

CT5170 - Principles of Machine Learning

Assignment 1

Daniel Verdejo - 22240224

1: Have a look at the given data, understand the problem based on the dependent variable and select a machine learning category that can solve the task/problem. Briefly explain why do you think it is the correct ML category for this problem?

Answer:

The data provided is a subset of the Iris dataset which typically includes 3 species. We have both a training and test dataset for our neural network to consume.

To solve the problem we will use a supervised learning classification task as we are provided 80 samples in total in a training dataset.

Classification will enable us to organise our data in categorical sense based off predictions made about each sample the model is given.

We first train the model with a dataset of known data, consisting of 2 classes with 40 samples per class (as seen below), each sample containing 4 attributes and the target label. The attributes are the sepal width and length (in cm), and the petal width and length (in cm).

Once trained we will then need to feed it new data (data it has not yet seen) to measure its performance.

Lets begin...

In [107..

```
import csv
import numpy as np
import keras as kr
import seaborn as sb
import pandas as pd

training_samples = list(csv.reader(open("./plant-data/plant-train.csv")))[1:]
names = list(csv.reader(open("./plant-data/plant-train.csv")))[:1]
testing_samples = list(csv.reader(open("./plant-data/plant-test.csv")))[1:]

training_attributes = np.array(training_samples)[:,0:4].astype(np.float32)
training_classes = np.array(training_samples)[:,4]
uniq_training_classes, output_indices, training_counts = np.unique(training_classes,
outputs_categories = kr.utils.to_categorical(output_indices).astype(np.float32)

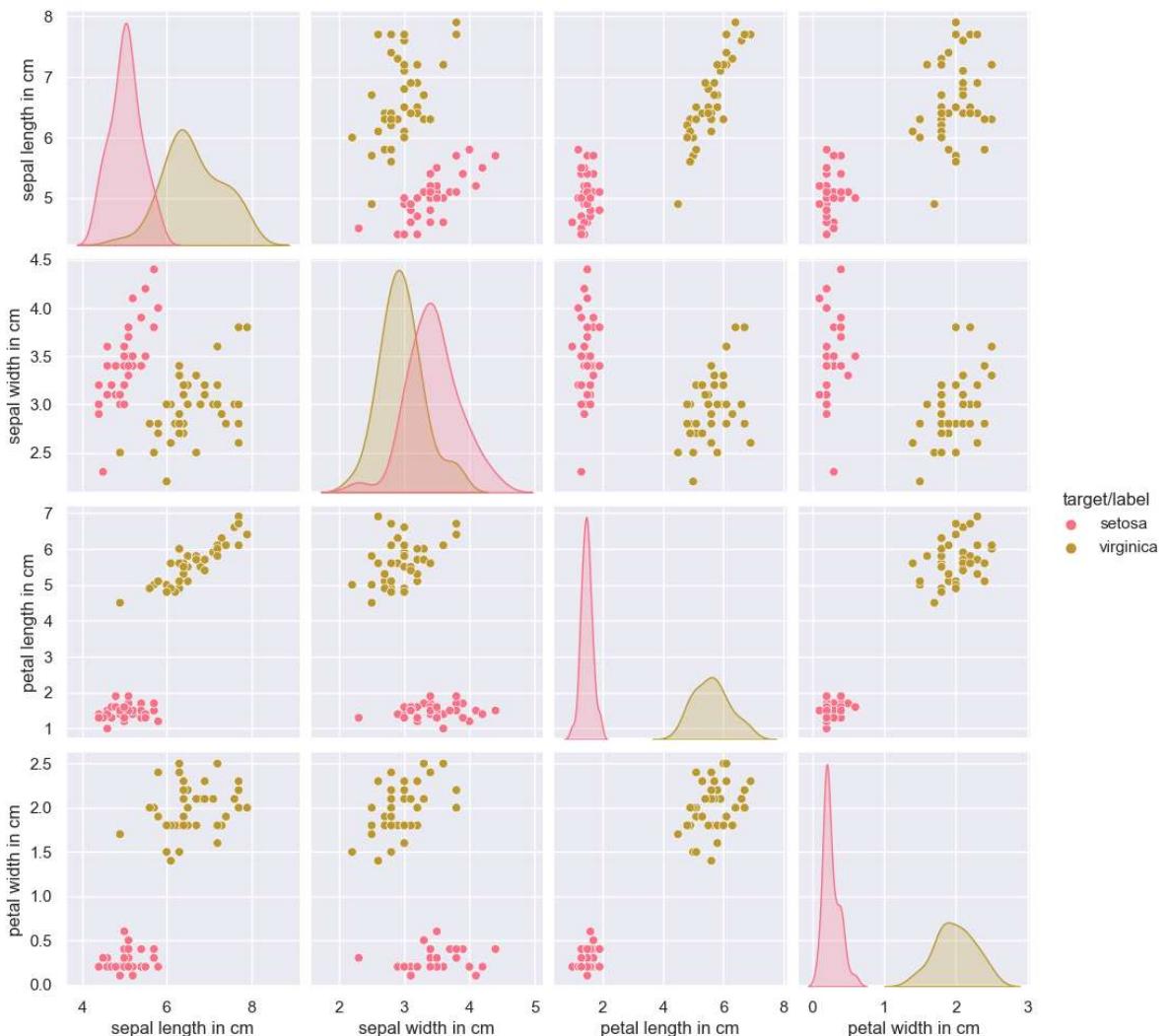
testing_attributes = np.array(testing_samples)[:,0:4].astype(np.float32)
testing_classes = np.array(testing_samples)[:,4]
uniq_test_classes, output_test_ind, test_counts = np.unique(testing_classes, return_index=True)
output_test_categories = kr.utils.to_categorical(output_test_ind).astype(np.float32)
```

