University of Western Ontario
London, Canada
Department of Computer Engineering
Software Engineering Program, Data Analytics Course

Assignment 1: Breast Cancer Detection

Written by: Tina Gholami
Student No: 251190343

Date: 18th October 2020
Supervisor: Prof. Katarina Grolinger

Table of Contents:

(I) Problem

The problem is a binary classification problem, and the goal is to classify breast cancer data as either benign or malignant. The dataset consists of several feature values of cancer tumors driven from precise medical examinations, and I have to implement my chosen Machine Learning models based on the dataset's features. The approach I took is further analyzed in section (III).

(II) Dataset

The dataset is from UCI's Breast Cancer Wisconsin (Modified) dataset, collected by Dr. William H. Wolberg, University of Wisconsin Hospitals, Madison, Wisconsin, USA. Samples arrive periodically, as Dr. Wolberg reported his clinical cases. The dataset, therefore, reflects this chronological grouping of the data. The dataset involves a total of 683 observations and 11 features. Such attributes include:

1. Sample code number: id number
2. Clump thickness: 1 - 10
3. Uniformity of cell size: 1 - 10
4. Uniformity of cell shape: 1 - 10
5. Marginal Adhesion: 1 - 10
6. Single Epithelial Cell Size: 1 - 10
7. Bare Nuclei: 1 - 10
8. Bland Chromatin: 1 - 10
9. Normal Nucleoli: 1 - 10
10. Mitoses: 1 - 10
11. Class: (2 for benign, 4 for malignant)

(III) Methodology and Algorithms

After operating several necessary data pre-processing techniques on the dataset, the methodology I used to resolve this problem was implementing three different classification algorithms on the dataset to evaluate their performance for comparison later on. I employed the following algorithms:

1. Decision Tree Classifier
2. Kernel SVM (support vector machine) Classifier
3. ANN (neural network) Classifier

Also, the metrics I used to evaluate my models are:

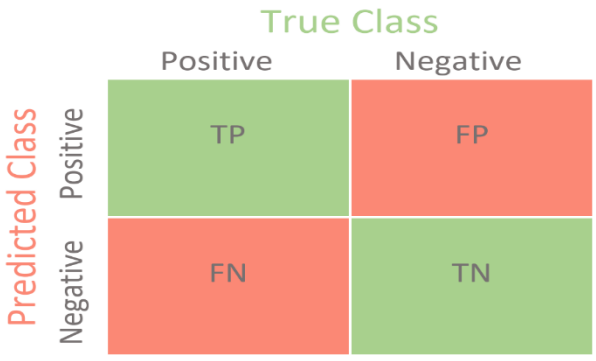
1. Accuracy score and confusion matrix
2. Precision
3. AUC (area under the curve) and ROC (receiver operating characteristic curve) curves

I prefer to give a short description of how each of the algorithms mentioned above works, and then I will delve into my implementation and results of the code accordingly. I will start with the first model and then move on to the other models. But before I begin discussing each model, the data engineering or the data pre-processing technics should be explained.

But for now, I will explain what each of the above metrics means:

1. Accuracy score and confusion matrix:

   As its name suggests, the confusion matrix returns a 4×4 matrix, explaining the model's performance. We also use this matrix as a basis to define other metrics. The columns and rows of the confusion matrix represent true positive, false positive, false negative, and true negative as in the following picture:



Confusion matrix schematic, towardsdatascience.com

True positive (TP) refers to the number of predictions where the classifier correctly predicts the positive class as positive.

False positive (FP) refers to the number of predictions where the classifier wrongly predicts the negative class as positive.

True negative (FN) refers to the number of predictions where the classifier correctly predicts the negative class as negative.

False negative (FN) refers to the number of predictions where the classifier wrongly predicts the positive class as negative.

