Assessment 1

Machine Learning
CS5062

Yassin Dinana
52094480

29th December 2020

1. Introduction:

In this paper, two different machine learning problems are introduced tackling the main supervised learning tasks which are *classification* and *regression*. Supervised learning is the idea of training a machine learning model using two sets of data, called the training set and testing set. The model learns from different inputs that are found in the training set and then it is tested on unknown and new data that are located in the testing set in order to retain the highest possible accuracy percentage [1].

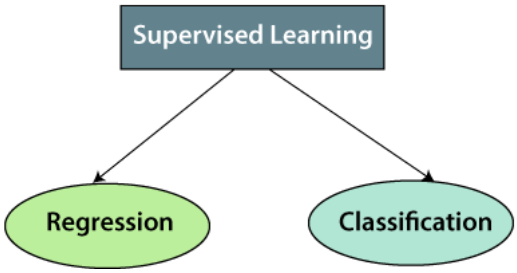

Figure 1: Supervised Learning subfields: Classification and Regression

As it can be seen in figure 1, supervised learning is divided into two different methods which are known as the regression method and the classification method; choosing which method to use is dependent on the given dataset and if the output is discrete or continuous.

In machine learning, *classification* is used to categorize the input dataset into different classes by predicting the class of data points, where in most cases, the input data is represented as (x), and the classes are often represented as the target (y). The classification method operates by taking input variables data and the output is discrete.

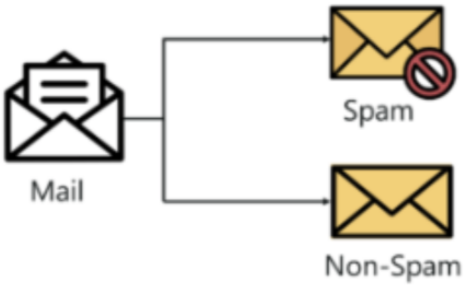

Figure 2: Classification Method Example

Figure 2 above shows an example of the classification method, where the input is a number of mails and after using the classification method, the mails are divided into two classes such as spam or not-spam, which is a discrete output [2].

The second method used in supervised learning is *regression,* which is used to predict continuous values known as (y) based on predictor input variable and data (x). When using regression, the output is continuous meaning it is not classified and is subject to change continuously.

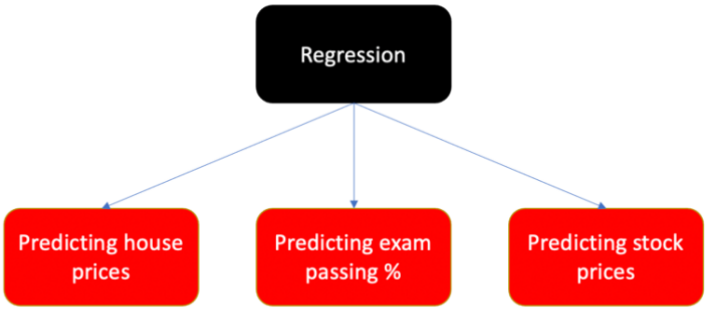

Figure 3: Regression Method Example

Figure 3 above shows different continuous prediction tasks that can be solved using the regression method.

When choosing the regression method, different regression models can be used to predict the output, the main three regression models used to solve the problem in this paper are:

- Linear Regression
- Ridge Regression
- Lasso Regression

Those regression models will be discussed in more details later in this paper.

In the next sections, two different tasks are introduced and solved using both supervised learning methods. The first task is a regression task where the objective is to explore the relationship between the level of a specific cancer antigen and different clinical measures on some patients.

The second task is a classification problem, where a dataset is given with 20000 images for the training set and 5000 images for the testing set. The images are a combination of dogs and cats, and the objective is for the model to be trained to recognize whether the images in the testing set include a dog or a cat and checking the accuracy percentage [1].

2. Task 1:

The first task is a regression problem, where the main objective is to analyse a given data set using three different regression models and choosing the best model after implementing all using python. The given dataset is a text file containing the level of a specific cancer antigen as well as different clinical measures of patients who will be receiving the antigen. The main objective of this task is to predict the relationship between the level of cancer antigen and the clinical measures using the given dataset. The dataset included a column named "Train" which included two different variables "T" that is representing training and "F" representing testing. Finally, the column l*evelCancerAntigen* is the target "Y".

2.1 Data Import:

The provided dataset is a text file that includes a table containing all the data, the table is given to be 97 rows and 11 columns numbered in order from 0 to 10. The data include 97 subject records also numbered accordingly from 0 to 96. The dataset columns include three different data types where 5 columns are of float64 data type, 5 columns are of Int64 data type, and finally one column is of type object.

Number of columns	Data Types
5	Float64
5	Int64
1	Object

Figure 4: Number of columns including different data types

The provided dataset memory usage is 8.5KB which is very efficient to be loaded on any device and easy to import and process. Google Colab is used in this task to execute the python program; therefore, it is efficient pre-load the dataset into a Google Drive and connect the Google Colab to the Google Drive to import the dataset in the program. The dataset is imported using the *Pandas* library which is imported together with all the libraries used in the task in the first cell of the program.

```
[5]  import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     get_ipython().run_line_magic('matplotlib', 'inline')
     import seaborn as sns
     from sklearn import preprocessing
     from sklearn.metrics import accuracy_score
     import warnings
     warnings.filterwarnings('ignore')
```

Figure 5: All libraries imported for the task

Figure 5 above, shows all the libraries that were imported to be used in the task, each library has its own purpose and used to undertake a specific action in this regression problem where each library will be explained when reaching its usage point.

