Cool Beans - Machine Learning Assignment

June 26, 2020

COMS3007 - Machine Learning Assignment 2020

Group Name: Cool Beans

Group Members: Michael Gomes – 1644868 , Tristen Paul – 1826461 , Tristan Le Forestier – 1835635 , Anrich Kok – 1893975

1) Dataset Description

1.1) Basic Description of Dataset:

The dataset we chose to use for this assignment is a phishing website features dataset which aims to determine the legitimacy of a website. The target values labels are either -1 or 1 and are used to respectively denote a legitimate or deceptive website. This dataset comprises of 11055 datapoints which each contain 30 attributes.

1.2) An in-depth description of each attribute and its possible value:

- having_IP_Address - This attribute describes if a website URL uses an IP address or a domain name. If the website uses an IP address, it is represented by the value 1 and indicates that the website is more likely to be a phishing site. A value of -1 indicates that the URL uses the domain name.
- URL_Length - This attribute describes whether the URL is suspiciously long. If the URL is longer than usual the attribute is given the value 1. If the URL length is unknown it gets the value 0. If the URL length is average or lower it gets the value -1. Sometimes phishers utilise longer URLs as a way to hide questionable parts of a URL and hence this attribute plays a key role in determining the authemticity of a website.
- Shortening_Service - This attribute describes if a URL shortening service like tinyURL has been used. This is a feature as some phishing websites may use this technique to hide their own suspicious URL. If a shortening service is used then it is assigned the value 1 otherwise it is assigned the value -1.
- having_At_Symbol - This attribute describes if a URL has an "@" symbol included in it. Some phishing websites may utilise the "@" symbol to make the browser ignore everything before the "@" symbol which may indicate that a site is a phishing site. If an "@" symbol is used then it is assigned the value 1 otherwise it is assigned the value -1.
- double_slash_redirecting - This attribute describes if a URL has another "//" included in it past the 6th and 7th position required for HTTP and HTTPS URLs respectively. An extra "//" in a seemingly non phishing URL may be used to redirect people to phishing websites and is thus included as a feature. If an extra "//" is found then it is assigned the value 1 otherwise it is assigned the value -1.

- Prefix_Suffix - This attribute describes if a URL has another “-” included in it. Adding a prefix/suffix to a URL is a technique used by phishers to make their URLs look more legitimate. If “-” is found in the URL then it is assigned the value 1 otherwise it is assigned the value -1.
- having_Sub_Domain - This attribute describes if a URL has multiple subdomains. Usually phishing websites use multiple subdomains in their URL’s to trick users to thinking it is a legitimate website. If the URL has more than 2 subdomains, it gets the value 1 and is classified as “phishing”. If it has 1 subdomain then it gets the value 0 and is classified as “suspicious”. If it has no subdomains then it gets the value -1 and is classified as “legitimate”.
- SSLfinal_State - This attribute describes if the website utilises HTTPS. The majority of legitimate websites which require the user to input sensitive information use HTTPS instead of HTTP in order to pretect user data. If the website uses a different protocol that is not well known unlike HTTP or HTTPS then it is assigned a value of 1 which means “phishing”. If the protocol type is well know like HTTP then it gets the value 0 which means it is “suspicious”. If the website uses HTTPS this attribute gets the value -1.
- Domain_registeration_length - This attribute describes the time period that a domain has been registered for. The majority of legitimate website domains are usually paid for multiple years in advance; conversely, phishing website domains have a short life span. If the domain has been registered for a longer period of time, for example, greater than 1 year, then it gets the value -1 otherwise it gets the value 1.
- Favicon - This attribute describes if the website has a correctly associated favicon (graphical image associated with a specific website). Some phising websites use favicons that are not associated with their current domain in the URL but ones associated with legitimate websites. If the favicon association is not matched with the current website domain, this feature gets the value 1 which indicates that it is phishing. If they match then the feature is assigned the value -1.
- port - This attribute describes if the website is trying to commuinicate over non-standard ports. Phishing websites sometimes try to run their services through non-standard ports. If this happens, the value assigned to the feature is 1 and if the site only uses standard ports the feature is assigned the value -1.
- HTTPS_token - This attribute describes if the domain part of the URL contains an HTTPS. Phishers use this to mislead users into believing that a website is authentic. If this is the case, this feature is assigned the value 1 and if not, it is assigned the value -1.
- Request_URL - This attribute describes if external components of the website such as pictures and videos are loaded from another domain. Phishers sometimes use assets from other legitimate websites in order to make their site look as legitimate as possible. If this is is true for the particular site, the feature is assigned the value 1 otherwise it is assigned the value -1.
- URL_of_Anchor - This attribute describes whether the anchor tag and request URL and the website have the same domain name. A phisher might get access to your main tab and change the anchor tag/request URL so you get redirected to their phishing website. If the anchor tag and request URL have different domain names then the feature is assigned the value 1. If the anchor tag/request URL has no domain and does not redirect it is assigned the value 0. If the anchor tag and request URL have the same main domain names then the feature is assigned the value -1.

- Links_in_tags - This attribute describes whether the Meta, Script and Link tags in the HTML code of the website have the same domain as the current website. This is a feature as majority of legitimate websites have all these tags corresponding to the main domain of the current page. If all of the tags do not correspond to the same domain then it is assigned the value 1. If some tags correspond and others do not then it is assigned the value 0. If they all correspond to the same domain then it is assigned the value -1.
- SFH - This attribute describes whether the domain in SFHs(Server Form Handler) is the same as the domain of the main page when information is submitted. When some information is submitted it processed by an SFH. If the SFH domain is not the same as the main website domain, it is most probably phishing as submitted information is not usually handled by an external domain. If this is the case then the value assigned is 1. If the SFH domain is blank then it is assigned the value 0 as it is suspicious as something should be done with the submitted data. If the domains match then the value -1 is assigned.
- Submitting_to_email - This attribute describes whether a user's submitted information on a website is also mailed to a third party using a “mailto:” function instead of going to a server for processing. Phishers can sometimes re-route submitted information from a website to their personal emails by using “mailto:” functions. If a “mailto:” function is used then the feature is assigned the value 1 otherwise it is assigned the value -1.
- Abnormal_URL - This attribute describes whether a website domain has a legitimate identity attached to its URL according to the WHOIS database. If this is not the case then the feature is assigned the value -1 otehrwise it is assigned the value 1.
- Redirect - This attribute describes if a website has been redirected a suspicious number of times. It has been found that legitimate websites redirect at maximum one time but phishising websites redirect at least 4 times. If a website redirects a maximum of 3 times and a minimum of once then the feature is assigned the value of 0. If it redirects 4 or more times then it is assigned the value 1.
- on_mouseover - This attribute describes if JavaScript and an “onMouseOver” event is used to create a falsified URL in the status bar of the current website. If this is true then the feature is assigned the value of 1 otherwise it is assigned the value of -1.
- RightClick - This attribute describes if the right-click function has been disabled on the website. Phishers disable right-click functionality on their websites in order to prevent users from accessing the website source in order to look for suspicious code. If the right-click function has been disabled the feature is given the value of 1 and if right-click function has not been disabled it is given the value -1.
- popUpWindow - This attribute describes if a pop-up window is used on the website for users to input their personal information. Pop-ups are almost never used in legitimate websites for data input purposes but are frequently used in phishing websites. If a pop-up is used for input then this feature is assigned the value of 1 and if not, it is assigned the value of -1.
- Iframe - This attribute describes if iFrame tags are used anywhere in the HTML code of the website. IFrame tags are used display a seperate website within the one currently shown. Phishers use these iFrame tags without borders to make the user think that they are accessing the correct website in order to obtain the user's details. If iFrame tags are present in the HTML code this feature is assigned the value of 1. If no iFrame tags are found it is assigned the value of -1.

- age_of_domain - This attribute describes the domain age according to the WHOIS database. Phishing websites are active for short periods of time and hence have small domain ages. If the domain has been active for less than 6 months the value assigned to it is 1. If the domain age is greater than 6 months it is assigned the value of -1.
- DNSRecord - This attribute describes if the DNS record identity according to the WHOIS database exists and is recognised. Phishing websites usually have unrecognisable or null DNS identities in their records in the WHOIS databases. If the identity is null or unrecognisable then this feature is assigned the value of 1. If the identity is known then this feature gets the value of -1.
- web_traffic - This attribute describes if a website's traffic is suspiciously low. Due to the fact that phishing websites are active for short periods of time, they should have a very little traffic. For example, if the phisher is pretending to be a popular online banking site and the traffic of that specific page does not represent the expected number of vistors then that website should be flagged for phishing. If a website has no traffic then this feature is assigned the value of 1. If the website has a low traffic then it is assigned the value of 0. If it has sufficient traffic then it is assigned the value of -1.
- Page_Rank - This attribute describes the page rank of the website. The page rank of a website is a value within the range of 0-1 which indicates the importance of a particular page; the higher the value, the more crucial it is. Most phishing websites will have a very low page rank due to their low visitor traffic as well as short operational period. If the page rank is low this feature is assigned the value of 1 and if the page rank is high it is assigned the value of -1.
- Google_Index - This attribute describes whether the web page is shown on Google Index. Google Index is a list of all web site pages that will show up in search results. Phishing websites are usually only accessible for a short amount of time and as a result will not show up on the Google Index. If the website is not on Google Index it is assigned the value of 1 and if it appears on the Google Index it is assigned the value of -1.
- Links_pointing_to_page - This attribute describes whether the website has a suspiciously small amount of external links pointing to the same domain. Most legitimate websites have multiple links pointing to themselves whereas phishing websites do not due to their lifespan. If the website has no external links poitning to it this feature is assigned the value of 1, if the website has 1 external link pointing to it this feature is assigned the value of 0 and if the website has a minimum of 2 external links pointing to it this feature is assigned the value of -1.
- Statistical_report - This attribute describes whether the domain of the website or IP address attached to website is on the respective lists provided by PhishTank and StopBadware. These lists are updated regularly and provide information about known or suspected phising sites. If the IP address appears on any of the lists this feature is assigned the value of 1 and if it is not on any of the lists it is assigned the value of -1.

1.3) Example of Data Points:

-1,1,1,1,-1,-1,-1,-1,-1,1,1,-1,1,-1,1,-1,-1,-1,0,1,1,1,1,-1,-1,-1,-1,1,1,-1,-1

The attribute order in the data point correspond to the order of the attributes in the description above as read from top to bottom. The last value in the data point is the Target Value which in this case is -1 which indicates that this website is a legitimate one.

1,0,-1,1,1,-1,1,1,-1,1,1,1,1,0,0,-1,1,1,0,-1,1,-1,1,-1,-1,0,-1,1,1,1,1

For the data point above this website is classified with a 1 which means it is not legitimate and is used as a source of data phishing.

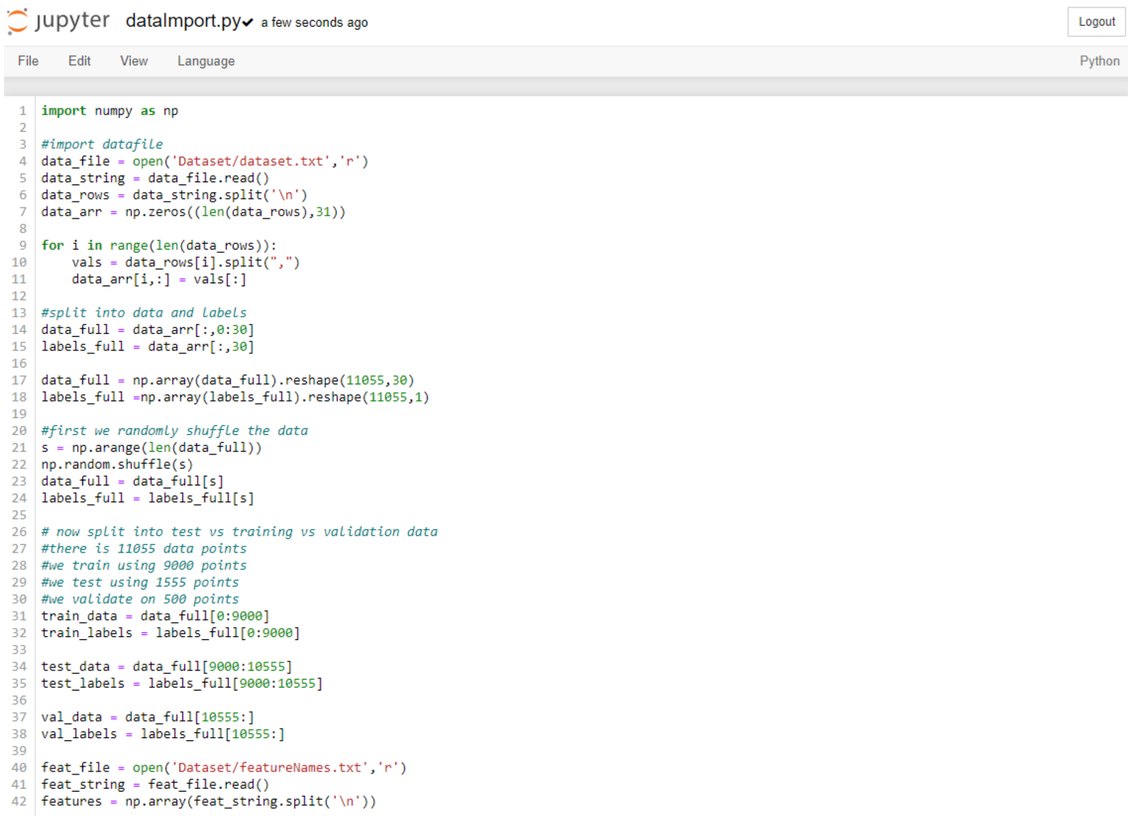
2) Description about the input structuring

We read in the data in the form of the above examples and split each data point using the "," character as a delimiter. Once the data has been split, we have a matrix with dimensions 11055x31. We then create a new matrix called data_full with dimensions 11055x30 which takes in everything from the original matrix except the final column which is contains our target values. We also create an array called labels_full with dimensions 11055x1 which takes in the last target values from our original matrix.

We then simultaneously shuffle our data in data_full and labels_full using a random function to ensure that the indices of the labels match the corresponding data.

After the data has been shuffled, we take the first 9000 data points and labels and put them into a train_data matrix (9000x30) and an array called train_labels (9000x1) respectively. This is used as training data. The next 1555 data points and labels are added to a test_data matrix (1555x30) and an array called test_labels (1555x1). This serves as our test data. Finally, the next 500 data points and labels are added to a matrix called val_data (500x30) and an array called val_labels (500x1). This data is used for validation.

This code can be found in the file called dataImport.py which does all of the above and passes the respective data and label matrices and arrays. The following is a screenshot of that code:



```python
import numpy as np

#import datafile
data_file = open('Dataset/dataset.txt','r')
data_string = data_file.read()
data_rows = data_string.split('\n')
data_arr = np.zeros((len(data_rows),31))

for i in range(len(data_rows)):
    vals = data_rows[i].split(",")
    data_arr[i,:] = vals[:]

#split into data and labels
data_full = data_arr[:,0:30]
labels_full = data_arr[:,30]

data_full = np.array(data_full).reshape(11055,30)
labels_full =np.array(labels_full).reshape(11055,1)

#first we randomly shuffle the data
s = np.arange(len(data_full))
np.random.shuffle(s)
data_full = data_full[s]
labels_full = labels_full[s]

# now split into test vs training vs validation data
#there is 11055 data points
#we train using 9000 points
#we test using 1555 points
#we validate on 500 points
train_data = data_full[0:9000]
train_labels = labels_full[0:9000]

test_data = data_full[9000:10555]
test_labels = labels_full[9000:10555]

val_data = data_full[10555:]
val_labels = labels_full[10555:]

feat_file = open('Dataset/featureNames.txt','r')
feat_string = feat_file.read()
features = np.array(feat_string.split('\n'))
```

3 & 5) Classification Algorithms and Code of their implementation

We used the following algorithms in this assignment:

- Decision Trees
- Naive Bayes
- Linear Regression
- Linear Regression with Regularisation
- Logistic Regression

The code and desciptions for each of these algorithms are given below.

Importing necessary libraries and the respective data and label structures discussed above in 2.

[2]:
```
import numpy as np
from ete3 import *
import math
import pandas as pd
from dataImport import * #Dataset imports
import copy
import matplotlib.pyplot as mp
import random
```

Decision Tree Classifier

For this classification method we used the normal ID3 algorithm without pruning to train our decision tree. We chose this algorithm as it was the first algorithm we were exposed to in this course and we wanted to see how it performed relative to some of the other more complex algorithms we learnt later in the course. The dataset given in the decision tree lab did not have as many features as our current dataset and we were curious to see if the algorithm still yielded accurate results given more complex data.

