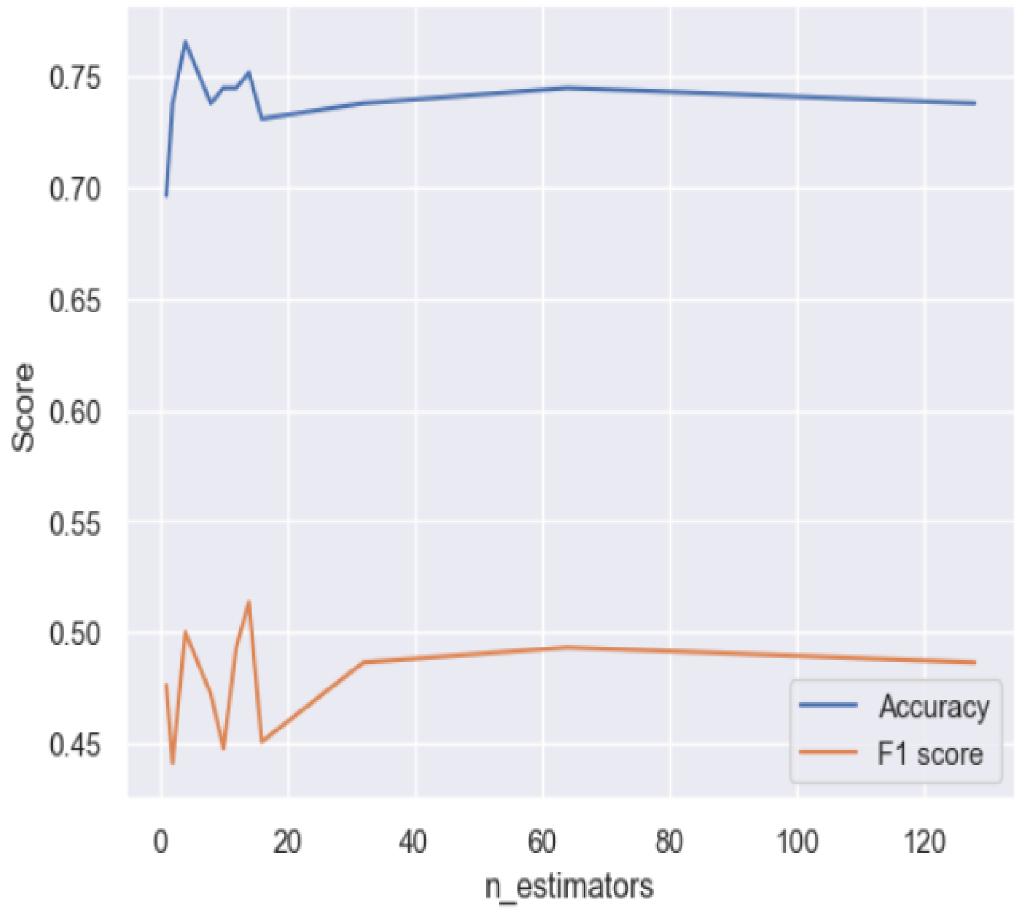
Accuracy = (true positives + true negatives) / (true positives + false positives + true negatives + false negatives)

F1 vs Accuracy

F1 score is a measure of the balance between precision and recall, which are both important metrics in classification problems. It considers both false positives and false negatives and provides a single score that summarizes the model's performance. F1 score is a better metric than accuracy in situations where there is class imbalance **(like our case)**, that is when the number of observations in each class is not equal. This is because accuracy can be misleading in such situations, as it can be high even if the model performs poorly on the minority class.

Accuracy, on the other hand, measures the proportion of correctly classified observations among all the observations. It is a good metric to use when the classes are balanced and there are no costs associated with misclassification. However, accuracy alone may not be a sufficient metric when the costs of false positives and false negatives are different.

