Universitetet i Tromsø

Fys-2021 - Machine Learning

Take home exam 2

Candidate number: 6

November 12, 2018

1 Problem 1

1.1 a

For the logistic discrimination classifier we model the ratio of the class likelihoods as opposed the linear regression, where we model the class likelihoods ($P(\mathbf{x}|C_i)$). We also assume that the ratio is linear,

$$\log \frac{p(\mathbf{x}|C_1)}{p(\mathbf{x}|C_2)} = \mathbf{w}^\intercal \mathbf{x} + w_0^o \tag{1}$$

Here the left hand side of the equation is the log ratio of the class likelihoods, and the right hand side is the linear equation where we have some line and a length w_0^o.

Using the logit function in Bayes rule where logit is given by,

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) \tag{2}$$

we get,

$$\begin{aligned}\text{logit}(P(C_1|\mathbf{x})) &= \log \frac{P(C_1|\mathbf{x})}{1 - P(C_1|\mathbf{x})} \\ &= \log\left(\frac{p(\mathbf{x}|C_1)}{p(\mathbf{x}|C_2)}\right) + \log \frac{P(C_1)}{P(C_2)} \\ &= \mathbf{w}^\intercal \mathbf{x} + w_0\end{aligned}$$

where $w_0 = w_0^o + \log\left(\frac{P(C_1)}{P(C_2)}\right)$. This can then be rearranged so that we get the known sigmoid function,

$$y = \hat{P}(C_1|\mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{w}^\intercal \mathbf{x} + w_0))} \tag{3}$$

This function works in such a way that all values plucked into it is always within 0 to 1, and since we solved it by using Bayesian statistics we can think of the result of this function as posterior probabilities.The sigmoid is also non-linear which makes it possible to model more advanced data which cant be separated with just a line. We can think of it as a neuron-activation function, in such a way that when we hit some value while training we make a choice. If we assume that our output r^t(which is 1 if $\mathbf{x} \in C_1$ or 0 if $\mathbf{x} \in C_2$)given data $\mathbf{x}^t$ is Bernoulli with the probability y^t of the sigmoid function, we then can use the sample likelihood,

$$l(\mathbf{w}, w_0|\mathcal{X}) = \prod_t (y^t)^{r^t}(1-y^t)^{(1-r^t)} \tag{4}$$

To maximize this we can minimize the error, which can be found by taking the negative log of the likelihood, $E = -\log l$,

$$E(\mathbf{w}, w_0|\mathcal{X}) = -\sum_t r^t \log y^t + (1 - r^t)\log\big(1 - y^t\big) \tag{5}$$

This is also called the *Cross-Entropy*, which has its relations to entropy in thermodynamics. This can be plotted when doing the training to see how the progress is doing.

Since the sigmoid function is nonlinear and in alot of situations do not have an analytic solution, we will use a numerical solution called gradient decent. Gradient decent iterates until a minimum value has been found using a given parameters. We use the cross entropy and take the derivative in terms of the parameters, in this case we have,

$$\Delta w_j = -\eta\frac{\partial E}{\partial w_j} = \eta\sum_t (r^t - y^t)x_j^t \tag{6}$$

and,

$$\Delta w_0 = -\eta\frac{\partial E}{\partial w_0} = \eta\sum_t (r^t - y^t) \tag{7}$$

where η is the learning rate(how big steps we take in our decent) if this is too big we might jump over the minimum, this can be seen if we plot the error after each iteration, this will be shown later in task b. After training we get parameters $\mathbf{w}$, we plug our line $\mathbf{w}^\intercal\mathbf{x}^t + w_0$ into our sigmoid, where $\mathbf{x}^\intercal$ is our testing data, and then choose C_1 if $y^t > 0.5$ and C_2 if otherwise (this is of-course for two classes).

1.2 b

When implementing the logistic discrimination, i started with making a class such that i have an object of the model with a given data set,

```
y = ml.predictor(training = training_data)
```

When training we send in the training data as a matrix where we will have the first row as ones,

$$
Data = \begin{bmatrix} 1 & x_{1,2} & \ddots \\ 1 & x_{2,2} & \ddots \\ 1 & x_{3,2} & \ddots \\ \ddots & \ddots & \ddots \end{bmatrix} \tag{8}
$$

In this way the first column will be our length w_0 when making the line $\mathbf{w}^\intercal\mathbf{x} + w_0$. Code snippet below shows how we start training with $3.18 \cdot 10^{-5}$ learning rate and 600 max iterations.

```
y.train(3.18e-5,it = 600)
```

We then assign a random weight vector around zero, in this case we use random uniform vector between $-0.01 \rightarrow 0.01$ with the d where d is the amount of features of the data set. We then perform the gradient decent, here we start with the random weight vector and take the inner product with the data, $\mathbf{X} \cdot \mathbf{w}$ and plug this into the sigmoid function, we then take the difference between the labels(classes which in this case is either 0 or 1, we want it to binary) and the predicted probability given by the sigmoid. This difference is then dotted $diff \cdot X$ and added, multiplied with the learning rate and then added to the to the weight, we have now done equation(6) and (7).This is called while doing the gradient decent, and is in the code snippet below,

```
for i in range(it):
    delta_w = np.zeros(d) # make empty delta vector with
                              zeros
    o = X@w # dot product of features and weight
    y = vect_func_sig(o) #Find predicted probability
    temp = np.array(r) - np.array(y) # subtract the label
                              with the probability
    # temp (958,1) X (958,9) = (1,9)
    delta_w += temp@X # update delta
    delta_w *= lr
    w += delta_w # update weight with new delta and the
                              learning rate
    # Assign self property weight to the trained weigth
    self.weight = w
    conf = self.conf_mat(self.class1, self.class2)[1]
    # Add to errorfunction
    self.errorfunction(r,y)
    if (conf[0,1] == conf2[0,1] and conf[1,0] == conf2[1,0]):
        # Assign self property weight to the trained weigth
                              and the learning rate
        self.weight = w
        self.lr = lr
        return
```

We add the error to a list for plotting to analyze the error to see if we have good enough learning rate/iterations. This process is done over n amount of iterations, but in figure(1) below we use 100 iterations, we can also see how the learning rate affects the training, we use the code snippet below for this,

```
def errorfunction(self, r, y):
    """Finds the value of cross-entropy"""
    r = np.array(r)
    y = np.array(y)
    # The cross entropy (error/cost function)
    err = -np.sum(r*np.log(y+1e-10) + (1 - r)*np.log(1 - y+1e
                                        -10) )
    self.error.append(err)
```

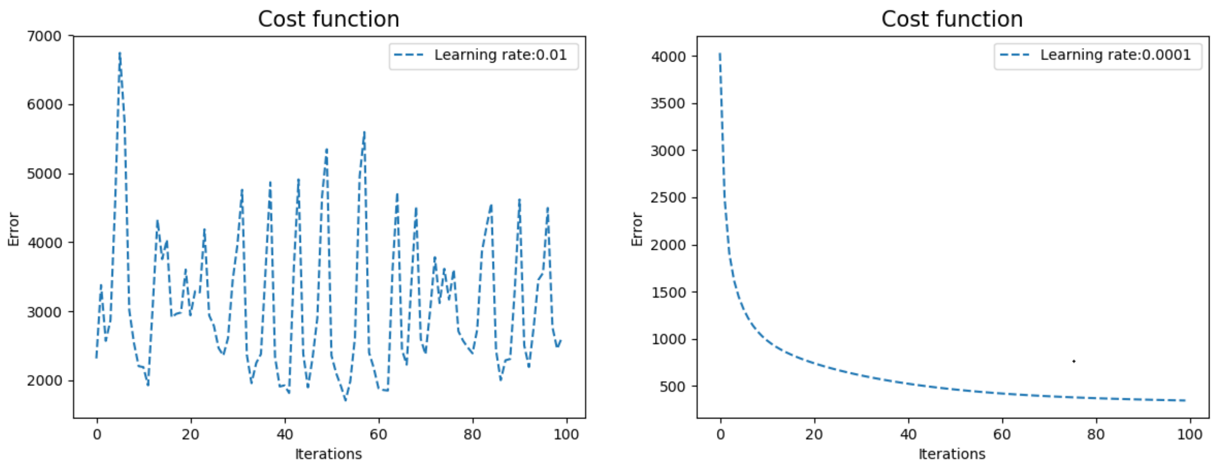

Figure 1: Plots of error with different learning rates

Here in figure(1) we can see that in this specific training when we use learning rate of 0.01 we ”wobble” around the minimum as we take to big steps, such that we go back and forth a minimum(or minimums), in the second plot we have a converging learning rate, which is what we want to have. One important aspect is to find out if we are over-fitting.