[3]:
```
#Calculate Entropy Function
def calc_entropy(data):
    unique = np.unique(data)
    count = np.zeros(unique.size)
    for iLoop1 in range(len(data)):
        for iLoop2 in range(unique.size):
            if data[iLoop1] == unique[iLoop2]:
                count[iLoop2] = count[iLoop2] + 1

    total = len(data)
    entropies = np.zeros(unique.size)
    for iLoop1 in range(unique.size):
        if count[iLoop1] == 0:
            entropies[iLoop1] = 0
        else:
            entropies[iLoop1] = (count[iLoop1]/total)*math.log((count[iLoop1]/
↪total),2)

    return([-(np.sum(entropies)), unique]) # returns the entropy of the data as␣
↪well as the set of unique value for the data
```

[4]:
```
#create class model for Decision Tree
class tree_node:
    def __init__(self,data,labels,diagram_node,available_features):
        self.data = data  # holds the data which the tree node will use to find␣
↪the best feature
        self.labels = labels  # holds the corresponding labels for each data␣
↪point
        self.diagram_node = diagram_node  # Used for creating the tree diagram
        self.available_features = available_features  # Holds the list of names␣
↪for the remaining features available at a tree node
        self.feature_index = None  # Holds the index of the feature in the list␣
↪of features which provides the most information gain
        self.feature_name = None  # Holds the name of the feature which␣
↪provides the most information gain
        self.node_values = None  # Holds the unique values of the feature which␣
↪provides the most information gain
        self.is_leaf = False  # Reflects if the node is a leaf node
        self.class_value = None  # Is set to True or False if a node classifies␣
↪a data point (it is a leaf node)
        self.children = None  # Array to hold the children node of this node
        self.node_entropy = calc_entropy(labels[:])[0]

        if not (self.node_entropy == 0.0 or self.node_entropy == -0.0 or self.
↪data.shape[1] == 0):
            self.children = []
            self.find_feature()
            self.descend_tree()


        elif self.data.shape[1] == 0:
            self.is_leaf = True
            unique, counts = np.unique(self.labels[:], return_counts=True)
            majority_class = np.argmax(counts)
            self.class_value = unique[majority_class]
            self.feature_name = str(self.class_value)
            self.diagram_node.name = self.feature_name
        else:
            self.is_leaf = True
            self.class_value = labels[0]
            self.feature_name = str(labels[0])
            self.diagram_node.name = self.feature_name


    #find features for nodes
    def find_feature(self):
        #print("Finding feature for new node")
        feature_entropies = np.zeros(len(self.available_features))  #␣
↪initialize the entropy of each feature to 0
        info_gains = np.zeros(len(self.available_features))  # init info gains

        for i in range(len(info_gains)):
            info_gains[i] += self.node_entropy

        for i in range(self.data.shape[1]):  # For each feature
            #print("Working on feature: ", i)
            feature_entropies[i], _ = calc_entropy(self.labels[:])
            feature_sub_classes = np.unique(self.data[:,i])
            #print(feature_sub_classes) #works

            for feature in feature_sub_classes:   # for each unique value of␣
↪the feature calc entropy for each subpoint
                sub_clas_data = np.where(self.data[:, i] == feature)[0]   #␣
↪find the data points where this feature value occurs
                #print(sub_clas_data)
                #print(self.labels[sub_clas_data])
                data_ratio = len(sub_clas_data) / self.data.shape[0]  # calc␣
↪how much of the total data has this feature value
                sub_clas_entropy = calc_entropy(self.labels[sub_clas_data])[0] ␣
↪ # calc entropy for the subset of data with the feature value
                info_gains[i] -= data_ratio * sub_clas_entropy  # update the␣
↪information gain
            #print(info_gains)

        chosen_feature = info_gains.argmax()   # choose feature which gives max␣
↪info gain
        #print(chosen_feature)
        self.feature_index = chosen_feature  # update features of the node class
        self.feature_name = self.available_features[self.feature_index]
        self.diagram_node.name = self.feature_name
        #print("Found feature: ", self.feature_name)

    # This function is used to add nodes to the tree once the best feature to␣
↪split the data is found
    def descend_tree(self):
        #print("Descending tree with node entropy value: ", self.node_entropy)
        unique, counts = np.unique(self.data[:, self.
↪feature_index],return_counts=True)  # Find the unique values of the chosen␣
↪feature
        #print(unique)
        self.node_values = unique  # Update class values which holds the values␣
↪of the feature it uses
        #print(self.node_values)
        for value in self.node_values: # For each unqiue value the chosen␣
↪feature can take
            data_for_feature_value = np.where(self.data[:, self.feature_index]␣
↪== value)[0] # Find data where unique value for the feature occurs
            #print(data_for_feature_value)
            remaining_features = np.arange(self.data.shape[1]) != self.
↪feature_index # This just drops the chosen feature from the list of unused␣
↪feature names
            #print(remaining_features)
            new_child_diagram_node = self.diagram_node.add_child(name="Temp") ␣
↪# used for making the tree diagram

            # For each unique value of the chosen feature we add a new node.␣
↪Some points to note
            # First we only use the data which assigned the unique value we are␣
↪looking for for the feature
            # this is found in the "data_for_feature_value" variable above
            # Secondly we remove the feature we used to split the data, the␣
↪unused features are found in the
            # variable "remaining_features"
            self.children = np.append(self.children, tree_node(self.
↪data[data_for_feature_value][:, remaining_features],
                                                            self.
↪labels[data_for_feature_value],
                                                           ␣
↪new_child_diagram_node,
                                                            self.
↪available_features[remaining_features]))


    # This function infers (predicts) the class of a new/unseen data point. We␣
↪call this on the test data points
    #need help
    def infer(self, data_point):
        if not self.is_leaf:                            # if the node we are␣
↪looking at is not a leaf node (can't classify the data point)
            for i in range(len(self.node_values)):                  # look␣
↪through the set of values the node looks for
                if data_point[self.feature_index] == self.node_values[i]:
                    remaining_features = np.arange(self.data.shape[1]) != self.
↪feature_index          # to find which branch to descend down
                    allocated_class = self.children[i].
↪infer(data_point[remaining_features])          # recursively run the infer␣
↪function on the child node (excluding the features which have been used␣
↪already, see how this was done to get "remaining_features" when decsending␣
↪the tree above)
                    return allocated_class  # return back up the tree
                #print("Error found new value, can't classify")
        else:
            #print("Classified data point as: ", self.class_value)
            return (
                self.class_value)  # If it is a leaf node then we just return␣
↪the classification given by the leaf node
```

Now we train the model using the code above and display it:

```
[5]: t = Tree()  # Used for diagram, creates tree and adds root node to tree diagram␣
↪(use as third input to tree_node class)
root = tree_node(train_data, train_labels, t, features)  # Use our class to␣
↪train a decision tree on the training data

#print(t.get_ascii(show_internal=True))
ts = TreeStyle()

def my_layout(node):
    F = TextFace(node.name, tight_text=True)
    F.rotable = True
    F.border.width = 0
    F.margin_top = 5
    F.margin_bottom = 5
    F.margin_left = 5
    F.margin_right = 5
    add_face_to_node(F, node, column=0, position="branch-right")


ts.layout_fn = my_layout
ts.mode = 'r'
ts.arc_start = 270
ts.arc_span = 185
ts.draw_guiding_lines = True
ts.scale = 100
ts.branch_vertical_margin = 100
ts.min_leaf_separation = 100
ts.show_scale = False
#t.show(tree_style=ts)
```

Now we compute the training, testing and validation errors:

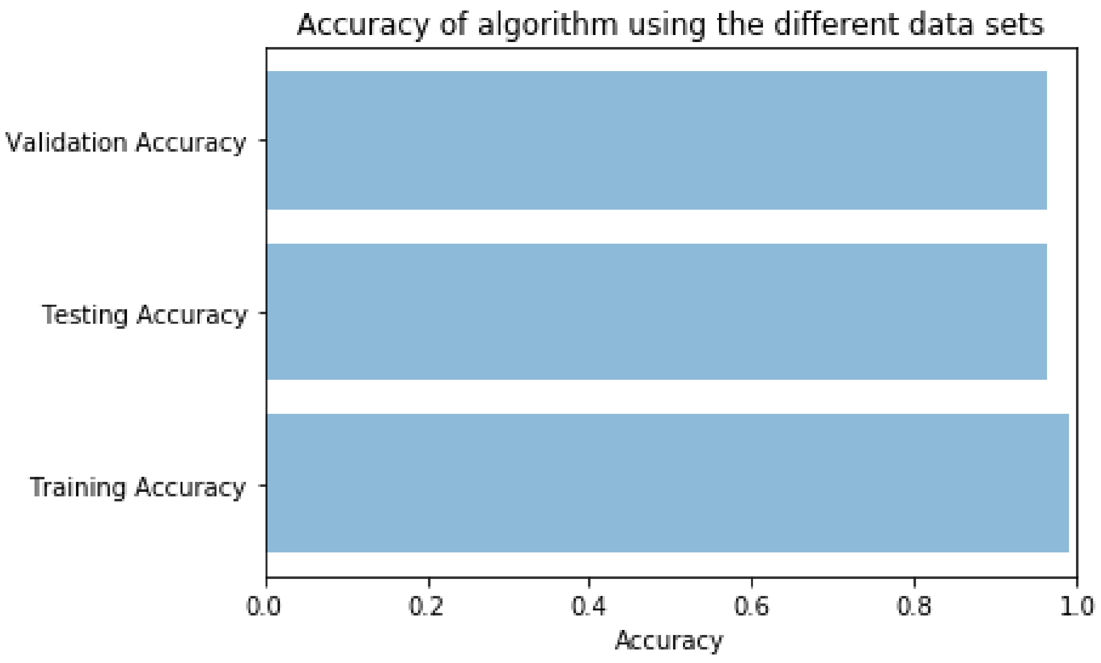
[5]:
```
#checking error values for tree

pred = []
count_correct = 0
for i in range(train_data.shape[0]):
    out = root.infer(train_data[i]) # infer the class on the data point
    if(type(out) == np.ndarray):
        pred.append(out[0])
    else:
        pred.append(out)
    is_correct = (pred[i] == train_labels[i])
    if is_correct:
        count_correct = count_correct + 1
train_ans = count_correct / train_labels.shape[0]

print("Confusion Matrix for Training:")
y_actu = pd.Series(train_labels.reshape(-1), name='Actual')
y_pred = pd.Series(np.array(pred), name='Predicted')
df_confusion = pd.crosstab(y_actu, y_pred)
print(df_confusion)
print()
print("Training error for tree model: ",train_ans)
print("-------------------------")
print()
Tree_Train = train_ans

pred = []
count_correct = 0
for i in range(test_data.shape[0]):
    out = root.infer(test_data[i]) # infer the class on the data point
    if(type(out) == np.ndarray):
        pred.append(out[0])
    else:
        pred.append(out)
    is_correct = (pred[i] == test_labels[i])
    if is_correct:
        count_correct = count_correct + 1
test_ans = count_correct / test_labels.shape[0]

print("Confusion Matrix for Testing:")
y_actu = pd.Series(test_labels.reshape(-1), name='Actual')
y_pred = pd.Series(np.array(pred), name='Predicted')
df_confusion = pd.crosstab(y_actu, y_pred)
print(df_confusion)
print()
```

```python
print("Testing error for tree model: ", test_ans)
print("-------------------------")
print()
Tree_Test = test_ans

pred = []
count_correct = 0
for i in range(val_data.shape[0]):
    out = root.infer(val_data[i]) # infer the class on the data point
    if(type(out) == np.ndarray):
        pred.append(out[0])
    else:
        pred.append(out)
    is_correct = (pred[i] == val_labels[i])
    if is_correct:
        count_correct = count_correct + 1
val_ans = count_correct / val_labels.shape[0]

print("Confusion Matrix for Validation:")
y_actu = pd.Series(val_labels.reshape(-1), name='Actual')
y_pred = pd.Series(np.array(pred), name='Predicted')
df_confusion = pd.crosstab(y_actu, y_pred)
print(df_confusion)
print()
print("Validation error for tree model: ", val_ans)
print("-------------------------")
Tree_Val = val_ans
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       3928    47
 1.0         42  4983

Training error for tree model:  0.9901111111111112
-------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        679    25
 1.0         21   817

Testing error for tree model:  0.9620578778135048
-------------------------

Confusion Matrix for Validation:
```

```
Predicted  -1.0   1.0
Actual
-1.0        205     9
 1.0          9   277

Validation error for tree model:  0.964
-------------------------
```

```
[6]: objects = ('Training Accuracy', 'Testing Accuracy', 'Validation Accuracy')
y_pos = np.arange(3)
performance = [train_ans,test_ans,val_ans]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithm using the different data sets')

mp.show()
```



From the results, we observe that the Decision Tree Algorithm performs spectacularly well on this dataset and achieves over 95% accuracy on the training, testing and validation data. It also performs consistently across all the data since the accuracies are relatively close to each other.

End of Decision Tree Algorithm

Naive Bayes Classifier

For this classification method we used the Naive Bayes algorithm. We chose this algorithm due to the fact that it is probabilistic and would allow us to view the certainty level of each prediction. Once again, the lab in which we used this algorithm did not have a dataset with as many features as the one we are using now, and we wanted to see if it still yielded accurate results.

First we split our data into it's 2 classes i.e ones that are legitimate denoted by -1 and ones that are phishing denoted by 1:

```
[7]: #seperate classes into positive and negative
negative = []
positive = []
for i in range(train_data.shape[0]):  # positive
    if train_labels[i] == -1:
        negative.append(train_data[i])
    elif train_labels[i] == 1:
        positive.append(train_data[i])

probLegit = len(positive)/train_data.shape[0]
probPhishing = len(negative)/train_data.shape[0]

positive = np.array(positive).reshape(len(positive),30)
negative = np.array(negative).reshape(len(negative),30)
```

Now we find unique feature values for each feature:

```
[8]: #get unique feature values for each feature
feature_vals = []
probPos = []
probNeg = []
for i in range(train_data.shape[1]):
    unique_vals = np.unique(train_data[:,i])
    zero1 = np.zeros(len(unique_vals))
    zero2 = np.zeros(len(unique_vals))
    probPos.append(zero1)
    probNeg.append(zero2)
    feature_vals.append(unique_vals)
```

Now we calculate different probability sets for our features:

```
[9]: for i in range(negative.shape[0]):
    for j in range(negative.shape[1]):
        if(negative[i,j]==1):
            index = np.where(feature_vals[j] == 1)
            probNeg[j][index]+=1

        elif (negative[i,j]==-1):
            index = np.where(feature_vals[j] == -1)
            probNeg[j][index] += 1
        else:
            index = np.where(feature_vals[j] == 0)
            probNeg[j][index] += 1


for i in range(positive.shape[0]):
    for j in range(positive.shape[1]):
        if(positive[i,j]==1):
            index = np.where(feature_vals[j] == 1)
            probPos[j][index]+=1

        elif (positive[i,j]==-1):
            index = np.where(feature_vals[j] == -1)
            probPos[j][index] += 1

        else:
            index = np.where(feature_vals[j] == 0)
            probPos[j][index] += 1

for j in range(train_data.shape[1]):
    probPos[j][:] /= train_data.shape[0]
    probNeg[j][:] /= train_data.shape[0]
```

We will use the following to train the Naive Bayes model with all our training data and labels:

[10]:
```
class naiveModel:
    def __init__(self,probPos,probNeg,feature_vals,probLegit,probPhishing):
        self.probPos = probPos
        self.probNeg = probNeg
        self.features = feature_vals
        self.probLegit = probLegit
        self.probPhis = probPhishing

        # print(self.probPhis)
        # print(self.probLegit)
        # print(self.features)
        # print(self.probPos)
        # print(self.probNeg)
    def probability(self,input,probabilitySet):
        pSc = 1
        for i in range(len(self.features)):
            # print(input[i])
            index = np.where(self.features[i] == input[i])
            # print(index)
            pSc *= (probabilitySet[i][index])
        return pSc
    def infer(self,input):
        valTrue = self.probability(input,self.probPos)
        valFalse = self.probability(input,self.probNeg)

        valTrue = (valTrue*self.probLegit)/(self.probLegit*valTrue +␣
↪valFalse*self.probPhis)
        valFalse = 1-valTrue

        if (valTrue > valFalse):
            probValue = valTrue
            probClass = "1"
        elif (valTrue < valFalse):
            probValue = valFalse
            probClass = "-1"
        else:
            # make random guess
            probValue = 0.5
            probClass = random.choice("1","-1")

        return ([probValue, probClass])
```

Now we create a function to test our different sets of values and return their confusion matrices and probabilties.

```
[11]: def testError(test_data,test_labels):
    y_pred = []
    y_actu = []
    for i in range(test_data.shape[0]):
        probability, resultClass = nModel.infer(test_data[i])  # calculate␣
↪probability of which class and return result
        if resultClass == "1":
            # print("URL features are: ")
            # print(test_data[i])
            # print("This URL is Legit")
            # print("Probability is: ")
            # print(probability)
            # print()
            y_pred.append(1.0)
        elif resultClass == "-1":
            # print("URL features are: ")
            # print(test_data[i])
            # print("This URL is Phishing")
            # print("Probability: ")
            # print(probability)
            # print()
            y_pred.append(-1.0)
    for i in range(len(test_labels)):
        y_actu.append(test_labels[i][0])

#     print("Results:")
#     print()
#     print("----------------------")
#     print("Predicted:")
#     print(y_pred)
#     print("Actual:")
#     print(y_actu)
#     print("----------------------")

    total = test_data.shape[0]
    count = 0
    for i in range(test_data.shape[0]):
        if y_pred[i] == y_actu[i]:
            count = count + 1

    y_actu = pd.Series(y_actu, name='Actual')
    y_pred = pd.Series(y_pred, name='Predicted')
    df_confusion = pd.crosstab(y_actu, y_pred)
    print(df_confusion)

    print()
    print("Accuracy:")
    print(float(count / total))
    print("--------------------------")
    print()
    return float(count / total)
```

