GOLDSMITHS, UNIVERSITY OF LONDON

Computing Department

BSc Computer Science Degree

# Films and series Recommender System using Naive Bayes classifier and Logistic regression: Investigation, evaluation and comparison

by Valentin Abrutin

# Abstract

Recommender Systems are currently one of the most popular field in machine learning. They are used in massive amount of situations mostly related to e-commerce websites and services. This paper will concentrate on film and series Recommender Systems algorithms. It will cover strengths and weaknesses of basic approaches such as Naive Bayes classifier and Logistic regression, and compare them to each other with discussion and evaluation of the future of the field. Second part of the report will show implementation of these algorithms on a real world data.

**Keywords** – Recommender Systems, Machine Learning, Naïve Bayes classifier, Logistic Regression, Decision Tree classifier, Content-based filtering.

# Acknowledgements

I would like to thank my supervisor Nikolai Nikolaev for all the time he spent with me, helped with advice and supported my ideas during all project. In addition, I would like to thank my parents who supported me throughout the course of my studies and made it all possible. Thanks to all my colleagues and friends who also contributed to this project.

# Contents

# Table of Figures

# I. Introduction

## 1.1 Overview

Recommender Systems (RSs) are a subfield of information filtering systems that provide suggestions for items that are most likely of interest to a particular user.[1, p. 1] These suggestions can be involved in many aspects of human life, mostly related to e-commerce and entertainment industry. For example, what product to purchase, what music to listen, what news are you interested in or in this paper particular case – what film or series to watch.

Since RSs usually are focused on one specific subject for recommendations, they create unique set of parameters, methods and design features that, when combined with context and knowledge of that particular subject, provide the most accurate and effective suggestions. [1, p. 1]

The research of RSs is relatively new compared to other classical information system tools and techniques such as for example search engines. RSs emerged as an independent research area in the mid-1990s. [1, p. 3] However, as mentioned in abstract paragraph, nowadays interest in this field has dramatically increased as it plays enormous role on websites and web application of such tech giants as Facebook, Google, Amazon and Netflix. In addition, there are special conference, workshops and books dedicated specifically to this field of study. One of the famous example will be Association of Computing Machinery's (ACM) Conference Series on Recommender Systems (RecSys), but there are many other that are closely connected to RSs, such as Intelligent User Interfaces (IUI), Special Interest Group on Information Retrieval (SIGIR), Knowledge Discovery and Data Mining (KDD) and so on.

## 1.2 Motivation

I am genuinely interested in machine learning field and all corresponding to it areas and sub-fields. We face RSs and algorithms every day in almost any application or website. It is hard to think of the world without any RSs, they help and advise us on a daily basis. Therefore, I have passion to dig deep into that area of ML to investigate key aspects and find out what challenges and perspectives does it have. And of course I would love to try build my own movie and series recommender system because nowadays streaming services are very popular what leads to a crucial role of RSs in them. Poor recommendations lead to bad user experience and eventually to loss of clients. I want to find out how Naive Bayes classifier and Logistic regression algorithms will perform in this task.

## 1.3 Aims and objectives

To sum up previous paragraphs aim of this report is to investigate RSs research field. Then after that study, evaluate, create and compare few machine learning algorithms. As a minimum it will be Naive Bayes classifier and Logistic regression algorithms. They both will be used to create films and series recommender system. Therefore, there are the following objectives:

- Research and evaluation into the history of recommender systems
- Identification of key approaches to recommender systems
- Research on Naive Bayes classifier and Logistic regression algorithms
- Finding films and series data
- Implementation of RSs based on Naive Bayes classifier and Logistic regression algorithms using that data
- Evaluation and discussion of practical work in this project and further steps

## 1.4 Sections Overview

**Chapter II** – background research on the history and general approaches of RSs. The study of algorithms that will be used in the future for implementation.

**Chapter III** – resources overview. What data and techniques will be used, and what needs to be found and studied.

**Chapter IV** – design and methods that will be used in implementation and testing stage. Description of any changes made from first proposition.

**Chapter V** – implementation and testing description, results overview. Detailed explanation of practical work made.

**Chapter VI** – discussion based on results and evaluation of the entire project. Limitations and thoughts on future work.

**Chapter VII** – conclusion.

# II. Background Research

## 2.1 History of Recommender Systems

Recommender Systems started their journey in 1990s when they established many of the basic principles that are currently in use. In 1992, Goldberg et al. proposed Tapestry system which can be named as the first RS based on collaborative filtering approach. Later, in 1994, a news recommendation system called GroupLens was created. The main idea of the project was to automate the rule-based collaborative filtering process of the Tapestry system.[2]–[4]

In 1996 was found, Net Perceptions, the first company that focused on offering the marketing recommender engine. One of their customer was Amazon. Since then, the academic studies and e-commerce experience became a locomotive of RSs.[2]

In 1997, the GroupLens research lab launched the MovieLens project where they have trained the first RS on MovieLens dataset. GroupLens released several MovieLens datasets through 2019 which became very popular for RSs studies. Interestingly, the film industry, as well as its datasets, has become crucial in the research work of MS. MovieLens is not the only one example. The early RSs was mostly build on a nearest-neighbour approach. Field was dominated be collaborative filtering technologies and studies around that. However, Netflix Prize (2006 to 2009) created huge boost in study of machine learning algorithms for rating prediction. It made huge impact on the matrix factorization models research. And again it was a movie dataset with a task to create RS for most accurate movie rating prediction.[2], [4]

Most recent studies are related to deep neural network based RSs which has been rapidly researched since 2016. At present, we can say with confidence that the main limitations of the original matrix-completion problem abstraction have been identified and investigated. RS research is a very rapidly developing field, making it difficult to predict its future. However, it can be said with certainty that classical RSs have a space to develop further. On the other hand new methods, such as the neural science, causal inference and knowledge-enhanced can be a way of resolving current problems.[2], [4]

## 2.2 The use and importance of Recommender Systems

As mentioned previously Recommender Systems are used in a wide variety of applications, mostly related to e-commerce and media content like movies, books, video games and music. However, it is not the end for use cases, RSs also can be implemented for location-based application, for example they are used by websites for vacation travels like Airbnb or Booking.com. Another popular example will be projects for health and fitness where RSs can recommend different variations of diets or workout plans based on user data.

Such a wide range of use cases actually has solid foundation of importance of RSs. We can highlight several major benefits[1]:

**Increase the number of items sold** – obviously it is a primary object for RSs. It is so even for non-commercial ones. RS complete this object as they are most likely to meet user needs with their recommendations. Accurate evaluation of this parameter is challenging task and may vary depending on which items RS is built for, however work in that directions shows the positive impact of RSs on sales.[5], [6]

**Sales diversity** – that benefit goes very closely with previous point as it helps user to find items that are not "mainstream". For example, without RS, a streaming service for films and series shows cannot risk wasting recommendation slots on niche products that are less likely to satisfy user needs. Interestingly that benefit can also vary from different use cases and RS types, so evaluation is again depends on many parameters, but still shows positive influence overall.[7]

**Understand user needs** – one of the most important features for RS owners. RS is not only a good instrument to provide useful information to the user, it also can provide same information about user predictions to owner. This information can be used to create more popular media content or to create promotions targeted at a more appropriate group of users, for example.

**Increase the user satisfaction** – this benefit works only if RS not only good at recommendations themselves, but also provide well designed human-computer interaction. Combination of these will increase chances that user will use system more often and therefore more likely accept recommendation that may lead to increasing effectiveness of rest benefits mentioned here.

**Increase user fidelity** – this is a continuous process where user give to RS information based on his previous interactions with the system and website or application and as a result RS compute new recommendations from that. This process helps treat old users as valued customers who received more and more accurate recommendations.

## 2.3 General approaches to Recommender Systems

Today's one of the most popular method to classify RSs approaches was made by R. Burke[1, pp. 12–16], [8]. R. Burke distinguishes between six different classes of recommendation approaches:

**Content-Based** – This type of system learns to recommend items that are similar to the ones that the user liked in the past. To find out what items are similar, system is comparing them based on associated features. For example, in films and series RS, user has highly rated film made by Quentin Tarantino, therefore system will suggest other films from same director.

There are two main content-based recommendation groups of techniques. First is a classical way that implies matching the attributes of the user profile against the attributes of the items. In this case, attributes are represented as keywords that are extracted from item description.

On the other hand, another group of techniques called “semantic indexing”, represent the item and user profiles using concepts instead of keywords. They may rely on external factors such as integration of data from ontologies and encyclopaedic knowledge or internal factors, which based on hypothesis that the meaning of words depends on distributional hypothesis, which states that “Words that occur in the same contexts tend to have similar meanings”, so algorithm should analyse usage of the word in large corpora of textual documents.[9]

**Collaborative Filtering** – This was an original approach to RSs and it is still the most popular approach nowadays. It is so, because of its simplicity as it produces suggestions to user based on items that other users with similar tastes liked in the past. The similarity in the taste of two users is calculated based on the similarity in the rating history of the users. In other words collaborative filtering can be also named “people-to-people correlation”. Therefore such approach uses Neighbourhood-based methods.

However, there are more advanced techniques for collaborative filtering which address the issue of sparsity and limited coverage. Solution can be found with latent factor models, such as matrix factorization (e.g., Singular Value Decomposition, SVD).[10]

In addition, since 2010, a variety of deep learning solutions have been developed for RS. Deep learning can save time as it requires less feature engineering work and it is also can work with unstructured raw data (e.g., image, text) which are very common in RSs.

**Community-Based** – This type of systems recommends items based on the preferences of the users friends. Such systems are very close to previous, however, they rely on observation that people are more likely to listen to the recommendations of their friends rather than similar but unfamiliar persons. They take data from user’s friends ratings and based on that provide recommendations.

**Demographic** – these systems use demographic profile of the user. The logic behind such approach is that different segments of society would have different interests, and therefore they need different recommendations. For example, recommendation can be made regarding user age or marital status. However, there is not much in-depth research in this particular area of RSs.

**Knowledge-Based** – are systems that recommend items based on specific domain knowledge about how certain item features meet users’ needs and preferences and, ultimately, how the item is useful for the user.

**Hybrid Recommender Systems** – as it states in the name, such systems simply combine several techniques described above. Idea is to use advantages of one approach to cover disadvantages of another one. In that case, deep learning methods start to shine as they have great tools to easily build such systems.

## 2.4 Naive Bayes classifier

Naive Bayes algorithm is one of the most popular and simple classification machine learning algorithms. It is a supervised learning algorithm that classify the data based on Bayes theorem. Bayes theorem (Figure 1.1) is a very simple, but effective prediction tool. Basically, it calculates the probability of all outcomes and then selects the one with the highest probability. Important note, Naive Bayes classifier assumes that the effect of a particular feature in a class is **independent** of other features. For example, it means that if film will be recommended or not depending on its genre, rating, cast, popularity and length, these features will not be considered interconnected even if they are so. Due to such assumption, Naive Bayes algorithm is easy scalable and works fast on large datasets. [11], [12]

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

**Figure II-1:** Bayes' theorem [13, Eq. 1.12]

Here, P(Y|X) – is the probability of Y being true given that X is true. This is known as the posterior probability of Y. Same works with P(X|Y) where it will be posterior probability of X.[13]

P(Y) and P(X) – are simply the probability of Y and X respectively. They are not dependent on any data input as previous parts of theorem. This is known as the prior probability.[13]

Naive Bayes is actually a family of algorithms based on this theorem that can be used not only for RSs, but are very popular for text classification and real-time predictions due to its high speed nature.

**Advantages:**

- Algorithm itself is very simple, but effective
- As mentioned previously, it is very fast algorithm compared to more complex ones
- It has a low propensity to overfit, so with small dataset it can achieve better result
- It works well with high-dimensional data such as text classification, email spam detection
- Less sensitive to missing data
- Due to its simplicity, it is easy to explain the result

**Disadvantages:**

- In real-life problems it is actually almost impossible to find data that has entirely independent set of features.
- The model is unable to make predictions if it finds item of class which was not observed in the training data set. This problem called as Zero Frequency. It can be solved with smoothing techniques such as, for example, Laplace estimation.

## 2.5 Logistic regression

Logistic regression is a linear classification model. The result of prediction in that model will be a binary variable. This means that we predict variable that has only two states yes/no or as it will be in the code – 0 or 1. Logistic regression like Naive Bayes can produce prediction for multiple class problems, but in our case we look only for two classes[14], [15]. Algorithm is based on following formula:

$$p(\mathcal{C}_1|\boldsymbol{\phi}) = y(\boldsymbol{\phi}) = \sigma\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}\right)$$

**Figure II-2:** Logistic regression formula [13, Eq. 4.87]

Where $P(C_2|\phi) = 1 - P(C_1|\phi)$. So, the probability of class $C_1$ is a result of a logistic sigmoid acting on a linear function of the feature vector $\phi$.[13, p. 205]

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

**Figure II-3:** Sigmoid function formula [13, Eq. 4.59]

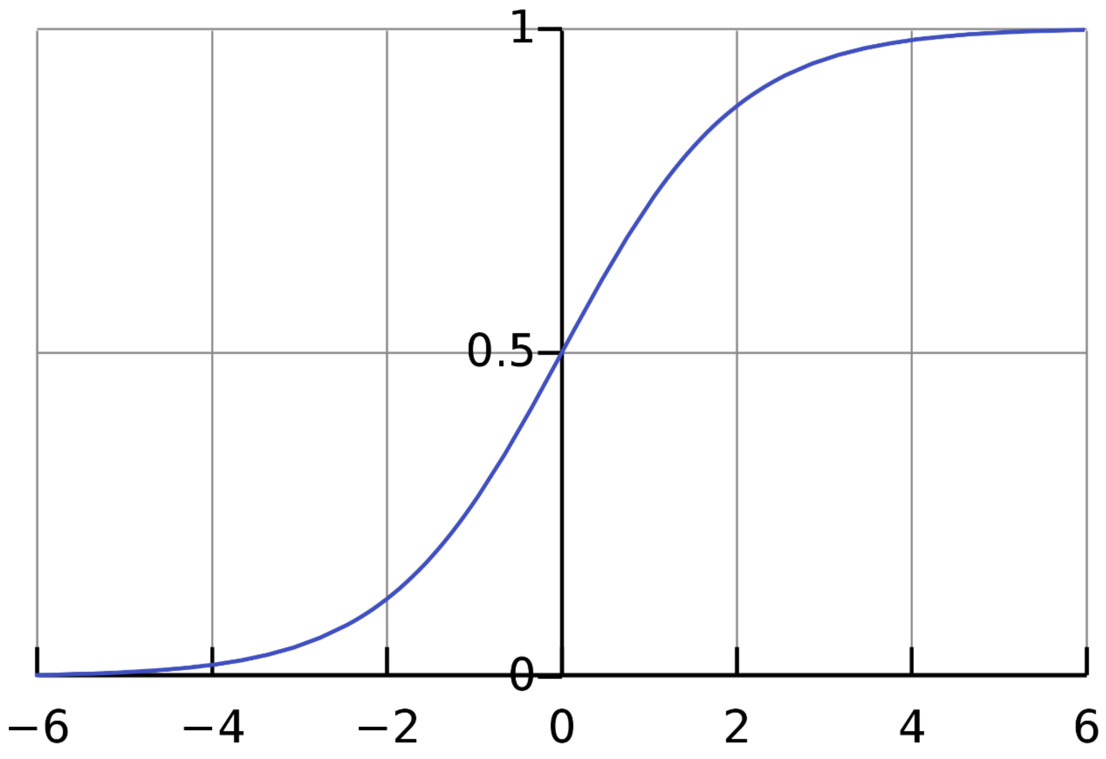


**Figure II-4:** Sigmoid function plot [16]

Above you can see sigmoid function and its plot. Sigmoid function is an activation function for logistic regression that gives us a prediction on weighted sum of input features and maps it to limited interval. As we can see, the sigmoid function always takes as maximum and minimum two values 1 and 0 (Y-axis) [14], [15].

In order to train logistic regression model we now need to determine its parameters using maximum likelihood method. This process is described in “Pattern recognition and machine learning”, C. M. Bishop[13, p. 205]. We can use the derivative of the logistic sigmoid function, which can be expressed in terms of the sigmoid function itself:

$$\frac{d\sigma}{da} = \sigma(1-\sigma).$$

**Figure II-5:** Derivative of the logistic sigmoid function [13, Eq. 4.88]

Now we can retrieve our likelihood function. For a data set $\{\phi_n, t_n\}$, where $t_n \in \{0, 1\}$ and $\phi_n = \phi(x_n)$, with n = 1,...,N the function will be:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \left\{1 - y_n\right\}^{1-t_n}$$

**Figure II-6:** Likelihood function of logistic regression [13, Eq. 4.89]

where $t = (t_1,\dots,t_N)^T$ and $y_n = p(C_1|\phi_n)$. Now we can define an error function by taking the negative logarithm of the likelihood. Error function is a function that shows us how much our prediction differs from real result, in our case it is 1 or 0.

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}$$

**Figure II-7:** Error function of logistic regression [13, Eq. 4.90]

where $y_n = \sigma(a_n)$ and $a_n = w^T\phi_n$. Taking the gradient of the error function with respect to w, we now obtain the gradient of the log likelihood. The purpose of gradient is to optimise the error function by finding the minimum of a differentiable function. Here it is:

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\phi_n$$

**Figure II-8:** Gradient descent of logistic regression [13, Eq. 4.91]

Finally, now we know how logistic regression can be trained using gradient and error functions and updating each of the weight vectors.

