A hybrid book recommender with stylistic elements

MSc data science project report

Department of Computer Science and Information Systems, Birkbeck College, University of London, 2020

Student: Alec Johnson

Student number: 13155061

Contents

1. Abstract

Book recommendation can be based on the reviews of other users (collaborative filtering) or on the text or other features of the books themselves (content-based methods). Hybrid methods can combine these methods to overcome the weaknesses of each. Challenges include the cold-start problem, which stops collaborative filtering recommending new or obscure books, and overspecialisation, which can make content-based recommendations repetitive and boring.

This project presents a recommender system that makes personalised book recommendations and avoids the cold start problem. Its hybrid approach combines collaborative filtering with content-based methods. The content-based methods include stylistic techniques, which previously have rarely been combined with collaborative filtering.

To enable this, I built a dataset of books and user ratings, and a framework for exploring different algorithms and vector representations of books. Hybridisation led to a small but consistent increase in overall accuracy, and using stylistic methods improved it further. As a next step I'd investigate using factorisation machines to capture the interaction between features.

My most successful approach used:

- naive Bayes for collaborative filtering
- latent semantic indexing to create vectors of book vocabulary
- a system of stylistic weightings developed by Brooke, Hammond and Hirst (2015)
- logistic regression to make vocabulary-based recommendations
- logistic regression to make style-based recommendations
- a support vector classifier to make a final hybrid recommendation

This project was supervised by Nigel Martin.

2. Introduction

Books are an ideal subject for recommender systems. There are huge numbers to choose from, and they take a long time to consume compared to, say, films or songs. Enjoyment goes far beyond topic and genre, also depending on difficult-to-define concepts like style, plot and characterisation.

The key techniques in book recommenders are collaborative filtering and content-based filtering (Alharthi, Inkpen and Szpakowicz, 2018). These are often combined in hybrid systems (Ning, Desrosiers and Karypis, 2015). Both approaches build a user profile based on known reviews or purchases, and then identify similar items to recommend.

2.1 Collaborating filtering

Collaborative filtering has traditionally been the most popular technique (Pera and Ng, 2014). It makes recommendations to one user based on how other users have rated books (Aggarwal, 2016). In a matrix of users against books, where each element is a rating, existing ratings are used to predict the ones that are missing.

One of the main problems with collaborative filtering is that books can't be recommended if they have few or no ratings. This is called the 'cold-start problem' (Zhang, Chow and Wu, 2016). To be recommended to one user, a book must already have been read and enjoyed by other users. This makes collaborative filtering bad at recommending new or obscure books - even if they'd be a good match for a user.

2.2 Content-based recommendation

Content-based recommenders express each book as a vector and then look at similarity with books a user has enjoyed. This is similar to collaborative filtering, but here the vectors are based on characteristics of the books themselves, rather than user ratings. Often those characteristics are tags, metadata, summaries or descriptions (Pera and Ng, 2014). Less commonly, it's the book's entire text (Alharthi, Inkpen and Szpakowicz, 2018).

In full-text systems a major question is how to turn the text into a vector. This often begins with a 'bag of words', made up of counts of vocabulary terms (Mooney and Roy, 2000). Dimension reduction techniques can help reduce the sparsity of those vectors.

Content-based recommendation is often a good way to avoid the cold start problem (Alharthi, 2019; Aggarwal, 2016; Zhang, Chow and Wu, 2016). A purely content-based system can recommend a book without any user ratings.

However, content-based methods have their own problems, such as overspecialisation. This means making recommendations that lack 'novelty and serendipity' (Aggarwal, 2016). This can happen when a system focuses all its recommendations on a narrow section of a user's tastes.

2.3 Hybrid recommender systems

Hybrids combine the strengths of different recommenders. They can offer a chance to avoid the cold start problem without plunging into overspecialisation.

The most popular family of hybrids is 'ensemble methods' (Burke, 2002). These take several recommenders and combine their results into a single, better output - for example, mixing content-based and collaborative filtering (Vaz, Martins de Matos and Martins, 2012). Ensembles have been described as being better than any single type of recommender system (Amatriain and Pujol, 2015).

3. Specification and design

My project proposal set 3 aims:

- make personalised book recommendations based on a user's ratings
- avoid the cold-start problem by enabling predictions for books with few or no ratings
- measure overspecialisation

To meet these aims, my proposal suggested a hybrid recommendation engine that combined collaborative filtering with content-based methods. On the content side, I planned to explore stylistic features, as so far little has been done to combine them with collaborative filtering.

In designing this system, I set requirements for each step of building a recommender engine.

3.1 Acquiring data

I needed 2 types of data: user ratings for collaborative filtering and assessing my results, and the full content of the books being rated, so I could pursue content-based recommendations.

3.1.1 Book content

For book content I required:

- text files with minimal formatting and consistent encoding
- English language files
- long-form prose fiction content

Straightforward text files were a practical necessity. It's possible to overcome inconsistencies and unusual encodings, but this would use time better spent elsewhere. To maximise focus on the actual recommender system, I needed to minimise time spent processing files.

Requiring English language was vital. With multiple languages, even basic techniques like counting vocabulary items would be hugely complex. By many measures a book would differ more from its translation than it would from a book with totally different style and topic in the same language. This would make it impractical to make content-based recommendations based on vocabulary or stylistic similarity.

I chose long-form prose fiction to make a more consistent dataset, and so attempt more meaningful recommendations. If the dataset combined poetry, plays, non-fiction, short stories and long-form fiction, recommendations might focus on those overt distinctions. As these differences are obvious to users, those recommendations would be less valuable.

3.1.2 User ratings

For ratings I needed:

- unique user identifiers
- clear quantitative or categorical ratings
- enough metadata to link the ratings to the book texts

User identifiers were necessary for creating user profiles and avoiding muddling users. Any recommendation engine has to know what someone already likes or dislikes before it can suggest anything.

Ordered ratings were an obvious necessity. Purely descriptive reviews are no use to a recommender system. However, it didn't matter whether the ratings were numeric or categorical. My proposal had found that categorical would be more suitable, but numerical ratings can be converted into categories.

The ability to link ratings back to book texts was a tougher requirement. Title and author might not be enough. Different books can have the same title, and the same book can have different titles (for example in different editions). Additional metadata like genre or year of publication could make joining the texts easier.

3.1.3 Data quantity

Whatever data sources I used would need to give me enough data for statistical analysis. There's no fixed way to measure this, so I looked for a benchmark in existing research.

The most notable dataset involving the full text of books is LitRec (Vaz, Martins de Matos, Martins and Calado, 2012). It contains 38,591 ratings from 1,927 users, and spans 3,710 books.

Other popular datasets are bigger, but tend to focus solely on user ratings. For example, the BookCrossing dataset (Ziegler, McNee, Konstan and Lausen, 2005) includes 1,149,780 ratings from 278,858 users, and covers 271,379 books. Goodbooks-10k (Zajac and Zygmunt, 2017) contains roughly 6 million ratings spread across 10,000 books.

However, my proposal had settled on using the full texts of books, and most of the books in these larger datasets were still in copyright. Similarly, most out-of-copyright books weren't in these datasets, which focused on more popular and more recent fiction. As a result, I used LitRec as my benchmark.

Based on LitRec I set approximate minimums for overall number of ratings (around 39,000), average ratings per user (roughly 20) and average ratings per book (roughly 10). It would be no use having 40,000 ratings if they were all by different people about different books.

3.2 Defining a recommendation

Many book review websites use a 5-star system. This looks straightforward but creates problems for analysis. My proposal noted that 5-point ratings can be hard to interpret because of variation in how people use the scale. For example, some reviewers rarely or never drop below 4 stars.

My proposal suggested splitting each user's ratings into those above that user's median and those at or below. This would help with the problem of people using the scale differently. However, it would create new problems. If someone only rates 4 and 5 stars, it makes little sense to consider their 4-star review equal to a 1-star review by someone who uses the whole scale. The same goes for treating a 2-star review as a recommendation just because it's from a reviewer who only rates 1s and 2s.

So in a change from my proposal I decided to make a binary classification that split ratings at a single point for the whole dataset, rather than for individual users. I would decide this point once I had my data, by looking at how its ratings were distributed - perhaps using its median.

My proposal noted a risk of some review sources being too positive (noting the top-5 reviewer on Goodreads who only gave 5-star ratings). This meant an additional requirement: my dataset had to contain enough variety to produce a roughly equal distribution of good and bad ratings - however I defined good and bad.

