Final Year Project Report

Full Unit – Final Report

Comparison of Machine Learning Algorithms

Daniil Adamov

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Ruth Blackwell

Department of Computer Science
Royal Holloway, University of London

April 26, 2021

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 10210

Student Name: Daniil Adamov

Date of Submission: April 26, 2021

Signature: Daniil Adamov

Table of Contents

Abstract

Artificial Intelligence (AI) is an important subject in modern life, and is used, for example, in driverless cars, face recognition and even trader bots, making thousands of profitable transactions a second. This is all based on mathematical algorithms that have been improved and combined to give astonishing results. This advancement stems from a more basic approach - Machine Learning.

Machine learning is the implementation of the algorithms that support the science of AI. They are the bare backbone logic. Machine Learning algorithms can learn from huge sets of data. For this report, a number of different machine learning algorithms have been implemented. The performance and uses of these different algorithms are compared. The results show that even simple algorithms can produce accurate predicted results, which is fascinating considering that this area of Computer Science is undergoing constant innovation, and there are even better implementations of this complex logic.

Chapter 1: **Introduction**

I would like to begin by going more in depth of what is considered Machine Learning. Nowadays the phrase ‘Artificial Intelligence’ is thrown around, often without the true understanding that it is Machines that demonstrate intelligence, through the ability of making decisions and/or performing tasks like deploying the brakes to avoid collision in modern cars. If you don't live under a rock you have heard of or perhaps seen Tesla autopilot in action. These products serve a great example to demonstrate huge advancements in the field of Machine learning. However machine learning is not to be confused with Artificial Intelligence as they are not the same. The official definition of Machine Learning is “the study of computer algorithms that improve automatically through experience”[1]. Machine learning is particular implementations of complex mathematical methods to analyse and make predictions based on real world data. Whereas Artificial Intelligence is a broader vision, pursuing a human-like intelligence and even superseding it in some industries.

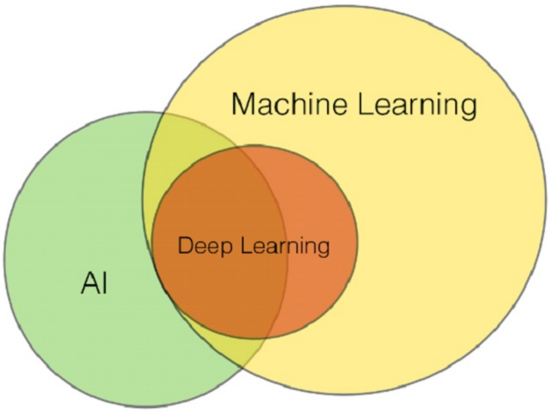


Figure 1. Machine learning, AI and Deep Learning visualised.[2]

As already mentioned Machine Learning algorithms operate with real world data referred to as training data. Training data, as the name suggests, is used by the algorithm to form a model for interpreting data. The model is the “physical” representation of Machine Knowledge formed from the quintessence of Mathematical algorithms and training data. A model can be used on unseen data (testing data) by “feeding in” the features of samples and expecting a label for that sample from the model. For a good example of what big data (*training and testing data*) is like, let's assume that we measured the length and width of all leaves in the forest and noted down the species of the trees this leaf belongs to. After selecting an appropriate algorithm, we would get a model. This model would “compare” different widths and length of leaves (*features*) to be able to predict the species (*label*) of the tree this leaf came from. A model is considered good if it matches the reality and gives accurate and correct labels to unseen samples.

Motivation and Aims

After the second year course introduction to machine learning algorithms, I was fascinated by programs learning and making decisions. At first I thought these decisions the machine makes are a plot of magic. Not long ago I read an online course/book about ‘neural networks and deep learning’ by Michael Nielsen. I came to understand that even the smallest tweaks in a machine learning algorithm can have a vast impact on the outcome. So I was inspired to investigate different machine learning algorithms, and objectively quantify their ‘usefulness’ in solving similar problems. This is important to me because machine learning and artificial intelligence is at the forefront of evolution, and as a future Artificial Intelligence specialist, I want to know for myself which algorithms have faster learning rate (how fast an algorithm can see a pattern emerge from data and start predicting outcomes), which algorithms can be modified for alternative outcomes and compare those outcomes. And lastly I could find out which machine learning algorithms are computationally faster. It is important for me to understand the nuances of these algorithms, their strengths and weaknesses and possibly investigate the real world applications of these algorithms and ways they could be remodified to fit for real life situations more appropriately.

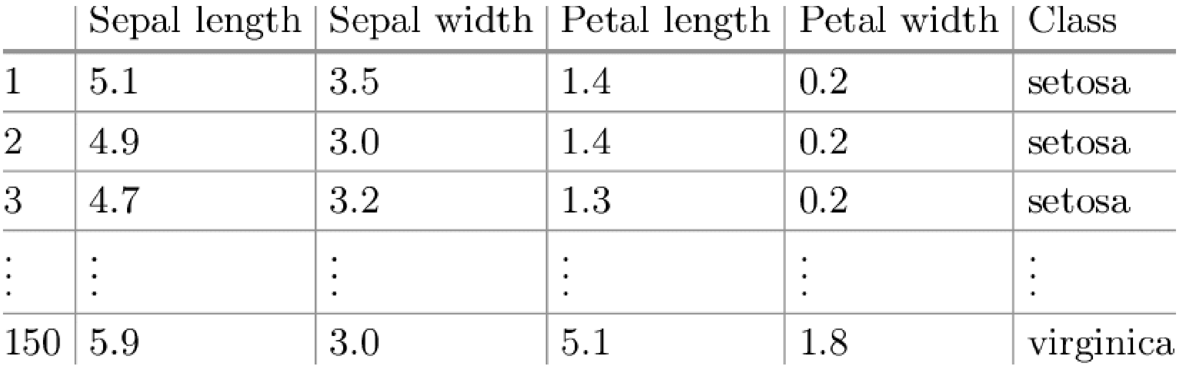
The main aim of this project is to compare machine learning algorithms. In order for me to do that I will first need to do that I want to implement these algorithms in code. My first step was broadening my knowledge by doing some further reading about machine learning and artificial intelligence as a whole. This was crucial for the foundation of the project, as it gave me a general idea of the realm of possible things that I could do to compare machine learning algorithms.

Now, let's look into this problem with a bit more structure and detail. I would like to start off with simpler algorithms (like nearest neighbour algorithm or K nearest neighbour) as proof of concept and then advance to more complex ones. In order to test the learning capabilities of these algorithms I will need relevant data sets e.g. MNIST and USPS for teaching the algorithm to distinguish handwritten digits. Once both of these steps are complete I can start the testing stage, where I will explore the capabilities of these algorithms and compare their overall performance in these tests. A crucial thing to consider is dataset division into training and test sets.

Chapter 2: **Background Theory**

In this section, describe the theory behind the algorithms which you have coded for this report. highlight your own achievements.

First of all I would like to describe two types of machine learning, which are essential in understanding how machines learn. The first type of learning is *supervised learning*. Let us understand what is meant by that. Typically when we say supervised we mean to direct and observe the execution of a task or an activity. This is exactly what we would like to do in this project, by having a prearranged dataset, find some sort of pattern in the dataset e.g. to distinguish setosa and virginica plants by their petal and sepal dimensions. This is similar to teaching a child who does not know anything about the world, by showing him squares and circles, and telling him which one is which. The telling part, where we show squares and circles is the supervision part. A child might get confused if you show him a triangle, as he has never seen it before and would not be able to make a correct prediction on what that object might be, similar to supervised machine learning algorithms.