**Advantages:**

- Logistic regression like Naive Bayes algorithm is easy to implement and it shows good efficiency
- It can show importance of each feature based on prediction. Moreover, it shows direction of association, which can be positive of negative.
- It is also easy to interpret using previous advantage

**Disadvantages:**

- The main limitation is the assumption of linearity between the dependent variable and the independent variables which is usually not the case for real-world data.
- Logistic Regression can only be used to predict discrete functions
- The number of labels or observations should be bigger than number of features or it will lead to overfitting

# III. Resources

**Dataset:**
https://www.kaggle.com/datasets/stefanoleone992/filmtv-movies-dataset?select=filmtv_movies+-+ENG.csv [17]

**Research on both Logistic regression and Naive Bayes classifier algorithms and how to implement them in Python:** "Pattern recognition and machine learning", C. M. Bishop [13]

**Papers for Naive Bayes classifier:**
https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/ [18]

https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece [19]

**Papers for Logistic regression**:
https://developer.ibm.com/articles/implementing-logistic-regression-from-scratch-in-python/ [14]

https://dhirajkumarblog.medium.com/logistic-regression-in-python-from-scratch-5b901d72d68e [20]

**Research on how to plot decision boundary**:
https://psrivasin.medium.com/plotting-decision-boundaries-using-numpy-and-matplotlib-f5613d8acd19 [21]

https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html [22]

There were enough time to research and develop one extra algorithm – Decision Tree. More about that will be discussed in Chapters V to VII.

**Papers for Decision Tree classifier:**
https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/ [23]

https://towardsdatascience.com/decision-tree-algorithm-in-python-from-scratch-8c43f0e40173 [24]

# IV. Methods and design specification

## 4.1 Summary of project idea, aims and objectives

The general idea and aim of the project itself and its design in particular is to compare at least two algorithms, naive bayes classifier and logistic regression, in a user films and series recommendation problem. This leads to the next objectives that are left:

- Define the scope of work by capturing project requirements
- Create system design using flowcharts, describe them
- Describe methodology and criteria for algorithms comparison
- Define development methodology and methods that will be used for building, testing and evaluation of the work
- If possible, create a prototype for the system and document it
- Implement and document the entire system
- Test and analysis of the final system
- Evaluation and summary of whole project, algorithms comparison result, next possible steps

## 4.2 Requirements capture

Requirements are needed to define what is expected of the system in terms of functionality, without any specific dive into how the system will do it. In this work we will use MoSCoW analysis of requirements, system:

- Must have (M):
  - Dataset
  - Data preprocessing
  - Both, Naive bayes classifier and Logistic regression algorithms
  - Performance comparison of both algorithms
- Should have (S):
  - Visualisation of algorithms performance and their comparison (Plots and graphs)
- Could have (C):
  - More complex machine learning algorithms and their comparison
  - More datasets
- Won’t have (W):
  - User interface
  - Other approaches to recommender systems such as collaborative filtering and hybrid systems.

## 4.3 Specifications

As it follows from chosen algorithms, both variations of recommenders system will be built with machine learning, and data mining technics. Both algorithms will be written in Python language. It is purely software project, so no physical materials will be needed. Both algorithms will be developed, trained and tested on the same dataset. It will contain rating for set of films and series from different users. Both algorithms will implement content-based filtering approach due to dataset nature. Project will also use some help from few Python libraries such as – Pandas, NumPy, SkLearn, Matplotlib. More information about their role will be presented in Chapter V.

## 4.4 System design

### 4.4.1. System flowchart

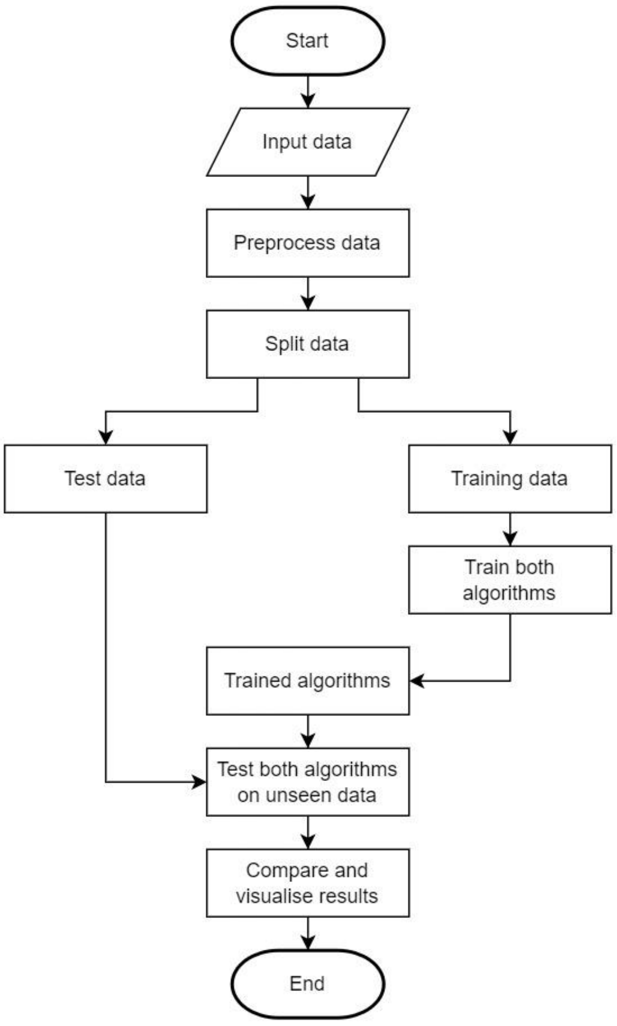


**Figure IV-1:** System flowchart

Overall system has straightforward design. First of all it takes input data from dataset, pre-process it, split in test training chunks and then proceed by Logistic regression and Naive bayes classifier algorithms. As a result, systems will show performance comparison of both algorithms.

### 4.4.2. Naive bayes classifier flowchart

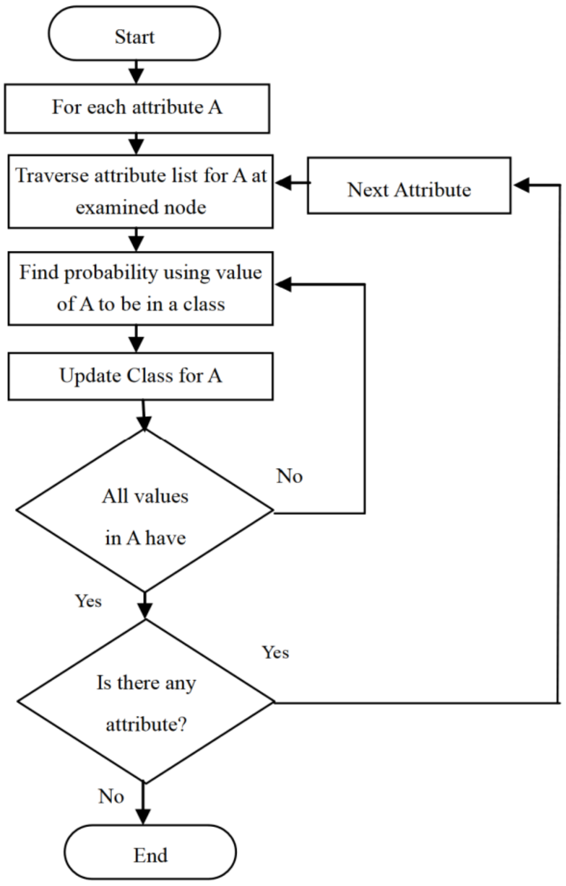


**Figure IV-2:** Naive bayes classifier flowchart [25, Fig. 2]

Naive bayes classifier algorithm simply implies Bayes' theorem (Fig. 1.1) in code. In order to use it, we have to compute all probabilities of likelihood of each attribute to be in every possible class.

In this project case it means that for each film attribute (genre, duration, cast etc.) we have to compute its probability of falling into each of two different classes that represent whether the user liked or disliked the film. This will be our numerator of Bayes' theorem.

After this we just need to substitute Bayes' theorem with real values as for denominator it will be always the same value for each class as it is simply probability of each attribute to be in whole dataset instead of each class.

When we substitute our formula with test data of film and calculate probability for it being in each class, we simply choose class with highest chance and assign film to it and therefore recommend or not this film to the user.[26]

### 4.4.3. Logistic regression flowchart

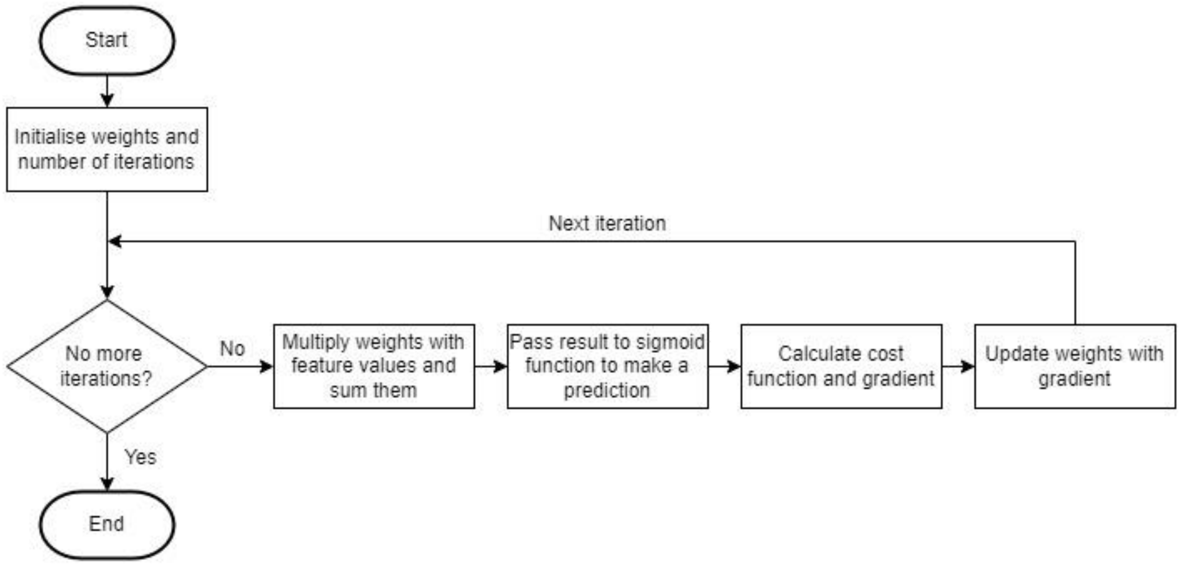

Start
Initialise weights and number of iterations
Next iteration
No more iterations?
No
Multiply weights with feature values and sum them
Pass result to sigmoid function to make a prediction
Calculate cost function and gradient
Update weights with gradient
Yes
End

**Figure IV-3:** Logistic regression flowchart [27]

As we can see on flowchart, logistic regression algorithm starts with initialisation iterations number that will define how many times algorithm will update the weights. Therefore, it means how much algorithm will try to fit to the training data, so if this parameter is too large, model will overfit. After this algorithm will initialise weight matrix with random values. Now we can start a loop. All training is made from simple steps on each iteration. First, algorithm multiply weights by feature values. Then sums them up and pass that result to link function which in our case is a sigmoid function (Fig. 2.2) that outputs one of two labels, 1 or 0, indicating whether the algorithm will recommend film to the user or not. In order to update weight, algorithm have to calculate how well it performed with current set of weights. This is made by cross-entropy error function (Fig. 2.6) and then using gradient descent (Fig. 2.7) extracted from it algorithm updates weights. Full iteration of loop is over. When all iterations are done, logistic regression algorithm is trained.[27]

Then, last set of weights will be used for test set of data, where algorithm will do one iteration of same steps without the last one with weights update. And by result of that iteration algorithm will predict final class label.

## 4.5 Development methodology

System will be made of machine learning algorithms that have a straight forward process of implementation with clear requirements that should not change much or at all. Therefore, it is decided to use Waterfall development approach as it perfectly suits to machine learning systems.

Waterfall is a classical approach with simple successive steps: Requirements, Design, Implementation, Testing and Verification and Deployment with Maintenance. This project already following waterfall methodology. We defined all requirements to our system and described design of it and its components. Following chapters will be written in the same order as steps in chosen methodology with possibility to develop some extra features or components if there is enough time as it was defined in Requirements Capture paragraph.

## 4.6 Project changes

Due to feedback and discussion with supervisor, it was decided that initial dataset has not enough data that can support efficient model training. It was so due to the nature of dataset. It had a lot of categorical and text data and many of it simply did not make any sense to include as a data feature. In addition, initial testing on the first dataset with Naïve Bayes algorithm showed very low accuracy. Therefore, it was necessary to look at other datasets.

As a result, there was found new solution with dataset that has more readable features. They still have categorical data type, however now they describe the magnitude of one aspect of the film such as humour, rhythm, effort, tension and erotism. They all have same possible values from 0 to 5 with a step of 1. These new five features significantly helped to improve accuracy score for algorithms. However, it did not solve the whole problem which will be discussed later in Chapter VI.

Old dataset:

https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset [28]

New dataset:

https://www.kaggle.com/datasets/stefanoleone992/filmtv-movies-dataset?select=filmtv_movies+-+ENG.csv [17]

With similar reason there was a change in recommender system approach. New dataset do not have any user data, so it was impossible to stick with collaborative-filtering approach. Hence, our project changed to another possible solution – content-based approach. Although that approach still needs user input, our project instead will use average rating of all users as a way to recommend film. Recommendations will be more general, but it is good base for future personalisation, which will be discussed in Chapter VI.

Last change was number of algorithms to compare. Initially there was plan to make only two algorithms and maybe third one as an addition. However, due to the simplicity of the first two chosen algorithms, it was decided to make one more, Decision Tree algorithm. This change were discussed during implementation phase, therefore Decision Tree do not have any design or background research materials due to the lack of the time.

# V. Implementation, Testing and Results

This chapter explains and describes in details all stages of implementation and testing of this project. As stated previously, whole project was written in Python programming language. It is a perfect tool for machine learning projects as it contains all necessary libraries for data manipulation and pre-processing. Therefore, this project is using following libraries: Pandas and NumPy to work with dataset with a help of dataframes and NumPy arrays, SkLearn for data normalisation and encoding techniques. It also has tool to split data in train and test chunks and PCA (Principal component analysis) tool for data dimension reduction in order to plot decision boundary. Last, but not least, this project uses Matplotlib library for various data and result plots. Project itself consists of one Jupiter Notebook file with all algorithms, including data pre-processing and one dataset file with all necessary data.

You can find source code in this GitHub repository:

https://github.com/valentin-ab/final_year_project

As mentioned previously in Chapter III, the dataset comes from Kaggle and complies with ethical data protection and GDPR requirements:

https://www.kaggle.com/datasets/stefanoleone992/filmtv-movies-dataset?select=filmtv_movies+-+ENG.csv [17]

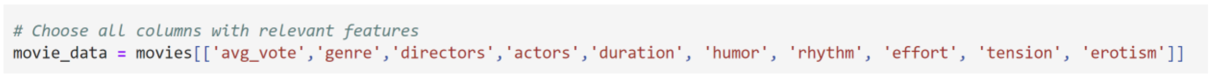
## 5.1 Data preparation and pre-processing

Data preparation and pre-processing is one of the crucial parts of machine learning workflow. This step ensures that data is correctly formatted and it has all relevant features before training and testing on algorithms. In this project data was processed through following steps:

- Check data structure and features
- Choose only relevant features
- Check for null values and handle them
- Check for duplicates values and handle them
- Handle outliers if there are any
- Handle all categorical/text data
- Normalise all numerical data
- Assign labels

Our dataset was consistent of 19 different columns that represent 19 features. It was decided to move forward only with 10 of them including one column for labels which has average vote rating from all users and critics.

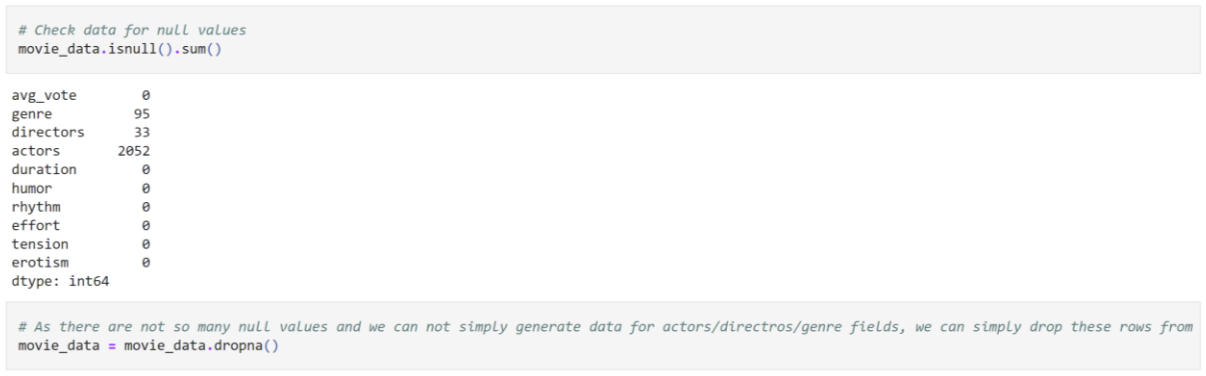
It was decided to continue with genre, directors and actors columns as it should have direct impact on the rating of film. There is also duration column as it may have impact on final rating as well due to assumption that statistically most of the films will be no longer than 1 hour and 40 minutes. Rest five columns represents a 5 point scale for one of an aspect of the film like humour, tension and so on.



```
# Choose all columns with relevant features
movie_data = movies[['avg_vote','genre','directors','actors','duration', 'humor', 'rhythm', 'effort', 'tension', 'erotism']]
```

**Figure V-1:** Screenshot of code with selected features

Next was the processing of null values. There were around 2 thousands of null values, mostly in actors column. It is not a small amount, however it was decided to drop all rows with null values as it is very hard to generate data for actors that will not affect training and testing results. The same was done for 6 duplicates that were found in the dataset.



```
# Check data for null values
movie_data.isnull().sum()
```

```
avg_vote        0
genre          95
directors      33
actors       2052
duration        0
humor           0
rhythm          0
effort          0
tension         0
erotism         0
dtype: int64
```

```
# As there are not so many null values and we can not simply generate data for actors/directros/genre fields, we can simply drop these rows from
movie_data = movie_data.dropna()
```

**Figure V-2:** Screenshot of code with null values treatment

As most of our data is either categorical or text, there should not be many outliers. However, we have had duration column with such histogram:

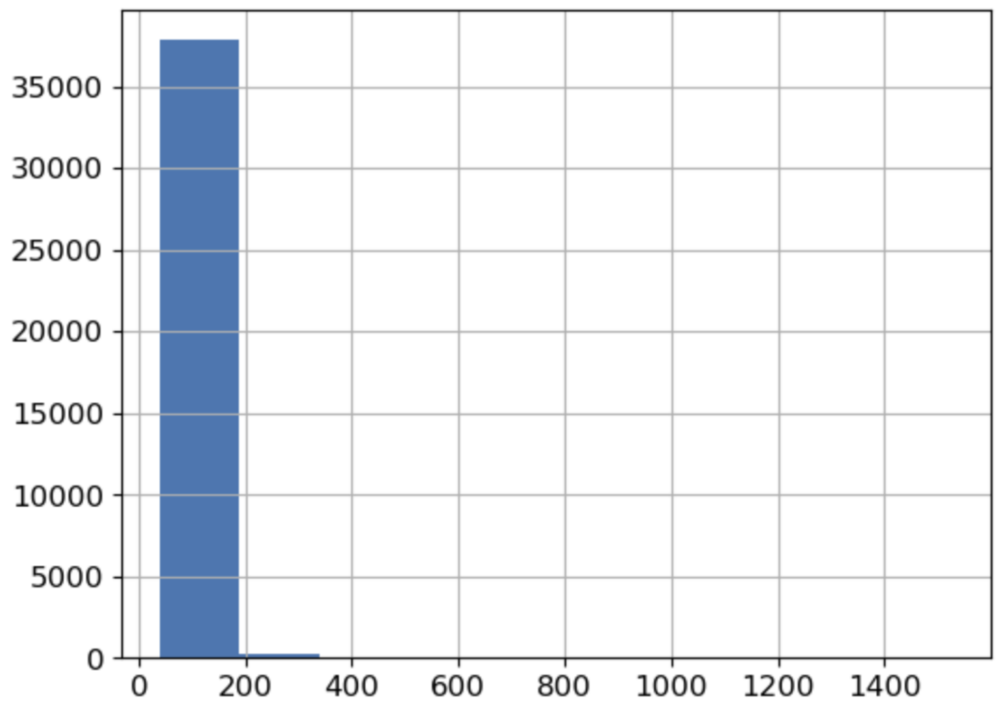


**Figure V-3:** Duration column data histogram

It clearly has too large values that affect whole data distribution. Therefore, it was decided to cap large numbers at value equal to 240, that is equal to 4 hours which should be enough for representation of long films. After this we have much better result.

