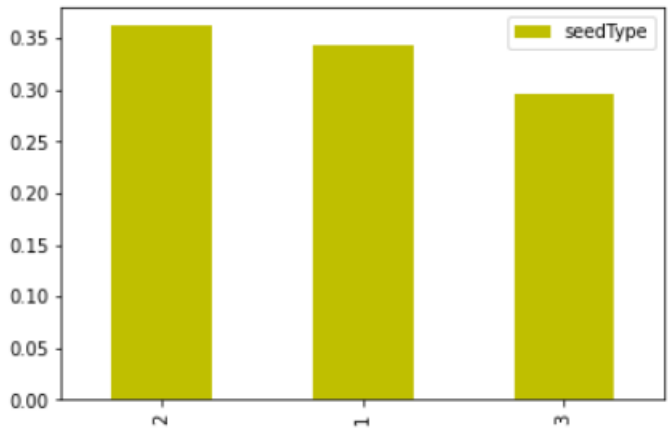
Seed Dataset Classification and Clustering

The examined group comprised kernels belonging to three different varieties of wheat: Kama, Rosa and Canadian, 70 elements each, randomly selected for the experiment. High quality visualization of the internal kernel structure was detected using a soft X-ray technique. It is non-destructive and considerably cheaper than other more sophisticated imaging techniques like scanning microscopy or laser technology. The images were recorded on 13x18 cm X-ray KODAK plates. Studies were conducted using combine harvested wheat grain originating from experimental fields, explored at the Institute of Agrophysics of the Polish Academy of Sciences in Lublin.

The data set can be used for the tasks of classification and cluster analysis.

First of all we need to import the libraries we'll need on this project. Next, we'll need to find the amount of seeds that correspond to each type of fruit out of 3. The 3 types of wheat varieties are Karma, Rosa and Canadian, 70 items each, randomly selected for the experiment. Once this is done, there are two equally important steps we'll take that will help us a lot in understanding our data. The info() and describe() functions are used. The info function gives as a result the type of data that exists in each of the data columns. Describe function is used to display some basic statistical details, such as percentage, average, std, etc., of a data frame or array of numeric values.