Here we would split our data-set into one validation data[1] and rest as the training data. Here we could plot the accuracy as a function of learning rate and see how the accuracy of the two sets are moving, if they move away from each other, then we have the case where we fit our model to the noise of the training set, in this case we would get very good accuracy of the training set, but as soon as we introduce the validation set or any other set we would get a bad accuracy, and we have then over-fit our model. One way to not over-fit is to choose the parameter which has the best accuracy for the validation set. In the figure(2) below we can see how the the training set is slowly becoming over-fittet as it approaches 100% accuracy and that our accuracy for the validation set gets worse and stagnate,

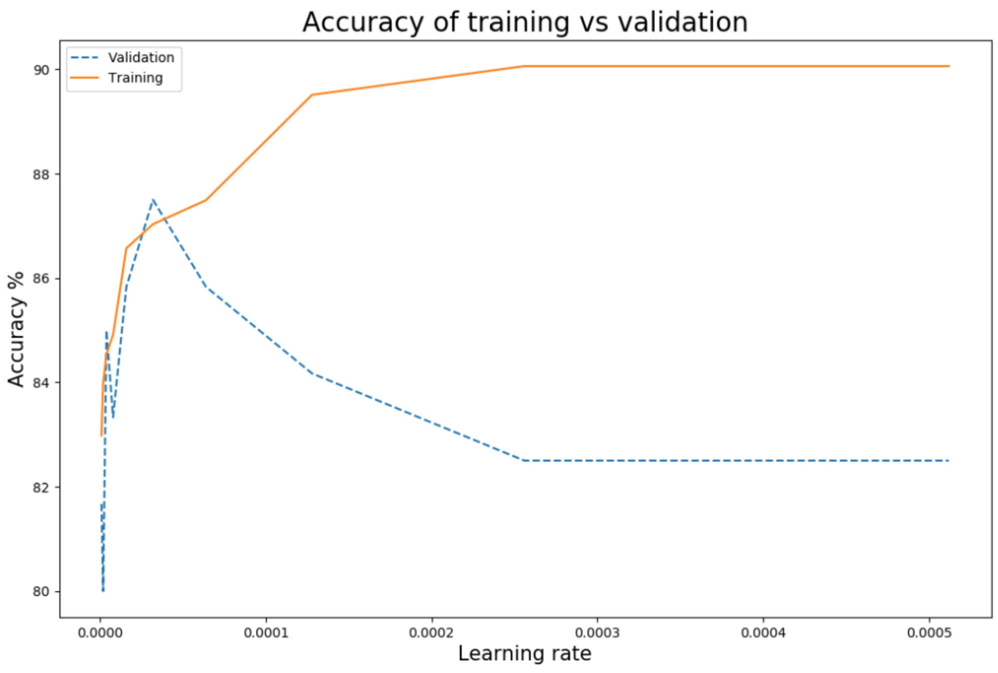

Figure 2: Here we have the validation set in blue and training set in orange, as we can see the best value for learning is about $\eta = 3.18 * 10^{-5}$, this also yields accuracy of about 87% on our validation set.

We can also plot the error function to see how we are minimizing the error in figure(3),

[1]Remark: We do not touch the test data before we are really ready to use our classifier, therefore we split our training set into training /validation set, this is to avoid introducing bias into our model as we know know the outcome of the test set.

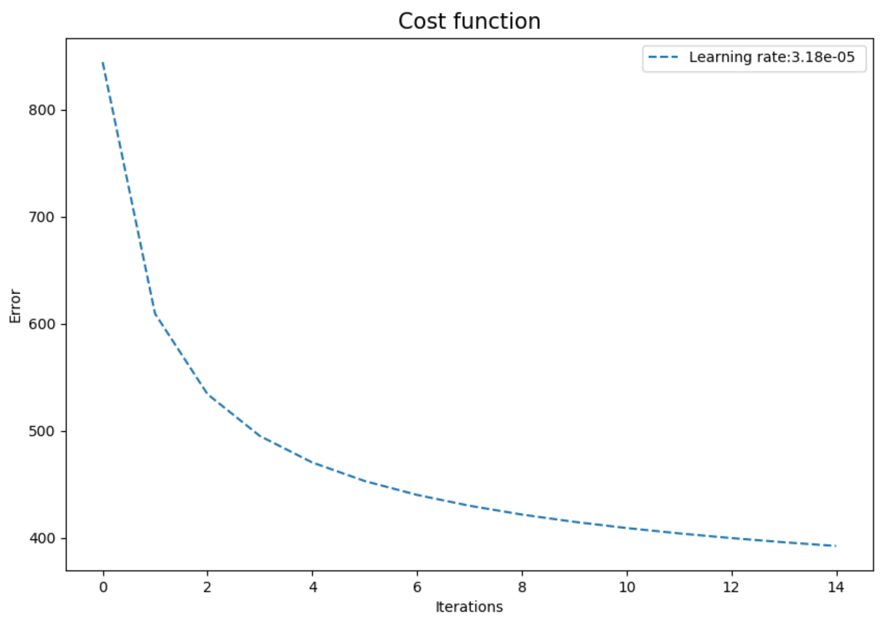

Figure 3: Here we see that the error is getting lower and lower as we iterate

When training we stop if we have no change in false classifications, although if we keep training we harden the sigmoid, (our projections will cluster the classes more and more together) we wont have a change in false classifications, so we want to just stop iterating. This is shown in snippet,

```
if (conf[0,1] == conf2[0,1] and conf[1,0] == conf2[1,0]):
    # Assign self property weight to the
    # trained weight and the learning rate
    self.weight = w
    self.lr = lr
    return
```

We might want to stop earlier then when we have zero error, because we initialize the weight around zero, so if we keep iterating we will move away, so stopping early will effectively yield less parameters, and therefore better generalization.

with 600 iterations we get the following confusion matrix on our test set.

We therefore have the **accuracy at 90.61%**. To classify a given data set we implemented a function which takes the weight vector that have been trained with the training data, and perform the inner product with the data set, which has *ones* in the first row as shown in equation(8), and plug it into our sigmoid function, we then use a decision boundary function which takes the probability from the sigmoid and returns a value of 1 if the probability

	Predicted class	
True Class	Positive	Negative
Positive	**91**	11
Negative	9	**102**

Table 1: Confusion matrix of the test data by using logistic discrimination

is 0.5 or higher,

```
return 1 if x >= 0.5 else 0
```

For the first two pictures(index 0 and 1) in figure(4) we got harp seal pup for index 0 and hooded seal pup for index 1.