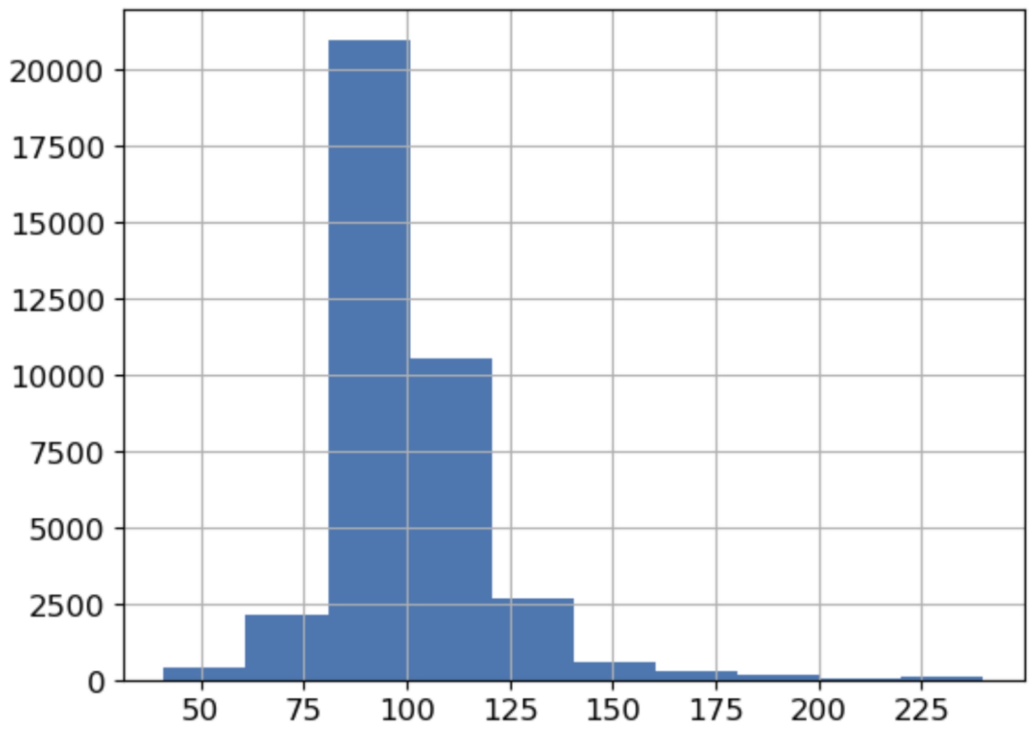
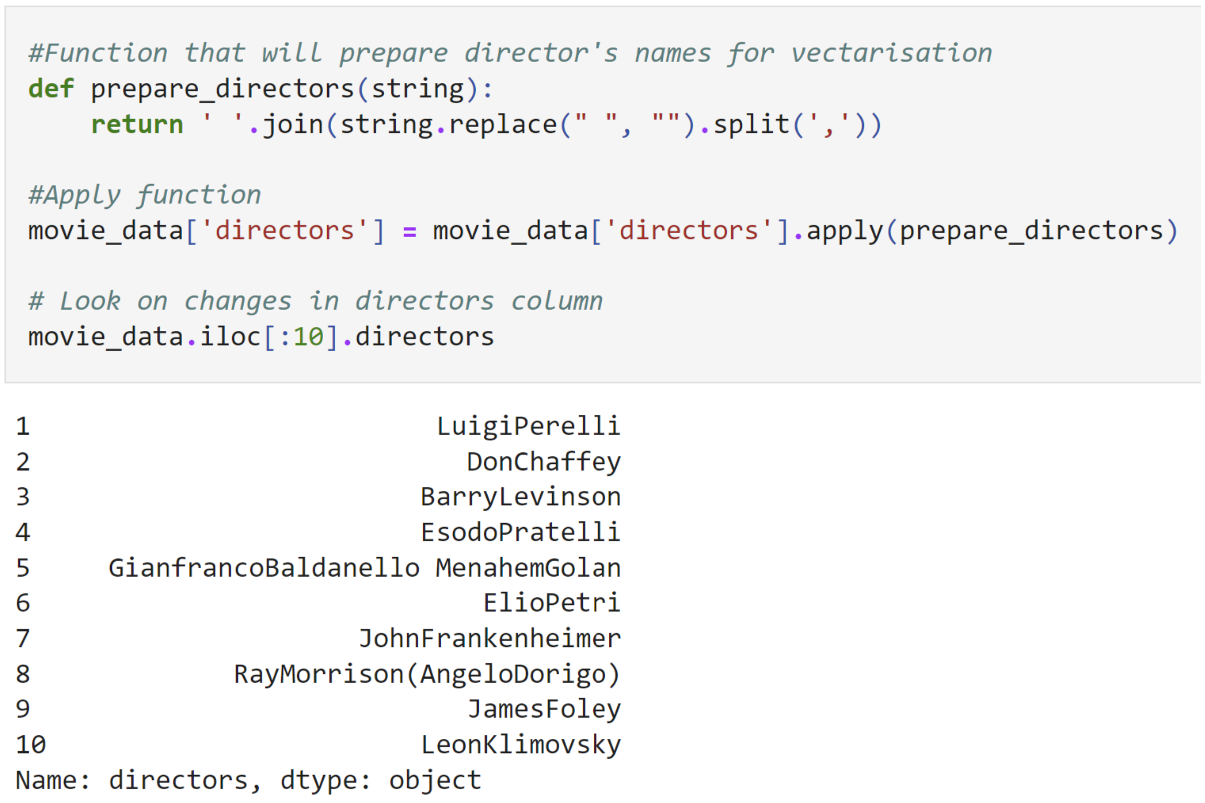


**Figure V-4:** Duration column data histogram after cap

We have actors and directors columns as text data in our dataset. It will be handled with vectorization, but before we have to unite first and last names as one “word” for vectorization purpose and in case of actors we will take only top 3 of them because of data storage limitations. Below we can see function to deal with it in directors column. For actors column it will be the same, except it will use take first 3 element of the array after splitting.

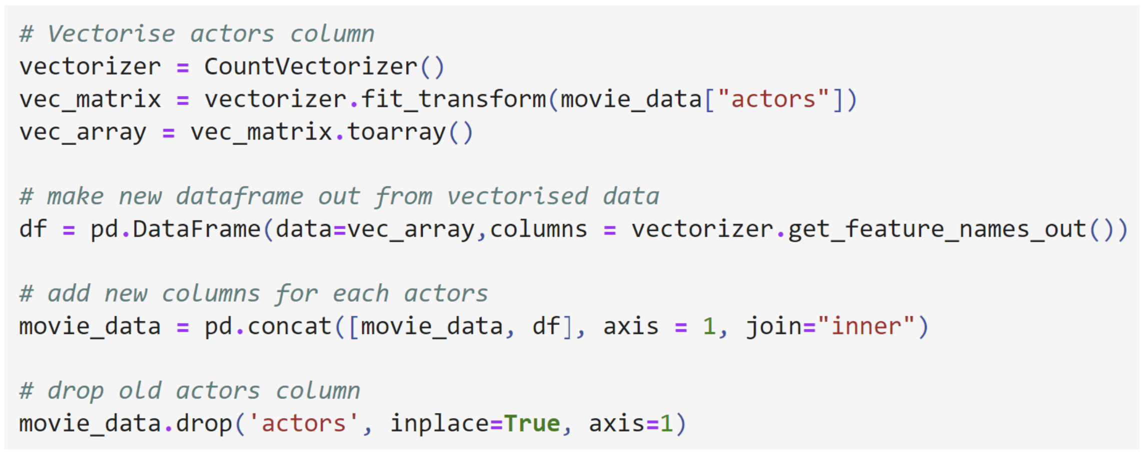


```
#Function that will prepare director's names for vectarisation
def prepare_directors(string):
    return ' '.join(string.replace(" ", "").split(','))

#Apply function
movie_data['directors'] = movie_data['directors'].apply(prepare_directors)

# Look on changes in directors column
movie_data.iloc[:10].directors
```

```
1                          LuigiPerelli
2                            DonChaffey
3                         BarryLevinson
4                         EsodoPratelli
5      GianfrancoBaldanello MenahemGolan
6                             ElioPetri
7                      JohnFrankenheimer
8              RayMorrison(AngeloDorigo)
9                            JamesFoley
10                        LeonKlimovsky
Name: directors, dtype: object
```

**Figure V-5:** Screenshot of code with director names pre-processing

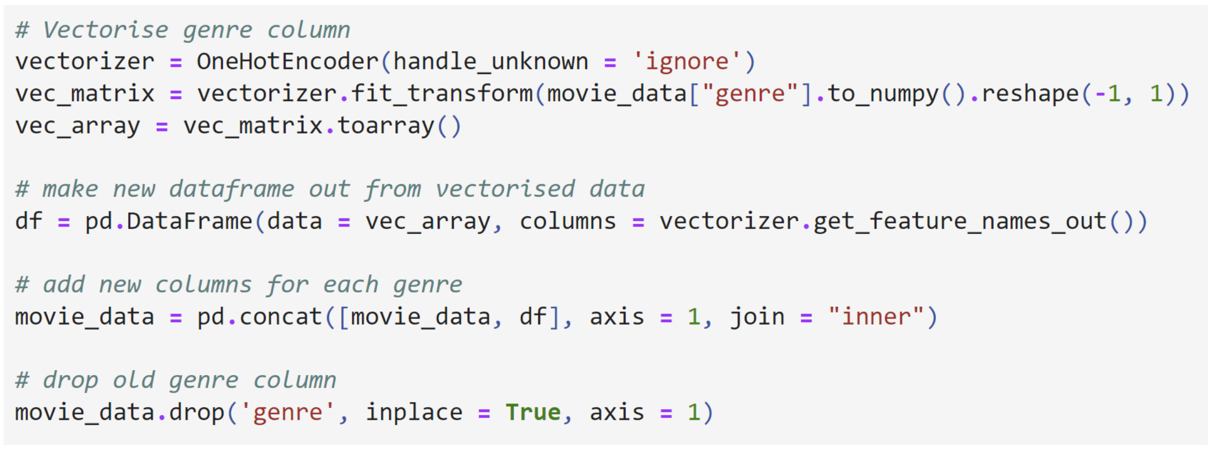
Next actors and directors columns will be vectorised with CountVectorizer from SkLearn. It will transform one column of words into multiple columns for each word and values of 1 or 0 to show if that word belongs to film or not. Below you can see how it was done for the actors column, the same was done for the directors column. Previous columns can be safely removed afterwards.



```
# Vectorise actors column
vectorizer = CountVectorizer()
vec_matrix = vectorizer.fit_transform(movie_data["actors"])
vec_array = vec_matrix.toarray()

# make new dataframe out from vectorised data
df = pd.DataFrame(data=vec_array,columns = vectorizer.get_feature_names_out())

# add new columns for each actors
movie_data = pd.concat([movie_data, df], axis = 1, join="inner")

# drop old actors column
movie_data.drop('actors', inplace=True, axis=1)
```

**Figure V-6:** Screenshot of code with actors and directors vectorisation

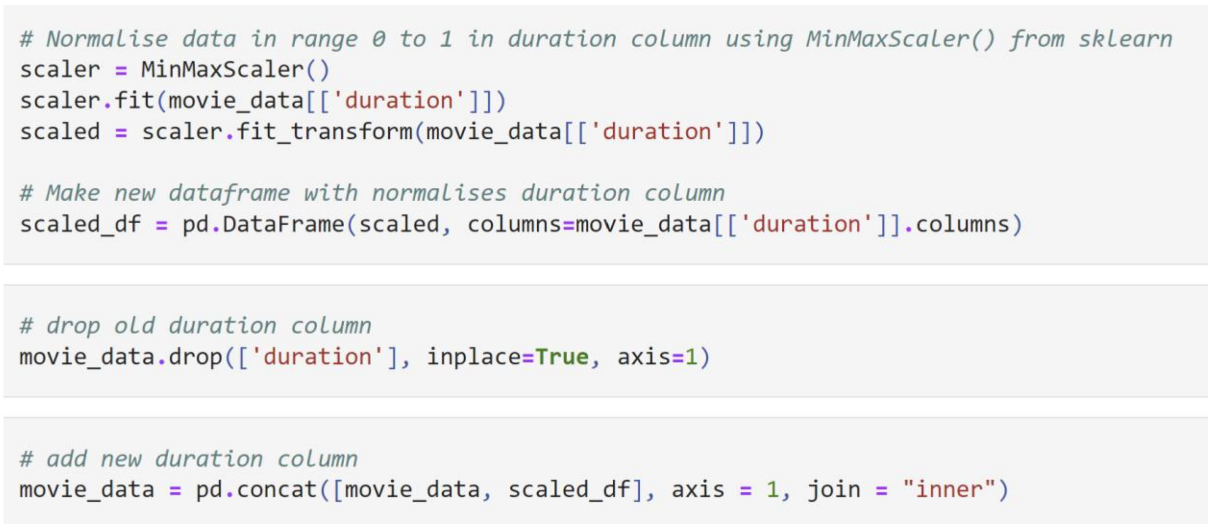
For categorical data in genre column we have OneHotEncoder. Basically it does the same job as CountVectorizer in this case, so for each genre we have separate column with 0s and 1s. Main difference that OneHotEncoder will count multiple actors or directors as one separate feature and this is why it is not used for these columns. After encoding new column, old one can be safely dropped from dataframe.



```
# Vectorise genre column
vectorizer = OneHotEncoder(handle_unknown = 'ignore')
vec_matrix = vectorizer.fit_transform(movie_data["genre"].to_numpy().reshape(-1, 1))
vec_array = vec_matrix.toarray()

# make new dataframe out from vectorised data
df = pd.DataFrame(data = vec_array, columns = vectorizer.get_feature_names_out())

# add new columns for each genre
movie_data = pd.concat([movie_data, df], axis = 1, join = "inner")

# drop old genre column
movie_data.drop('genre', inplace = True, axis = 1)
```

**Figure V-7:** Screenshot of code with actors and directors vectorisation

For numerical data, that is duration column, we need to normalise numbers on a scale from 0 to 1 with MinMaxScaler from SkLearn. It is made so because large values of duration column compared to categorical data may have to large influence on end result, particularly for Logistic Regression algorithm. Below shown how it was handled.