3.3 Evaluation strategy

Before I could design the detail of my recommender system I needed to specify how I'd assess its accuracy.

My proposal had settled on offline evaluation based on unranked accuracy. Many papers use precision at K (with K usually set to 10), but this was unreliable for a fixed offline dataset.

The problem is that each user will have read only a tiny proportion of all possible books. In an offline evaluation, when an unread book appears in someone's top recommendations, there's no way to know whether that's a good recommendation. So I'd have to leave those books out of the precision calculation: if I recommended 10 books and the user had rated 3 of them, I'd calculate precision based solely on those 3. But as most recommendations would be for books the user hadn't read, I'd have few or no ratings to assess precision on.

An alternative would be to score precision at K based only on books the user had read. However, my benchmark dataset, LitRec, only had an average of 20 ratings per user. With K set to 5, even with a 50:50 split of test and training data, I'd still have many users whose entire test set would be no larger than K. For those users, all books would end up being recommended.

To escape these problems, I decided to split the whole dataset into training and test sets, make predictions for the whole test set, and measure their accuracy. As well as avoiding the problems of ranking, this would let me use a smaller test set, leaving more training data.

In this approach my main metric would be accuracy: the proportion of correct predictions in the test set. I could use additional measures like precision, recall and F1 score to understand what my recommendations got wrong.

This meant I'd need a way to produce consistent test and training splits. I'd also need evaluation functions to quickly check predictions and produce summary statistics. Finally, I'd have to store and combine each user's scores into an overall assessment of the experiment.

3.3.1 Discarding overspecialisation

The third aim of my project was to investigate overspecialisation. However, by settling on overall accuracy evaluation, rather than ranking, I could no longer realistically assess this.

Measuring overspecialisation is only relevant to systems that select a specific number of top recommendations. Its aim is to make sure those recommendations are interestingly varied. However, when evaluating by overall accuracy I'd be classifying all books as recommended or not recommended, and comparing these predictions to reviews in the test set. There's no set of top recommendations to select.

Reducing overspecialisation is about removing a book from the top recommendations because it's boringly similar to others in the list. It doesn't mean the user won't enjoy that book. So with an evaluation strategy based on overall accuracy, there's no sensible way to adjust for overspecialisation.

I'd already noted in my proposal that the overspecialisation objective was optional, so I wasn't concerned about removing it. It reduced the scope of the project, but there's still plenty to explore when trying to personalise recommendations and avoid the cold start problem.

3.4 Making recommendations

I'd be trying to make 3 types of recommendation: those based on user ratings, those based on book content, and hybrids that combined or chose between these. For each, I'd need to be able to explore different techniques to identify what worked best. This meant requirements for:

- switching between algorithms and parameters without large code changes
- configuring settings to select different algorithms and parameters for each test
- recording which settings were used for each test
- controlling randomisation so I could fully replicate each experiment, to make sure my code was working as intended and didn't depend on the computer running it

3.4.1 Algorithm choices

Given the variety of options for recommending and hybridising, I needed to select an initial set of binary classification algorithms to test. My proposal had noted that K nearest neighbours (KNN) and naive Bayes would be good starting points. To these I added logistic regression, support vector classifiers (SVC), and decision trees.

Logistic regression is widely used for probabilistic classification and can perform well at a variety of tasks. Similarly, support vector classifiers are known for their versatility. They also have a range of kernel options that can drastically change their effect. This could make them be useful for both hybridisation and initial recommendation.

I selected decision trees because they're highly interpretable, especially when heavily pruned. This could make them useful for hybridisation, when I'd have a small number of features and want to know how they interacted. However, individual trees tend to be high variance, which might make them unsuitable for high-dimension recommendation based on content or user ratings.

These algorithms are all available in Scikit Learn, where they have consistent methods and parameters. This will make it easy to fulfil my requirement for easily switchable settings.

3.4.2 Content processing

Content-based recommendation relies on converting each book into a vector representation of its content. To do this, I had to set requirements for how I'd preprocess the content and turn it into vectors.

For preprocessing, I knew I'd be working with a large amount of data. Books can be long, so were likely to require more processing than ratings, for which I'd just need a user ID, a book ID and a binary classification. So I needed to make sure my preprocessing could be run in advance and saved. This way I wouldn't have to repeat it every time I tried new settings.

There are many ways to preprocess, with decisions around how to lemmatise or stem words, how to deal with names and places, how to deal with numbers, what to do with case, and how to approach stopwords. I could easily plan a whole project around working out the most valuable techniques.

If I wanted to test different methods of personalising and hybridising, I wouldn't have time to also explore preprocessing thoroughly. So I made it a requirement that I'd find a single approach to preprocessing. It would have to create versatile texts that could be viable across different algorithms and vectorisation approaches.

For vectorisation, as with recommendation, I had many methods to choose from. They fall into 2 main categories. The first are vocabulary-based methods, such as term-frequency/inverse document frequency (TFIDF), plus dense-vector variants such as latent semantic indexing (LSI). The second are stylistic methods. These are less well defined, but something I planned to explore based on previous research such as Alharthi (2019).

Knowing I'd be exploring these different vectorisation approaches, my settings would need to specify a vectorisation method for each experiment. The vectors therefore had to be fully interchangeable, regardless of their length or scale. I'd need to normalise or otherwise adjust them in a way that lets them work with different algorithms - for example adjusting for naive Bayes' inability to work with negative numbers.

3.5 Code structure

Because I'd be experimenting with different settings, I knew I'd benefit from an object-oriented approach that created objects to encompass the data, each experiment, and various derived characteristics. Compared to using dozens of function parameters, passing objects around my code would be easier to track and easier to keep consistent.

I'd also need a single way to run each experiment, regardless of its settings. This would make it easier to run varied but replicable tests. I therefore set a requirement for a central script that reads my settings and calls modules to prepare the necessary data and perform the different stages of analysis and evaluation. Each module would encompass one of the main steps of the recommender engine:

- loading data
- processing data
- vectorising content

- analysing users
- making predictions
- hybridising predictions
- evaluating results

3.6 Summary of requirements

Book content	• Consistent, unformatted text files • English language content • Long-form prose fiction
User ratings	• Unique, unnamed user identifiers • Clear quantitative or categorical ratings • Enough metadata to connect books to other data sources
Data quantity	• Similarity to LitRec in overall size (38,591 ratings), average reviews per user (20) and average reviews per book (10) • Roughly equal numbers of good and bad ratings
Evaluation	• Consistent way to split into training and test sets • Statistical functions to evaluate predictions • A way to combine each user's evaluation into overall results
Making recommendations	• Configurable settings for each experiment • Results that record the settings used • Fully replicable experiments
Using algorithms	• At least 5 basic algorithms: KNN, naive Bayes, SVC, logistic regression and decision trees • Ability to easily switch between different algorithms
Content vectors	• Preprocess in advance and store and reuse results • Simple preprocessing rules valid for all vocabulary vectors • Normalise vectors to work with different algorithms • Ability to specify a vectorisation method for each experiment
Code structure	• Object-oriented approach to data, experiment and results • Central runner script to start all experiments • Separate modules for loading data, processing data, vectorising content, analysing users, making predictions, hybridising predictions and evaluating results

4. Implementation

The whole project was implemented in Python. Full dependencies are listed in my requirements.txt file.

4.1 Getting book content

My proposal had identified Project Gutenberg as a possible source for book content. It easily fulfilled my requirements for English language content in a clean, consistent format. Content is available in many languages and formats, but I could filter their download pages to just English-language .txt files.

My requirement for long-form prose fiction took more work. I downloaded the Project Gutenberg metadata catalogue, then isolated the books that mentioned 'fiction' in their tags. This used the Requests library and BeautifulSoup to parse the metadata's RDF files (notebooks/1_read_rdf.ipynb).

This confirmed Gutenberg's suitability. Next I wrote a script to get a full list of download links from Gutenberg and merge it with my list of books tagged as fiction. The script then downloaded all the relevant books (notebooks/2_get_books.ipynb). I again used Requests and BeautifulSoup to parse Gutenberg's download pages, plus Pandas to merge the lists of books.

I used notebooks for these one-off scripts, as once I had the texts downloaded I didn't need to maintain and improve the code.

These scripts gave me 15,167 books to work with. This may not be precisely replicable, as Gutenberg regularly adds more books. Running the scripts today could gather more books.

