Report Project 1: Tool wear classification
TB 3: Image and Pattern recognition UP3

Najlaa SRIFI

8 January 2023

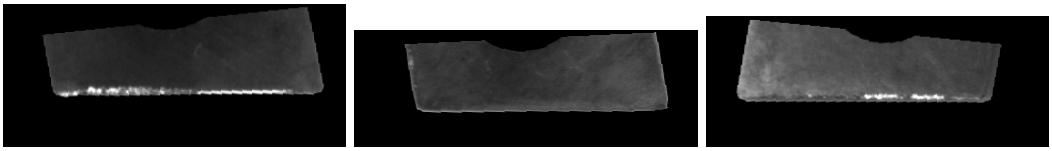
Table des matières

1 Introduction

In this tutorial, we will work on the classification method of logistic regression. We will :
1. Describe a set of images using a set of region descriptors.
2. Using the computed descriptors, we will simulate a classification experiment and compute some performance measures from the results of the logistic regression classifier.
In this experiment, we will divide the dataset into two disjoint sets : learning and testing. There are two separate Python scripts in this TP :
$-extract\_features.py$, in which the descriptors are extracted from the images and saved on the hard disk.
$-classify\_inserts.py$, in which the classification experiment is performed with the aforementioned dataset.

2 Description of the images

The first step in this TP is to describe the images. The images are binary regions corresponding to cutting edges of insert tools used in milling processes.



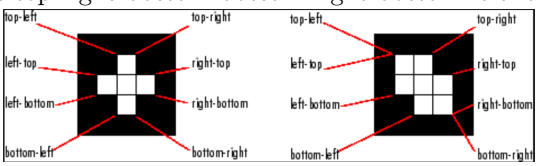
To do this, we will code the get_shape_features function using the regionprops function to extract the following features :

- Convex area
- Eccentricity
- Perimeter
- Equivalent Diameter
- Extent
- Filled Area
- Minor Axis Length
- Major Axis Length
- Ratio between the major and minor axis
- Solidity

```
# Convex Area: Number of pixels of convex hull image, which is the
# smallest convex polygon that encloses the region
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].area)
# Eccentricity: Eccentricity of the ellipse that has the same second-
# moments as the region
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].eccentricity)
# Perimeter: Perimeter of object which approximates the contour as a
# line through the centers of border pixels using a 4-connectivity
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].perimeter)
# Equivalent Diameter: The diameter of a circle with the same area as
# the region
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].equivalent_diameter)
# Extent: Ratio of pixels in the region to pixels in the total
# bounding box
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].extent)
# Filled Area: Number of pixels of the region will all the holes
# filled in
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].filled_area)
# Minor Axis Length: The length of the minor axis of the ellipse that
# has the same normalized second central moments as the region.
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].minor_axis_length)
# Major Axis Length: The length of the major axis of the ellipse that
# has the same normalized second central moments as the region.
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].major_axis_length)
# R: Ratio between the major and minor axis of the ellipse that has
# the same second central moments as the region.
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].major_axis_length/props[0].minor_axis_length)
# Solidity: Ratio of pixels in the region to pixels of the convex
# hull image.
# ==================== YOUR CODE HERE ====================
shape_features.append(props[0].solidity)
return (shape_features)
```

The function will return a vector with all the descriptors of the ShapeFeat.
Other geometric features can be extracted to better describe the wear regions such as :

- **BoundingBox** :Position and size of the smallest box containing the region, returned as a 1-by-(2*Q) vector, where Q is the image dimensionality. The first Q elements are the coordinates of the minimum corner of the box. The second Q elements are the size of the box along each dimension
- **Centroid** :Center of mass of the region, returned as a 1-by-Q vector, where Q is the image dimensionality. The first element of Centroid is the horizontal coordinate (or x-coordinate) of the center of mass. The second element is the vertical coordinate (or y-coordinate)
- **Extrema** :Extrema points in the region, returned as an 8-by-2 matrix. Each row of the matrix contains the x- and y-coordinates of one of the points. The format of the vector is [top-left top-right right-top right-bottom bottom-right bottom-left left-bottom left-top]



- **MaxFeretProperties** :Feret properties that include maximum Feret diameter, its relative angle, and coordinate values, returned as a structure with field.

Field	Description
MaxFeretDiameter	Maximum Feret diameter measured as the maximum distance between any two boundary points on the antipodal vertices of convex hull that enclose the object.
MaxFeretAngle	Angle of the maximum Feret diameter with respect to horizontal axis of the image.
MaxFeretCoordinates	Endpoint coordinates of the maximum Feret diameter.

— **MinFeretProperties** :Feret properties that include minimum Feret diameter, its relative angle, and coordinate values, returned as a structure with fields.

Field	Description
MinFeretDiameter	Minimum Feret diameter measured as the minimum distance between any two boundary points on the antipodal vertices of convex hull that enclose the object
MinFeretAngle	Angle of the minimum Feret diameter with respect to horizontal axis of the image.
MinFeretCoordinates	Endpoint coordinates of the minimum Feret diameter.