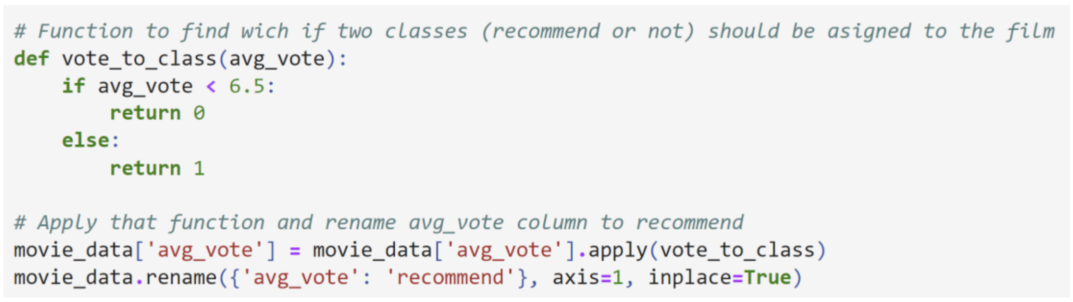
```
# Normalise data in range 0 to 1 in duration column using MinMaxScaler() from sklearn
scaler = MinMaxScaler()
scaler.fit(movie_data[['duration']])
scaled = scaler.fit_transform(movie_data[['duration']])

# Make new dataframe with normalises duration column
scaled_df = pd.DataFrame(scaled, columns=movie_data[['duration']].columns)
```

```
# drop old duration column
movie_data.drop(['duration'], inplace=True, axis=1)
```

```
# add new duration column
movie_data = pd.concat([movie_data, scaled_df], axis = 1, join = "inner")
```

**Figure V-8:** Screenshot of code with duration column normalisation

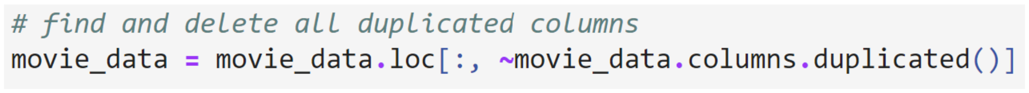
Lastly we have assigned labels based on average vote score for each film. It was decided that we will consider to recommend film if it has more that 6.5 average score. Therefore, each film with score higher than this will get label 1, meaning “recommended”, else it will have label 0, meaning “not recommended”.



```
# Function to find wich if two classes (recommend or not) should be asigned to the film
def vote_to_class(avg_vote):
    if avg_vote < 6.5:
        return 0
    else:
        return 1

# Apply that function and rename avg_vote column to recommend
movie_data['avg_vote'] = movie_data['avg_vote'].apply(vote_to_class)
movie_data.rename({'avg_vote': 'recommend'}, axis=1, inplace=True)
```

**Figure V-9:** Screenshot of code with class label assignment

Before training it was decided to doublecheck columns for duplication and delete if there are any as it might be the case after vectorisation. As a result there were around 2 thousands of duplicates.



```
# find and delete all duplicated columns
movie_data = movie_data.loc[:, ~movie_data.columns.duplicated()]
```

**Figure V-10:** Screenshot of code data duplication treatment

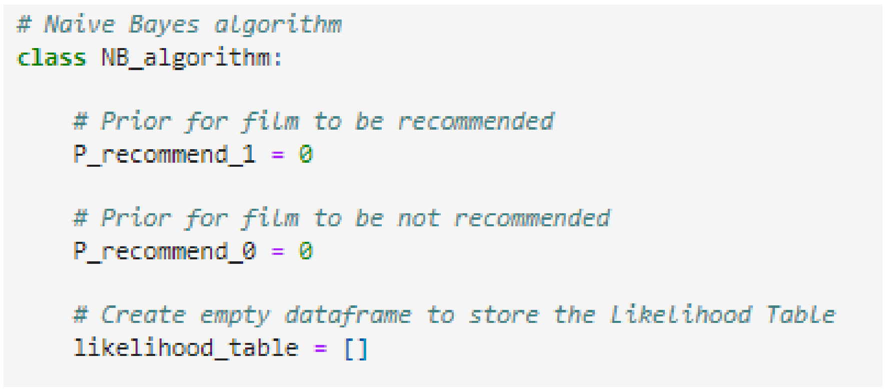
With this data preparation is completed and dataset is ready for training and testing algorithms.

## 5.2 Training models

This section of the implementation process will explain how each of the algorithms was written and trained. All algorithms were implemented as classes where new class instance initialisation is also a training process. In addition to that, all algorithm training and testing were made as a cross validation with three different dataset splits. It was 70%  of train data to 30% of test data, then 60% of train data to 40% of test data and lastly 50% to 50% of train and test data.

### 5.2.1. Train Naïve Bayes Classifier

Naïve Bayes class has three main properties. Two to store Prior probability for both labels (0 or 1) as we have binary classification problem and another one to store likelihood table. Likelihood table is the main engine of Naïve Bayes algorithm. For each feature It stores a probability for it to be in one or another class (0 or 1). Later, based on these probabilities, the algorithm will predict labels for the test dataset. However, here, in training process it first needs to be filled.



```
# Naive Bayes algorithm
class NB_algorithm:

    # Prior for film to be recommended
    P_recommend_1 = 0

    # Prior for film to be not recommended
    P_recommend_0 = 0

    # Create empty dataframe to store the Likelihood Table
    likelihood_table = []
```

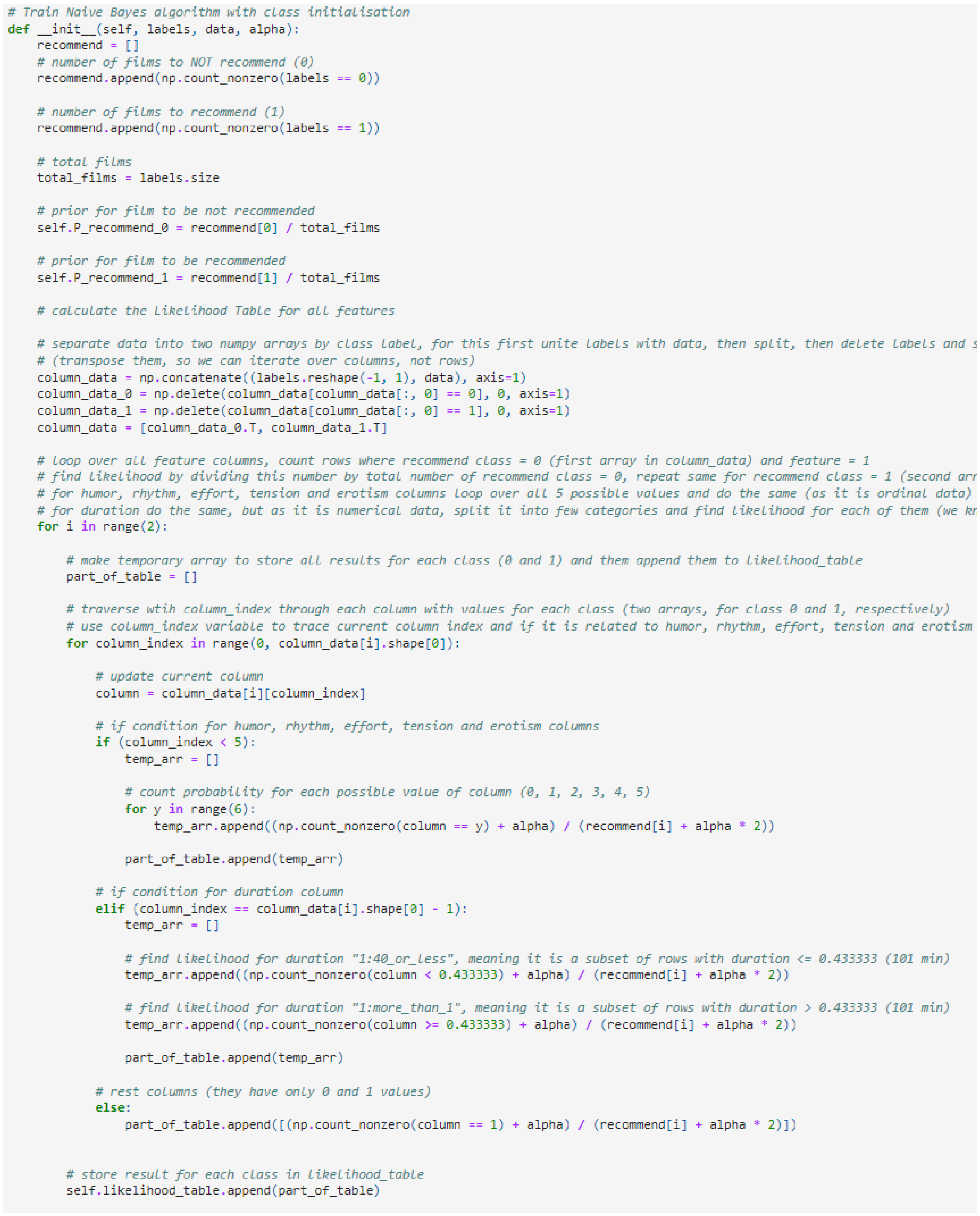
**Figure V-11:** Screenshot of Naïve Bayes class properties

When new instance of NB_algorithm is initialised, it takes training data and labels, and proceed training, in this case it is simple calculations for likelihood table, but before that, algorithm has to count few more variables.

First of all, it needs Prior Probabilities for both classes. To do so, it divides number of each class instances by total number of data points.

After this, we traverse through all columns in train dataset except labels column and count likelihood of each feature. For all ordinal columns we count how many of each possible values (from 0 to 5) belongs to each class and divide it by total number of relevant class instances. Same process is done for duration column, but here we divide all possible values into two bins, for films under 101 minutes or films that are longer. And lastly, for the rest of the columns, we have to find all values equal to 1, which means this feature belongs to this data point and it should be counted. After traversing through all columns training is complete and our likelihood table is ready for testing.

In order to find right column we simply use knowledge of what index was each unique feature like 5 columns with categorical data with values 0 to 5 are in first 5 indices and duration feature is stored in the last index of the NumPy array. Rest columns have same possible values (0 or 1), so they do not need any special conditions.

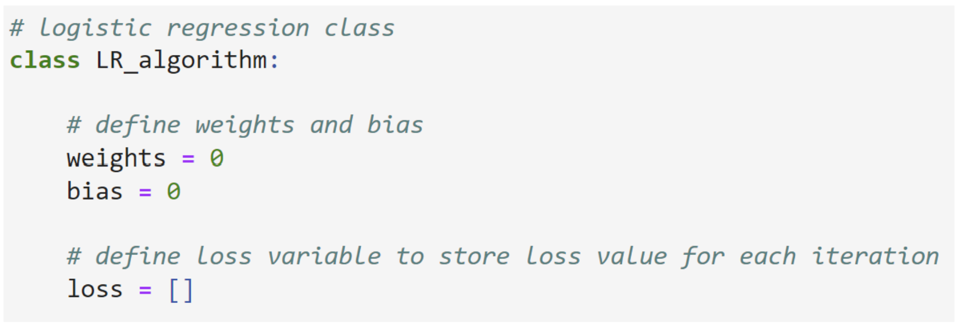


```python
# Train Naive Bayes algorithm with class initialisation
def __init__(self, labels, data, alpha):
    recommend = []
    # number of films to NOT recommend (0)
    recommend.append(np.count_nonzero(labels == 0))

    # number of films to recommend (1)
    recommend.append(np.count_nonzero(labels == 1))

    # total films
    total_films = labels.size

    # prior for film to be not recommended
    self.P_recommend_0 = recommend[0] / total_films

    # prior for film to be recommended
    self.P_recommend_1 = recommend[1] / total_films

    # calculate the Likelihood Table for all features

    # separate data into two numpy arrays by class label, for this first unite labels with data, then split, then delete labels and s
    # (transpose them, so we can iterate over columns, not rows)
    column_data = np.concatenate((labels.reshape(-1, 1), data), axis=1)
    column_data_0 = np.delete(column_data[column_data[:, 0] == 0], 0, axis=1)
    column_data_1 = np.delete(column_data[column_data[:, 0] == 1], 0, axis=1)
    column_data = [column_data_0.T, column_data_1.T]

    # loop over all feature columns, count rows where recommend class = 0 (first array in column_data) and feature = 1
    # find likelihood by dividing this number by total number of recommend class = 0, repeat same for recommend class = 1 (second arr
    # for humor, rhythm, effort, tension and erotism columns loop over all 5 possible values and do the same (as it is ordinal data)
    # for duration do the same, but as it is numerical data, split it into few categories and find likelihood for each of them (we kn
    for i in range(2):

        # make temporary array to store all results for each class (0 and 1) and them append them to likelihood_table
        part_of_table = []

        # traverse wtih column_index through each column with values for each class (two arrays, for class 0 and 1, respectively)
        # use column_index variable to trace current column index and if it is related to humor, rhythm, effort, tension and erotism
        for column_index in range(0, column_data[i].shape[0]):

            # update current column
            column = column_data[i][column_index]

            # if condition for humor, rhythm, effort, tension and erotism columns
            if (column_index < 5):
                temp_arr = []

                # count probability for each possible value of column (0, 1, 2, 3, 4, 5)
                for y in range(6):
                    temp_arr.append((np.count_nonzero(column == y) + alpha) / (recommend[i] + alpha * 2))

                part_of_table.append(temp_arr)

            # if condition for duration column
            elif (column_index == column_data[i].shape[0] - 1):
                temp_arr = []

                # find likelihood for duration "1:40_or_less", meaning it is a subset of rows with duration <= 0.433333 (101 min)
                temp_arr.append((np.count_nonzero(column < 0.433333) + alpha) / (recommend[i] + alpha * 2))

                # find likelihood for duration "1:more_than_1", meaning it is a subset of rows with duration > 0.433333 (101 min)
                temp_arr.append((np.count_nonzero(column >= 0.433333) + alpha) / (recommend[i] + alpha * 2))

                part_of_table.append(temp_arr)

            # rest columns (they have only 0 and 1 values)
            else:
                part_of_table.append([(np.count_nonzero(column == 1) + alpha) / (recommend[i] + alpha * 2)])


        # store result for each class in likelihood_table
        self.likelihood_table.append(part_of_table)
```

**Figure V-12:** Screenshot of Naïve Bayes training code

### 5.2.2. Train Logistic Regression

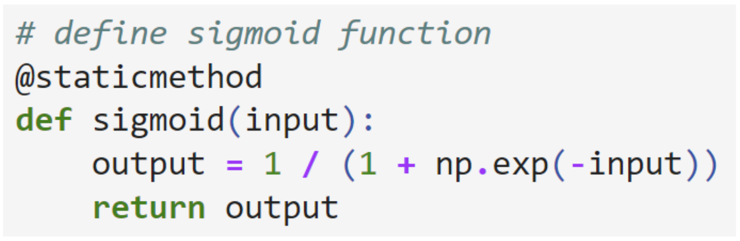
Logistic regression class has three properties as well. Weights and bias are made to store results of training, so then they can be used on the test set and loss variable stores array for loss functions values for each iteration of training.



```
# logistic regression class
class LR_algorithm:

    # define weights and bias
    weights = 0
    bias = 0

    # define loss variable to store loss value for each iteration
    loss = []
```

**Figure V-13:** Screenshot of Logistic regression class properties

Before explanation of training process we also have to look on sigmoid function which basically transforms our Linear Regression into Logistic one. Sigmoid function was already mentioned in background research, it takes output from Linear Regression and transforms it into number between 0 and 1 which represents each class.

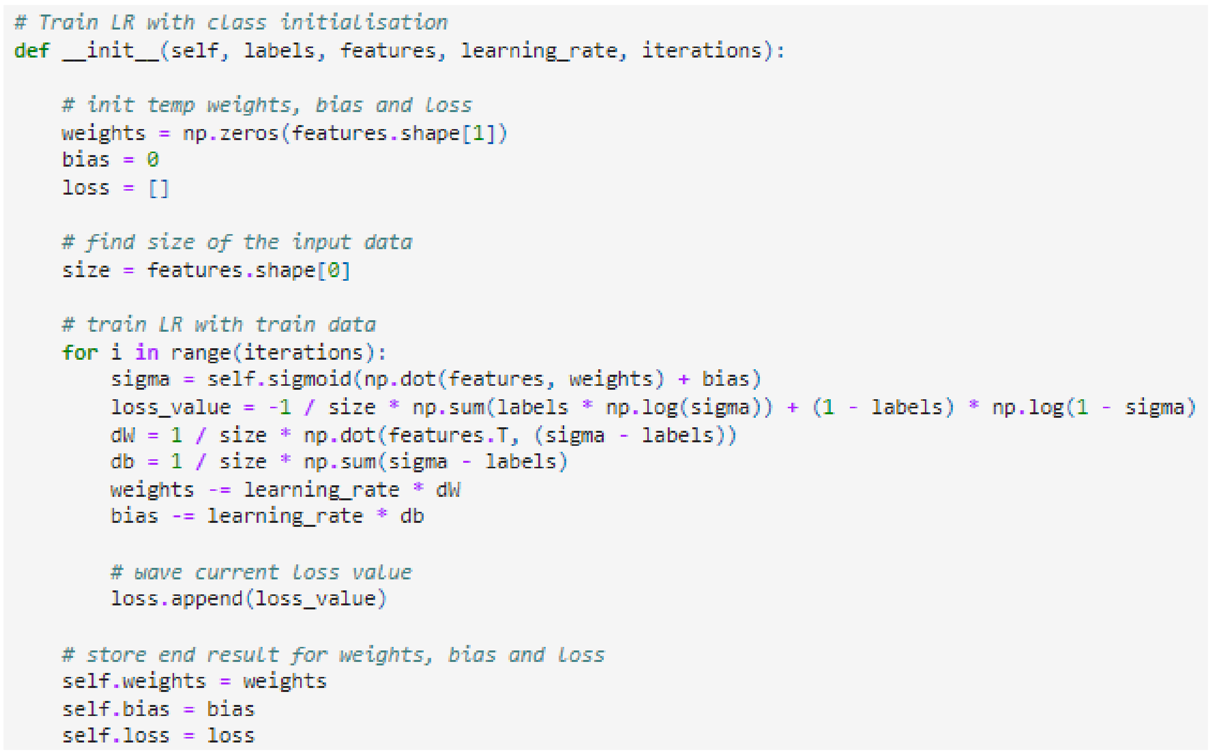


```
# define sigmoid function
@staticmethod
def sigmoid(input):
    output = 1 / (1 + np.exp(-input))
    return output
```

**Figure V-14:** Screenshot of Logistic regression class sigmoid method

Logistic Regression as well as Naïve Bayes algorithm trains during the initialisation of new class instance. It takes as input parameters data and labels as well as two hyperparameters, learning rate and number of iterations. Number of iterations controls how many times LR will try to adjust weights and bias and learning rate will define how big is the change of weights and bias in each iteration.