Lastly we train our class model and test the different data sets.

```
[12]: nModel = naiveModel(probPos,probNeg,feature_vals,probLegit,probPhishing)
print("Model trained using training data...")
```

```
Model trained using training data…
```

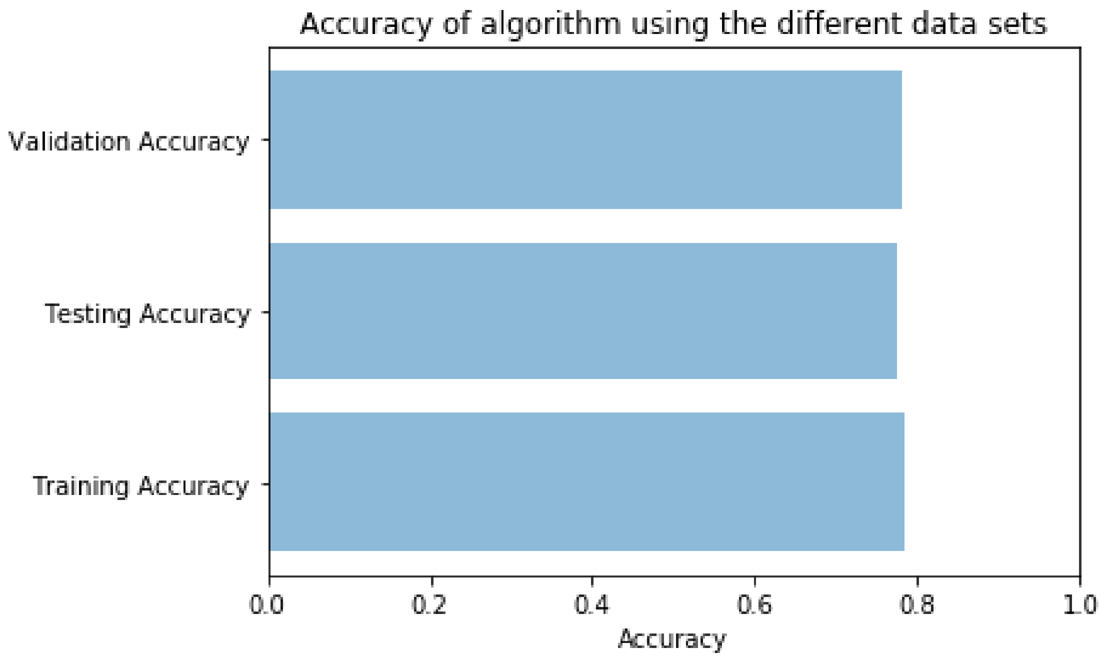
Now we compute the training, testing and validation errors:

```
[13]: print("Confusion Matrix for Training:")
train_ans = testError(train_data,train_labels)
Naive_Train = train_ans

print("Confusion Matrix for Testing:")
test_ans = testError(test_data,test_labels)
Naive_Test = test_ans

print("Confusion Matrix for Validation:")
```

```
val_ans = testError(val_data,val_labels)
Naive_Val = val_ans
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       2022  1953
 1.0          0  5025

Accuracy:
0.783
-------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        358   351
 1.0          0   846

Accuracy:
0.7742765273311897
-------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0        105   109
 1.0          0   286

Accuracy:
0.782
-------------------------
```

[14]:
```
objects = ('Training Accuracy', 'Testing Accuracy', 'Validation Accuracy')
y_pos = np.arange(3)
performance = [train_ans,test_ans,val_ans]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithm using the different data sets')

mp.show()
```



From the results, we can clearly see that the Naive Bayes Algorithm performs quite well on this dataset but not as well as the Decision Tree classifier. It still manages to achieve around 78% accuracy on the training, testing and validation data. We can also once again say that it performs quite consistently across all the data as the accuracies were quite close to each other.

End of Naive Bayes Algorithm

Linear Regression Classifier

For this classification method we used the normal linear regression via gradient descent but without regularisation. We chose this algorithm as we were wondering how well we could use it as a classifier especially with all the features this dataset has.

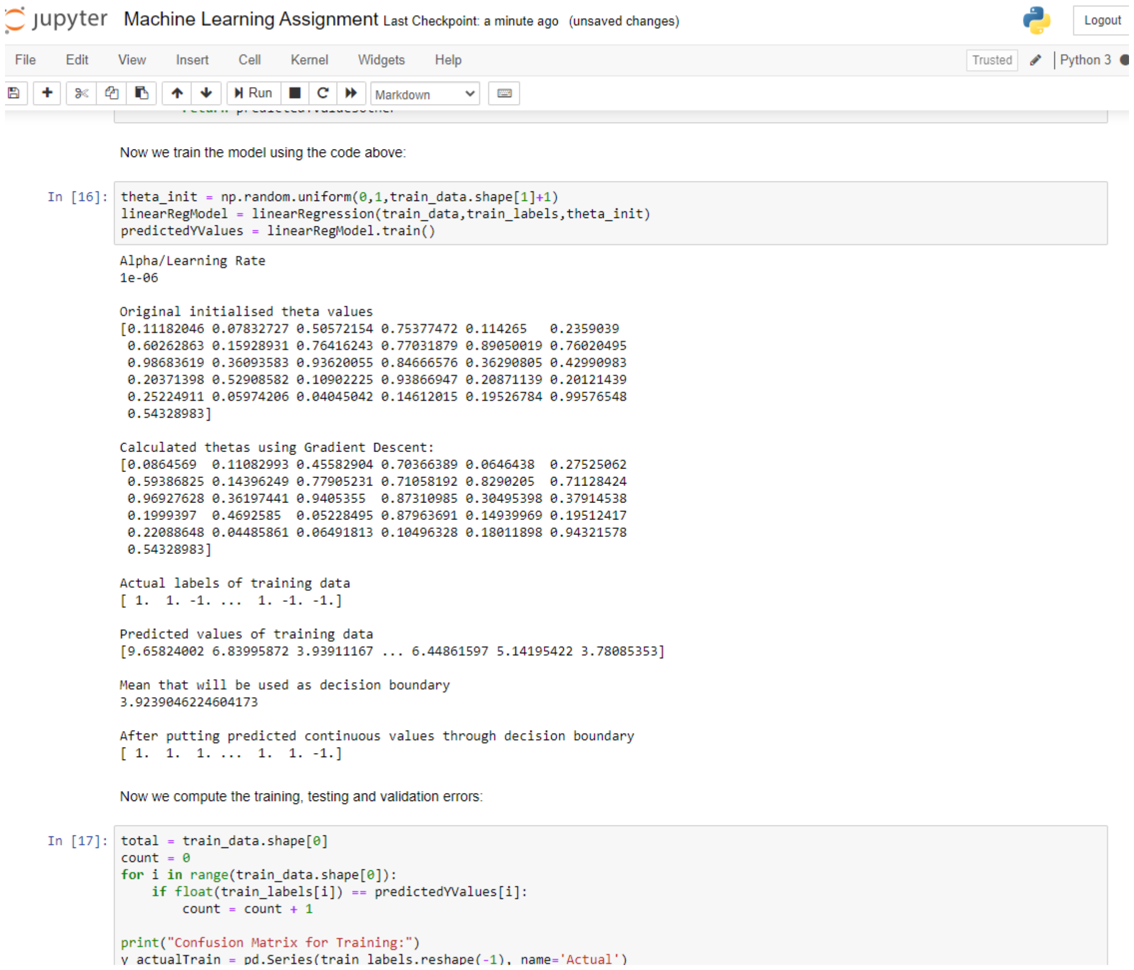
We used 31 basis functions for each different datapoint in the design matrix: 1,feature 1 value of datapoint,feature 2 value of datapoint,feature 3 value of datapoint,...,feature 30 value of datapoint. Thus we end up with a design matrix for the training data of shape 9000x31, test data of shape 1555x31 and validation data of shape 500x31. We chose these basis functions as we wanted to see how well linear regression would work with basis functions that were of the power of 1 and not products between different functions. We also randomly initialised the 31 weights/thetas using random values from a uniform distribution of mean 0 and standard deviation of 1.

During the tuning of hyperparameters, we decided to use a learning rate/alpha of value 0.00001 as we observed that this learning rate yielded the greatest accuracy. We found that when using a smaller learning rate the classifier took too long to converge and when using a larger learning rate, it would sometimes never converge. Please see the cells below for proof of these claims.

We found that the predicted continuous values using this model had some degree of seperation from a central value which may be used to classify them as 1.0 or -1.0. We took the predicted continuous values after training and futhermore calculated their mean. Any predicted value greater than or equal to this mean would be classified as a 1.0 and any predicted value less than this mean would be classified as -1.0. Thus the same mean is used as a decision boundary for the training, testing and validation data.

The training of the algorithm(gradient descent) runs until the 2-norm of the theta values < 1E-8 which indicates convergence.

The following screenshots are an example of what accuracies we got when the made the alpha value smaller than 0.00001: N.B. I would show an example of accuracies when we make the alpha value larger also but most of the time it never converged.


jupyter Machine Learning Assignment Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Markdown

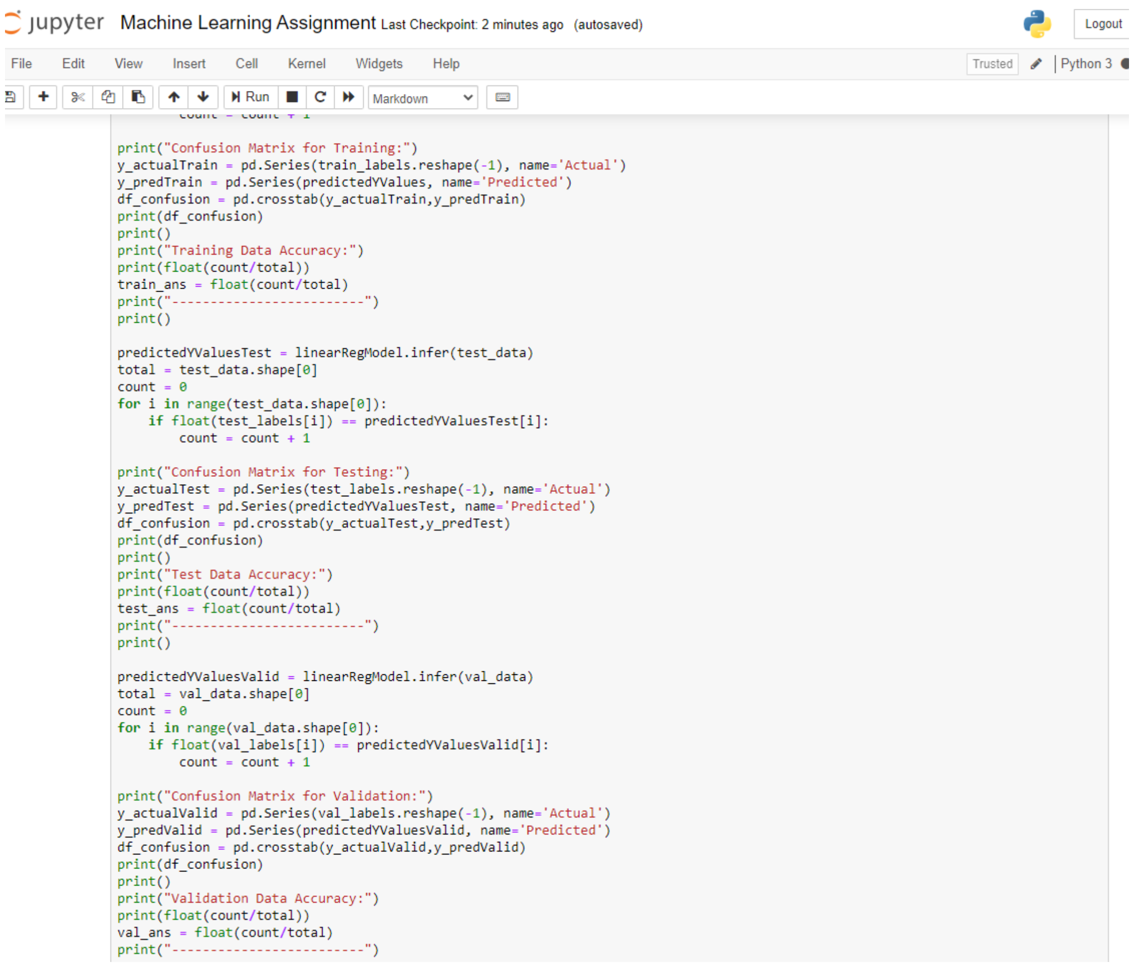
Now we train the model using the code above:

```
In [16]: theta_init = np.random.uniform(0,1,train_data.shape[1]+1)
         linearRegModel = linearRegression(train_data,train_labels,theta_init)
         predictedYValues = linearRegModel.train()
```

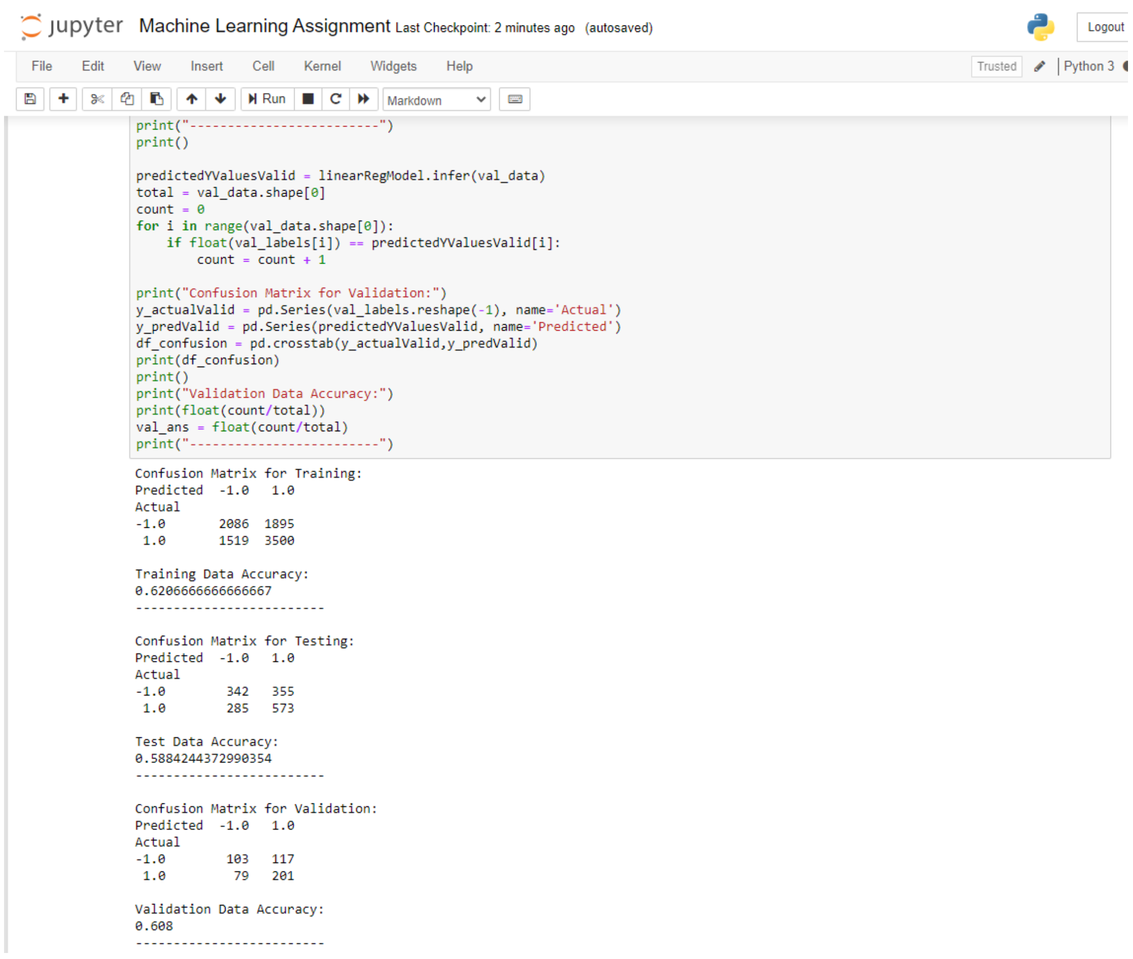
```
Alpha/Learning Rate
1e-06

Original initialised theta values
[0.11182046 0.07832727 0.50572154 0.75377472 0.114265   0.2359039
 0.60262863 0.15928931 0.76416243 0.77031879 0.89050019 0.76020495
 0.98683619 0.36093583 0.93620055 0.84666576 0.36290805 0.42990983
 0.20371398 0.52908582 0.10902225 0.93866947 0.20871139 0.20121439
 0.25224911 0.05974206 0.04045042 0.14612015 0.19526784 0.99576548
 0.54328983]

Calculated thetas using Gradient Descent:
[0.0864569  0.11082993 0.45582904 0.70366389 0.0646438  0.27525062
 0.59386825 0.14396249 0.77905231 0.71058192 0.8290205  0.71128424
 0.96927628 0.36197441 0.9405355  0.87310985 0.30495398 0.37914538
 0.1999397  0.4692585  0.05228495 0.87963691 0.14939969 0.19512417
 0.22088648 0.04485861 0.06491813 0.10496328 0.18011898 0.94321578
 0.54328983]

Actual labels of training data
[ 1.  1. -1. ...  1. -1. -1.]

Predicted values of training data
[9.65824002 6.83995872 3.93911167 ... 6.44861597 5.14195422 3.78085353]

Mean that will be used as decision boundary
3.9239046224604173

After putting predicted continuous values through decision boundary
[ 1.  1.  1. ...  1.  1. -1.]
```