— **EulerNumber** :Number of objects in the region minus the number of holes in those objects, returned as a scalar. This property is supported only for 2-D label matrices. regionprops uses 8-connectivity to compute the Euler number (also known as the Euler characteristic)

3 Classification

3.1 Division of the data between Train and Test sets

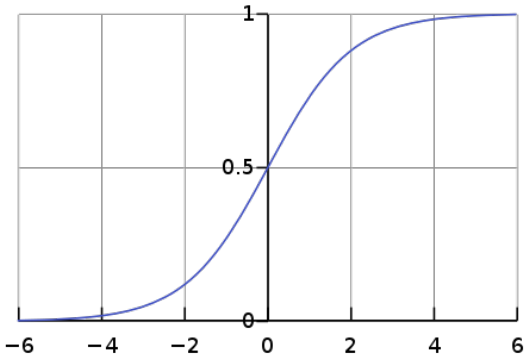
The construction and training of our classification algorithm requires the division of the data set into training and test data. Indeed, the logistic regression model will be initially fitted on a training data set (X_train), which is a set of examples used to fit the model parameters. These adjusted parameters will be used later with the algorithm to classify the elements of the test database (X_test).**We shall use 70% for the training.**

3.2 Training of the classifier and classification of the test set

3.2.1 The sigmoid function

In a linear regression model, the hypothesis function is a linear combination of parameters given as y = ax+b for a simple single parameter data. This allows us to predict continuous values effectively, but in logistic regression, the response variables are binomial, either 'TRUE' or 'FALSE'. So,

it makes less sense to use the linear function to predict anything except the values between 0 and 1. And the most effective function to limit the results of a linear equation to [0,1] is the sigmoid or logistic function.



As you can see, the sigmoid function intersects the y-axis at 0.5. In most cases, we use this point as a threshold for classification. Any value above it will be classified as 1, while any value below is 0. This is not a rule of thumb. We can also use different values instead of 0.5, depending on the requirements.
We code the sigmoid function which takes as input two parameters (vector x and theta) and returns the following output :

$$h_\theta(X) = g(\theta^T X) = \frac{1}{1 + e^{-\theta^T X}} \tag{1}$$

The implementation is done as follows :

```
X_T=np.transpose(X)
g=1/ (1+np.exp(-theta*X_T))
```

3.2.2 Cost function

After the implementation of the sigmoid function we set up this time a cost function that calculates the average error between the outputs of the sigmoid function for the training elements and their real classes.
the cost function is given by the following equation :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^i ln(h_\theta(x^i)) + (1 - y^i) ln(1 - h_\theta(x^i)) \tag{2}$$

we code this function in the following way :

```
cost_i=y*log(y_hat)+(1-y)*log(1-y_hat)
```

This optimisation is carried out by means of the gradient descent algorithm, which is an iterative procedure in which each of the components of the vector are updated on each iteration :

$$\theta_j^k = \theta_j^{k-1} - \alpha \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i \tag{3}$$

where the element x_j^i is the j-th feature of the feature vector that represents the i-th object. Be aware that the value of the element x_0 is always 1. In practice, it will be obtained adding the value 1 at the beginning of the feature vector

3.2.3 Training

After setting up the essential elements for the gradient descent. Now it is time to implement our training code in the train_logistic_regression function which takes as input parameters :

- X_train : Matrix with dimensions (mxn) with the training data, where m is the number of training patterns and n is the number of features.
- Y_train : Vector that contains the classes of the training patterns. Its length is m.
- Alpha : learning rate for the gradient descent algorithm.

On each iteration of the gradient descent algorithm three steps are carried out :

- the value of the output of the function h is calculated for each feature vector of the training set
- the components of θ are updated
- the cost function J are updated as well

3.2.4 Classification

Once the classifier is trained, the classification of elements that have not been used in the classification is carried out using θ. We do this in the function classify_logistic_regression which takes as input parameter :

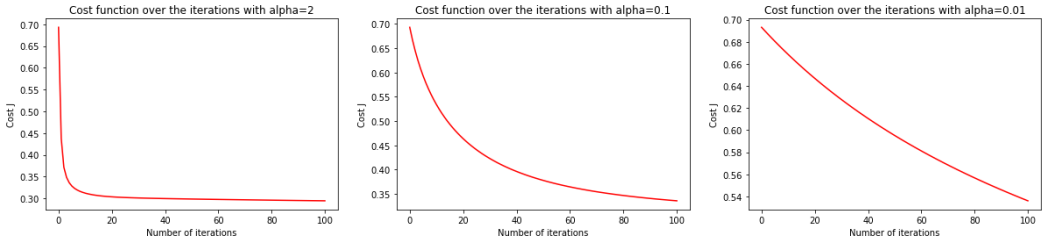
- X_test : Matrix containing the test data
- theta : Vector with the length n, the number of features of each pattern along with the parameters theta of the estimated function h after the training