Training Random Forest with cleaned data that Features with "0" values omitted

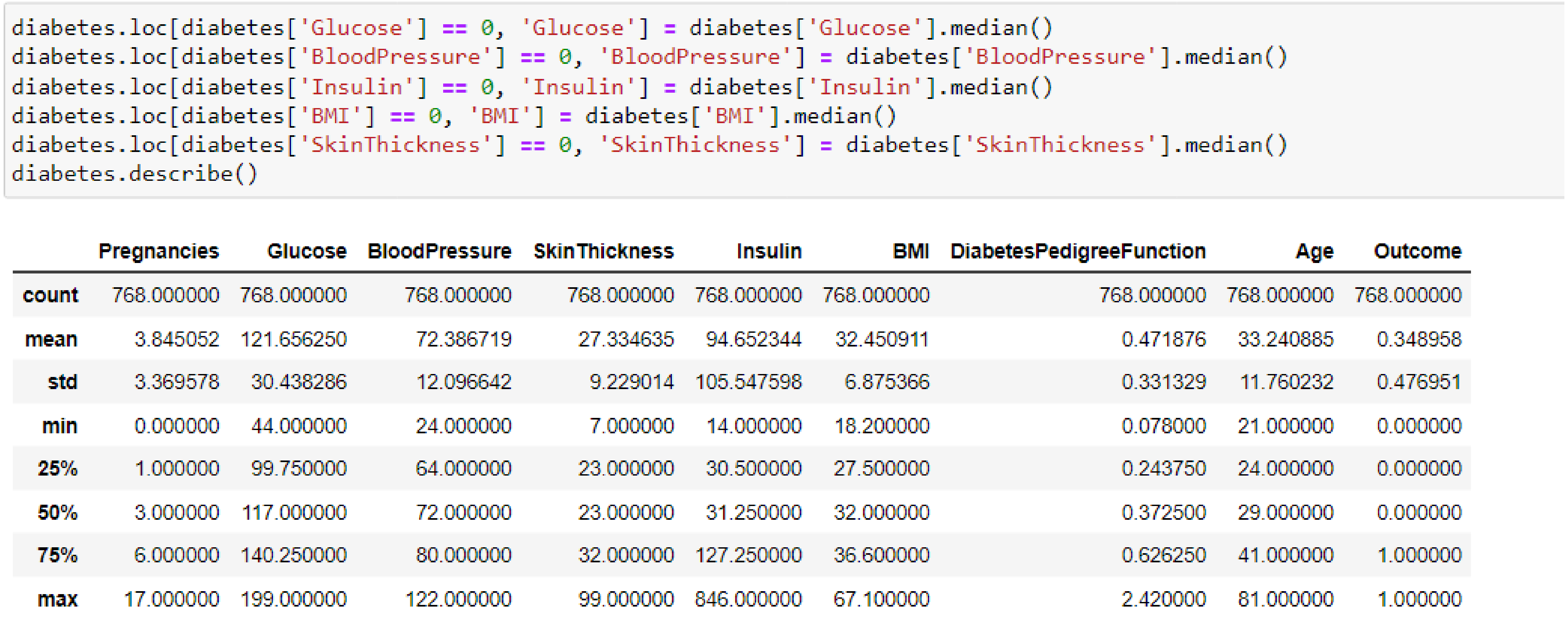


We used the same Random Forest algorithm to train this updated data, however, as we can observe the F1 score of the model drops dramatically, but why?

The F1 score is a metric that measures the balance between precision and recall of a binary classification model. Cleaning the data, i.e., removing some rows with values of 0, can have an impact on the F1 score if the removed rows contain significant information that the model requires for making accurate predictions.

For example, if the rows with values of 0 that were removed represented a certain subgroup of the population that had a higher incidence of diabetes, then removing those rows could lead to a drop in the F1 score since the model would no longer be able to capture the characteristics of that subgroup.

