Multilayer Perceptron Project

Arnas Juravicius – 18257305
Oisin McNamara - 18237398

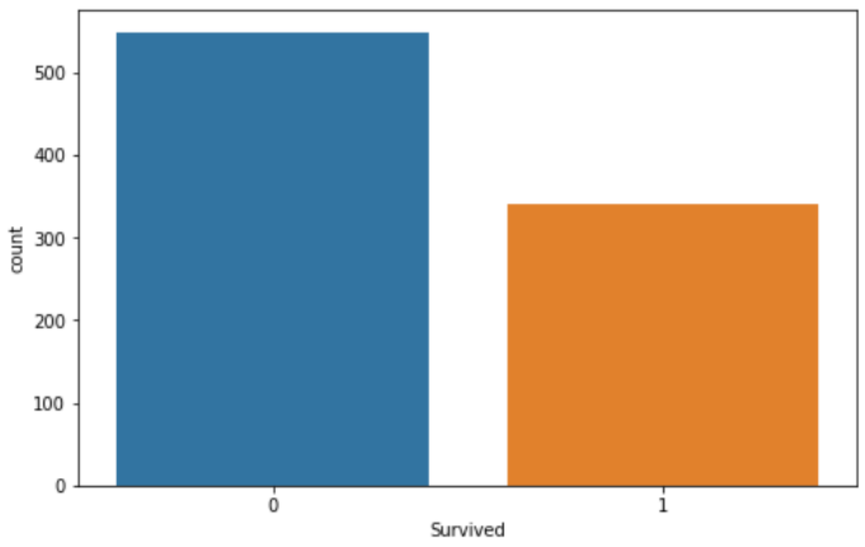
The data set

The dataset we used for the MLP was the Titanic Survival Dataset. This dataset conveyed whether passengers survived the sinking. There are two files in our dataset train.csv and test.csv. Train.csv has all the data we are going to train the model with. Test.csv has all the same data except the “survived” column, this is because we need to test our model on unseen data. The information in the dataset was Passenger Sex, Passenger Class, Passenger Age, etc. We identified that dropping the Age column returns biased, linear results when running our model, so we decided to keep it in.

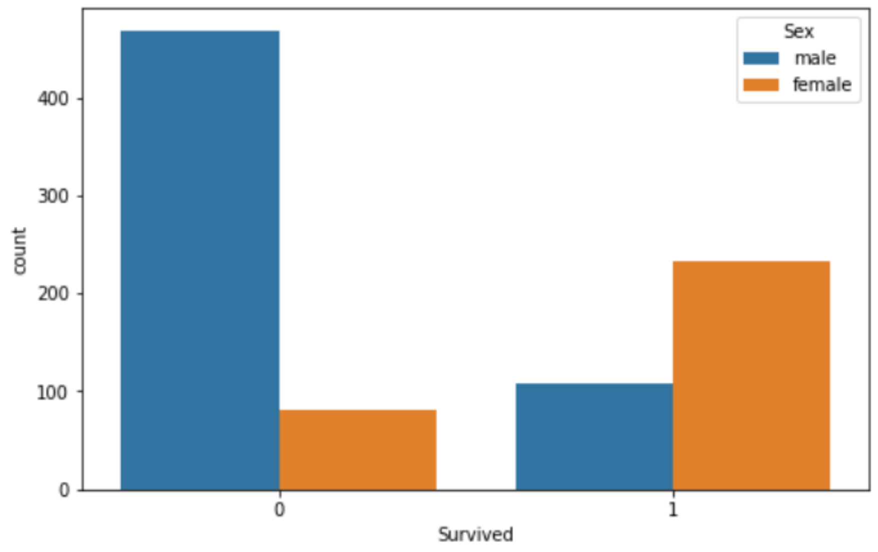
When using the dataset for training the MLP we dropped several columns that we found to be irrelevant when training the data. The columns that we dropped were Name, Ticket, Cabin, Embarked, Person, Fare. We dropped Name, Ticket, Embarked, Person (We created this column to generate whether a person was a child or an adult), and Fare as these columns had no impact on the passenger's survival. For instance, embarked is which port the passenger left from, not impacting their survival rate. We then dropped Cabin; the cabin is what room the passenger stayed in on the trip. Although, hypothetically speaking, the cabin a passenger stayed in could have impacted their survival rate, I.e., a bottom deck cabin possibly has lower odds of survival than a top deck cabin, we could not use this in training our MLP.

When visualizing the data, we plotted several different graphs. In the graphs, 0 represents did not survive, and one represents that they did.