4.2 Getting user ratings

My proposal had suggested Goodreads as a source of user ratings. It clearly met my requirements for unique user identifiers and clear ratings: its reviews come with a numerical user ID and a rating on a 5-star scale. Both are consistently structured in the HTML. Some reviews don't have ratings, but most do, and it's easy to skip those that don't.

4.2.1 Matching data

My final requirement was metadata to link Goodreads ratings back to Gutenberg texts. For this I turned to the Goodreads API, and used its search endpoint to find each book I'd downloaded from Project Gutenberg (notebooks/3_goodreads_data.ipynb).

The API didn't give me ratings data: for that I'd need to look at each book's page on Goodreads itself. To construct URLs for these pages I'd need the Goodreads ID of each book - which I could get from the API. I also collected some additional data: publication year for matching with Gutenberg, review counts for dropping unreviewed books, and average rating for use in hybrid algorithms.

The API returned information on 14,783 of my 15,167 books. However, not all responses were helpful. Some lacked reviews - though these were easily removed using the review counts. The bigger challenge was responses that didn't match the book I'd searched for. The API sometimes returned very popular books rather than strong matches. For example, a search for RM Ballantyne's 1879 book *Philosopher Jack* returned a Harry Potter book bundle.

I tried to use publication date to remove these mismatches, but Goodreads often uses the year of a particular edition rather than the year of original publication. For example, the Victorian novelist Edward Bulwer-Lytton has many publication dates listed as 2015 and 2012, because recent editions have been added to Goodreads.

The unreliability of years meant I had to rely on author name and book title to match Goodreads to Gutenberg.

I first tried matching by Jaccard similarity and Levenshtein distance, assuming greater similarity between titles and authors on Goodreads and Gutenberg would mean a better book match. However, this didn't work well. A strict threshold for similarity led to so many false negatives my dataset became too small. A simple difference like listing an author's first names as initials produced a very low similarity score. A looser threshold led to a large number of incorrect matches based on coincidences of naming, such as Jack London's 1905 boxing novel *The Game* being identified as *The Hunger Games*.

Instead I found the most effective technique was to require at least one exactly matching word in the author field and the same for the title field. After some preprocessing (such as removing bracketed subtitles and author initials), this worked well. It wasn't 100% accurate, but manual checking showed that errors were rare enough to leave a satisfactory dataset of 8,767 texts (notebooks/4_data_clean.ipynb). The results are saved in the data folder of my repo as book_data_cut.csv.

4.2.2 Downloading the ratings

I decided to use an sqlite database to store user ratings. This was a fast, simple solution: a single table of user ID, book ID and rating (notebooks/5_sqlite_setup.ipynb).

Filling this database took more work. Using my Goodreads API data I constructed a URL for the first page of reviews for each book. Subsequent pages of reviews had to be accessed using a Javascript 'next' button. I used Selenium to run a headless version of Chrome to automate finding and clicking on this button. This was slow, as I needed to follow Goodreads rate limits, and unreliable, as I'd be disconnected after a varying number of books and pages.

To cope with the unreliability I processed books in batches, with pauses and ways to avoid or ignore errors (notebooks/6_goodreads_votes.ipynb). I continued to use notebooks at this stage so I could adjust batch sizes and starting points on the fly, and rerun parts of the script depending on how Goodreads responded.

Because this process was slow and unreliable, I only collected data on books with at least 7 ratings. Books with more reviews would give me a more viable dataset for collaborative filtering, and increase the chance of getting multiple reviews from the same users. With this done I had 5,168 books in my database, and a total of 280,360 ratings from 135,710 users.

This process for getting ratings is replicable, but the exact ratings downloaded won't be the same because they depend on where the Goodreads connection fails. I've included my full database in the data folder of my repo as book_ratings.db.

4.3 Refining the dataset

My specification aimed for a dataset similar to LitRec. At this point I had plenty of reviews (280,000 compared to LitRec's 38,591), but they were split between too many users - with an average of only 2 reviews per user. To fix this I added a **trim_ratings** function to my load_data module. This limits the dataset to books and users that reach a minimum number of reviews.

I explored different limits (described in my analysis chapter) and settled on requiring at least 10 ratings per user and 5 ratings per book. This led to a dataset of 74,940 ratings by 3,335 users across 5,066 books - an average of 22 ratings per user and 15 per book. This compared well to LitRec's average of 20 ratings per user and 10 per book.

4.3.1 Defining 'recommendation'

Having settled on binary classification in my design, I needed to decide how to split ratings on a 5-star scale into 'recommended' and 'not recommended'. My divide needed to be statistically useful and also fit the way readers used their star ratings.

Alharthi (2019) used a binary classification where '1-2 indicates a dislike and 3-5 a like'. This was based on Goodreads' own descriptions of their star ratings, where 3 stars means 'I liked it'. However, this didn't match user behaviour in the ratings I'd extracted. In my reduced dataset, 87% of the ratings were 3 stars or higher. This wouldn't produce a helpful recommender system - it would recommend almost everything, rather than helping users choose a book they'll love. It would also inflate any success metrics. I could get 87% accuracy simply by predicting 'recommend' for every book.

Instead I split my dataset at 4 stars or more. Here 55% of the dataset would be classed as 'recommend' - which is satisfactorily balanced. It also makes more sense for users, by narrowing recommendations down to more positive reviews.

I coded this as **set_threshold** in my load_data module, letting me apply this decision whenever I loaded data.

4.3.2 Selecting challenging users

Although my dataset now looked balanced on the surface, it wasn't balanced at a user level. Every user had given at least 10 ratings, but some were very predictable. 474 users only ever gave positive ratings, accounting for 12% of all ratings. Another 140 only ever gave negative ratings - another 3% of the ratings. Others weren't 100% or 0% positive, but came close.

These users would give misleading recommendation results. In a test set I could easily score 100% for a user whose ratings are all positive, but in a production environment this would lead to a system that recommended everything to them. It's unlikely these users love every book in the world - it's more likely they're good at choosing what they read, or that they only publish reviews of books they like.

To solve this, I added **set_class_proportions** to the load_data module. This function takes arguments for a minimum and maximum percentage of positive reviews per user, and removes users who don't meet the requirements.

With some experimentation I settled on keeping users with 20% to 80% positive reviews. With at least 10 reviews each, this gave enough variety to make correct predictions meaningful. These thresholds also kept me around my target dataset size, with 42,360 rows - similar to LitRec's 38,591. The user averages also compared well: 1,850 users, averaging 23 reviews each - slightly more focused than LitRec's 1,927 users averaging 20 reviews each.

My book count remained high, at 4,854, with around 9 reviews each. This was lower than LitRec's 10 reviews per book, but not problematically so. With my focus on the cold start problem, books with few reviews were a useful opportunity to test content-based approaches.

4.4 Preprocessing

As well as book text, each Gutenberg file contains background information, a licence, and notes about transcribers. I needed to remove this and leave only the text of the book. This extra information wasn't consistently structured. Depending on the file, different details were appended before and after the book text, and different strings marked the start and end of each section.

I used regex to filter out these extras. To improve accuracy I searched the results after each iteration of my functions. I identified phrases that always appeared in the licence or preamble, and whenever one was still present after preprocessing I adjusted my regexes accordingly. I also checked any results file that was 0 bytes in size.

Once I was successfully extracting the book text, I turned to more detailed preprocessing. Following my specification, this needed to be simple and consistent so I could devote more time to the recommender system itself. I created a tokenisation function that extracted lemmatised lowercase versions of each token and then removed punctuation, spaces and stopwords. This used a pretrained Spacy model.

I also used Spacy's default stopwords list. Stopwords are meant to be irrelevant to a text's topics. However, they can contribute to a text's style. For example, 'among' and 'amongst' are both location descriptions and say nothing about a book's theme, but 'amongst' has a more formal register. So these words could contribute differently to an assessment of a text's style. The same is true of punctuation: structure of sentences is crucial to style, but irrelevant to topic.

To deal with this, I kept the raw versions of the books as well as the processed versions. When I made vector representations of vocabulary, I'd use the processed versions, but when I did stylistic analysis I'd use the raw texts (minus the non-book content).

Preprocessing functions are stored in the preprocessing module. Following the code structure set by my specifications, they're used in a central runner file (run_preprocessing.py) that iterates through all the downloaded texts applying the preprocessing.

4.5 Vectorising content

Each approach to vectorisation became its own function in my vectorise module.

4.5.1 Vocabulary vectors

I started with the most basic approach - counting how many times each vocabulary term appeared. It was a useful proof of concept, but even after removing stopwords it overemphasised common words.