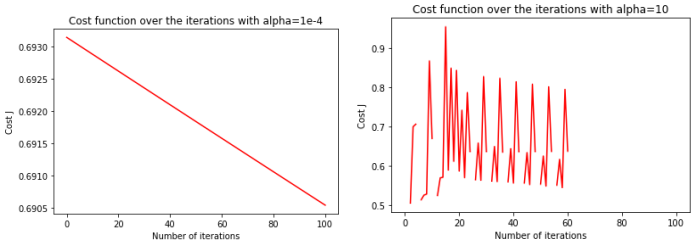
And returns the probability for each element on the test set to belong to the positive class.
We should know that our problem is binary :

- class 0 correspond to a low or medium wear level
- class 1 correspond to a high wear level.

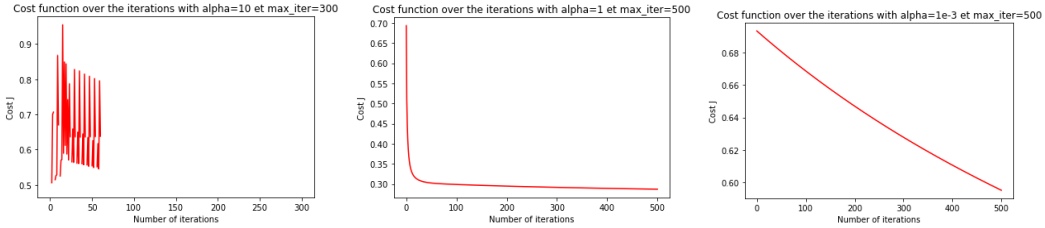
3.3 Running the code and testing

We train our code with all the functions of our model, and we are interested in the convergence of the gradient descent by plotting the graph of the loss function as a function of the maximum number of iterations. We know that the learning rate affects too much the convergence, so we will try different values and see the resulting changes.





We also vary the number of iterations :



We vary the learning rate between 0.0001 and 10 and we notice that when the value of alpha is too small the speed of convergence is almost zero even with a number of iterations too large. However, with a value of alpha too large, the gradient does not even converge with any number of iterations The number of iterations does not really affect the convergence of the gradient, it just increases the convergence time. The ideal will be to choose a learning rate neither too big nor too small like 1 and an average number of iterations in the averages of 200.

3.4 Assessment of the performance of the classifier

There are several methods for estimating the generalization error of a model during training. The estimated error supports the learning algorithm to do model choice ; i.e., to discover a model of the right complexity that is not affected by over fitting.
Because the model has been constructed, it can be used in the test set to forecast the class labels of earlier unseen data. It is often useful to measure the performance of the model on the test set because such a measure provides an unbiased estimate of its generalization error. The accuracy or error rate evaluated from the test set can be used to compare the associative performance of multiple classifiers on the equal domain.
In this section we will evaluate the performance of our logistic regression model by using the confusion matrix and computing the accuracy and the F1_score. These metrics will also help us to compare the logistic regression with other classification methods that we will implement in the last part of this project.
−**Confusion Matrix** : This is a performance measure for a machine learning classification problem where the output can be two or more classes. It is a table with 4 different combinations of predicted and actual values :

		Predicted	
		Negative **(N)** -	Positive **(P)** +
Actual	Negative -	True Negative **(TN)**	**False Positive (FP) Type I Error**
	Positive +	**False Negative (FN) Type II Error**	True Positive **(TP)**

— True Positive : You predicted positive and it's true
— True Negative : you predicted negative and it's true
— False Positive : (Type 1 Error) you predicted positive and it's false
— False Negative : (Type 2 Error) you predicted negative and it's false.

−**Accuracy** : The accuracy allows to know the proportion of good predictions compared to all predictions. The operation is simply : Number of good predictions/ Total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4}$$

−**Precision** :Precision is the number of items correctly assigned to class i relative to the total number of items predicted to belong to class i.

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

−**Recall** :The recall corresponds to the number of items correctly assigned to class i compared to the total number of items belonging to class i.