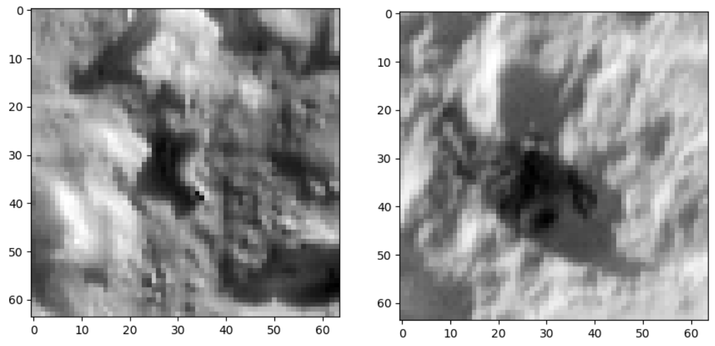

Figure 4: Hooded seal pup to the left and harp seal pup to the right in greyscale

2 Problem 2

2.1 a

A decision tree is a hierarchical model for supervised learning, where we split a given data set in the best way, this split can be done by measuring the impurity of the splitted data. The impurity is a measure of how mixed a data set is, there are different measurements as gini index or entropy to mention a few. The entropy is given by,

$$S_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i \tag{9}$$

Here p_m^i is the ratio of class i reaching a node m, so we would have a pure node if entropy is either 1 or 0. Here we will define $0 \log 0 \equiv 0$.

When making the tree we start with a good split by measuring the information gained when splitting the data at the first feature. Information gain tells us how important a given attribute of the feature vectors is, in this case we take the parent entropy of the node, and weighted average of the child nodes times the entropy of each child node. We then go through all the features and see if the information gain is higher than the previous, if it is higher then we assign the new threshold and feature index. When we have gone through all the features we return the threshold value, the index of the feature and the information gained. When we return the values we check if the information gain was less then some minimum value, if this is the case we say that we have met our condition for classifying the data-set sent in to the node, and we turn it into a leaf. A leaf is where we classify our data, here we assign the class which has the majority class, or in other words the class which has the highest probability of being true. If they do not meet the condition for becoming classified we send the threshold value and data-set into a split function, this split function takes the threshold and splits the data which satisfy the threshold to the left(branch) and the data which do not meet the threshold to the right branch, we now continue to build the left branch until we reach a leaf, then we go back to last node and build the right branch and then we continue with the left branch until leaf. This process is called recursion, when we finally have all all data-points in our data-set sent into leaves and classified, we then want to return the tree structure. In figure(5) below we have a visualization of a small tree structure. Here the elliptical nodes decides where the data should go, and the squares are leaves which classifies the data by majority label.

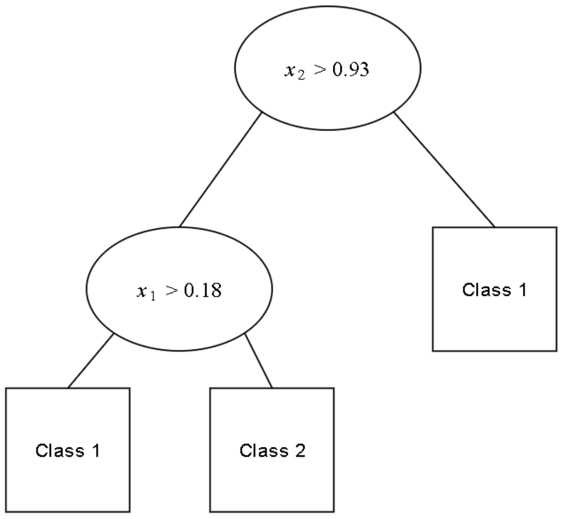

Figure 5: Example of a small decision tree. Decision trees can become very big.

Pruning, Usually we don't want to split data into leaf which contains very few data points, the reason for this is that if we decide to classify on too few points we will cause variances and generalization errors. So what we can do is stopping at some value, like 5%, or that we use a maximum depth.

2.2 b

When implementing the decision tree we start by making the entropy function, this function calculates the impurity in the data-set, this impurity is found by calculating how much there are of the different classes in the given training data. In this case, instead of using the general case as given in equation(9), we will use the simplified version for just two classes.

$$Entropy = -p\log_2 p - (1-p)\log_2(1-p) \tag{10}$$

Here p is the ratio/probability of class 1, so we then have the code

```
entropy = -p*np.log2(p + 1e-20) - (1-p)*np.log2(1-p+1e-20)
    return entropy
```

Here we have put in a very small number 10^{-20} to avoid having error in the log.

Next we make a *count_class* function, this function finds how many classes there are in a given data-set and returns the amount and the labels in a dictionary to handle it easier, this function will be used to find which is the majority class in a leaf. Next we have the info gain as previously described,

```
def inf_gain(left,right, parent_entropy):
    # https://en.wikipedia.org/wiki/Decision_tree_learning
    # Get the weight if the child nodes
    p = float(len(left)) / (len(left) + len(right))
    # Return the information gained by this split.
    return parent_entropy-p*entropy(left)-(1-p)*entropy(right
                                     )
```

Here we use the parent entropy of the node and subtract the weighted child node entropy, if we gain zero information gain we no longer are able to gain more information, and it would be ideal to make a leaf at this point. As mentioned in previous tasks there are different ways of measuring impurity, in this task we used the entropy and not the gini index, altough there is not a significant difference between the measurements, we use the entropy because its more intuitive.

We then have a threshold class, this class makes an object which holds the threshold for where to separate the data, and a method for checking the threshold against the features in the data-set which happens to be tumbling down the tree. In this report we will only show the small snippets of the code, rest is put in the appendix at the end. Next up we have the leaf class,

```
class leaf:
    """Leaf node, this is where a node
    becomes classified with the majority class."""
    def __init__(self,data):
        self.amount = count_classes(data)
        if self.amount[0] > self.amount[1]:
            self.classified = 'Class 0'
        else:
            self.classified = 'Class 1'
```

This object is made whenever we satisfy a given entropy or information gain, when this object is made we use the *count_class* method, then we assign a string describing the majority class which will be our classification. Next we have the node,

```
class node:
    """A node in the tree"""
    def __init__(self, thresh, index, left, right):
        self.thresh = thresh
        self.left = left
        self.right = right
        self.index = index
```

This class makes the node object, which holds the threshold object, the left and right branch and the index of which the feature had the best split. Going on we have the split function,

```
def split(dataset, threshold):
    """ Splits the data """
    left, right = [], []
    for row in dataset:
        # Uses the threshold.check method to test if feature
                                             meets the threshold
                                             condition
        if threshold.check(row):
            left.append(row)
        else:
            right.append(row)

    # Return left and right branch data
    return left, right
```

this function takes the data-set (which could be the whole data-set, left or the right branch data-set), we also take in the threshold object, and then take each row(image in this case) and checks all the feature that satisfy the threshold, all images who do satisfy the threshold is sent to the left branch and if not we send it to the right branch. Next we have the best split function, this function takes in a data and finds the split which yields the most information, this function uses the split function described above, when we find the best threshold, its feature and the information gain we return the values to the tree builder(fit function). The next part is the tree class, this class is our tree structure, it has a fit function which builds the structure of the tree. Here the function uses all the function described above, first we initialize self variables which holds the classification stats in our model for the confusion matrix, the first part in our tree is the depth check, we have this to be able to apply any depth we want for our tree, (how far do we want our left branch to go) in some cases our tree could be done before we reach this max depth depending on our entropy limit etc, but in some cases we want to make a max depth. This is to maybe fine-tune our model in such a way that we don't over-fit our model. If our depth is reaches we assign the data to a leaf object. We then assign left and right variable to the $find\_best\_split$ so that we get the threshold, index and the information gained, this is the function which will take the longest time and will decrease as we get less and less data into nodes. We denote the amount of operations needed by $\mathcal{O}(N)$(Big O notation) where the N is the amount of operations needed(so we might have $\mathcal{O}(N \log N)$ etc.). There are also different optimization methods such as threading or libraries like numba and pytorch. After we find the best split values, we see if our information gain was less then or equal to 0.11, the reason for this is because by splitting the training set into training and validation set we find that the best accuracy where we try to avoid over-fitting, as we see in figure(6) below,