From there I moved to TFIDF. This counts vocabulary frequency, but weights it according to how common the word is across the corpus of texts.

TFIDF was more effective but still had 2 downsides. First, the overall vocabulary across my corpus was large - 180,000 terms - so the vector for any individual book was sparse. This could lead to dimensionality problems where the vector space distances between texts would all become close to equal. Second, the values in the vectors depended on the corpus used to generate them. As a result, I'd have to calculate new vectors each time the corpus changed. This meant each training fold needed its own set of vectors. This meant experiments ran slowly.

To deal with the dimensionality problem I introduced LSI - the language processing version of singular value decomposition. LSI uses matrix decomposition to reduce a dataset to a preset number of dimensions. LSI results can include negative numbers, so I included a min-max normalisation option so I could experiment with algorithms that couldn't use negative values.

As a final vocabulary vector option I tried non-negative matrix factorisation. This would let me avoid the negative values problem of LSI, and might produce vectors with better predictive value. However, it ran extremely slowly when dealing with large numbers of books, so I didn't use it in my final tests.

4.5.2 Style vectors

There are many ways to create vectors based on specific elements of a text's style. Alharthi (2019) trained his algorithms on '120 linguistic aspects'. I knew I wouldn't have time to explore many of these, so I started with elements he found most effective, with plans to investigate more once I'd constructed the overall framework.

One effective technique was the stylistic weighting developed by Brooke, Hammond and Hirst (2015). Their research also used Project Gutenberg texts, so it was strongly applicable to my work. They used word embeddings to weight words by how strongly they represented 6 literary styles: literary, abstract, objective, colloquial, concrete and subjective. By adding up the weightings of the different words in a text, it's possible to create a vector showing how strongly the text represents each of these styles.

They implemented their research in a Python tool called Gutentag. Unfortunately Gutentag has a convoluted design (a single 6,000-line file) and isn't available as a package. So instead of importing any of its code I used only its dictionary of stylistic weights.

As well as the 6 styles, Gutentag's weightings dictionary includes 'polarity', a sentiment analysis estimate. I included this in my style vectors, and also added word count, as this scored well in Alharthi's analysis. I used 0 to 1 scaling to avoid word count being on a totally different scale to the other values.

This gave me an 8-element style vector for each book. Unlike TFIDF I only needed to do this once, as each text's values were independent. I included this as a function in my preprocessing module, then added a script to save a style vector for each text (extract_features.py).

I initially used these style vectors on their own instead of a vocabulary vector. I then tried combining them with vocabulary vectors in 2 ways. First I combined them with 8-component LSI vocabulary vectors, creating a single 16-element vector. Then I passed vocabulary and style vectors to the hybridiser separately.

4.6 Experiment configuration

My specification required an easy way to run multiple experiments. Each experiment would try different algorithms and parameters, and I'd compare them to work out what gave the most accurate recommendations.

I created consistently structured json settings files that let me change options between experiments, plus documentation on how to write them. The files contain lists of dictionaries, each dictionary representing a single experiment and its options for vectorisation, collaborative filtering, content-based filtering and hybridisation. Each settings file could run a collection of experiments in one go.

My settings files dealt with decisions that changed frequently between experiments. Other aspects of the recommender system would change more rarely. For these I created a configuration file (also documented). This was simpler, with a single set of variables that could be shared between experiments, such as the names of results files and the variables that defined my dataset.

Dataset definition variables included recommendation threshold, minimum number of reviews per user and book, and proportions of positive reviews per user. I also added a random state to the config file, to make sure each group of experiments was replicable. Finally, the config included a boolean flag for whether to use test data or just cross-validate on the training set.

I stored config and settings files in their own directories, and loaded them through a central runner script (run_experiments.py). This meant I could start all experiments with the same command, and keep a record of all experiments I'd run.

4.6.1 Multiprocessing

I tested multiprocessing experiments, but in the end removed this code. Although multiple experiments could run at once, this slowed them down rather than speeding them up. This was because loading and vectorising data had to be repeated for each experiment. Although I could pass CPU activity to different processors, I'd have to constantly load and unload data from memory. This slowed down the overall activity.

I could have multiprocessed sections within experiments, such as the profiling and analysis of each user, but I chose not to. Some approaches to multiprocessing can't order results, so it would risk disconnecting the predicted user ratings from their correct training classifications. With more development time I could solve this, but experiments ran quickly enough that it wasn't essential.

4.7 Object-oriented design

Based on the configuration and settings, I constructed experiment objects that I could pass to different functions in the recommender system (experiment_objects.py).

The **ExperimentData** class is initialised with the data that's been loaded and the options set in the configuration file. The dataset stays the same between experiments, so this object can be initialised once and passed to all experiments being run.

As well as users and ratings, ExperimentData holds the texts of the books it references, and dictionaries mapping between Goodreads IDs, Gutenberg IDs and the indexes of vector matrixes. It has no methods - it's there solely to store and structure the data.

The main **Experiment** class is more complex. It takes ExperimentData and a selection of algorithms, and splits the data into training folds (defined as **ExperimentFold** objects). Each fold can make predictions and analyse and export results. There are different methods for each main part of the recommender process:

- making recommendations using collaborative filtering and content vectors
- using hybrid algorithms to make final predictions
- evaluating the results

4.8 Making recommendations

Because I'd be exploring lots of different approaches, I wanted to reuse code as much as possible. This meant using a similar approach for both collaborative filtering and content-based filtering.

Collaborative filtering uses a matrix of ratings, where columns are users and rows are books. When using content vectors, columns represent features (such as vocabulary terms) and each row is a vector representing one book. So books are row vectors in both approaches. I took advantage of this similarity when building my recommender system.

When each experiment or training fold starts, a function in my analyse module creates a collaborative filtering matrix that defines each book as a vector of user ratings. The vectoriser module has already constructed content-based representations of the books.

A single **analyse_user** function then runs that experiment's algorithms for both collaborative filtering and content-based filtering. Because I've structured the collaborative filtering matrix in the same way as the content vectors, I can identify all the books a user has read in the training data, then select the relevant items from both sets of vectors. Those vectors then become the X_train dataset for the prediction algorithms.

Rather than just returning boolean predictions, **analyse_user** constructs a list of tuples detailing the user id, book ids, the true class if known (such as when cross-validating on the training set), the predicted classes and the probabilities attached to those predictions. This extra data is used in the hybridisation stage.

4.8.1 Algorithm switching

Because I've structured my vectors consistently, I can use the same functions to make recommendations, regardless of whether I'm using collaborative filtering or content vectors. Taking the Scikit Learn implementations of the 5 classifiers identified in my specification, I constructed a single **run_algo** function. This unpacks settings from the Experiment object into keyword arguments for the selected algorithm. No matter which algorithm is used it returns an identically structured set of prediction probabilities.

There were 2 complications. First, KNN wouldn't work if K was larger than a user's training set. I fixed this by comparing K to each user's training data. If K was too big, I reduced it for that user, limiting it to the largest odd number that was smaller than the training set.

Second, some algorithms, such as logistic regression, require training data from at least two classes. But when working with cross-validation folds there's a chance some users end up with all their positive ratings (or, less frequently, negative ratings) in the training set. To deal with this, I made my recommender skip the algorithm and instead return a single class of predictions if all the items in the training set are of a single class.

Ironically, this decreased my prediction accuracy. As I'd limited my dataset to users with a combination of positive and negative ratings, a user whose training ratings were all positive must have their negative ratings in the validation set. I'd be predicting positive for these, even though I knew the true class was negative. If I wanted to maximise my accuracy scores above all else, I could predict negative if all training data is positive. But this would clearly be overfitting to the training data - users don't follow such artificial patterns in real life.

I included my solutions to these complications in a **predict_user** function which sits between **analyse_user** and **run_algo**. This way the Scikit Learn components in **run_algo** are fully separated from my own code, but I also avoid cluttering the user profile analysis with exceptions needed by the prediction algorithms.

4.9 Hybridisation

I planned to apply hybridisation algorithms to the predictions already made by the collaborative filtering and content-based methods. The hybrid would produce a final prediction based on these previous ones. Each hybrid would become a function in my hybridise module.

4.9.1 Rule-based hybrids

I started with 3 rule-based hybrids:

- **Confidence hybrid** looks at the prediction probabilities from collaborative filtering and content-based filtering, and selects the one that's more confident.

