Maciej Zagórski
Computer Science, 3rd year

PROJECT REPORT
Evaluating the performance of selected classifiers

1. Object of the project

The object of the project was to assess how the following learned classifiers (methods, machine learning (supervised) algorithms) work:

- Naive Bayes classifier,
- k-Nearest Neighbours (k-NN) classifier – with different parameter k:
    - k = 3
    - k = 5
    - k = 11
- decision tree algorithm.

The performance of the above classifiers was evaluated using the sample dataset "diabetes.csv", which contains information about women of Native American origin from the USA (their medical parameters) who did or did not develop diabetes. The use of a classifier in this context serves to diagnose the disease based on known medical data.

2. Project implementation

2.1. Analysis of the dataset

The "diabetes.csv" dataset, as indicated, contains the medical parameters of a group of women – 768 – such as the number of pregnancies (*Pregnancies* column), glucose value (*Glucose*), blood pressure value *(BloodPressure),* skin thickness (*SkinThickness),* insulin value (*Insulin),* BMI, rate of family history of diabetes *(DiabetesPedigreeFunction*) and age (*Age)*.

The collection also included information on whether or not a person had developed diabetes – in the *Outcome* column, using a value of "0" or "1". For the purposes of this project, it was assumed that a value of "1" ("true") in the Outcome column means that a person has developed diabetes (respectively, it was assumed that a value of "0" means "false" – a person has not developed diabetes).

Apart from the above columns' names and values, no information is included in the dataset, e.g. about the units in which the values are expressed or how they were calculated (e.g. in the case of *DiabetesPedigreeFunction*).

The dataset in question does not contain ordinal data (in the form of, for example, a column containing the ordinal numbers of individual records), which could inappropriately influence ("distort") the effects of the individual classifiers. Therefore, it was not necessary to prepare the dataset for further use from this perspective.

However, it should be noted that not all data included in the dataset are correct (complete). For example: the blood pressure value was indicated as “0” in several cases, which could indicate that the persons surveyed, for example, were not alive at the time of the survey; the skin thickness value was “0” in more than 200 cases, which could in turn indicate that the survey had not been carried out.

2.2. Breakdown of the dataset

For the project, the “diabetes.csv” dataset was first imported into the programme, using the *pandas* library for data analysis and manipulation. The dataset thus imported was then divided into two separate subsets:

- subset “x” containing the medical data (columns) of the individuals, but without information on whether the particular individual has or has not developed diabetes (no *Outcome* column),
- subset “y”, containing outcome information on whether or not an individual (with certain medical parameters) has developed diabetes (*Outcome* column only).

```
data_diabetes = pd.read_csv('../data/report_1_classifiers_data_diabetes.csv')

y = data_diabetes['Outcome']
x = data_diabetes.drop('Outcome', axis=1)
```

2.3. Data normalisation

At this stage of the project, data normalisation was also carried out – with the aim of testing whether the results of the different classifiers would differ depending on whether they were operating on normalised data or not. To this end, the data contained in the individual columns of the subset “x” were normalised – within each of those columns – using the following normalising function *(normalise).*

```
def normalize(data, column):
    return (data[column] - data[column].min()) /
           (data[column].max() - data[column].min())
# [...]
x_norm = x.copy()

for column in x_norm:
    x_norm[column] = normalize(x_norm, column)
```

As the information contained in the subset “y” is outcome information, with values of “0” or “1” (person has or has not developed diabetes, respectively), it was not necessary to perform such normalisation on this data.

2.4. Division of data into training and test data

The data were then randomly split into subsets of training (67%) and test (33%) data. For this purpose, the function *train_test_split* from the *sklearn.model_selection* module (from the *scikit-learn* library) was used. This splitting was done separately for the “raw”, unnormalized data, and for the normalised data (variables with the infix “*_norm_*”).