Therefore, the accuracy score can be calculated by taking the average of the values lying across the main diagonal of the confusion matrix over all the samples, i.e.:

$$Accuracy = \frac{True\ positive\ (TP) + True\ negative\ (TN)}{Total\ samples}$$

Thus, accuracy helps us understand how well our model is performing. But we have to beware of a common issue with the accuracy score, which happens when one particular class in the labels is quite rare. In that case, we might have a pretty lousy model, but the accuracy is misleadingly high. Therefore, it is suggested to employ other Machine Learning metrics, just in case.

2. Precision:

$$Precision = \frac{True\ positive\ (TP)}{True\ positive\ (TP) + False\ positive\ (FP)}$$

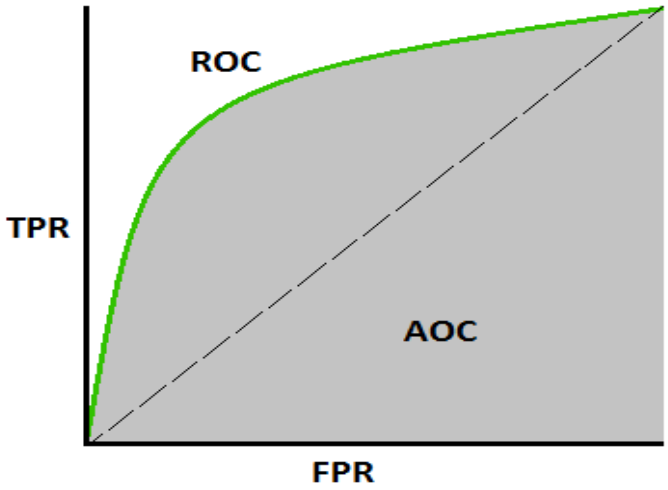
Precision is about how precise our model is from the predicted positives, meaning how many of the predicted positives are actual positives. Precision is a good measure to determine if the costs of false positives are high. For instance, in an email spam detection model, if the cost of false positives is high, many non-spam emails will be predicted as spam. Therefore, the user might lose vital information in the process.

Hence, the precision score is a useful metric to use alongside with the accuracy score.

3. AUC (area under the curve) and ROC (receiver operating characteristic curve) curves:

"roc_auc_score" method in Sklearn computes area under the receiver operating characteristic curve from prediction scores. It is a performance measurement specifically for the classification problems at various thresholds. ROC is a probability curve, and AUC displays a degree of separability. It indicates the model's ability to distinguish between classes. Hence, the higher the AUC, the better our model predicts class 0 as 0 and class 1 as 1. In my dataset, the higher the AUC, the better my model distinguishes between patients with benign tumors and those with malignant type.

Here is an analogy of AUC and ROC:



AUC and ROC (X-axis: false positive rate; Y-axis: true positive rate), towardsdatascience.com

Where TPR and FPR are:

$$TPR = Recall = Sensetivity = \frac{True\ positive\ (TP)}{True\ positive\ (TP) + False\ negative\ (FN)}$$

$$FPR = 1 - Specificity = \frac{False\ positive\ (FP)}{True\ negative\ (TN) + False\ positive\ (FP)}$$

An excellent model has AUC near to 1, meaning great ability to differentiate classes or separability. And a model with AUC near 0 indicates a worse measure of separability.

(0) Data Engineering (Data Preparation)

When dealing with any data science and data analytics problem, the first thing that should be considered is how clean and consistent the data is. Towards that end, I have to prepare the raw, messy data for further analysis. This process is called data engineering, data preparation, or simply data cleaning. When cleaning the data, one should always look for the followings:

1. Missing values
2. Encoding categorical data
3. Encoding categorical labels
4. Splitting the data into train set and test set
5. Feature scaling

After taking care of the above points, the data will be prepared to be fed into the Machine Learning model.