In order to be able to use the dataset in our program, it needs to be imported from the Google Drive. In order to import the dataset, the *Pandas* library is used which can be seen called in figure 5.

```
[7] rawData = pd.read_csv("/content/drive/MyDrive/Masters Degree/Machine Learning/Assessment1/task1/Task1_RegressionTask_CancerData.txt",
                          delim_whitespace=True) #importing the dataset

    print ("The shape of the data: ", rawData.shape) # prints data shape

    rawData.info()
    rawData
```

Figure 6: Importing the dataset and printing the shape of the dataset.

Figure 6 above shows the code snippet that imports the dataset in the program. The dataset is given a variable name of "rawData" which means this is the data before being pre-processed. It is imported using pandas by pasting the path of the dataset from the Google Drive in the *pd.read_csv function.*

It can be seen in the function the "*delim_whitespace = True*" which specifies that in the text file the data are separated using the whitespace tab.

In order to make sure that the dataset is imported and can be read by the function. The program is asked to print the shape of the dataset and the information of the dataset which was discussed and can be found in figure 4. After running this cell, figure 7 below shows the output.

```
The shape of the data:  (97, 11)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 97 entries, 0 to 96
Data columns (total 11 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   index               97 non-null     int64
 1   logCancerVol        97 non-null     float64
 2   logCancerWeight     97 non-null     float64
 3   age                 97 non-null     int64
 4   logBenighHP         97 non-null     float64
 5   svi                 97 non-null     int64
 6   logCP               97 non-null     float64
 7   gleasonScore        97 non-null     int64
 8   gleasonS45          97 non-null     int64
 9   levelCancerAntigen  97 non-null     float64
 10  train               97 non-null     object
dtypes: float64(5), int64(5), object(1)
memory usage: 8.5+ KB
```

	index	logCancerVol	logCancerWeight	age	logBenighHP	svi	logCP	gleasonScore	gleasonS45	levelCancerAntigen	train
0	1	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	-0.430783	T
1	2	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	-0.162519	T
2	3	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	-0.162519	T
3	4	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	-0.162519	T
4	5	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	0.371564	T
...	...	...	...	...	...	...	...	...	...	...	...
92	93	2.830268	3.876396	68	-1.386294	1	1.321756	7	60	4.385147	T
93	94	3.821004	3.896909	44	-1.386294	1	2.169054	7	40	4.684443	T
94	95	2.907447	3.396185	52	-1.386294	1	2.463853	7	10	5.143124	F
95	96	2.882564	3.773910	68	1.558145	1	1.558145	7	80	5.477509	T
96	97	3.471966	3.974998	68	0.438255	1	2.904165	7	20	5.582932	F

97 rows × 11 columns

Figure 7: Output printing the dataset, dataset shape, and dataset info

It can be seen in figure 7 that after running the cell the whole data set is printed in order to see that there are no missing data, the data shape (97,11) is also printed as well as a table showing the data info such as datatypes and dataset size.

2.2 Data Pre-processing:

In order to make the data efficient for the task, applying pre-processing is important, which is the technique of preparing our rawData by cleaning, organizing, and normalising the data.

As stated earlier in the report, the "levelCancerAntigen" column is the target column, therefore, it does not need to be pre-processed along all the dataset. The target column is always referred to as Y.

```
[11] y1=rawData['levelCancerAntigen']
     print("y1 shape is ", y1.shape)

     x1=rawData.drop(columns=['index','train','levelCancerAntigen'])
     print("x1 shape is ",x1.shape)

     y1 shape is  (97,)
     x1 shape is  (97, 8)
```

Figure 8: Defining and dropping the target column before pre-processing

As it can be seen in figure 8, the target column "levelCancerAntigen" is defined in the program as "y1" and then the program is asked to print the shape of y1, meaning to print how many columns and rows is has.

As the target column y1 does not need to be in the pre-processing phase, we ask the program to drop it from the dataset "rawData" and call the new dataset x1. As seen, the "Train" column is also dropped as this column is not important in the pre-processing as it only shows the number of "T" for training and "F" testing data.

By looking at the dataset in figure 7, it can be seen in the dataset table that the column "*Index*" only contains the number of the columns which is 0 to 96, therefore it is not needed to be pre-processed, so it is also dropped.

By looking at the output in figure 8, it can be seen that the shape of y1 is 97 rows as it is only one column and finally the shape of x1 after dropping three columns went from (97, 11) to (97, 3) confirming that out dropping function is successful.