During initialisation, LR algorithm firstly separates class labels from the rest of the dataset and initialise starting weights and bias. Then during all iterations it simply counts how close it is to predict actual labels with current set of weights and bias. After this it updates them based on previous results and try again. During that process algorithm also collects result of loss function, so it can be plot for results section. In the end it stores final weights and bias as well as loss values array in the class properties.

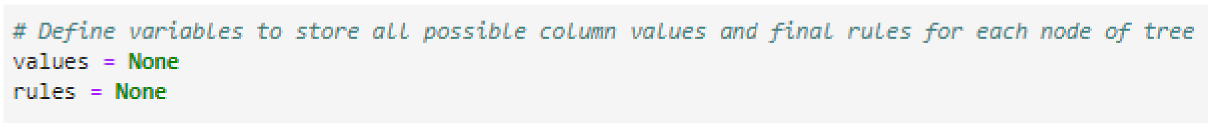


```python
# Train LR with class initialisation
def __init__(self, labels, features, learning_rate, iterations):

    # init temp weights, bias and loss
    weights = np.zeros(features.shape[1])
    bias = 0
    loss = []

    # find size of the input data
    size = features.shape[0]

    # train LR with train data
    for i in range(iterations):
        sigma = self.sigmoid(np.dot(features, weights) + bias)
        loss_value = -1 / size * np.sum(labels * np.log(sigma)) + (1 - labels) * np.log(1 - sigma)
        dW = 1 / size * np.dot(features.T, (sigma - labels))
        db = 1 / size * np.sum(sigma - labels)
        weights -= learning_rate * dW
        bias -= learning_rate * db

        # wave current loss value
        loss.append(loss_value)

    # store end result for weights, bias and loss
    self.weights = weights
    self.bias = bias
    self.loss = loss
```

**Figure V-15:** Screenshot of Logistic regression training code

### 5.2.3. Train Decision Tree

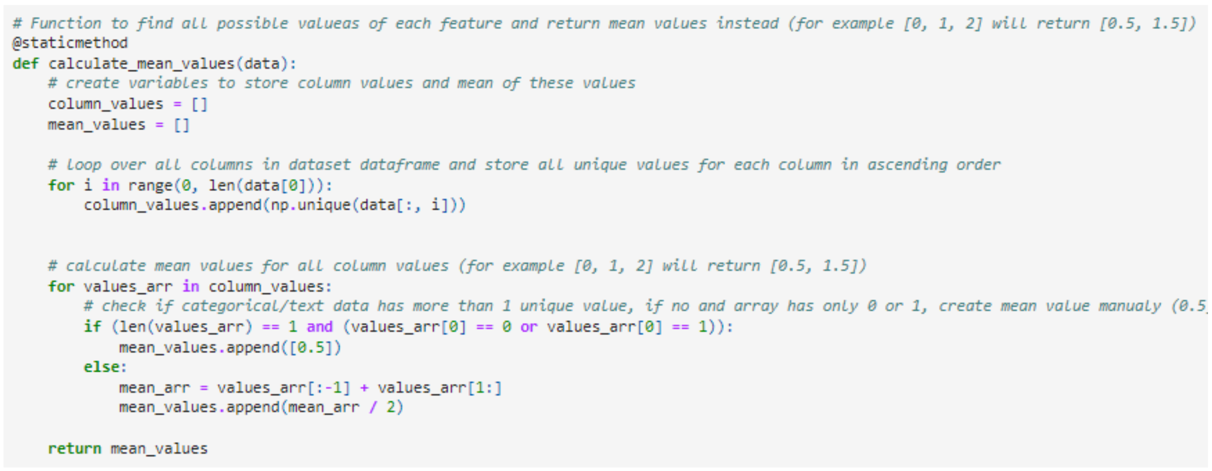
Decision tree class has two variables, “values” array stores all possible values that were found in data set columns, it is actually values in-between of each column value. It will be discussed further below. Another property is called “rules” and it will store all conditions and final values from our tree when it will be constructed.



```python
# Define variables to store all possible column values and final rules for each node of tree
values = None
rules = None
```

**Figure V-16:** Screenshot of Decision tree class properties

During initialisation our class firstly find out all column values for “values” array with a help of “calculate_mean_values” method. Firstly it will find all unique values of each column and store them inside temporary array in ascending order. Then, it will count mean value between each column value in temp array. This is done because mean values will be easier to use as a condition rule in a tree node. For example, if column has only 0 and 1 values, instead of checking two rules like this: n > 0 and n >= 1, we will simply use one rule like this: n < 0.5 . Such measure will simplify our code later. Lastly our method will return array of mean values.

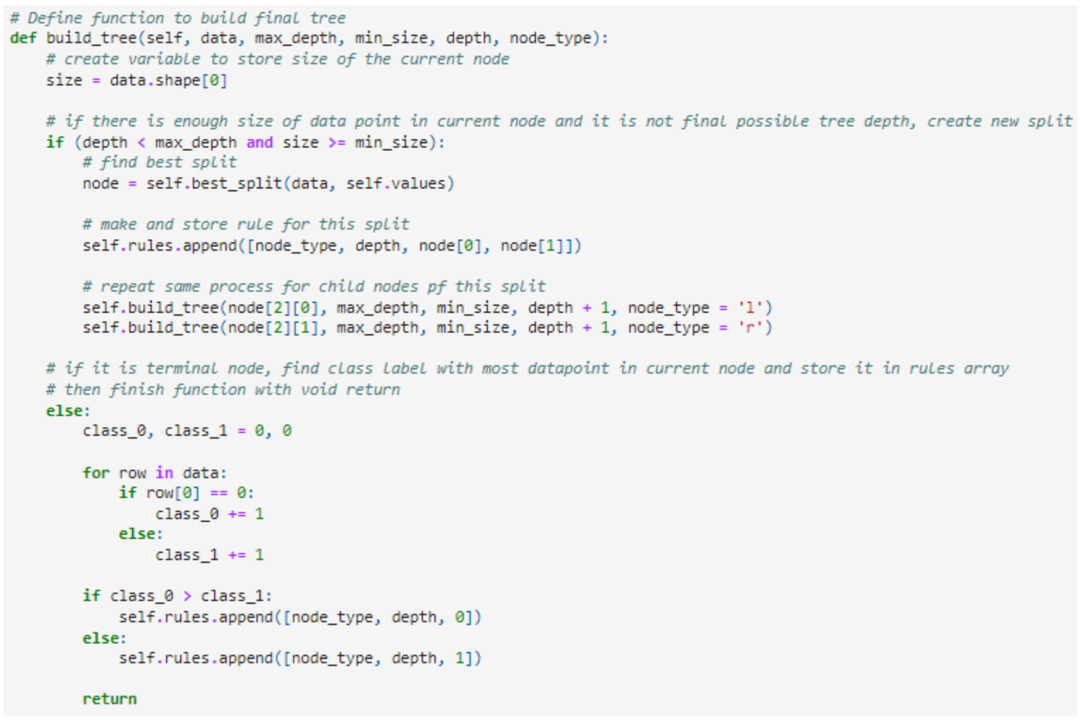


```python
# Function to find all possible valueas of each feature and return mean values instead (for example [0, 1, 2] will return [0.5, 1.5])
@staticmethod
def calculate_mean_values(data):
    # create variables to store column values and mean of these values
    column_values = []
    mean_values = []

    # loop over all columns in dataset dataframe and store all unique values for each column in ascending order
    for i in range(0, len(data[0])):
        column_values.append(np.unique(data[:, i]))


    # calculate mean values for all column values (for example [0, 1, 2] will return [0.5, 1.5])
    for values_arr in column_values:
        # check if categorical/text data has more than 1 unique value, if no and array has only 0 or 1, create mean value manualy (0.5
        if (len(values_arr) == 1 and (values_arr[0] == 0 or values_arr[0] == 1)):
            mean_values.append([0.5])
        else:
            mean_arr = values_arr[:-1] + values_arr[1:]
            mean_values.append(mean_arr / 2)

    return mean_values
```

**Figure V-17:** Screenshot of Decision tree class calculate_mean_values method

Now, we can train or in other words build our decision tree. It is also done during class initialisation, same as with other algorithms. For this puprose class has "build_tree" method. This method is a recursive function that takes data and two hyperparameters in order to build our tree. Hyperparameters create a condition till what moment we want our tree to be grown. "max_depth" is responsible for how many levels tree can have. Second parameter, called "min_size" ensures that each node has at least n of data samples, so we will not split our data into nodes  with just 1 sample which will lead to overfiting.

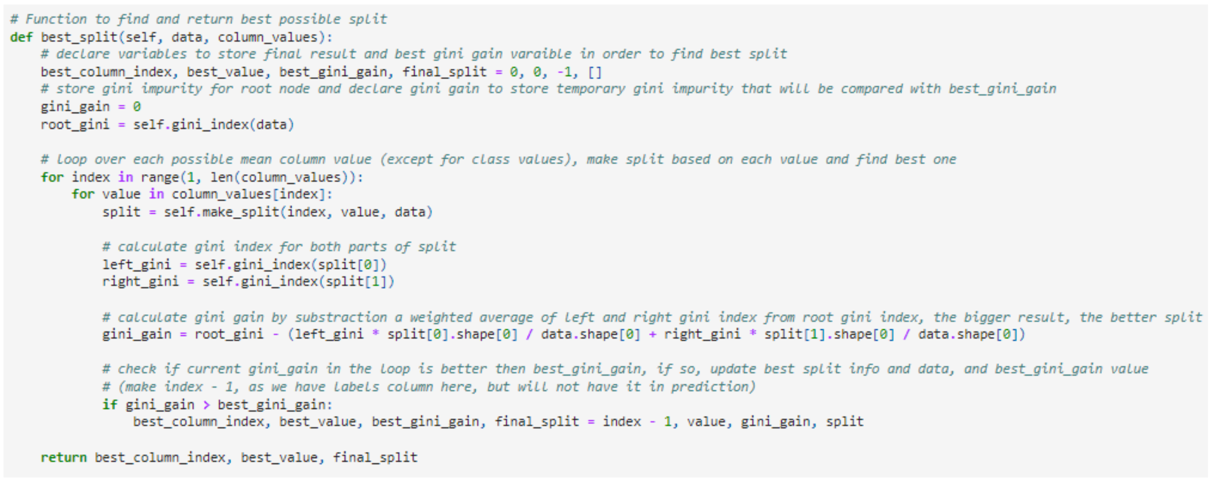
If all these parameters are ok, "build_tree" method split data in current node into two seets by best condition with a help from "gini_index" and "best_split" methods and call itself for new two nodes (left and right). It also stores condition for current node into "rules" array. This method stops when it cannot meet condition of one of the hyperparameters. In that case, it stores label of most prevalent class in current chunk of data as terminal node in "rules" array. After all recursive calls have been executed, our tree is built.



```python
# Define function to build final tree
def build_tree(self, data, max_depth, min_size, depth, node_type):
    # create variable to store size of the current node
    size = data.shape[0]

    # if there is enough size of data point in current node and it is not final possible tree depth, create new split
    if (depth < max_depth and size >= min_size):
        # find best split
        node = self.best_split(data, self.values)

        # make and store rule for this split
        self.rules.append([node_type, depth, node[0], node[1]])

        # repeat same process for child nodes pf this split
        self.build_tree(node[2][0], max_depth, min_size, depth + 1, node_type = 'l')
        self.build_tree(node[2][1], max_depth, min_size, depth + 1, node_type = 'r')

    # if it is terminal node, find class label with most datapoint in current node and store it in rules array
    # then finish function with void return
    else:
        class_0, class_1 = 0, 0

        for row in data:
            if row[0] == 0:
                class_0 += 1
            else:
                class_1 += 1

        if class_0 > class_1:
            self.rules.append([node_type, depth, 0])
        else:
            self.rules.append([node_type, depth, 1])

        return
```

**Figure V-18:** Screenshot of Decision tree class build_tree method

“best_split” method helps our train function to find best split for current chunk of data, it makes split for each possible condition with “make_split” function and calculate “Gini gain” for each of such operation. It compares every “Gini gain” and passes column index, condition values and result of split from the best Gini value to our “build_tree” method. “Gini gain” is a relative value that shows how much better Gini value of our new split nodes are compared to the parent one. Gini value – it is a value that shows the impurity of decision tree node, meaning how classes are distributed inside the node. It has value from 0 to 1. Gini equal to 0.5 shows that both classes are equally distributed (50% to 50%) and Gini equal to 0 or 1, shows that we have perfect node where one class has 100% of the node. Our purpose is to have Gini in each node as much closer to 0 as possible. This is whole idea behind the training.

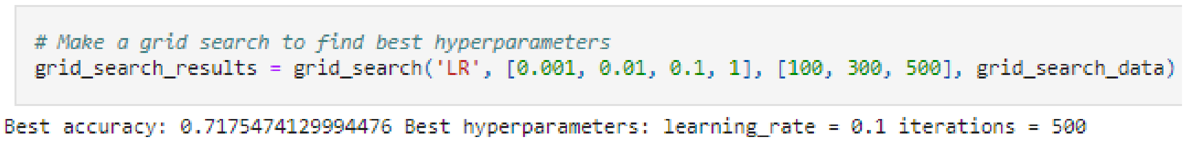


```
# Function to find and return best possible split
def best_split(self, data, column_values):
    # declare variables to store final result and best gini gain varaible in order to find best split
    best_column_index, best_value, best_gini_gain, final_split = 0, 0, -1, []
    # store gini impurity for root node and declare gini gain to store temporary gini impurity that will be compared with best_gini_gain
    gini_gain = 0
    root_gini = self.gini_index(data)

    # loop over each possible mean column value (except for class values), make split based on each value and find best one
    for index in range(1, len(column_values)):
        for value in column_values[index]:
            split = self.make_split(index, value, data)

            # calculate gini index for both parts of split
            left_gini = self.gini_index(split[0])
            right_gini = self.gini_index(split[1])

            # calculate gini gain by substraction a weighted average of left and right gini index from root gini index, the bigger result, the better split
            gini_gain = root_gini - (left_gini * split[0].shape[0] / data.shape[0] + right_gini * split[1].shape[0] / data.shape[0])

            # check if current gini_gain in the loop is better then best_gini_gain, if so, update best split info and data, and best_gini_gain value
            # (make index - 1, as we have labels column here, but will not have it in prediction)
            if gini_gain > best_gini_gain:
                best_column_index, best_value, best_gini_gain, final_split = index - 1, value, gini_gain, split

    return best_column_index, best_value, final_split
```

**Figure V-19:** Screenshot of Decision tree class best_split method

## 5.2.4. Grid search

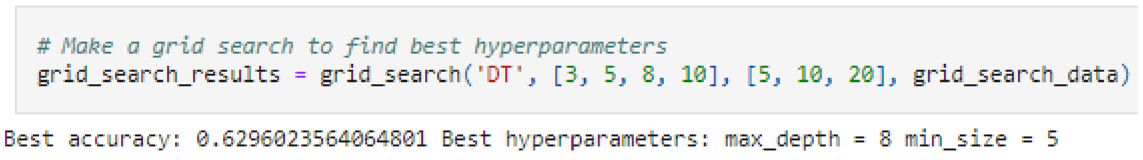
For both, Logistic Regression and Decision Tree we have performed grid search in order to find best hyperparameter. Grid search was made on a training set that was split into another training and validation subsets of data. As a result, for Logistic regression we will use learning rate equal to 0.1 and number of iterations equal to 500. For the Decision Tree we found that the best tree depth is equal to 8 and min size of node can be any from tested values, so it was decided to stick with a middle one – 10.



```
# Make a grid search to find best hyperparameters
grid_search_results = grid_search('LR', [0.001, 0.01, 0.1, 1], [100, 300, 500], grid_search_data)
```

```
Best accuracy: 0.7175474129994476 Best hyperparameters: learning_rate = 0.1 iterations = 500
```

**Figure V-20:** Screenshot of grid search results for Logistic regression



```
# Make a grid search to find best hyperparameters
grid_search_results = grid_search('DT', [3, 5, 8, 10], [5, 10, 20], grid_search_data)
```

```
Best accuracy: 0.6296023564064801 Best hyperparameters: max_depth = 8 min_size = 5
```

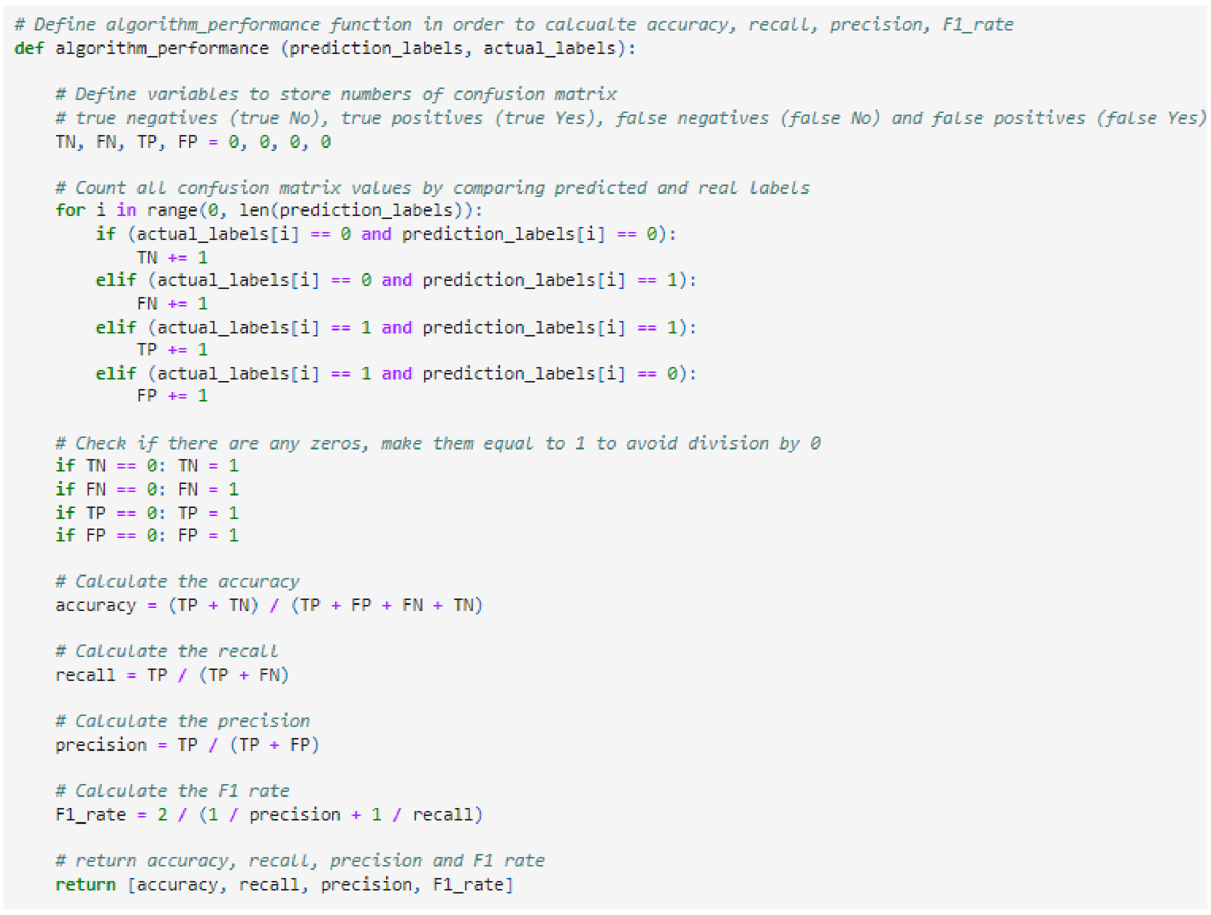
**Figure V-21:** Screenshot of grid search results for Decision tree

## 5.3 Testing and Results

Project has function for evaluation purposes. It is based on confusion matrix principle. Confusion matrix contains four different values based on predicted and actual labels.