	Sepal length	Sepal width	Petal length	Petal width	Class
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
⋮	⋮	⋮	⋮	⋮	⋮
150	5.9	3.0	5.1	1.8	virginica

Figure 2. Iris dataset snippet.[3]

Now let's look at a common example of a dataset, iris dataset (see Figure 2), which could be used to train a machine. Just like with a child example where we tell the child which shape is which, the dataset for the machine has to be labeled. We can see that every line (representing an instance of some observation) has a label in the Class column. This is essential for supervised learning, as data without a label is useless for training a model, as it does not know what it is ‘looking’ at.

The other important part of the dataset are the features which correspond to a label. In the example with iris dataset features are the sepal and petal widths and lengths. This is somewhat similar to a child touching and seeing the objects. If an object is round and has no sharp edges a child would say it is a circle. The same principle is apple to machine learning on big datasets. Our model will be trained on those features and labels to get some sort of pattern to emerge, where if we give this model a shape, or a leaf and it would be able to give a label to that data.

The other type of learning is unsupervised learning. As the name suggests the supervision factor is eliminated, and datasets will not be labeled. This learning approach is based on a more complex idea, where we let the model work on its own to discover patterns that might not be visible to even a human eye. As the model has little to no information about the dataset or the expected outcomes, with the use of advanced mathematical algorithms the machine tries to find groups or clusters, look at density of datapoints and even utilize dimensionality reduction. These ideas are far too complex to evaluate and discuss in this project time frame, so the main focus of this project will be purely supervised learning.

All supervised learning algorithms work by the same principle. The algorithm is given a set of training features (commonly referred as X) with corresponding labels (commonly referred to as y) to formulate some rules and build 'knowledge' which is a model. However for different types of learning problems we need to apply different learning approaches.

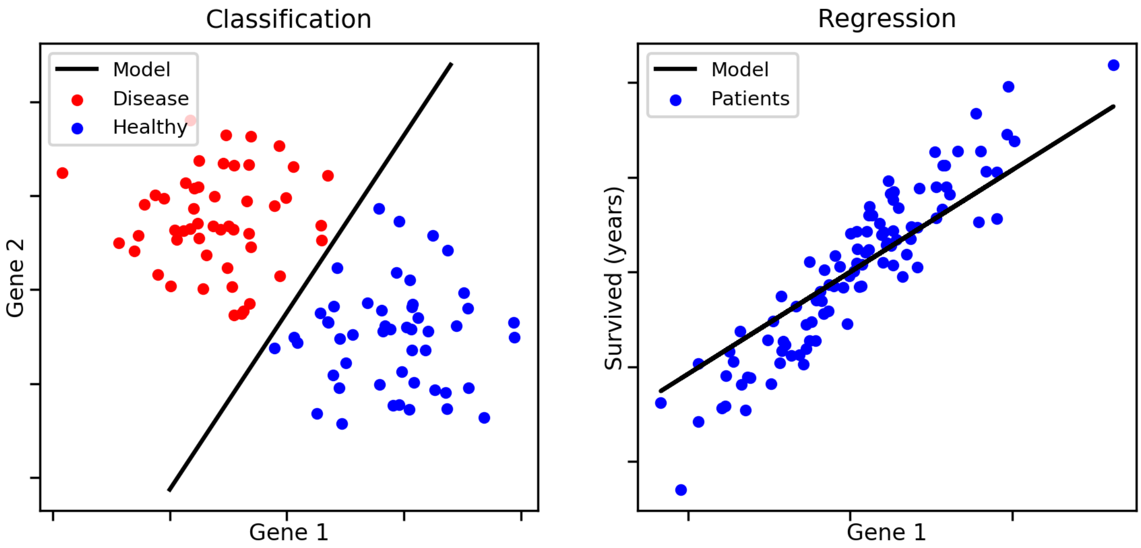


Figure 3. Classification vs. Regression Machine Learning Problems. [4]

The two most common types of learning problems are *classification* and *regression*. Classification aims to build a model which can be used to distinguish between different classes. There is always a set number of classes and each set of features has to correspond to some class. If there are only two classes, that is binary classification, it is like yes or no. However in most cases there are more than two classes to distinguish from - multiclass classification.

Regression also produces a model, however there are an infinite amount of possible labels, so for some training features and their labels the model has to produce a continuous output variable (see Figure 3). A good example of a regression model is estimating housing prices, based on factors like distance from public transport, crime etc. The easiest way to know which one is which is to figure out how many possible labels there are. If you can do that, most likely this will be a classification problem e.g. is the animal in the picture a dog or a cat. If there are too many to count it is a regression problem e.g. housing prices.

It is important to note that the process of machine learning is usually a two phase process. First you let the algorithm explore the dataset to produce a model - Learning step (exploration). Then that model is used to make predictions on unseen samples - Prediction step (exploitation). Algorithms that function in this way are fast at giving a prediction for some input value, because they have already built a model in the exploration state, and all they need to do is use the model to give the prediction. They do not need to analyse all the training samples (there could be thousands of training samples) in the exploitation stage. However there are some exceptions to this, like Nearest Neighbour algorithm, which I decided to explore in more detail for this project.

In order for me to make an accurate comparison of the algorithms I will solely focus on classification for this project, as some algorithms that will be implemented in code focus on classification, rather than regression problems. It is also easier to quantify the number of errors for classification problems, because the predicted label is either correct or wrong, there is no inbetween. On the other hand for regression problems the predicted label can be in range of the true label and quantifying the error of this range for thousands of data samples can be challenging. To make the most accurate and honest comparison of machine learning algorithms I decided to focus on the same type of machine learning problem (classification). To strictly follow the scientific method it is necessary to keep all the datasets exactly the same (constants) and alter the algorithms (variable) to make a comparison which is accurate to the best of my ability.

Nearest Neighbour Algorithm

Nearest Neighbour algorithm is a sort of exception to the general way machine learning algorithms work. Instead of building a model which can then be applied to unseen samples, when it encounters an unseen sample it looks for the 'closest' sample (sample with similar input values) in the training set this is called an Instance-based Learning Algorithm. The prediction for the unseen sample will be the label of the closest sample. This algorithm works on the assumption that similar instances of the same type are in close proximity to each other, similar to the saying "Show me your friends and I will tell you who you are".

It is important to define what is the 'closest' sample. This depends on the type of distance metrics the algorithm implements, which has to be selected by the programmer to best fit a particular dataset. There are lots of ways to calculate this distance, but I will cover the most common ones, describe what they are useful for and select the best fitting one for my datasets.

Euclidean Distance

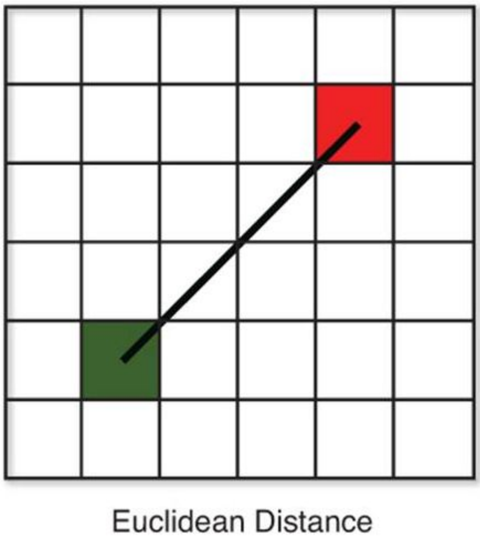


Figure 4. Euclidean Distance visualised in 2 dimensions. [5]

Euclidean distance is a measure of true straight line distance between two points in Euclidean distance. This is the most common way to interpret distance in a 3 dimensional space and the type of measurement we are used to. Imagine a plane with two point objects (an object that has no volume or area it is just a point in space), drawing a straight line from one to the other and measuring the length of the line will yield the euclidean distance between them. However in machine learning we are rarely dealing with just a plane, we are likely to be working in some n-th dimension. Where n also represents the number of features the instance has. So how would this distance be calculated, if we can not take a 'ruler' and draw a line in some dimension we cannot see, because we are 3 dimensional creatures. This is where some simple pythagoras maths saves the day.

$$d(x, y) = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2}$$

Figure 5. Euclidean distance equation.