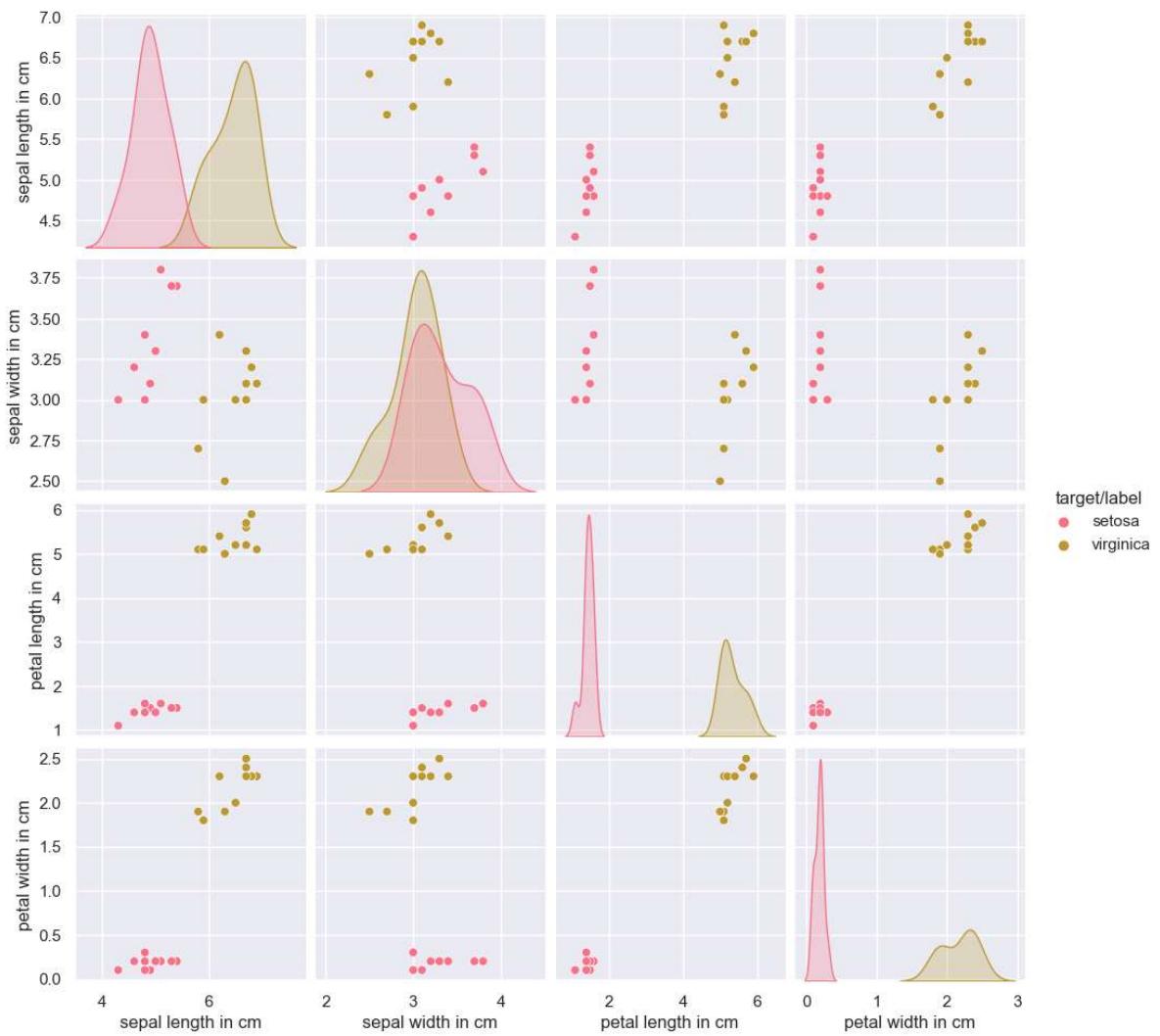
```
training_df = pd.read_csv("./plant-data/plant-train.csv")

sb.set_palette("husl")
sb.pairplot(training_df, hue="target/label")

test_df = pd.read_csv("./plant-data/plant-test.csv")
sb.pairplot(test_df, hue="target/label")
```