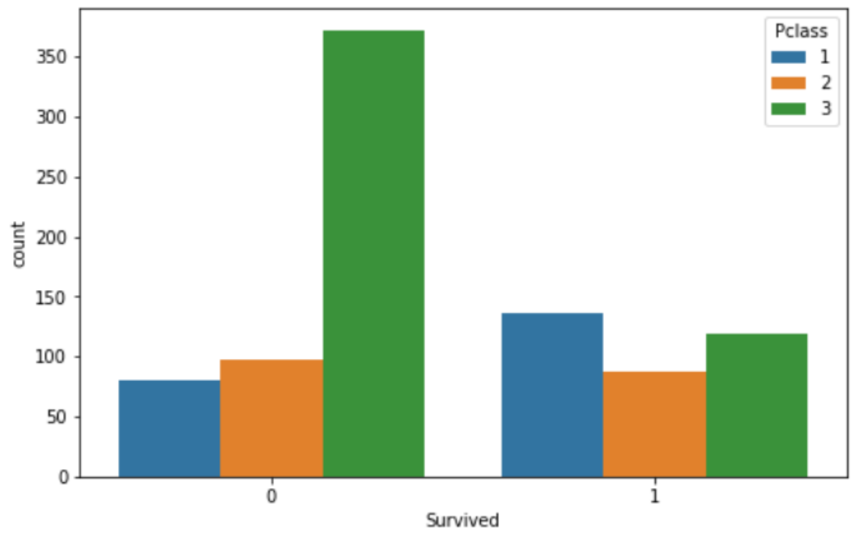
The first one we plotted was the survival of the whole dataset.



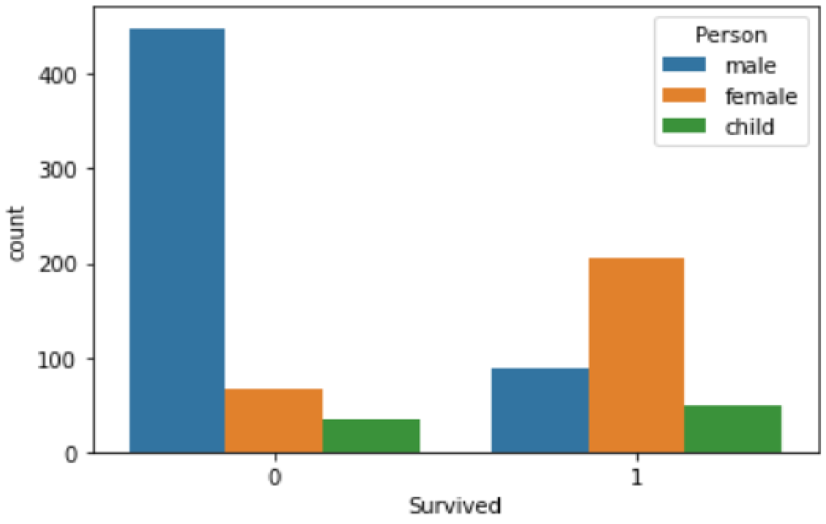
The second one we plotted was the survival rate among men and women.



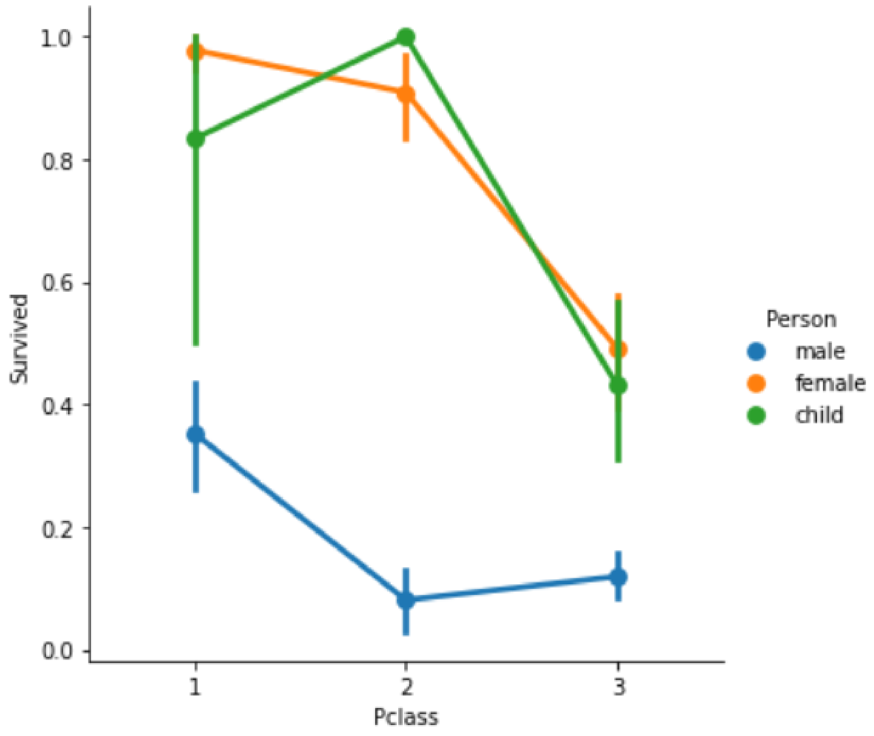
The next one is the survival rate among the passenger classes.