This is the equation (see Figure 5) for finding the Euclidean distance in the n-th dimension between two points x and y. This is what SKlearn library in python uses for its nearest neighbour algorithm. This algorithm is easy to understand, easy to implement and it is extremely reliable for Euclidean spaces, as it does the most logical thing finds the shortest, straight line distance between two points.

Manhattan Distance

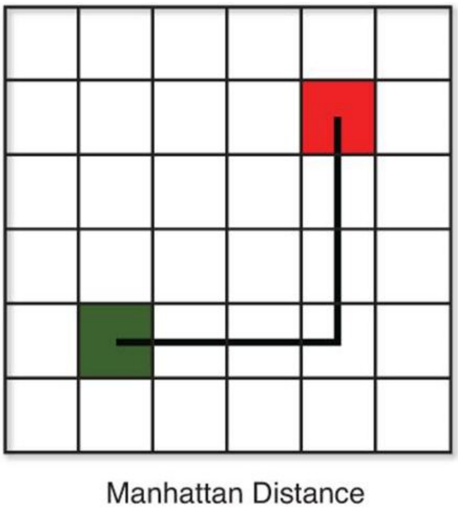


Figure 6. Manhattan Distance visualised in 2 dimensions.[5]

Manhattan distance is a simpler calculation than Euclidean distance, as it measures distance by summing the absolute difference of the Cartesian coordinates of two points (see Figure 6). This is similar to travelling in city blocks where you could not travel in a straight line between two points, like you would in a helicopter or a plane. This is useful because there is no need to perform complex computations like working with square roots. It is also applicable in n number of dimensions as seen from the equation below.

$$d(x, y) = \sum_{k=1}^{n} |x_k - y_k|$$

Figure 7. Manhattan Distance equation.

As seen from the equation Manhattan distance (see Figure 7) is calculated by finding the sum of the absolute difference of two points x and y. Where n is the total number of dimensions for this point in space.

Minkowski Distance

Minkowski distance is a way of measuring the distance between two points in a vector space over real or complex numbers. It is only applicable in the space where distances have a strictly positive length. It is a combination of both the Manhattan and the Euclidean distance algorithms, but it can also be used to calculate curved line distance between two points. So in a sense it can calculate everything in between and including the Manhattan and Euclidean distances. Thus using the Minkowski distance we can adjust and tune the cost of error in vector space.

$$d(x, y) = (\sum_{k=1}^{n} |x_k - y_k|^p)^{\frac{1}{p}}$$

Figure 8. Minkowski Distance equation.

The formula for Minkowski distance (see Figure 8) is very similar to the ones for Manhattan and Euclidean distances, because it is a generalisation of both of the algorithms. Where n is the number of dimensions of points x and y, and p is some real number. For example if 1 is selected it is Manhattan distance and if 2 is selected it is Euclidean distance. The variation of this constant defines the curvature of the line connecting the two points.

Conclusion

As Minkowski distance is used for finding a distance in a vector space, which is exactly what I will be dealing with in my datasets in this research it fits perfectly for the given task. However being a generalisation and which adds complexity to the equation can noticeably slow down the algorithm for finding the Nearest Neighbour on a large dataset, because of the need to perform additional mathematically heavy operations. Manhattan distance would be ideal for finding the nearest neighbour if the goal of the algorithm is to work as fast as possible, as it only uses subtraction and addition to calculate the distance between two points in vector space. But because of computationally efficient numpy libraries and methods such as linalg.norm (further discussed in chapter 3) I used numpy operations in this project to calculate Euclidean distance in multidimensional vector space.

K Nearest Neighbours Algorithm

K Nearest Neighbours algorithm is a slight modification of the Nearest Neighbour algorithm as the name suggests. K stands for the number of nearest neighbours the algorithm looks for in the training dataset instead of just one. Once the k number of neighbours in the training set are found, each one of their labels counts as a ‘vote’ for the predicted label, and the most common label is the predicted label for the test sample. This algorithm still utilises the same calculation for the distance of the nearest neighbours but because it takes into account more samples from the training set to make a prediction it can be seen as a more precise way of giving a prediction for an unseen sample.

It is interesting to explore this statement to figure out if the extra samples taken into account actually give a better prediction. Perhaps on some data samples which are close to a number of different classes in classification problems it could give a more precise estimate of the true label. Also this algorithm can be useful in exploring the assumption that similar instances of the same type are in close proximity to each other, as it would take in account more than just one closest sample.

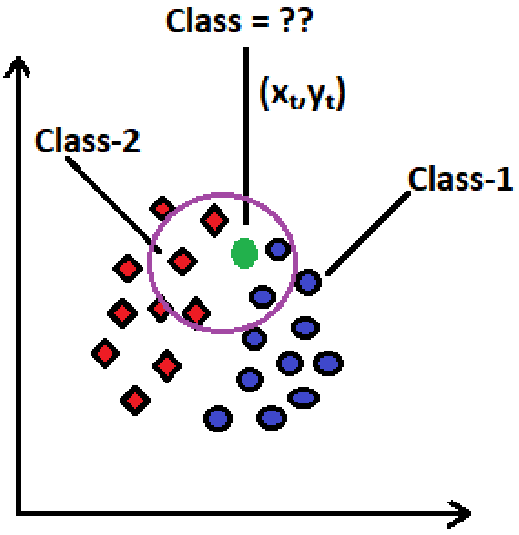


Figure 9. Classification of an unseen sample with KNN. [6]

Although KNN takes in account more data samples it is possible to make an educated assumption that on edge cases (samples that are not common and could be seen as anomalies) this algorithm could give a wrong prediction label. This could happen because these test samples lie close to training samples with different labels (see Figure 9) and the closest neighbour (in blue) might have the correct label for the given sample (in green), but if a number of samples are taken into account (e.g k = 5) the predicted label would be incorrect. This happens if there are more samples of the wrong class than there are of the correct class closer to the test sample and after taking a 'vote' an incorrect prediction is made as seen from the graph.

Conclusion

It is interesting to explore what number of nearest neighbours gives the more accurate prediction for an unseen sample, by varying the number of neighbours in the algorithm. Finding the correct number of neighbours is crucial for reaching the best precision, keeping in mind that the more neighbours the algorithm has to deal with the more computationally demanding it is, which could also be interesting to explore.

Decision Trees

Decision trees are an example of what a typical machine learning algorithm is. It has the earlier mentioned two step process of exploration where the decision tree model is built and the exploitation process where the model is used to give a very quick prediction label for the unseen sample by traversing the tree. This process is extremely quick because trees are easy to navigate following the decision rules set in each node to traverse to the bottom of the tree to the last node which contains the prediction. Trees are the one of most effective ways of storing ordered data.

It is important for later understanding to know some terminology used for describing a tree.

- **Root Node** – Represents the first node of the tree (similar to head of a stack) can be used to access the rest of the tree. It contains the whole training dataset which is usually split into two or more homogenous sets by some condition stated in the node

- **Splitting** – Is a process of dividing a subset contained within a node into two or more nodes each containing the subdivision of the subset of training data.

