LetterBoxd Movie Recommender

PHILIP BRADISH - 16339490, SHAUN JOSÉ - 16308185, and SARAH STAFFORD-LANGAN - 16316349

1 INTRODUCTION

We chose to attempt to build a recommender for a Letterboxd user's rating of a particular film using information about the film and the user. This will hopefully help users select films they are more likely to enjoy. The variety of features for each user and film make this problem interesting. Feature engineering is a significant aspect of the project. We have a large range of features to choose from, for both the film and the user. Details about the film include genres, ratings, crew, budget, revenue, release date, production companies and countries of origin. For users, we focus on their average rating, their country of residence, and their genre, actor, and director preferences. Combinations of these features can be used to train Linear, Lasso, Ridge, k-nearest neighbours, Multi-Layer Perceptron and collaborative filtering models. These models can then be used to predict a user's rating for a film. If we converted the model to an application, the user could input their Letterboxd username and the Letterboxd URL of a film and they would get back a predicted rating for the film.

2 DATASET AND FEATURES

Our dataset is from the movie-oriented social network, Letterboxd (www.Letterboxd.com). Letterboxd has over 1.5 million users who can track which films they have watched, 'like' films, and leave ratings and reviews on films. Users can rate a film between 0.5 and 5 stars. Each film's page has the number of times it has been 'watched' and 'liked' by Letterboxd users, as well as the average rating and the number of ratings. The names of each film's cast and crew are also available. Unfortunately, the Letterboxd API is still in a closed beta, so we decided to use BeautifulSoup to scrape our data from the website's HTML. Some other features we wanted to look at, such as a film's profit, were not available on Letterboxd. We found a website called the Movie Database (www.themoviedb.org, hereafter TMDb) where this data was available and used the TMDb API to collect this information.

2.1 Training Data

Letterboxd categorises its films into one or more of 19 different genres. To have a balanced dataset, we chose 500 films in each genre. 100 of these were the most-watched films of that genre on Letterboxd and the other 400 were randomly selected. After removing duplicates from the list, we were left with 7771 films. For each of these films, we collected the features we were interested in from the Letterboxd site and the TMDb API.

We gathered the usernames of Letterboxd's most popular "reviewers" of all time. These are users with a lot of followers and whose reviews get a lot of likes from other users. To ensure that each of the films we had selected would be part of our training data, we saved the username of the most recent reviewer of each film and added them to our training data. In total, we had 9565 users in our training set. For each user, we collected their ratings for all 'watched' films and generated an average for each genre and all directors and actors who worked on these films. We also collected the user's location, in plain text. We used the Google Maps API to turn this location into a country and retrieved the continent from the country using a look-up table.

We searched through every user to find any of the films in our training set that they'd seen. Once we found a valid user-film pair, we made sure to remove that particular film from the

Authors' address: Philip Bradish - 16339490, bradishp@tcd.ie; Shaun José - 16308185, josesh@tcd.ie; Sarah Stafford-Langan - 16316349, sastaffo@tcd.ie.

user's information and then 'merged' the relevant information, such as the user's average rating for that film's genres, directors and actors. After running this merge on all of the films and users, we had almost 2.75 million data-points (2,747,830).

2.2 Testing Data

Since each user and film would appear multiple times in our dataset, testing our model on an 80/20 split would likely mean testing on users and films that the model had been trained on. We decided to also test the models on sets of users and films that were completely unknown to them. We retrieved 150 more films for each genre. After removing duplicates, we had 1639 films to test on. As before, we looked at the other popular reviewers and collected one user who had rated each of our test films. In total, we had 3687 users to make up our testing dataset. After merging the users and films as we did in the training set, we had 165,427 completely unseen data-points to test our model along side an 80/20 split.

2.3 Features

2.3.1 Discarded Features before Collection. Initially, we wanted to use the titles of the films to create a feature matrix from TF-IDF using lemmatisation. TF-IDF, or term frequency-inverse document frequency, uses the product of the frequency that a token appears in the term (the film's title in this case) and the inverse of the frequency that the token appears in the document (the collection of all film titles). We dismissed this idea early on in the coding of our model because titles of films are, by necessity, specific and fairly unique, which would lead to most of our tokens being too 'rare' to influence the model.