Out[107]:





2: Explore and report the data and its distribution among training and testing data. Can we call it imbalanced dataset, explain your answer (yes/no) briefly?

Answer:

No, we cannot say the provided dataset is imbalanced. As we can see above, both the training and testing datasets show a 50:50 ratio for each class. The dataset is perfectly balanced with no specific class showing a majority or minority in either the training or testing datasets. Also, we can see good distribution and clear deliniation between the 2 classes on both datasets.

3: Research and write down about open-source machine learning package that are freely available, and select one that you think will be good and easy for this task. Your report should include a short overview of the main features of the package you have chosen.

Answer:

Some of the most popular machine learing libraries for python are as follows:

TensorFlow - Developed by Google, written in Python and C++, it offers a free and open-source library used for artificial intelligence and machine learning applications. The library offers both a high-level and low-level API to users to enable creating machine leanring models which can be run on many different host devices including, cloud, desktop, mobile

or even on edge devices. It offers tools to validate and transform large datasets, and tools to discover and remove bias in data to improve outcomes on models. Tensorflow offers in depth API documentation and learning materials for users of all experience levels.

Keras - Is a machine learning API written purely in Python, which runs on top of the TensorFlow platform. It offers an easy to use high-level API which abstracts some of the more complex functionality of TensorFlow. Keras offers a quick and easy way to get up and running for smaller projects.

It was built with simplicity in mind to promote fast prototyping via user friendliness, modularity and extensibility. Supports convolutional and recurrent networks, and can utilise either the host CPU or GPU for computational work.

PyTorch - Developed by the Linux Foundation and Meta AI (a branch of Meta, formerly Facebook) was released in September 2016. The library is written in Python, CUDA and C++, and offers a machine learning framework based on the Torch library. It can be used to develop machine learning applications and also offers capabilities for the creation of REST API endpoint to ease application integration. It only offers low level apis so it can be more difficult to use than Keras.

The machine learning library that I will be using here is [Keras](#). Its ease of use makes carrying out tasks like the one we are tackling today quick and easy.

Using the Keras library over something like the TensorFlow or PyTorch libraries will reduce the amount of effort on my part to build the neural network, and should also be fairly easy to read and understand for you, the reader. As this dataset is quite small and relatively trivial using one of the other options, while they may operate faster, would add an unnecessary level of complexity as we should not need to dig into the lower level apis, or spend a lot of time debugging our neural network.

4: In order to use the dataset (Plant-dataset) supplied below, you might need to do some work to prepare it for input into the ML package, depending on the ML category requirements. Document any data preparation (e.g. normalisation) steps in your report.