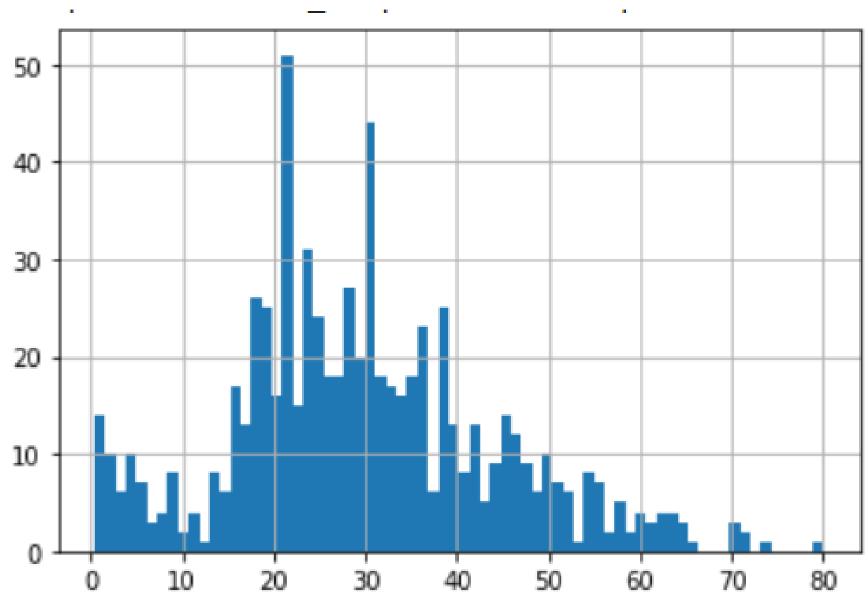
This was followed by plotting the survival rate of men, women, and children.



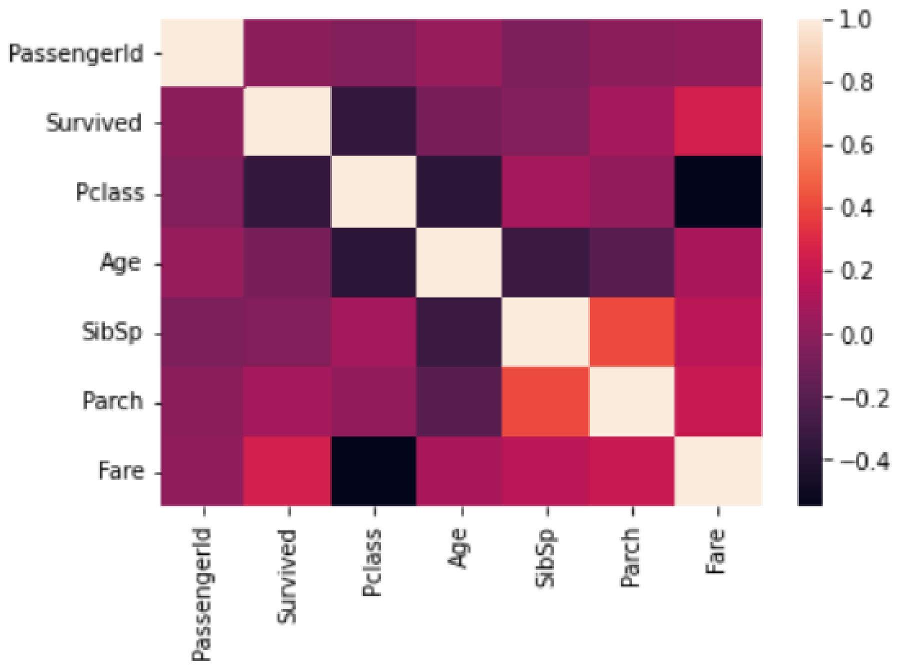
We then plotted the survival rate of the passenger class among men, women, and children.



After this, we plotted the ages of the dataset using a histogram.



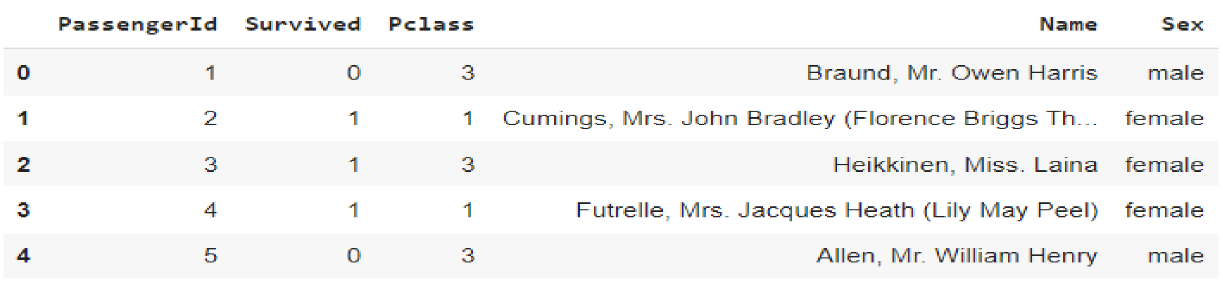
Finally, we plotted the correlation among the data.