Now that the dataset is cleaned from the rows that do not need to be included in the pre-processing. The pre-processing process can now begin; there are many different techniques in which pre-processing can be applied on the data. Looking at figure 5, it be seen that pre-processing is imported from the library *sklearn.* We choose to use the MinMaxScaler technique for scaling which rescales the value of the data to [0, 1], where is estimates each data (feature) independently, therefore, avoid the data to be biased before model fitting.

$$\mathrm{X}[:,\mathrm{i}] = \frac{x[:,i] - \min\left(X[:,i]\right)}{\max(X[:,i]) - \min\left(X[:,i]\right)} \qquad (1)$$

Equation (1) above, shows mathematically how the MinMaxScaler function rescales the input data (x) to a minimum and maximum range between [0, 1].

```
[29] scaler = preprocessing.MinMaxScaler().fit(x1)
     x1_Norm = scaler.transform(x1)
     x1 = pd.DataFrame(columns=x1.columns,data=x1_Norm)
     print(x1_Norm)

     sns.displot(x1_Norm)
```

Figure 9: Pre-processing, scaling, and normalising the data.

Looking at figure 9, it shows the python implementation of the pre-processing *MinMaxScaler* function which fits the input features between [0, 1]. It can also be seen that x1 which is the dataset after dropping the unimportant columns is being normalised and the normalised dataset is now called x1_Norm which will be used when using the regression later on. After running this cell, the values of the normalised dataset will be printed as an output and it will also be represented as a plot. The plot can be seen in the figure below where the x-axis is between 0 and 1 where the data is normalised, and the y-axis represents the columns which are counted to be 97.

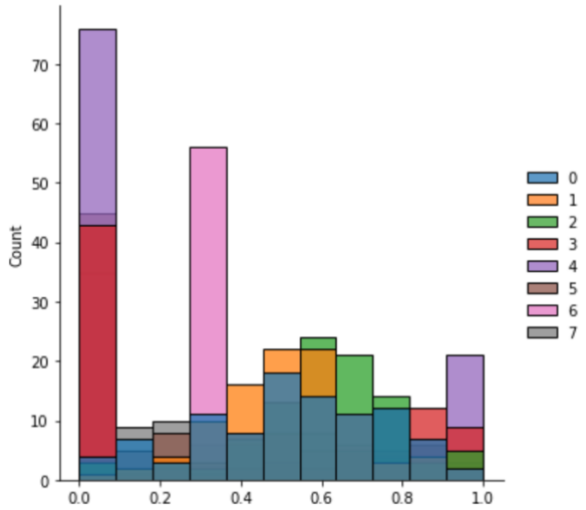

Figure 10: Plot representing the values x1 after being normalised

After the data have been pre-processed and normalised, it is now needed to bring back the dropped columns, especially the “Train” column as this is where the training and testing data are specified.

```
[49] x1 = np.array(rawData)
     print(x1)
```

Figure 11: Bringing back the dropped columns are pre-processing

The line of code in figure 11 above, assigns back all the 11 columns that were in the original rawData dataset back and then to make sure the data is back and assigned to x1, when running this cell, the x1 is printed including all the available column ready for assigning training and testing data.

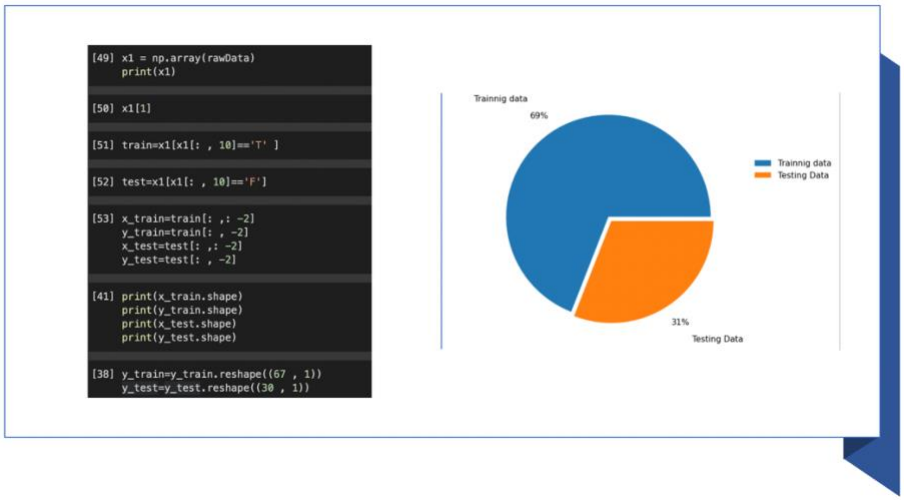

Figure 12: Dividing the data into training and testing + plotting the percentage of each

The figure above, shows the python script that divides the x1 data into training and testing data, each set of data specified as x_train and y_train also has a testing sample known as x_test and y_test. After printing the shape of the data, it is important to reshape the data in order to avoid sizing and matrices issues which were fixed using this function. A plot function is later implemented (Will be found in the code attached) which shows the plot in figure 12, showing that we have 69% of training data and 31% of testing data in total.

It is then important to visualise the relationship between the rawData between the data in order to know what are the most important columns that will have effect on our target column "levelCancerAntigen". In order to visualize the relationship, this is done by implementing a heatmap, it can be seen in figure 13 below [2].