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

−**F_score** : This is the harmonic mean of Precision and Recall and gives a better measure of the incorrectly classified cases than the Accuracy Metric

$$F1\_score = 2 * \frac{Recall * Precision}{Recall + Precision} \tag{7}$$

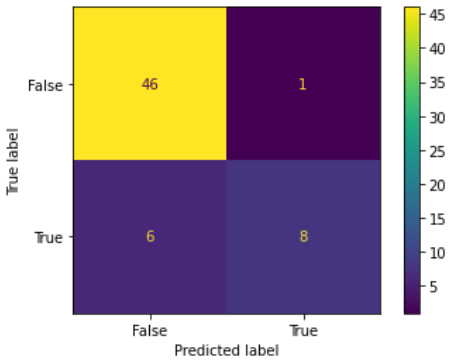
We implement these metrics using the following code :

```
Confusion_matrix=np.zeros((2,2))
for i in range(np.shape(Y_test_asig)[1]):
    if ((Y_test[0,i]==True).any() and (Y_test_asig[0,i]==True).any()):
        Confusion_matrix[0,0]=Confusion_matrix[0,0]+1
    if ((Y_test[0,i]==True).any() and (Y_test_asig[0,i]==False).any()):
        Confusion_matrix[0,1]=Confusion_matrix[0,1]+1
    if ((Y_test[0,i]==False).any() and (Y_test_asig[0,i]==True).any()):
        Confusion_matrix[1,0]=Confusion_matrix[1,0]+1
    if ((Y_test[0,i]==False).any() and (Y_test_asig[0,i]==False).any()):
        Confusion_matrix[1,1]=Confusion_matrix[1,1]+1

print(Confusion_matrix)
accuracy=Confusion_matrix.trace()/(Confusion_matrix[0,0]+Confusion_matrix[1,0]+Confusion_matrix[0,1]+Confusion_matrix[1,1])
Precision=Confusion_matrix[0,0]/(Confusion_matrix[0,0]+Confusion_matrix[1,0])
Recall=Confusion_matrix[0,0]/(Confusion_matrix[0,0]+Confusion_matrix[0,1])
f_score=2*((Precision*Recall)/(Precision+Recall))
```

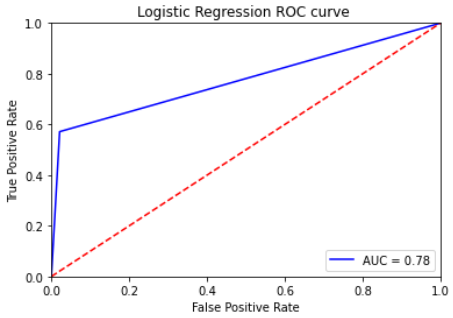
We should know that F1 score and accuracy are often discussed in similar contexts and have the same end goal, but they are not the same and have very different approaches to measuring model performance. F1 score's definition can be complicated for end users to understand, and would require an explanation around the difference between precision and recall. On the other hand, accuracy is universally understood which makes it much easier to use when communicating

model results.However, Accuracy does not perform well on imbalanced datasets which often leads to misleading results, whilst F1 is still able to measure performance objectively when the class balance is skewed. So,F1 score should be used when we have an imbalanced dataset or when communicating results to end users is not a key factor of the project and accuracy should be used when the dataset is balanced or when communicating the results to end users is important.
In our case we find the following values of the metrics and the confusion matrix :

```
**********
The accuracy of the Logistic Regression classifier is 0.8852
**********
**********
The Recall of the Logistic Regression classifier is 0.5714
**********

**********
The Precision  of the Logistic Regression classifier is 0.8889
**********

**********
The F1-score of the Logistic Regression classifier is 0.6957
**********
```



We obtain a fairly high accuracy as well as all the other metrics, namely the F_score which is higher than 0.5, which shows that our model is very efficient. In parallel, in the confusion matrix, we obtain that 7 individuals misclassified, a quantity that is small as well and a thing that proves moreover that the model is relatively good. To make sure more, we plot the ROC curve.



The ROC curve (Receiver Operating Characteristic) represents the sensitivity as a function of 1 - specificity for all possible threshold values of the marker studied. Sensitivity is the ability of the test to detect high wear levels and specificity is the ability of the test to detect low or medium wear levels. In our case we find an AUC of 0.78, which shows that the proportion of true positives and true negatives is quite high

4 Other Classifiers

In the previous sections, we have seen how to implement a powerful logistic regression model. However, it will always be interesting to test other classification models. For this reason we will code 3 other methods :

— Support Vector Machines
— K-nearest neighbors
— Decision Tree
— Random Forest
— Recurrent Neural Networks

In all these methods, we will use the functions already predefined in sckitlearn by starting to train the classifier with the training database, then make predictions with the test database and finally evaluate the performance using the metrics defined in the previous section.
But before testing these methods, we thought of trying logistic regression using the **LogisticRegression** function already defined in the sklearn linear library. In order to compare its performance with that of the from scratch model we implemented before.