Preprocessing and normalization of the data

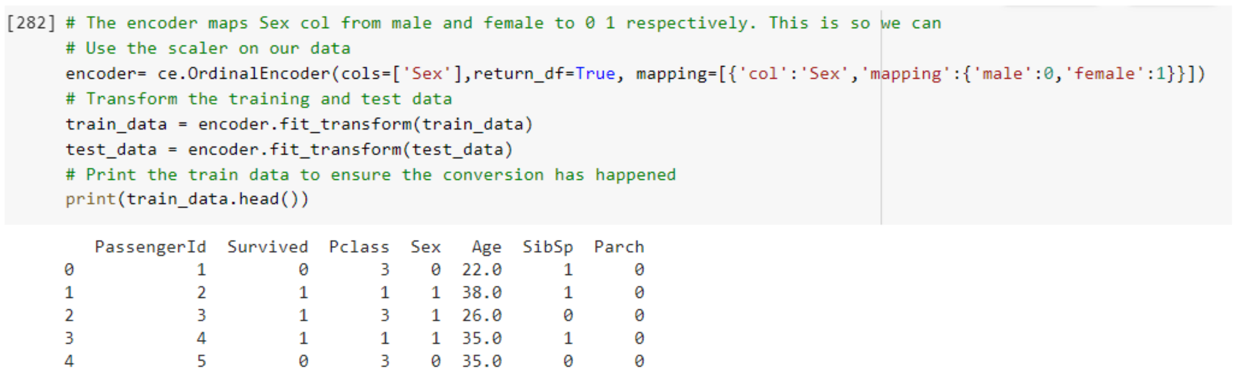
In data preprocessing, we examine the data for any errors and additional features that we could add to analyze the data. First, we explored how many rows have Not a Number (NaN) values in them. We found that there were 891 rows and that 179 rows were missing a value for an age so only 712 rows with complete data. To fix this problem we could drop all 179 rows and use the remaining 712 rows as our dataset, but this would mean less data for the MLP to use. So instead, we got the average age of all 712 rows that had a value for age. We then filled in the 179 rows left with the average age of the dataset, this meant that we had 891 rows with complete values. This prevented us in reducing the dataset by getting rid of the rows with NaN values, a larger dataset is better for training the model.

We then preprocessed our data to transform “Male” and “Female to 0 and 1. This will allow us to use the values than when training our data.



	PassengerId	Survived	Pclass	Name	Sex
0	1	0	3	Braund, Mr. Owen Harris	male
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female
2	3	1	3	Heikkinen, Miss. Laina	female
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female
4	5	0	3	Allen, Mr. William Henry	male

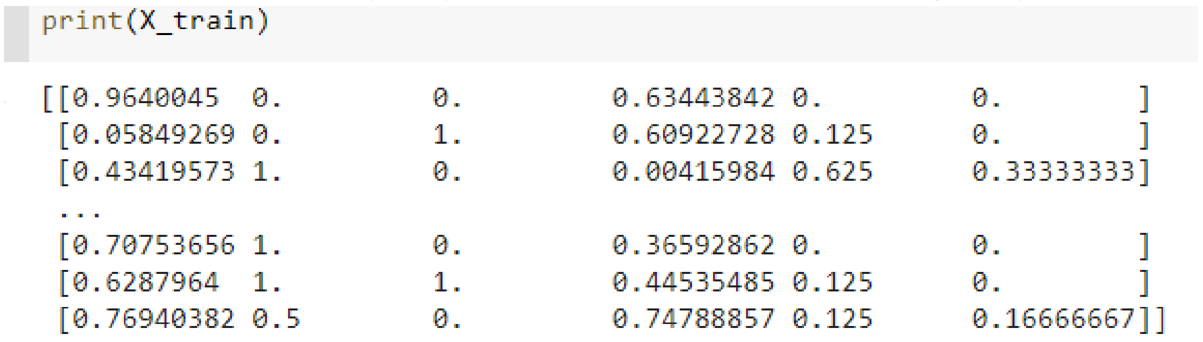
We used the ordinal encoder to transform the data. This was the result after transforming the data.



```
[282] # The encoder maps Sex col from male and female to 0 1 respectively. This is so we can
      # Use the scaler on our data
      encoder= ce.OrdinalEncoder(cols=['Sex'],return_df=True, mapping=[{'col':'Sex','mapping':{'male':0,'female':1}}])
      # Transform the training and test data
      train_data = encoder.fit_transform(train_data)
      test_data = encoder.fit_transform(test_data)
      # Print the train data to ensure the conversion has happened
      print(train_data.head())
```

```
   PassengerId  Survived  Pclass  Sex   Age  SibSp  Parch
0            1         0       3    0  22.0      1      0
1            2         1       1    1  38.0      1      0
2            3         1       3    1  26.0      0      0
3            4         1       1    1  35.0      1      0
4            5         0       3    0  35.0      0      0
```

The next step was to normalize our data. Normalization is a scaling technique in which values are rescaled so that they end up ranging between 0-1. It is also known as Min-Max scaling. In our case, we used the sklearn preprocessing library and incorporated a MinMax Scaler. This scaler was set to a range of 0-1 then it was fitted to the training data. The data is then transformed for the training process. After scaling the data, the data was off values between 0 and 1 as noted above. Finally, we print the dataset out to see the scaling take place



```
print(X_train)
```

```
[[0.9640045  0.         0.         0.63443842 0.         0.        ]
 [0.05849269 0.         1.         0.60922728 0.125      0.        ]
 [0.43419573 1.         0.         0.00415984 0.625      0.33333333]
 ...
 [0.70753656 1.         0.         0.36592862 0.         0.        ]
 [0.6287964  1.         1.         0.44535485 0.125      0.        ]
 [0.76940382 0.5        0.         0.74788857 0.125      0.16666667]]
```