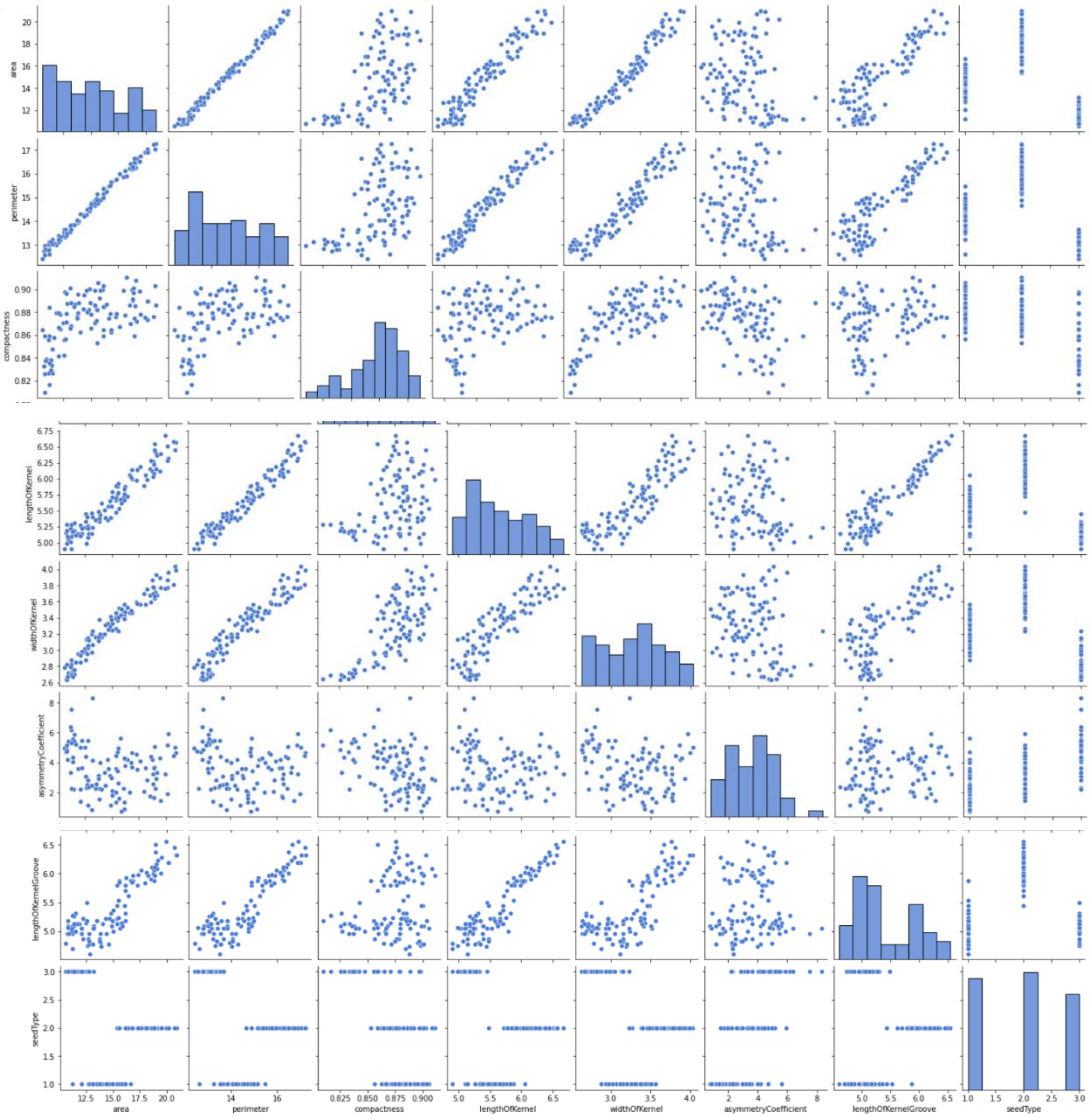
Another step for further understanding and analysis of our data is representation. We use df ['seedType'] to access the seedType field and call value_counts to get a set of unique values. We normalize these values. Data normalization is used to make model training less sensitive to the scale of features. This allows our model to converge better and leads to a more accurate model. Then we call the plot method and transfer to the bar (since we want a bar type graph), the argument. The resulting graph is as follows:



The next step will be to calculate the correlation coefficient. One way to find the correlation between two variables is to create a scatter plot where we represent pairs