Now we compute the training, testing and validation errors:

```
In [17]: total = train_data.shape[0]
         count = 0
         for i in range(train_data.shape[0]):
             if float(train_labels[i]) == predictedYValues[i]:
                 count = count + 1

         print("Confusion Matrix for Training:")
         y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
```


jupyter Machine Learning Assignment Last Checkpoint: 2 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Markdown

```
print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
print()
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("------------------------")
print()

predictedYValuesTest = linearRegModel.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print(df_confusion)
print()
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("------------------------")
print()

predictedYValuesValid = linearRegModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```


jupyter
Machine Learning Assignment Last Checkpoint: 2 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Markdown

```
print("------------------------")
print()

predictedYValuesValid = linearRegModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       2086  1895
 1.0       1519  3500

Training Data Accuracy:
0.6206666666666667
------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        342   355
 1.0        285   573

Test Data Accuracy:
0.5884244372990354
------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0        103   117
 1.0         79   201

Validation Data Accuracy:
0.608
------------------------
```

Code we will use to train model and infer on the other sets with alpha set to 0.00001:

```
[15]: class linearRegression:
          def __init__(self,train_data,train_labels,theta_init):
              self.train_data = train_data
              self.train_labels = train_labels
              self.dMatrixTrain = np.zeros((train_data.shape[0],train_data.
       ↪shape[1]+1))
              self.dMatrixTrain[:,0]=1
              i = 1
              for i in range(train_data.shape[1]):
                  self.dMatrixTrain[:,i]=train_data[:,i]

              self.y_vals = train_labels[:].reshape(-1)
              self.theta_pred_GD = theta_init.copy()
              self.theta_diff = np.zeros(train_data.shape[1]+1)
              self.descent_vals = np.dot(self.dMatrixTrain,self.theta_pred_GD)
              self.meanTrain = 0

          def train(self):
              alpha = 0.00001
```

```
        iterations = 1
        print("Alpha/Learning Rate")
        print(alpha)
        print()
        i=0
        j=0
        print("Original initialised theta values")
        print(self.theta_pred_GD)
        print()
        while(np.linalg.norm(self.theta_diff,2)>0.00000001) or (i==0 and␣
↪iterations==1):
            #for each theta
            for j in range(self.train_data.shape[1]+1):
                gradi = (self.descent_vals[i]- self.y_vals[i])*self.
↪dMatrixTrain[i][j]
                self.theta_diff[j] = abs(self.theta_pred_GD[j]-alpha*gradi-self.
↪theta_pred_GD[j])
                self.theta_pred_GD[j] = self.theta_pred_GD[j]-alpha*gradi
                self.descent_vals = np.asarray(self.dMatrixTrain.dot(self.
↪theta_pred_GD)).reshape(-1)

            i = i + 1

            if i==self.train_data.shape[0]:
                i=0
            iterations += 1

        print("Calculated thetas using Gradient Descent:")
        print(self.theta_pred_GD)
        self.descent_vals = np.asarray(self.dMatrixTrain.dot(self.
↪theta_pred_GD)).reshape(-1)

        print()
        print("Actual labels of training data")
        print(self.y_vals)

        print()
        print("Predicted values of training data")
        print(self.descent_vals)

        print()
        print("Mean that will be used as decision boundary")
        self.meanTrain = np.sum(np.array(self.descent_vals))/self.train_data.
↪shape[0]
        print(self.meanTrain)
        predictedYValues = []
        for j in range(self.train_data.shape[0]):
            if(self.descent_vals[j] >= self.meanTrain):
                predictedYValues.append(1.)
            else:
                predictedYValues.append(-1.)
        print()
        print("After putting predicted continuous values through decision␣
↪boundary")
        print(np.array(predictedYValues))
        return predictedYValues

    def infer(self,input):
        dMatrixOther = np.zeros((input.shape[0],input.shape[1]+1))
        dMatrixOther[:,0]=1
        i = 1
        for i in range(input.shape[1]):
            dMatrixOther[:,i]=input[:,i]

        descent_valsOther = np.dot(dMatrixOther,self.theta_pred_GD)

        predictedYValuesOther = []
        for j in range(input.shape[0]):
            if(descent_valsOther[j] >= self.meanTrain):
                predictedYValuesOther.append(1.)
            else:
                predictedYValuesOther.append(-1.)
        return predictedYValuesOther
```

Now we train the model using the code above:

```
[16]: theta_init = np.random.uniform(0,1,train_data.shape[1]+1)
linearRegModel = linearRegression(train_data,train_labels,theta_init)
predictedYValues = linearRegModel.train()
```

```
Alpha/Learning Rate
1e-05

Original initialised theta values
[0.03052677 0.63704864 0.32018559 0.12463098 0.02304814 0.55619361
 0.51792906 0.50477338 0.23232713 0.9863715  0.82262516 0.99414685
 0.39898618 0.34300795 0.02986272 0.73676063 0.86461745 0.66765734
 0.24684599 0.42157647 0.2576714  0.35773843 0.17056024 0.93254676
 0.12962183 0.94627489 0.64728959 0.37841903 0.24243509 0.48946095
 0.75335357]

Calculated thetas using Gradient Descent:
[-0.02366495  0.68271432  0.20586428  0.01731139 -0.0904842   0.63075801
  0.49441781  0.45068304  0.28036174  0.84365263  0.67680895  0.87898991
 0.34984514  0.33217985  0.04809576  0.78379939  0.72277135  0.54929378
 0.2395644   0.28150046  0.12945699  0.21771082  0.03251069  0.88356002
 0.05127419  0.88593889  0.68089105  0.28143084  0.20932801  0.36854648
 0.75335357]

Actual labels of training data
[ 1. -1. -1. … -1.  1.  1.]

Predicted values of training data
[-0.81763231  3.0155923   0.02197189 …  2.18569083  2.34374303
  3.54701735]

Mean that will be used as decision boundary
2.3383354072461957

After putting predicted continuous values through decision boundary
[-1.  1. -1. … -1.  1.  1.]
```

Now we compute the training, testing and validation errors:

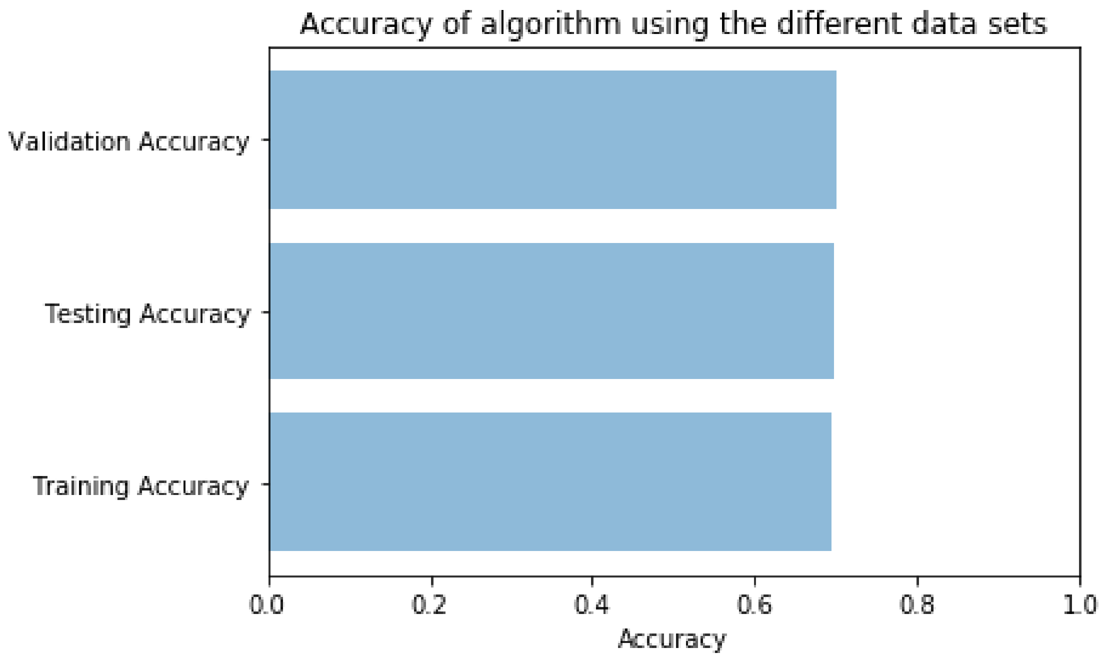
```
[17]: total = train_data.shape[0]
count = 0
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
print()
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("--------------------------")
print()
Lin_Train = train_ans

predictedYValuesTest = linearRegModel.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
```

```
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print(df_confusion)
print()
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("--------------------------")
print()
Lin_Test = test_ans

predictedYValuesValid = linearRegModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("--------------------------")
Lin_Val = val_ans
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       2661  1314
 1.0       1451  3574

Training Data Accuracy:
0.6927777777777778
-------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        480   229
 1.0        242   604

Test Data Accuracy:
```

```
0.6971061093247588
-------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0        144    70
 1.0         80   206

Validation Data Accuracy:
0.7
-------------------------
```

[18]:
```python
objects = ('Training Accuracy', 'Testing Accuracy', 'Validation Accuracy')
y_pos = np.arange(3)
performance = [train_ans,test_ans,val_ans]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithm using the different data sets')

mp.show()
```



From the results, we can clearly see that this algorithm performs decently on this dataset but does not yield extremely precise results. It gets around 70% of predictions correct, but this may be due to our choice of decision boundary (training predicted continuous mean). It is possible that using more complex basis functions could yield better results. This algorithm is actually one of the most consistent so far as the accuracies are between 0.1% of each other.

End Of Normal Linear Regression Classifier

Linear Regression with Regularisation Classifier

For this classification method we used the normal linear regression via gradient descent but this time with the inclusion of regularisation. We chose this algorithm as we were wondering how much of an impact regularisation could have on the performance of this algorithm on our dataset.

We used the same basis functions and design matrices that we used in the previous method and also used the same initial weights/thetas.

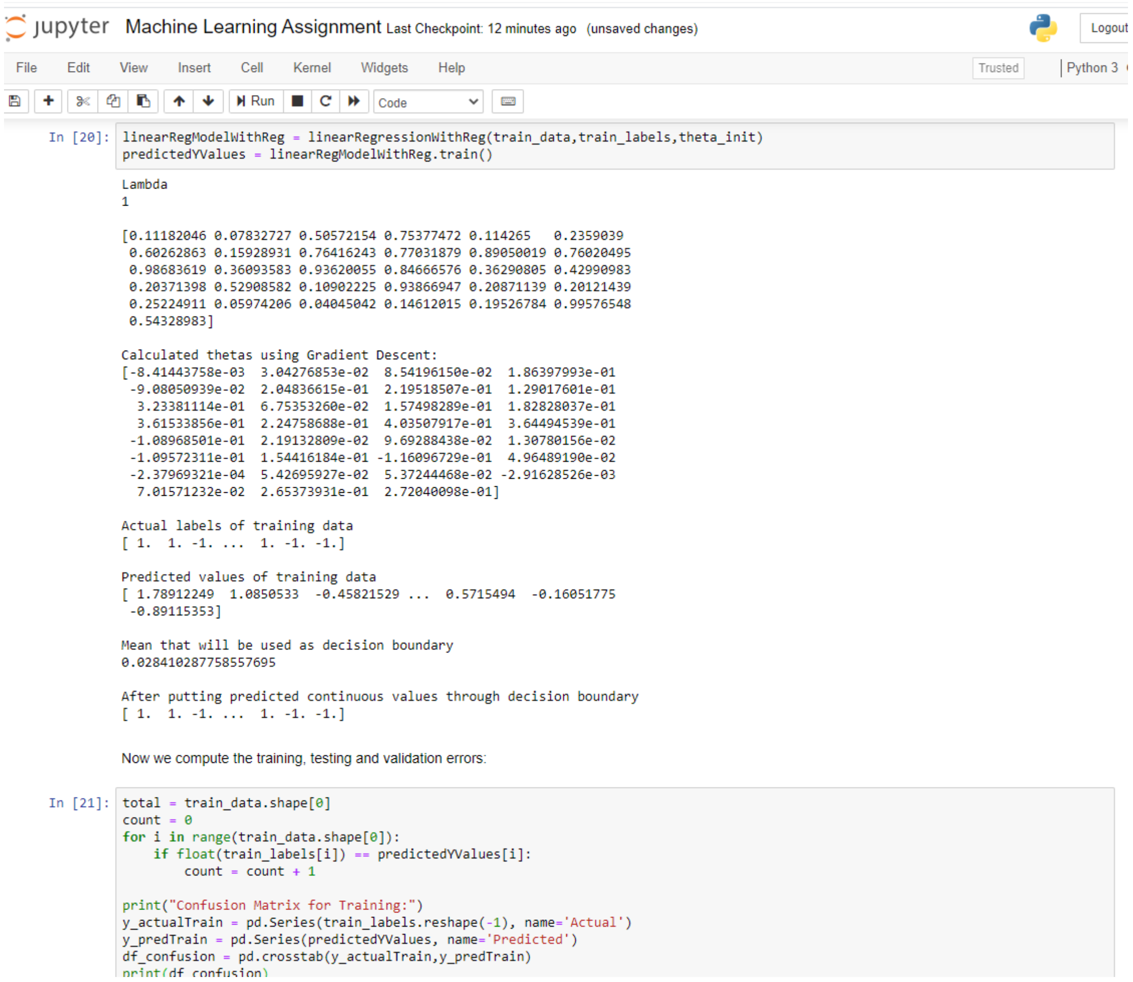
We kept the learning rate/alpha 0.00001 in order to keep the hyperparameters as constant as possible in order to accurately evaluate the effect of regularisation.

During the tuning of hyperparameters, we found that a lambda value of 2 yielded the best results. Using a number greater or smaller by a value of 1 led to decreased accuracy of the mdoel. Please see the cells below for proof of these claims.

We found that the predicted continuous values using this model had some degree of separation from a central value which may be used to classify them as 1.0 or -1.0. We took the predicted continuous values after training and furthermore calculated their mean. Any predicted value greater than or equal to this mean was classified as a 1.0 and any predicted value less than this mean was classified as -1.0. Thus the same mean is used as a decision boundary for the training, testing and validation data.

The training of the algorithm(gradient descent) runs until the 2-norm of the theta values $< 1E-5$ which indicates convergence.

The following screenshots are an example of what accuracies we got when the made the lambda value set to 1:

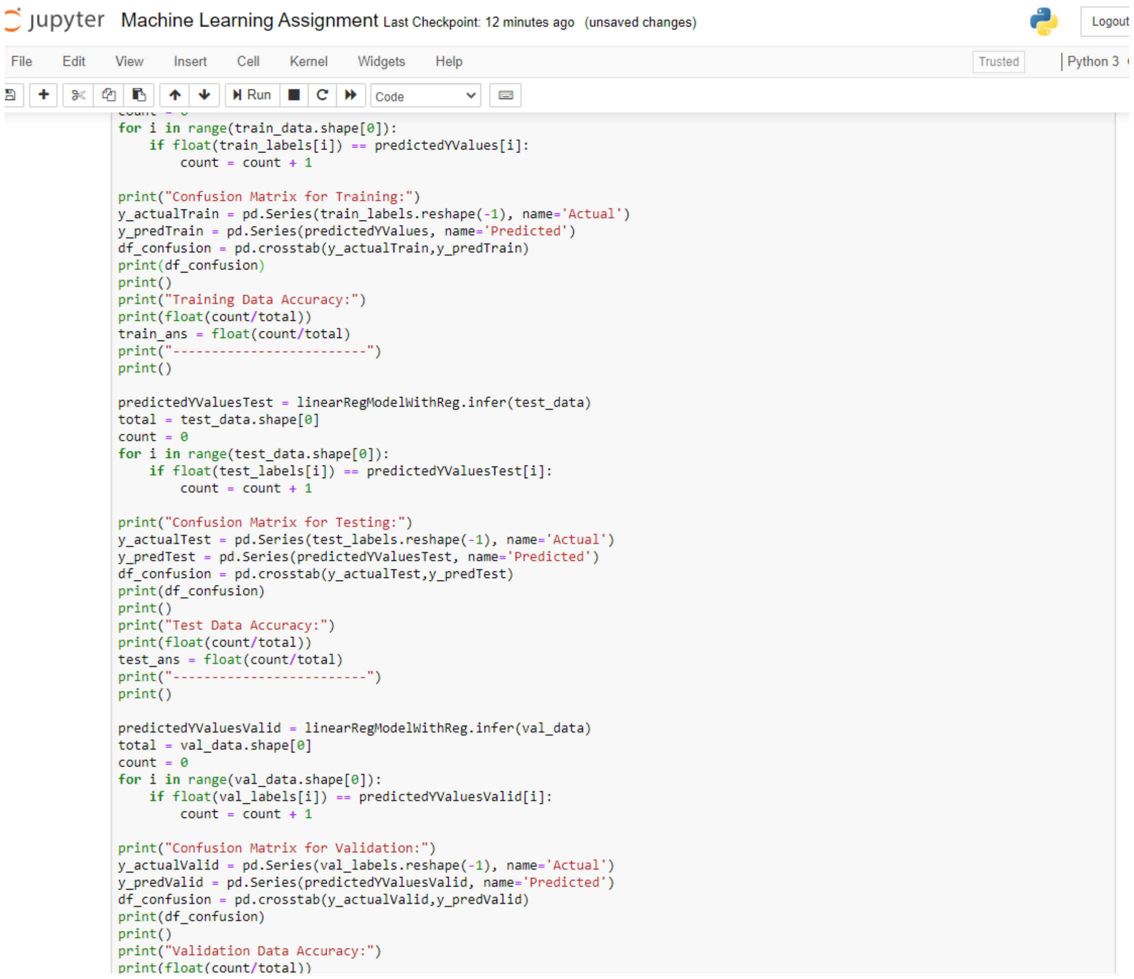

jupyter Machine Learning Assignment Last Checkpoint: 12 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

In [20]:
```
linearRegModelWithReg = linearRegressionWithReg(train_data,train_labels,theta_init)
predictedYValues = linearRegModelWithReg.train()
```

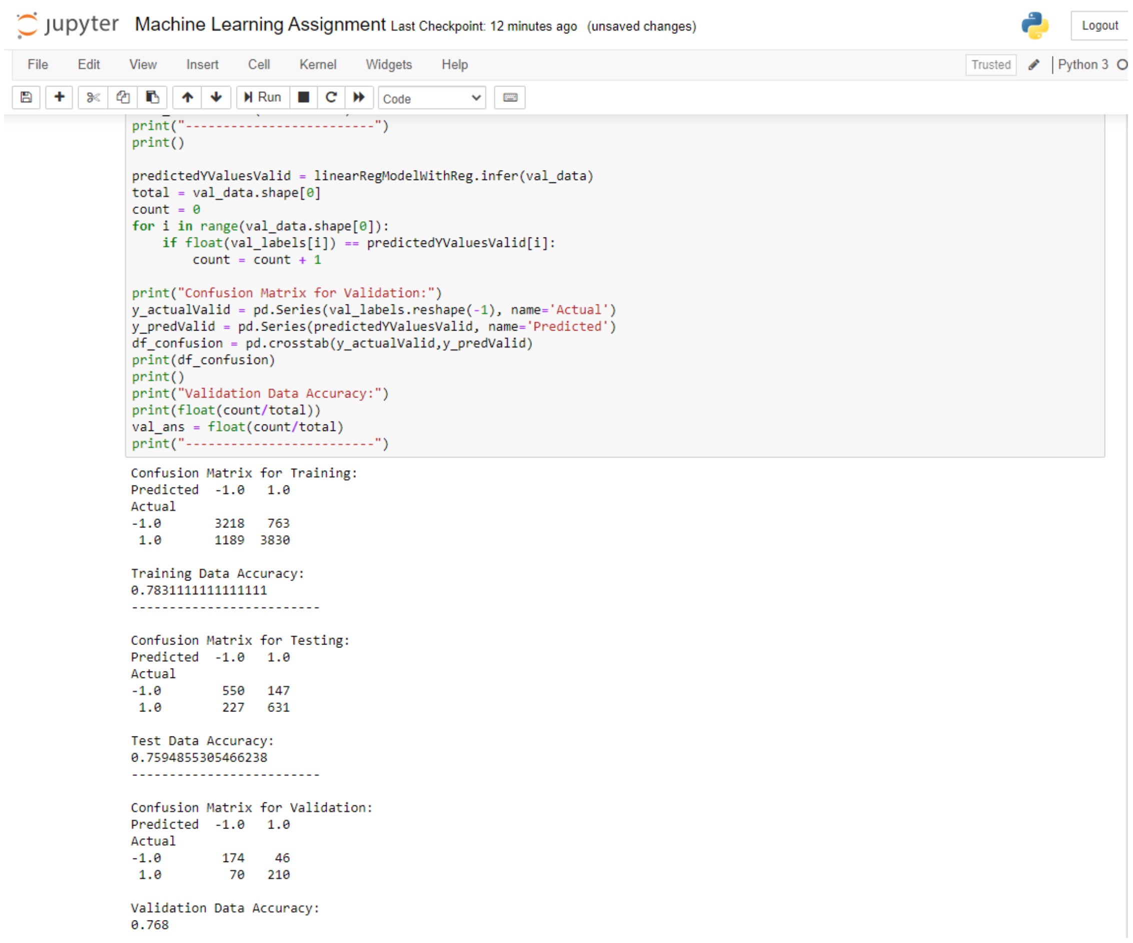
```
Lambda
1