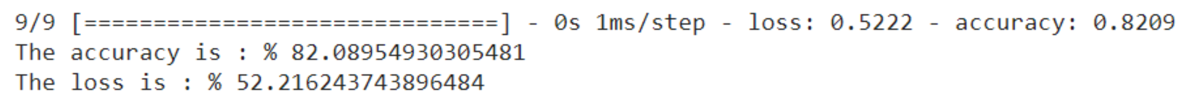
The network structure and other hyperparameters

We created a sequential or also known as a feed-forward model. The model consisted of 3 layers. The first layer had 6 nodes, random weights using the he_normal initializer, and relu as the activation function. The second layer had 12 nodes and an activation function of relu again. For the final layer, it had one node and sigmoid as the activation function. We used mean squared logarithmic error and the ‘adam’ optimizer with a learning rate of 0.01 for the model also. The model is 30 in epochs and has a batch size of 10. We also used the early stop method to prevent under/over fitting.

The Cost/Loss/Error/objective functions

Binary Crossentropy

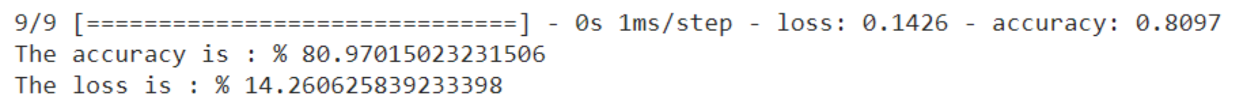
We first started using binary_crossentropy as the loss function. For binary classification problems, the default loss function is cross-entropy. It is designed for binary classification with target values in the range of 0 to 1. When we used the binary_crossentropy loss function, we received an extremely high loss percentage. We received a loss function of around 52%.



```
9/9 [==============================] - 0s 1ms/step - loss: 0.5222 - accuracy: 0.8209
The accuracy is : % 82.08954930305481
The loss is : % 52.216243743896484
```

Mean Squared Error

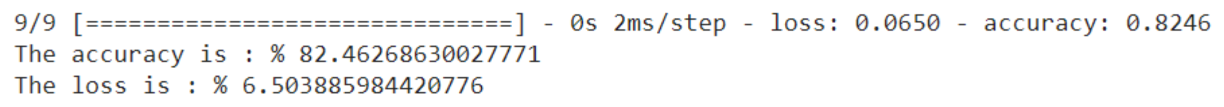
The next loss function we tried was the Mean_squarred_error or MSE. The MSE loss, or Mean Squared Error, is the default loss for regression situations. The average of the squared variance between the expected and actual values is used to determine mean squared error. Regardless of the sign in the predicted and actual numbers, the result is always positive, and a perfect value is 0.0. Because of the squaring, larger errors result in more errors than smaller errors, implying that the model is penalized for making larger errors. Using MSE we got a much lower loss function of around 14%.



```
9/9 [==============================] - 0s 1ms/step - loss: 0.1426 - accuracy: 0.8097
The accuracy is : % 80.97015023231506
The loss is : % 14.260625839233398
```

Mean Squared Logarithmic Error

Lastly, there may be regression issues where the target value has a range of values, and you may not want to penalize a model as harshly as mean squared error when predicting a huge number. Instead, you can determine the mean squared error by first calculating the natural logarithm of each of the anticipated numbers. This is known as the MSLE (Mean Squared Logarithmic Error Loss). Using MSLE, we managed to get our loss function even further down to around 6%.



```
9/9 [==============================] - 0s 2ms/step - loss: 0.0650 - accuracy: 0.8246
The accuracy is : % 82.46268630027771
The loss is : % 6.503885984420776
```

This is the reason we decided to opt with MSLE as our chosen loss function.

The optimizer

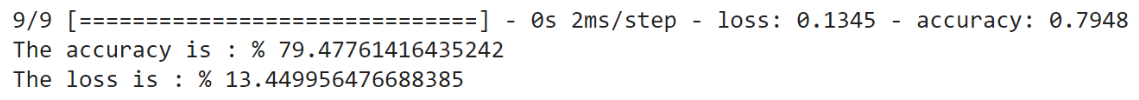
SGD

During our first run we used a Stochastic gradient descent optimizer. This is a basic optimizer algorithm. The SGD optimizer has various advantages, such as frequent updates to the model's parameters, finding new minima's, and requiring less memory than traditional gradient descent.

However, SGD suffers from the issue of needing a lot of memory to carry out its gradient calculation. There is also other various disadvantages we became aware of researching about SGD. These include High variance in model parameters, it may keep executing even after achieving global minima. To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

After implementing it into our model we found the results were ok, but we wanted to experiment with other optimizers to see what effect it would have on the accuracy of our model.