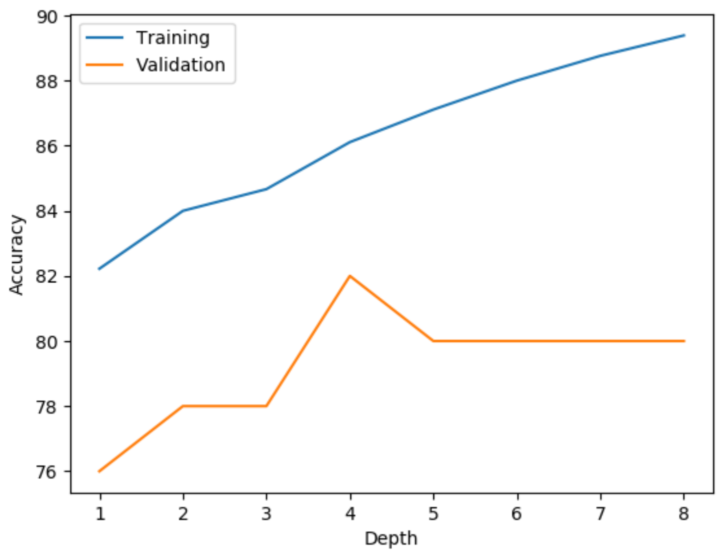

Figure 6: We see how the training set diverges away from the validation data as we increase the depth, the training data get better and better accuracy because we are starting to fit the noise in the data. Here we made leaves if $info\_gain <= 0.11$

Here we also see that the best depth is 4, so we will use this in our model as max depth. We then have a *node_stats* function which prints out the tree recursively so that we see the nodes and their threshold, and the leaf classifications and their data set. We then have a predict function which takes the tree structure, which holds all the nodes and leaves, and an image(each row of the test set), here we check if the row features match the thresholds in the node, if it does match we send it to the left(which is also called the *True branch*), if not we send it to the right. This is done until it falls into a leaf where it gets classified to what the leaf was classified during building. When predicting the test data we get **accuracy of 87.79%** and the confusion matrix given in the table below,

	Predicted class	
True Class	Positive	Negative
Positive	**82**	6
Negative	20	**105**

Table 2: Confusion matrix of the test data by using decision tree classification

3 3

3.1 a

We get very good accuracy's for both classifiers, 90.61% for logistic discrimination and 87.79% for the classification tree, so we actually got better accuracy with the logistic discrimination then the decision tree even though the tree is more "advanced". It also took 5.30 minutes for the tree and 1 second for the logistic discrimination, so here we might want to apply the Occam's razor, and maybe choose the less advanced algorithm. In some cases we may have very few data-points and therefore the accuracy wont be as impact full or have a lot of meaning.

3.2 b

The ROC curve is a diagnostic tool to analyze the performance of our model, as we can see in figure(7) below we can observe how the change in parameter values make our model hit/miss more on our classification. We can see the x axis as cost, and the y axis as benefit. Here we want to get as close to the left top corner as possible. The line in the middle represents where we have a random decision threshold, and as we pass this line we get a worse threshold then just having a random one. If we get points below the line which are very bad we might want to mirror it in such a way that we go from 0.10 tp to 0.90 tp.

The fp rate will tell how when we have "false alarm", this could in some cases be very bad, like if we have wrong user logging into a bank account. So in this case we might want to lower fp-rate on the mercy of tp rate. As we can see in the figure of the ROC curve, we might want to choose a decision boundary at 0.5-0.6 because it is where we have the most tp and the least fp. As we can see in figure(7) we can easily change tp and fp if we change boundary conditions for classification.

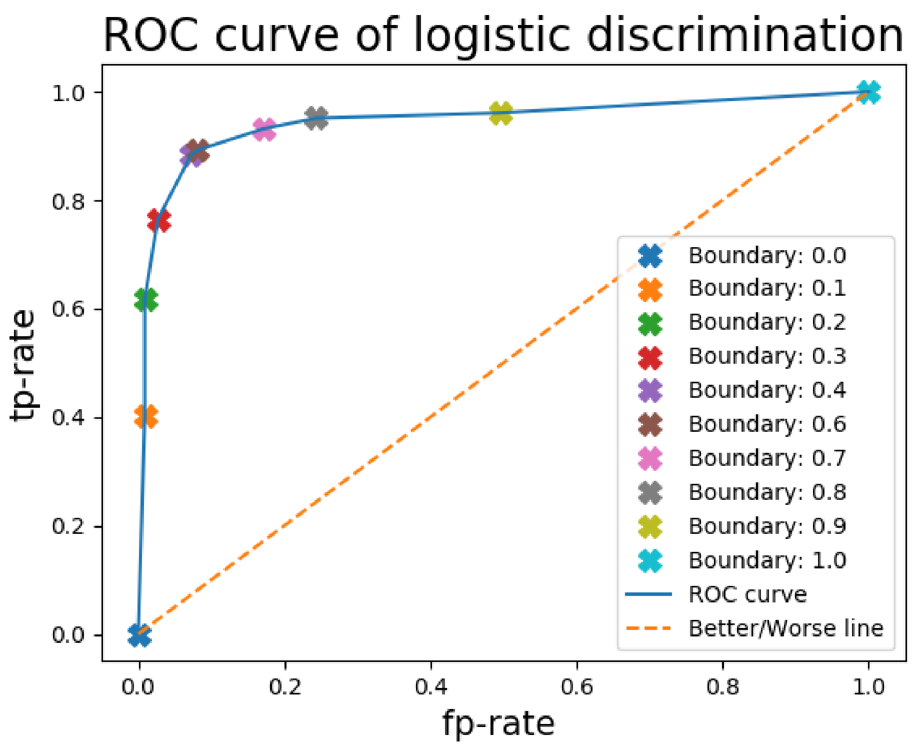

Figure 7: ROC curve of the logistic discrimination with test set

3.3 c

To implement soft decision we can change our leaf object to contain the probability of the two classes, as such,

```
class leaf
def predict(self, row, tree, threshold_value):
    """Leaf node, this is where a node becomes classified
                                with the majority class.
                                """
    def __init__(self,data):
        self.amount = count_classes(data)
        # probability classes
        self.pc0 = self.amount[0]/(self.amount[1] + self.
                                        amount[0])
        self.pc1 = self.amount[1]/(self.amount[1] + self.
                                        amount[0])
```

We can then change our prediction method to check for a given threshold(given by a loop in the range 0 to 1) in the leaf and see if the probability for class 0 is greater or equal to the threshold value, if it satisfy this condition we say it is class 0 and class 1 otherwise. This can be done like this,

```
if isinstance(tree, leaf):
            ss0 = tree.pc0
            print(ss0)
            if ss0 >= threshold_value:
                return 'Class 0'
            else:
                return 'Class 1'
```

The ROC curve for the decision tree is given in figure(8) below,

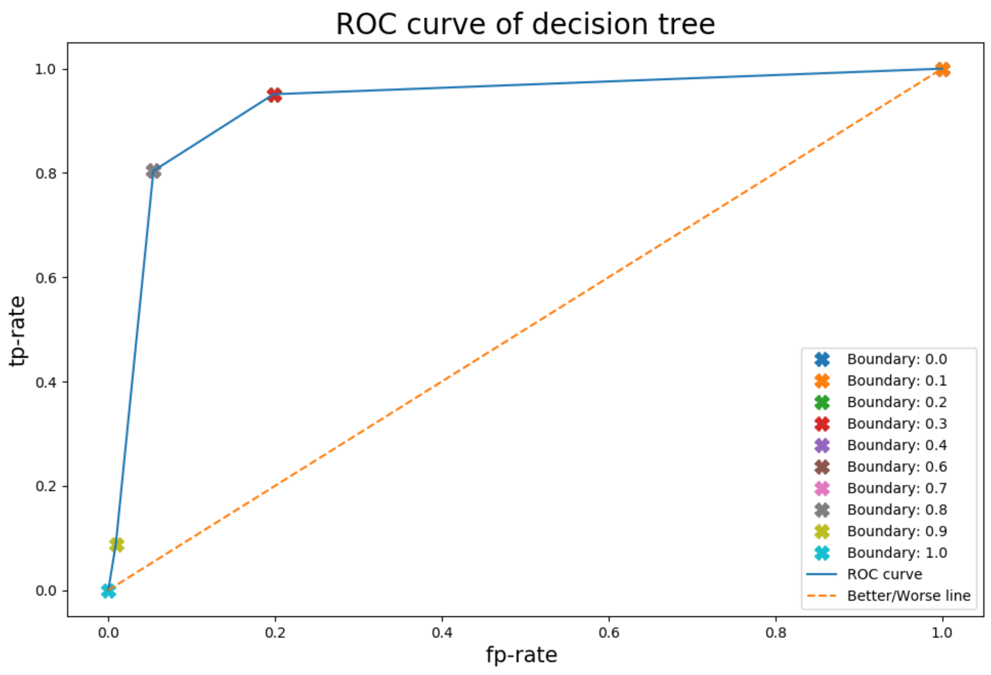

Figure 8: ROC curve of the decision tree, here we have the case that some boundary thresholds do overlap, it looks like we have the best threshold at 0.6