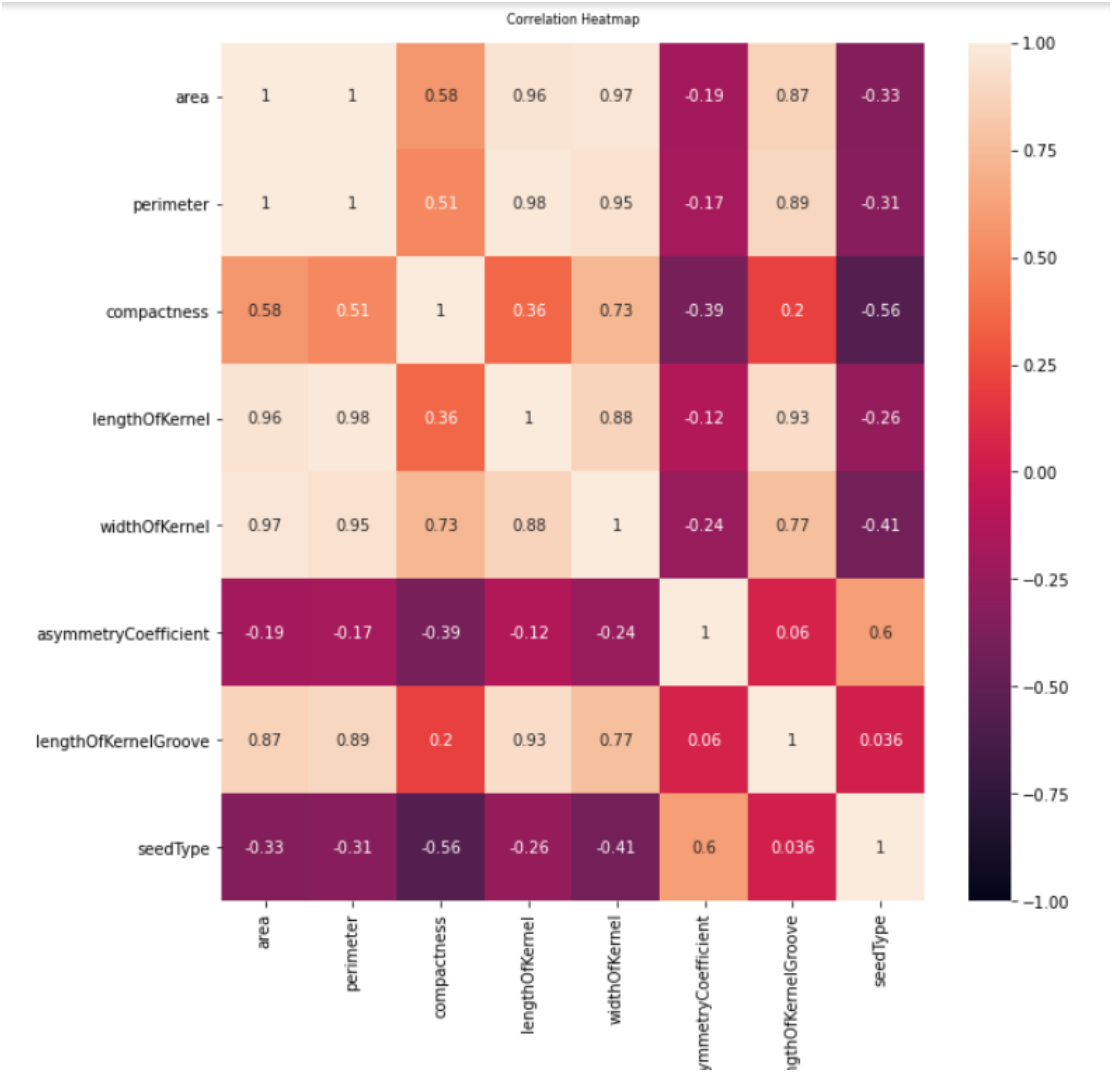
of observations. If we perform the specific procedure for all combinations of columns (variables) we can graphically represent their correlation. In this way we have the ability to graphically observe the correlation between two variables.

We will create diagrams for this purpose with the help of the seaborn library. Seaborn is a Python data visualization library based on matplotlib. More specifically we will use PairGrid. This object maps each variable to a data set in a column and a row in a multi-axis grid. But since we want to represent a lot of data, we will use the pairplot () method which serves the representation of many graphs in one line. The resulting graph is as follows.



It is also possible to use different functions in the upper and lower triangles of the above plot (which otherwise would be unnecessary). The corresponding diagram with these functions is given below.

Another way to find the correlation between the columns of the data box is to calculate the linear correlation coefficient (Pearson coefficient). This coefficient can estimate the degree of linear correlation between the variables and ranges from -1 (full linear negative correlation) to +1 (full linear positive correlation), with the value zero corresponding to unrelated variables (linear). The Pandas corr () function is used to calculate the correlation between the different columns (variables) for the different parameters. Below is the heatmap that clearly shows the correlation.



The correlation ranges from -1 to +1. Values close to zero mean that there is no linear trend between the two variables. The closer the correlation is to 1, the more positive it is. As one grows, so does the other, and the closer it is to 1, the stronger the relationship. A correlation closer to -1 is similar, but instead of both increasing, one will increase while the other will decrease. The diagonals are all 1. These squares correlate each variable with itself. For the rest, the larger the number and the lighter the color, the higher the correlation between the two variables. The diagram is also symmetrical about the main diameter diagonal, as the same two variables are together in the same box.

Next, the process is followed by the pre-processing of the data. For data preparation one of the most useful transformations is data scaling. This process converts the data in such a way that the values of the different columns (variables) change within a certain period of time. This reduces the effect of the difference in the numerical scales of the data (eg one variable can vary between 0-1 and another between -1000 and 10000000) on the machine learning algorithms. We import the StandardScaler () function from scikit-learn. The data standardization method will be used. Centering and scaling are performed independently in each operation by calculating the relevant

statistical data for the samples in the training set. The mean and standard deviation are then stored for use in later data using transform.