Replacing "0" values by median values

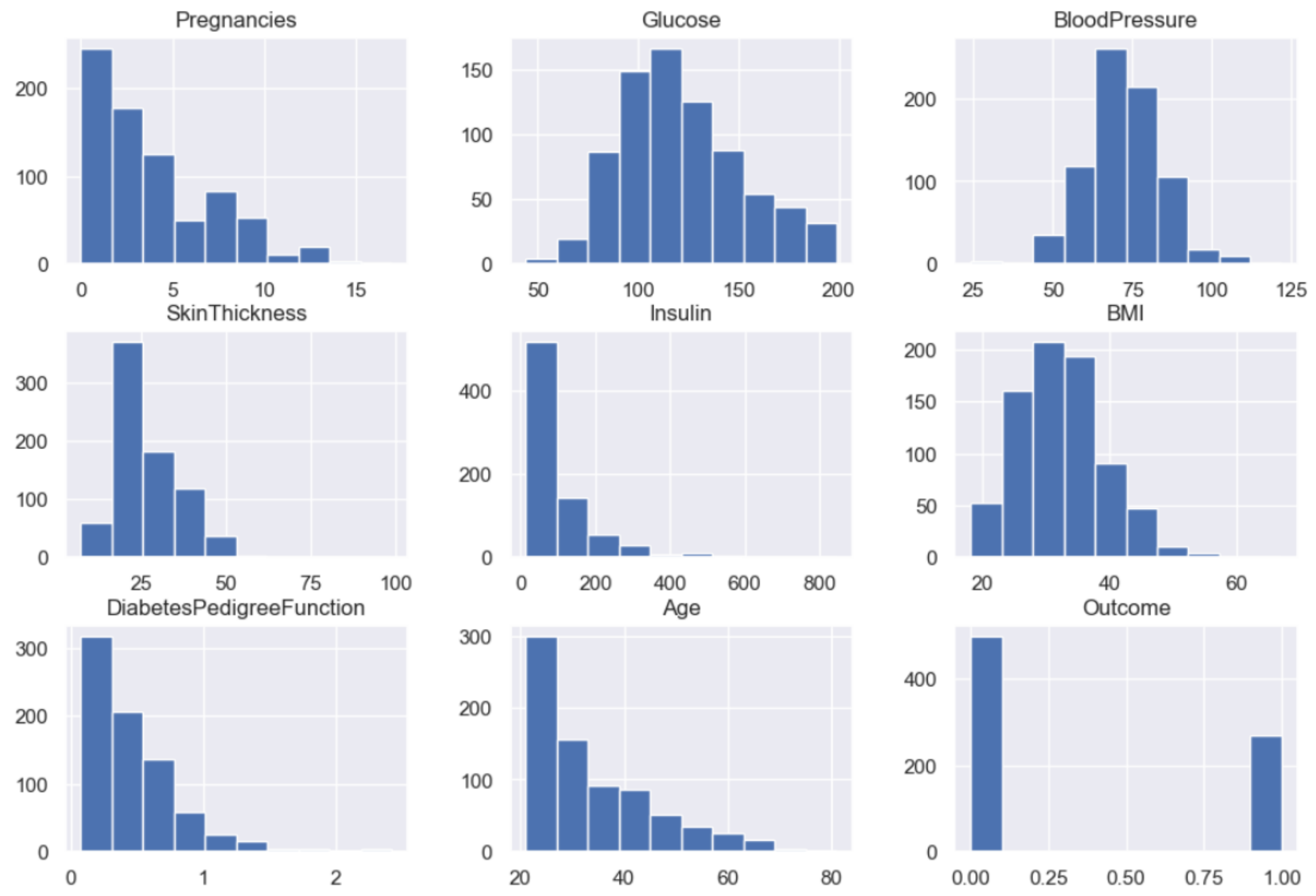


```python
diabetes.loc[diabetes['Glucose'] == 0, 'Glucose'] = diabetes['Glucose'].median()
diabetes.loc[diabetes['BloodPressure'] == 0, 'BloodPressure'] = diabetes['BloodPressure'].median()
diabetes.loc[diabetes['Insulin'] == 0, 'Insulin'] = diabetes['Insulin'].median()
diabetes.loc[diabetes['BMI'] == 0, 'BMI'] = diabetes['BMI'].median()
diabetes.loc[diabetes['SkinThickness'] == 0, 'SkinThickness'] = diabetes['SkinThickness'].median()
diabetes.describe()
```

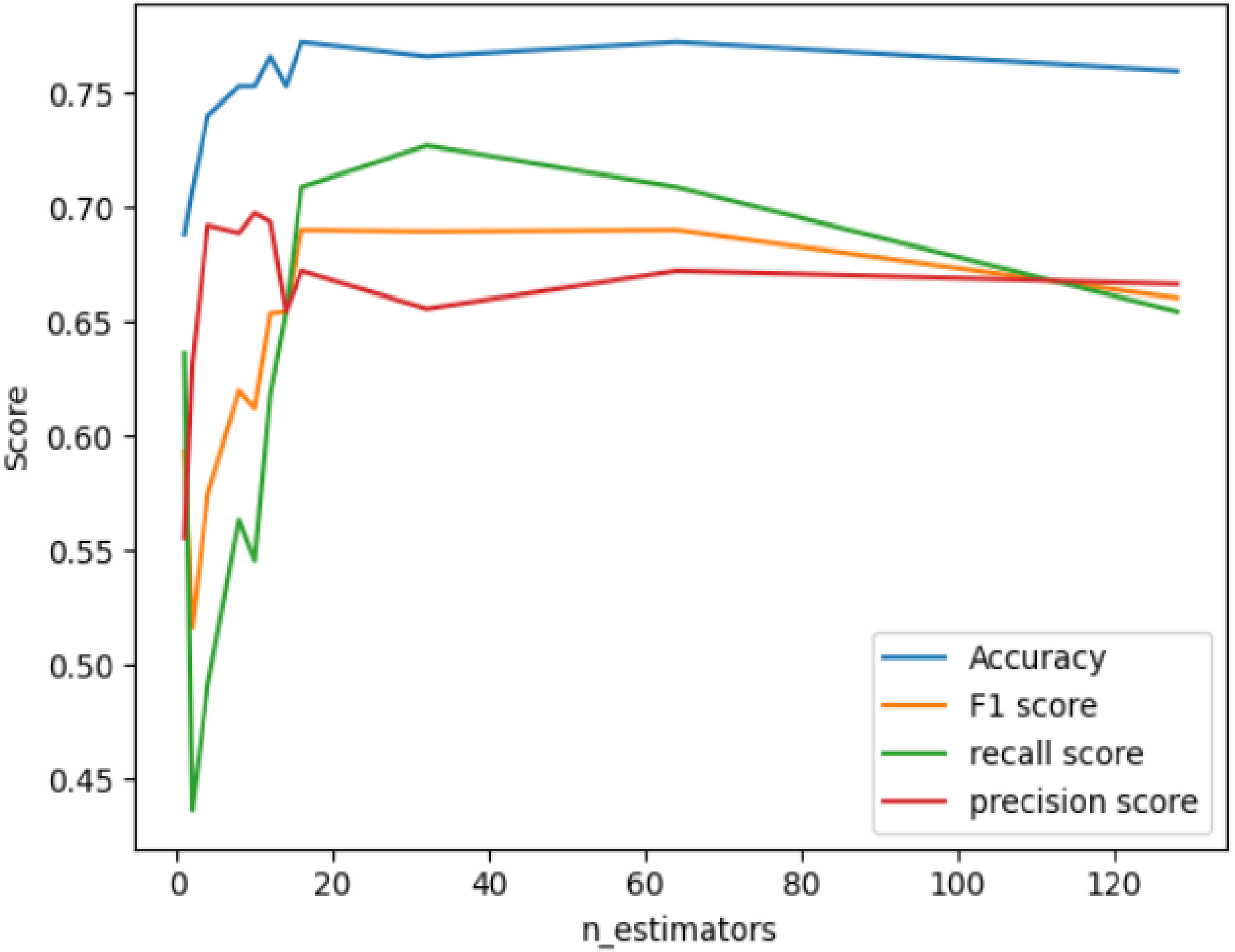
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.656250	72.386719	27.334635	94.652344	32.450911	0.471876	33.240885	0.348958
std	3.369578	30.438286	12.096642	9.229014	105.547598	6.875366	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	23.000000	30.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	31.250000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Histograms of Diabetes Dataset Features.

```
diabetes.hist(figsize = (12,8))
plt.show()
```