- **Node** – Represents a subset of training samples. It also contains the split for the subset it represents.
- **Parent/Child Node** – A node with a subset which has a division is a parent node to the child nodes it creates. Child nodes contain the subdivisions of the parent nodes's subset.
- **Leaf Node** – The final node of the tree. It does not split, because as a final node it contains the prediction label or some information we want to access after traversing the tree.
- **Branch** – It is a subsection of the entire tree. It can contain multiple nodes.

Decision trees can be both used for regression and classification. However due to the fact that decision trees model contains the prediction label in its leaf node regression problem would require leaf nodes to represent some range (Continuous Variable Decision Tree) for a given sample and often result in creation of trees with a big depth for the model to be precise on . Trees with big depth are computationally heavier to create for large datasets and I do not have a powerful enough machine which could do it in a timely fashion, without taking days, so for this project I will focus on classification problems for my decision trees algorithm.

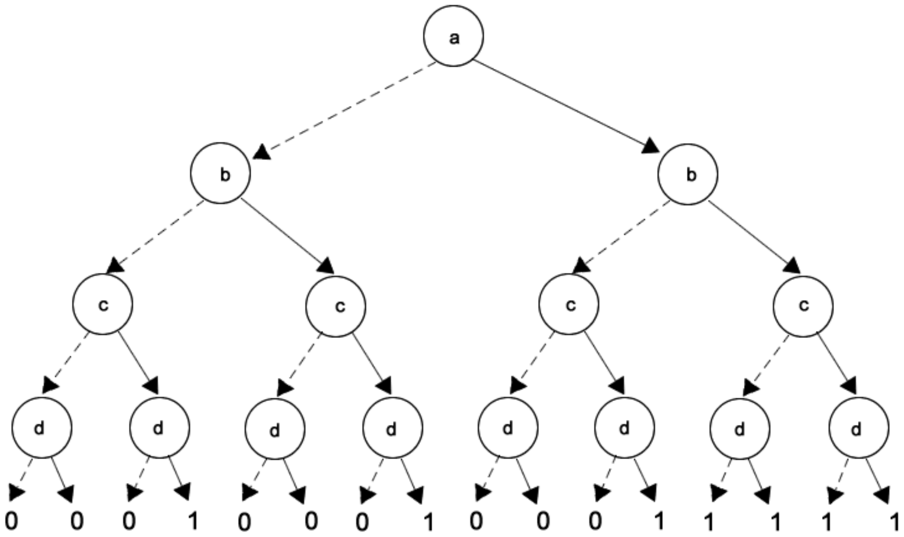


Figure 10. Decision Tree visualised.[7]

Categorical decision trees are used for making a model for classification problems. The model will contain prediction labels in the leaf nodes, which correspond to some class in the dataset. The leaf nodes are reached by traversing the tree, starting from the Root node. Each of the decision nodes (all nodes apart from the leaf nodes) will contain a statement regarding one of the features of the dataset which is applied to a test (unseen) sample. Based on that statement we descend to one of the node's children which correspond to the possible answer to the earlier test. This is done recursively until a leaf node containing the prediction is reached.

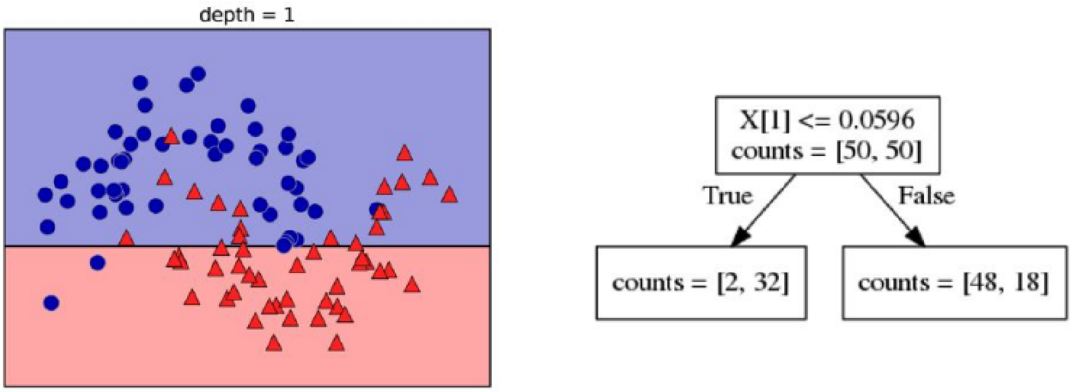


Figure 11. Decision Tree first split. [8]

When building a decision tree the most important objective of each node is to select the best attribute (feature) to be tested. This attribute has to provide the best possible split of the dataset. A split in the first few nodes is unlikely to have a 100% precision (see Figure 11). There will be samples of the ‘wrong’ class present in the split. However the more splits are created the more precise the model becomes, reaching 100% precision where a division by some attribute will only contain one type of samples in each of its child nodes. When this happens we stop splitting further, to prevent the tree growing and creating unnecessary nodes which do not provide any additional information in making the model more accurate.

Split Selection

As already mentioned, selecting the correct attribute and its value is the most crucial part of making an accurate decision tree model. A common way to select the most informative split is to select one feature from the dataset and vary its value to get the least error rate, where there are the most amount of red samples on one side of the split (see Figure 11) and barely any of them on the other side of the split. The less red samples end up on the wrong side of the split the smaller the error rate will be, which signifies the informativeness of the split. This is done for all features until the best and most informative feature and its feature value is selected.

Selecting the correct feature value is the hardest part of the algorithm, because once it is selected we can get the error rate for it and find the feature which gives the least error rate with some threshold. There are multiple algorithms to find the most informative threshold, however in my work I will try to implement my own improvised way. I want to count the different values of training samples to find the most common value of some feature with label a. Then I will carry out the same operation for samples with label b on the same feature. This will give me two values for a feature, one corresponding to label a and one to label b. Then I will find the average of those numbers and call that the feature threshold. I will then go through all the features of the training set to determine which one is best and gives the least error rate. That feature number and feature threshold will be the value of the node. All samples that have the feature number larger than the one in the node will be assigned to a child node to the right of the parent node, lower ones will be assigned to the left child. This process is repeated for the children and the children of children and so on, meaning this process is recursive in its nature, until no further splits can be made or some depth n is reached after which I do not want to split the nodes any more, as this could create a massive tree, which is computationally demanding to compute. Also trees with a big depth can have a more defined problem of overfitting.

Overfitting

Decision Trees tend to have a common problem which is overfitting. It is when a tree has trained so well on the training set that it performs poorly on any test sample, because it has solely focused on fitting to the training set. In the worst case such a tree would contain only one sample in its leaf nodes. It occurs when there are too many divisions which are sometimes unnecessary. This happens when there are some anomalies in the training set which are not seen as anomalies by the tree, but the tree still takes those samples in account when creating a model. There are two ways to account for that pruning and random forests.

Pruning helps overcome the problem of overfitting by removing some of the nodes, starting from leaf nodes, which are overtrained. This is done by further splitting the training dataset into two: the actual training set and the validation set. The tree model is made on the new training set. Then it utilizes the validation set to determine which leaf nodes do not add additional accuracy, but instead return the wrong label for the validation samples. Those leaf nodes are then deleted and the parent node of that leaf node now becomes the new leaf node. This process is carried out recursively until the overfitted nodes are removed.

Random forests work in a slightly different way. Instead of modifying an already built tree, it tries to make the process of tree model creation more random, in an attempt to avoid those anomalies in the dataset. It works by using a technique called bagging. It splits the dataset into smaller subsets with a predetermined amount of samples selected randomly. A tree model is built on each randomized subset. Then those trees are combined into one, by selecting the averages for the predictions. The averages taken from randomizing the datasets get rid of the anomalous samples making the tree less precise on the training dataset, but more accurate on unseen samples.