- **Book rating hybrid** adds a third prediction that's positive if the book's average rating in the training set is above the average rating for all books in the training set. Then the hybrid picks positive or negative based on the best of 3 from collaborative filtering, content and book rating.

- **Best of 3 hybrid** creates a separate style vector and vocabulary vector for each book, and makes a prediction for each of these as well as collaborative filtering. Its final prediction is based on the best of these 3.

To make it easy to swap between hybrids, I gave Experiment and ExperimentFold methods to apply one of these algorithms to their existing results. That method would add an extra results column containing the hybrid prediction.

4.9.2 Statistical hybrids

These hybrids used the same 5 algorithms as earlier stages, adopting the prediction probabilities of collaborative filtering and content-based filtering as features. I also included data like each book's average rating and the number of reviews by that user and for that book.

Applying these statistical hybrids was more complex than rule-based hybrids. This was because each hybrid had to be trained before it could make predictions. As the hybrids used prediction probabilities as features, they could only be trained on data that had already been used for training and prediction.

This was fine when I was just using training data. When each fold was initially trained it produced predictions for its validation set. Because I knew the true class of each item in the validation set, I could add a second cross-validation step. This split the validation set into 5 new folds to train the hybrid without any bleed-through of the true classes.

For test data this wouldn't work. If I trained the collaborative filtering and content-based algorithms on the full training set and predicted for the test set, the hybrid would have nothing to train on. It couldn't use the test set's true classes as that would clearly be training on test data. But it couldn't train on the training set because the collaborative filtering and content-based algorithms hadn't made predictions for that data - they'd only made predictions for the test set.

To solve this I restructured the way I used the data. I now start with 5-fold cross-validation in exactly the same way as when training. This produces a set of collaborative filtering and content-based predictions for the complete training set. Because I know the true classes for this data, I can then use those predictions to train a hybrid classifier.

With the hybrid classifier trained, I retrain the collaborative filtering and content-based classifiers on the entire training set. This gives me 3 fully trained classifiers to use on the test data.

This restructuring made the test process much slower as I had to vectorise and predict for 5 folds of training data as well as for the test data. However, the extra time was worthwhile as it made the system statistically sound. It also followed the same overall code structure as the rule-based algorithms, making it easy to select different hybrid options for different experiments.

4.10 Evaluation functions

Overall accuracy was easy to compute, as my Experiment objects stored the prediction results alongside the true classes. I counted the true positives, false positives, true negatives and false negatives for each experiment, and calculated an accuracy percentage for each of the 3 algorithms: collaborative filtering, content-based filtering and hybridisation.

As planned in my specification, I also used these results to calculate precision, recall and F1 measures. I added these as functions in my evaluation module. Having these additional statistics proved useful when analysing the results, as described in my analysis chapter.

My evaluation module also contains options for extracting settings and results into csv files so I can compare results between experiments.

4.11 Testing

I needed to be confident that my code worked correctly. I was joining multiple lists of vectors, ratings and books; slicing user profiles out of larger dataframes; and passing results between multiple algorithms. This meant a high risk of errors like matching the wrong book to a content vector, or the wrong set of ratings to a user.

4.11.1 Unit testing

Unit tests were the first step to reducing this risk. They cover all the main functions, including loading data, analysing users, vectorising texts, hybridising and evaluating. I used pytest for its clear framework, with additional assertion functions from Pandas and Numpy for checking equality between dataframes and arrays.

I didn't test the results of the algorithms, as these were directly from Scikit Learn and so already tested as part of that package. Instead I tested that the inputs leading to those algorithms were correct, and that the results of those algorithms were correctly fitted back into my evaluation.

4.11.2 End-to-end testing

I also wanted to check that my overall results remained consistent when I changed my code. So as well as unit tests I ran slower end-to-end tests using the initial_runs section of my settings folder. The experiments here check that all the algorithms and sets of vectorisation options work without errors.

Once each of these initial runs was complete, their results became valuable for testing. Because my experiments were replicable, running the same initial file again should lead to precisely the same results.

This was much slower than unit testing, because I had to fully train and predict the models, but it gave me full confidence whenever I made a large change.

4.11.3 Benchmarking

As well as checking my code worked, I wanted to make sure it performed well. To do this, I used Surprise, a Scikit package for making collaborative filtering recommender systems.

I initially considered extending this package with my content-based and hybrid recommenders. However, Surprise's design was so closely tied to collaborative filtering it was hard to pass in content vectors instead. Furthermore, its underlying maths was done in C, which made evaluation difficult. I concluded it would be easier to use Scikit Learn algorithms and build the framework myself.

Surprise was still useful, however, because it made collaborative filtering easy. This meant I could use it to check my collaborative filtering results reached an acceptable level of accuracy.

In notebooks/surprise.ipynb I used my load_data functions to select exactly the same data as in my main experiments. I then ran the Surprise KNN algorithm against that data to create benchmark accuracy statistics - as discussed in my analysis chapter.

5. Results and analysis

Differences in performance between vectorisers were small but consistent. On the algorithmic side, some were persistently unsuccessful, but differences were small between those that performed well. The best accuracy came from separate vectors for style and content, using:

- naive Bayes for collaborative filtering
- logistic regression on LSI-based content vectors
- logistic regression on style vectors
- an SVC with a radial basis function (RBF) kernel for hybridisation

These settings reached 65% accuracy. Their 65% F1 score shows a good balance of precision and recall.

The accuracy isn't spectacular, but it's significantly better than random and based on a difficult dataset (see "Effect of the dataset", below). The overall accuracy metric I'm using is also more challenging than a live recommender system, which would only recommend a small list of books, rather than trying to predict for all books, even when confidence is low.

The top performing combinations are shown below, with the full results in spreadsheet form on Google Sheets and in my project's results folder on Github. I discuss these results in more detail in the rest of this chapter.

Top results

Vectoriser	**Collaborative filtering**	**Content-based filtering**	**Hybrid algorithm**	**Hybrid accuracy**	**Hybrid F1**
Separate vocabulary and style vectors	Naive Bayes	Logistic regression	SVC (RBF)	64.58%	64.91%
Separate vocabulary and style vectors	Naive Bayes	KNN (k=5, distance weights)	SVC (RBF)	64.44%	64.75%
TFIDF	Naive Bayes	Logistic regression	Book rating	64.19%	65.76%
LSI (100 components, normalised)	Naive Bayes	Logistic regression	Book rating	64.19%	65.86%
Combined style and vocabulary	Naive Bayes	KNN (k=5, distance weights)	SVC (RBF)	64.15%	64.48%
Separate vocabulary and style vectors	Naive Bayes	Naive Bayes	SVC (RBF)	64.12%	64.52%
Style vectors	Naive Bayes	KNN (k=5, distance weights)	Book rating	64.11%	65.70%
LSI (100 components, normalised)	Naive Bayes	KNN (k=5, distance weights)	SVC (RBF)	64.08%	64.34%
LSI (100 components, normalised)	Naive Bayes	KNN (k=5, distance weights)	Book rating	64.03%	65.74%
LSI (100 components, normalised)	Naive Bayes	Logistic regression	SVC (RBF)	64.00%	64.27%

5.1 Experiment selection

With choices to make for vectorisation, collaborative filtering, content-based filtering and hybridisation, there were a huge number of possible experiments. I couldn't test all permutations, so instead I ran tests in batches to identify which choices were most viable.

I began by making a single experiment for each algorithm and vectoriser. These tests are shown in the initial settings folder. These showed me broadly which techniques performed effectively, and gave me enough data to create a longlist. This mixed the most effective algorithms, using different choices for collaborative filtering, content-based filtering and hybridising. This helped me see how they performed when combined with one another. I also examined some of the bigger parameter changes, such as SVC kernels.

Finally, from this longlist I identified the most effective groups of settings. I tested these more thoroughly, including checking how they reacted to all types of content vector. My best 3 combinations are in the best_3.json settings file.

5.2 Collaborative filtering effectiveness

Algorithm choice made a dramatic difference to collaborative filtering. Naive Bayes performed best, reaching 62% accuracy. This was consistent in both training and test, with little variation across cross-validation folds, which varied only from 61.5% to 63%. Its F1 score was 63%, showing a good balance of precision and recall.

SVC, KNN and logistic regression all achieved similar accuracy, from 57% to 58%. All were a notable step back from naive Bayes. Their F1 scores were particularly bad. In all 3 cases the problem was low recall - 51% for SVC, 43% for KNN and 28% for logistic regression.