Training Random Forest with cleaned data that 0 values replaced by median values

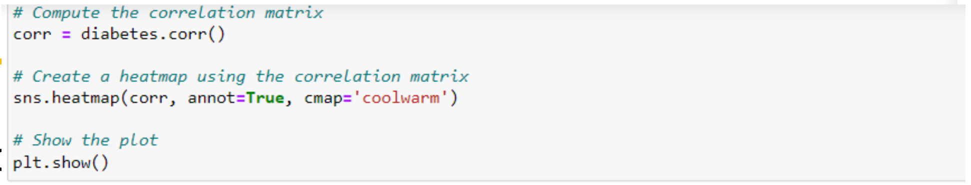


```
n_estimators: 32
Accuracy: 0.7662337662337663
F1 score: 0.689655172413793
Recall score: 0.7272727272727273
Precision score: 0.6557377049180327
Runtime: 0.07053709030151367

n_estimators: 64
Accuracy: 0.7727272727272727
F1 score: 0.6902654867256638
Recall score: 0.7090909090909091
Precision score: 0.6724137931034483
Runtime: 0.12407302856445312
```

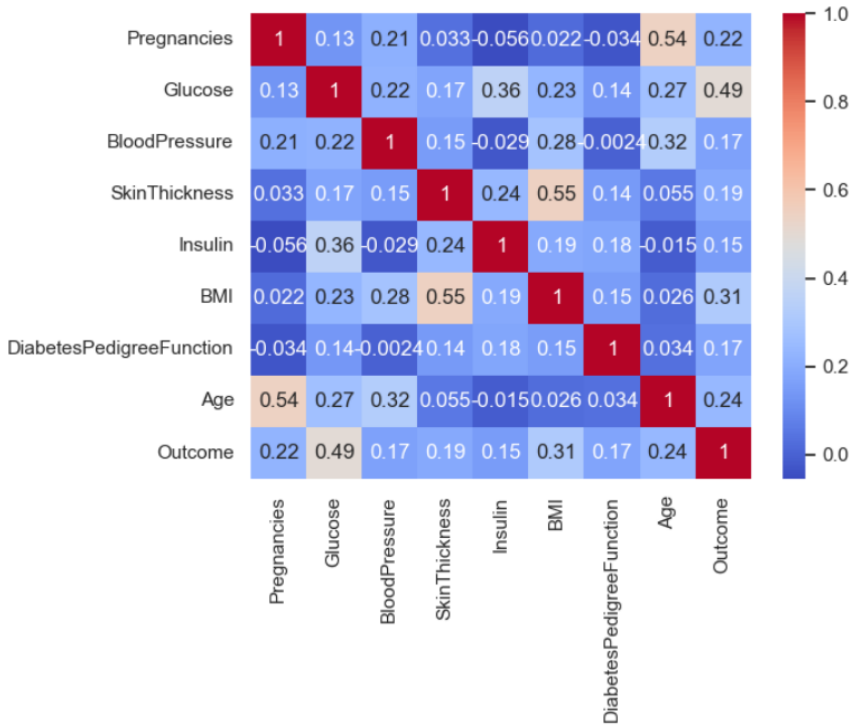
As it is observable we have decent Accuracy, and also F1_score for number of n estimators in range of 16 to 64