Conclusion

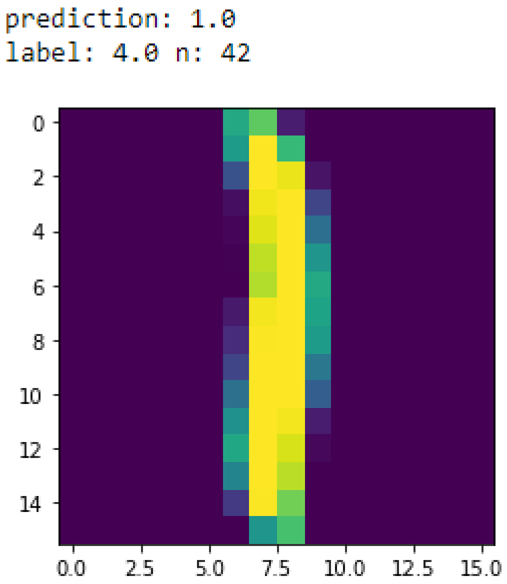
The quality of the Decision Tree model is highly dependent on the algorithm used for splitting. In this project I decided to implement my own way for calculating the threshold which will split the subsets. I will keep in mind that creating a decision tree model is computationally demanding especially when implemented only using Python 3.7 and try to minimise the complexity of calculations done for finding the feature threshold. By nature Decision Trees are highly overtrained unless using the methods to prevent overtraining. I will keep this in mind creating my own implementation of the algorithm.

Chapter 3: **Extensions**

In this section I will outline what extensions I have used when implementing the outlined algorithms in Python 3.7 code. It is useful to understand why they are necessary and why I decided to use them.

As I am working with big datasets, which can contain upwards of 10000 samples with multiple features and python code is computationally heavy I will try to use as much of Numpy library, which is written in C and C++, when it is applicable. This makes all the computations on massive arrays much less demanding. I will also use numpy arrays for storing training and testing samples and their labels in four different arrays.

When programming and debugging the code it would also be useful to visualise the data samples, instead of just looking at massive arrays filled with numbers. I will use a library plot. For example for plotting the greyscale features for samples representing digits.



```
prediction: 1.0
label: 4.0 n: 42
```

Figure 12. Example of using plot extension

Later in the project I ran into a problem when making a model for decision trees, as it would take more than an hour to create each tree with 10000 samples from the Mnist dataset and my computer would randomly shutdown. I decided to use a pickle library that would create a file copy of the trees as they were created. This allowed me to load the trees which were created earlier and keep going without losing any progress.

Chapter 4: **Datasets**

In this section I will outline the two datasets I obtained for the two algorithms. I wanted my algorithms to have some sort of significance in the real world, and decided to train them to recognise handwritten digits.

The first dataset was MNIST dataset with handwritten digits. It has already been preprocessed, which also made my job easier. This dataset is considered a standard for training digit recognition, as it contains 50000 samples for training set and 10000 samples for testing with variation of more clear digits, which are easy to classify and some which are more difficult even for a human eye (see Figure 12, which looks like 1 but is actually 4). This variation is a good test for each algorithm. Each sample in the dataset has 784 features (28*28 pixels) with grayscale values ranging from 0-255 and a corresponding label. This dataset is well preprocessed containing little to no ‘noise’ which is very common with handwritten digits, around the edges of a digit. This should improve the results and the accuracy when training on this dataset.

As the second dataset with handwritten digits I decided to use the USPS dataset. It contains a total of 9298 samples of handwritten digits and automatically scanned by the US postal service. Each sample contains 256 features corresponding to a pixel (16*16 pixels) on the scanned image with grayscale also ranging from 0-255 . It is also preprocessed especially nicely, with little to no ‘noise’ around the edges of the digit. As seen both datasets are fairly large as a result they should produce a somewhat accurate model.

It is important to reiterate the fact that both datasets were preprocessed, which made my task a lot easier, as sometimes correct preprocessing which gets rid of ‘noise’ in the samples is more important than a good algorithm. This is something that would be interesting to explore in the next iteration of this project, if I get a chance to improve it some time later. It would be interesting to test the models from two algorithms on digits that I myself would write and scan, without preprocessing them. This would give a real world representation of the worst case scenario and test the accuracy in that case. Also each pixel has a grayscale value ranging from 0-255 which is also useful, as it means that I do not have to preprocess the datasets to make them fit for creating an accurate model.

Chapter 5: **Software Engineering**

In this section I will outline the software engineering techniques that I have applied in producing the software.

All of the code was done in Python 3.7 so I decided to use anaconda to create a virtual environment on my machine to run a Jupyter Notebook, which is a very useful and intuitive code editor, allowing me to run code as I write it with ease in the same window.

I have also utilized GitHub when engineering the software. GitHub is a useful tool in creating a software, with features like version control, which allowed me to roll back some of the incorrect changes to the code easily. It is also a useful tool when working in a team allowing more than one user to work on the code with features like branching, which allow simultaneous testing and working on new code by more than one programmer. It is also useful for accessing the project from anywhere in the world if there is a machine which is connected to the internet. This also came in useful as I have two machines and committing and pushing a bit of code on one machine could allow me to pull that code to the other and continue working on a different machine.

When working on the code I had a product owner to whom I had to report at the end of each sprint, or every two weeks. This demonstrates agile methodology of code development, as the feedback loop on the output of the programmer is shortened. This allowed me to adjust the project to what was a better implementation in the eyes of the product owner and made the programm better suited for the needs of the client. This process is crucial in achieving a software that meets the demands of the product owner, as often in the industry it is hard for a person to convey the ideas of what the product should look like in the end, and often there are a lot of changes made to the initial idea, as time goes on.

As a software developer, I made a plan with the product owner, to satisfy the time frame required for the project. However due to covid and other personal issues it was sometimes hard to follow the delivery dates. This should be taken in account in the future, and sometimes it is a good idea to get as far ahead when possible, because there could be unpredictable circumstances, which can affect the output of the code. As a result most of the project I was trying to catch up to the deadlines instead of gliding ahead.

Chapter 6: **Implementation**

In this chapter I will outline the implementation of these machine learning algorithms, some crucial decisions about data structure and explain some of the more complicated bits of code which make my implementations of machine learning algorithms different from everybody elses. This will contain some more important methods and snippets of them, which need explaining, because they are slightly unusual and represent my own work and ideas about the implementation of certain algorithms.

K Nearest Neighbours

Instead of implementing two different methods for Nearest Neighbour and K Nearest Neighbour I decided to make just one method. This is because it is possible to use the value of k=1 which is equivalent to Nearest Neighbour algorithm.

```
def neighArray(X_testSample, X_train, y_train, neighNumb):
  nearDist = {}
  for i in range(X_train.shape[0]):
    eucSum = np.linalg.norm(X_train[i,:] - X_testSample)

    if len([*nearDist]) < neighNumb:
      nearDist[i] = eucSum
    else:
      maxDistKeyVal=max(nearDist,key=nearDist.get),max(nearDist.values())

      if eucSum < maxDistKeyVal[1]:
        del nearDist[maxDistKeyVal[0]]
        nearDist[i] = eucSum

   return [*nearDist]
```

The code above is the implementation of finding K nearest neighbours and returning them in an array. The method is called neighArray, which has 4 arguments. The first one passes the sample we are testing, and finding the neighbours for. The second and third are the dataset with features and the dataset with labels. The second is used to find the indices of samples which are the closest to the testing sample by looking at the feature values and utilizing the Euclidean distance to find the closest samples. Once the indices are found we know which sample numbers are closest, through that we find which sample number has to be returned, because X_train (feature values) and y_train (labels) have corresponding numbering. The fourth argument is the number of neighbors we are looking for, if I was to implement just Nearest Neighbour algorithm this value would not be present, as we are always looking for one neighbour.