There were also some features we had to discard before collecting. There was no simple way to access the gender of the directors and actors of a particular film and neither of the sites had information about what awards, if any, a film had won.

2.3.2 Discarded Features after Collection. Some of the information we collected for the films was discarded before and during the training stage. We had gathered each film's financial data from TMDb, including its budget and profit. However, many of the films we had chosen did not have financial information available in TMDB so we couldn't use these fields as features in the models. We also gathered a list of the production companies involved in each film. We were interested to see if the larger companies produced films that were generally higher rated than films from smaller companies. We discovered that there were too many companies who had only produced a small number of our films, which resulted in too many 'rare' features to use one-hot encoding effectively.

2.3.3 Included Features. Many of our features were numeric initially so they didn't need to be changed for the model, such as the average ratings or the number of ratings a film had. We used the film's number of 'views' and 'likes' to calculate the percentage of people who watched the film who had also liked it. For some of the features in the final dataset, it made sense to use one-hot encoding. We used one-hot encoding for the genres of the films, since films had variable numbers of genres. We also added one-hot encoding for the decade and the month of the year that the film was released in and as well as the continents on which it had been produced. Additionally, we compared the countries that the film was produced in to the country the user was from, and did the same for the continents.

3 METHODS

We wanted to try out as many different models as possible so that we could compare them. Unfortunately, our choice was somewhat restricted by the massive size of our data set and the huge number of parameters we were experimenting with.

We started with Linear regression. It assigns a parameter coefficient to each input feature. The values of these coefficients are changed to make the model's predicted values of the training data as close to their actual values as possible. It uses a gradient descent algorithm to ensure that the cost function, calculated using the mean square error, decreases at every step. Despite its simplicity, Linear regression ended up performing very well. The fact that it lacks a penalty parameter didn't seem to matter since we used millions of training points which made over-fitting less of an issue.

Next, we used Lasso and Ridge regression models. These are both based on Linear regression, except they have penalty parameters. Ridge regression uses an L2 penalty. This adds the square of each coefficient to the cost function. This incentivises the model to keep the coefficient values low in order to minimise the cost function. The added penalty is weighted by the inverse of the hyperparameter, C. This is used to control how big an impact the sizes of the parameters will have on the cost function. As C grows larger, the penalty will become smaller and the model will begin to behave like a Linear regression model. As C tends towards 0, the model will start to prioritise keeping the parameters small even when they aren't a good fit for the data. Lasso regression uses an L1 penalty which uses the absolute value of the coefficients instead of their square. This has several notable effects. For L1 penalties, unimportant values will be set to zero to minimise their penalty. In the case of an L2 penalty, they would just be given very small values. Also, L2 penalties typically require smaller values of C to introduce a notable penalty. We performed cross-validation to choose the optimal C values for these models.

Despite our huge amount of data and features, we decided to attempt k-nearest neighbours and kernalised ridge regression. We were interested to see if we could get them to work and how their results would compare with our other models. K-nearest neighbours (kNN) models directly use the training points to predict the output. It finds the distance between the input points and all the training points. There are several methods to do this but the most common is to calculate the Euclidean distance. It then takes the K nearest points and uses them to predict the value. Originally, we had planned to use a Gaussian weighting function to give a higher weight to nearer neighbours. However, given the huge number of parameters and training points we were dealing with, we ran into lots of numeric issues. In the end we used a uniform weighting function which assigns all the nearest points equal weights. The model ended up working, but the shortcomings of kNN were evident. It took significantly longer to run and the results were worse than our other models. Kernalised ridge regression fared even worse. Similar to kNN it uses the training data directly to predict the output values. However, it uses all data points instead of just the nearest neighbours. It also assigns each point a parameter which can be adjusted when training to reflect the importance of the point. In the end it required too much memory to run on even one of our data files containing 100,000 points.

Another model that we wanted to use was a neural network. Neural networks are flexible and work well with non-linear data and a high volume of data. They can easily lead to over-fitting, and can take a long time to compute when there are a lot of data points. However, they are quick to predict results, so a neural network regressor would work well if we built an application using our model. We decided to use sklearn's MLPRegressor model. MLPs can have multiple hidden layers, each containing a number of nodes. The default activation function for each node is 'relu', the rectified linear unit function, which simply returns its input value, or 0 if the input is below 0. MLP uses an L2 penalty, as explained above.