Then we continue to create training and control sets. The purpose is to train the algorithm to respond correctly to future data (efficiency in making correct predictions). We will use the train_test_split () function of scikit-learn. Essentially, it splits the tables into random train and test subsets. The training set is a subset to 'train' the model. The test set is a subset to test the trained model.

Next, we use 3 algorithms for categorization. The algorithms used are as follows:

• Random forest

• Gaussian Naive Bayes

• k-nearest neighbors

Firstly, we'll analyze the first algorithm, that of the random forest. A random forest is an estimator that fits a number of tree decision classifiers into different subsets of the data set and uses the average to improve forecast accuracy and over-alignment control. The random forest, as its name suggests, consists of a large number of individual decision trees that function as a whole. Each individual tree in the random forest unfolds a class prediction and the class with the most votes becomes our model prediction. This algorithm works very well as a large number of relatively unrelated models (trees) that act as a panel will outperform any of the individual component models. some trees may be wrong, many other trees will be right, so as a group the trees can move in the right direction. Each tree in a random forest can only be selected from a random subset of features. This imposes even greater variation between trees in the model and ultimately leads to lower correlation between trees and more differentiation. The following happens in the code. First, we import the StandardScaler () function from the sklearn.model_selection library. The StandardScaler function will transform the data so that the distribution has an average value of 0 and a standard deviation of 1. Then we adjust the data so that we can then convert it. We train the data and test them and then continue with the random forest.

We import the appropriate libraries and use the RandomForestClassifier function to define the evaluators. We make the forest with the trees from the set with the trained variables and then with the functions predict () and predict_proba () we calculate the order and the probability classes of the trained x. The same procedure is repeated for test_X. We print for both the confusion table as well as the accuracy score. The confusion table evaluates the output quality of a classifier. The diagonal elements represent the number of points for which the predicted label is equal to the actual label, while the other elements outside the main diagonal are those that have not been correctly marked by the classifier. The higher the diagonal values of the confusion table, the more accurate predictions we have.

On the other hand the accuracy calculation is calculated as follows. Given two lists, y_pred and y_true, for each position index i, compare the i element of y_pred with the i element of y_true and first count the numbers of the common elements and divide it

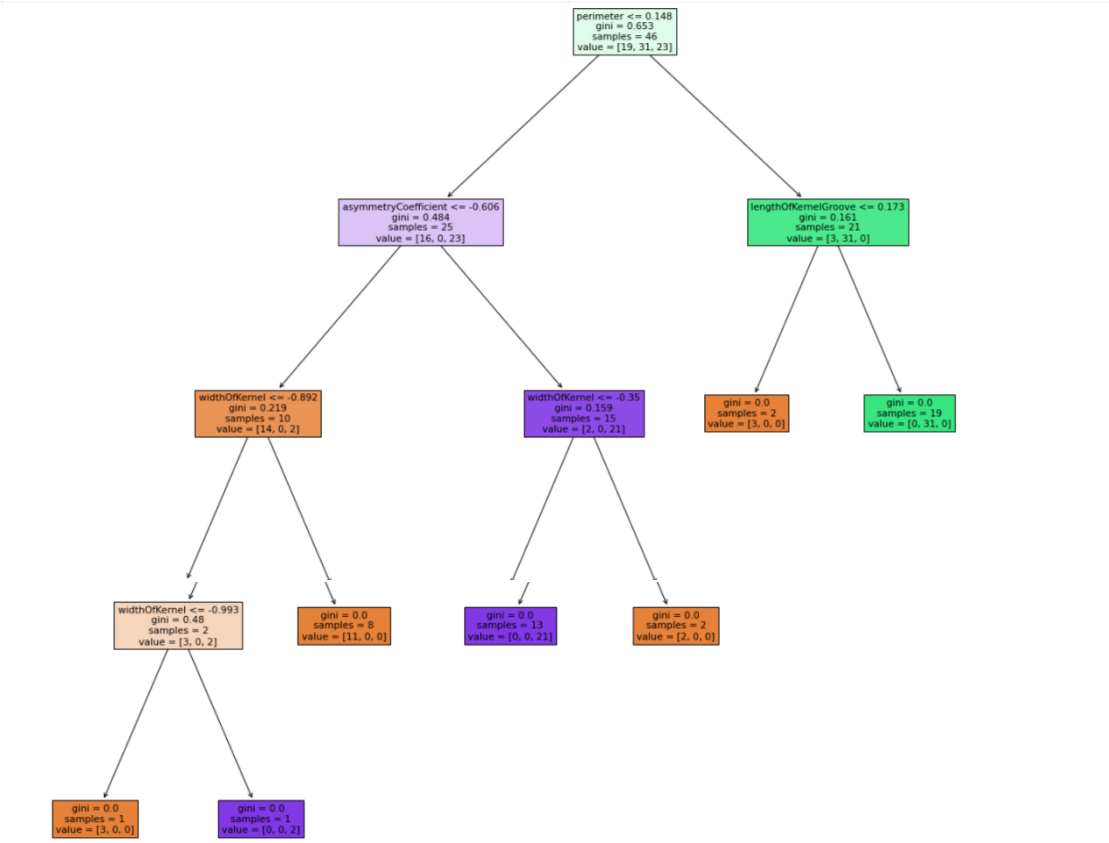
by the number of samples. For the data set we have the result that appears is the following.

```
Confusion Matrix - Train:
 [[22  0  0]
 [ 0 30  0]
 [ 0  0 21]]


Overall Accuracy - Train:  1.0


Confusion Matrix - Test:
 [[11  1  2]
 [ 0  8  0]
 [ 2  0  8]]


Overall Accuracy - Test:  0.84375
```