3.4 d

Here i use the numerical integral from numpy to get the area under the curve for the logistic discrimination classifier,

```
# Find area under curve with trapezoid rule
print('AUC: %.2f '%(np.trapz(y = tp, x = fp)))
>> AUC: 0.95
```

Here we got AUC at ≈ 0.95, the area under the curve represents the percentage of randomly drawn for which the classifier classifies true classes. So here we have a good model.

For the decision tree we have AUC at ≈ 0.93, but here we got alot less points, so we might want to use a deeper depth other then max depth 4. As is,it looks like we have a better model using the logistic discrimination, or that we have not yet found the best depth/ gain threshold for the decision tree.

3.5 e

Here is a function for plotting seal pictures from "seals_images_test.csv"

```
def get_seal_img(data,index):
    #Extract row, that contains 4096=64    64 points
    one_image = data[index]
    # reshape image to 64x64
    reshaped_image = np.reshape(one_image, (64,64))

    # Show grayscale image
    plt.imshow(reshaped_image)
    plt.show()
```

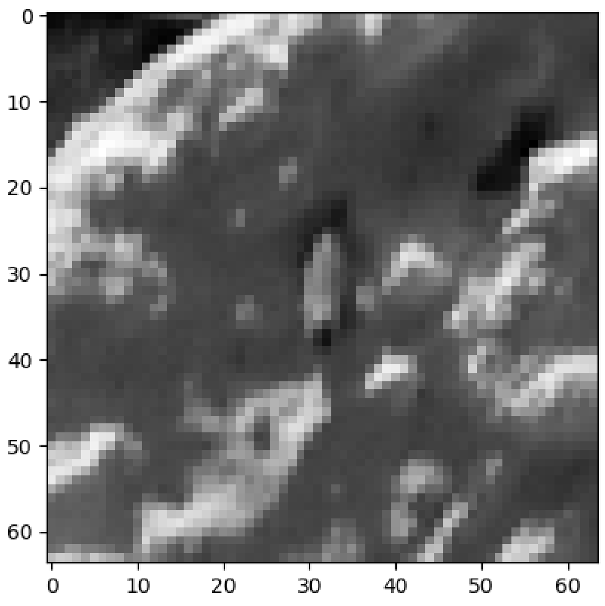

Figure 9: Seal picture with index corresponding to my candidate number.

3.6 f

Using the code below we see which index is miss-classified,

```
# Set up classifier
y = ml.predictor(training = training_data , decision= 0.5)
# Train with given learning rate and 600 iterations
y.train(3.18e-5,it = 600)
# Initialize array
lst = []
# How many picture to look at
N = 100
# Iterate through a given amount of picture
for i in range(0,N+1):
    test = np.copy(test_data)[i,:]
    # Hold which class it actually is
    classif = test[0]
    # Set it ready to be classified
    test[0] = 1
    # Append array with index of wrongly classified image
    if y.predict(test) != classif:
        lst.append(i)
print(lst)
>>[22,29,31,35,50,53,65,74,88,93]
```

So if we look at some of them in figure(10) below,

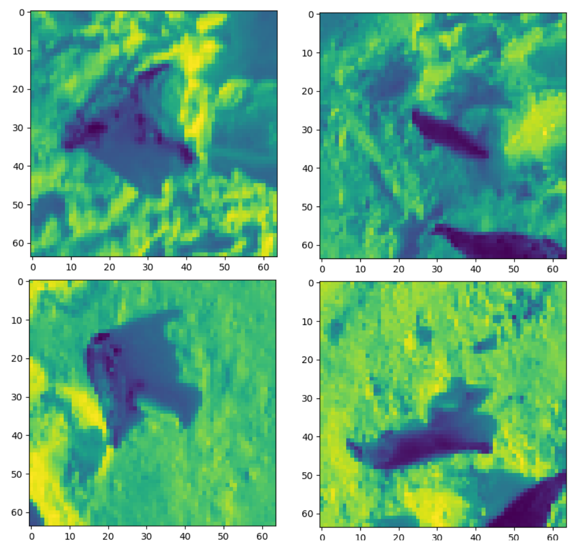

Figure 10: Some of the miss-classification's, here we see that most of the ones which are miss-classified have more than one seal in the pictures, this may confuse the classifier

4 Appendix

Task 1

```
# Import data
import numpy as np
import matplotlib.pyplot as plt
import machinepy as ml
import time

# Load data
test_data = np.genfromtxt('seals_test.csv')
training_data = np.genfromtxt('seals_train.csv')

# percentage of data
p = 0.1
percentage_data = int((len(training_data)*p))

#Split data into training and validation data
validation_data = np.copy(training_data)[0:percentage_data,:]
training_data = np.copy(training_data)[percentage_data:,:]


def clean_test(x):
    """Order the data into classes for the confusion matrix
                                """
    class1 = []
    class2 = []


    for row in x:
        if row[0] == 0:
            class1.append(row)
        else:
            class2.append(row)


    class1 = np.array(class1)
    class2 = np.array(class2)

    class1[:,0] = 1
    class2[:,0] = 1

    return class1, class2

# Copy to avoid pointers to data set
validation = np.copy(validation_data)
validation_cl = clean_test(validation)
```

```
training = np.copy(training_data)
training_cl = clean_test(training)

# Set ones in first column
validation[:,0] = 1
training[:,0] = 1

test = np.copy(test_data)
data = clean_test(test)
# Sets object with logistic discrimination and assign
                                training data
y = ml.predictor(training = training_data)


#Trains the model
val = []
tra = []
lrr = []
lr = 0.000001
for i in range(1,11):
    y.train(lr, it = 600)
    #Print out the predictions
    tra.append(y.conf_mat(training_cl[0],training_cl[1])[0])
    #Show confusion matrix of the test data.
    val.append(y.conf_mat(validation_cl[0],validation_cl[1])[
                                    0])


    # y.conf_mat(data[0],data[1])

    # append the current learning rate
    lrr.append(lr)
    lr += lr



# Plot data
plt.plot(lrr,val,'--', label = 'Validation')
plt.plot(lrr,tra, label = 'Training')
plt.ylabel('Accuracy %',fontsize = 15)
plt.xlabel('Learning rate',fontsize = 15)
plt.title('Accuracy of training vs validation',fontsize = 20)
plt.legend(loc = 'best')

plt.show()
```

```
test = np.copy(test_data)
data = clean_test(test)


# Sets object with logistic discrimination and assign
                                 training data
y = ml.predictor(training = training_data, decision= 0.5)


y.train(3.18e-5,it = 600)
 #Print out the predictions
print(y.conf_mat(data[0],data[1])[1])

y.plterror()
plt.show()
```