Finally, we wanted to try item-based collaborative filtering. This algorithm is specifically designed for recommender systems so we believed it could provide good results. We used a data frame with each row containing a user id, film id and that user's rating of the film. We then performed item-based collaborative filtering, by finding the set of users who rated every possible pair of films. Once we had this set, we could find out how similar each pair was. This could be done by multiplying a user's rating for one film by their rating for the other film. We did this for the set of all users who watched both films. After normalising the sum of the products (by dividing it by the root of the sum of ratings per film by

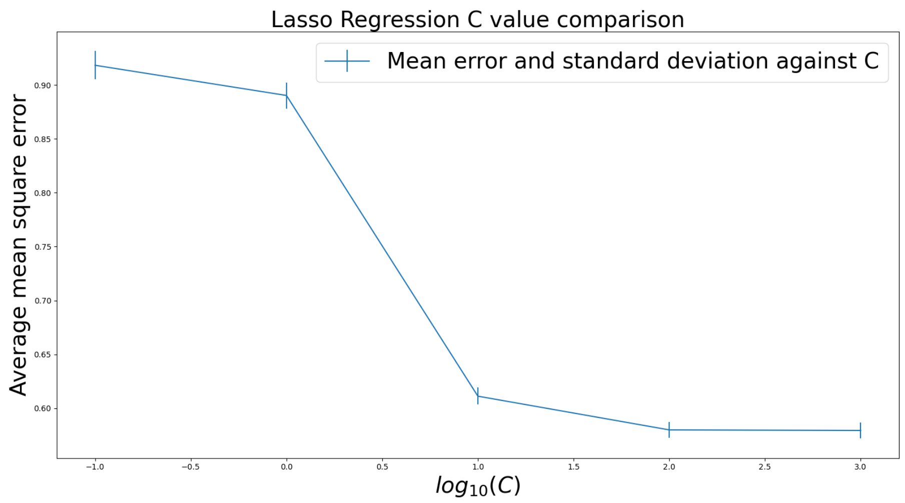

Fig 4.1 Average mean square error of various C values during 5-fold cross-validation of Lasso Regression models

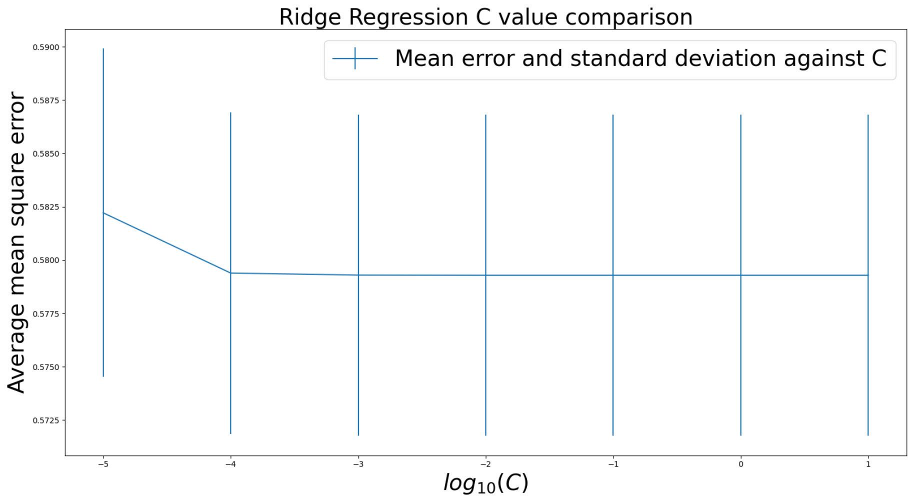

Fig 4.2 Average mean square error of various C values during 5-fold cross-validation of Ridge Regression models

each user), we end up with a numerical value of how similar the two movies in the pair are. Now that we have the similarity information, for each film, we find the ‘k’ most similar films (nearest neighbours) and add up the product of the films’ similarity ratio with the user’s rating of the particular film. A high rating of a film which is very similar would yield a high number. A low rating of a film which is very similar would yield a lower number, and so would a high rating of a film which is not very similar to the film. After normalising this sum by dividing it by the sum of similarities of the ‘k’ nearest films, we end up with a predicted rating of the film by that user. Unfortunately, this system wouldn’t work on our unseen testing, since all ’users’ and ’items’ (films in our case) tested have to have been part of the training set.