Correlation matrix

- A c
  co
  va
  rep
  va
- A c
  pe
  wh
  va
  co
  co
  inc
  co
  co

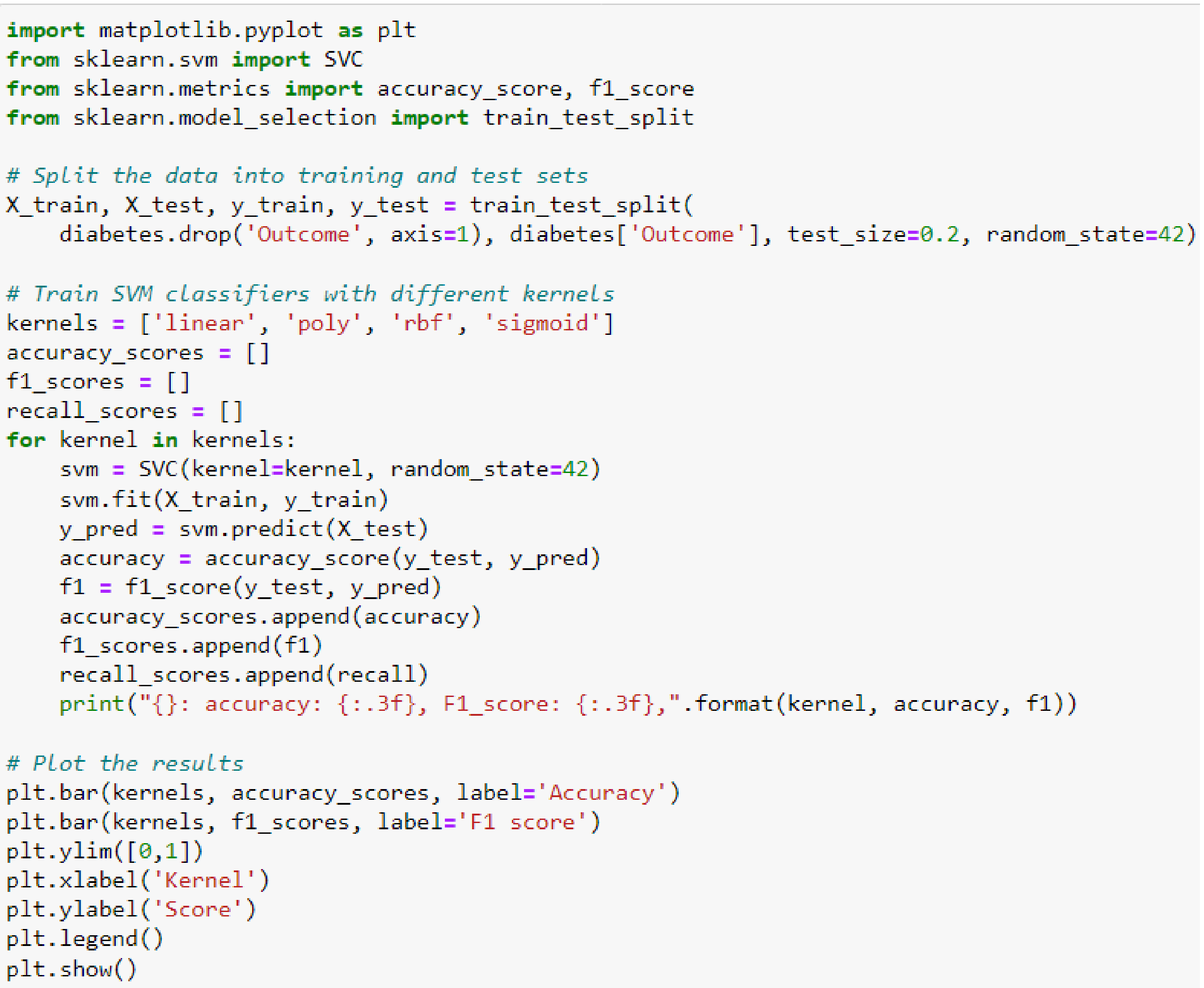


```
# Compute the correlation matrix
corr = diabetes.corr()

# Create a heatmap using the correlation matrix
sns.heatmap(corr, annot=True, cmap='coolwarm')

# Show the plot
plt.show()
```