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33,
                                                    random_state=0, stratify=y)
x_norm_train, x_norm_test, y_norm_train, y_norm_test = train_test_split(x_norm, y,
                                                                        test_size=0.33,
                                                                        random_state=0,
                                                                        stratify=y)
```

This was made using the parameter *stratify=y,* which ensured that the subsets returned by the *train_test _split* function retained the same proportions of class labels as the input dataset.

2.5. Application of classifiers

In the next step, the classifiers indicated in sec. 1 above were applied to the subset prepared in the way described above – each separately for the unnormalized data and for the normalised data. As a result of running each of them, its percentage accuracy and error matrix were displayed. This was done using, respectively, the score method for each classifier (from the relevant modules of the *scikit-learn* library) and the *confusion_matrix* function from the *metrics* module of that library (actual values are indicated on the x-axis, while values predicted by the classifier are indicated on the y-axis).

- The Naive Bayes classifier:

```
print("\n* * * Naive Bayes * * *\n")

gnb = GaussianNB()

gnb.fit(x_train, y_train)
y_pred = gnb.predict(x_test)

print("Score: %f" % (gnb.score(x_test, y_test)))
print("Confusion matrix:\n", confusion_matrix(y_test, y_pred), "\n")

print("Normalized:")

gnb.fit(x_norm_train, y_norm_train)
y_norm_pred = gnb.predict(x_norm_test)

print("Score: %f" % ((y_norm_test == y_norm_pred).sum() / x_norm_test.shape[0]))
print("Confusion matrix:\n", confusion_matrix(y_norm_test, y_norm_pred), "\n")
```

```
* * * Naive Bayes * * *

Score: 0.767717
Confusion matrix:
 [[141  24]
 [ 35  54]]

Normalized:
Score: 0.767717
Confusion matrix:
 [[141  24]
 [ 35  54]]
```

- The k-Nearest Neighbours (k-NN) classifier – with parameter k having values of 3, 5 and 11, respectively, and using the Euclidean distance (*p=2*) and the *Minkowski* metric (*metric='minkowski'*):

```
print("* * * k-NN * * *\n")

k_list = [3, 5, 11]
for k in k_list:
    print("k-%d neighbors:\n" % (k))


    knn = KNeighborsClassifier(n_neighbors=k, p=2, metric='minkowski')

    knn.fit(x_train, y_train)
    y_pred = knn.predict(x_test)

    print("Score: %f" % (knn.score(x_test, y_test)))
    print("Confusion matrix:\n", confusion_matrix(y_test, y_pred), "\n")

    print("Normalized:")

    knn.fit(x_norm_train, y_norm_train)
    y_norm_pred = knn.predict(x_norm_test)

    print("Score: %f" % (knn.score(x_norm_test, y_norm_test)))
    print("Confusion matrix:\n", confusion_matrix(y_norm_test, y_norm_pred), "\n")
```

```
* * * k-NN * * *

k-3 neighbors:

Score: 0.692913
Confusion matrix:
 [[129  36]
 [ 42  47]]

Normalized:
Score: 0.755906
Confusion matrix:
 [[138  27]
 [ 35  54]]

k-5 neighbors:

Score: 0.732283
Confusion matrix:
 [[137  28]
 [ 40  49]]

Normalized:
Score: 0.748031
Confusion matrix:
 [[142  23]
 [ 41  48]]

k-11 neighbors:

Score: 0.736220
Confusion matrix:
 [[141  24]
 [ 43  46]]