Given more time, we would have explored other recommender techniques such as SLIM and matrix factorisation.

Root Mean Square Errors	80/20 train/test split	Unseen Testset	Average star error
Baseline: Mean of Training Output	1.01703	1.01942	±1.0182
Baseline: Constant = 2.5	1.01676	1.01851	±1.0176
Baseline: User's average rating	0.96669	0.96632	±0.9665
Baseline: Film's average rating	0.83458	0.83512	±0.8349
Collaborate Filtering	0.7614441	NaN	±0.7614
Lasso Regression (C=100)	0.7609915	0.7613429	±0.7612
Linear Regression	0.7605842	0.7609039	±0.7607
Ridge Regression (C=0.0001)	0.7605845	0.7609045	±0.7606

Fig 4.3 - Root mean square errors of each of the baseline and predictive models that were performed

4 RESULTS

4.1 Training and testing points

As mentioned in section 2, we collected 9565 users and 7771 films and combined them to create 2747830 training points. We used 80% of our data for training and 20% for testing. This meant we trained on 2198264 points and tested on the remaining 549556. We also collected independent testing data from Letterboxd and TMDb. We collected 3687 additional users and 1639 additional films to generate 165427 points that were completely new to our trained model. This gave us some confidence that our system wasn't over-fit to our trained users and films.

4.2 Hyperparameters and Functions

4.2.1 Cross Validation. All hyperparameters of the Lasso and Ridge models were chosen using cross-validation. Since we were working with almost 2.75 million data points, we used five-fold cross-validation to keep training times reasonable. We attempted to run cross-validation on kNN regression models but we found that the process was too computationally expensive to run. For our Lasso and Ridge regression models we needed to choose *C* values. For both models, we iterated through a list of potential values and used them to instantiate models. For each one, we collected the average mean square error and standard deviation and plotted their results using `pyplot`.

In the case of the Lasso regression, we tested values 0.1, 1, 10, 100 and 1000. The plot can be seen in *Fig 4.1.* A *C* value of 100 gives the lowest standard deviation and the joint lowest error. For these reasons, we chose it as our optimal value.

In the case of the Ridge regression, we tested values 0.00001, 0.0001, 0.001, 0.01, 0.1, 1 and 10. As mentioned in the Methods section we need to make the ridge regression C values smaller than the Lasso regression ones. The plot can be seen in *Fig 4.2.* A lot of the C values have the joint lowest error and standard deviation so we chose 0.0001. This is the smallest value and therefore the one least likely to produce an over-fit model.

4.2.2 Gradient Descent. Our main models were Linear regression and its variants (Lasso and Ridge). We used gradient descent to find their optimal coefficients. With Linear regression, the cost function must have a global minimum point. By calculating the gradient of our current point on the curve, with respect to each coefficient value, we end up with the slope of the curve for each parameter. We then need to go down the slope towards the minimum. The extent of how much we travel down depends on the step size, which is multiplied by the negative value of the slope. This value is used to change each coefficient. A small step size could result in taking too long for the algorithm to converge or the algorithm settling on a local minimum instead of the global minimum. A large step size might cause us to continuously overshoot the minimum.

4.2.3 Primary Metrics. For all of our models, including the baselines, we calculate the mean square error to evaluate their performance. This is done using 20% of the data for testing, and also on testing data with users and films that don't appear in the training data. We then took the root mean square error (hereafter RMSE) to evaluate the average number of stars we are off from the users actual rating.

4.3 Predictions

To evaluate the performance of our models we also created several baselines. We used sklearn's `DummyRegressor` class to create a baseline to predict the mean of the training outputs or a constant baseline of 2.5 stars, halfway between 0 and 5. Neither of these baselines was able to get a mean square error of less than one. We also designed our own 'intelligent' baseline which takes the value of an input feature and always predicts that. We called it `InputFeatureBaseline` and used it to create two extra baselines that predict the film's average rating and the user's average rating for each pair. These were a lot more competitive. Furthermore, they are what the user would intuitively expect to rate a film so if we beat them we could prove the worth of our system.