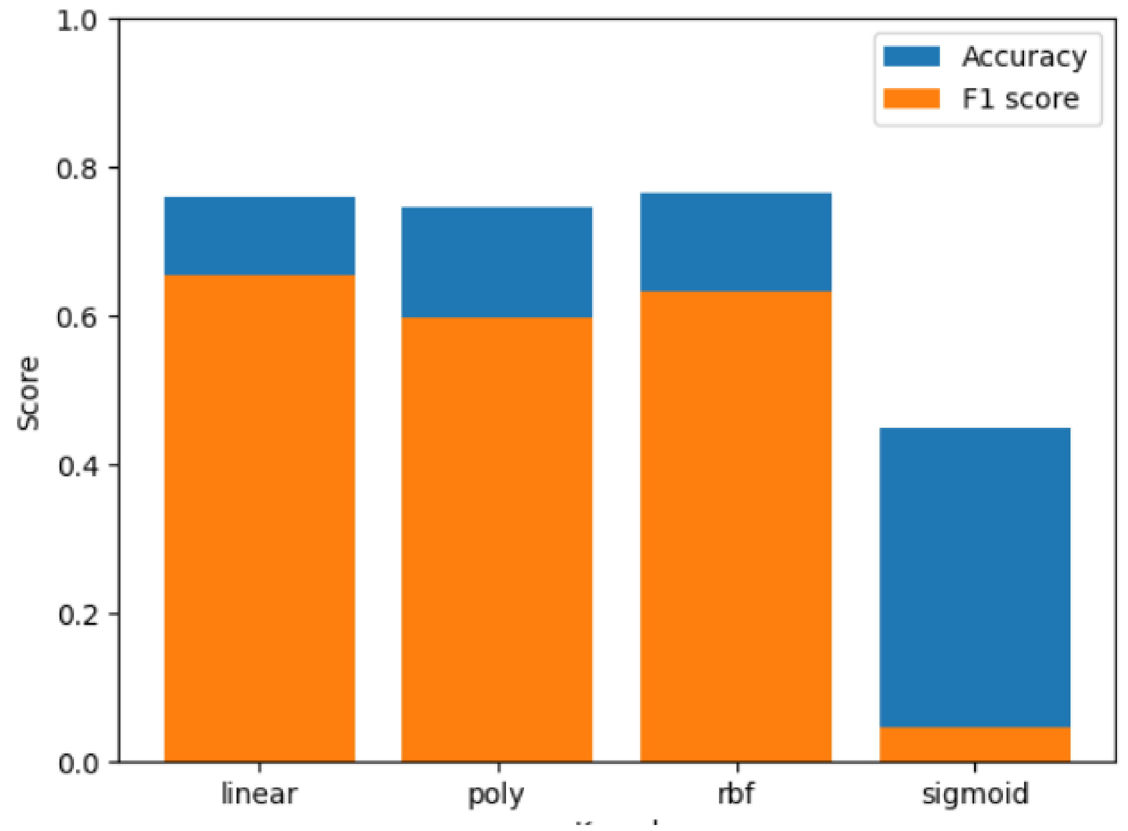
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies	1	0.13	0.21	0.033	-0.056	0.022	-0.034	0.54	0.22
Glucose	0.13	1	0.22	0.17	0.36	0.23	0.14	0.27	0.49
BloodPressure	0.21	0.22	1	0.15	-0.029	0.28	-0.0024	0.32	0.17
SkinThickness	0.033	0.17	0.15	1	0.24	0.55	0.14	0.055	0.19
Insulin	-0.056	0.36	-0.029	0.24	1	0.19	0.18	-0.015	0.15
BMI	0.022	0.23	0.28	0.55	0.19	1	0.15	0.026	0.31
DiabetesPedigreeFunction	-0.034	0.14	-0.0024	0.14	0.18	0.15	1	0.034	0.17
Age	0.54	0.27	0.32	0.055	-0.015	0.026	0.034	1	0.24
Outcome	0.22	0.49	0.17	0.19	0.15	0.31	0.17	0.24	1

Training SVC algorithm with diabetes data



```python
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    diabetes.drop('Outcome', axis=1), diabetes['Outcome'], test_size=0.2, random_state=42)

# Train SVM classifiers with different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
accuracy_scores = []
f1_scores = []
recall_scores = []
for kernel in kernels:
    svm = SVC(kernel=kernel, random_state=42)
    svm.fit(X_train, y_train)
    y_pred = svm.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    recall_scores.append(recall)
    print("{}: accuracy: {:.3f}, F1_score: {:.3f},".format(kernel, accuracy, f1))

# Plot the results
plt.bar(kernels, accuracy_scores, label='Accuracy')
plt.bar(kernels, f1_scores, label='F1 score')
plt.ylim([0,1])
plt.xlabel('Kernel')
plt.ylabel('Score')
plt.legend()
plt.show()
```

```
linear: accuracy: 0.760, F1_score: 0.654,
poly: accuracy: 0.747, F1_score: 0.598,
rbf: accuracy: 0.766, F1_score: 0.633,
sigmoid: accuracy: 0.448, F1_score: 0.045,
```



4 Types of SVC kernels

1

Linear Kernel: The linear kernel is the simplest kernel function, and it assumes that the data is linearly separable. It works well when the data is linearly separable, and it is computationally efficient.

2

Polynomial Kernel: The polynomial kernel function maps the input data into a higher-dimensional feature space by using a polynomial function. This kernel function can capture non-linear relationships between the input features, but it is more computationally expensive than the linear kernel.