I first imported the required libraries, namely Pandas, Numpy, Matplotlib, and later on, Sklearn and Tensorflow. Second, I loaded the data named "Data.csv" and separated the features (denoted by variable "x") from the labels (indicated by "y"). One important thing to mention is that I excluded the first column of the dataset, "Sample code number," from "x" because this column is merely patients' samples' numbers, which does not contain any valuable information to base the model upon. Therefore, I started to include the second column up to the very last column in the features set "x." The last column is named "class," which is the labels I included in "y." Next, I checked missing values by the ".isnull" method. But since the function returned 0, that means I don't have missing values in the dataset. Although if there were any, I could either drop those rows or fill in the missing values using the previous samples' mean. But since my dataset has a sensitive medical application, if the dataset had few missing values, I would have probably dropped those rows because I would not want to make up false numbers in my dataset by filling in the missing parts with the mean or any other method. Then, I used the hold-out method to separate train data from the test data. I used 25% of the whole dataset as the test set and the remaining 75% as the train set. The method to do so is called "train-test-split" from the Sklearn library. By now, the train set has shape (512, 9) and the test set has shape (171, 9). Accordingly, the train set has 512 labels, and the test set has 171 labels. The next step is feature scaling. I employed standardization rather than normalization since the former will take care of the outliers far better than the latter method. I used this scaler (denoted by "sc") to transform and fit the train set. But, I only used the scaler to transform the test set. I cannot fit any scaler of any sort to the test set since the model should have no prior knowledge of the test data, and fitting this scaler to the test set will calculate its mean and variance; hence, extracts information from the test set. The dataset is now ready to be fed into the model that I will explain in the next part.

(1) Decision Tree Classifier

As I mentioned in (III), I trained three distinct classification algorithms on my dataset to evaluate their performance. The first model I decided to implement was the decision tree classifier. But first, I intend to give a concise description of how this algorithm works and then delve into my use of it.

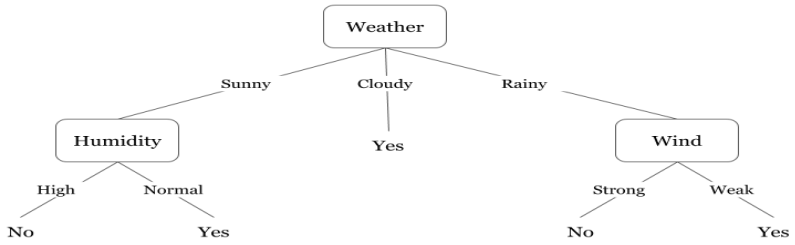
It is better to start with CART. CART stands for "Classification and Regression Trees." So, classification trees help classify the data into different categories. But, regression trees are designed to predict real-valued outcomes and find an appropriate function that maps input data to the output with the most minimum possible objective function, i.e., error or cost function. But since the problem is a classification problem, I used the classification tree model. So, the way a classification decision tree works is to slice the input data into several splits. The split is done in such a way to maximize the number of a certain category in each of these splits, meaning the algorithm aims to minimize the general "entropy". The root of each sub-tree is the variable having the highest "information gain." Therefore, going down from the main root of a decision tree, the entropy decreases by each level downwards. The following relations calculate the entropy:

$$\text{Entropy } H = -\sum p(X).\log p(X)$$

$$\text{Information Gain } \mathrm{G(T,X)} = H(T) - H(T,X)$$

In the above equations, p denotes the probability of each feature class.

The tree's main root is the feature that has the highest information gain, i.e., the lowest entropy (lack of information). Then we choose each sub tree's root using the same method until we reach the "leaves" of the tree. "Leaves" are the final categorical labels of the dataset. These were the general steps a classification decision tree model takes.



Example of a Decision Tree Classifier, hackerearth.com

In my implementation, after I prepared the data in part (0), I used the decision tree classifier method in Sklearn called "DecisionTreeClassifier." For parameter tuning, I got a fair result with criterion as "entropy." Then I fit the train data to my model. Next, I predicted the test set labels as "y_pred" and used it alongside the actual test set labels "y_test" to calculate the metrics, i.e., confusion matrix, accuracy, precision, and roc_auc score.