4.1 Logistic Regression

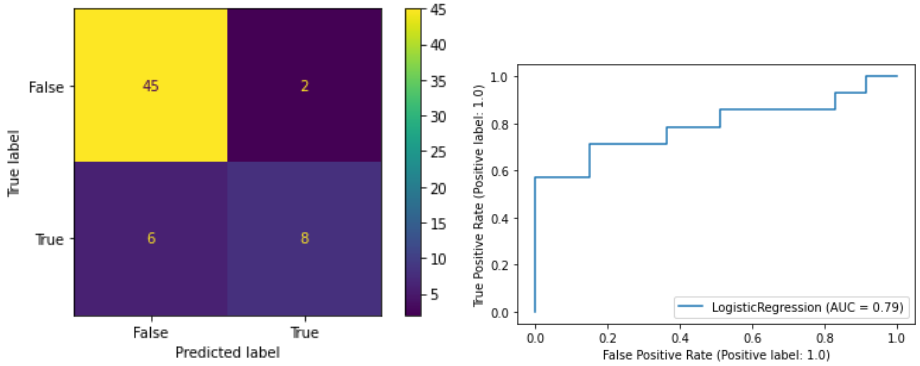
We use the following code :

```
ModelLogr=LogisticRegression()
ModelLogr.fit(X_train,Y_train)
PredLogr=ModelLogr.predict(X_test)
```

After predicting, we display the Accuracy and Fscore :

```
The accuracy of Logistic Regression is 0.8033
***************

***************
The F1-score of Logistic Regression is 0.6667
```

We display also the confusion matrix and the ROC curve :

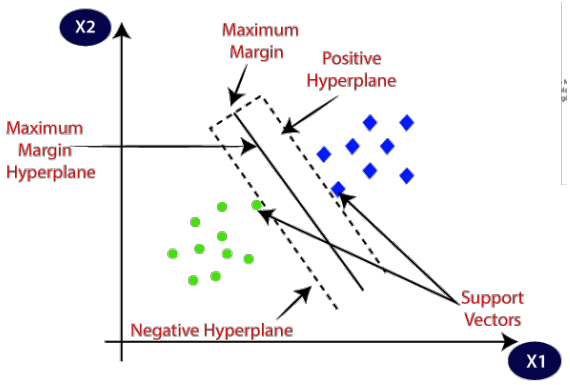


There is not a big difference between the two methods, the one from scratch and the one predefined in python. Indeed, we get almost the same values of the metrics and the same ROC curve.

4.2 Support Vector Machines

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.
SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane



This notion of frontier assumes that the data are linearly separable, which is rarely the case. To overcome this, SVMs often rely on the use of "kernels". These mathematical functions allow to separate the data by projecting them into a feature space. The technique of margin maximization allows us to guarantee a better robustness to noise - and therefore a more generalized model.
In sklearn, the function of SVM use different kernels such as :

- linear
- polynomial
- rbf
- sigmoid

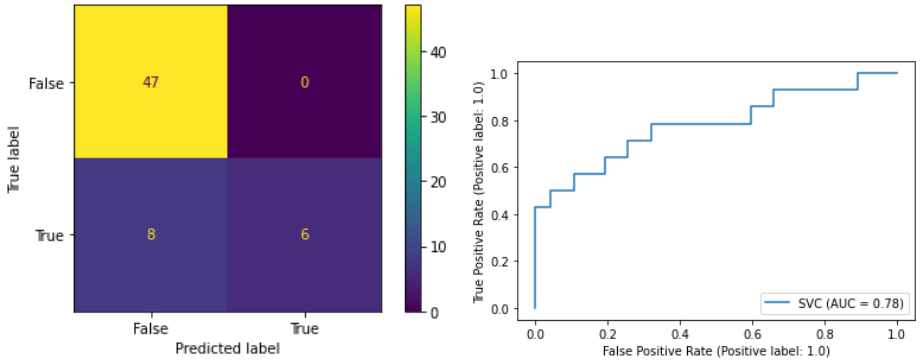
Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset. So, In order to determine the best function for our case, we tested all the functions and we found that the function ' rbf' is the one which gives the best performance.

```
ModelSVM = svm.SVC(kernel='rbf')
ModelSVM.fit(X_train, Y_train)
PredSVM=ModelSVM.predict(X_test)
```

After predicting, we display the Accuracy and Fscore :

```
***************
The accuracy of the SVM classifier is 0.8689
***************

***************
The F1-score of the SVM classifier is 0.6000
***************
```

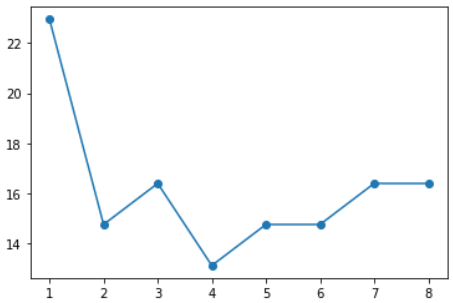
We display also the confusion matrix and the ROC curve :



The confusion matrix represents much better values than the confusion matrix of our own classifier built from scratch : we have no items falsely classified 1. However, we have classified 8 items in class 0 wrongly, we will try to improve it accordingly.

4.3 K-nearest neighbors

This time we use the nearest neighbor method to do the classification. k-Nearest Neighbours (k-NN) is a standard classification algorithm that relies exclusively on the choice of the classification metric. It is "non-parametric" (only k must be fixed) and is based only on the training data. Indeed, from a labeled database, we can estimate the class of a new data by looking at which is the majority class of the k nearest neighboring data.
Since K the number of neighbors is the most important parameter in this algorithm, we will calculate the error for different values of k in order to choose the one that minimizes the error rate for the test set.