Some preparation of the dataset is required, we first got our attributes above like so:

```
attributes = np.array(training_samples)[:, :4].astype(np.float)
```

 these our inputs.

Next we got our classes: `training_classes = np.array(training_samples)[:, 4]` these will be our outputs.

Above we used `unq_training_classes, inverse, training_counts = np.unique(training_classes, return_counts=True, return_inverse=True)` for the training dataset to view how the data is distributed among the 2 class in both datasets. We also got the indices of the unique array. This consists of 0 and 1s which points us to the index for each output. As we can see below.

```
In [108...]: print(output_indices)
```

5: From the ML package, select two different algorithms from the category you selected and apply to the dataset. In your report, include a clear description of both algorithms. Ensure that you acknowledge all of your sources of information. Report the results with and without normalisation of the data.

For our neural network we will use the Sigmoid and Relu activation functions both with softmax as an output layer.

1 - The Sigmoid Activation function

Formula:

$$f(x) = 1/(1 + e^{-x})$$

The sigmoid function itself exists between 0 and 1 and takes an S shape curve. Since we are attempting to predict between 0 and 1 as we have just seen above from our `output_indices` list, the sigmoid activation function is the best fit here.

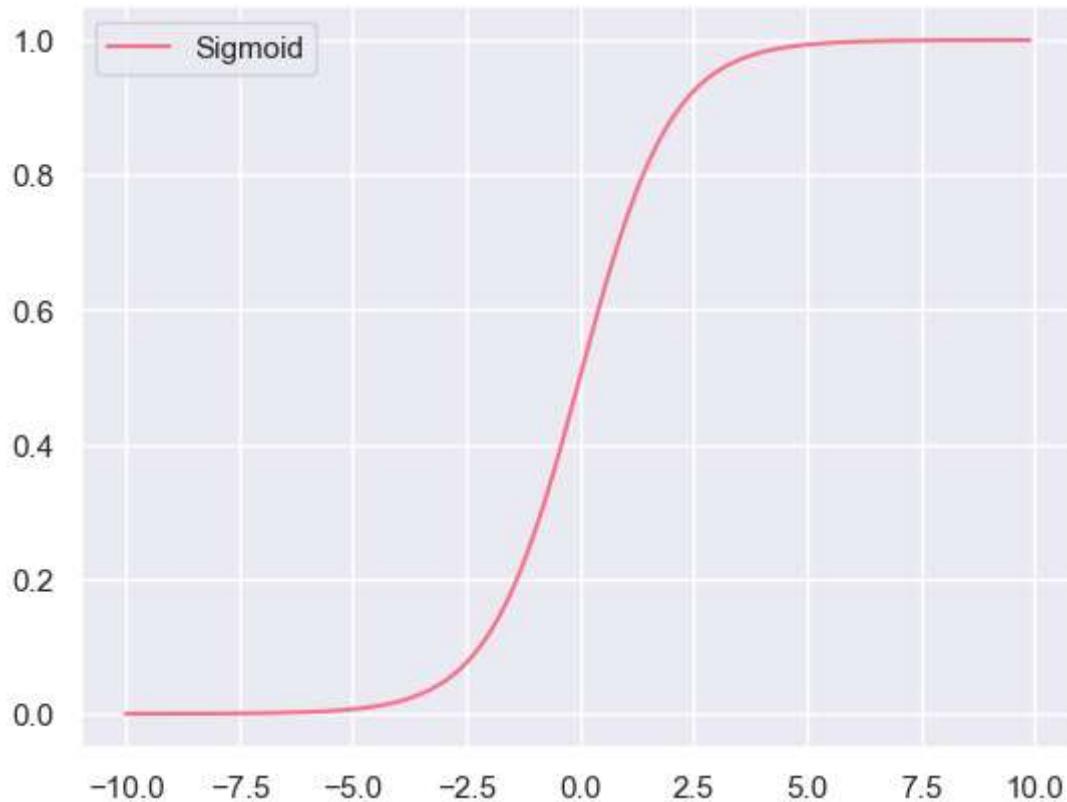
This function is used in logistic regression to classify data. It takes a numeric input value and maps it to a probability between 0 and 1 like so: If the value is above 0.5 its 1, otherwise if its below 0.5 it 0 (rounding up or down to the nearest whole number).