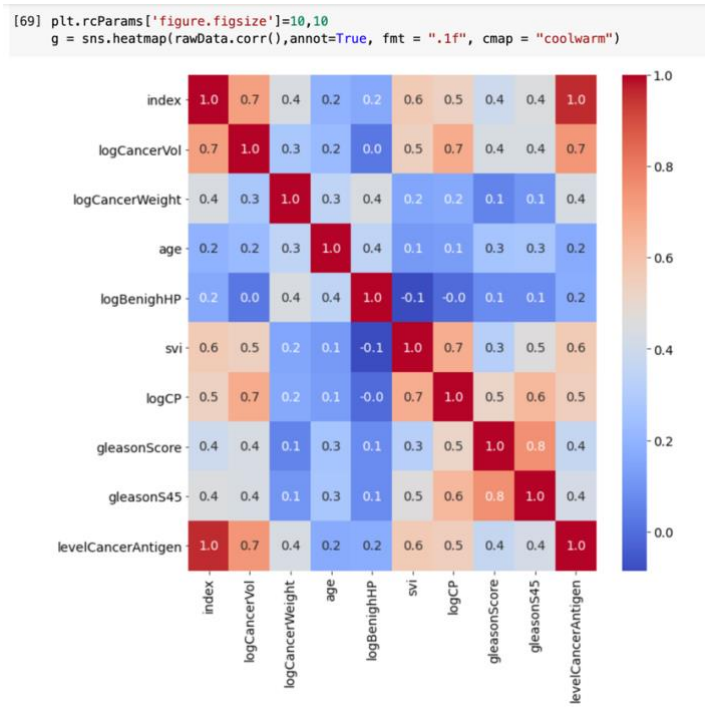

Figure 13: Heatmap representing the correlation between the columns

The heatmap in the figure above shows the correlation between the columns in the dataset after pre-processing the data, as it can be seen all the values are between 0 and 1; this is achieved after implementing the MinMaxScaler and normalization on the data.

The red boxes with a value 1.0 show a strong relationship and effect on the data, where the blue boxes show less relation and effect on the rest of the dataset and the target column Y.

2.3 Implementation of regression models:

After pre-processing, adjusting the shapes and sizing, and visualising the relationship between the data, it is now possible to start implementing different regression models on the data, as discussed before, there are many different regression models, in this task 3 different regression models are implemented and then checking which one is the best for our task.

The implementation of the regression model in two different python scripts in two different ways. This is done for comparison and better understanding of the regression models. All three models are implemented twice, the first time is an implementation mathematically without using a library and the second time is an implementation of the regression models using the sklearn library.

Starting off mathematically, before implementing the regression models, the prediction function, the gradient descent function, and the cost function are all first implemented using their mathematical equations. Where "*w*" represents the weight, "*x*" represents the input data, "*y*" represents the target value, and finally the "*LR*" represents the learning rate.

$$Gradient\ Descent = \frac{1}{n}\sum_{i=1}^{n}\big(pred(i) - y(i)\big)^2 \qquad (2)$$

Equation (2) above is for the gradient descent when it is used in linear regression models. The implementation of the gradient descent algorithm is in figure 14 below including all the parameters mentioned above.

```
[96] def prd(w , x):

         return np.dot(x , w.T)

[97] def grad(it , x , y , lr ):
         w=np.zeros((1 , 10))
         for i in range(it):
             error=prd(w , x)-y
             w=w-lr*(1/m)*np.dot( error.T , x )
         return w
```

Figure 14: Implementation of the prediction and gradient descent mathematically

As seen in figure 14, before implementing the mathematical implementation of the gradient descent in python, the prediction function is implemented to predict the weight "w".

Looking at the gradient descent function, it uses all the explained parameters needed such as the iterations, the input data, target, and learning rate. The python implementation of the gradient descent simply translates equation (2), including the learning rate and error function which will be represented graphically later. When using the sklearn library, it is not needed to implement the gradient descent mathematically as it is pre-implemented in the library.

```
[118] def cost_fun(w , x1 , y , m):
          print(w.shape , x1.shape , y.shape)
          error=prd(w , x1)-y
          cost=(1/m)*np.sum(np.square(error))
          return cost

[119] x_test.shape

      (30, 10)
```

Figure 15: Implementation of the cost function

As no library is being called for usage now, the cost function is implemented manually, the cost function is used to return the error between the predicted output and the actual output, it is the average loss across a number of samples. Figure 15 above shows the implementation of the cost function in python, it is a translation of equation (3) below:

$$Cost\ Function = J(\theta, \theta 1) = \frac{1}{2m} \sum_{i=1}^{n} \big(h\theta(i) - y(i)\big)^2 \qquad (3)$$

After implementing the prediction, gradient descent, and cost function mathematically. We can now visualise how the regression models where implemented both mathematically and using sklearn.

2.3.1 Lasso Regression:

```
[ ]  def lasso(it , x , y , lr , alpha , m ):
         w=np.zeros((1 , 10))
         for i in range(it):
             error=prd(w , x)-y
             w=w-lr*(1/m)*(np.dot( error.T , x )+alpha*(np.abs(w)))
             print(w)
         return w
```

Figure 16: Implementation of the Lasso regression mathematically

The lasso regression mathematical implementation in figure 16 uses all the functions in figures 14 and 15. A new parameters is introduced which is the alpha, when visualising the output later, changing alpha will change the output.

$$Lasso\ Regression = \arg min \sum_{i=1}^{n} \frac{1}{2} (yn - \beta)^2 + \lambda \sum_{i=1}^{P} abs(\beta) \qquad (4)$$

After visualising the mathematical representation of the lasso regression which is translated from equation (4). Now, the implementation of lasso regression using the sklearn can be observed in another python script.