As shown in the picture based on the confusion table for both train and test data the diagonal consists of quite large numbers, which indicates that we have many correct predictions.

Based on the validity score for the train data we have a validity of 1.0 (ie 100%), which means that we have valid results. For the test data we have a validity score of 0.84 (ie 84%), which is an equally high validity rate.

Then we print the Decision Tree, which is listed below:



Next is the Gaussian Naive Bayes algorithm for categorization. Bayesian naive classifiers are a family of simple "probabilistic classifiers" based on the application of Bayes' theorem with strong (naive) assumptions of independence between attributes. In general, Naive Bayes methods are a set of supervised learning algorithms based on the application of Bayes' theorem with the "naive" assumption of conditional independence between each pair of attributes given the value of the variable class. The naive Bayes classifiers work quite well in many real-world situations, such as document sorting and spam filtering. They require a small amount of training data to assess the necessary parameters. Naive Bayes learners and classifiers are quite fast compared to other methods. Each distribution can be independently estimated as a one-dimensional distribution. This helps to solve the problems that arise due to the spaces. Nevertheless, the naive Bayes may be good classifiers but as appraisers they are not as good. In the code we start once again with the training and testing of the data after we have first introduced from the sklearn.naive_bayes library the GaussianNB equation (which is used for classification in Naive Bayes. Then we find with the functions predict () and predict_proba ( ) the forecasts for the training and testing sets and as shown above.The following is a picture of the results.

```
Confusion Matrix - Train:
 [[21  0  1]
 [ 1 29  0]
 [ 2  0 19]]


Overall Accuracy - Train:  0.9452054794520548


Confusion Matrix - Test:
 [[12  1  1]
 [ 0  8  0]
 [ 0  0 10]]


Overall Accuracy - Test:  0.9375
```

As can be seen from the confusion table and the validity score, we have a lot of good predictions. However, this particular algorithm is not as reliable as it has done with predictions. Of course, based on the previous algorithm, we do have some good predictions. The reason for using this algorithm is essentially to check whether it is unreliable by comparing the results of the two algorithms.

At last, we have the nearest neighbor algorithm. The nearest neighbor categorization is part of a more general technique, known as snapshot training, which does not construct a general model, but uses specific instructional samples to make predictions for a control snapshot. Such algorithms require a proximity measure to determine the similarity or distance between the snapshots and a categorization function, which returns the predicted category of a control snapshot based on its proximity to other snapshots.
Nearest neighbor categorizers can produce erroneous predictions unless the appropriate proximity measure is used, and data preprocessing steps are taken.
In the code, we first import the appropriate libraries and then find the nearest

neighbors of a point. We specify the parameters and create a variable to find the nearest neighbors and then print them. With the fit () function we adapt the categorizer of the nearest neighbors to the elements we have trained. After that we use once again the functions predict () and predict_proba () to calculate the order and the probability classes of the trained x. The same procedure is repeated for test_X. We print for both the confusion table as well as the accuracy score.

```
Confusion Matrix - Train:
 [[21  1  0]
 [ 1 29  0]
 [ 4  0 17]]


Overall Accuracy - Train:  0.9178082191780822


Confusion Matrix - Test:
 [[13  1  0]
 [ 0  8  0]
 [ 1  0  9]]


Overall Accuracy - Test:  0.9375
```

As the proper pre-processing and training of the data has been done the algorithm is quite reliable. As shown in the picture based on the confusion table for both train and test data the diagonal consists of quite large numbers, which indicates that we have many correct predictions.