3

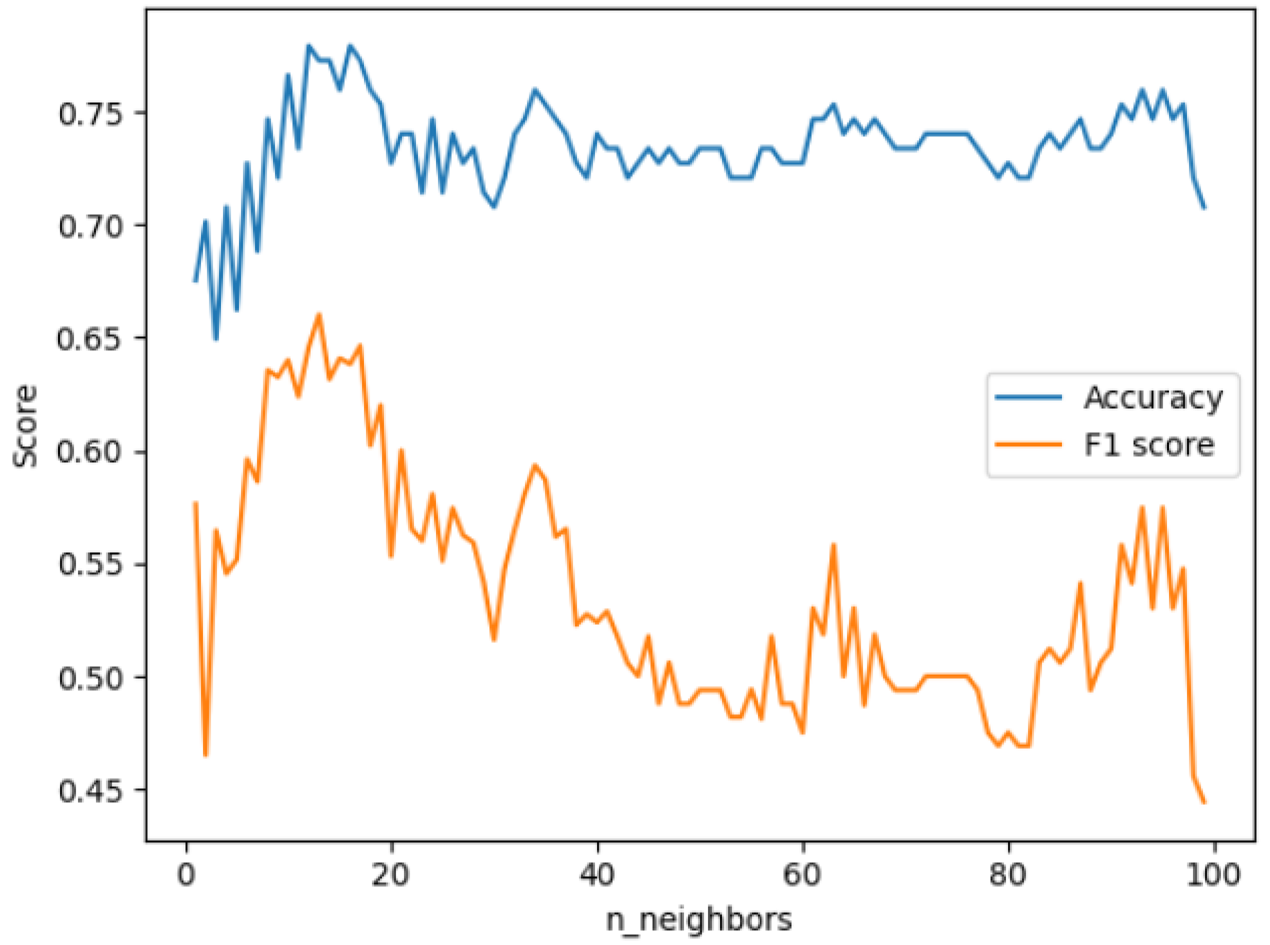
Radial Basis Function (RBF) Kernel: The RBF kernel function is the most commonly used kernel in SVMs. It maps the input data into a higher-dimensional feature space using a Gaussian function. This kernel can capture complex non-linear relationships between the input features, but it is also computationally expensive.

4

Sigmoid Kernel: The sigmoid kernel function maps the input data into a higher-dimensional feature space using a sigmoid function. It can be used for non-linear classification problems, but it is less commonly used than the linear, polynomial, and RBF kernels.

Training KNN classifier with diabetes data

```
# Evaluate the model for each value of n_neighbors
for n_neighbors in n_neighbors_range:
    start_time = time.time()


    # Train a KNN classifier
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = knn.predict(X_test)

    # Calculate accuracy and F1 score
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    # Append the accuracy and F1 scores to the lists
    accuracy_scores.append(accuracy)
    f1_scores.append(f1)

    # Print the results and runtime
    end_time = time.time()
    print('n_neighbors:', n_neighbors)
    print('Accuracy:', accuracy)
    print('F1 score:', f1)
    print('Runtime:', end_time - start_time)
    print('')
```



Why the performance of KNN is so poor?