I made a decision to make a dictionary for storing the Nearest Neighbours values. The key of the dictionary is the index of the Neighbour and the value is its Euclidean distance to the testing sample. I could have used an array for this, but that would mean creating the array the size of the dataset, so a dictionary where I can map a key to a value is a much easier and more concise implementation of storing values. The keys and values for the nearDist dictionary are selected in the for loop, which iterates through the whole length of the training set. For each training sample the Euclidean distance to the test sample is found. Then in the if statement nested in the for loop we check that the size of the dictionary nearDist (len([*nearDist])) is strictly smaller than the number of neighNumb (4th argument of method neighArray) corresponding to the number of neighbours to look for. If that is the case we fill the dictionary with a starting value, which will be improved in later iterations. If that statement is not satisfied and there is already a max number of neighbours in the dictionary we look for the furthest neighbour, finding the max Euclidean distance in the dictionary's values. If the Euclidean distance to the current sample is smaller than the maximum distance in the dictionary, that value and key get deleted and replaced with the current one. In the end return a dictionary filled with indices as keys and shortest Euclidean distances as values representing the Nearest Neighbours.

```
def knnClassifier(X_testSample, X_train, y_train, neighNumb):

  nearDist = neighArray(X_testSample, X_train, y_train, neighNumb)
  knnLabelVote = {}

  for indX in nearDist:
    label = y_train[indX]

    if label in [*knnLabelVote]:
      knnLabelVote[label] += 1
    else:
      knnLabelVote[label] = 1

  return max(knnLabelVote, key = knnLabelVote.get)
```

Method knnClassifier then uses those dictionaries to make a prediction for the unseen sample. It also has the same 4 arguments, because they are also used inside the method for the nighArray method, to get the dictionary for Nearest Neighbours. First we get the dictionary by passing the 4 arguments to the neighArray method inside the knnClassifier method and create a new dictionary knnLabelVote which has key as label of training samples of Nearest Neighbors and value will be the number of times this label is seen in the Nearest Neighbours. To count the number of labels. In the for loop for each Key containing the index of the sample, we find it's label. If that sample is already in the dictionary we increment it's value in the dictionary, which corresponds to the frequency of that label in the K number of nearest neighbours. If it is not already in the dictionary a key with current label with value 1 is added, this being the first instance of a given label. After the 'votes' are cast, we find the maximum value inside the dictionary and return the key mapped to that value, which is the most common label in this K amount of neighbours for a given sample. Thus knnClassifier returns the prediction label for a test sample.

Decision Trees

It is important to first talk about the data structure which I decided to use to implement decision trees. I made a class decision tree, which is an instance of a node in a decision tree. Because the creation of decision trees is a recursive process we can call a decision tree root node as the decision tree itself, which will later represent the whole decision tree as its children are built. We can access any part of the tree just from the root node. So let's look at what each individual node will contain.

```
class DecisionTree:
    def __init__(self, featureNmbr = None, featureThreshold = None,
predictedLabel = None):
      self.featureNmbr = featureNmbr
      self.featureThreshold = featureThreshold
      self.predLabel = predictedLabel
      self.right = None
      self.left = None
      self.subSetVolume = None
```

The class constructor has 4 arguments. It has self which is used to represent an instance of a class, so when a decision tree is created it is possible to access all of its attributes and methods. featureNmbr is initialised to None, because when the node is first created it does not yet know where the most informative split will be, same goes for the featureThreshold. When the most informative split is found those 2 arguments will be changed from None to some whole number. The predictedLabel is initialised to None for the root node. However after the first node this will take the value of the most common label within the subset of the current node. This value is the value that will eventually be used in the exploitation stage, to give a predicted label for an unseen sample. Each node also has self.right and self.left corresponding to the two children nodes. The right node will contain the subset after the split, where one of its features corresponding to the feature number has a bigger or equal to value of the featureThreshold. The left node will be strictly smaller. self.subSetVolume is mainly used for debugging and in case there is a need to see how many samples there are in the subset of the current node. This can be useful to see is the splitting is done correctly.

```
def featureThresholdSelectorV3(self, X_train, y_train, featureNmbr, path):

  indicesOfTreeLabel = 0, 1
  errorRate = 1

  self.subSetVolume = 0

  instancesOfFeatureLabel = np.zeros(256)
  instancesOfFeatureNotLabel = np.zeros(256)
```

Now let's look at the most important method in the class DecisionTree which finds the most informative split for some feature number (featureNmbr). It takes in 5 arguments. self is used to access the argument of the node that the featureThresholdSelector is working in. X_train and y_train are features and labels of training samples of a training set selected. featureNmbr is the number of feature we want to find the most informative split for. Lastly path argument, represents the constraints for the current node’s subset (explained later). This argument represents the divisions of the earlier nodes, so for the first split this will be an empty array. After the first split for the right node it would contain the featNum (feature number of the split in parent node), featThr (feature threshold of the split in the parent node), and an operator (if this is a right node it is greater or equal to, if it is a left node it is strictly less than).

This class has some initial variables that are used in the method or are returned. indicesOfTreeLabel represents the label we choose for this node. Each tree only works with 2 labels. So indicesOfTreeLabel are used to represent which node of a tuple of labels of a tree is predominant in the current node. If it is 0 we use the first label of tuple for the node. If it is 1 we select the second label from the tuple for the current node. Error rate is a base error rate of the current feature number and returned by the method to indicate the informativeness of the division. subSetVolume is a class variable which is initialized to zero, when samples are detected in the current node this value will be iterated by +1 for each sample.

The most important things to note are the two numpy arrays which count instances of training samples to find the most common value for each of two labels. These two arrays are initialized to zero and when looking at a given feature number (featureNmbr) of a sample there is a value for that feature. That value number increments the value of the np.array with a corresponding index to the value of a feature. So if we are looking at feature 0 and it’s value is 100 we will increment the value of np.array with index 100 by +1.

```
for i in range(X_train.shape[0]):
  goodSample = True
  if path:

    for t in path:
      if not t[2] (X_train[i, t[0]], t[1]):
        goodSample = False
        break

    if not goodSample:
      continue

  if y_train[i] == treeLabels[0]:
    instancesOfFeatureLabel[X_train[i,featureNmbr]] += 1
    self.subSetVolume += 1
  elif y_train[i] == treeLabels[1]:
    instancesOfFeatureNotLabel[X_train[i,featureNmbr]] += 1
    self.subSetVolume += 1
```

This bit of code utilizes the path attribute of the method to only process the samples that are in the current node's subset. As the name suggests it is a path that represents how to get to the current node. So effectively in the for statement we are first 'cutting off' the sample that would not be present in the current node. The for loop goes through all the samples in the training dataset. First we assume that the current sample is a goodSample (sample which should be in the current node's subset). Then for each tuple in the path the sample is compared to the values of path tuple using the operator contained within the path tuple (in third position). If it does not meet the criteria, it is considered a bad sample (goodSample = False). If at least one of the tuples is not satisfied by the current sample that sample is not processed. However if the sample satisfies all the conditions from path it is considered a good sample, and another test is done in the next if and elif statements. The last two statements only select the samples which have the two correct labels that this tree is designed to deal with, and for each instance of a sample with correct labels it increments the earlier mentioned np.array filled with zeros, and counts all the instances of their values. So the result of this is an array which counted all possible values of samples and the amount of times they occur in the current node's subset. Also the total amount of samples are counted within the current subset by incrementing self.subSetVolume by +1 within the if and elif statements

```
sumLabel = np.sum(instancesOfFeatureLabel)
sumNotLabel = np.sum(instancesOfFeatureNotLabel)

if sumLabel == 0 and sumNotLabel == 0:
  return 0, (0, 1), 1.1
if sumLabel == 0:
  return -1, (1, 0), 1.1
if sumNotLabel == 0:
  return -2, (0, 1), 1.1

featureArgmax = np.argmax(instancesOfFeatureLabel)
notFeatureArgmax = np.argmax(instancesOfFeatureNotLabel)
featureThreshold = round((featureArgmax + notFeatureArgmax) / 2)
```

Then the sums of each type of label are counted, to determine if there even are sufficient amounts of samples to make a split on, and which type of label is predominant in the current node. In the if statements there is a check if there is a full split, where there is only one type of sample, or no samples at all, and which labels it has to return. The first return values are artificial values for the threshold signifying a full split, and are later caught by the tree building function, to prevent further splitting of the node if this occurs. The error rate 1.1 is also artificial so it can easily be seen in the debugging and in the methods later on, which handle tree creation and feature number selection.