Using the Sklearn library imported in figure 5, the lasso regression implementation using the library is simpler as the gradient descent and cost function do not need to be implemented manually.

```
[ ]  reg = Lasso(alpha=0.0001).fit(x1_Norm_train,y1_train)
     y1_computed = reg.predict(x1_Norm_test)
     score = r2_score(y1_test, y1_computed)

     print("The r-squared score in Lasso Regression is: ", score*100,"%")
```

Figure 17: Implementation of the Lasso regression using Sklearn

Figure 17 above, shows the python implementation of the lasso regression using the Sklearn library. Where the x1_Norm_train is the trained data after being normalised and the y1_computed is the predictive Y output.

2.3.2 Ridge Regression:

```
[80] def ridge(it , x , y , lr , alpha , m  ):
         w=np.zeros((1 , 10))
         for i in range(it):
             error=prd(w , x)-y
             w=w-lr*(1/m)*(np.dot( error.T , x )+alpha*(np.square(w)))
             print(w)
         return w
```

Figure 18: Implementation of the Ridge regression mathematically

The ridge regression mathematical implementation in figure 17 uses all the functions in figures 14 and 15. It also uses the number iterations, input, target (y), learning rate, alpha, and slope.

$$Ridge\ Regression = \frac{1}{2}\sum_{i=1}^{n} (yn - w\emptyset(xn))^2 + \frac{\lambda}{2} w\ (T)\ w \qquad (5)$$

After visualising the mathematical representation of the ridge regression which is translated from equation (5). Now, the implementation of ridge regression using the sklearn can be observed in another python script.

Using the Sklearn library imported in figure 5, the ridge regression implementation using the library is simpler as the gradient descent and cost function do not need to be implemented manually.

```
[ ]  reg = Ridge(alpha=0.7).fit(x1_Norm_train,y1_train)
     y1_computed = reg.predict(x1_Norm_test)
     score = r2_score(y1_test, y1_computed)

     print("The r-squared score in Ridge Regression is: ", score*100,"%")
```

Figure 19: Implementation of the Ridge regression using Sklearn

Figure 19 above, shows the python implementation of the lasso regression using the Sklearn library. Where the x1_Norm_train is the trained data after being normalised and the y1_computed is the predictive Y output – which is very similar to the lasso regression.

2.3.3 Lease Square Estimate:

Regarding the least square estimate, it can be known as normal linear regression, this was implemented only using the existing sklearn library.

```
[ ]  reg = LinearRegression().fit(x1_Norm_train,y1_train)
     y1_computed = reg.predict(x1_Norm_test)
     print(y1_computed.dtype)
     score = r2_score(y1_test, y1_computed)
     print("The r-squared score in Linear Regression is: ", score*100,"%")
```

Figure 20: Implementation of the linear regression using Sklearn

Figure 20 above, shows the python implementation of the lasso regression using the Sklearn library. Where the x1_Norm_train is the trained data after being normalised and the y1_computed is the predictive Y output – which is very similar to the lasso regression.

2.4 Output and evaluation of the regression models:

After implementing the regression models in two different methods, it is now time check the output of the models, the output is based on the functions written using the Sklearn library. It can be seen in the python snippets implemented using Sklearn that the R^2 needs is calculated. The R^2 score is a measure of how close the data are to be fitted on the regression line; it measures how well of a fit is the regression model.

After running each model and check the R^2 value for the linear regression, the lasso regression, and the ridge regression. It can be seen that the values are very close to each other when alpha is set to be 0.7, such that:

1- R^2 score for linear regression: 50.33%
2- R^2 score for ridge regression: 51.37%
3- R^2 score for lasso regression: 50.42%

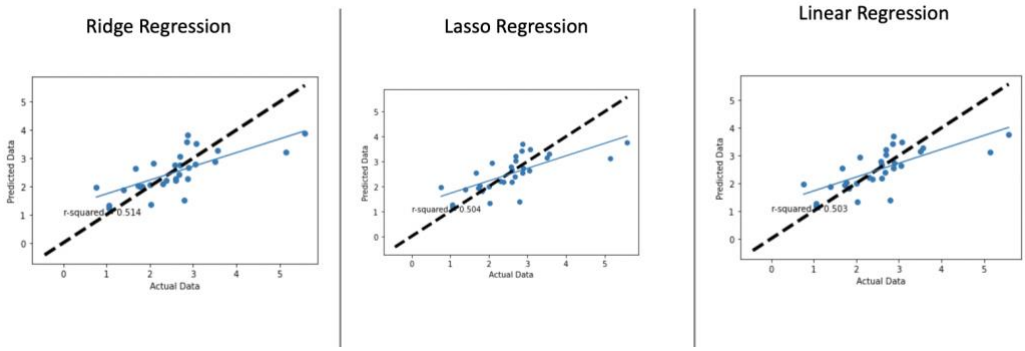

Figure 21: R^2 score for all regression models

The figure above shows the R^2 score for all three models plotted. Where the black line represents the real output Y, the blue dots around it represent the predicted output values and the blue lines represents the R^2 score.

The distance between the black line and each dot in the loss function.

When changing the "alpha" value the R^2 score is also changing, different trials where done and it can be seen that when alpha is larger in the ridge regression, the R^2 score is lower therefore, it is an inverse relationship between alpha and R^2 score.