- True negatives (TN) – datapoints that were predicted as false label (in our case 0) and that have same real class label (again 0).
- False negatives (FN) – datapoints that were predicted as false label (in our case 0), but that have different real class label (in this case 1).
- True positives (TP) – datapoints that were predicted as true label (in our case 1) and that have same real label (again 1).
- False positives (FP) – datapoints that were predicted as true label (in our case 1), but that have different real class label (in this case 0).

After this function create confusion matrix, it can calculate accuracy, recall, precision and F1 rate with following formulas in the code snippet below and output these results. This function will be used to calculate all algorithms performance.



```python
# Define algorithm_performance function in order to calcualte accuracy, recall, precision, F1_rate
def algorithm_performance (prediction_labels, actual_labels):

    # Define variables to store numbers of confusion matrix
    # true negatives (true No), true positives (true Yes), false negatives (false No) and false positives (false Yes)
    TN, FN, TP, FP = 0, 0, 0, 0

    # Count all confusion matrix values by comparing predicted and real labels
    for i in range(0, len(prediction_labels)):
        if (actual_labels[i] == 0 and prediction_labels[i] == 0):
            TN += 1
        elif (actual_labels[i] == 0 and prediction_labels[i] == 1):
            FN += 1
        elif (actual_labels[i] == 1 and prediction_labels[i] == 1):
            TP += 1
        elif (actual_labels[i] == 1 and prediction_labels[i] == 0):
            FP += 1

    # Check if there are any zeros, make them equal to 1 to avoid division by 0
    if TN == 0: TN = 1
    if FN == 0: FN = 1
    if TP == 0: TP = 1
    if FP == 0: FP = 1

    # Calculate the accuracy
    accuracy = (TP + TN) / (TP + FP + FN + TN)

    # Calculate the recall
    recall = TP / (TP + FN)

    # Calculate the precision
    precision = TP / (TP + FP)

    # Calculate the F1 rate
    F1_rate = 2 / (1 / precision + 1 / recall)

    # return accuracy, recall, precision and F1 rate
    return [accuracy, recall, precision, F1_rate]
```

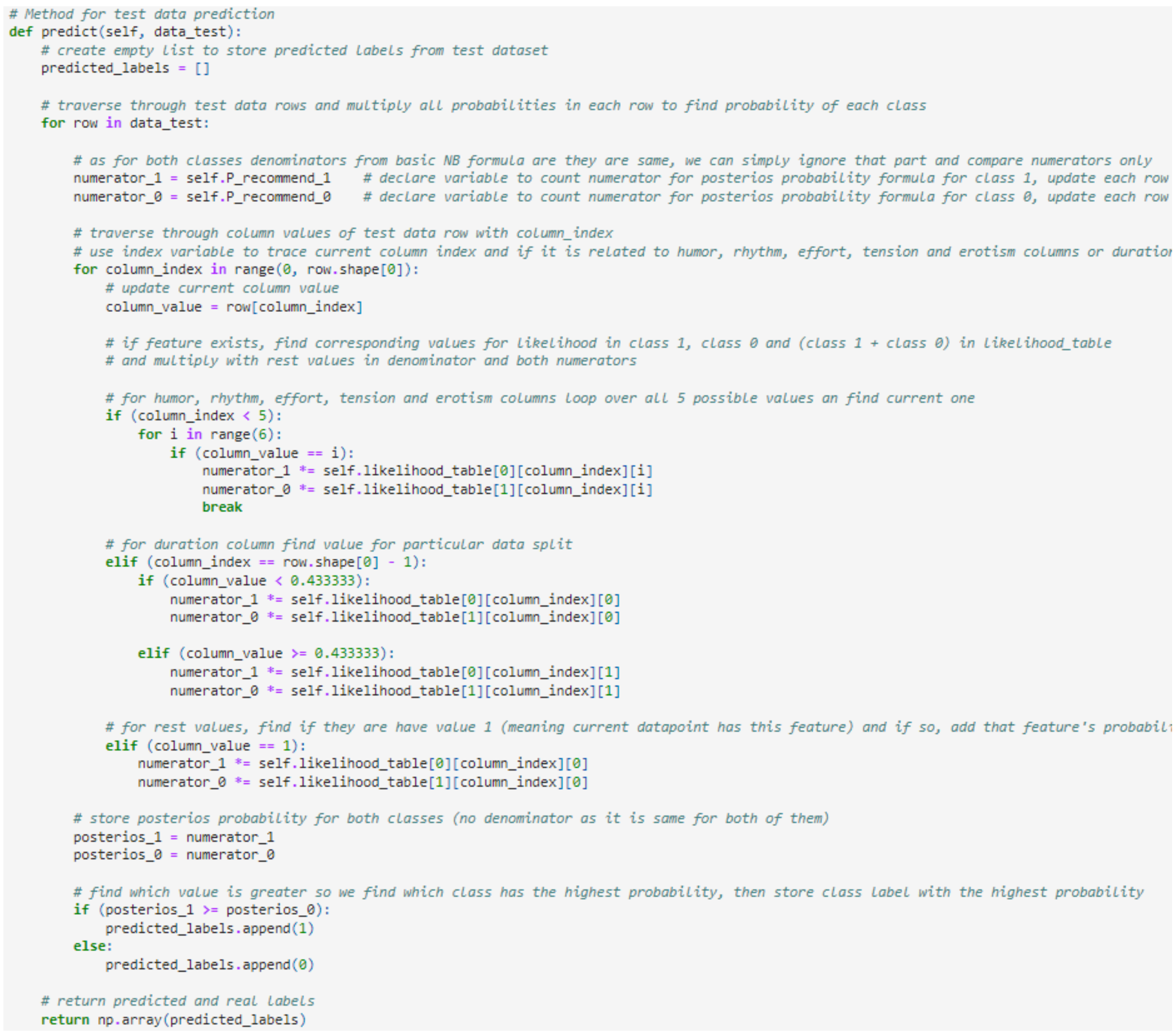
**Figure V-22:** Screenshot of algorithm_performance function code

### 5.3.1. Test Naïve Bayes Classifier

For testing purposes Naïve Bayes class has method called predict. It takes test data and labels and creates array to store predicted labels.

The predictive method then iterates over each row and then over each column of the current row. Before column iteration it cleans numerators values for both classes that will be used for prediction. In the inner loop predict method finds what kind of value each column of current row stores, then it finds possibility of that feature to be inside both classes in the likelihood table and multiplies all these possibilities for each feature together with Posterior probability of each class in a separate numerator variable.

In the end we can count probability for each class with Naïve Bayes theorem, as denominator in this formula will be the same for both classes we can simply compare our numerators and the biggest one will define our final label. Method stores that label in the final array and moves to next row until test data is finished. In the end it outputs both predicted and real labels arrays to later evaluate performance of the algorithm.

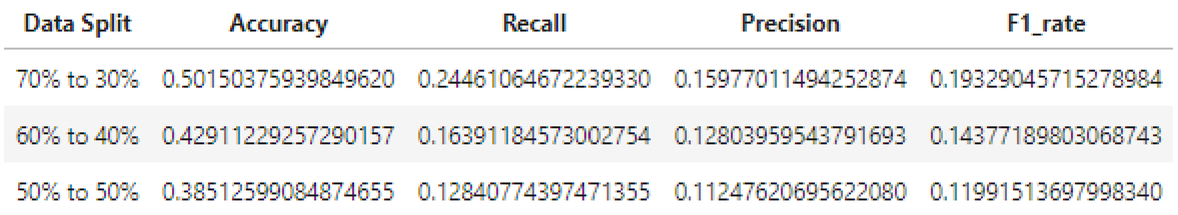


```python
# Method for test data prediction
def predict(self, data_test):
    # create empty list to store predicted labels from test dataset
    predicted_labels = []

    # traverse through test data rows and multiply all probabilities in each row to find probability of each class
    for row in data_test:

        # as for both classes denominators from basic NB formula are they are same, we can simply ignore that part and compare numerators only
        numerator_1 = self.P_recommend_1    # declare variable to count numerator for posterios probability formula for class 1, update each row
        numerator_0 = self.P_recommend_0    # declare variable to count numerator for posterios probability formula for class 0, update each row

        # traverse through column values of test data row with column_index
        # use index variable to trace current column index and if it is related to humor, rhythm, effort, tension and erotism columns or duratior
        for column_index in range(0, row.shape[0]):
            # update current column value
            column_value = row[column_index]

            # if feature exists, find corresponding values for likelihood in class 1, class 0 and (class 1 + class 0) in likelihood_table
            # and multiply with rest values in denominator and both numerators

            # for humor, rhythm, effort, tension and erotism columns loop over all 5 possible values an find current one
            if (column_index < 5):
                for i in range(6):
                    if (column_value == i):
                        numerator_1 *= self.likelihood_table[0][column_index][i]
                        numerator_0 *= self.likelihood_table[1][column_index][i]
                        break

            # for duration column find value for particular data split
            elif (column_index == row.shape[0] - 1):
                if (column_value < 0.433333):
                    numerator_1 *= self.likelihood_table[0][column_index][0]
                    numerator_0 *= self.likelihood_table[1][column_index][0]

                elif (column_value >= 0.433333):
                    numerator_1 *= self.likelihood_table[0][column_index][1]
                    numerator_0 *= self.likelihood_table[1][column_index][1]

            # for rest values, find if they are have value 1 (meaning current datapoint has this feature) and if so, add that feature's probabili
            elif (column_value == 1):
                numerator_1 *= self.likelihood_table[0][column_index][0]
                numerator_0 *= self.likelihood_table[1][column_index][0]

        # store posterios probability for both classes (no denominator as it is same for both of them)
        posterios_1 = numerator_1
        posterios_0 = numerator_0

        # find which value is greater so we find which class has the highest probability, then store class label with the highest probability
        if (posterios_1 >= posterios_0):
            predicted_labels.append(1)
        else:
            predicted_labels.append(0)

    # return predicted and real labels
    return np.array(predicted_labels)
```

**Figure V-23:** Screenshot of Naïve Bayes class predict method

### 5.3.2. Results and decision boundaries for all tests with Naïve Bayes classifier:

Below you can see table for all test results made for three different data splits as well as a decision boundary plot. In case of Naïve Bayes classifier, decision boundary plot stays exactly the same for all three test cases. It is so due to the high dimension dataset structure. It is reduced with a PCA tool and then used for creating the plot, however it is still might be not so accurate as actual results, like in our case.



| Data Split | Accuracy | Recall | Precision | F1_rate |
|---|---|---|---|---|
| 70% to 30% | 0.50150375939849620 | 0.24461064672239330 | 0.15977011494252874 | 0.19329045715278984 |
| 60% to 40% | 0.42911229257290157 | 0.16391184573002754 | 0.12803959543791693 | 0.14377189803068743 |
| 50% to 50% | 0.38512599084874655 | 0.12840774397471355 | 0.11247620695622080 | 0.11991513697998340 |

**Figure V-24:** Naive Bayes classifier results table

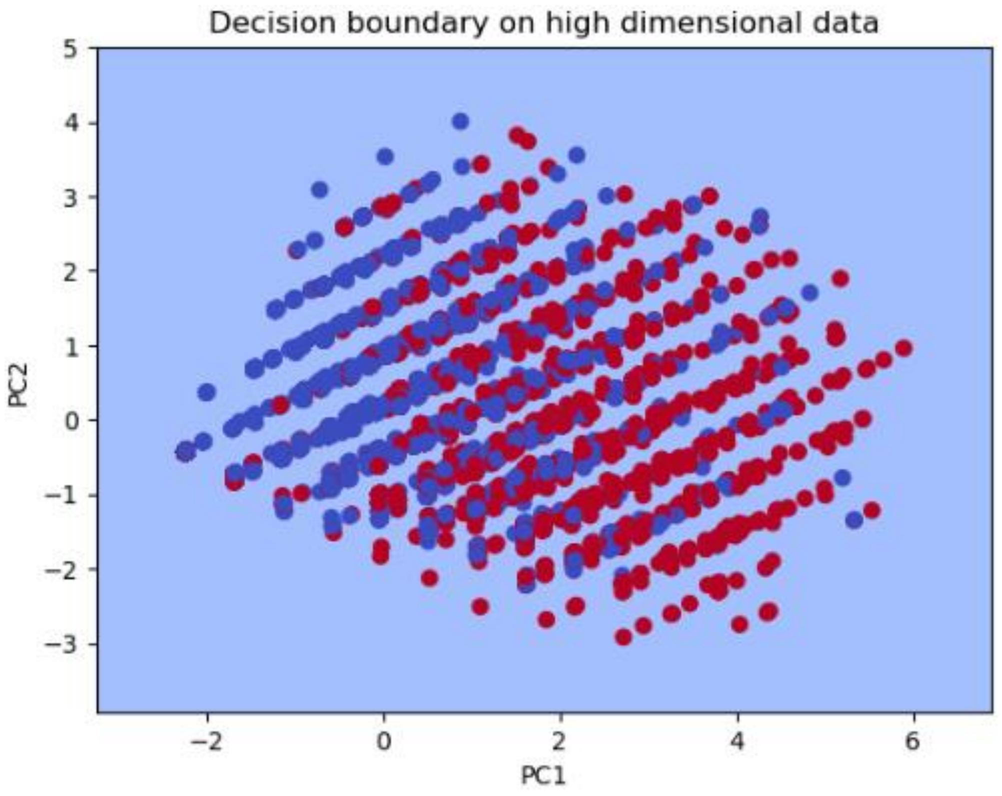
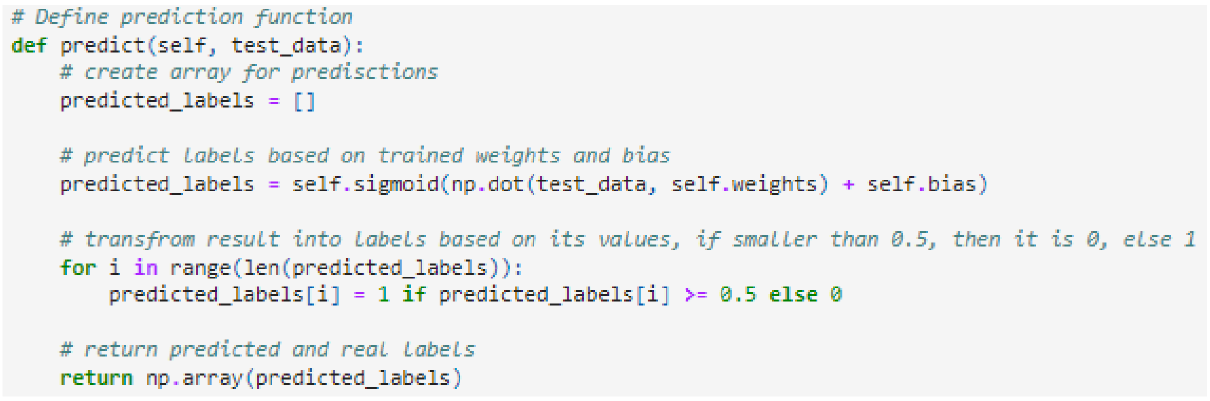


**Figure V-25:** Decision boundary for Naïve Bayes classifier in all three splits

### 5.3.3. Test Logistic Regression

Logistic Regression class has the same predict method to predict test values. It also takes test data as an input. This method repeats training process, but now it does it just once as training algorithm has already found weights and bias. It splits labels and features and creates array to store prediction results. Then it finds results of sigmoid functions with an input data of dot product of features and weight plus bias. After this, predict method transforms results into labels based on value from sigmoid function. If value is bigger or equal to 0.5, then it is a class with label 1, else it is a class with label 0. In the end it returns both predicted and real labels arrays, so algorithm performance function can count results of prediction.