Task 2

```


% of data set
p = 0.10
dist = int(len(training_data)*p)

# Initaite training, validation and parameter list
tr = []
val = []
de = []


# Set up training and validation
training = np.copy(training_data)[dist:,:]
validation = np.copy(training_data)[0:dist,:]
my_tree = ml.Tree()

# use timer to time the algorithm
import time
for i in range(1,9):

    e0 = time.time()
    #
    tree = my_tree.fit(training, i)
    #
    elapsed_time = time.time() - e0

    # Create tree object
    my_tree.node_stats(tree)
    # Make confusion matrix for training set
    mat = np.array([[my_tree.class0pos,my_tree.class0neg],[
                                   my_tree.class1neg,my_tree.
```

```
                                               class1pos]])
      # Find accuracy
      accuracy = sum(np.diag(np.copy(mat)))/sum(sum(np.copy(mat
                                               )))
      print(f'Accuracy training:{accuracy*100:.2f} %')
      tr.append(accuracy*100)


      # Make confusion matrix for validation set
      confusion_mat = np.zeros((2,2))
      for row in validation:
          predicted = my_tree.predict(row, tree)
          if predicted[1] == 'Class 0':
              if row[0] == 0:
                  confusion_mat[0,0] += 1
              else:
                  confusion_mat[0,1] += 1
          else:
              if row[0] == 1:
                  confusion_mat[1,1] += 1
              else:
                  confusion_mat[1,0] += 1

      # print out accuracy and confusion matrix for validation
                                               set
      print(confusion_mat)
      confusion_mat = np.array(confusion_mat)
      accuracy = sum(np.diag(np.copy(confusion_mat)))/sum(sum(
                                               np.copy(confusion_mat)))
      print(f'Accuracy testing:{accuracy*100:.2f} %')

      val.append(accuracy*100)
      de.append(int(max(my_tree.nodenumber)))

      # print out deopth of the tree
      print('Depth: ',max(my_tree.nodenumber))
      print('Tree building took: %.2f minutes'%(float(
                                               elapsed_time)/60))

# Plot
print(tr)
print(val)
plt.plot(de, tr, label = 'Training')
plt.plot(de, val, label = 'Validation')
plt.legend()
plt.ylabel('Accuracy')
plt.xlabel('Depth')
plt.show()
```

Task2 testing data set

```
# Load training set
training_data = np.genfromtxt('seals_train.csv')

# avoid pointing to data set
training = np.copy(training_data)
test = np.copy(test_data)

# make tree
my_tree = ml.Tree()

# make timer
import time
e0 = time.time()
#
tree = my_tree.fit(training, 4)
#
elapsed_time = time.time() - e0

# Print node stats of tree / tree structure
my_tree.node_stats(tree)
mat = np.array([[my_tree.class0pos,my_tree.class0neg],[
                                my_tree.class1neg,my_tree.
                                class1pos]])
accuracy = sum(np.diag(np.copy(mat)))/sum(sum(np.copy(mat)))
print(f'Accuracy training:{accuracy*100:.2f} %')


# Make confusion matrix
confusion_mat = np.zeros((2,2))
for row in test:
    predicted = my_tree.predict(row, tree)
    if predicted[1] == 'Class 0':
        if row[0] == 0:
            confusion_mat[0,0] += 1
        else:
            confusion_mat[0,1] += 1
    else:
        if row[0] == 1:
            confusion_mat[1,1] += 1
        else:
            confusion_mat[1,0] += 1

# Print out confusion matrix
print(confusion_mat)
confusion_mat = np.array(confusion_mat)
accuracy = sum(np.diag(np.copy(confusion_mat)))/sum(sum(np.
                                copy(confusion_mat)))
```

```
print(f'Accuracy testing:{accuracy*100:.2f} %')

# get depth and how long it took to make tree
print('Depth: ',max(my_tree.nodenumber))
print('Tree building took: %.2f minutes'%(float(elapsed_time)
                              /60))
```

Task 3 ROC curve logistic

```


# Import data
import numpy as np
import matplotlib.pyplot as plt
import machinepy as ml
import time

test_data = np.genfromtxt('seals_test.csv')
training_data = np.genfromtxt('seals_train.csv')

def clean_test(x):
    """Order the data into classes for the confusion matrix
                              """
    class1 = []
    class2 = []


    for row in x:
        if row[0] == 0:
            class1.append(row)
        else:
            class2.append(row)


    class1 = np.array(class1)
    class2 = np.array(class2)

    class1[:,0] = 1
    class2[:,0] = 1

    return class1, class2



test = np.copy(test_data)
data = clean_test(test)

tp = []
fp = []
```

```
lr = 3.18e-5
for i in np.arange(0.01,1,0.10):
    y = ml.predictor(training = training_data, decision = i)
    y.train(lr, it = 20)
    res = y.conf_mat(data[0],data[1])[1]
    x = (res[1,0]/(res[1,0]+res[1,1]))
    y = (res[0,0]/(res[0,0]+res[0,1]))
    plt.plot(x,y,'X', markersize = 10, label = 'Boundary: %.
                                  1f'%i)
    #print(res[0,0]+res[1,0])
    tp.append(y)
    fp.append(x)
# Append start and finish of ROC curve
tp.append(1)
fp.append(1)
tp.append(0)
fp.append(0)
# Sort for integral
fp = sorted(fp)
tp = sorted(tp)
print(fp)
print(tp)
# Deletes NaN's from list
del fp[-2]
del tp[0]

# Find area under curve with trapezoid rule
print('AUC: %.2f '%(np.trapz(y = tp, x = fp)))
plt.plot(fp,tp, label = 'ROC curve')
plt.plot([0,1],[0,1],'--', label = 'Better/Worse line')
plt.title('ROC curve of logistic discrimination', fontsize =
                              '20')
plt.xlabel('fp-rate ', fontsize = '15')
plt.ylabel('tp-rate', fontsize = '15')
plt.legend(loc = 'best')
plt.show()
```