Regarding the lasso regression, when the alpha is changed to 0.01, the R^2 score went from 50.04% to 54.4% boosted giving a much better R^2 score.

2.5 Conclusion:

As a conclusion, when applying a regression on a dataset, it is important to undertake the pre-processing and normalisation on the dataset in order to clean and organize the data before applying the regression models as well as removing all the unnecessary columns of the dataset when doing the pre-processing. In this task, the pre-processing and normalization was applied with python on the given dataset. In order to understand the relationship between dataset's columns and the target a heat map which represented the correlation was developed and showed the most important aspects of the dataset.

The regression models where implemented with two different python scripts, once mathematically, and once using pre-existing libraries. It was concluded that it was easier when implementing using the libraries, as there is no need to implement the gradient descent and loss function manually. When testing the output on both methods to check the R^2 score, it was faster and more accurate using Sklearn as it implemented graphs concluding better understanding.

Finally, after testing the three different regression models, the model with the best R^2 score is the Lasso regression, which was later optimized by adjusting the alpha value. Both full programs (Using Sklearn and Using Mathematical Functions) will be found attached in the submission.

3. Task 2:

The second task is a classification problem, where the given dataset is a set of images that include a dog or a cat figure where the objective of the task is to train a machine learning classifier to classify if the image contains a dog or a cat. This task is solved using a pre-existing library which is TensorFlow especially the TensorFlow Estimator, which is used for many actions such as training, evaluation, and prediction. A convolutional neural network is built to train the model and predict the output. Different trials are introduced in order to optimize the output and improve the accuracy of the model.

3.1 Data pre–processing:

Data pre-processing is important in this task in order to train the model efficiently. The dataset is divided into two different folders which are the training folder which includes 20002 images and the testing folder which includes 4998 images. In order to do the pre-processing different libraries needed to be imported before importing the dataset from the Google Drive account.

```
[1]  import numpy as np #for the arrays
     import os #dealing with directories
     from random import shuffle #to shuffle the data randomly
     from tqdm import tqdm #for looping with progress bar
     import matplotlib.pyplot as plt
     import cv2 #for resizing the images
     from PIL import Image

[3]  TRAIN_DIR = '/content/drive/MyDrive/tempx1/train_new'
     TEST_DIR = '/content/drive/MyDrive/tempx1/test_new'
     IMG_SIZE = 50 #resizing images to 50x50 square
     LR = 1e-3
```

Figure 22: Libraries used and importing the dataset

Figure 22 above shows the libraries that are imported mainly for the pre-processing phase, the libraries are commented in the figure to understand their purpose.

After calling the libraries, it is now the time to import the dataset to the program, the dataset is available on the Google Drive, therefore, two different folders are imported using their Google Drive path, those are the training and testing folders.

As part of the pre-processing to make the training easier and optimizable, it is important to resize all the images to be the same size, therefore, in figure 22, we introduce a new variable stating that the images size will be (50 x 50) and it will be used later.

Finally, it is important to add a learning rate to determine the step size at each iteration, the learning rate, also represented as LR is given a value of 1e-3 which is also 0.001.

```
[5]  CLASSIFICATION_MODEL = 'dogsvscats-{}-{}.model'.format(LR, '2conv-basic-video')

[6]  def label_img(img):
       word_label = img.split('.')[-3]

       if word_label == 'cat': return np.array([1,0])
       elif word_label == 'dog':return np.array([0,1])
```

Figure 23: Specifying the model name and the label image

In the next step, as seen in figure 23; before explaining the label image function, it can be seen that a model name is specified, it has a name "Classification_Model" and this is created as many different models can be tested on the CNN later in the program, if many models with different dimensions are tested on the CNN this will cause errors because of the model vs CNN dimension, therefore, this is where the main model will be saved in the same dimensions as the CNN so no confusion happens when loading different models, where the learning rate and convolutional neural network types are specified.

In the second cell, the label image function is defined, where it is directly linked to the names of the images in the dataset and how the names are split using a dot "." Which will be used to differentiate between the image's labels. The if-statement basically specifies if the word label is a cat return an array of [1,0], where 1 represent cat and 0 represent dogs. Same function works vice versa for the dogs.

```
[7]  def create_train_data():
       training_data=[]
       for img in tqdm(os.listdir(TRAIN_DIR)):
         label = label_img(img)
         path = os.path.join(TRAIN_DIR, img)
         img = cv2.resize(cv2.imread(path, cv2.IMREAD_COLOR),(IMG_SIZE, IMG_SIZE))
         training_data.append(img)

       shuffle(training_data)
       np.save('train_data.npy', training_data)
       return training_data

     def process_test_data():
       testing_data=[]
       for img in tqdm(os.listdir(TEST_DIR)):
         label = label_img(img)
         path = os.path.join(TEST_DIR, img)
         img = cv2.resize(cv2.imread(path, cv2.IMREAD_COLOR),(IMG_SIZE, IMG_SIZE))
         testing_data.append(img)

       shuffle(testing_data)
       np.save('test_data.npy', testing_data)
       return testing_data
```

Figure 24: Creating training and testing function

In figure 24 above, two functions are built which are used for the training and testing phases. The same function is applied for the training and the testing datasets, we first specify the label image specified in figure 23 to be used in the training function to read the pictures with their label name. Then, we specify the path where the training and testing sets is available, which is the same path that can be seen for training in figure 22. Using the CV library imported earlier in figure 22, we resize all the images in the training folder to be the same as our IMG_SIZE which was given to be (50 x 50). In order for the model to read the images in the dataset, using *IMREAD* it is specified that the images are given in colour and not grayscale.