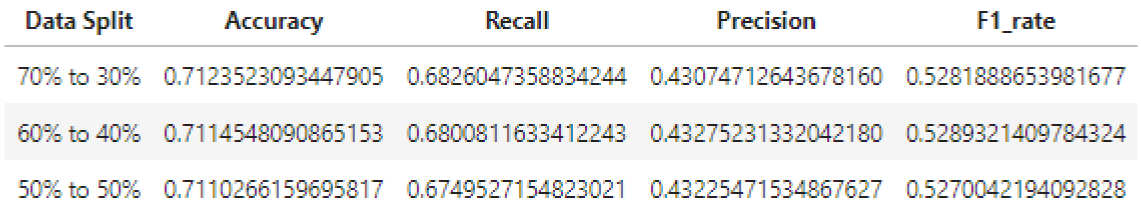


```
# Define prediction function
def predict(self, test_data):
    # create array for predisctions
    predicted_labels = []

    # predict labels based on trained weights and bias
    predicted_labels = self.sigmoid(np.dot(test_data, self.weights) + self.bias)

    # transfrom result into labels based on its values, if smaller than 0.5, then it is 0, else 1
    for i in range(len(predicted_labels)):
        predicted_labels[i] = 1 if predicted_labels[i] >= 0.5 else 0

    # return predicted and real labels
    return np.array(predicted_labels)
```

**Figure V-26:** Screenshot of Logistic regression class predict method

### 5.3.4. Results and decision boundaries for all tests with Logistic regression:

Below we can see all results for Logistic regression tests, including decision boundary plot for each case. PCA data dimension tool shows better and more accurate plot result with this algorithm, therefore we can see more objective picture.



| Data Split | Accuracy | Recall | Precision | F1_rate |
|---|---|---|---|---|
| 70% to 30% | 0.7123523093447905 | 0.6826047358834244 | 0.43074712643678160 | 0.5281888653981677 |
| 60% to 40% | 0.7114548090865153 | 0.6800811633412243 | 0.43275231332042180 | 0.5289321409784324 |
| 50% to 50% | 0.7110266159695817 | 0.6749527154823021 | 0.43225471534867627 | 0.5270042194092828 |

**Figure V-27:** Logistic regression results table

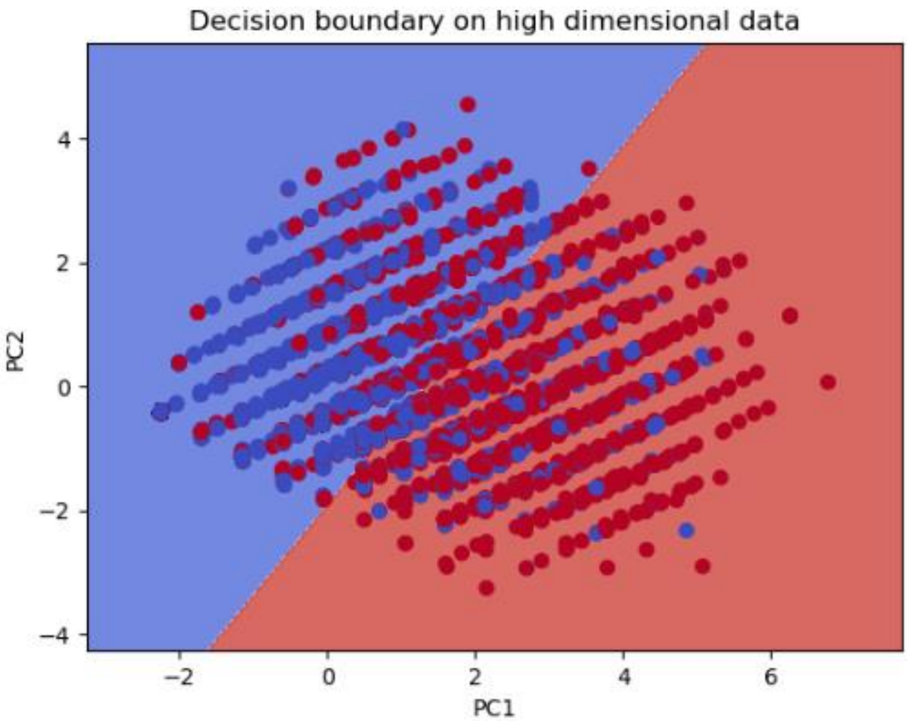


**Figure V-28:** Logistic regression decision boundary for 70% to 30% split

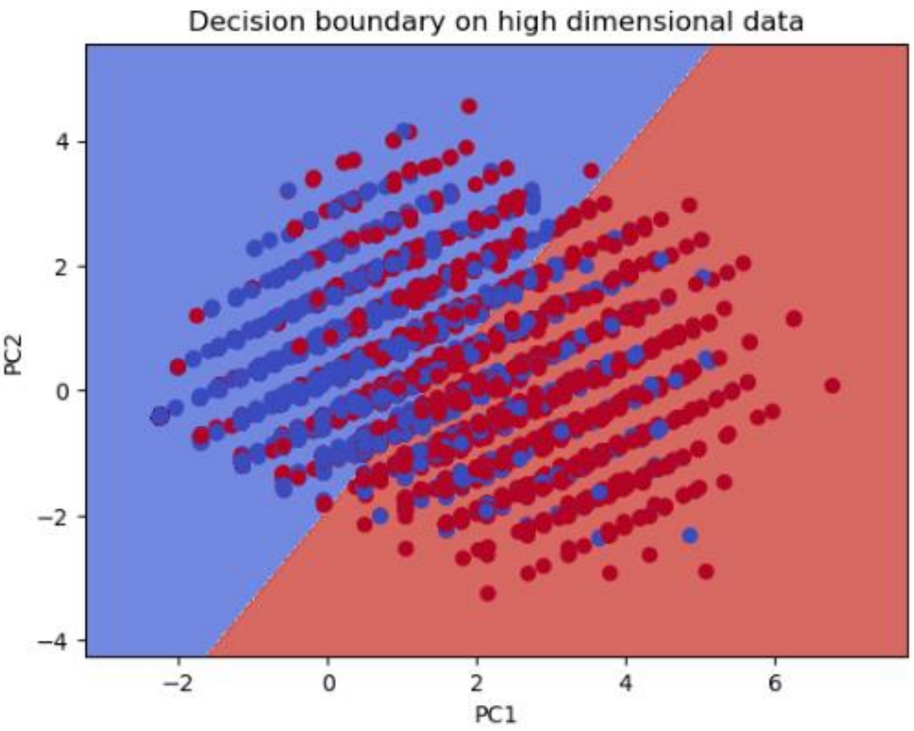


**Figure V-29:** Logistic regression decision boundary for 60% to 40% split

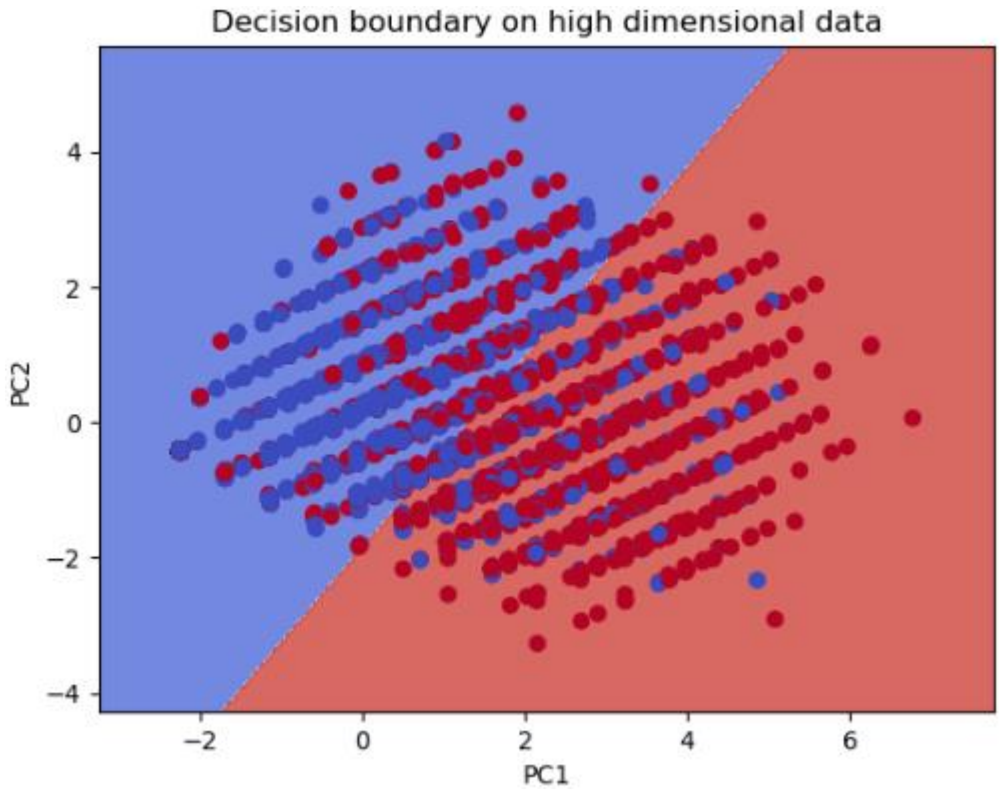
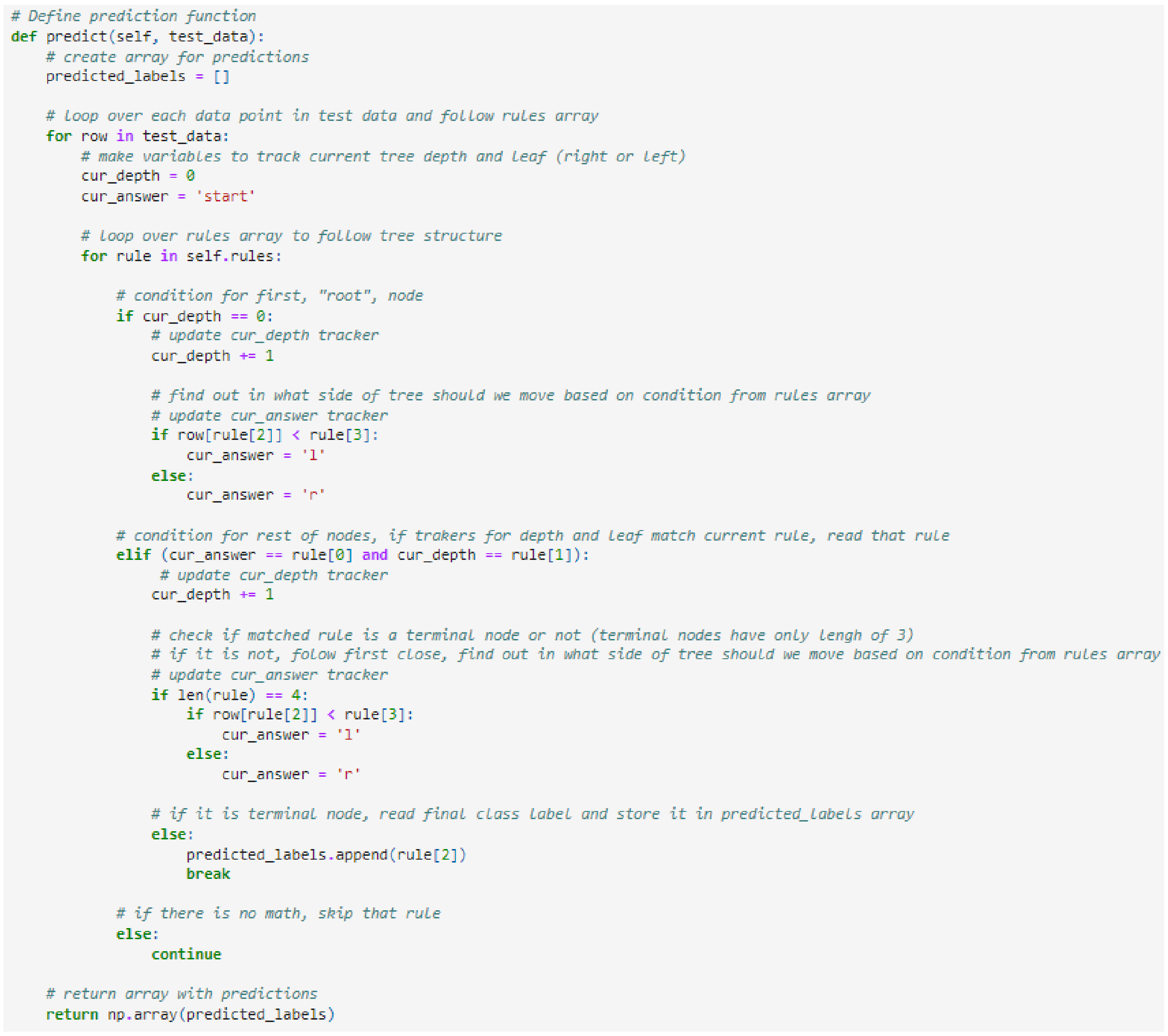


**Figure V-30:** Logistic regression decision boundary for 50% to 50% split

## 5.3.5. Test Decision Tree

In order to test Decision Tree we pass our test data to “predict” method.  That method simply iterate each data point through “rules” array, find appropriate rule for each depth of tree, correspondingly updates trackers of current depth and node type (left or right) in order to find next rule. Then it finally finds terminal node with class label, it stores that label in “predicted_labels” array and moves on to the next data point. After all iterations it returns array with predictions.

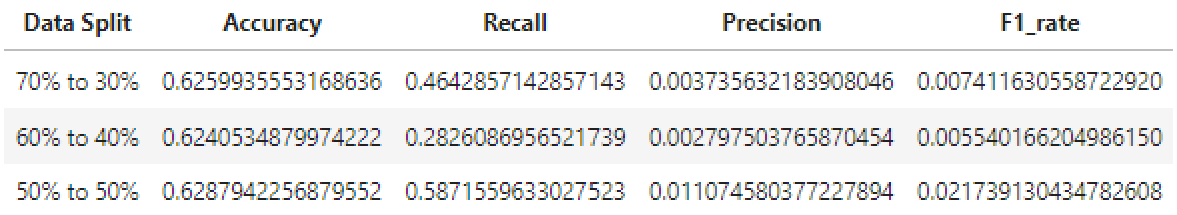


```python
# Define prediction function
def predict(self, test_data):
    # create array for predictions
    predicted_labels = []

    # loop over each data point in test data and follow rules array
    for row in test_data:
        # make variables to track current tree depth and leaf (right or left)
        cur_depth = 0
        cur_answer = 'start'

        # loop over rules array to follow tree structure
        for rule in self.rules:

            # condition for first, "root", node
            if cur_depth == 0:
                # update cur_depth tracker
                cur_depth += 1

                # find out in what side of tree should we move based on condition from rules array
                # update cur_answer tracker
                if row[rule[2]] < rule[3]:
                    cur_answer = 'l'
                else:
                    cur_answer = 'r'

            # condition for rest of nodes, if trakers for depth and leaf match current rule, read that rule
            elif (cur_answer == rule[0] and cur_depth == rule[1]):
                 # update cur_depth tracker
                cur_depth += 1

                # check if matched rule is a terminal node or not (terminal nodes have only lengh of 3)
                # if it is not, folow first close, find out in what side of tree should we move based on condition from rules array
                # update cur_answer tracker
                if len(rule) == 4:
                    if row[rule[2]] < rule[3]:
                        cur_answer = 'l'
                    else:
                        cur_answer = 'r'

                # if it is terminal node, read final class label and store it in predicted_labels array
                else:
                    predicted_labels.append(rule[2])
                    break

            # if there is no math, skip that rule
            else:
                continue

    # return array with predictions
    return np.array(predicted_labels)
```

**Figure V-31:** Screenshot of Decision tree class predict method

### 5.3.6. Results and decision boundaries for all tests with Decision tree classifier:

Here we can see Decision tree results for each test data splits. It is very hard to produce decision boundaries for Decision tree algorithm for high dimensional data and it is often fails to show the whole picture properly. However, the plots below show the general idea of how this algorithm performs on our dataset.



| Data Split | Accuracy | Recall | Precision | F1_rate |
|---|---|---|---|---|
| 70% to 30% | 0.6259935553168636 | 0.4642857142857143 | 0.003735632183908046 | 0.007411630558722920 |
| 60% to 40% | 0.6240534879974222 | 0.2826086956521739 | 0.002797503765870454 | 0.005540166204986150 |
| 50% to 50% | 0.6287942256879552 | 0.5871559633027523 | 0.011074580377227894 | 0.021739130434782608 |