Machinelearning module

```



import numpy as np
import matplotlib.pyplot as plt




def get_seal_img(data,index):
    #Extract row, that contains 4096=64    64 points
    one_image = data[index]
    # reshape image to 64x64
    reshaped_image = np.reshape(one_image, (64,64))

    # Show grayscale image
    plt.imshow(reshaped_image, cmap='gray')
    plt.show()



class predictor:
    def __init__(self, training, decision = 0.5):
        self.training_data = training
        self.error = []
        self.decision = decision

        class1 = []
        class2 = []

        for row in training:
            if row[0] == 0:
                class1.append(row)
            else:
                class2.append(row)


        class1 = np.array(class1)
        class2 = np.array(class2)

        class1[:,0] = 1
        class2[:,0] = 1

        self.class1 = class1
        self.class2 = class2


    def sigmoid(self, x):
        """Sigmoid prediction function.

        Args:
            x: Data to be predicted?

        Returns:
            value between 0 and 1

        """
        return 1/(1+np.exp(-x))

    def Decision_boundary(self,x):
        """Decision boundary of the classifier
        Args:
            x: give the probability from the sigmoid function

        Returns:
            1 if x higher or equal to 0.5, and 0 if anything
                                        else

        """
        return 1 if x >= self.decision else 0


    def errorfunction(self, r, y):
        """Finds the value of cross-entropy"""
        r = np.array(r)
        y = np.array(y)
        # The cross entropy (error/cost function)
        err = -np.sum(r*np.log(y+1e-10) + (1 - r)*np.log(1 -
                                    y+1e-10) )
        self.error.append(err)


    def plterror(self):
        """Plots the error function of the training"""
        plt.plot(self.error,'--', label = 'Learning rate:%g '
                                    %(self.lr))
        plt.title('Cost function', fontsize = 15)
        plt.ylabel('Error')
        plt.xlabel('Iterations')
        plt.legend(loc = 'best')

    def conf_mat(self,x,y):
        """Sets up the confusion matrix."""
        # Sets the weight parameter
        w = self.weight

        # Vectorizes the functions
        vect_func_sig = np.vectorize(self.sigmoid)
        vect_func_dec = np.vectorize(self.Decision_boundary)

        # Finds the probability for each class
        prob_1 = vect_func_sig(x@w)
        prob_2 = vect_func_sig(y@w)

        # Finds the predicted class of the data
        result1 = vect_func_dec(prob_1)
        result2 = vect_func_dec(prob_2)


        # Sets up the confusion matrix
        confusion = np.zeros((2,2))
        for cl in result1:
            if cl == 0:
                confusion[0,0] += 1
            else:
                confusion[0,1] += 1
        for cl in result2:
            if cl == 1:
                confusion[1,1] += 1
            else:
                confusion[1,0] += 1

        # prints the confusion matrix
        #print(confusion)
        # prints the accuracy of the model
        accuracy = sum(np.diag(confusion))/sum(sum(confusion)
                                             )
        #print(f'Accuracy:{accuracy*100:.2f} %')
        return float('{:.2f}'.format(accuracy*100)),confusion



    def train(self, lr, it = 100):
        # Reset in case we train again
        self.error = []
        rows, cols = self.training_data.shape
        X = np.copy(self.training_data)
        r = np.copy(self.training_data[:,0]) # Get labels
        w0 = np.ones(cols)
        w0 = w0.reshape((cols,1))
        # make first column ones
        X[:,0] = 1

        #Reshape matrices
        w0_class1 = np.ones(cols)
        w0_class1 = w0_class1.reshape(cols,1)

        w0_class2 = np.ones(cols)
        w0_class2 = w0_class2.reshape(cols,1)
        N, d = X.shape # Get shapes of features

        # Vectorize sigmoid function
        vect_func_sig = np.vectorize(self.sigmoid)
        # Initialize random vector
        w = np.random.uniform(-0.01, 0.01, size = d )

        # start value for conf2
        conf2 = np.array([[99,99],[99,99]])
        # Gradient descent
        for i in range(it):
            delta_w = np.zeros(d) # make empty delta vector
                                            with zeros
            o = X@w # dot product of features and weight
            y = vect_func_sig(o) #Find predicted probability
            temp = np.array(r) - np.array(y) # subtract the
                                            label with the
                                            probability
            # temp (958,1) X (958,9) = (1,9)
            delta_w += temp@X # update delta
            delta_w *= lr
            w += delta_w # update weight with new delta and
                                            the learning rate
            # Assign self property weight to the trained
                                            weigth
            self.weight = w
            conf = self.conf_mat(self.class1, self.class2)[1]
            # Add to errorfunction
            self.errorfunction(r,y)
            if (conf[0,1] == conf2[0,1] and conf[1,0] ==
                                            conf2[1,0]):
                # Assign self property weight to the trained
                                                weigth and the
                                                 learning rate
                self.weight = w
                self.lr = lr
                return
            # make the last confusion matrix
            conf2 = self.conf_mat(self.class1, self.class2)[1
                                            ]
            #print('Current iteration: {0}/{1}'.format(i,it),
                                             end = '\r')


        # Assign self property weight to the trained weigth
        self.weight = w
        self.lr = lr



    def predict(self,x):
        """Predicts the which class the data is in using the
                                        weights found by the
                                        training """
        w = self.weight
        # Vectorizes the functions
        vect_func_sig = np.vectorize(self.sigmoid)
        vect_func_dec = np.vectorize(self.Decision_boundary)
        prob = vect_func_sig(x@w)
        result = vect_func_dec(prob)
        self.result = result
        return result






#########################################
#
#
# Decision tree
#
#
#########################################


def entropy(data):

    """Finds the entropy from the data by using equation 9.4
                                in the book,
     Introduction to machine learning, Alpaydin, Ethem"""

    A, B  = 0, 0
    # Check if data is either class 0 or class 1
    for i in data:
        if i[0] == 1:
            A += 1
        else:
            B += 1

    length = A+B
    tot = A
    # Avoid division by zero
    try:
        p =  tot/length
    except:
        p = 0
    # Finds the impurity of the data (how much the data is
                                mixed with different
                                classes.)
    # if 0log0 then 0
    if p == 0:
        entropy = 0
    else:
        entropy = -p*np.log2(p + 1e-20) - (1 - p)*np.log2(1 -
                                     p + 1e-20)
    return entropy
```

```
def count_classes(data):
    """Find the amount of each class and return a dictionary
                              with the labels and amount
                              """
    amount = {0:0,1:0}
    for row in data:
        label = int(row[0])
        amount[label] += 1

    return amount



def inf_gain(left,right, parent_entropy):
    # https://en.wikipedia.org/wiki/Decision_tree_learning

    # Get the weight if the child nodes
    p = float(len(left)) / (len(left) + len(right))

    # Return the information gained by this split.
    return parent_entropy - p*entropy(left) - (1 - p) *
                              entropy(right)




class threshold:
    """Makes a threshold object to hold the features and the
                              threshold in the nodes"""
    def __init__(self, feature, thresh):
        # holds features and threshold before seperating into
                              branches
        self.feature = feature
        self.thresh = thresh

    def check(self, data):
        # gets value of feature
        value = data[self.feature]

        # Returns the boolean logic True/False if threshold
                              is met or not
        return value >= self.thresh
    # Return a string representation of the threshold
    def __repr__(self):
        return str(self.thresh)





class leaf:
    """Leaf node, this is where a node becomes classified
                                with the majority class.
                                """
    def __init__(self,data):
        self.amount = count_classes(data)
        if self.amount[0] >= self.amount[1]:
            self.classified = 'Class 0'
        else:
            self.classified = 'Class 1'



class node:
    """A node in the tree"""
    def __init__(self, thresh, index, left, right):
        self.thresh = thresh
        self.left = left
        self.right = right
        self.index = index






def split(dataset, threshold):
    """ Splits the data """
    left, right = [], []
    for row in dataset:
        # Uses the threshold.check method to test if feature
                                    meets the threshold
                                    condition
        if threshold.check(row):
            left.append(row)
        else:
            right.append(row)

    # Return left and right branch data
    return left, right
```

```
def find_best_split(data):
    """
    Finds the best split given that the split yields the best
                                information about the
                               data
    """
    best_ent, bestf, ind, current_entropy = 0, 0, 1, entropy(
                               data)
    for index in range(1, len(data[0])):
        for row in data:
            condition = row[index]
            thresh = threshold(index, condition)

            # Splitting dataset
            left, right = split(data, thresh)

            # Find the information gain
            e = inf_gain(left, right, current_entropy)


            # If maximum gain choose bestfeature,(index) and
                                       update bestentropy
                                        to the best
                                       information gain
            if e > best_ent:
                best_ent, bestf, ind = e, thresh, index

    # Return the information gain, the bestfeature threshold
                               and the feature index
    return best_ent , bestf, ind




class Tree:
    """
    The decision tree, the tree includes the methods
                               build_tree for building
                               the tree with the training
                                data,
    and a node_stats method to get the whole tree structure,
                               and a prediction method
                               which uses the tree built
                               with
    the training data to find the classes from the data.
    The tree object
    """
    def __init__(self):
        self.class0pos = 0
        self.class0neg = 0
        self.class1pos = 0
        self.class1neg = 0
        self.leafnumber = 0
        self.nodenumber = [1]


    def fit(self, data, N, depth = 0):

        """
        Building the tree, here we have a depth variable that
                                        chooses how deep our
                                       tree will go,
        We are here building the tree using recursion (we
                                       call the build
                                       function within the
                                       function).
        """


        # Max depth of the tree
        if depth == N+1:
            # If we have reached our max depth we stop
                                            building and we
                                            make leaf
            # No matter what the information gain is
            L = leaf(data)
            # set the number of leafes
            self.leafnumber += 1
            return L


        # Loading tree print, a way to visualize that the
                                       process is ongoing
        print('Building Tree: {:.0f}% '.format((max(self.
                                       nodenumber)/N)*100))

        # Get the best information gain, threshold of the
                                       node and the feature
                                       index
        inf_gain, thresh, index = find_best_split(data)

        # If we have less then 0.12 information gain, we make
                                        it a leaf
        if inf_gain <= 0.10:
            # Get the number of leaves
            self.leafnumber += 1
            # Make leaf object
            L = leaf(data)
            return L


        # Split the data set into two branches
        left, right = split(data, thresh)

        # Recoursion, here we the branches left will keep
                                               building until all we
                                               have all the leaves,
        # then it goes up to last node and builds the right
                                               branch, then it build
                                               all the left until
                                               leaves,
        # and so on.
        left = self.fit(left, N,depth + 1)
        right = self.fit(right, N, depth + 1)
        # Add depth, where max value of array will yield
                                               depth
        self.nodenumber.append(depth)

        # Returns the node at each threshold point in the
                                               tree
        N = node(thresh, index, left, right)
        return N


    def node_stats(self, node):

        """
        Print out the whole tree structure
        """

        # If node object is leaf, then write the dictionary
                                               of the data inside
                                               leaf and the
                                               classification
        if isinstance(node, leaf):
            print('Leaf: {}'.format(node.amount))
            print('{}'.format(node.classified))
            # {0:2,1:3}
            # Create stats for confusion matrix
            for label in node.amount:
                if label == 0:
                    if str(node.classified) == 'Class 0':
                        self.class0pos += node.amount[label]
                    else:
                        self.class1neg += node.amount[label]
                else:
                    if str(node.classified) == 'Class 1':
                        self.class1pos += node.amount[label]
                    else:
                        self.class0neg += node.amount[label]
            return


        # Print the feature and its threshold for the node
        print('X{} <= {}'.format(node.index, node.thresh))

        # Print if left branch
        print('--------------')
        print('Left branch')
        self.node_stats(node.left)
        print('--------------')


        # Print if right branch
        print('--------------')
        print('Right branch')
        self.node_stats(node.right)
        print('--------------')


    def predict(self, row, tree):
        """
        This function predicts using the pre-made tree, sends
                                               each row from the
                                              data-set into the tree
                                               and find
        which leaf it falls into and classify the the data
                                              with the class
                                              assigned to the leaf
                                              by the training data.
        """

        # If we have a leaf print return the leaf data from
                                              the training and the
                                              classification
        if isinstance(tree, leaf):
            return tree.amount,tree.classified

        # Here we use the threshold found for the nodes in
                                              the training and use
                                              this on the row data
        # coming into the prediction function.
        if tree.thresh.check(row):
            return self.predict(row, tree.left)
        # if the row data do not meet the threshold we send
                                                it to the right branch
                                                 to test it in a new
                                                node
        # with a new threshold
        else:
            return self.predict(row, tree.right)
```