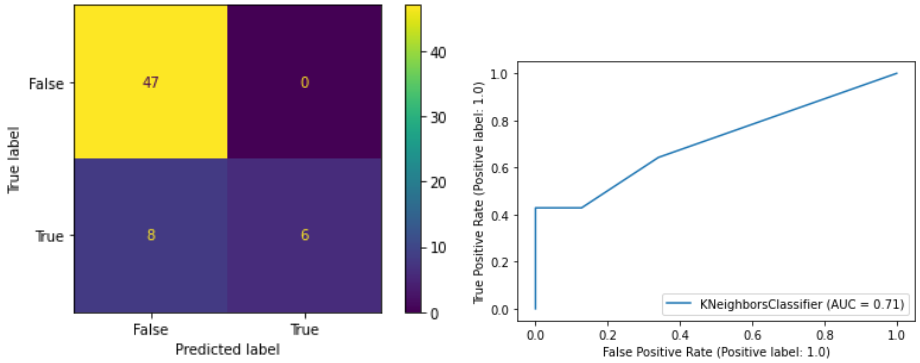
We notice that the best value is k=4. Above this value, we can observe the phenomenon called overfitting which occurs when the training data used to build a model explains the data very well or even "too well" but fails to make useful predictions for new data.
We implement the method using the following code.

```
Modelknn = KNeighborsClassifier(n_neighbors = 4)
Modelknn.fit(X_train,Y_train)
PredKnn=Modelknn.predict(X_test)
```

After predicting, we display the Accuracy and Fscore :

```
The accuracy of KNN is 0.8689
***************

***************
The F1-score of KNN is 0.6000
```

We display also the confusion matrix and the ROC curve :



The AUC value displayed is less than those of the previous methods

4.4 Decision Tree

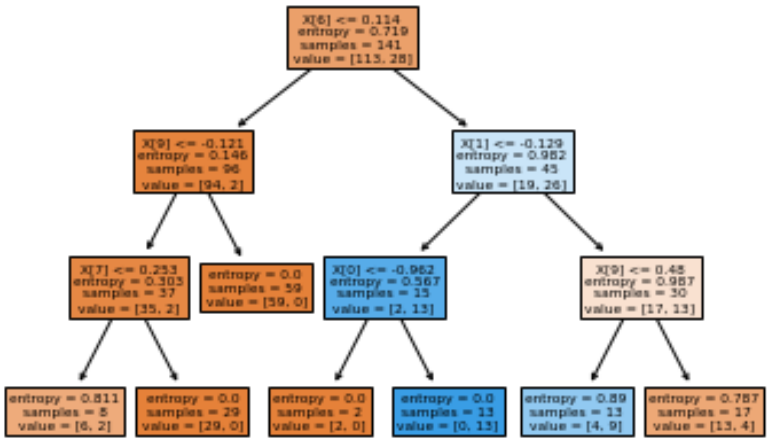
Decision Tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.
However, the criterion for selecting variables and hierarchy in decision trees can be tricky. Indeed, Variables are selected based on a complex statistical criterion that is applied to each decision node.
Now, the criterion for selecting variables in decision trees can be done via two approaches :

— 1. Entropy and information gain
— 2. Gin Index

Entropy is a measure of the disorder or impurity in a node. The Gini or impurity index measures the probability that a random instance will be misclassified when chosen at random.
In our case, we have chosen to work with entropy and a maximum tree depth of three levels, in order to avoid over-fitting.

```
ModelTree=DecisionTreeClassifier(criterion='entropy',max_depth=3)
ModelTree.fit(X_train,Y_train)
PredTree=ModelTree.predict(X_test)
```

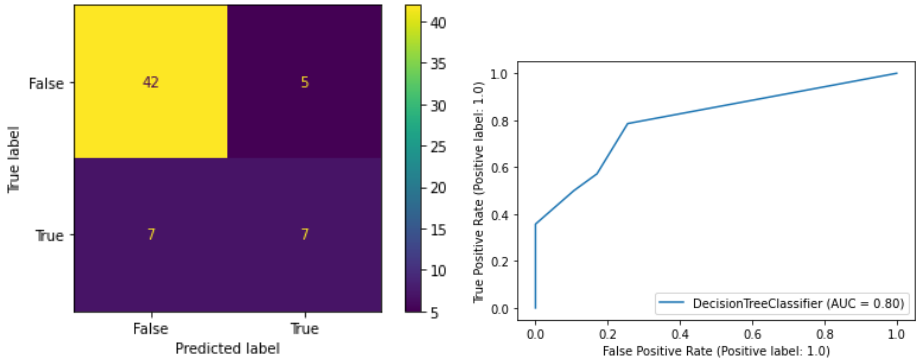
this model gives us the following tree



We see that the classifier uses each feature of the image to divide into decision forms. For example, it starts in the first root to divide according to x[6] which matches well with minor_axis_length with a threshold of 0.114. This classifier also calculates the entropy at each level, and we can see how it decreases as we go deeper into the tree, until it reaches a zero value, that is, the impurity decreases, which means that the images are better classified.
After predicting, we display the Accuracy and Fscore :

```
The accuracy of the Decsion Tree classifier is 0.8033
***************

***************
The F1-score of the Decision Tree classifier is 0.5385
```