The confusion matrix is:

```
[[104   3]
 [  4  60]]
```

Confusion matrix for the decision tree classifier

And the accuracy is:

```
0.9590643274853801
```

Accuracy score of the decision tree classifier

And the precision is:

```
0.9576719576719577
```

Precision score of the decision tree classifier

And the ROC_AUC score is:

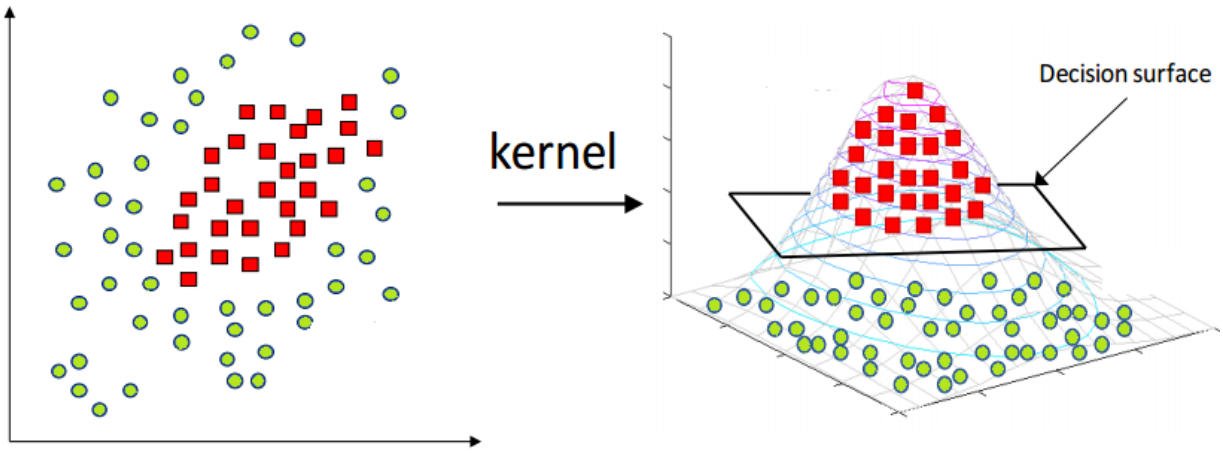
```
0.9547313084112149
```

ROC_AUC score of the decision tree classifier

Therefore, this is a relatively accurate decision tree classifier.

(2) Kernel SVM (Support Vector Machine) Classifier

The intuition behind the Kernel SVM model is similar to the SVM model. Still, the difference is that despite using a relatively linear decision boundary, which is the default in the SVM model, we use a non-linear classifier. We define the non-linear function in the "kernel" part. For instance, an "RBF" (radial basis function) kernel is widely used for Kernel SVM models. We know that the SVM model tries to find a linear boundary to classify data points into distinct classes. To do so, it first tries to find the "support vectors," the points on the decision boundary, and then fits a line in a way to minimize the sum of distances between the line and other non-support vector points. But if the data is no longer linearly separable, we have to use the kernel trick, aka, Kernel SVM model.



Kernel trick, medium.com

So, the Kernel SVM model converts lower dimension space (2D in the leftmost picture above) to a higher dimension space (3D in the rightmost picture above); hence, we will get a linear classification again (the decision surface above). We are basically projecting (mapping) the data with some extra features to a higher dimension. Furthermore, there are multiple kernels that we can benefit from, such as:

Gaussian RBF Kernel: $K(\vec{x}, \vec{l_i}) = e^{-\frac{\|\vec{x} - \vec{l_i}\|^2}{2\sigma^2}}$

Sigmoid kernel: $K(X, Y) = \tanh(\gamma . X^T Y + r)$

Polynomial kernel: $K(X, Y) = (\gamma . X^T Y + r)^2 , \gamma > 0$

In my implementation, after I prepared the data in part (0), I used the SVC (support vector classifier) method in Sklearn called "SVC" and specified the kernel as RBF. The reason why I preferred the RBF kernel is that not only it introduces nonlinearity to the model, but also it is relatively easy to calibrate. For instance, the polynomial kernel has three parameters (offset, scaling, degree). But the RBF kernel has only one.

After that, I predicted the test set labels as "y_pred" and used it alongside the actual test set labels "y_test" to evaluate my SVM model.

