## First working version (baseline code)

The iterative development process of the project started with the implementation of the baseline code. In order to keep the code clean and readable, all necessary for its execution libraries were imported at the very beginning. These libraries are:

* Numpy - useful for large, multi-dimensional arrays and matrices; it also contains a vast amount of useful mathematical functions.
* Pandas - it enables a quick and convenient way to load data from a csv file; it provides many Data Frames that allow an easy organization, exploration, representation, and manipulation of the data.
* Matplotlib - it is a comprehensive library for creating static, animated, and interactive visualizations in Python; a number of third-party libraries, such as seaborn can be integrated with it.
* Seaborn - it is a data visualization library based on matplotlib; it provides a high-level interface for drawing attractive and informative statistical graphics.
* Scikit Learn (Sklearn) - it features various classification, regression and clustering algorithms; it interoperates with the Python numerical and scientific libraries. Algorithms that were imported from this package:
  + CountVectorizer - a feature extraction method that converts the text content into numerical feature vectors.
  + TfidfTransformer - transforms a count matrix to a normalized tf or tf-idf representation
  + MultinomialNB - a classifier suitable for classification with discrete features, for example, word counts for text classification.
  + Pipeline - implements utilities to build a composite estimator, as a chain of transforms and estimators
  + train\_and\_test\_split - used for data preparation, by dividing it into training and testing sets.
  + GridSearchCV - searches over specified parameter values for an estimator; uses cross-validation to evaluate each value of a parameter grid and then chooses the best one.
  + Metrics - includes score functions, performance metrics and pairwise metrics, and distance computations.
* Warnings - used to ignore all occurring warnings by using warnings.filterwarnings()

Underneath the imported libraries all functionalities of the program were instantiated.

Firstly, by using the Pandas library the data from a csv file that contains movies plots, genres, titles, etc., was loaded. Subsequently, three functions were called by using Pandas DataFrame. The head() function was used to print the head of a DataFrame in a readable tabular format and display the first ten records of the data used in the program. The shape() function was used to display the dimensionality of the used data. The third function was dtypes(), which clearly depicted the data type of each column from the imported data.

The next step was to split the dataset into training and testing subsets. It is a crucial step in every machine learning process. It was done by using the train\_test\_split() function. The data relevant to our program were genres and plots. Their division proportions were established to 80% of the dataset for training and 20% for testing.

After splitting the data a Pipeline object was created and fitted to the plot and genre data. Methods used in the Pipeline setup were CountVectorizer with the stop-words filtered out in order to gain better performance, TfidfTransformer to normalise the counts to better represent the importance of words and MultinomialNB, which is a common classifier for text classification.

Another action performed by our program was the extraction of the ten most significant features for each movie genre that is in the database. It enabled us to check what words defined particular genres and whether it was rational to take them into account or should they be removed.

Following, a prediction test was carried out on the test set in order to check if the program works properly.

Then, in order to compare previously created pipeline with and without stop-words being filtered out, and with and without using a Tfidf transform, a grid search was used. It clearly estimated which parameters were the best for better performance. Moreover, using the „score” method the accuracy of the model was calculated.

Consecutively, another prediction test was carried out. This time for specific textual data consisting of words that were listed as those that have the greatest significance for a particular movie genre (action). The reason for this was to check whether the program would show the expected movie genre.

Further action in our program was the creation of a diagram with the fifteen most frequently occurring genre types in the chosen dataset. Based on that, we were able to identify to what extent some genres outweigh others, which was very important because such inequalities could also affect the training of the model and cause incorrect prediction of movie genres.

Additionally, in the end, the evaluation of the pipeline performance on the training set was implemented. The confusion matrix was computed to calculate the number of correctly and incorrectly classified examples in the data and based on that numbers a text report was built showing the precision, recall and f1-score metrics.

## Pipeline

One of the very first changes we made to our model after the baseline implementation was to switch from the CountVectorizer to the TfidfVectorizer which resulted in a small performance increase. [1] The difference between the two is only a small one, the value of TFIDF increases proportionally to count, but is offset by the frequency of the word in the corpus. This means less importance is given to extremely common words such as “the”, and more importance is given to rarer and more unusual words.

In addition to the change mentioned above, we also set use\_idf=True in the TfidfTransformer [2], this enables inverse document frequency reweighting when normalizing the count matrix from the previous step. This also led to a small performance boost.