We display also the confusion matrix and the ROC curve :



The confusion matrix is worse than the confusion matrices presented previously : we have 5 falsely 1-classified elements, unlike the other implemented algorithms where the falsely 1-classified elements were at most 1 . However, we have classified 7 elements in class 0 wrongly. This makes a total of 12 misclassified elements.

4.5 Random Forest

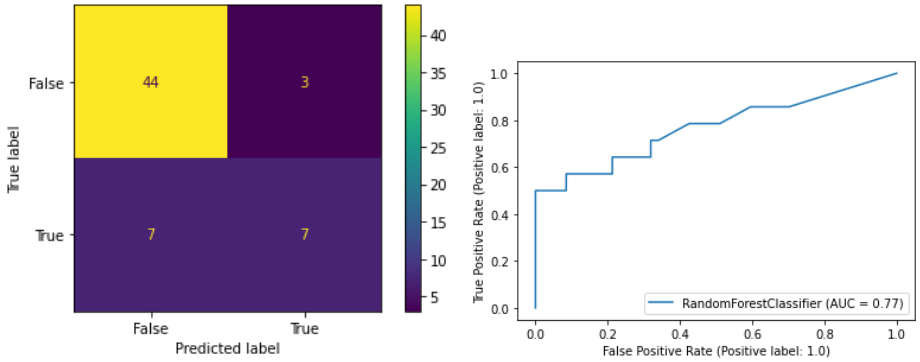
The random forest algorithm is an extension of the bagging method as it utilizes both bagging and feature randomness to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or the random subspace method, generates a random subset of features, which ensures low correlation among decision trees. This is a key difference between decision trees and random forests. While decision trees consider all the possible feature splits, random forests only select a subset of those features.
We implement our random forest model using the following code :

```
ModelRanF=RandomForestClassifier(n_estimators=100,criterion='entropy')
ModelRanF.fit(X_train,Y_train)
PredRanF=ModelRanF.predict(X_test)
```

After predicting, we display the Accuracy and Fscore :

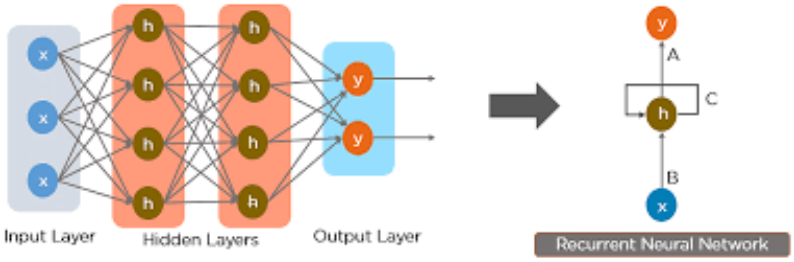
```
The accuracy of the Random Forest classifier is 0.8361
***************

***************
The F1-score of the Random Forest classifier is 0.5833
```

We display also the confusion matrix and the ROC curve :



4.6 Recurrent Neural Networks

Neural networks, also known as artificial neural networks or simulated neural networks, are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.
Artificial neural networks are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.
A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes. This allows it to exhibit temporal dynamic behavior.



We implement a neural network with an input layer of 5 neurons, 4 hidden layers so that each layer contains 5 neurons and an output layer that contains 2 neurons. We use the activation function Relu in all layers except the last one in which we use the softmax function.

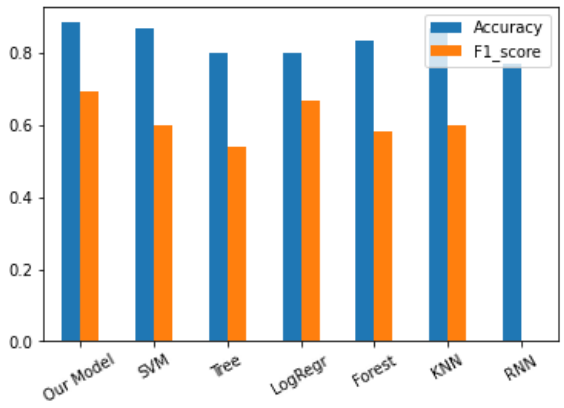
```
y_train=to_categorical(Y_train)
y_test=to_categorical(Y_test1)
model=Sequential()
n_cols=X.shape[1]
model.add(Dense(5,activation="relu",input_shape=(n_cols,)))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(2,activation="softmax"))
model.compile(optimizer="adam",loss="categorical_crossentropy",metrics=["accuracy"])
model.fit(X_train,y_train,epochs=10)
```