Reference: O'Reilly - Introduction to Machine Learning with R: Scott V. Berger, Chapter 2: Supervised and Unsupervised Machine Learning - Neural Networks, Pages 35-36.

Reference: O'Reilly - Introduction to Machine Learning with R: Scott V. Berger, Chapter 4: Regression in a Nutshell - The Sigmoid Function, Page 94.

We can plot out the Sigmoid activation function using matplotlib and the equation:

```
In [109...]:  
from matplotlib.pyplot import plot, show, legend  
from math import exp  
  
xAxis = np.arange(-10., 10., .1)  
sigmoid = [ 1/(1+exp(-item)) for item in xAxis]  
  
plot(xAxis, sigmoid)  
legend(["Sigmoid"])  
show()
```



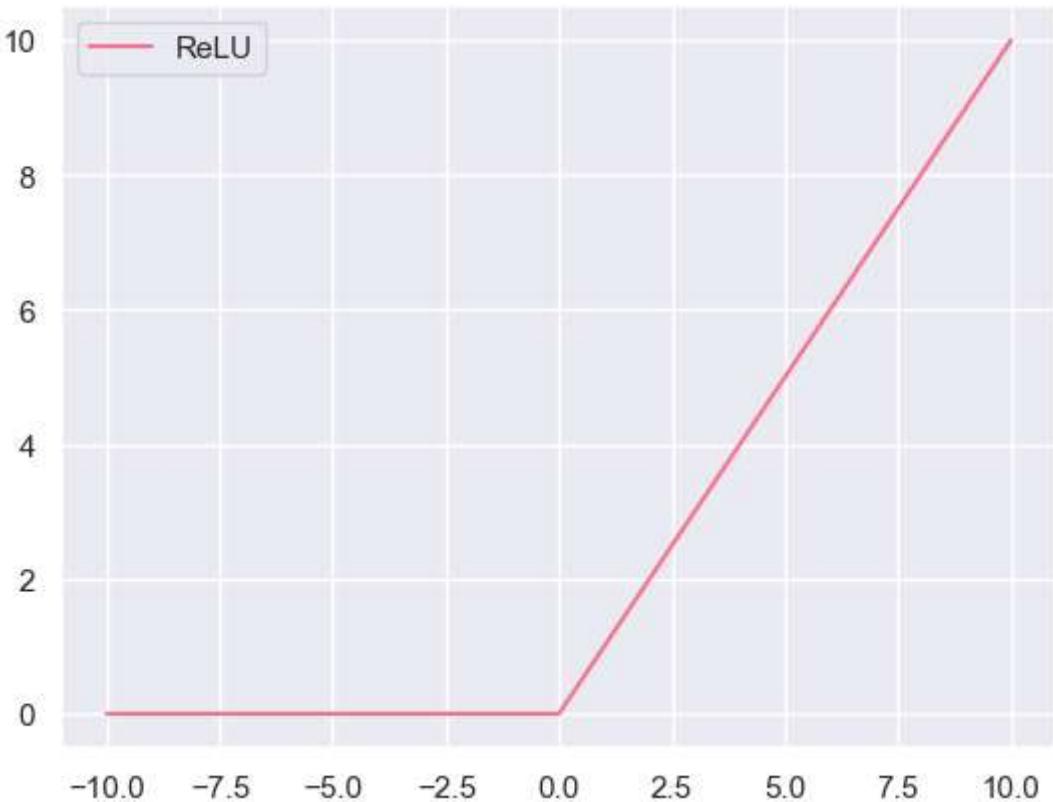
2- Rectified Linear (ReLU) activation function

Formula:

$$f(x) = \max(0, x)$$

With the ReLU activation function, it will return 0 for inputs below 0, its relationship with the input is linear beyond a certain threshold, if it is above this threshold it directly returns the input. As a result of the gradient being 0 or a constant, controlling the vanishing/exploding gradient issue is possible.

```
In [110]: x = np.linspace(-10,10,1000)
plot(x, np.maximum(0, x))
legend(["ReLU"])
show()
```



We will also be using the following activation function on our output layer **Softmax output layer**

$$\text{Formula: } f(x)_i = e^{x_i} / \sum_{j=1}^k e^{x_j}$$

We want our output to be a probability distribution over a collection of classes. In this example we have 2 mutually exclusive classes `Setosa` and `Virginica`. Although it is not likely that the neural network will be able to predict the output with 100% accuracy, when we use a probability distribution we have a better idea of how accurate our predictions are. The output of a node in a softmax layer is dependant on all of the other nodes outputs in its layer as we require the sum of all outputs.