Our models were trained using twenty-nine features so they took a significant amount of time to process our 2.75 million points. The features were chosen by experimenting with different combinations of input features to see which ones would generate the best models. The features we ended up using were; the film's average rating, the user's average rating, the film's total likes, views and ratings, the film's age, whether the film belongs to a franchise, the film's rate and like ratios, the user's number of films watched, the user's average rating for the director of the film, and the user's average rating for each genre of the film using minus ones for genres which the film doesn't belong to.

The table *Fig 4.3* shows our results across the two testing sets for all models and baselines. We can see that all of our models beat all of our baselines by a notable amount. The average rating of a certain film is by far the best baseline indicator of a user's rating for a film, which is what we expected, with an RMSE of 0.835.

Our best performing models were Ridge Regression for C=0.0001 and Linear Regression, whose RMSE values were very similar, at 0.7606 and 0.7607 respectively. Linear Regression appears to be marginally better at predicting the results for the completely unseen data, but this isn't enough to be notable. Lasso Regression for C=100 is almost as good as the best two models, with an RSME of 0.7612.

Collaborate Filtering is also a reasonable model for predicting the ratings in the 80/20 split, with an RMSE of 0.7614. However, this algorithm doesn't have the ability to handle unforeseen users and items, so it would only work to predict ratings for users and films that we've trained for. This means that the collaborative filtering model wouldn't work to solve the problem we set out to solve, unless we collected the ratings of all films seen by all of Letterboxd's over 1.5 million users.

Finally, we ran MLP models using various combinations of alphas, numbers of layers, layer sizes and learning rate functions. Since neural networks are slow to train on massive amounts of data, we trained the model using 131867 points and tested with 13720. We wanted to investigate if any MLP model was notable enough to run on the whole dataset. The RMSE for all of the sample perceptrons ranged from 0.87 to 1.05, so we didn't run any MLP models on the entire dataset.

5 SUMMARY

The results for our Linear Regression as well as one of each of our Lasso and Ridge regression models ended up being the same as the item-based collaborative filtering model. However, they had the added benefit of fast training times and could predict a rating for any user and film we input. All of these models had a prediction error of about three-quarters of a star. This is an improvement over our best performing baseline which always predicts the film's average rating. This had an error of approximately 0.83 of a star. This proves that our system can provide a better recommendation than simply looking at the average rating on Letterboxd. Additionally, we showed that our best models can perform well on completely new data which shows that we have avoided over-fitting. However, an error of three-quarters of a star is still quite significant. Our regression models could probably be improved by performing more feature engineering. We could also improve our item-based collaborative filtering to make it more efficient and able to process the whole dataset. Alternatively, we could have tried SLIM or matrix factorisation and see if they could generate better predictions.

6 CONTRIBUTIONS

6.1 Philip Bradish

Collected a list of films and random users off Letterboxd. Collected information on the films from Letterboxd. Combined the data from TMDB and Letterboxd for each film. Wrote the code to test the various models and combinations of features. Also wrote the cross-validation and graph code. Worked on the methods, experiments and results parts of the report.

PB

6.2 Shaun José

Collected the details of training and testing films that were retrieved by Philip, from the TMDb API. Created a manual country-continent pair file, and formatted it to countries-list by continent for Sarah to work with. Used Sarah's file of the user-film pair merge, then filtered it appropriately to run Collaborative filtering on those user-film pairs. For the report, I wrote the introduction, collaborative filtering part of methods, and some parts of experiments and results.

[illegible]

6.3 Sarah Stafford-Langan

Collected the usernames of the popular users from Letterboxd. Collected information on all of the popular users and Philip's users using web scraping from Letterboxd. Used the Google Maps API to get a user's country from their string location. Used Shaun's country-continent mapping to categorise the users' countries. Merged each user with each of the films that they'd seen in the dataset to create the data-points for the model. Generated CSV files using the merged JSON files which would be faster for the computer to process before modelling. Ran models using Philip's code and added MLP regression. Wrote the Dataset and Features section of the report as well as the parts of the methods and results sections about Multi-Layer Perceptrons. Formatted and edited the report.

[illegible]

7 APPENDIX

All of our code is available at:
github.com/sastaffo/LetterBoxd_Movie_Recommnder