The confusion matrix is:

```
[[101   6]
 [  3  61]]
```

Confusion matrix for Kernel SVM model

And the accuracy is:

```
0.9473684210526315
```

Accuracy score for the Kernel SVM model

And the precision is:

```
0.9408008036739379
```

Precision score for the Kernel SVM model

And the ROC_AUC score is:

```
0.9485251168224299
```

ROC_AUC score for the Kernel SVM model

Therefore, we have a relatively accurate Kernel SVM classifier (SVC).

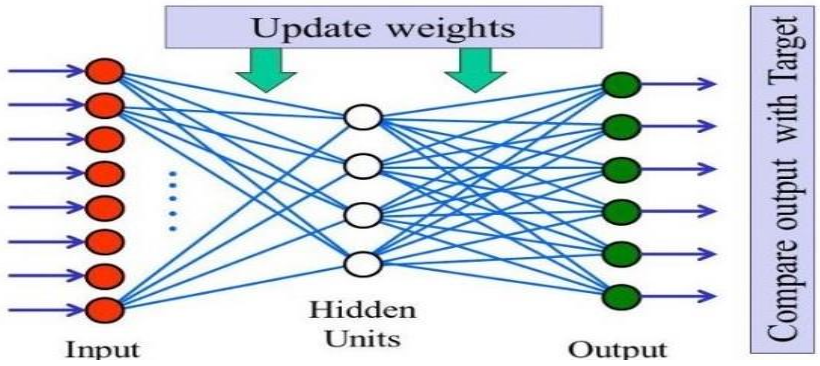
(3) ANN (Neural Network) Classifier

A neural network is a series of algorithms that endeavor to recognize underlying relationships in a dataset through a process that mimics how the human brain operates. Towards this end, it uses a set of inputs and assigns them initial weights and a total bias that will be updated later in the back-propagation stage. First, the inputs are multiplied by their weights, and then they will be fed into an activation function, aka a perceptron or a neuron. This layer is called the input layer. Then, the outputs can again go through similar layers, each getting multiplied by different weights. These middle layers are called hidden layers. At last, the values go into the output layer to yield the final output(s). This is the first epoch or iteration of the algorithm. This output is then compared to the real output, aka labels, and the relative error (For instance, MSE or minimum square error) is calculated. Then this error is back-propagated through the whole network to update the weights. This process repeats all over again until either we reach a desired small error or until the weights converge to a specific constant value. If so, this will be the last epoch. This network is called an artificial neural network (ANN) because it mimics how the brain neurons work. By updating the weights, the network is basically learning the importance of each feature (feature extraction) and then fires it up/down if the input is fed to that neuron in the feature.

The whole algorithm should run in several epochs for weights to converge or for the error to decrease. But we have to be aware of overfitting. If the epochs are too much or the model follows the train set too accurately, then the model will give a fair accuracy on the train set but a relatively low accuracy on the new test set. Meaning, the accuracy increase on the train data increases, but the generalization, which is the goal, decreases.

Moreover, we can use methods such as early stopping to combat overfitting. This method will introduce a new parameter named "patience" not to let the algorithm run the whole epochs if overfitting begins to happen. So this will reduce the run time as well.

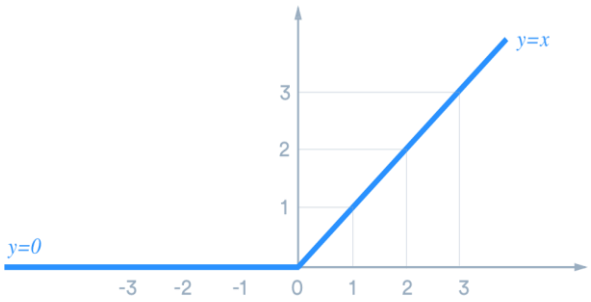
Furthermore, different optimization methods are used when dealing with neural networks. A very popular method is the Adam optimizer. We can also take advantage of SGD (stochastic gradient descent) as well.