Towards the end, we tested all the relevant text classification algorithms to see which offered the highest prediction accuracy, we tested the following algorithms [2]:

* (Linear) SGDClassifier is the algorithm we decided on in the end due to its impressive performance increase of around 20% compared to Naïve Bayes (the algo we started off with). It also has a large selection of loss functions for further fine-tuning (will get to that later).  
  accuracy: 78%
* (Linear) LogisticRegression is known as a crowd favourite among text classification circles and came in at a close second in terms of accuracy, making a strong case for linear-type classifiers being the best choice for our kind of problem. On a technical level, Logistic Regression is extremely similar to SGDClassifier, the only difference being that the latter has a Stochastic Gradient Descent solver.  
  accuracy: 70%
* (ensemble) RandomForrestClassifier methods correct for the overfitting found in decision trees.  
  accuracy: 62%
* (naïve bayes) MultinomialNB is a probabilistic classifier which depends on the Bayes theorem which statistically calculates an unknown parameter based on already known information but with a strong emphasis on making the features strongly independent. This algorithm performed ok and made a great baseline to compare to, however was just not suitable for our data.  
  accuracy: 54%
* (nearest neighbours) KNeighboursClassifier is better known for more loose data classification problems such as categorising hand written text due to its way of finding the next nearest match. However, in the English language, the next nearest match isn’t beneficial to the model and may even cause problems.  
  accuracy: 46%

Finally, we tried out various loss functions on the SGDClassifier, in the end we found modified\_huber to be the best [3] – not only because it had the highest accuracy but also because it allowed us to return the % certainty of the prediction. This meant making small changes to the API and GUI, as well as our model class in order for this to become visible to the user [4].

## File structure

Initially, our project was just a python notebook file, we quickly realised that making the program modular and readable on GitHub commits, we would require the use of standard python files. [5] We later decided that the program is better split up into multiple files, the first sign of us doing so is when we imported the data loader from an external file into our model script. [6] We followed this archetype of separating code by function into separate files thought our project.

## Class Structure

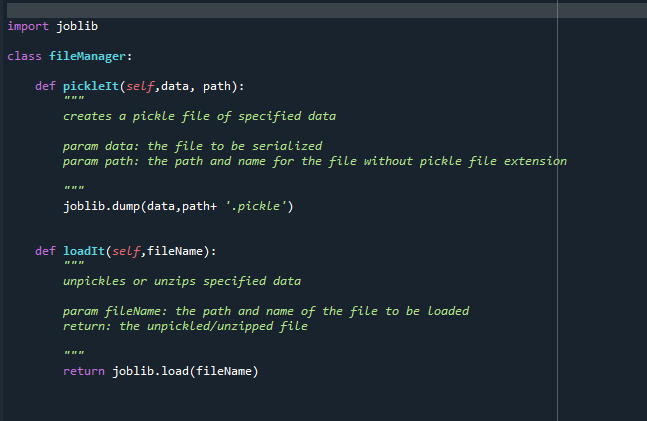
Part of the success of separating files out is that we can have a neat and organised class structure. When importing a class from a different file it is as easy as:

* from file import class

This allows us to instantiate classes from other files as objects, which was extremely useful when using the model in the API.py for example on lines 4, 14 and 23; in model.py on lines 14, 18 and on virtually every line of the demonstration script. This development choice paves the way towards our model being used as a library. You can see where we switched from a linear system to a class based system in [9].

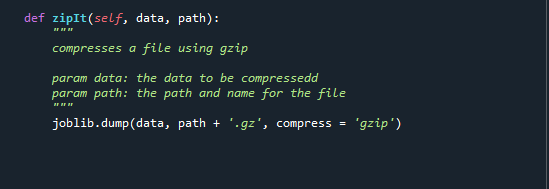
## Pipeline serialisation and API integration

When creating the API, we needed a way to integrate the pipeline so that our model could be used to service requests and return predictions on passed in movie plots. To achieve this, we created a file called modelLoader.py that held a set of functions to load and serialise objects. These functions are shown below:



These functions use the joblib module to serialise and load objects. A directory was included in our project repository to store the serialised models.