This means a high rate of false negatives - predicting a negative reaction to books the user liked. By contrast, their precision was comparable to naive Bayes, and in the case of logistic regression actually higher, at 67%. So although I quickly discarded these algorithms from future tests based on accuracy, they could be useful in a system designed to return a small list of ranked recommendations.

Worst of all were decision trees, which at best reached only 50% accuracy - no better than random. The fault again lay in recall, but here it was only 2%. This wasn't a result of only recommending at extremely high confidence - only 48% of that minute number of recommendations were correct.

The main problem here was the dimensionality of the data. Decision trees rely on binary splits, but employing user ratings as features meant thousands of dimensions. This left decision trees with too many potential splits to find any useful patterns. I would expect them to perform better after dimension reduction, or when using random forests to reduce overfitting.

5.2.1 Benchmark comparison

I used the Surprise package to check my results reached a reasonable level of accuracy, as discussed in my implementation chapter. Surprise only implements one of my 5 algorithms -

KNN. It achieved almost exactly the same performance, reaching 57% accuracy on the same dataset. This gave me confidence my code was working well.

5.3 Content-based effectiveness

Content-based recommendation was harder to assess than collaborative filtering because I also had to consider vectorisation options. Differences were often small, but consistent patterns did emerge.

5.3.1 Algorithms

Initial tests showed decision trees and SVC performing badly. Neither broke past 60% with any settings I tested. However, unlike collaborative filtering, these scores weren't due to low recall - they were worse than other algorithms, but their F1 scores were similar to their accuracy.

Logistic regression, KNN and naive Bayes were more promising. All performed at similar levels, with the best results at 63% for logistic regression, 62% for naive Bayes and 61% for KNN (with K = 5, distance-based weighting and Minkowski distance).

These differences were consistent, with the same ranking happening across all cross-validation folds, as well as the test data.

5.3.2 Vectorisation

Unlike collaborative filtering, there were no existing packages to benchmark against. Instead, I tried running my recommender on vectors of random numbers between 0 and 1. These managed a surprisingly high 58% accuracy with logistic regression. It's likely this relative success was mostly based on each user's overall likelihood of giving a positive rating.

Using genuine content vectors was consistently better. The different options all had similar performance, though their tiny differences nonetheless revealed a consistent order of effectiveness.

The top performer was 100-component LSI, which reached 63% accuracy with logistic regression. 100 components is the setting recommended in Scikit Learn's documentation.

I also tried LSI with just 10 components, but this consistently reduced accuracy by roughly 1 percentage point. Its best performance was 62% for KNN with 5 neighbours. Removing normalisation from 100-component LSI had the same effect - a slight dip in performance across all tests. Its best result was 62% with logistic regression.

TFIDF vectors, style vectors and combined vocabulary and style vectors all had best results of 62%. For combined and style vectors this was with KNN, and for TFIDF this was with logistic regression.

These results suggested that my most effective content option was 100-component LSI vectors, but that style-based vectors weren't far behind. This meant it was worth exploring both at the hybrid stage.

5.3.3 Running time

Vector choice had a huge effect on experiment running time. Exact times are system-dependent, but I've included how long each of my experiments took in my results, and the differences are dramatic.

TFIDF could be very slow, especially with logistic regression, as shown in the table below. As logistic regression had been performing well, I was keen to speed it up.

LSI uses TFIDF as a starting point, then employs singular value decomposition to reduce the number of dimensions. This makes the vectorisation stage slightly slower, but the smaller number of features dramatically speeds up algorithms like logistic regression that calculate a weight for each feature. This nearly halved the running time of the full process.

Reducing LSI components increased speed, but the speed difference between 10 and 100 components was much smaller than the difference between 100 components and TFIDF. Combining LSI with style vectors also made little difference to speed.

Style vectors were faster still. As well as having only 8 elements to weight, they had the advantage of being already calculated. Because LSI used TFIDF as a starting point, it had to calculate the vectors based on the books loaded into each experiment and training fold. Style vectors, on the other hand, are completely independent, so I could save them all in advance. At their best these ran in 20% of the time of LSI vectors.

Although in terms of accuracy pure style vectors weren't the strongest option, their speed could make them useful in some production settings. It also meant combining them with other approaches added little overhead.

Table of running times

Vectoriser	Collaborative filtering	Content-based filtering	Hybrid algorithm	Time taken (seconds)
TFIDF	Naive Bayes	Logistic regression	Book rating	2,822
LSI (100 components, normalised)	Naive Bayes	Logistic regression	Book rating	1,560
Separate vocabulary and style	Naive Bayes	Logistic regression	Best of three	1,436
Combined style and vocabulary	Naive Bayes	Logistic regression	Book rating	1,399
LSI (10 components, normalised)	Naive Bayes	Logistic regression	Book rating	1,367
Style vectors	Naive Bayes	Logistic regression	Book rating	266

5.4 Hybrid effectiveness

Hybrid effectiveness depends on the prediction probabilities used as inputs. Although with some weak initial algorithms (such as decision trees) the hybridisation stage could increase accuracy dramatically, overall performance was still better when I used more effective algorithms in the collaborative filtering and content-based stages.

My implementation chapter describes my separation between rule-based and statistical hybrids. It also explains the choice between using 2 input probabilities (collaborative filtering and vocabulary-based content filtering) and 3 (the previous 2 plus style-based content filtering).

5.4.1 Using 2 input probabilities

Rule-based hybrids performed well, with the book rating hybrid being most effective. It reached 64% accuracy when hybridising the results of naive Bayes collaborative filtering and logistic regression content filtering of 100-component LSI vectors. This showed the hybrid only adding a single percentage point to the accuracy - small, but still an improvement. As rule-based systems add minimal overheads, this benefit came at little cost.

The confidence-based hybrid was also fairly effective - not as good as book ratings, but still better than all but one of the algorithm-driven statistical options. The only algorithm that compared was SVC, which was only minutely less effective than average ratings. It performed best when paired with KNN content vectors and naive Bayes collaborative filtering.

The other algorithms all fared worse, adding little to the accuracy of collaborative filtering or content-based filtering.

KNN was particularly weak, because there was no reason to suppose one result should be similar to those near it. Each classification's vector space coordinates depend only on prediction probabilities from the user analysis. This means totally different books rated by totally different users could be close to each other - and therefore influence the hybrid algorithm - just because their collaborative filtering and content-based filtering happened to have similar levels of confidence. This was essentially a random choice, and the low accuracy reflects this.

By contrast, the book rating hybrid is effective because it's a tie-breaker that uses the overall probabilities of the dataset. If a book is consistently popular with other users, it's effective to err on the side of recommending it - and vice versa.

5.4.2 Adding a third input

Results differed once I split the style and content vectors and used them as separate inputs to the hybrid. These yielded the best overall accuracy results of all the experiments. Here, SVC overtook the rules-based hybrids and was consistently the best option.

The top performer reached 65% accuracy. This was a notable increase from its inputs, which were naive Bayes collaborative filtering (62%) and logistic regression for vocabulary vectors (63%) and style vectors (61%).

Although the style vectors performed worse than the collaborative filtering and content vectors, they still made a valuable contribution. They helped the final hybrid result become more accurate than when I combined style and vocabulary into a single vector, or when I used content or style alone. This is a good indication that style can contribute something extra to recommender systems.

5.4.3 Accuracy against running time

Although SVC was more accurate than rule-based methods when given separate style and content vectors, the difference was small. The cost of this difference was a substantial slowdown - one that isn't reflected in the previous table of running times.

Rule-based hybrids don't need training. If I didn't need to train a hybrid algorithm, I could remove the entire cross-validation process from the recommender system when predicting on the test data. At the moment, cross-validation is included solely to create training data for statistical hybrids, in the form of predictions based on the training set.

Without this training, I'd expect the whole process to take less than 20% of the time. This is because the whole vectorisation, training and prediction process happens 6 times: 5 times for cross-validation and once more to retrain on the full training set. If I didn't need to cross-validate I could run the whole process only once. Depending on the data and computing power available, this could be worth the small drop in accuracy.

5.5 Adjusting the dataset

65% isn't a strikingly high accuracy score. However, in selecting challenging users I added a requirement for each user to have at least 20% negative and 20% positive ratings. This removes users who are easy to predict, for example because nearly all their reviews are positive.