[0.11182046 0.07832727 0.50572154 0.75377472 0.114265   0.2359039
 0.60262863 0.15928931 0.76416243 0.77031879 0.89050019 0.76020495
 0.98683619 0.36093583 0.93620055 0.84666576 0.36290805 0.42990983
 0.20371398 0.52908582 0.10902225 0.93866947 0.20871139 0.20121439
 0.25224911 0.05974206 0.04045042 0.14612015 0.19526784 0.99576548
 0.54328983]

Calculated thetas using Gradient Descent:
[-8.41443758e-03  3.04276853e-02  8.54196150e-02  1.86397993e-01
 -9.08050939e-02  2.04836615e-01  2.19518507e-01  1.29017601e-01
  3.23381114e-01  6.75353260e-02  1.57498289e-01  1.82828037e-01
  3.61533856e-01  2.24758688e-01  4.03507917e-01  3.64494539e-01
 -1.08968501e-01  2.19132809e-02  9.69288438e-02  1.30780156e-02
 -1.09572311e-01  1.54416184e-01 -1.16096729e-01  4.96489190e-02
 -2.37969321e-04  5.42695927e-02  5.37244468e-02 -2.91628526e-03
  7.01571232e-02  2.65373931e-01  2.72040098e-01]

Actual labels of training data
[ 1.  1. -1. ...  1. -1. -1.]

Predicted values of training data
[ 1.78912249  1.0850533  -0.45821529 ...  0.5715494  -0.16051775
 -0.89115353]

Mean that will be used as decision boundary
0.028410287758557695

After putting predicted continuous values through decision boundary
[ 1.  1. -1. ...  1. -1. -1.]
```

Now we compute the training, testing and validation errors:

In [21]:
```
total = train_data.shape[0]
count = 0
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
```


jupyter
Machine Learning Assignment Last Checkpoint: 12 minutes ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code

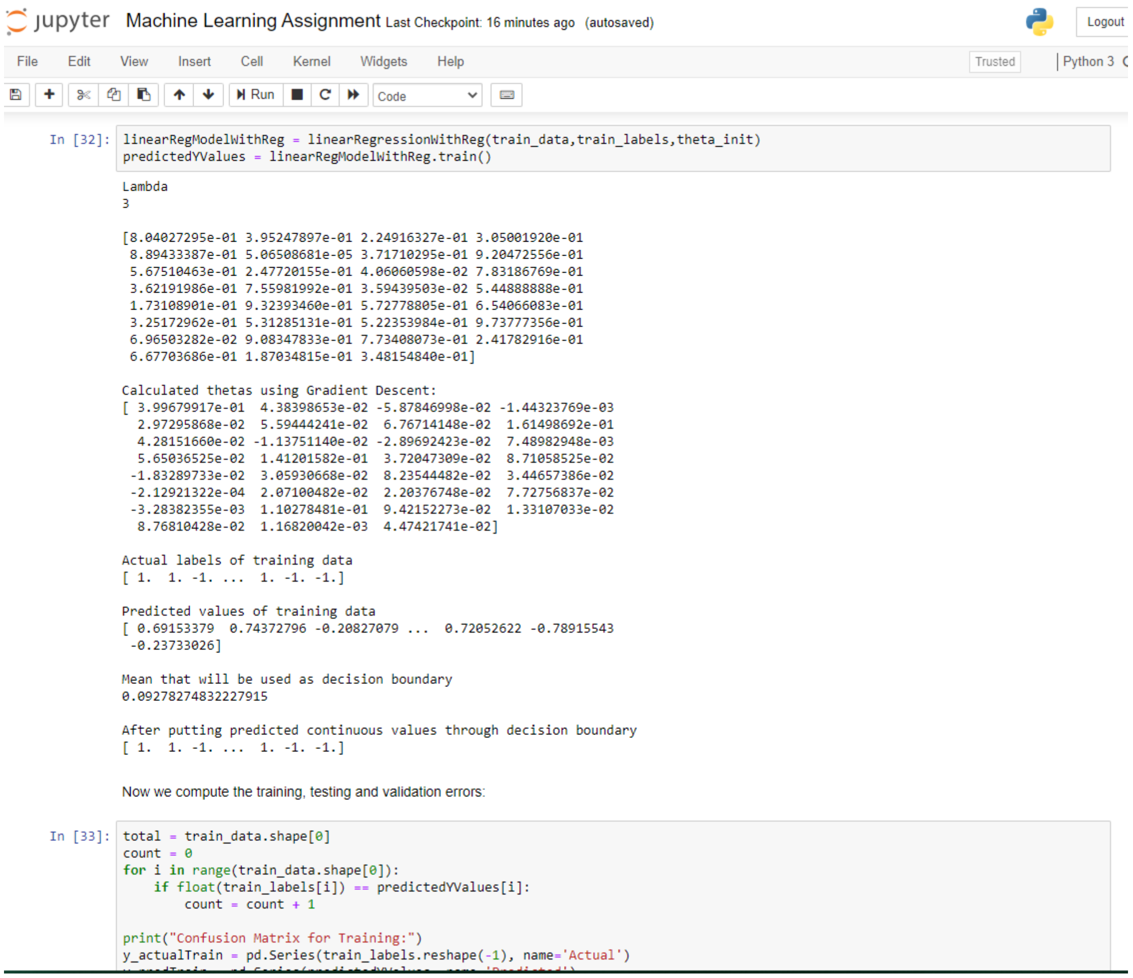
```
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
print()
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("--------------------------")
print()

predictedYValuesTest = linearRegModelWithReg.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print(df_confusion)
print()
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("--------------------------")
print()

predictedYValuesValid = linearRegModelWithReg.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
```


Jupyter Machine Learning Assignment Last Checkpoint: 12 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
print("------------------------")
print()

predictedYValuesValid = linearRegModelWithReg.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       3218   763
 1.0       1189  3830

Training Data Accuracy:
0.7831111111111111
------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        550   147
 1.0        227   631

Test Data Accuracy:
0.7594855305466238
------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0        174    46
 1.0         70   210

Validation Data Accuracy:
0.768
```

The following screenshots are an example of what accuracies we got when the made the lambda value set to 3:

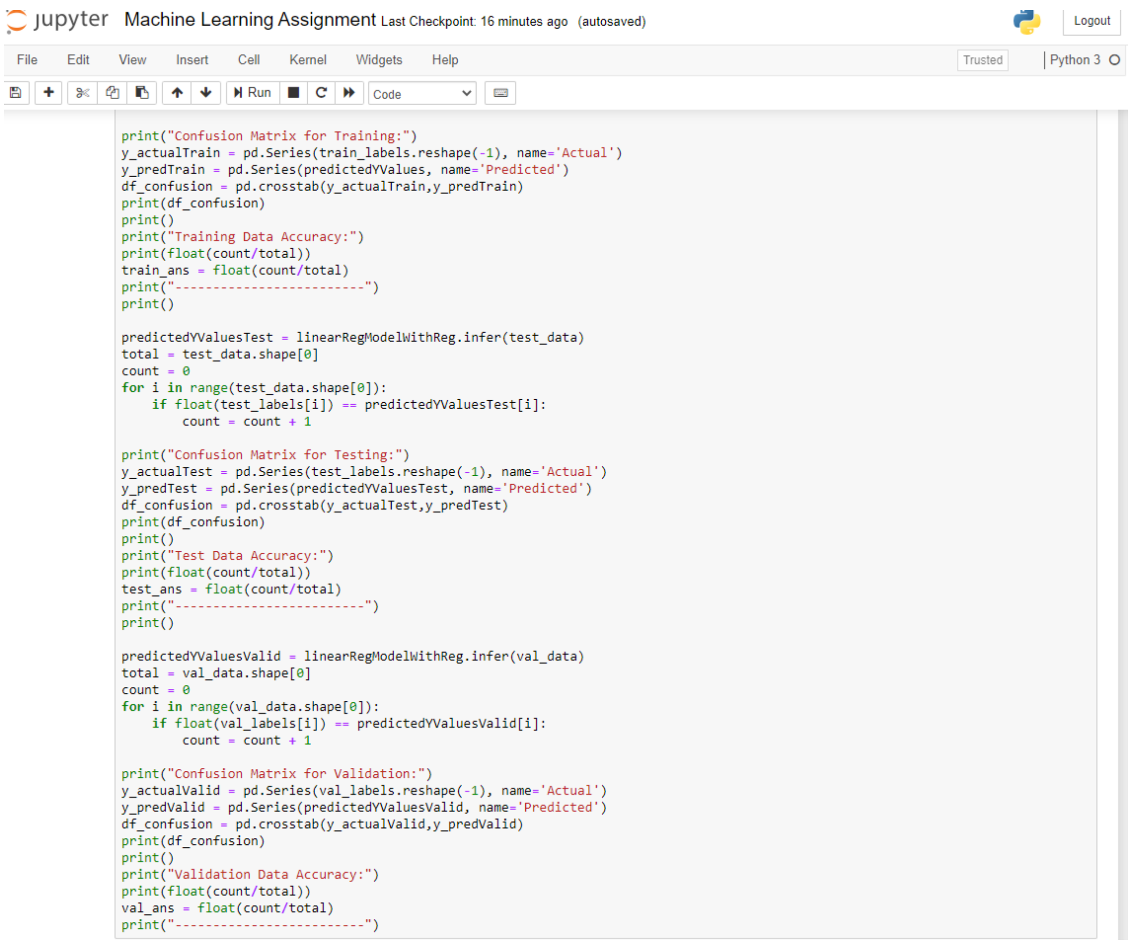

jupyter
Machine Learning Assignment Last Checkpoint: 16 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [32]: linearRegModelWithReg = linearRegressionWithReg(train_data,train_labels,theta_init)
         predictedYValues = linearRegModelWithReg.train()
```

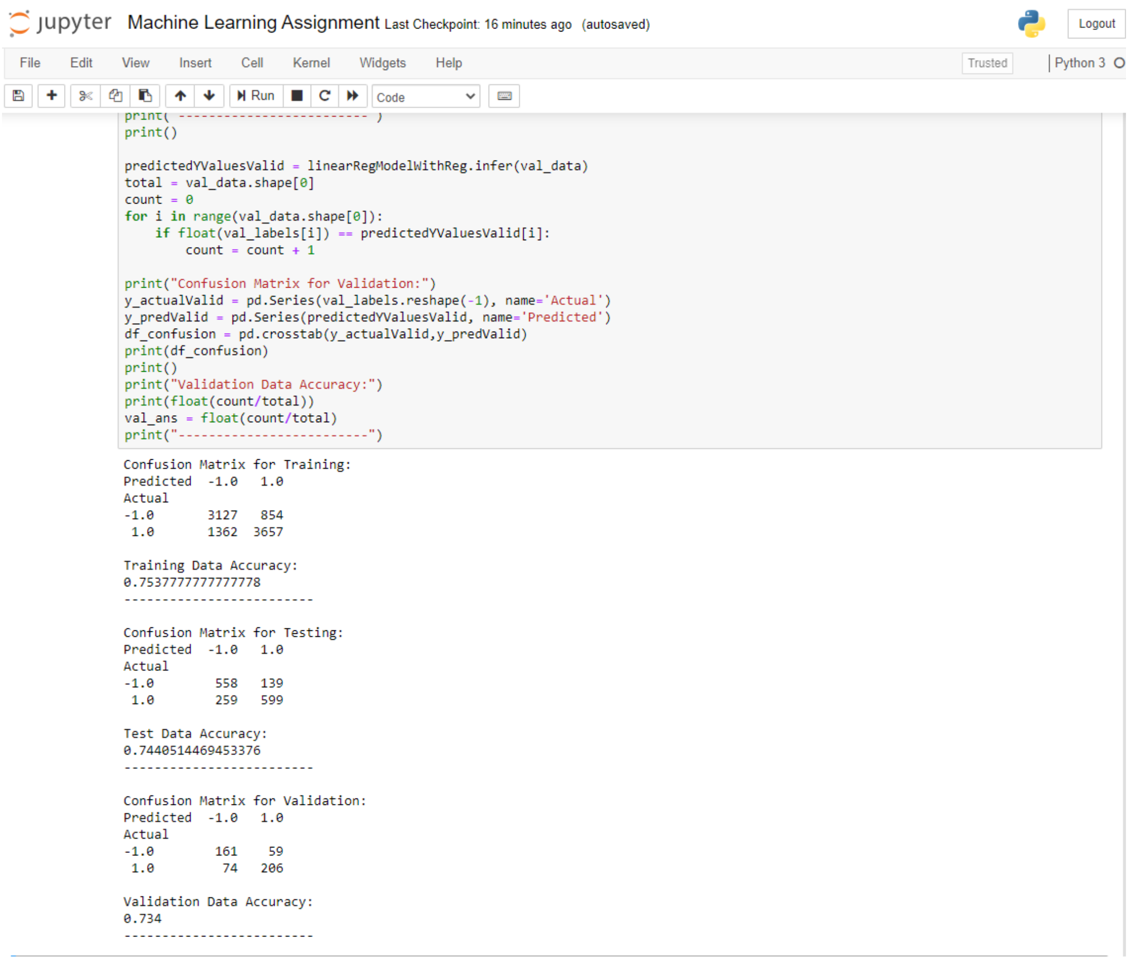
```
Lambda
3