It is important to note that in the if statement the test for sumLabel and sumNotLabel values are hardcoded to zero, but in further research I would like to explore what value for the number of samples is best for accuracy of decision trees, as to prevent overtraining. However after implementing this algorithm, it was very computationally heavy even on my machine equipped with quite a high end processor, which sped up the time taken to make a model for the decision trees. I could predict this value would largely depend on the total number of samples in the dataset, but I will cover this in more detail

Then the two most common values of a selected feature for the two labels are found, called featureArgmax and notFeatureArgmax, using the np.argmax function on the two np.arrays. Then from those two numbers the average is found and rounded to a whole number, as a featureThreshold. It is important to reiterate this, that for every possible computation with arrays a numpy module is used, because when handling large amounts of data samples there could be huge arrays with big numbers and heavy computations. Numpy module is a lot more efficient at computing them because it is written in c++.

```
cumSumFeature = np.cumsum(instancesOfFeatureLabel[::-1])[::-1]
cumSumNotFeature = np.cumsum(instancesOfFeatureNotLabel[::-1])[::-1]

if featureThreshold != 0:
  if featureArgmax >= notFeatureArgmax:
    indicesOfTreeLabel = 0, 1
    errorRate = 1- cumSumFeature[featureThreshold]/cumSumFeature[0]
  else:
    indicesOfTreeLabel = 1, 0
    errorRate = 1- cumSumNotFeature[featureThreshold]/cumSumNotFeature[0]
```

This part of the code handles the case when there is some threshold found that is somewhat informative. Firstly for each numpy array there is a calculation of the cumulative number of samples, which is later used to calculate the error rate at a given threshold. Then there is a test that the threshold is not equal to zero, which would be an uninformative threshold as it does not create a division. Then in the if and else statements there is a decision made which label in the current node is more predominant. If it is the first label then the total number of first label (denoted by featureArgmax) is bigger or equal to the number of instances of samples with the second label (notFeatureArgmax). The the label indices stay the same as defined earlier in the method and an error rate is calculated, by dividing the total number of samples above the selected threshold (cumSumFeature[featureThreshold]) by the total number of samples of this label (cumSumFeature[0]) and subtracting that from 1.

```
return featureThreshold, indicesOfTreeLabel, errorRate
```

This method then returns a tuple containing the featureThreshold for the given feature number in the attributes of this method, the order of labels if this threshold is selected and the error rate.

This return is handled by a method called featureSelector. It iterates through all feature numbers to select the division with the least error rate, which is the division that is most informative.

```
if curFeature[0] < 0:
  return None, curFeature[0], curFeature[1], 0

if curFeature[2] < leastErrorRate:
  leastErrorRateFeatureIndex = featNbr
  leastErrorRateFeatureThreshold = curFeature[0]
  curFeatureLabelIndices = curFeature[1]
  leastErrorRate = curFeature[2]
```

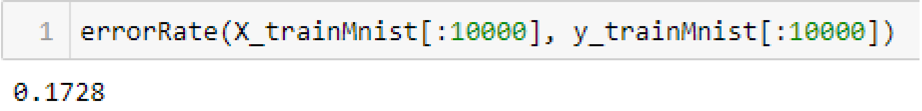
When this method iterates through different feature numbers it catches the artificial values returned by the method featureThresholdSelectorV3 representing a node which cannot be divided any further because it only has one type of label in its subset of samples and returns a tuple of artificial values which are handled when creating the nodes for the tree.

When it iterates through the features and the error rate is improved that feature number and feature threshold are noted in the method variables, and when all of the feature numbers are checked it returns a tuple containing the index of feature with the best error rate, its threshold, the order of labels, and the least error rate.

Then the decision trees are generated with the most informative splits as values of the nodes. This is done recursively. As mentioned before each tree only handles a tuple with two labels. So for the dataset with 10 possible labels for each digit from 0-9 there has to be all possible iterations of those 10 labels in a tuple there has to be 45 trees (9+8+7+6+5+4+3+2+1). After all those trees are created we can begin to classify unseen samples, by traversing each of the created trees recursively, to get a predicted label for each tree and selecting the most common prediction from the 45 trees.

Chapter 7: **Experimental Results**

This section contains the experimental results of two different machine learning algorithms, and describes the best number for K in nearest neighbours for the Mnist dataset.

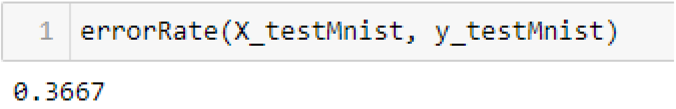


```
errorRate(X_trainMnist[:10000], y_trainMnist[:10000])
```

```
0.1728
```

Figure 13. Error rate of decision tree model on the training set

I decided to firstly run a test on the training set, to see if there was any overtraining, as the error rate in that case would be 0. I was surprised to see that there was a 17.2% error (see Figure 13) on the training dataset, which suggests that overtraining is not the problem of my implementation of the algorithm or there are some errors in the model which result in some error rate. I could assume this happens when the training sample's values are exactly on the split in the exploration stage.



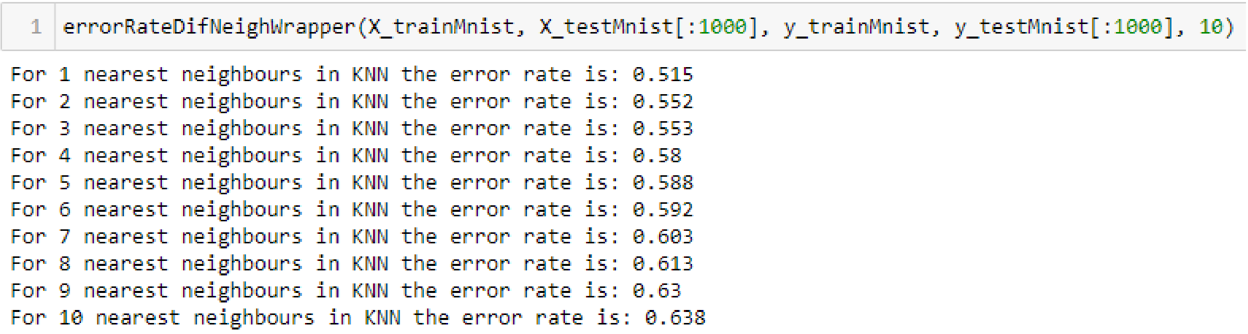
```
errorRate(X_testMnist, y_testMnist)
```

```
0.3667
```

Figure 14. Example of using plot extension

This is the result of testing the decision trees model on the test Mnist dataset (see Figure 14), which is supposed to be slightly more challenging that the training set, with numbers that are harder to classify even for the human eye (see Figure 12). The error rate of 36.7% is somewhat within the range of expected values.