Results from SGD:

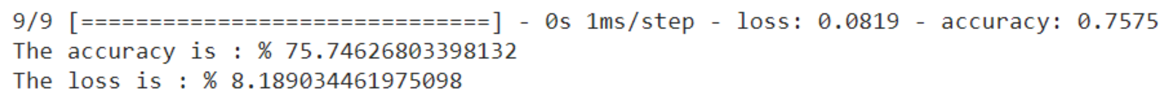


```
9/9 [==============================] - 0s 2ms/step - loss: 0.1345 - accuracy: 0.7948
The accuracy is : % 79.47761416435242
The loss is : % 13.449956476688385
```

Adagrad

We further explored the “Adagrad” optimizer. The Adagrad optimizer changes the learning rate constantly. This means that Adagrad does not need to tune the learning rate manually and Adagrad has the ability to train on sparse data. A disadvantage of Adagrad is, that it is Computationally expensive as there is a need to calculate the second-order derivative. We received lower accuracy that SGD and Adam using Adagrad.

Results from Adagrad:

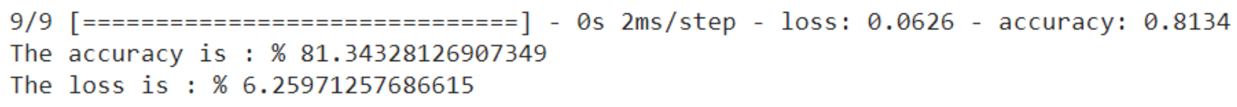


```
9/9 [==============================] - 0s 1ms/step - loss: 0.0819 - accuracy: 0.7575
The accuracy is : % 75.74626803398132
The loss is : % 8.189034461975098
```

Adam

We used the “Adam” optimizer in our model. The Adam optimizer is a replacement optimization for stochastic gradient decent. Adam has two main advantages over using stochastic gradient.
It is an Adaptive Gradient Algorithm that maintains a pre parameter learning rate. This improves performance on problems with sparse gradients (Network is not receiving enough signals to tune its weights). Adam, however, can be computationally demanding.

Results from Adam:

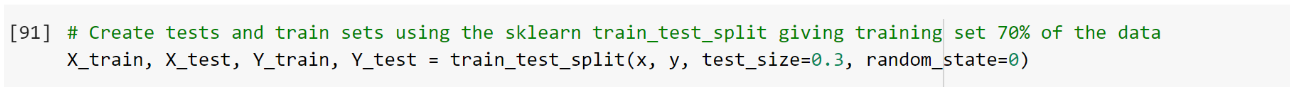


```
9/9 [==============================] - 0s 2ms/step - loss: 0.0626 - accuracy: 0.8134
The accuracy is : % 81.34328126907349
The loss is : % 6.25971257686615
```

We decided to go with the Adam optimizer as we got the best accuracy and lowest loss function.

Cross fold validation

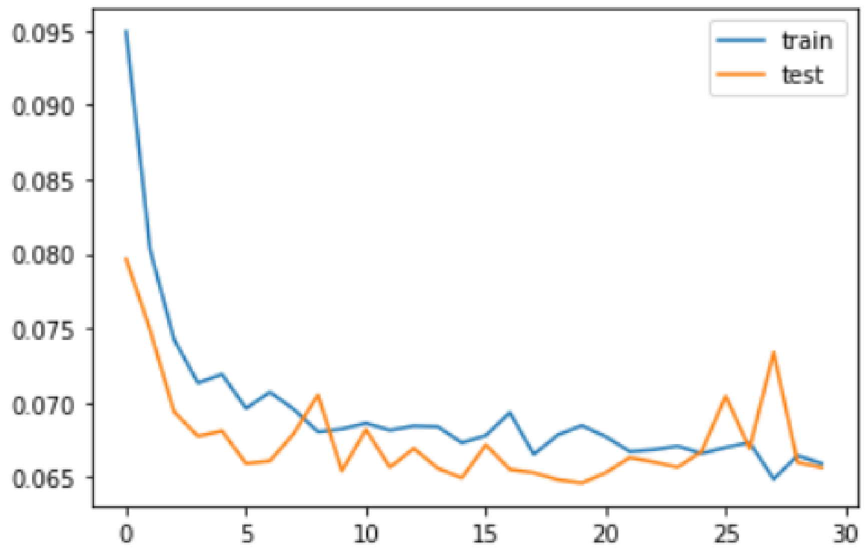
We used 2-fold cross validation for our data. The Test_Train_split approach randomly splits the data into training and data sets. We went with this approach as we believed we had enough data to have a lower bias. A Test_Train_split approach can have a negative effect on smaller data sets as it will cause a higher bias. In our code we went with a 70:20 split which is a recommended spit.