Using the shuffle library, we shuffle the data in the training dataset which will optimize the training and testing. Finally, after training the data, it saved, therefore, it won't be necessary to re-size the images every time the session ends.

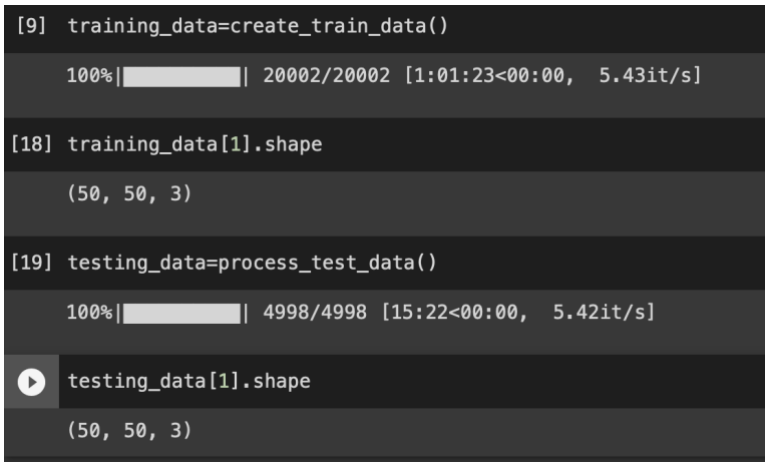

Figure 25: Resizing the shape of training and testing sets of images

Using the training and testing functions represented in figure 24, the resizing and appending is performed on both the training set which includes 20002 images and the testing set including 4998 images. Figure 25 above, shows lines of code which does this implementation and shows that the implementation is successful on both the training and the testing sets, as well as printing the shape of both after applying the resizing method which is given to be (50, 50, 3), the 3 represents the RGB.

3.2 Implementation of the Convolutional Neural Network (CNN):

When it comes to image classification problems the convolutional neural network, known as CNN, are usually chosen because of their high accuracy by outputting a one connected layer where the neurons are combined processing the output; the CNN always gets a better accuracy than the Recurrent Neural Networks (RNN) or the Long-Term Short Memory (LTSM). The reason why the accuracy is high is because the CNN make the hidden layer receptive fields local.

When implementing a CNN, different libraries and sub-libraries are used, the figure below shows the libraries importing for building the CNN.

```
[79] import tflearn
     from tflearn.layers.conv import conv_2d, max_pool_2d
     from tflearn.layers.core import input_data, dropout, fully_connected
     from tflearn.layers.estimator import regression
```

Figure 26: Libraries imported to build the CNN

As stated, classification algorithm used in the CNN is the estimator which uses regression, it is also imported with the libraries, the other imported libraries help stabilizing the size of the network with the size of the input data as well as using the pooling technique.

The figure below shows the CNN build for this model where it uses the libraries imported in figure 26 to activate and train the model efficiently, it also specifies the optimization techniques such as Adam technique which will be discussed later.

```
convnet = input_data(shape=[None, IMG_SIZE, IMG_SIZE, 3], name='input')

convnet = conv_2d(convnet, 32, 2, activation='relu')
convnet = max_pool_2d(convnet, 2)

convnet = conv_2d(convnet, 64, 2, activation='relu')
convnet = max_pool_2d(convnet, 2)

convnet = fully_connected(convnet, 1024, activation='relu')
convnet = dropout(convnet, 0.8)

convnet = fully_connected(convnet, 2, activation='softmax')
convnet = regression(convnet, optimizer='adam', learning_rate=LR, loss='categorical_crossentropy', name='targets')

model = tflearn.DNN(convnet, tensorboard_dir = 'log')
```

Figure 27: Convolutional Neural Network Architecture [3]

When building a CNN, it is really important to take the dimensions into consideration, as seen in figure 27 above, it is important to introduce the dimensions of the input data to the CNN, where the input of the training dataset is given of dimension [None, IMG_SIZE, IMG_SIZE, 3]; where the None is included to fit the dimension of the CNN to be the same, where IMG_SIZE is 50, and finally the 3 is given to read the RGB images.

The CNN is built to be 2-dimensional; it can be seen that there are 2 layers of convolution and pooling, it is important to add pooling after the convolution as it reduces the spatial size and also reduces the computation in the network operating independently. Where also the “relu”, known as the nonlinearity, can be found in the layers.

Finally, a fully connected layer is applied, which is the main reason why CNN is used for this image classification task. Finally, the connected layer works to produce the output after training the data, giving the accuracy, optimization using Adam Optimizer which is adapt to train deep neural networks.

The last line in the python snipper in figure 27, just creates the network.

```
if os.path.exists('{}.meta'.format(CLASSIFICATION_MODEL)):
    model.load(CLASSIFICATION_MODEL)
    print('Model loaded!')
```

Figure 28: Saving the progress after building the CNN

After building the CNN, the lines of code in figure 28 above save the new trained weights in our model which was initialised and introduced in figure 23.