Normalized:
Score: 0.748031
Confusion matrix:
 [[141  24]
 [ 40  49]]
```

- The decision tree algorithm, using the Gini coefficient as a branching criterion (*criterion='gini'*) and specifying the maximum depth (height) of the tree as 4 (*max_depth=4*):

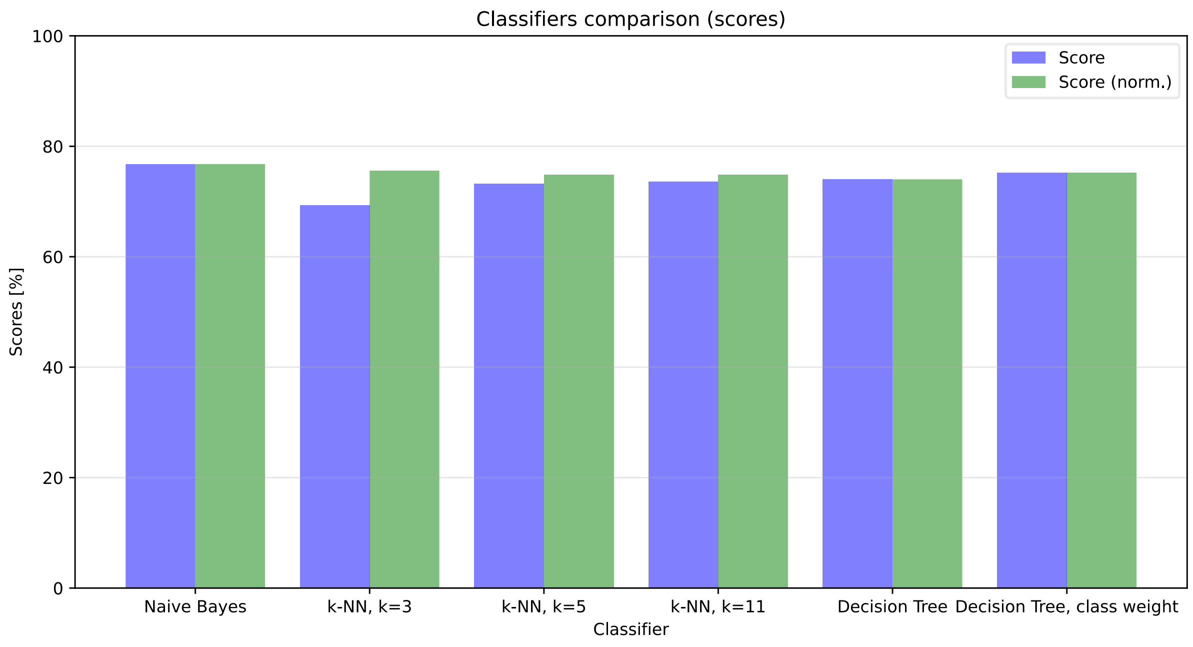
```
print("* * * Decision Tree * * *\n")

tree = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state=1)

tree.fit(x_train, y_train)
y_pred = tree.predict(x_test)

print("Score: %f" % (tree.score(x_test, y_test)))
print("Confusion matrix:\n", confusion_matrix(y_test, y_pred), "\n")

print("Normalized:")

tree.fit(x_norm_train, y_norm_train)
y_norm_pred = tree.predict(x_norm_test)

print("Score: %f" % (tree.score(x_norm_test, y_norm_test)))
print("Confusion matrix:\n", confusion_matrix(y_norm_test, y_norm_pred), "\n")
```

```
* * * Decision Tree * * *

Score: 0.740157
Confusion matrix:
 [[128  37]
 [ 29  60]]