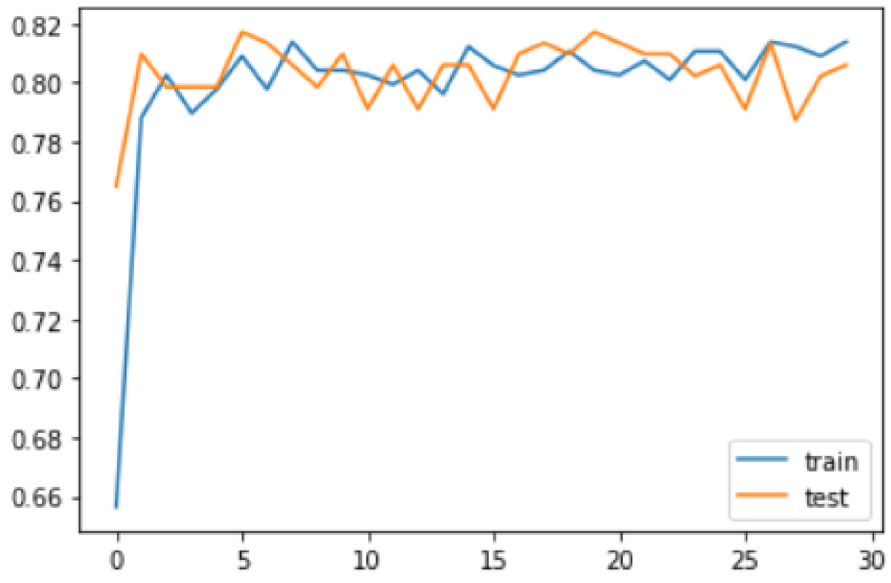


```
[91] # Create tests and train sets using the sklearn train_test_split giving training set 70% of the data
     X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size=0.3, random_state=0)
```

Results

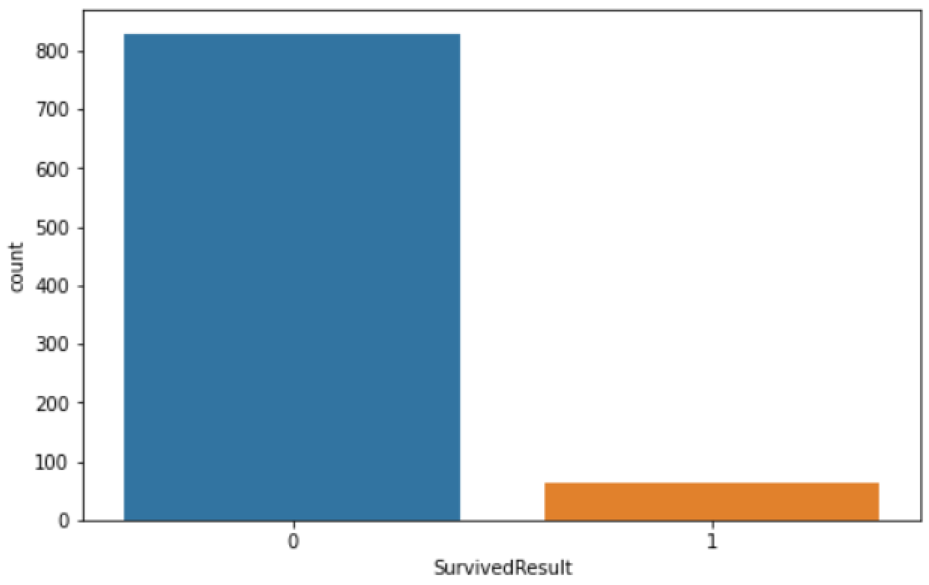
The model had an accuracy of around 80%. Below is the loss and accuracy of the model for the train and test data.



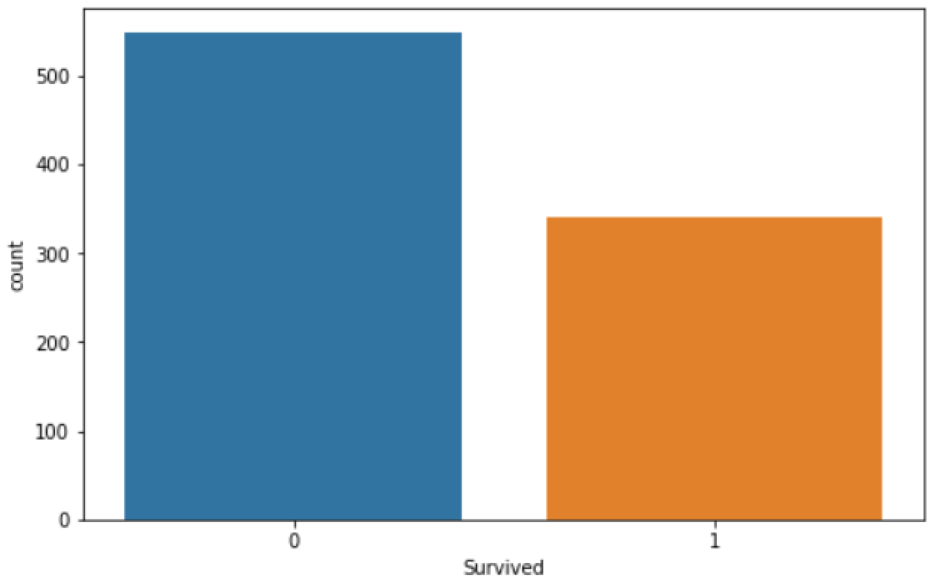


The prediction was not all that accurate. The actual survival rate of the data was around 300 to survive and just over 550 not to survive, our model predicted only around 72 to survive and around 819 not to survive. In the notebook there are graphs and plots to show the results of the predicted data which you can compare against the first graphs on the actual dataset.