Machinepy2 module, softree part

```
class softTree:
    """
    The decision tree, the tree includes the methods
                                   build_tree for building
                                   the tree with the training
                                    data,
    and a node_stats method to get the whole tree structure,
                                   and a prediction method
                                   which uses the tree built
                                   with
    the training data to find the classes from the data.
    The tree object
    """
    def __init__(self):
        self.class0pos = 0
        self.class0neg = 0
        self.class1pos = 0
        self.class1neg = 0
        self.leafnumber = 0
        self.nodenumber = [1]


    def fit(self, data, N, depth = 0):

        """
        Building the tree, here we have a depth variable that
                                                 chooses how deep our
                                                tree will go,
        We are here building the tree using recursion (we
                                                call the build
                                                function within the
                                                function).
        """


        # Max depth of the tree
        if depth == N+1:
```

```
        # If we have reached our max depth we stop
                                        building and we
                                        make leaf
        # No matter what the information gain is
        L = leaf(data)
        # set the number of leafes
        self.leafnumber += 1
        return L


    # Loading tree print, a way to visualize that the
                                    process is ongoing
    print('Building Tree: {:.0f}% '.format((max(self.
                                    nodenumber)/N)*100))

    # Get the best information gain, threshold of the
                                    node and the feature
                                    index
    inf_gain, thresh, index = find_best_split(data)

    # If we have less then 0.05 information gain, we make
                                     it a leaf
    if inf_gain <= 0.12:
        # Get the number of leaves
        self.leafnumber += 1
        # Make leaf object
        L = leaf(data)
        return L


    # Split the data set into two branches
    left, right = split(data, thresh)

    # Recoursion, here we the branches left will keep
                                    building until all we
                                    have all the leaves,
    # then it goes up to last node and builds the right
                                    branch, then it build
                                    all the left until
                                    leaves,
    # and so on.
    left = self.fit(left, N,depth + 1)
    right = self.fit(right, N, depth + 1)
    # Add depth, where max value of array will yield
                                    depth
    self.nodenumber.append(depth)

    # Returns the node at each threshold point in the
                                    tree
    N = node(thresh, index, left, right)
    return N


def node_stats(self, node):

    """
    Print out the whole tree structure
    """

    # If node object is leaf, then write the dictionary
                                    of the data inside
                                    leaf and the
                                    classification
    if isinstance(node, leaf):
        print('Leaf: {}'.format(node.amount))
        print('{}'.format(node.classified))
        # {0:2,1:3}
        # Create stats for confusion matrix
        for label in node.amount:
            if label == 0:
                if node.classified == 'Class 0':
                    self.class0pos += node.amount[label]
                else:
                    self.class1neg += node.amount[label]
            else:
                if node.classified == 'Class 1':
                    self.class1pos += node.amount[label]
                else:
                    self.class0neg += node.amount[label]
        return


    # Print the feature and its threshold for the node
    print('X{} <= {}'.format(node.index, node.thresh))

    # Print if left branch
    print('--------------')
    print('Left branch')
    self.node_stats(node.left)
    print('--------------')


    # Print if right branch
    print('--------------')
    print('Right branch')
    self.node_stats(node.right)
    print('--------------')
def predict(self, row, tree, threshold_value):
    """
    This function predicts using the pre-made tree, sends
                                       each row from the
                                    dataset into the tree
                                    and find
    which leaf it falls into and classify the the data
                                    with the class
                                    assignet to the leaf
                                    by the training data.
    """

    # If we have a leaf print return the leaf data from
                                    the training and the
                                    classification
    if isinstance(tree, leaf):
        ss0 = tree.pc0
        if ss0 >= threshold_value:
            return 'Class 0'
        else:
            return 'Class 1'

    # Here we use the threshold found for the nodes in
                                    the training and use
                                    this on the row data
    # coming into the prediction function.
    if tree.thresh.check(row):
        return self.predict(row, tree.left,
                                          threshold_value)
    # if the row data do not meet the threshold we send
                                    it to the right branch
                                       to test it in a new
                                    node
    # with a new threshold
    else:
        return self.predict(row, tree.right,
                                          threshold_value)
```

References

Ethem Alpaydin. *Introduction to machine learning.* The MIT Press, third edition, 2014.

Josh Gordon. Decision tree, 2017. URL `https://github.com/random-forests/tutorials/blob/master/decision_tree.ipynb`. [Online; accessed 04.11.2018].

J.R. QUINLAN. Induction of decision trees, 1985. URL `https://link.springer.com/content/pdf/10.1007%2FBF00116251.pdf`. [Online; accessed 07.11.2018].

Linda Shapiro. Information Gain, *. URL `https://homes.cs.washington.edu/~shapiro/EE596/notes/InfoGain.pdf`. [Online; accessed 07.11.2018].

Troy Walters. Logistic regression and roc curve primer, 2018. URL `https://www.kaggle.com/captcalculator/logistic-regression-and-roc-curve-primer`. [Online; accessed 07.11.2018].

University of Nebraska Medical Center. The area under an roc curve, 2018. URL `http://gim.unmc.edu/dxtests/roc3.htm`. [Online; accessed 06.11.2018].