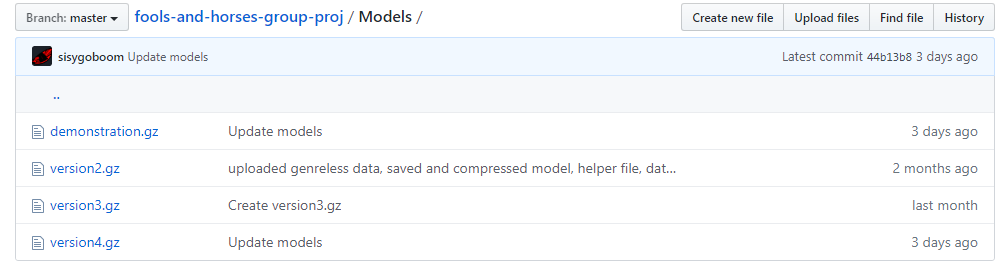
The first version of the pipeline we serialised was 375 mb, this caused issues with performance and made it difficult to push the pipeline to the project repository so we decided to add a new function to the modelLoader file that also added compression to objects when serialising them. This function is shown below:



After applying this compression to our pipeline, the size of the file reduced to 5.73 mb.

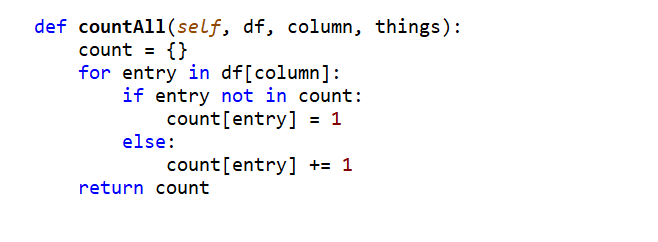
This process of compression was repeated for each version of the pipeline and each new serialised model was stored in the model directory to provide access to the API and any other application that needed to access them.

Below is a snapshot of the model directory from the project repository showing each version of the model:



## Data Prepossessing

In order to further the accuracy of our models’ predictions we looked at the dataset in which it was trained. The dataset used to train the previous models consisted of 34886 movie records with 186 unique genres. To explore this further a function was created to count the number of movie records for each genre, this function is shown below:



A sample of the top 16 genres in descending order of number of movie entries are shown below:

|  |  |
| --- | --- |
| Movie Genre | Movie count |
| drama | 9397 |
| comedy | 6468 |
| unknown | 6083 |
| action | 1355 |
| horror | 1209 |
| thriller | 1197 |
|  | 1148 |
| romance | 1006 |
| western | 874 |
| crime | 583 |
| adventure | 564 |
| musical | 491 |
| film | 465 |
| science | 445 |
| war | 376 |
| mystery | 315 |

It was evident from these results that the dataset was highly unbalanced, 70% of the total movie records were in the top five genres. This was causing our model to overfit and consistently predict the top 3 genres, drama, comedy and unknown. On top of the model’s imbalances it also had a large number of miscellaneous and empty movie genres. These were acting as noise in our dataset and lowering its prediction accuracy.

To correct these issues, we decided to remove these miscellaneous genres and balance the number of movies for each genre. In order to order to balance the dataset we found 2 additional datasets with the same format as the original and all 3 datasets were combined.

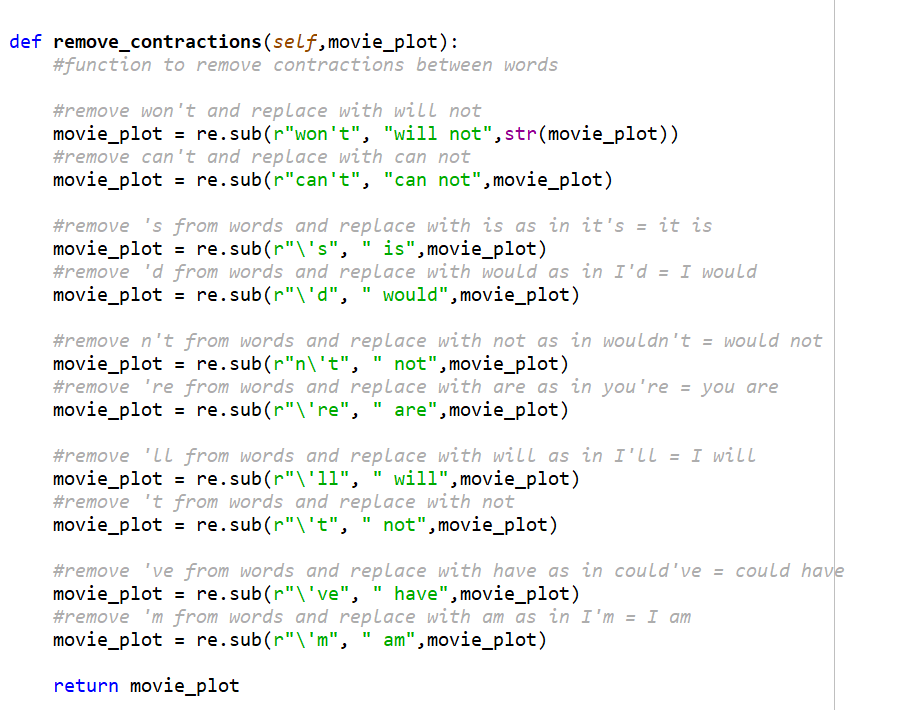
The 2 additional datasets we chose can be found here:

<https://github.com/ishmeetkohli/imdbGenreClassification/tree/master/data>

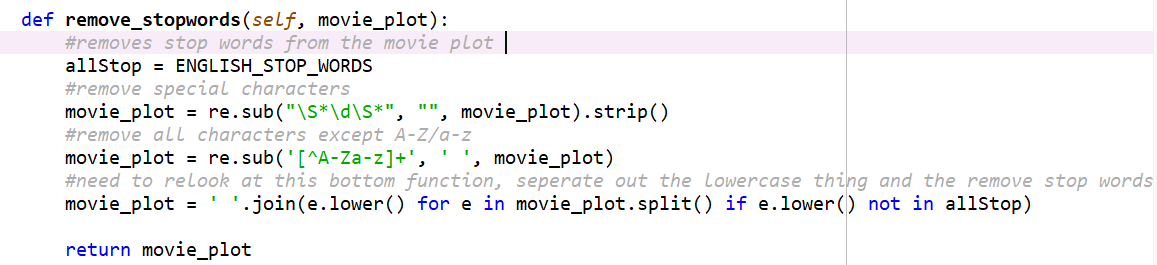
<https://github.com/shashankvmaiya/Movie-Genre-Multi-Label-Text-Classification/tree/master/Data>

The collective dataset was then filtered, any genres with less than 500 movies were removed and all other genres had their movie count limited to 1000. After the dataset was balanced, we set about standardising the movie plots by removing any special characters or numbers, we also removed any contractions between words and ensured all words were in lower class. The functions we wrote to standardise the movie plots is shown below:

Function to remove contractions:



Function to remove stop-words, special characters, numbers and to turn all words lowercase



After training the model on this new dataset the accuracy improved by 15% and was no longer overfitting on any 1 genre.

## Conda environment

As advised by our tutors, creating an environment .yml file is an effective way to make sure the environment we run our code in such as installed packages is reproducible across various machines. [7] For users who do not use Conda, we included pip installation instructions in the readme. [8]

## Demonstration script

To show off our code in all its modular nature, we created a demonstration script that works in a similar way to how a user would work with any other python library, they import the model class from the model file, instantiate it with the relevant parameters etc. Once a model has been loaded or trained, it opens the GUI and starts the API and the two cooperate instantly.

## Commit bibliography

[1] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/e4f802d07219dffa513882d902b9191702628ddf>

[2] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/e361aae1a36b260bb40d4d8d4a3629148af3a8f0>

[3] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/9aceea75963a6c217af16ecb73a84f4d2f068c90>

[4] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/2f808e8ea82f1f2210da3addc4522d143c808fd9>

[5] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/2d4c9b246674c16bc8b54b4bc85d1b62b8046c02>

[6] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/d40f817548d9b370e6228201bc9d2e2f64a606d0>

[7] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/00b5b2338d3e6bfb213d0408f365f0f8b3c7802d>

[8] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/86060792c19aa1c8a9b3652eb5245f10f83d5f7c>

[9] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/e9a51a0f44a21d71bb2cc28c0be891dc1bb4bb91>

[10] <https://github.com/sisygoboom/fools-and-horses-group-proj/commit/6d3d482878bf6617b92754de8ed6b19bc70d6f03>