[Reference: O'Reilly - Fundamentals of Deep Learning: Nikhil Buduma, Chapter 1: The Neural Network - Softmax Output Layer, Page 15.](#)

6: Train and test your chosen algorithms using the training set provided in plant-train.csv. You should then test your trained models using the test set provided in plant-test.csv. Report on the results with appropriate performance metric e.g. accuracy that you consider best for each model on the training set and on the test set. Also include details of the classification models constructed – these may include graphics if appropriate.

We're going to create a neural network which will take 4 inputs (our 4 attributes: sepal width, sepal length, petal width, petal length). Then pass it through each layer, these layers contain nodes which are computational points. We add an activation function to each layer (sigmoid or ReLU in this case) to be able to make these calculations. Finally they will be passed to the output layer in this case as a 0 (setosa) or 1 (virginica).

Our neural network will resemble something like so:



We will compile the the data using the Adam optimizer, and binary cross entropy, for binary-class classification where each sample belongs to a single class.

In [111...]

```
# We can Create a sequential model passing no args to the constructor
model_a = kr.models.Sequential()

# And add our initial Layer after the fact, with an input for each of our attributes
model_a.add(kr.layers.Dense(16, input_shape=(4,), activation="sigmoid")) # We're using sigmoid here because we have two classes

# And build our model Layer by Layer
model_a.add(kr.layers.Dense(2, activation="softmax"))

# Alternatively we can add all the layer information into the constructor and be done with it
model_b = kr.models.Sequential(layers=[
    kr.layers.Dense(16, input_shape=(4,), activation="sigmoid"),
    kr.layers.Dense(2, activation="softmax")
])

# configure the models for training
# use the adam optimizer and binary cross entropy as the loss function our metric
model_a.compile(kr.optimizers.Adam(learning_rate=0.05), loss="binary_crossentropy")
model_a.summary()

model_b.compile(kr.optimizers.Adam(learning_rate=0.05), loss="binary_crossentropy")
model_b.summary()
```

Model: "sequential_18"

Layer (type)	Output Shape	Param #
<hr/>		
dense_36 (Dense)	(None, 16)	80
dense_37 (Dense)	(None, 2)	34
<hr/>		
Total params: 114		
Trainable params: 114		
Non-trainable params: 0		

Model: "sequential_19"

Layer (type)	Output Shape	Param #
<hr/>		
dense_38 (Dense)	(None, 16)	80
dense_39 (Dense)	(None, 2)	34
<hr/>		
Total params: 114		
Trainable params: 114		
Non-trainable params: 0		

We use an optimizer with the aim to maximize the performance of our model by making iterative adjustments to its parameters to minimize error. This means we will avoid having to use a gradient descent strategy to minimize error.

With gradient descent we would evaluate the gradient at a given starting position, find and travel in the direction of the steepest descent, from this new position we would repeat this

process, continuing until we get to a point of minimum error.

Thankfully, we do not have to worry about this when we use the Adam optimizer. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure in order to update network weights based on training data. The Adam algorithm calculates the exponentially weighted moving average of the gradient, and it estimates a vector in of the first moment of the gradient. In the second moments of the gradient another vector is estimated and the moving average of the historical gradients. The estimation of the vectors initial values to zero result in a bias relative to the real moments, thus a correction factor for both vector estimations is derived. This is Adam's corrective measure against the zero initialization bias.

Reference: O'Reilly - Fundamentals of Deep Learning: SNikhil Buduma, Chapter 2: Training Feed-Forward Neural Networks - Gradient Descent, Pages 19 - 20.