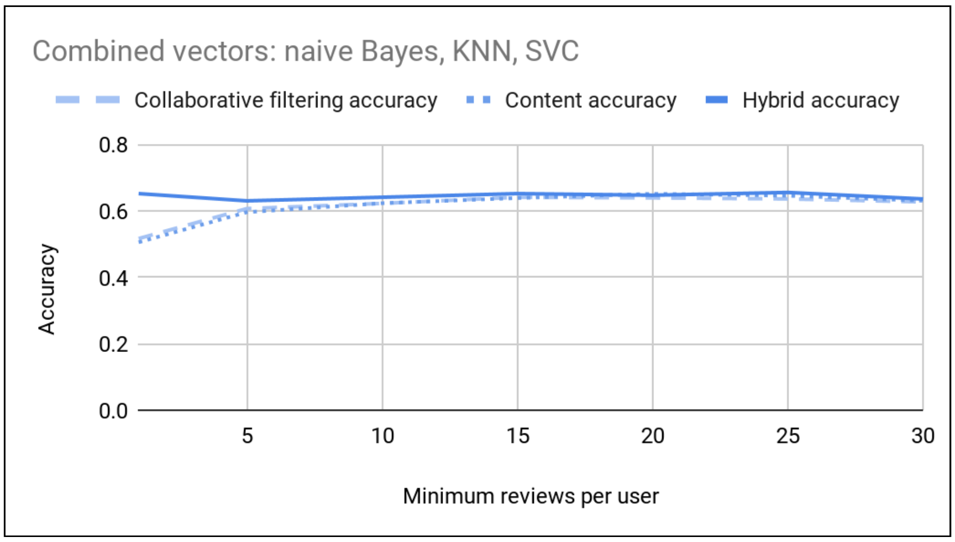
If I remove this requirement, the accuracy on my best combination of algorithms increases from 65% to 78%. I saw similar levels of improvement on other algorithm choices. However, with these easy predictions included even random content vectors can achieve 73% accuracy. This means leaving the easy predictions out is better as it gives a clearer picture of which algorithms are most effective, with bigger gaps between good and bad.

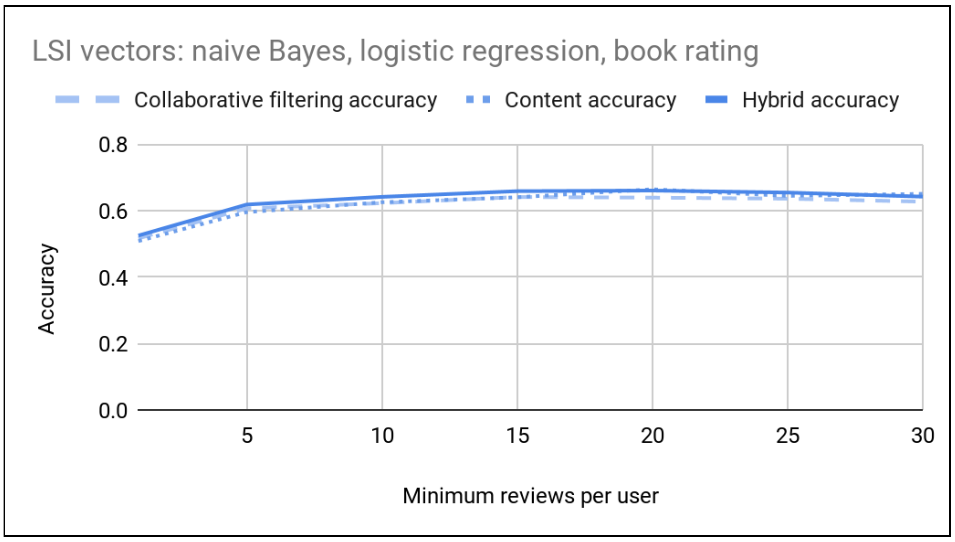
5.5.1 Minimum reviews per user

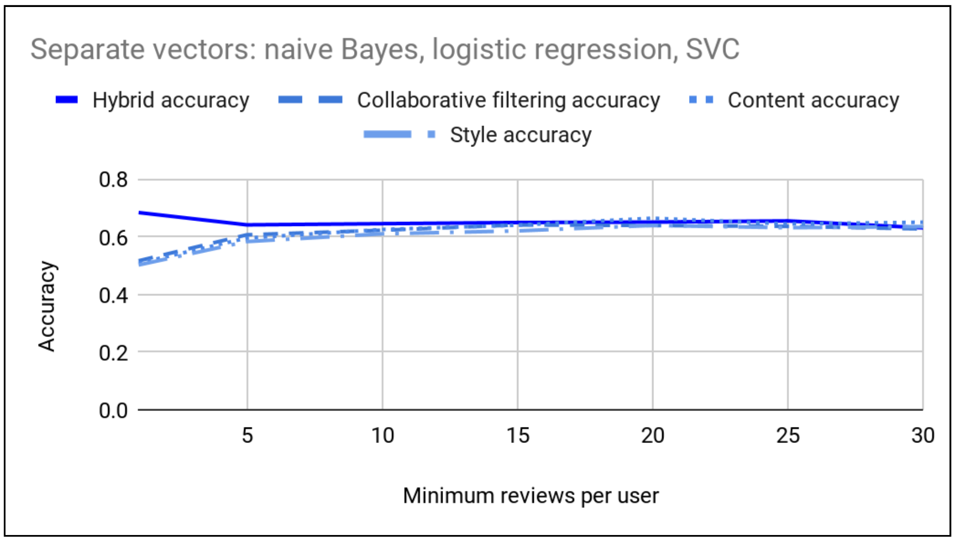
My main results require at least 5 reviews per book and at least 10 per user.

It was possible that higher minimums might make prediction easier by making sure each user profile and book profile contained more data. It might be easier to spot patterns in reviews when a user has 30 reviews rather than, say, 5. However, increasing the minimums would also reduce the overall amount of training data by leaving fewer users in the dataset. This drop could make prediction harder.

I tested different settings to see how these factors interacted, using my 3 most successful combinations of vectors and algorithms. When I varied the minimum required user ratings from 1 to 30, I saw some change:







(Full data for varying minimum number of ratings per user)

In all 3 experiments, setting no minimum reduced accuracy dramatically for collaborative filtering and content-based filtering. However, SVC hybrid algorithms managed to rescue this and in fact lead to a higher accuracy. This is likely to be due to overfitting.

A minimum of 5 ratings also reduced accuracy, again affecting collaborative-filtering and content approaches equally.

With 10 or more ratings per user, the graphs become more or less flat. There's a slight increase up to around 20 ratings (Goodreads' minimum for personalised ratings), but after that it starts to fall minutely.

This might be because the increase in user detail is balanced by the decrease in training data, or it might be because 10 or more ratings is enough to make relatively useful predictions. Acquiring more training data would help explore this further. However, for the data I had available, this confirmed that 10 was a good minimum number of ratings per user.