This mode that will be created will be used when the data is called to be trained to check the accuracy and optimization of our data.

As the training and testing data are available after being pre-processed and resized to similar dimensions as well as fitting in the CNN, it is now possible to separating the training and testing data to get the accuracy of the model.

```
[ ]  X = np.array([i for i in training_data]).reshape(-1, IMG_SIZE, IMG_SIZE, 3)

     Y = np.array([label_img(i) for i in tqdm(os.listdir(TRAIN_DIR))])

     print(Y)

[ ]  X1 = np.array([i for i in testing_data]).reshape(-1, IMG_SIZE, IMG_SIZE, 3)

     Y1 = np.array([label_img(i) for i in tqdm(os.listdir(TEST_DIR))])

     print(X1.shape)
     print(Y1.shape)
```

Figure 29: Separating the training and testing data as X and Y.

In figure 29 above, (X) represents the features set and (Y) represents the target labels. Where the feature set X is the NumPy array of the training data, then it is reshaped to the dimension seen in the code, this code snippet is what is getting fit.

Two more variables X1 and Y1 are introduced which represent the testing data same way the training data was represented above; this code snippet is the testing accuracy.

```
[ ]  model.fit({'input': X}, {'targets': Y}, n_epoch=5, validation_set=({'input': X1}, {'targets': Y1}),
         snapshot_step=500, show_metric=True, run_id = CLASSIFICATION_MODEL)
```

Figure 30: Training the network

After separating the training and testing data, the code in figure 30 above uses the variables we introduced in figure 29 (X, Y, X1, Y1) to train the network. Where is says that the input is X, target is Y for the training data. Regarding the validation set related to the testing, the input is X1 and the target is Y1.

Starting off the training with 5 epochs which is the number of passes of the training set which the classification algorithm completed. Finally, the Run ID is the model chosen to run which is the model that is specified in figure 23.

```
[30] model.fit({'input': X}, {'targets': Y}, n_epoch=5, validation_set=({'input': X1}, {'targets': Y1}),
         snapshot_step=500, show_metric=True, run_id = MODEL_NAME)

     Training Step: 3129  | total loss: 11.68330 | time: 4.975s
     | Adam | epoch: 010 | loss: 11.68330 - acc: 0.4926 -- iter: 19968/20002
     Training Step: 3130  | total loss: 11.73822 | time: 6.044s
     | Adam | epoch: 010 | loss: 11.73822 - acc: 0.4902 | val_loss: 11.51293 - val_acc: 0.5000 -- iter: 20002/20002
     --
```

Figure 31: Results of the first training

After running the first training on the network with an epoch of only 5 – the accuracy that was achieved is 50% after training all the images.

Different techniques were tried in order to increase the accuracy such as:

1- Increasing the number of epochs
2- Adding padding to the CNN
3- Changing the size of the images from 50x50 to larger (128x128)
4- Adding a batch size to the model fit

Unfortunately, after trying all of the above techniques and running the model, the accuracy kept swinging only between 49% and 50%. It was considered adding different classification algorithms that would help increase the accuracy [4].

3.3 Conclusion:

As a conclusion, using the convolutional neural network (CNN) is very active with image classification problems as known in wide different applications. Starting the task, it was important to resize all the images to one size which is given in the program to be of dimension (50 x 50), applying the same dimension on all images in both the training and the testing sets makes the model easier to train and more optimizable.

Tensorflow was used to train the data in the CNN, it is a pre-existing library that has a faster compilation time then Keras and Touch. Many different errors appeared during the compilation such as dimensions, reading the images, and training the images. Those errors were solved using intensive research and applying different methods to solve the problem.

Unfortunately, after applying all the steps to resize, build the CNN, and train the model, the accuracy achieved was only 50%, which is the same accuracy as flipping a coin and not reliable.

Different methods were applied to try and solve the error such as adding early stopping to the CNN to avoid overfitting, the number of epochs was changed to more rounds to try and optimize the model by more training and therefore increasing the accuracy, and finally changing the dimensions of the input images as 50 x 50 which was small and some pixels were not visible making it harder to train, changing it to 64 x 64 once and 128 x 128 making the images more visible and trainable for the model.

4. References:

[1] E. Miller. 2014. “Introduction to supervised learning”. Available at: https://people.cs.umass.edu/~elm/Teaching/Docs/supervised2014a.pdf [Online]. Accessed 24 November 2020.

[2] M. Waseem. 2020. “How to implement classification in Machine Learning”. Available at: https://www.edureka.co/blog/classification-in-machine-learning/ [Online]. Accessed 23 November 2020.

[3] Python Programming. “TFLearn – High Level Abstraction Layer for TensorFlow Tutorial”. Available at: https://pythonprogramming.net/tflearn-machine-learning-tutorial/ [Online]. Accessed 26 November 2020.

[4] T. Jajodia. P. Grag. M. Agrasen. 2019. “Image Classification – Cat and Dog Images”. Available at: https://www.irjet.net/archives/V6/i12/IRJET-V6I1271.pdf [Online]. Accessed 25 November 2020.

[5] R. Chawla. 2018. “Using deep learning to classify dogs and cats” Available at: https://ravishchawla.files.wordpress.com/2017/02/deeplearning-report.pdf [Online]. Accessed 28 November 2020.