Reference: O'Reilly - Fundamentals of Deep Learning: SNikhil Buduma, Chapter 4: Beyond Gradient Descent - Adam-Combining momentum and RMSProp, Page 81.

```
In [112...]: #fit the model using our training data  
model_a.fit(training_attributes, outputs_categories, epochs=50, batch_size=1, verbose=1)  
  
#evaluate the model using the test data set  
loss, accuracy = model_a.evaluate(testing_attributes, output_test_categories, verbose=1)  
  
print("Model using the Sigmoid activation function:\n\nloss: %6.4f\nAccuracy: %6.4f" % (loss, accuracy))  
  
1/1 [=====] - 0s 108ms/step - loss: 2.8350e-05 - accuracy: 1.0000  
Model using the Sigmoid activation function:  
  
loss: 0.0000  
Accuracy: 1.0000
```

```
In [113...]: #fit the model using our training data  
model_b.fit(training_attributes, outputs_categories, epochs=50, batch_size=1, verbose=1)  
  
#evaluate the model using the test data set  
loss, accuracy = model_b.evaluate(testing_attributes, output_test_categories, verbose=1)  
  
print("Model using the ReLU activation function:\n\nloss: %6.4f\nAccuracy: %6.4f" % (loss, accuracy))  
  
1/1 [=====] - 0s 103ms/step - loss: 4.1993e-05 - accuracy: 1.0000  
Model using the ReLU activation function:  
  
loss: 0.0000  
Accuracy: 1.0000
```

```
In [114...]: a_pred = model_a.predict(testing_attributes)  
b_pred = model_b.predict(testing_attributes)  
  
a = np.argmax(a_pred, axis=1)  
b = np.argmax(b_pred, axis=1)  
l = np.argmax(output_test_categories, axis=1)  
  
print(a)  
print(b)  
print(l)
```

```
# print("That means it's a %s" % uniq_test_classes[prediction.astype(np.bool)])
1/1 [=====] - 0s 46ms/step
1/1 [=====] - 0s 55ms/step
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

As we can see above the predictions are spot on, but lets normalize the data and re-run the training and tests, to see what the difference is when we do this.

```
In [115...]: from sklearn import preprocessing

norm_train = preprocessing.normalize(training_attributes)
norm_test = preprocessing.normalize(testing_attributes)

model_a.fit(norm_train, outputs_categories, epochs=50, batch_size=1, verbose=0)

loss, accuracy = model_a.evaluate(norm_test, output_test_categories, verbose=1)

print("Model using the Sigmoid activation function with normalized data:\n\nloss: ")
1/1 [=====] - 0s 16ms/step - loss: 1.7662e-05 - accuracy: 1.0000
Model using the Sigmoid activation function with normalized data:

loss: 0.0000
Accuracy: 1.0000
```

```
In [116...]: model_b.fit(norm_train, outputs_categories, epochs=50, batch_size=1, verbose=0)

loss, accuracy = model_b.evaluate(norm_test, output_test_categories, verbose=1)

print("Model using the ReLU activation function with normalized data:\n\nloss: %.6f")
1/1 [=====] - 0s 19ms/step - loss: 1.4616e-05 - accuracy: 1.0000
Model using the ReLU activation function with normalized data:

loss: 0.0000
Accuracy: 1.0000
```

```
In [117...]: a_pred_norm = model_a.predict(norm_test)
b_pred_norm = model_b.predict(norm_test)

a = np.argmax(a_pred, axis=1)
b = np.argmax(b_pred, axis=1)

print(a)
print(b)
print(l)
```

```
1/1 [=====] - 0s 19ms/step
1/1 [=====] - 0s 19ms/step
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
```

7: Discuss in your report whether the two models give very similar or significantly different results, and why.

The result of the 2 models is near identical with the only real differences visible being the loss, when it is not normalized. But what benefit did we see when we normalized the data?

The main benefit to this is the speed.

Example: The `model_a` with non normalized data took ~102ms / step vs ~16ms / step.
There is a clear performance uplift between the 2 results. But why?

The point of data normalization is to get an uplift in performance and stability on our model by transforming the data to be more uniform.

Findings:

Non-normalized data:

The `model_a` (sigmoid activation fn) has lower loss than `model_b` (ReLU)

Normalized data:

Both models were significantly faster than their non-normalized variant. Interestingly enough, `model_a` had a significant increase in loss over the non-normalized run.