[8.04027295e-01 3.95247897e-01 2.24916327e-01 3.05001920e-01
 8.89433387e-01 5.06508681e-05 3.71710295e-01 9.20472556e-01
 5.67510463e-01 2.47720155e-01 4.06060598e-02 7.83186769e-01
 3.62191986e-01 7.55981992e-01 3.59439503e-02 5.44888888e-01
 1.73108901e-01 9.32393460e-01 5.72778805e-01 6.54066083e-01
 3.25172962e-01 5.31285131e-01 5.22353984e-01 9.73777356e-01
 6.96503282e-02 9.08347833e-01 7.73408073e-01 2.41782916e-01
 6.67703686e-01 1.87034815e-01 3.48154840e-01]

Calculated thetas using Gradient Descent:
[ 3.99679917e-01  4.38398653e-02 -5.87846998e-02 -1.44323769e-03
  2.97295868e-02  5.59444241e-02  6.76714148e-02  1.61498692e-01
  4.28151660e-02 -1.13751140e-02 -2.89692423e-02  7.48982948e-03
  5.65036525e-02  1.41201582e-01  3.72047309e-02  8.71058525e-02
 -1.83289733e-02  3.05930668e-02  8.23544482e-02  3.44657386e-02
 -2.12921322e-04  2.07100482e-02  2.20376748e-02  7.72756837e-02
 -3.28382355e-03  1.10278481e-01  9.42152273e-02  1.33107033e-02
  8.76810428e-02  1.16820042e-03  4.47421741e-02]

Actual labels of training data
[ 1.  1. -1. ...  1. -1. -1.]

Predicted values of training data
[ 0.69153379  0.74372796 -0.20827079 ...  0.72052622 -0.78915543
 -0.23733026]

Mean that will be used as decision boundary
0.09278274832227915

After putting predicted continuous values through decision boundary
[ 1.  1. -1. ...  1. -1. -1.]
```

Now we compute the training, testing and validation errors:

```
In [33]: total = train_data.shape[0]
         count = 0
         for i in range(train_data.shape[0]):
             if float(train_labels[i]) == predictedYValues[i]:
                 count = count + 1

         print("Confusion Matrix for Training:")
         y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
```


jupyter Machine Learning Assignment Last Checkpoint: 16 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
print()
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("-------------------------")
print()

predictedYValuesTest = linearRegModelWithReg.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print(df_confusion)
print()
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("-------------------------")
print()

predictedYValuesValid = linearRegModelWithReg.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("-------------------------")
```


jupyter Machine Learning Assignment Last Checkpoint: 16 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
print("------------------------")
print()

predictedYValuesValid = linearRegModelWithReg.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0        3127   854
 1.0        1362  3657

Training Data Accuracy:
0.7537777777777778
------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0         558   139
 1.0         259   599

Test Data Accuracy:
0.7440514469453376
------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0         161    59
 1.0          74   206

Validation Data Accuracy:
0.734
------------------------
```

Code we will use to train model and infer on the other sets with lambda set to 2:

[19]:
```
class linearRegressionWithReg:
    def __init__(self,train_data,train_labels,theta_init):
        self.train_data = train_data
        self.train_labels = train_labels
        self.dMatrixTrain = np.zeros((train_data.shape[0],train_data.
 ↪shape[1]+1))
        self.dMatrixTrain[:,0]=1
        i = 1
        for i in range(train_data.shape[1]):
            self.dMatrixTrain[:,i]=train_data[:,i]

        self.y_vals = train_labels[:].reshape(-1)
        self.theta_pred_GD = theta_init.copy()
        self.theta_diff = np.zeros(train_data.shape[1]+1)
        self.descent_vals = np.dot(self.dMatrixTrain,self.theta_pred_GD)
        self.meanTrain = 0

    def train(self):
        alpha = 0.00001
        iterations = 1
        lambdaVal = 2
        print("Lambda")
        print(lambdaVal)
        print()
        i=0
        j=0
        print(self.theta_pred_GD)
        print()
        while(np.linalg.norm(self.theta_diff,2)>0.00001) or (i==0 and␣
↪iterations==1):
            #for each theta
            for j in range(self.train_data.shape[1]+1):
                if(j == 0):
                    gradi = (self.descent_vals[i]- self.y_vals[i])*self.
↪dMatrixTrain[i][j]
                    self.theta_diff[j] = abs(self.
↪theta_pred_GD[j]-alpha*gradi-self.theta_pred_GD[j])
                    self.theta_pred_GD[j] = self.theta_pred_GD[j]-alpha*gradi
                    self.descent_vals = np.asarray(self.dMatrixTrain.dot(self.
↪theta_pred_GD)).reshape(-1)
                else:
                    gradi = (self.descent_vals[i]- self.y_vals[i])*self.
↪dMatrixTrain[i][j] + lambdaVal*self.theta_pred_GD[j]
                    self.theta_diff[j] = abs(self.
↪theta_pred_GD[j]-alpha*gradi-self.theta_pred_GD[j])
                    self.theta_pred_GD[j] = self.theta_pred_GD[j]-alpha*gradi
                    self.descent_vals = np.asarray(self.dMatrixTrain.dot(self.
↪theta_pred_GD)).reshape(-1)

            i = i + 1

            if i==self.train_data.shape[0]:
                i=0
            iterations += 1

        print("Calculated thetas using Gradient Descent:")
        print(self.theta_pred_GD)
        self.descent_vals = np.asarray(self.dMatrixTrain.dot(self.
↪theta_pred_GD)).reshape(-1)

        print()
        print("Actual labels of training data")
        print(self.y_vals)

        print()
        print("Predicted values of training data")
        print(self.descent_vals)

        print()
        print("Mean that will be used as decision boundary")
        self.meanTrain = np.sum(np.array(self.descent_vals))/self.train_data.
 ↪shape[0]
        print(self.meanTrain)
        predictedYValues = []
        for j in range(self.train_data.shape[0]):
            if(self.descent_vals[j] >= self.meanTrain):
                predictedYValues.append(1.)
            else:
                predictedYValues.append(-1.)
        print()
        print("After putting predicted continuous values through decision␣
 ↪boundary")
        print(np.array(predictedYValues))
        return predictedYValues

    def infer(self,input):
        dMatrixOther = np.zeros((input.shape[0],input.shape[1]+1))
        dMatrixOther[:,0]=1
        i = 1
        for i in range(input.shape[1]):
            dMatrixOther[:,i]=input[:,i]

        descent_valsOther = np.dot(dMatrixOther,self.theta_pred_GD)

        predictedYValuesOther = []
        for j in range(input.shape[0]):
            if(descent_valsOther[j] >= self.meanTrain):
                predictedYValuesOther.append(1.)
            else:
                predictedYValuesOther.append(-1.)
        return predictedYValuesOther
```

Now we train the model using the code above:

[20]:
```
linearRegModelWithReg =␣
 ↪linearRegressionWithReg(train_data,train_labels,theta_init)
predictedYValues = linearRegModelWithReg.train()
```

```
Lambda
2

[0.03052677 0.63704864 0.32018559 0.12463098 0.02304814 0.55619361
```

```
 0.51792906 0.50477338 0.23232713 0.9863715  0.82262516 0.99414685
 0.39898618 0.34300795 0.02986272 0.73676063 0.86461745 0.66765734
 0.24684599 0.42157647 0.2576714  0.35773843 0.17056024 0.93254676
 0.12962183 0.94627489 0.64728959 0.37841903 0.24243509 0.48946095
 0.75335357]

Calculated thetas using Gradient Descent:
[-0.01691928  0.0826419   0.00498927 -0.0202058  -0.05495603  0.15920782
  0.10697827  0.16811167  0.04218915  0.10374645  0.07542349  0.14524168
  0.06716245  0.13082416  0.04663652  0.14932453  0.06715434  0.06877448
  0.05779631  0.00329444  0.00050339 -0.0417233  -0.04854172  0.11805689
 -0.023189    0.18987253  0.10996853  0.06032932  0.04999007  0.037681
  0.18028165]

Actual labels of training data
[ 1. -1. -1. … -1.  1.  1.]

Predicted values of training data
[ 0.18928947 -0.12174987 -0.54149071 … -0.36818292 -0.01525683
  0.3380913 ]

Mean that will be used as decision boundary
0.059793687385437506

After putting predicted continuous values through decision boundary
[ 1. -1. -1. … -1. -1.  1.]
```

Now we compute the training, testing and validation errors:

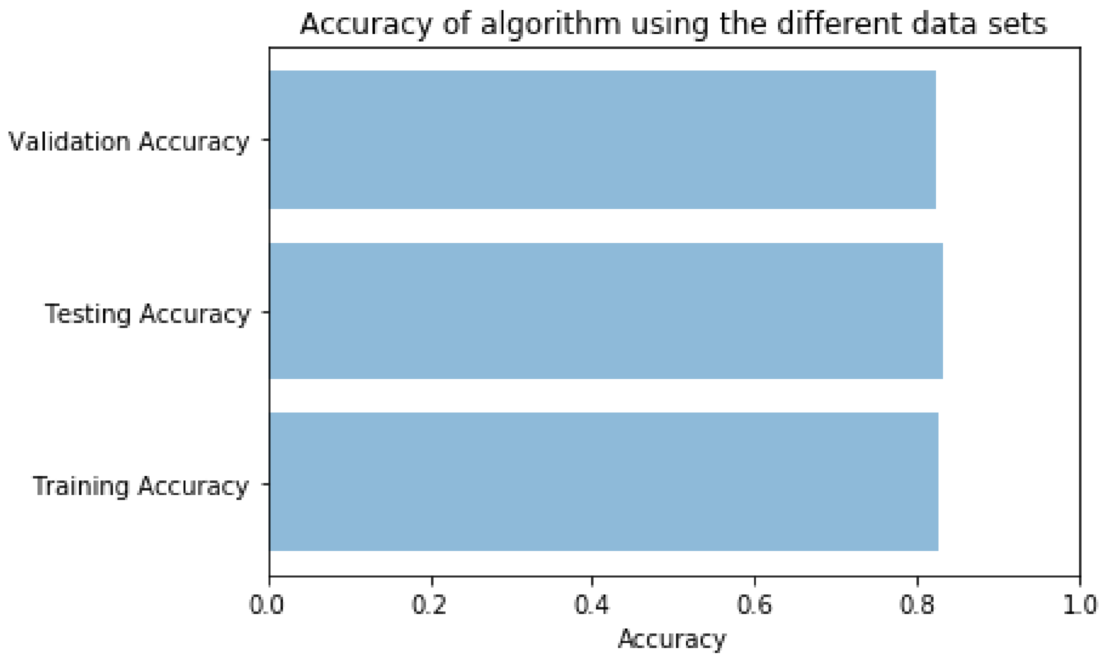
[21]:
```python
total = train_data.shape[0]
count = 0
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print(df_confusion)
print()
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("--------------------------")
print()
LinReg_Train = train_ans
```

```python
predictedYValuesTest = linearRegModelWithReg.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print(df_confusion)
print()
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("-------------------------")
print()
LinReg_Test = test_ans

predictedYValuesValid = linearRegModelWithReg.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print(df_confusion)
print()
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("-------------------------")
LinReg_Val = val_ans
```

```
Confusion Matrix for Training:
Predicted  -1.0   1.0
Actual
-1.0       3404   571
 1.0        996  4029

Training Data Accuracy:
0.8258888888888889
```

```
-------------------------

Confusion Matrix for Testing:
Predicted  -1.0   1.0
Actual
-1.0        610    99
 1.0        164   682

Test Data Accuracy:
0.8308681672025724
-------------------------

Confusion Matrix for Validation:
Predicted  -1.0   1.0
Actual
-1.0        186    28
 1.0         60   226

Validation Data Accuracy:
0.824
-------------------------
```

[22]:
```python
objects = ('Training Accuracy', 'Testing Accuracy', 'Validation Accuracy')
y_pos = np.arange(3)
performance = [train_ans,test_ans,val_ans]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithm using the different data sets')