Normalized:
Score: 0.740157
Confusion matrix:
 [[128  37]
 [ 29  60]]
```

3. Presentation of results

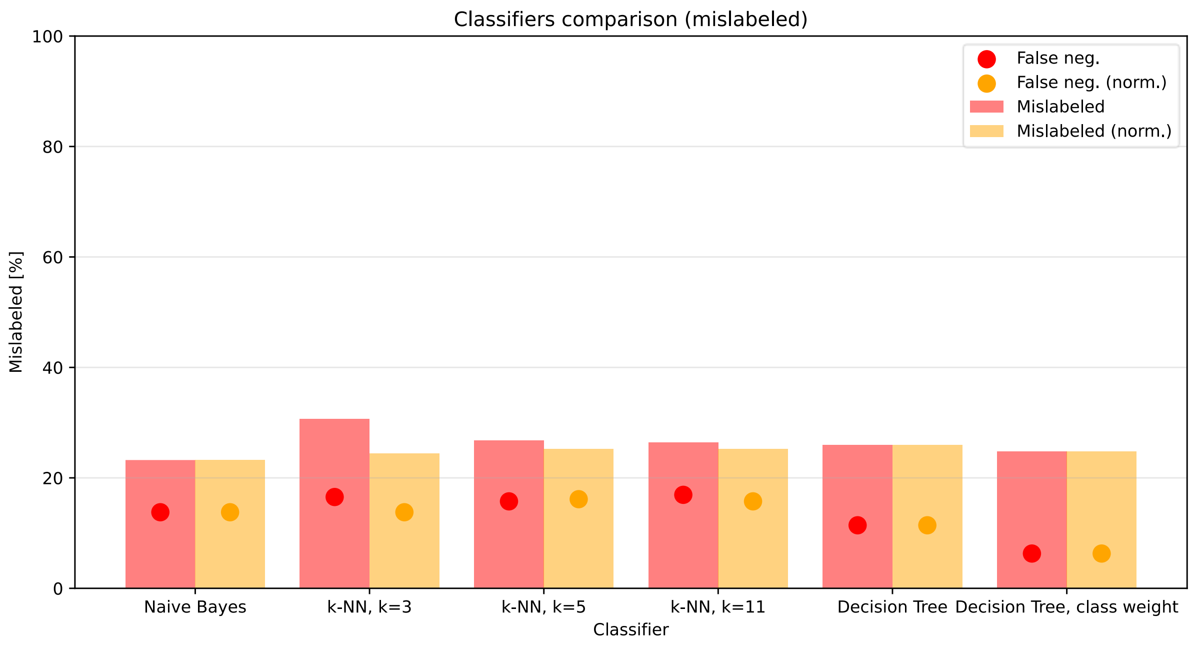
The results obtained above are then presented in the form of a table for information on the percentage accuracy of the classifier and the *false negative* results it returns, as well as in the form of a bar chart showing the percentage accuracy data of the classifiers.

	Classifier	***Score***	***Score (norm.)***	***False neg.***	***False neg. (norm.)***
1.	*Naive Bayes*	0.767717	0.767717	35.0	35.0
2.	*k-NN, k=3*	0.692913	0.755906	42.0	35.0
3.	*k-NN, k=5*	0.732283	0.748031	40.0	41.0
4.	*k-NN, k=11*	0.736220	0.748031	43.0	40.0
5.	*Decision Tree,*	0.740157	0.740157	29.0	29.0
6.	*Dec. Tree, class weight*	0.751969	0.751969	16.0	16.0



Values in the categories labelled *(norm.)*, e.g. *Score (norm.)*, are shown for the results obtained using the normalised data. The table and the graph also show the results of using the decision tree classifier with the *class weight* parameter – comments on this are provided in sec. 4 below.

In addition, a graph comparing the percentage inaccuracies of the classifiers (“inversion” of the first graph) is shown below, with the *false negative* results returned by each classifier highlighted (expressed as a percentage of the total results of a given classifier).



4. Comments and conclusions

Based on the results obtained, it can be indicated that:

- The Naive Bayes classifier showed the highest accuracy (reaching nearly 77%), while the k-NN classifier with a parameter k equal to 3, operating on unnormalized data, showed the lowest accuracy (approximately 69%).
- Removal of records (cases) for which complete medical data have not been collected (e.g. the lack of blood pressure information mentioned in sec. **Error! Reference source not found.**) from the dataset resulted in some classifiers obtaining better results – depending on which data were deleted (which deletion criterion was adopted; an example of deletion of records for which the blood pressure value was defined as 0 is shown below).