Example of a Neural Network. groupfuturista.com

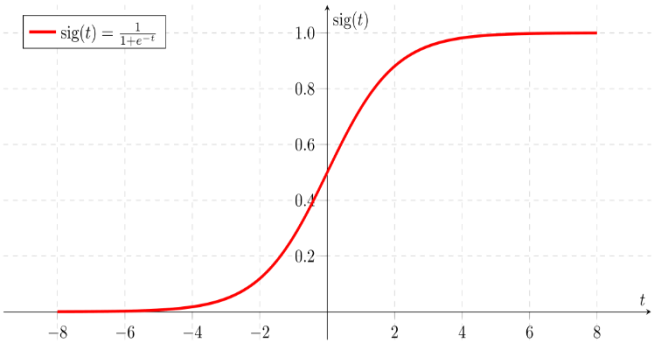
In my implementation, after loading the dataset, I performed label encoding on labels "y." The reason is that the classes in "y" are "2" and "4". These are categorical binary data, and the model will predict them as "0" and "1" later on. Hence, for the model to understand and analyze the labels easier and to prevent leakage, I need to encode the labels. Therefore, I encoded them using the "LabelEncoder" method to convert them to "0" and "1". Also, the reason for not using section (0) instead is that I wanted to encode the labels for this model. And encoding happens right after loading the data. Therefore, I reloaded the data and performed data cleaning separately for the neural network model. Next, I split the data into the train set and test set by 75% and 25%. After that, I performed feature scaling the way I did in section (0). Next, I designed the neural network model.

For this part, I first imported the Tensorflow library. The "ann" model is created by calling the "Sequential" method at first. This method will lay the general format and structure of a fully connected neural network. Next, I added the hidden layers consecutively using the "Dense" method. The architecture I chose is for the network to have two hidden layers, each containing ten neurons (or "units"), and activation function "Relu" (rectified linear unit).



Relu activation function, medium.com

Then, I added the output layer. Since there are binary labels ("0" or "1"), only one neuron is required for the output layer with the "Sigmoid" activation function.



Sigmoid activation function, wikimedia.org

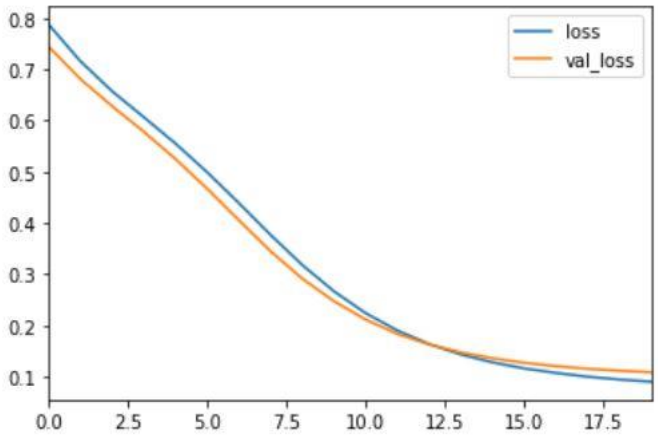
Next, I have to compile the model. I used the "Adam" optimizer and "binary_crossentropy" as my loss/cost function (because there are categorical binary labels). Since I wanted to make sure whether or not overfitting is happening, I chose the hyper-parameter "metric" as "accuracy."

In many cases where the dataset is large, or the model is being trained through too many epochs, to prevent overfitting, we employ the Early stopping method denoted by "EarlyStopping" in Python, and it monitors the validation loss "val_loss." I chose hyper-parameter "patience" as one since the model is not too complex. Then, I fit the "ann" model to the train set. I trained the model for twenty epochs and used the test set as my validation data. Also, I assigned the Early stopping transformer to the "callbacks" parameter. As it was noticed, using the Early stopping method reduces the run time as well.

Here are the train set error and the test set error displayed in both a table and a diagram. As we see below, the error on the test set is not much larger than it is on the train set. It also is a strictly decreasing line. Thus, no overfitting has occurred.