mp.show()
```



From the results, we can clearly see that this updated model shows an increase in performance on this dataset and gets around 82% of predictions correct; this makes it our second best algorithm at present. We also observe that regularisation has a major affect on the linear regression algorithm for this dataset with an approximate 10% increase in correct predictions. This algorithm is one of the most consistent that we have found as the Training,Testing and Validation data are within 2% of each other. This algorithm could be improved by using more complex basis functions or finding a better solution to the decision boundary and we would highly recommend making these changes should someone want to use this algorithm and dataset.

End Of Linear Regression Classifier With Regularisation

Logistic Regression Classifier

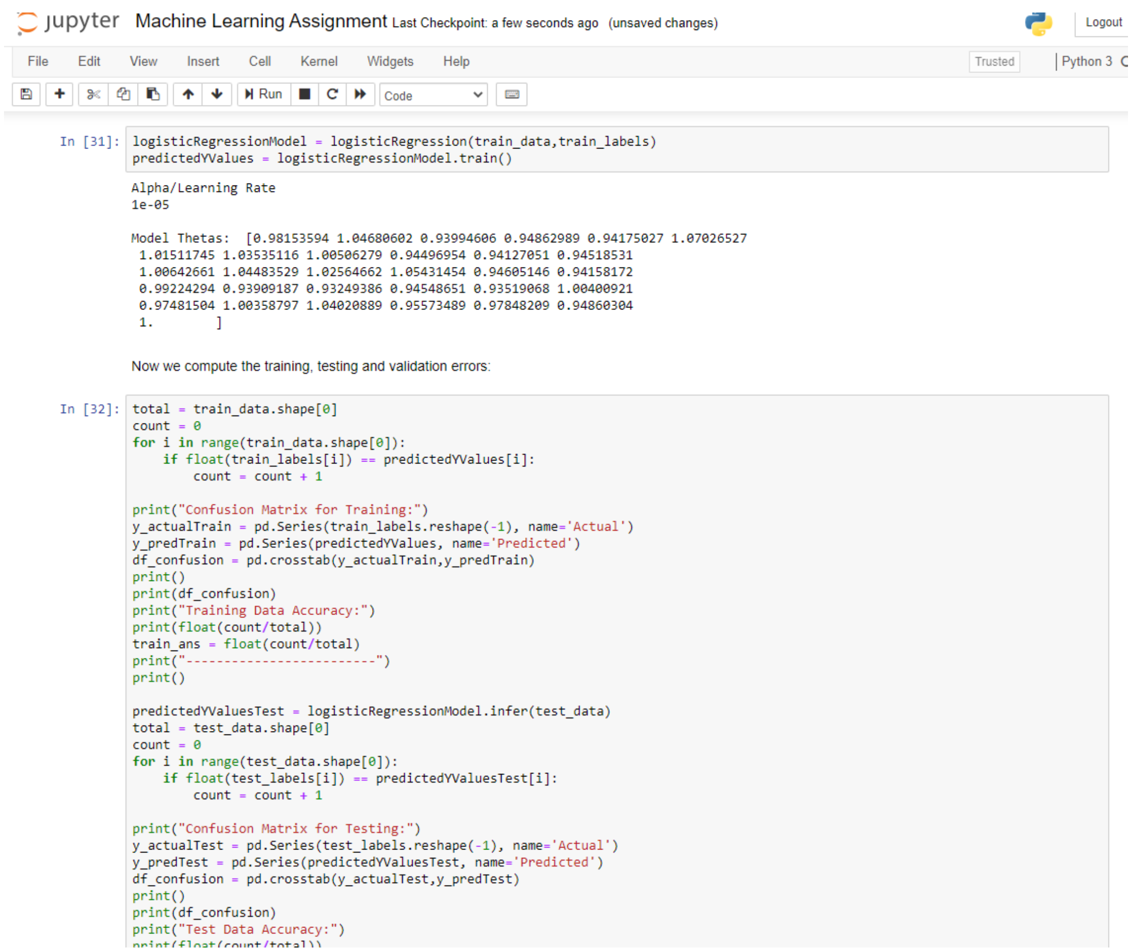
For this classification method we used the normal logistic regression via gradient descent without regularisation. We chose this algorithm as we were wondering how much better the Logistic Regression algorithm would compare to the Linear Regression algorithm implemented earlier.

We used the same basis functions and design matrices as the algorithm above and initialised all thetas/weights to 1.

During the tuning of hyperparameters, we found that a learning rate/alpha of value 0.0001 yields the greatest accuracy. Using a smaller or larger learning rate decreased the accuracy would yield vastly less accurate results. Please see the cells below for proof of these claims.

The training of the algorithm (gradient descent) runs until the euclidean norm of the theta values < 0.0005 which indicated convergence.

The following screenshots are an example of what accuracies we got when the made the alpha value smaller than 0.0001:

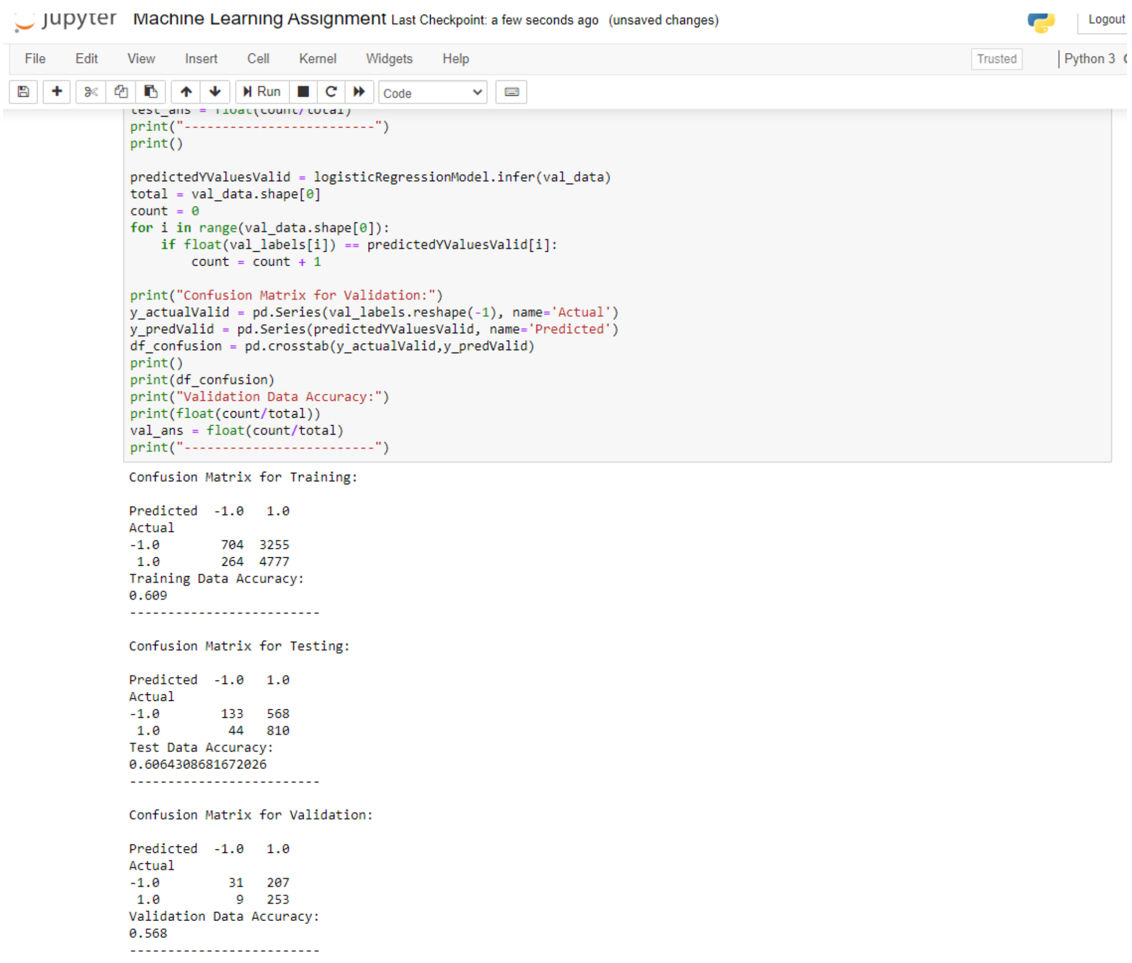

jupyter Machine Learning Assignment Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

In [31]:
```
logisticRegressionModel = logisticRegression(train_data,train_labels)
predictedYValues = logisticRegressionModel.train()
```

```
Alpha/Learning Rate
1e-05

Model Thetas:  [0.98153594 1.04680602 0.93994606 0.94862989 0.94175027 1.07026527
 1.01511745 1.03535116 1.00506279 0.94496954 0.94127051 0.94518531
 1.00642661 1.04483529 1.02564662 1.05431454 0.94605146 0.94158172
 0.99224294 0.93909187 0.93249386 0.94548651 0.93519068 1.00400921
 0.97481504 1.00358797 1.04020889 0.95573489 0.97848209 0.94860304
 1.        ]
```

Now we compute the training, testing and validation errors:

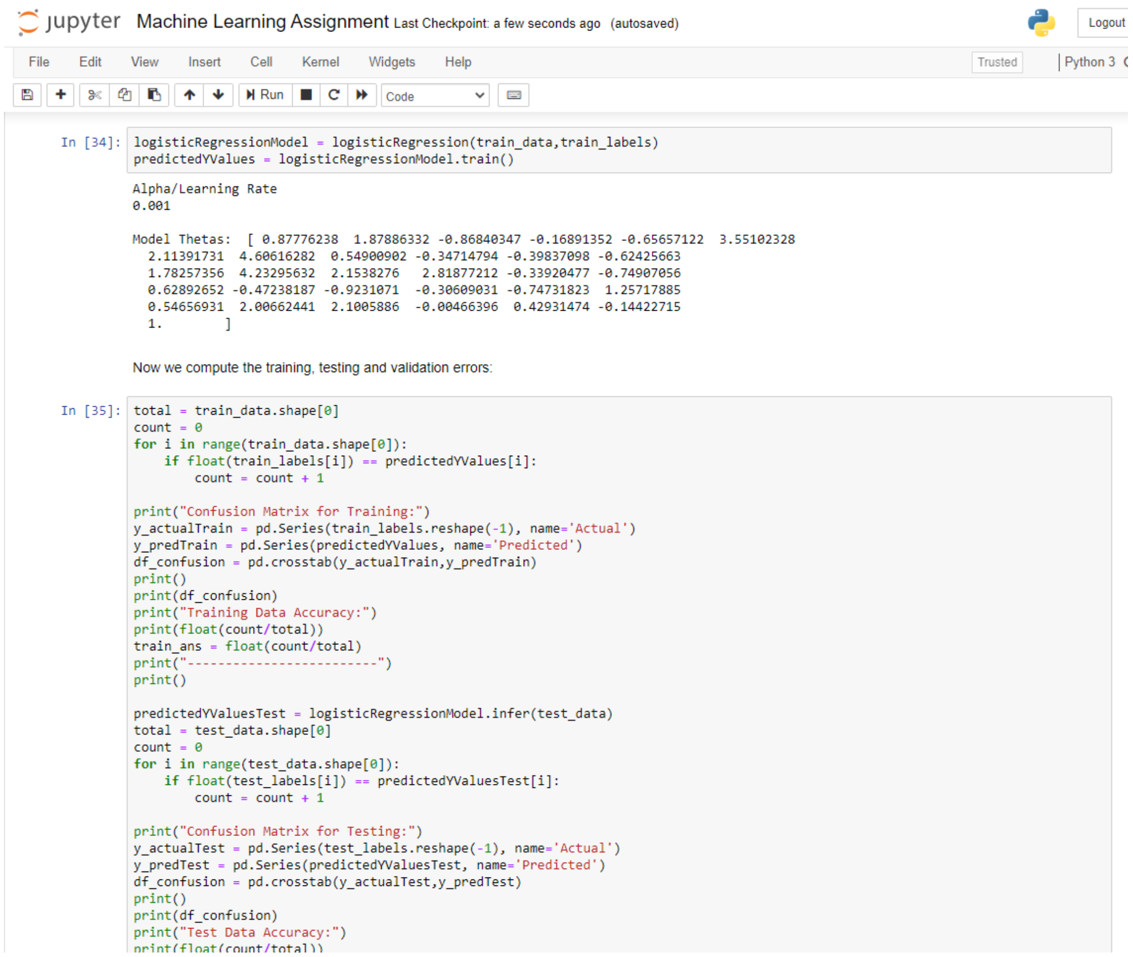
In [32]:
```
total = train_data.shape[0]
count = 0
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print()
print(df_confusion)
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("-------------------------")
print()

predictedYValuesTest = logisticRegressionModel.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print()
print(df_confusion)
print("Test Data Accuracy:")
print(float(count/total))
```


Jupyter Machine Learning Assignment Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
print("------------------------")
print()

predictedYValuesValid = logisticRegressionModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print()
print(df_confusion)
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```

```
Confusion Matrix for Training:

Predicted  -1.0   1.0
Actual
-1.0        704  3255
 1.0        264  4777
Training Data Accuracy:
0.609
------------------------

Confusion Matrix for Testing:

Predicted  -1.0   1.0
Actual
-1.0        133   568
 1.0         44   810
Test Data Accuracy:
0.6064308681672026
------------------------

Confusion Matrix for Validation:

Predicted  -1.0   1.0
Actual
-1.0         31   207
 1.0          9   253
Validation Data Accuracy:
0.568
------------------------
```

The following screenshots are an example of what accuracies we got when the made the alpha value bigger than 0.0001:

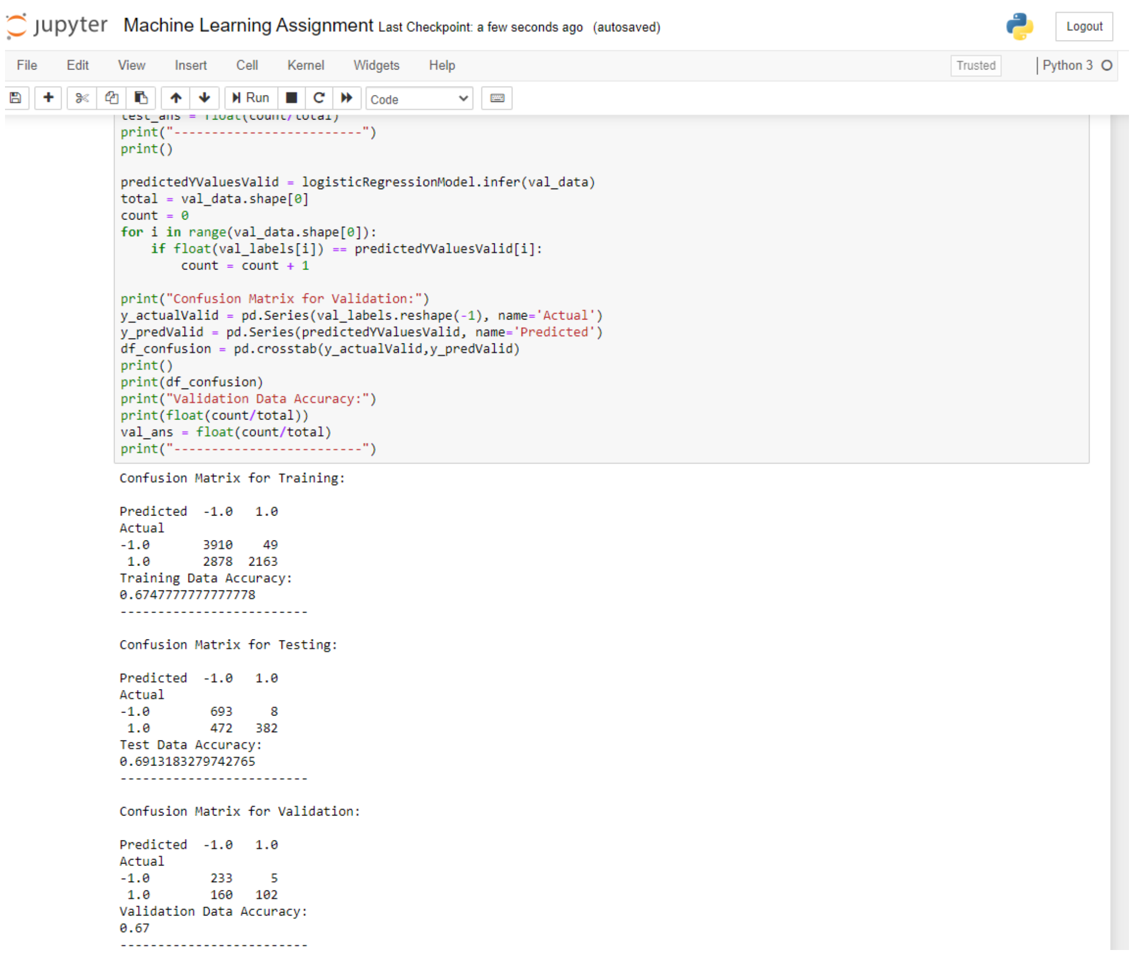

jupyter
Machine Learning Assignment Last Checkpoint: a few seconds ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [34]: logisticRegressionModel = logisticRegression(train_data,train_labels)
         predictedYValues = logisticRegressionModel.train()
```

```
Alpha/Learning Rate
0.001

Model Thetas:  [ 0.87776238  1.87886332 -0.86840347 -0.16891352 -0.65657122  3.55102328
  2.11391731  4.60616282  0.54900902 -0.34714794 -0.39837098 -0.62425663
  1.78257356  4.23295632  2.1538276   2.81877212 -0.33920477 -0.74907056
  0.62892652 -0.47238187 -0.9231071  -0.30609031 -0.74731823  1.25717885
  0.54656931  2.00662441  2.1005886  -0.00466396  0.42931474 -0.14422715
  1.        ]
```

Now we compute the training, testing and validation errors:

```
In [35]: total = train_data.shape[0]
         count = 0
         for i in range(train_data.shape[0]):
             if float(train_labels[i]) == predictedYValues[i]:
                 count = count + 1

         print("Confusion Matrix for Training:")
         y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
         y_predTrain = pd.Series(predictedYValues, name='Predicted')
         df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
         print()
         print(df_confusion)
         print("Training Data Accuracy:")
         print(float(count/total))
         train_ans = float(count/total)
         print("--------------------------")
         print()

         predictedYValuesTest = logisticRegressionModel.infer(test_data)
         total = test_data.shape[0]
         count = 0
         for i in range(test_data.shape[0]):
             if float(test_labels[i]) == predictedYValuesTest[i]:
                 count = count + 1

         print("Confusion Matrix for Testing:")
         y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
         y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
         df_confusion = pd.crosstab(y_actualTest,y_predTest)
         print()
         print(df_confusion)
         print("Test Data Accuracy:")
         print(float(count/total))
```


jupyter Machine Learning Assignment Last Checkpoint: a few seconds ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
test_ans = float(count/total)
print("------------------------")
print()

predictedYValuesValid = logisticRegressionModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print()
print(df_confusion)
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("------------------------")
```

```
Confusion Matrix for Training:

Predicted  -1.0   1.0
Actual
-1.0        3910    49
 1.0        2878  2163
Training Data Accuracy:
0.6747777777777778
------------------------

Confusion Matrix for Testing:

Predicted  -1.0   1.0
Actual
-1.0         693     8
 1.0         472   382
Test Data Accuracy:
0.6913183279742765
------------------------

Confusion Matrix for Validation:

Predicted  -1.0   1.0
Actual
-1.0         233     5
 1.0         160   102
Validation Data Accuracy:
0.67
------------------------
```

Code we will use to train model and infer on the other sets:

[23]:
```
class logisticRegression:
    def __init__(self,train_data,train_labels):
        self.train_data = train_data
        self.train_labels = train_labels
        self.dMatrixTrain = np.zeros((train_data.shape[0],train_data.
 ↪shape[1]+1))
        self.dMatrixTrain[:,0]=1
        i = 1
        for i in range(train_data.shape[1]):
            self.dMatrixTrain[:,i]=train_data[:,i]

        self.y_vals = train_labels[:].reshape(-1)
        self.theta_pred_GD = np.ones(train_data.shape[1]+1)
        self.theta_old = np.zeros(train_data.shape[1]+1)

    def h(self,x,theta): # Regression function
        return 1/(1+np.exp(-np.dot(x, theta)))

    def train(self):
          = 1e-4 # define our learning rate
        print("Alpha/Learning Rate")
        print( )
        print()

        while np.sqrt(np.sum(np.power(self.theta_pred_GD - self.theta_old, 2)))␣
↪> 0.0005: # while euclidean norm > 0.0005 (so   = 0.0005)
            self.theta_old = self.theta_pred_GD # set old parameter values to␣
↪parameter values before they are updated
            for i in range(self.dMatrixTrain.shape[0]): # loop over each row of␣
↪the design matrix (each data point)
                for j in range(train_data.shape[1]+1):
                    self.theta_pred_GD[j] = self.theta_pred_GD[j] -  *( (self.
↪h(self.dMatrixTrain[i,:], self.theta_pred_GD) - self.y_vals[i]) * self.
↪dMatrixTrain[i][j] ) # update the parameters using the update rule
        print("Model Thetas: ", self.theta_pred_GD) # Print model parameters␣
↪after convergence
        model_predictionsTrain = self.h(self.dMatrixTrain, self.theta_pred_GD)

        predictedYValues = []
        for j in range(train_data.shape[0]):
            if(model_predictionsTrain[j] >= 0.5):
                predictedYValues.append(1.)
            else:
                predictedYValues.append(-1.)
        return predictedYValues

    def infer(self,input):
        dMatrixOther = np.zeros((input.shape[0],input.shape[1]+1))
        dMatrixOther[:,0]=1
        i = 1
        for i in range(input.shape[1]):
            dMatrixOther[:,i]=input[:,i]

        model_predictionsOther = self.h(dMatrixOther,self.theta_pred_GD)

        predictedYValuesOther = []
        for j in range(dMatrixOther.shape[0]):
            if(model_predictionsOther[j] >= 0.5):
                predictedYValuesOther.append(1.)
            else:
                predictedYValuesOther.append(-1.)
        return predictedYValuesOther
```

Now we train the model using the code above:

```
[24]: logisticRegressionModel = logisticRegression(train_data,train_labels)
      predictedYValues = logisticRegressionModel.train()
```

```
Alpha/Learning Rate
0.0001

Model Thetas:  [0.83646598 1.40700803 0.44903819 0.52895585 0.46487743
1.63767246
 1.12267273 1.326608   1.02827793 0.50145784 0.4669962  0.4961027
 1.04760529 1.40868414 1.23247305 1.48389094 0.50884471 0.46482946
 0.93189994 0.45527304 0.39154054 0.50806028 0.41886181 1.01291822
 0.75797378 1.02676416 1.35304882 0.59268716 0.80138454 0.53122142
 1.        ]
```

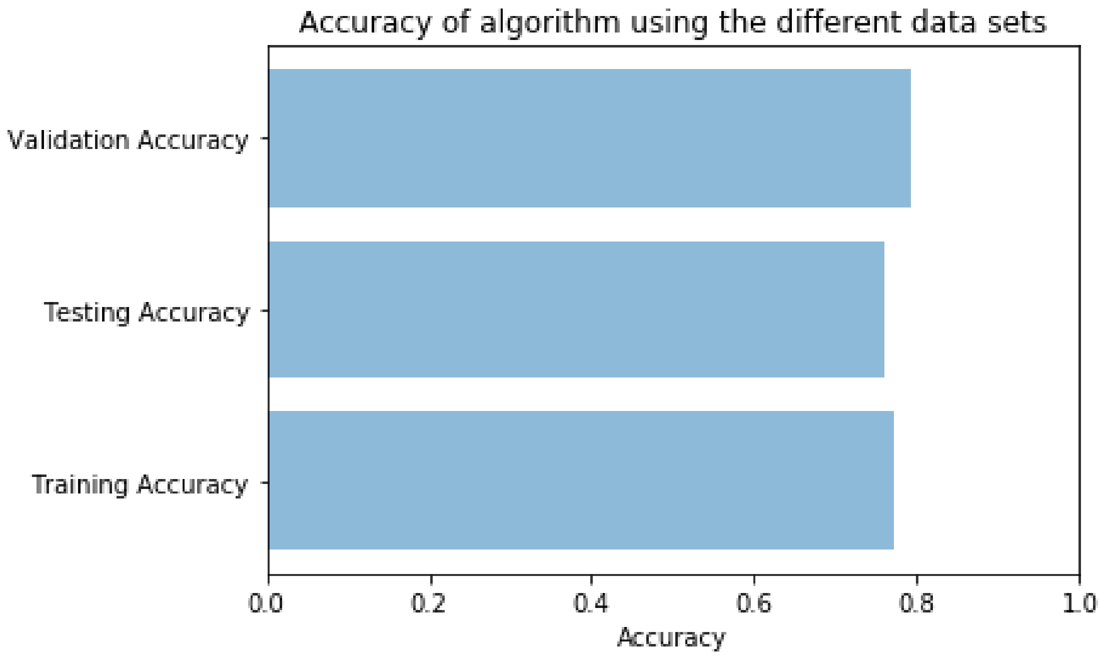
Now we compute the training, testing and validation errors:

```
[25]: total = train_data.shape[0]
count = 0
for i in range(train_data.shape[0]):
    if float(train_labels[i]) == predictedYValues[i]:
        count = count + 1

print("Confusion Matrix for Training:")
y_actualTrain = pd.Series(train_labels.reshape(-1), name='Actual')
y_predTrain = pd.Series(predictedYValues, name='Predicted')
df_confusion = pd.crosstab(y_actualTrain,y_predTrain)
print()
print(df_confusion)
print("Training Data Accuracy:")
print(float(count/total))
train_ans = float(count/total)
print("--------------------------")
print()
log_Train = train_ans

predictedYValuesTest = logisticRegressionModel.infer(test_data)
total = test_data.shape[0]
count = 0
for i in range(test_data.shape[0]):
    if float(test_labels[i]) == predictedYValuesTest[i]:
        count = count + 1

print("Confusion Matrix for Testing:")
y_actualTest = pd.Series(test_labels.reshape(-1), name='Actual')
y_predTest = pd.Series(predictedYValuesTest, name='Predicted')
df_confusion = pd.crosstab(y_actualTest,y_predTest)
print()
print(df_confusion)
print("Test Data Accuracy:")
print(float(count/total))
test_ans = float(count/total)
print("-------------------------")
print()
log_Test = test_ans

predictedYValuesValid = logisticRegressionModel.infer(val_data)
total = val_data.shape[0]
count = 0
for i in range(val_data.shape[0]):
    if float(val_labels[i]) == predictedYValuesValid[i]:
        count = count + 1

print("Confusion Matrix for Validation:")
y_actualValid = pd.Series(val_labels.reshape(-1), name='Actual')
y_predValid = pd.Series(predictedYValuesValid, name='Predicted')
df_confusion = pd.crosstab(y_actualValid,y_predValid)
print()
print(df_confusion)
print("Validation Data Accuracy:")
print(float(count/total))
val_ans = float(count/total)
print("-------------------------")
log_Val = val_ans
```

```
Confusion Matrix for Training:

Predicted  -1.0   1.0
Actual
-1.0       2431  1544
 1.0        506  4519
Training Data Accuracy:
0.7722222222222223
-------------------------

Confusion Matrix for Testing:

Predicted  -1.0   1.0
Actual
-1.0        424   285
 1.0         88   758
Test Data Accuracy:
0.760128617363344
-------------------------

Confusion Matrix for Validation:
```

```
Predicted  -1.0   1.0
Actual
-1.0        140    74
 1.0         30   256
Validation Data Accuracy:
0.792
-------------------------
```

[26]:
```
objects = ('Training Accuracy', 'Testing Accuracy', 'Validation Accuracy')
y_pos = np.arange(3)
performance = [train_ans,test_ans,val_ans]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithm using the different data sets')

mp.show()
```



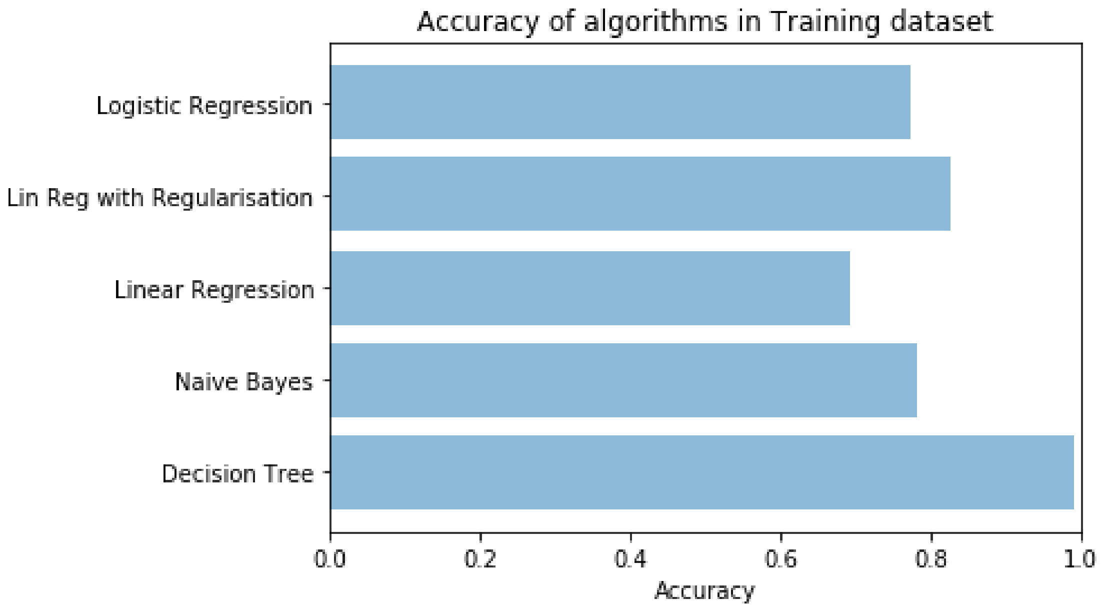
From the results, we can clearly see that using logistic regression yields marginally better results than the linear regression algorithm without regularisation. The algorithm makes more than 77% of the predictions correctly and thus is a decent classifier. This algorithm is pretty consistent amongst the Training, Testing and Validation datasets as their accuracies are fairly close. If someone wanted to use this algorithm and dataset I would recommend maybe putting more complex basis functions and tweaking the alpha values simultaneously in order to yield better results.

End Of Logistic Regression Classifier

4) Summary and discussion of results

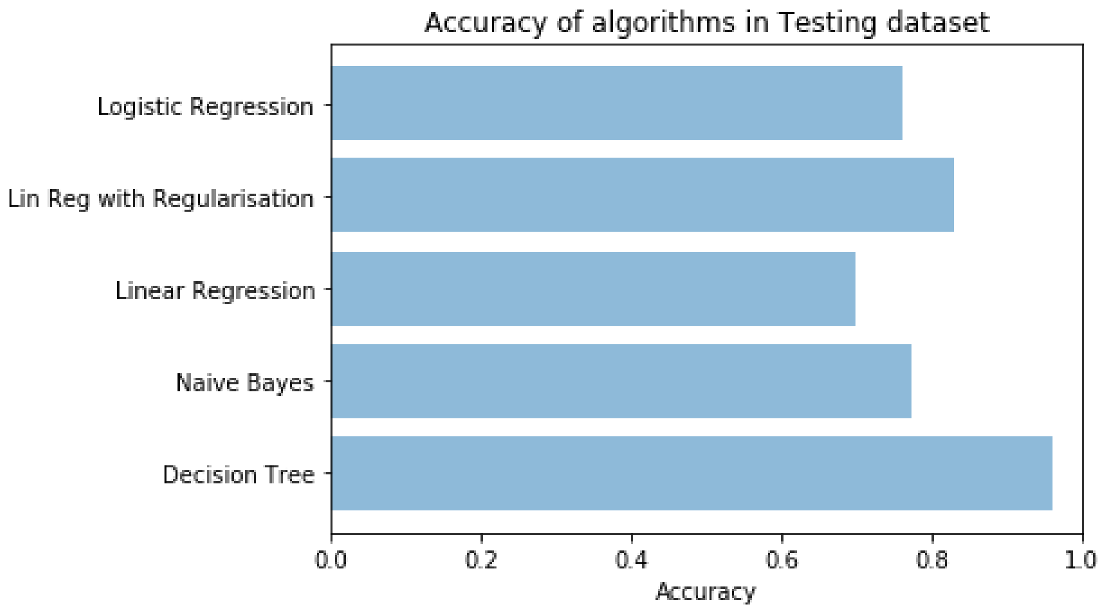
Let us summarise by showing the accurcies of the different methods on the Training dataset:

[27]:
```
objects = ('Decision Tree', 'Naive Bayes', 'Linear Regression','Lin Reg with␣
 ↪Regularisation','Logistic Regression')
y_pos = np.arange(5)
performance = [Tree_Train,Naive_Train,Lin_Train,LinReg_Train,log_Train]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithms in Training dataset')

mp.show()
```



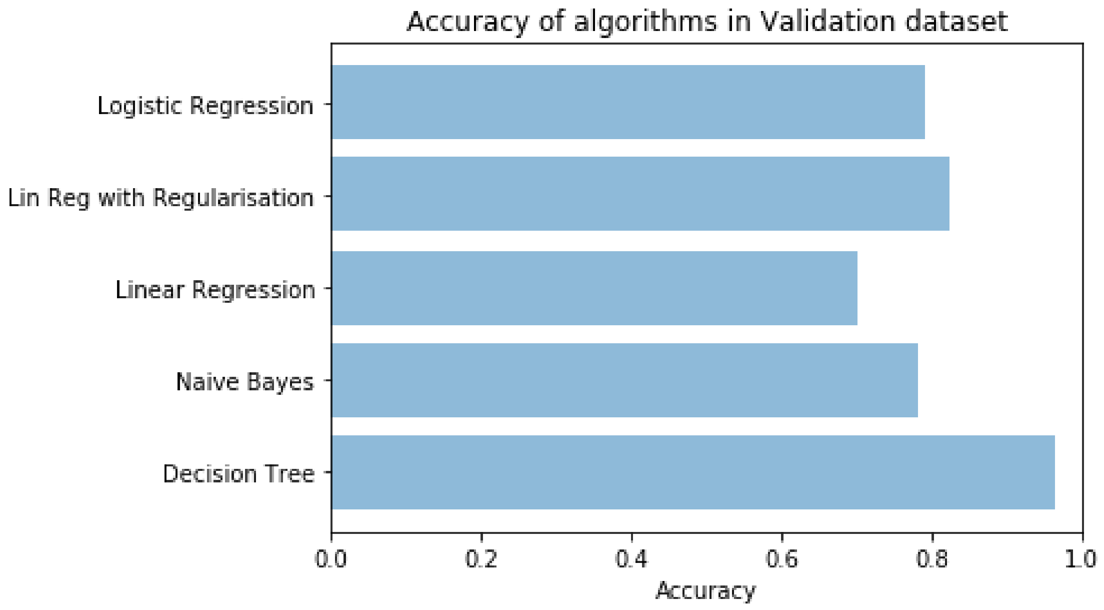
Let us summarise by showing the accurcies of the different methods on the Testing dataset:

[28]:
```
objects = ('Decision Tree', 'Naive Bayes', 'Linear Regression','Lin Reg with␣
 ↪Regularisation','Logistic Regression')
y_pos = np.arange(5)
performance = [Tree_Test,Naive_Test,Lin_Test,LinReg_Test,log_Test]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithms in Testing dataset')

mp.show()
```



Let us summarise by showing the accurcies of the different methods on the Validation dataset:

[29]:
```
objects = ('Decision Tree', 'Naive Bayes', 'Linear Regression','Lin Reg with␣
 ↪Regularisation','Logistic Regression')
y_pos = np.arange(5)
performance = [Tree_Val,Naive_Val,Lin_Val,LinReg_Val,log_Val]

mp.barh(y_pos, performance, align='center', alpha=0.5)
mp.yticks(y_pos, objects)
mp.xlim(0,1)
mp.xlabel('Accuracy')
mp.title('Accuracy of algorithms in Validation dataset')

mp.show()
```



From the graphs above we can determine the relative accuracy of each of the algorithms we chose to implement for this assignment. We can definitively say that the Decision Tree/ID3 algorithm is clearly the best for this dataset with an accuracy of over 90%. This is a favourable outcome as an algorithm which can correctly identify 9 out of every 10 phishing sites will be effective in any practical environment. The algorithm which yielded the worst accuracy is the normal Linear Regression algorithm. Whilst it yielded average results, its relative performance was less than satisfactory. This may be due to the fact that Linear Regression algorithms are generally not used for binary classification problems as they output continuous values. This is where Logistic Regression is better suited as it can be used for binary classification problems which is illustrated by its performance on our dataset, yielding 10% higher accuracy than the Linear Regression algorithm. The Linear Regression algorithm with Regularisation was clearly the most surprising result as we did not expect it to be the second best algorithm due to the poor performance of the normal Linear Regression method. From this we can determine that regularisation plays a major role in increasing the accuracy of the model. The Naive Bayes algorithm performed well on the dataset with slightly lower accuracy than Linear Regression with Regularisation. We can also state that all algorithms performed consistently on their respective levels as their Training,Testing and Validation accuracies were very close to each other.

If someone were to work with this dataset, we would recommend extending the work done in this assignment by implementing Logistic Regression with Regularisation to see if that comparible accuracy. If the regularisation has the same effect it has on normal Linear Regression it can be assumed that it will lead to a large increase in accuracy over the normal Logistic Regression algorithm. Due to time constraints we were unable to implement a Neural Network for this dataset, however it would also be interesting to observe the results and accuracies which would be achieved and compare them to that of the Decision Tree.

From this assignment, we learnt that the Decision Tree/ID3 algorithm is the best classifier we have for this dataset and we would definitely use this to predict whether a website is phishing site.