```
data_diabetes.drop(data_diabetes[(data_diabetes.BloodPressure == 0)].index,
                   inplace=True)
```

  With, for example, the removal of records where skin thickness was specified as 0, the accuracy of the Naive Bayes classifier increased to over 79%; while the removal of records where the blood pressure value was 0 resulted in the Decision Tree classifier achieving an accuracy of over 77% when specifying the maximum tree height as 2 (*max_depth=2*). This may mean that proper data preparation, combined with expert knowledge and the assumption of the choice of a classifier (its parameters), **influences the accuracy (effectiveness) of the solution used.**

- The results of applying the k-NN classifier with different values of the k parameter using the unnormalized data **were worse than its applications using the normalised data** (in the extreme case of k = 3, there was a difference in accuracy between about 69% and about 76%). The above resulted from the fact that the k-NN classifier implemented in the project used the Euclidean metric (the Minkowski metric being a generalisation of the Euclidean metric). The use of this metric may require normalisation of the data so that each feature is scaled appropriately (to the distances based on which the classifier makes its selection).
- As indicated in sec. **Error! Reference source not found.**, in the situation that is the subject of the project, the use of the classifier serves to diagnose a disease on the basis of known medical data. In such a practical context, it would be a much "worse" *false* result if the algorithm considers a certain individual to be healthy ("0" value in the *Outcome* column), while he or she is in fact ill ("1" *Outcome*) – a *false negative* case. From a socio-health perspective, it would be better to (erroneously) conclude that a healthy person is sick – a *false positive case (*which might be verified later by further testing) – than the other way around ("sending home" such a person, without testing and medication).
- For the Naive Bayes and k-NN classifiers, the number of *false negatives* was higher than the number of *false positives*. A different result was obtained with the decision tree classifier, where the number of *false negatives* (29) was lower than the number of *false positives* (37). This classifier **also allows the use of the class *weight* parameter – assigning appropriate "weights" to the classes into which the results are classified.** Since we consider information about the disease (potentially being ill) to be more important than information about being healthy, class "1" in the *Outcome* column should be assigned with a correspondingly higher weight. The use of a said classifier using the class *weight*

parameter, assigning a value of 1 to class "1" and 0.25 to class "0" (*class_weight={0:0.25, 1:1}*), was presented below.

```
print("Using class weight:\n")

tree = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state=1,
                              class_weight={0:0.25, 1:1})

tree.fit(x_train, y_train)
y_pred = tree.predict(x_test)

print("Score: %f" % (tree.score(x_test, y_test)))
print("Confusion matrix:\n", confusion_matrix(y_test, y_pred), "\n")

print("Normalized:")

tree.fit(x_norm_train, y_norm_train)
y_norm_pred = tree.predict(x_norm_test)

print("Score: %f" % (tree.score(x_norm_test, y_norm_test)))
print("Confusion matrix:\n", confusion_matrix(y_norm_test, y_norm_pred), "\n")
```

```
Using class weight:

Score: 0.751969
Confusion matrix:
 [[118  47]
 [ 16  73]]

Normalized:
Score: 0.751969
Confusion matrix:
 [[118  47]
 [ 16  73]]
```

It can be seen from the above that the use of the classifier with the *class weight* parameter yielded slightly worse accuracy than the (most accurate) Naive Bayes classifier (approx. 75% vs. approx. 77%) **while, at the same time, with the number of *false negative* cases more than twice as low** (16 vs. 35).

- The decision tree classifier also allows the maximum "depth" ("eight") parameter of the decision tree to be set. The value of 4 proposed in the project (*max_depth=4*) was determined based on a trial-and-error basis – the classifier returned worse results with lower or higher values (had poorer accuracy). The lower accuracy with a higher value of this parameter may be explained by the fact that the larger the tree, the more complex the decisions at its subsequent levels become, which can in turn lead to overtraining of the model.

5. Sources

[n.a.], *Kaggle: Your Home for Data Science*, https://www.kaggle.com/, [accessed: 28.04.2023]

[n.a.], *Matplotlib – Visualization with Python*, https://matplotlib.org/, [accessed: 28.04.2023]

[n.a.], *pandas – Python Data Analysis Library*, https://pandas.pydata.org/, [accessed: 28.04.2023]

[n.a.], *scikit-learn: machine learning in Python – scikit-learn 1.2.2 documentation*, https://scikit-learn.org/stable/, [accessed: 28.04.2023]

S. Raschka, V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2,* 3rd ed., Gliwice 2021.