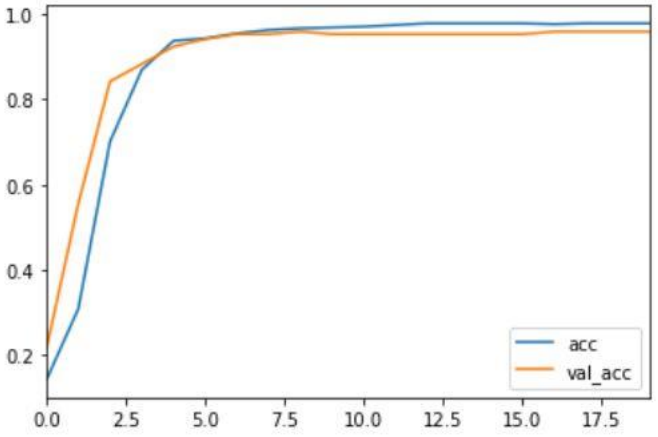
	loss	acc	val_loss	val_acc
0	0.787795	0.142578	0.744429	0.216374
1	0.716188	0.308594	0.681340	0.555556
2	0.657532	0.701172	0.628280	0.842105
3	0.606735	0.869141	0.578335	0.883041
4	0.555112	0.937500	0.524863	0.923977
5	0.498929	0.943359	0.466778	0.941520
6	0.438707	0.955078	0.405126	0.953216
7	0.376532	0.962891	0.344513	0.953216
8	0.317678	0.966797	0.291026	0.959064
9	0.266170	0.968750	0.246819	0.953216
10	0.223567	0.970703	0.211061	0.953216
11	0.189764	0.974609	0.183118	0.953216
12	0.163264	0.978516	0.162547	0.953216
13	0.143018	0.978516	0.147063	0.953216
14	0.127502	0.978516	0.135561	0.953216
15	0.115728	0.978516	0.126704	0.953216
16	0.106957	0.976562	0.120137	0.959064
17	0.099675	0.978516	0.115165	0.959064
18	0.094138	0.978516	0.111378	0.959064
19	0.089814	0.978516	0.108244	0.959064

Error (loss) and accuracy on the train set and test set



Error on the train set VS. test set

And also:



Accuracy on the train set VS. test set

Now that the "ann" model is successfully trained, I need to evaluate it using the metrics mentioned in section (III).

The confusion matrix is:

```
[[103   4]
 [  3  61]]
```

Confusion matrix of the ANN model

And the accuracy is:

```
0.9590643274853801
```

Accuracy score of the ANN model

And the precision is:

```
0.9550798258345428
```

Precision score of the ANN model

And the ROC_AUC score is:

```
0.9578709112149533
```

ROC_AUC score of the ANN model

Therefore, this is a relatively accurate ANN model.

(IV) Results Comparison

To yield a result, I have implemented three different Machine Learning algorithms on the UCI's breast cancer dataset.

The decision tree and the ANN model both had the same accuracy. However, the decision tree model has higher precision. On the other hand, the ANN model has a higher ROC_AUC score. All these three metrics for both of the decision tree and the ANN model are in the range of 95% and are quite similar. But to choose one model over the other, I would like to mention that I am classifying cancerous tumors. It is crucial that the model confidently distinguishes between benign and malignant tumors. Also, accuracy is the same for both models. Therefore, I believe the ROC_AUC score (which also has precision embedded inside) is the defining factor. This score indicates that the ANN model has a better performance. However, the difference in the decision tree and the ANN model performance is relatively small and in the range of 0.1%.

The accuracy, precision, and the ROC_AUC score for the SVM model are all less than that of the decision tree and the ANN model. To be more precise, these three metrics for the SVM model are all in the range of 94%.

Moreover, there are several methods to improve each model's performance. For instance, I could employ a Random forest or even a CatBoost instead of the decision tree model. For the ANN, I could add more hidden layers containing more neurons. I could also use other complex activation functions. Lastly, I could benefit from a more complex kernel for the SVM model, capable of introducing more nonlinearity to the problem.

Thus, according to the evaluation of these three models performance, I can rank them in the following order:

1. ANN (deep neural network) model
2. Decision tree model
3. Kernel SVM model