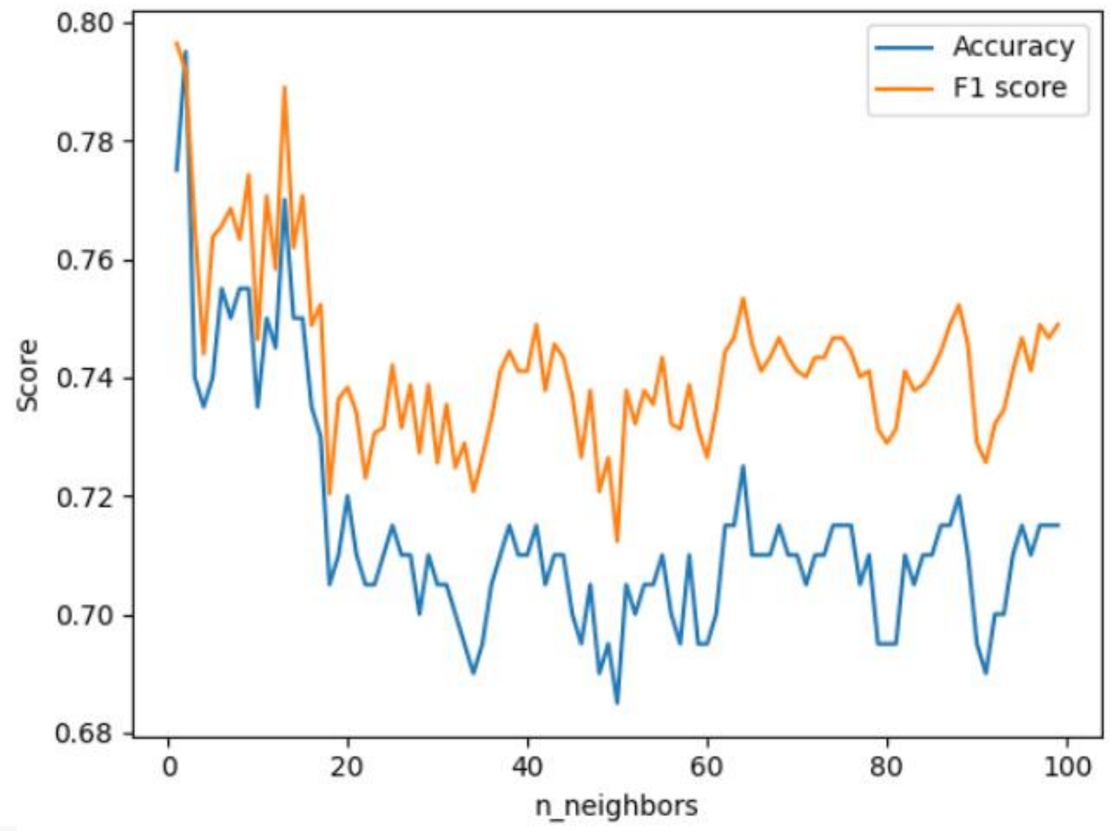
KNN is a simple algorithm that relies on the distance between samples to make predictions. In an imbalanced dataset, the nearest neighbors of a sample are likely to belong to the majority class, and the model may predict that class for the sample. This can result in poor performance, as the minority class is not well represented in the training data.

To address this issue, resampling techniques such as oversampling or under sampling can be used to balance the dataset. Oversampling involves increasing the number of instances of the minority class, while under sampling involves decreasing the number of instances of the majority class. This can help the model learn more about the minority class and improve its performance.

In our case, it is likely that resampling the data helps to balance the classes and improve the performance of the KNN model. By increasing the number of instances of the minority class, the model may have learned more about the features that distinguish that class, and thus may make better predictions.

Training KNN with oversampling on data

```python
from imblearn.over_sampling import SMOTE

# Split the data into features and target
X = diabetes.drop('Outcome', axis=1)
y = diabetes['Outcome']

# Resample the data using SMOTE
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X, y)

# Specify a range of n_neighbors values to evaluate
n_neighbors_range = range(1, 100)

# Initialize lists to store the accuracy and F1 scores
accuracy_scores = []
f1_scores = []

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

# Evaluate the model for each value of n_neighbors
for n_neighbors in n_neighbors_range:
    start_time = time.time()

    # Train a KNN classifier
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = knn.predict(X_test)

    # Calculate accuracy and F1 score
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    # Append the accuracy and F1 scores to the lists
    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
```



Conclusion

- We used 3 different classification algorithm to accurately predict whether or not a case has a diabetes or not.
- First of all, to achieve a satisfying result with Random Forest we replaced all the cases with values that were equal to 0 with the median value of that feature among the total cases, consequently, the efficiency of the model increased.
- Secondly, we trained Svc model with 4 different types of kernel, linear kernel worked well and rbf was not bad.
- Finally, we trained a KNN model with our diabetes data, at first, the outcome was poor, then to improve the result we over sampled our dataset to make it balanced and consequently, the final score improved dramatically.