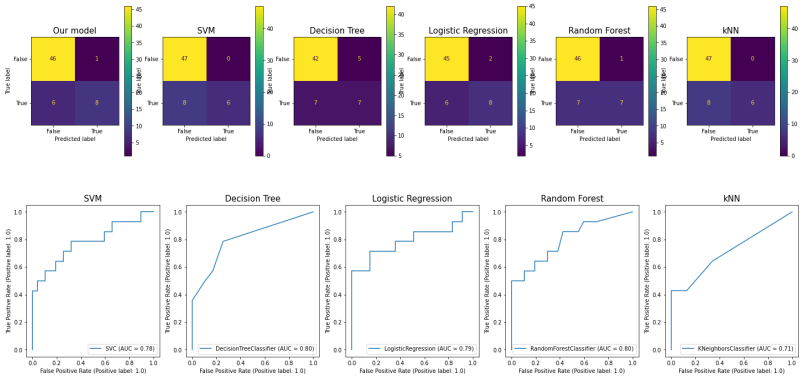
After predicting, we display the Accuracy and Error :

```
Accuracy of RNN is:0.7704917788505554
 Error  of RNN is:0.22950822114944458
```

5 Comparison

We compare the different methods based on the accuracy and F1_score values as well as the confusion matrix and the ROC curves





We notice that all the models have succeeded in classifying our data very well. Indeed, for all the methods the accuracy is in the range of 80% and the F1_score is between 0.5 and 0.7. However, it remains clear that the logistic regression, whether it is the one we created from scratch or the one using the predefined function, gives the best results, especially according to the F1_score.

6 Improve the classification performance apart from using a different classifier

We have been able in the previous sections to try several classification algorithms. However, we noticed that they all have almost the same performance. This leads us to think of other ways to improve the performance other than trying several methods.

One of the easiest ways to improve the accuracy of machine learning models in general is to deal with missing values and outliers. This may not be our case. However, if the presence of data that are missing values or contain outliers, the models will probably be less accurate. This is because missing values and outliers can cause the model to make incorrect assumptions about your data. It is also important to note that missing values and outliers can cause models to overfit or underfit.

The second method is feature selection, which helps identify the most useful features in the dataset. The goal is to reduce or eliminate noise and improve model accuracy by removing redundant information. This can be done by using the **PCA** method which identifies the smallest number of features needed to make an accurate prediction, as well as using the **Pearson correlation coefficient** to measure the strength of the relationship between two variables, and then removing the variables that are less correlated than the others.

The setting of hyperparameters is very important. These parameters can include things like the number of layers or the type of activation function in a deep neural network, the type of kernel in SVM, the number of neighbors in KNN. This is where cross-validation can be useful. By dividing the data into training and test sets, one can try different combinations of hyperparameters on the training set, and then see how they perform on the test set. This helps to find the best combination of hyperparameters.

We try in this context to improve our neural network model in order to increase the accuracy value and decrease the error value. We then change the loss function by the binary cross entropy function and the activation function of the last layer by the sigmoid function instead of the Softmax function. It is important to know that these two activation functions return values between 0 and 1 except that the outputs of softmax are interdependent and the probabilities will always total one by design. Softmax is also used for multi-classification in the logistic regression model, while Sigmoid is used for binary classification in the logistic regression model. Since our problem is binary, sigmoid will be the best choice and indeed we see that the accuracy to increase to 81% instead of 77% and the error to decrease to 0.18 instead of 0.22.

```
y_train=to_categorical(Y_train)
y_test=to_categorical(Y_test1)
model=Sequential()
n_cols=X.shape[1]
model.add(Dense(5,activation="relu",input_shape=(n_cols,)))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(5,activation="relu"))
model.add(Dense(2,activation="sigmoid"))
model.compile(optimizer="adam",loss='binary_crossentropy',metrics=["accuracy"])
model.fit(X_train,y_train,epochs=10)
#Prediction
predRNN=model.predict(X_test)
Accurracy=model.evaluate(X_test,y_test,verbose=0)
print("")
print("***************")
print("Accuracy of RNN is:{} \n Error  of RNN is:{}".
      format(Accurracy[1],1-Accurracy[1]))
```

7 Conclusion

During this project, a logistic regression model was implemented and also tested on binary images. We were also able to compare it with other classification methods like support vector machine, RNN, and decision tree. Finally, we evaluated the performance of all these models using different metrics.
Of course, in this first project, the processed images are binary images. The simplicity of the input data explains this small difference between the predictions of the different classifiers.
In project 2, we will process RGB images, and we will see what impact this will have on the results.
You will find the two code files attached to this report. The file classify_inserts.py also contains the algorithms of the different classifiers.