As mentioned when testing the K nearest algorithms, I was also interested in finding out which number for K gives the best error rate.



```
errorRateDifNeighWrapper(X_trainMnist, X_testMnist[:1000], y_trainMnist, y_testMnist[:1000], 10)
```

```
For 1 nearest neighbours in KNN the error rate is: 0.515
For 2 nearest neighbours in KNN the error rate is: 0.552
For 3 nearest neighbours in KNN the error rate is: 0.553
For 4 nearest neighbours in KNN the error rate is: 0.58
For 5 nearest neighbours in KNN the error rate is: 0.588
For 6 nearest neighbours in KNN the error rate is: 0.592
For 7 nearest neighbours in KNN the error rate is: 0.603
For 8 nearest neighbours in KNN the error rate is: 0.613
For 9 nearest neighbours in KNN the error rate is: 0.63
For 10 nearest neighbours in KNN the error rate is: 0.638
```

Figure 15. Example of using plot extension

After running the KNN algorithm for a range of K from 1 - 10 (5th argument of function errorRateDifNeighWrapper, see Figure 15) it is clear that the best number of neighbours for Mnist dataset is 1. All further numbers of K result in a gradual decrease of accuracy (increase in error rate).

Chapter 8: **Conclusion**

Both programs we executed to the fullest of my potential. I have successfully implemented the K Nearest Neighbours algorithm and also made an extension not only writing a classifier, which is required for classifying digits, but also writing a regressor in case I would want to test this algorithm with regression problems in the future. I have also successfully implemented decision trees.

My implementation creates 45 trees when making a model for classifying digits ranging from 0-10. However due to the fact that this algorithm is very computationally demanding, I did not have enough time to make a model to for USPS dataset, and for Mnist I only used 10000 samples instead of all 50000, which cut the time at least 5 fold, but the model took more than 2 days to create. This is because the exploration stage in trees is very computationally demanding for a large dataset, because my implementation of trees can have up to 1024 nodes in each tree and there are 45 instances for each possible iteration of 2 out of 10 labels. However once the model was created it classified 10000 test samples almost instantaneously. This is due to the fact that trees are one of the

K Nearest Neighbours algorithm on the other hand does not have an exploration stage as it does not create a model. Therefore the exploitation stage gets increasingly computationally demanding the more samples there are to test. When comparing the speed of KNN to decision trees it is important to note the amount of samples there are to test. Although the decision trees model took way longer to complete that the whole classification using KNN (with k ranging from 0 - 10), it is important to note that once this model was complete the classification was almost instantaneous. In the real world if there are thousands of postcodes to scan and input into a programm the decision trees are significantly faster. Which is a win for the more complex algorithm of decision trees.

Speed of exploitation is very important, but it is more crucial to compare their precision and the error rate of the two methods on different datasets. I found that for precision the best number of neighbours in KNN was either 1 or 2 depending on the dataset, and added complexity of checking for more nearest neighbours rarely pays off. This is more visible on harder Mnist dataset, as the error rate does not improve past k = 3 whatsoever, because the earlier assumption that similar instances of the same type are in close proximity to each other is not always true for complex datasets, that have a lot of edge cases, and in real world comparing values of same pixels on different pictures yields little to no real value as opposed to looking at the picture as a whole. So in a way nearest neighbours method would only work for datasets which are really well preprocessed, to the point that digits must have the same dimension and position on the scanned image.

The best error rate I was able to achieve for KNN with K = 1 on Mnist dataset was 51.5% error, and for K = 2 error rate was 55.2%. Decision trees outperformed this massively, only utilizing 10000 samples instead of the full 50000 for training, with error rate of 36.7%. KNN in the best case gets only half of the test samples' labels right, which is still far better than just blindly guessing the label (90% error rate with 10 possible labels). Decision trees not only perform significantly faster in the exploitation but also have a much higher accuracy. This is to be expected from a more complex algorithm that takes more averages within feature values and also the predictions from each of the 45 trees, which 'smooths' out the anomalies, justifying the complexity of the algorithm with accurate results.

Chapter 9: **Professional Issues**

A major concern in machine learning is where the data is obtained from, how it is used and who is able to see it. As part of this project I did not use any private data, but I did obtain my datasets from the internet. If the USPS dataset was just a random collection of digits completely unrelated to actual postcodes and using private data was not a concern for this project as this data is publicly available. My datasets also do not reveal any identities or private information, and I was not in danger of violating any privacy and data protection laws. However in the industry of machine learning this is often neglected, the Facebook–Cambridge Analytica data scandal is a good example of that. ‘The Facebook–Cambridge Analytica data scandal concerned the obtaining of the personal data of millions of Facebook users without their consent by British consulting firm Cambridge Analytica, predominantly to be used for political advertising’ [9].

In the future I will keep this ethics in mind because machine learning and the use of private data is somewhat interconnected. It is hard to obtain big data and nowadays it is very expensive, so some companies like Facebook try to make a profit off that, violating user’s privacy and data protection for better advertising. This is a difficult issue as this is a gray area, because as a user I do not want anyone to see my private information, but on the other hand if I was to get advertisements I would rather have it related to my needs instead of it being just some random advertisement of a product that I do not need.

Chapter 10: **Self Evaluation**

I have achieved the set task of comparing machine learning algorithms. I have found the best value for K in the Nearest Neighbour algorithm and also compared that to the Decision Trees algorithm. All the algorithms were implemented in my own code, which is important in demonstrating my programming skills. I have used GitHub to demonstrate software development skills.

I have mentioned that I would use two datasets to compare machine learning algorithms, but due to time constraints I was not able to run decision trees on the second USPS dataset. In my defence Mnist dataset is a much more difficult dataset, with more challenging test samples than USPS dataset. If I had more time I would preprocess the USPS dataset and train another set of 45 decision trees on it and classify the second dataset.

This project could also be extended in a lot of different ways as mentioned earlier. It would be interesting to explore more machine learning algorithms and even implement neural networks to see how more recent advancements in data science compare to some of the older machine learning algorithms. It would also be valuable to use more datasets not only for classification, but also regression problems. Another way to tie this research to real world applications would be to produce my own dataset of handwritten digits to explore how my models would perform with real world tasks. This would also mean having my own methods for preprocessing data and normalizing it to fit for the current models.

Chapter 11: **Bibliography**

[1] https://en.wikipedia.org/wiki/Machine_learning. Wikipedia. Retrieved April 10, 2021

[2] https://en.wikipedia.org/wiki/Machine_learning#/media/File:Fig-y_Part_of_ML_as_subfield_of_AI_or_AI_as_subfield_of_ML.jpg Wikipedia. Retrieved April 10, 2021

[3] https://www.researchgate.net/figure/The-Iris-flower-dataset_tbl2_311555646 Researchgate. Retrieved April 11, 2021

[4] https://aldro61.github.io/microbiome-summer-school-2017/sections/basics/ Microbiome Summer School 2017. Retrieved April 15, 2021

[5] https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html KDnuggets. Retrieved April 15, 2021

[6] https://towardsdatascience.com/k-nearest-neighbor-85bd50ea3e4d Towards Data Science. Retrieved April 16, 2021

[7] https://www.researchgate.net/figure/An-example-of-Binary-Decision-Tree_fig1_328514085 Researchgate. Retrieved April 16, 2021

[8] Royal Holloway University of London, Machine Learning Course. Retrieved April 16, 2021

[9] https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal#:~:text=The%20Facebook%E2%80%93Cambridge%20Analytica%20data,be%20used%20for%20political%20advertising. Wikipedia. Retrieved April 20, 2021