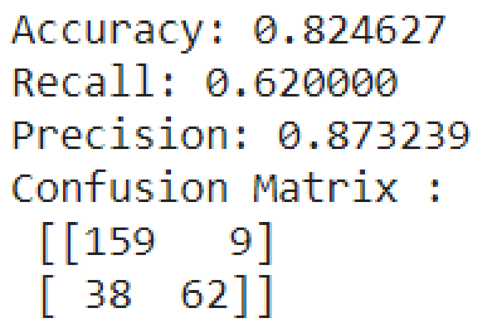
Predicted data:



Actual data:



We also got the accuracy, recall, precision, and confusion matrix.



```
Accuracy: 0.824627
Recall: 0.620000
Precision: 0.873239
Confusion Matrix :
 [[159   9]
 [ 38  62]]
```

Evaluation

Overfitting/Underfitting

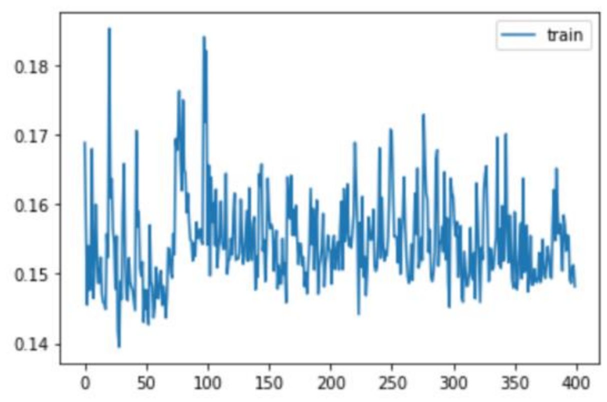
To ensure there was no overfitting in our model we fitted it with an EarlyStopper function. EarlyStopper stops the model from running if the epoch it's on drops in performance. This is because we wanted to give our model a chance to learn and a drop in performance is inevitable in the early stages. To resolve this issue, we set the "patience" parameter to 10. This gives the model 10 chances to drop in performance any more drops would cause the model to stop. This ensures that there is no overfitting as if there are more than 10 drops in performance, it will stop running the epochs no matter how many is given. From reading the graphs of performance and loss, we found that 30 epochs were a substantial number to train our data with no underfitting.

Impact of varying hyper parameters

- **Learning rate:** We found by increasing the learning rate we got interesting results. Increasing the learning rate in the optimizer decreased the model's performance and accuracy. We found the model worked best at a learning rate of 0.01. Below is an example of the varying learning rates.

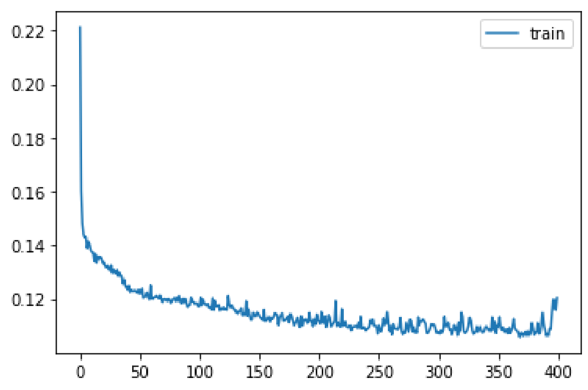
Here the learning rate is 0.1. As we can see from the graph the loss varies significantly among the epochs.

```
opt = Adam(lr=0.1)
```



```
9/9 [==============================] - 0s 2ms/step - loss: 0.1664 - accuracy: 0.7985
The accuracy is : % 79.85074520111084
The loss is : % 16.64482355117798
```

Lowering this to a rate of 0.01 on the other hand accuracy improved, and loss decreased. This significantly changed the loss graph through each epoch.



```
9/9 [==============================] - 0s 2ms/step - loss: 0.1517 - accuracy: 0.8022
The accuracy is : % 80.22388219833374
The loss is : % 15.165236592292786
```

We found changing the learning rate up or down had large difference in how the model predicted the data.

- **Activation Functions:** We used the relu activation function for the first two layers and sigmoid for the output layer. We found to get the best results using these as the activation functions. We used different combinations of relu, softmax, sigmoid, tanh, and selu and as said above, two relus and one sigmoid got the best results for us. We used sigmoid as the final layer activation function as sigmoid always returns a value between 0 and 1. This seemed appropriate as our final prediction should be a number between 0 and 1.

- **Weight Initialization:** For the MLP we used the he_normal weight initializer as the first layers weights. The he_normal works better with the relu activation function that we used on the first layer. We also tried the One and Zero weight initializers. The One weight initializer reduced the accuracy of the model by around 2% and Zero returned the result as linear, all the predictions were 0. For this reason, we decided to use he_normal as the weight initializer.

Experiments

- Firstly, we experimented with dropping columns in the dataset when training our model. We dropped Age but that gave us linear results, so we decided that we need the Age column to train the model.
- The second experiment we done was playing around with all the hyperparameters of the MLP. We found that the current hyperparameters gave us the best accuracy and least amount of loss.
- Something we learned which we found interesting was in the labs, we only ever saw the accuracy printed out. We were able to get the predictions, plot them, and see them in the table compared to the actual survival rate for this project. It was interesting as we got to visualize the MLP in work. We would consider this a new insight into neural computing.
- We also found that varying the batch size of the MLP decreased and increased the processing time of the MLP. A larger batch size spreads the work across more threads, doing the work concurrently.