**Figure V-32:** Decision tree results table

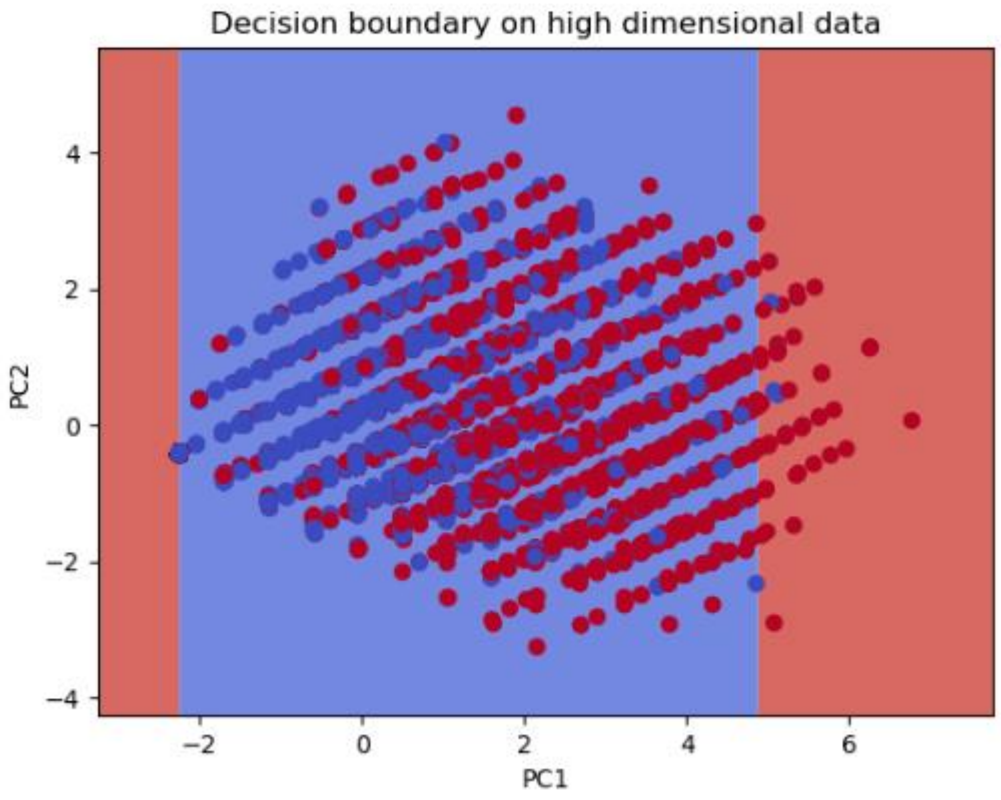


**Figure V-33:** Decision tree decision boundary for 70% to 30% split

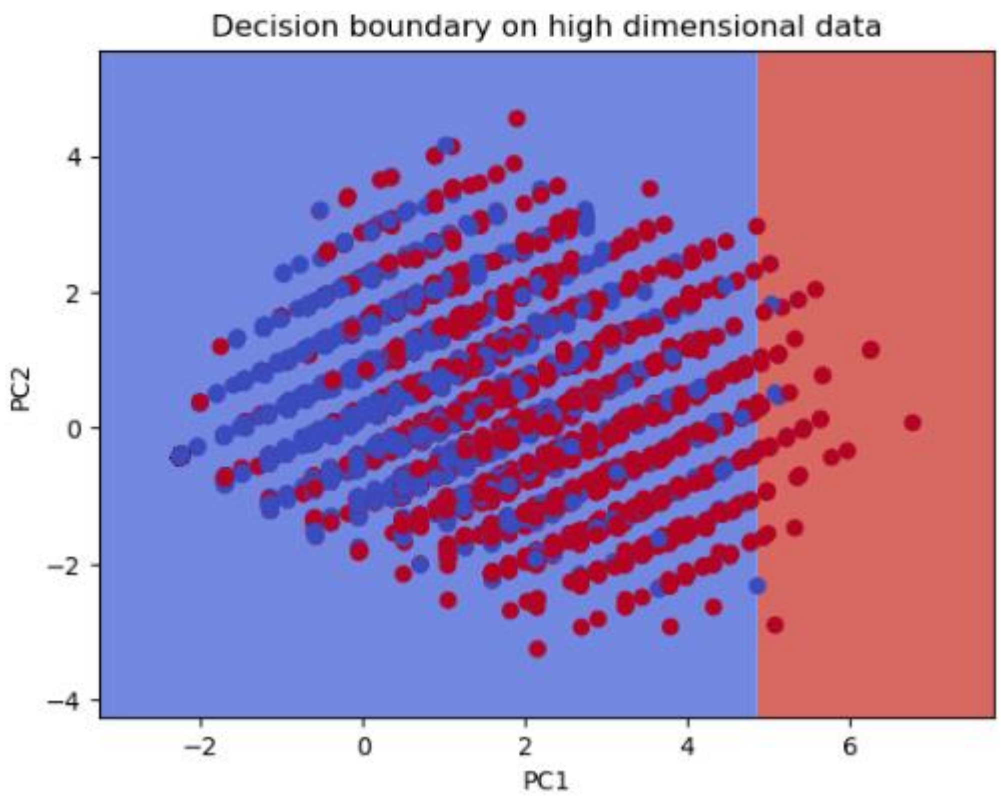


**Figure V-34:** Decision tree decision boundary for 60% to 40% split

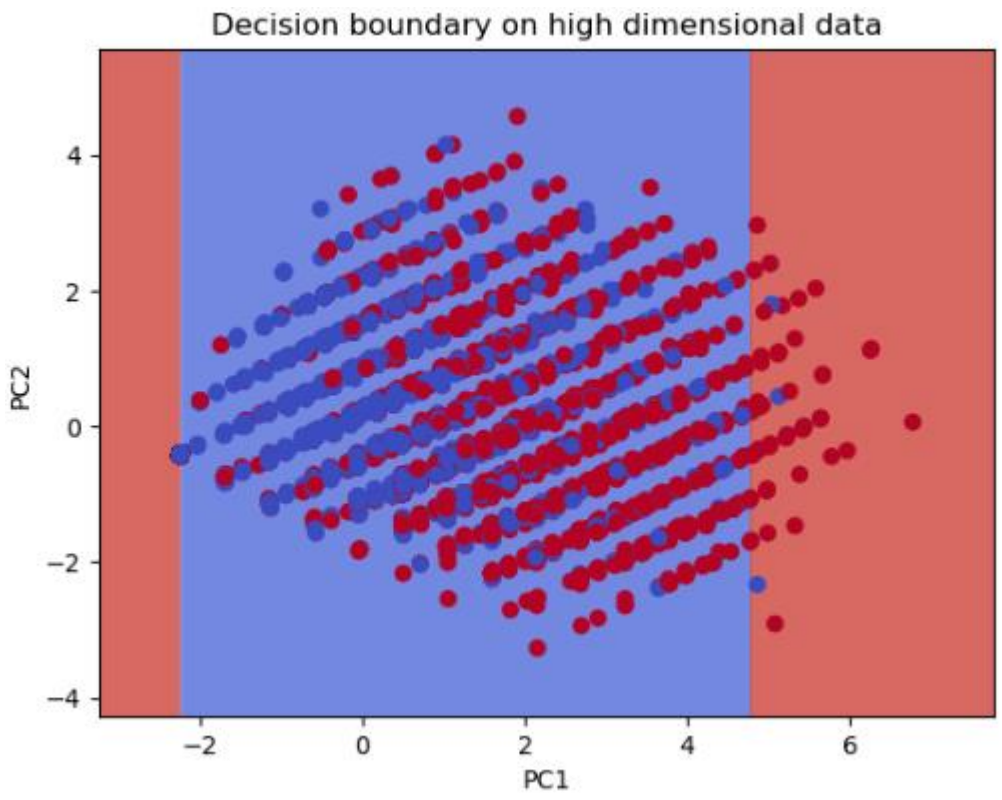


**Figure V-35:** Decision tree decision boundary for 50% to 50% split

# VI. Discussion, limitations and future work

## 6.1 Results evaluation

Naïve Bayes classifier is one of the simplest Machine learning algorithms and so it is usually not the best one. Same picture we can see in our results. However, it in some way surprising as Naïve Bayes usually works quite well with categorical data and in case of our dataset almost all features are categorical. Therefore, it is interesting to see best accuracy of around 50% (0.5) for 70% to 30% data split. Moreover, we can notice very low rate for recall (best = 24% ) and precision (best = 16%) values that usually a good way to measure results for imbalanced dataset like our one, where we mostly have classes with label 0 (do not recommend) as logically major part of all movies has low or average rating.

Unfortunately, we could not produce decision boundary for this algorithm and therefore it is hard to see visualization of any tendency in the prediction logic.

Decision tree algorithm results confirm same point regarding categorical data. Logically, Decision tree may have best results among all three algorithms, on the other hand it has lower performance than Logistic regression, even with mostly categorical features in the dataset. Still, this algorithms shows around 63% of accuracy. Moreover, it shows important improvement in recall value which equal to around 58% in best result scenario. It shows that Decision tree is very good at defining positive predictions (class label = 1, meaning film should be recommended). In case of recommender systems it is extremely important parameter.

Decision boundary plots shows logic behind decision tree algorithm where it tries to make straight cuts of dataspace with a help of tree nodes conditions. Note that it is data with reduced dimensions, so in reality there are much more rules and therefore cuts on the plot for each feature separately.

Lastly, our best algorithm by test results is the Logistic regression. It has 71% of accuracy, 68% of recall and even 43% of precision. Last parameter shows that this algorithm not only good at making positive predictions, but a lot of positive predictions are actually correct. It means that Logistic regression is not labeling all films to be recommended (class 1) and therefore it has good recall value. Instead it is actually good for true feature-based predictions.

So show plots of decision boundary. We can clearly see how Logistic regression tries to fit the decision boundary line in a most effective way separate both class data points.

Such results bring to the hypothesis that most impactful features in the dataset are duration and 5 categorical features of some film aspects (humour, tension, etc.). As by logic they should be better interpreted by Logistic regression.

Despite so different results across all three algorithms, they all clearly show that machine learning methods are very effective instrument in solving real world problems. For example, if we had to rely on a blind guess, it will have only around 37% of accuracy (All positive labels divided by total number of data points). This is almost half our best result with Logistic regression and Decision tree algorithms. Even Naïve Bayes classifier has much better accuracy. This is only basic algorithms and techniques. It shows that more complex recommender systems with hybrid approach and more advanced algorithms can be very accurate and deliver precise recommendation. Netflix, Amazon Prime Video, Disney+ and other video streaming services prove that point. In turn, this project shows the basic principles of these systems and proves their viability.

## 6.2 Limitations

Unfortunately, all projects inevitably have some limitations. Same applies to this one. As noted before, there was a problem with initial dataset and therefore we have lost individual user data as it is not presented in the new data. This means that currently our recommender system makes general recommendation based on average preferences of all users what is expressed in “avg_vote” column. Hence, current project does not fulfil initial recommender systems approach.

Another limitation is visualisation of algorithms work. Due to the high dimensionality of the data and the fact that all algorithms were made from scratch, it is difficult to compare results across graphs, as shown earlier. Before final solution with PCA, there were few more tries including t-SNE (t-Distributed Stochastic Neighbor Embedding) technique, but it was decided to stick with first option. It is still lack of some accuracy, but at least provides general idea of algorithm logic.

Final limit is that current project is able to work only with one particular dataset that contains crucial features for predictions such as humour, rhythm, effort, tension and erotism. Many other datasets that could be used for future work, or even current work given enough time, do not have these fields and therefore cannot be tested.

## 6.3 Future work

Based on the all the above, a set of future improvements can be identified:

- All three algorithms can be generalised for bigger number of datasets, so project can have more data to be tested with.
- Create more complex recommender approach using users data in order to make recommendations personalised
- Develop more complex machine learning algorithms such as random forest and SVD to evaluate and compare more options
- Explore different types of recommender systems such as collaborative-filtering and community-based approaches
- Last idea is to create user interface in order to insert custom data and see how different algorithms create suggestions. Compare results.

# VII. Conclusion

The main goal was to investigate recommender systems and create few algorithms that implement one of their approaches to finally evaluate and compare them. During background research was done big work to describe history and importance of recommender systems. In addition, principles and approaches underlying such systems were described in details. This paper also outlined initial machine learning algorithms that will be used in testing and evaluating as well as their mathematical principals, design, advantages and disadvantages.

Different techniques, tools and methods were used during implementation phase, including Data mining to find, prepare and process dataset for training and testing, Machine learning and Python for model assembling, training, testing and visualisation.

In addition to that, there was enough time to create extra algorithm – Decision tree. That was successfully tested and evaluated.

In conclusion, this project can be identified as a success. Despite some limitations and changes in the original plans, such as changing the approach to the recommender system or searching for a new dataset, the project gave all the necessary results. It successfully investigated and described the topic, showed examples of models, compared and evaluated them. Moreover, it clearly shows the values and capabilities of recommender systems.

## 7.1 Final words

During this project, I gained a lot of valuable experience. This inspired me to have even greater passion and interest in machine learning and computer science. Taught me to explore topics from different perspectives and pushed myself to find solutions to unexpected problems. In spite of all difficulties and hard work, I enjoyed this project a lot and developed a better and deeper understanding of recommender systems and the whole field of machine learning, as well as various tools that helped to develop this project.

# Bibliography

[1] F. Ricci, L. Rokach, and B. Shapira, ‘Recommender Systems: Techniques, Applications, and Challenges’, in *Recommender Systems Handbook*, F. Ricci, L. Rokach, and B. Shapira, Eds., New York, NY: Springer US, 2022, pp. 1–35. doi: 10.1007/978-1-0716-2197-4_1.

[2] Z. Dong, Z. Wang, J. Xu, R. Tang, and J. Wen, ‘A Brief History of Recommender Systems’. arXiv, Sep. 05, 2022. doi: 10.48550/arXiv.2209.01860.

[3] P. Resnick and H. R. Varian, ‘Recommender systems’, *Commun. ACM*, vol. 40, no. 3, pp. 56–58, Mar. 1997, doi: 10.1145/245108.245121.

[4] D. Jannach, P. Pu, F. Ricci, and M. Zanker, ‘Recommender Systems: Past, Present, Future’, *AI Magazine*, vol. 42, no. 3, Art. no. 3, Nov. 2021, doi: 10.1609/aimag.v42i3.18139.

[5] B. Pathak, R. Garfinkel, R. D. Gopal, R. Venkatesan, and F. Yin, ‘Empirical Analysis of the Impact of Recommender Systems on Sales’, *Journal of Management Information Systems*, vol. 27, no. 2, pp. 159–188, Oct. 2010, doi: 10.2753/MIS0742-1222270205.

[6] D. Jannach and M. Zanker, ‘Value and Impact of Recommender Systems’, in *Recommender Systems Handbook*, F. Ricci, L. Rokach, and B. Shapira, Eds., New York, NY: Springer US, 2022, pp. 519–546. doi: 10.1007/978-1-0716-2197-4_14.

[7] ‘Blockbuster Culture’s Next Rise or Fall: The Impact of Recommender Systems on Sales Diversity’. https://pubsonline.informs.org/doi/epdf/10.1287/mnsc.1080.0974 (accessed Nov. 11, 2022).

[8] R. Burke, ‘Hybrid Web Recommender Systems’, in *The Adaptive Web: Methods and Strategies of Web Personalization*, P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 377–408. doi: 10.1007/978-3-540-72079-9_12.

[9] C. Musto, M. de Gemmis, P. Lops, F. Narducci, and G. Semeraro, ‘Semantics and Content-Based Recommendations’, in *Recommender Systems Handbook*, F. Ricci, L. Rokach, and B. Shapira, Eds., New York, NY: Springer US, 2022, pp. 251–298. doi: 10.1007/978-1-0716-2197-4_7.

[10] Y. Koren, S. Rendle, and R. Bell, ‘Advances in Collaborative Filtering’, in *Recommender Systems Handbook*, F. Ricci, L. Rokach, and B. Shapira, Eds., New York, NY: Springer US, 2022, pp. 91–142. doi: 10.1007/978-1-0716-2197-4_3.

[11] ‘Naive Bayes Classifier Tutorial: with Python Scikit-learn’. https://www.datacamp.com/tutorial/naive-bayes-scikit-learn (accessed Nov. 14, 2022).

[12] A. Pujara, ‘Naïve Bayes Algorithm With Python’, *Analytics Vidhya*, Jul. 03, 2020. https://medium.com/analytics-vidhya/na%C3%AFve-bayes-algorithm-with-python-7b3aef57fb59 (accessed Nov. 14, 2022).

[13] C. M. Bishop, *Pattern recognition and machine learning*. in Information science and statistics. New York: Springer, 2006.

[14] C. H. P. 15 February 2022, ‘Implementing logistic regression from scratch in Python’, *IBM Developer*. https://developer.ibm.com/articles/implementing-logistic-regression-from-scratch-in-python/ (accessed Nov. 13, 2022).

[15] J. Thorn, ‘Logistic Regression Explained’, *Medium*, Sep. 26, 2021. https://towardsdatascience.com/logistic-regression-explained-9ee73cede081 (accessed Nov. 10, 2022).

[16] ‘Sigmoid function’, *Wikipedia*. Oct. 29, 2022. Accessed: Nov. 14, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=1118893984

[17] ‘FilmTV movies dataset’. https://www.kaggle.com/datasets/stefanoleone992/filmtv-movies-dataset (accessed May 04, 2023).

[18] J. Brownlee, ‘Naive Bayes Classifier From Scratch in Python’, *MachineLearningMastery.com*, Oct. 17, 2019. https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/ (accessed May 04, 2023).

[19] V. Jayaswal, ‘Laplace smoothing in Naïve Bayes algorithm’, *Medium*, Nov. 22, 2020. https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece (accessed May 04, 2023).

[20] D. K, ‘Logistic Regression in Python from Scratch’, *Medium*, Dec. 01, 2020. https://dhirajkumarblog.medium.com/logistic-regression-in-python-from-scratch-5b901d72d68e (accessed May 04, 2023).

[21] P. S, ‘Plotting Decision Boundaries using Numpy and Matplotlib’, *Medium*, Jan. 11, 2022. https://psrivasin.medium.com/plotting-decision-boundaries-using-numpy-and-matplotlib-f5613d8acd19 (accessed May 04, 2023).

[22] ‘sklearn.decomposition.PCA’, *scikit-learn*. https://scikit-learn/stable/modules/generated/sklearn.decomposition.PCA.html (accessed May 04, 2023).

[23] J. Brownlee, ‘How To Implement The Decision Tree Algorithm From Scratch In Python’, *MachineLearningMastery.com*, Nov. 08, 2016. https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/ (accessed May 04, 2023).

[24] E. Bujokas, ‘Decision Tree Algorithm in Python From Scratch’, *Medium*, Apr. 14, 2021. https://towardsdatascience.com/decision-tree-algorithm-in-python-from-scratch-8c43f0e40173 (accessed May 04, 2023).

[25] M. Karim and R. M. Rahman, ‘Decision Tree and Na&#239;ve Bayes Algorithm for Classification and Generation of Actionable Knowledge for Direct Marketing’, *JSEA*, vol. 06, no. 04, pp. 196–206, 2013, doi: 10.4236/jsea.2013.64025.

[26] S. Prabhakaran, ‘How Naive Bayes Algorithm Works? (with example and full code) | ML+’, *Machine Learning Plus*, Nov. 04, 2018. https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/ (accessed Dec. 30, 2022).

[27] S. Sekhar, ‘Math Behind Logistic Regression Algorithm’, *Analytics Vidhya*, Aug. 30, 2019. https://medium.com/analytics-vidhya/logistic-regression-b35d2801a29c (accessed Dec. 06, 2022).

[28] ‘The Movies Dataset’. https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset (accessed May 05, 2023).