Based on the validity score for the train data we have a validity of 0.91 (ie 91%), which means that we have valid results. For the test data we have a validity score of 0.93 (ie 93%), which is an equally high validity rate.

After that, we print the main categorization metrics for this algorithm using the classification_report () function.

```
Classification Report-Test:
              precision    recall  f1-score   support

           1       0.93      0.93      0.93        14
           2       0.89      1.00      0.94         8
           3       1.00      0.90      0.95        10

    accuracy                           0.94        32
   macro avg       0.94      0.94      0.94        32
weighted avg       0.94      0.94      0.94        32
```

Precision is the ability of the categorizer not to label as positive a sample that is negative.

Recall is the ability of the categorizer to find all the positive samples.

F1-score is an average of accuracy and recall (precision & recall). The best value reaches 1 and the worst at 0.

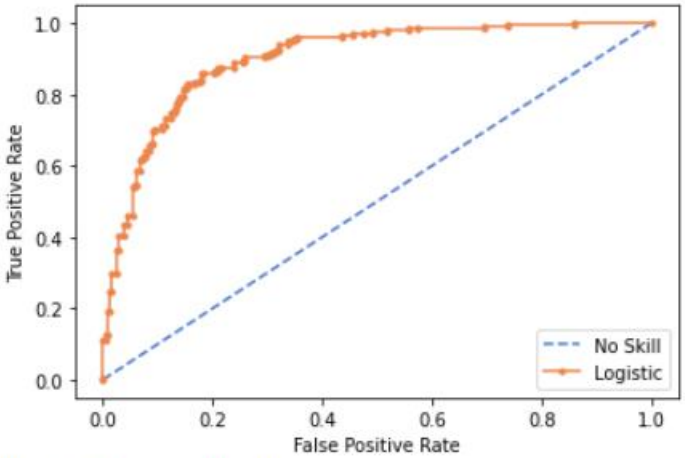
Support is the number of occurrences of each y_true class.

Accuracy calculates the accuracy of the subset: the set of labels provided for a sample must exactly match the corresponding set of labels in y_true.

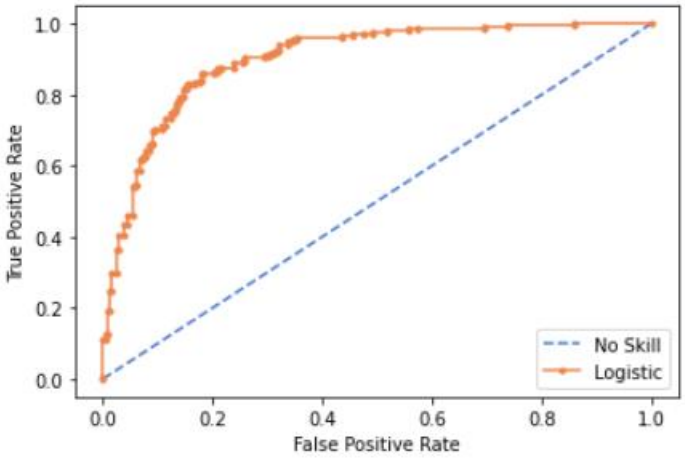
Macro avg calculates the measurements for each tag and finds the unweighted average.

Weighted avg calculates the metrics for each tag and finds their average weight based on support.

Then I print the roc graph which is as follows:




```
No Skill: ROC AUC=0.500
Logistic: ROC AUC=0.903
```



ROC curves usually have a true positive rate on the Y axis and a false positive rate on the X axis. This means that the upper left corner of the graph is the "ideal" point - a false positive percentage zero, and a real positive percentage one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better. The "deviation" of the ROC curves is also important, as it is ideal to maximize the true positive percentage while minimizing the false positive percentage.