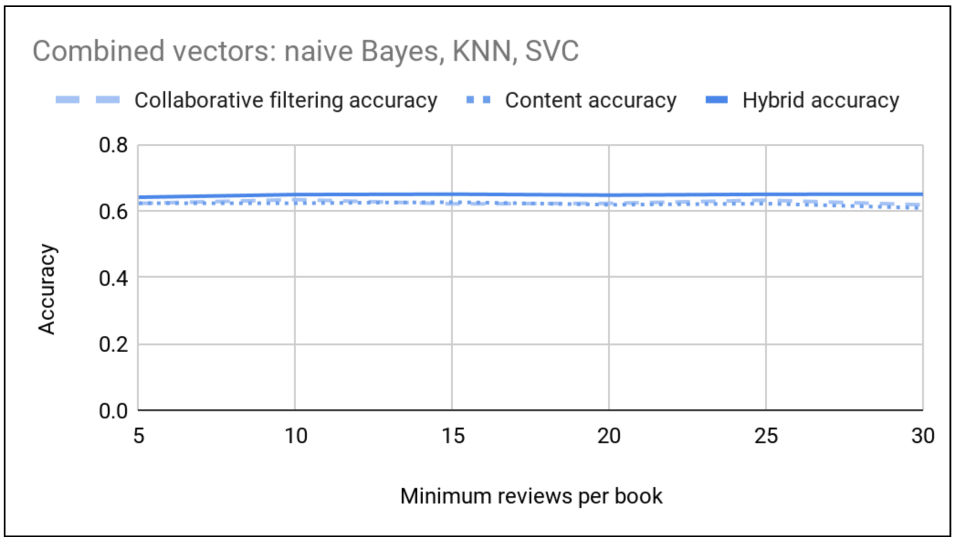
5.5.2 Minimum ratings per book

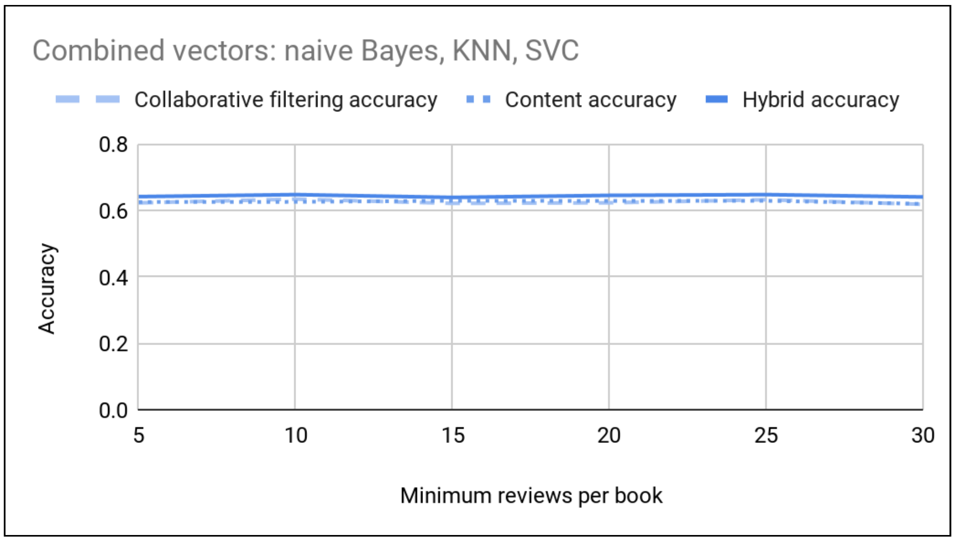
When I varied the minimum number of ratings per book, I couldn't drop the minimum below 5, as my dataset only covered books with at least 7 ratings on Goodreads.

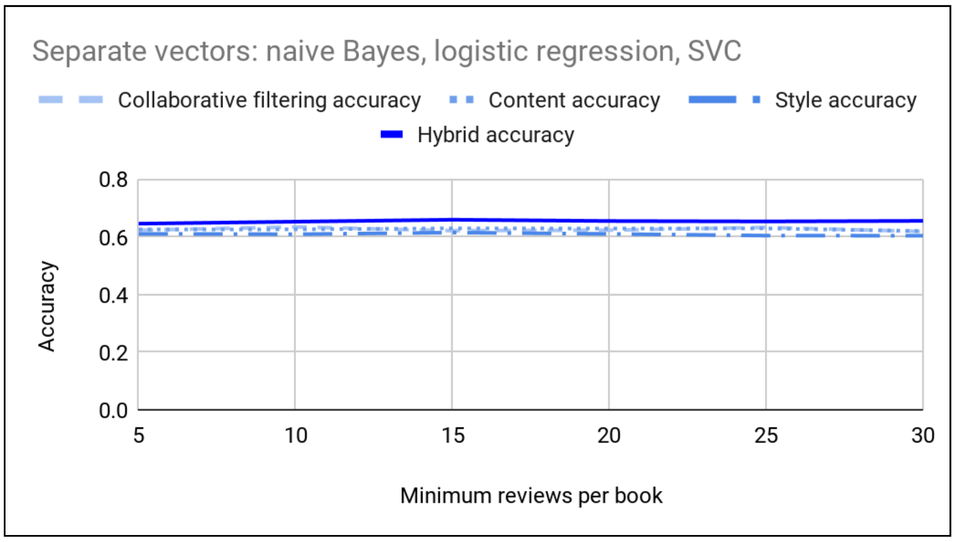
However, varying between 5 and 30 ratings per book made no notable difference to the results. This was as expected - books with very low numbers of reviews have little effect on a collaborative filtering matrix because so few people have read them. And on the content-based side, the number of reviews doesn't affect recommendations because reviews are irrelevant.

The charts on the next page show that 5 was an appropriate minimum number of reviews per book - raising it wouldn't have helped my results at all. If I had books with fewer than 5 reviews, I expect I could have removed the minimum entirely.

More importantly, this also shows that my system fully resists the cold start problem - recommendation accuracy is unaffected by books lacking reviews.







(Full data for varying minimum number of ratings per book)

6. Evaluation

I've mostly met my aims in this project, and created a strong basis for further work.

My primary aim was to make personalised recommendations. This is definitely met, with each model trained on an individual user's profile. Recommendations aren't at a spectacular level of accuracy, but they compare well to benchmarks. In addition, the consistent improvements made by the hybrid and stylistic elements show that the overall structure is effective. My recommender system has definitely benefited from going beyond collaborative filtering.

My second aim was to avoid the cold-start problem. Here I've also succeeded. The recommender system is unaffected by the number of reviews per book, as the content-based methods can still make recommendations without losing accuracy. There was less relationship than I'd expected between collaborative filtering success and review count, but this aim was still met.

The third aim - of measuring overspecialisation - I abandoned from the start as it didn't fit the way I'd assessed my data. This would be more important with a larger dataset or a live system that could assess user responses to recommendations. But for the data (and time) available in this project, removing this objective was the right thing to do.

I'm pleased with the code structure developed for this project. It provides a reliable framework for exploring different techniques and approaches. It's well-documented and highly extensible, so it would be easy to introduce new algorithms or vectorisation techniques. It's also easy to compare results and extract detailed figures.

6.1 Limitations

6.1.1 Fixed number of algorithms

The main limitation of the design is the hard-coded combination of one collaborative filtering algorithm and one content-based algorithm. Even splitting the content side into vocabulary vectors and style vectors required considerable extra work. Also, in the current design it isn't possible to use different algorithms for vocabulary and style.

If I redesigned this from scratch I'd let each experiment choose a variable number of different algorithms, each operating on a configurable type of data. For example, I'd make it possible to run 2 collaborative filtering algorithms, 2 vocabulary algorithms and 3 kinds of style algorithm, and then combine all those results with a single hybrid.

I could do this by having each Experiment object contain a list of Algorithm objects. These would each run their own vectorisation, training and prediction, and store their own results. The hybridisation function would then loop through all these objects when collecting input data. This would make it possible to explore hybridisation options in much more detail.

6.1.2 Slow vectorisation

At the moment each experiment is slowed down by having to train its own vectors. I could speed this up with large-scale memoisation.

Currently experiments abandon their vectors after completion. Instead, I could save the vectors to memory or disk along with a unique identifier based on the settings and data subset used to create them. Each subsequent experiment could check the identifiers to see if vectors for its setup have already been calculated. If so, loading them will be much faster than recalculating them.

This would save large amounts of CPU time by taking advantage of the replicable nature of the experiments and their fixable random states.

6.2 Potential extensions

The biggest area to explore further is natural language processing (NLP). Although I envisaged this project as a chance to explore NLP techniques for identifying a book's style, it didn't turn out that way. Instead I spent most of my time constructing the recommendation framework and exploring algorithm choices. This was necessary, because jumping straight to NLP without a clear framework for experimentation and analysis would have produced meaningless results.

However, there's clearly much more to explore, particularly in the field of vectorisation. I'd be keen to try word embedding using models like Bert and Doc2Vec, as these have performed well in other text classification tasks.

The current algorithms are a good starting point, but they do little to capture the interaction between features. For example, in hybridisation it's plausible that number of reviews affects whether collaborative filtering or content vectors should be more influential. The current algorithms can't capture this, which might explain why rules-based approaches perform just as well. Techniques like factorisation machines could be productive here, as they can capture the interaction between features. The polylearn and FastFM packages implement factorisation machines using a similar framework to Scikit Learn, so could fit into my recommender system.

I could also spend more time on text preprocessing. This might include customising stopwords and replacing numbers and names, as well as more carefully stripping out the forewords and afterwords from each book. At the moment some leftovers remain, such as lists of illustrations.

Finally, I could extract a wider range of stylistic features. Although separate style vectors are already proving useful in my results, the elements I use are simple. The style weightings adapted from Gutentag are vocabulary-driven, so aren't independent of the vocabulary vectors they're designed to support.

It would be useful to look more at areas that aren't affected by vocabulary. I've started this by using book length, so the logical next steps would be chapter, paragraph and sentence lengths. There's also the potential to look at amount of dialogue, and variety of characters, places and other entities.

7. Bibliography

C Aggarwal, 'Recommender systems: the textbook', 1st edition, 2016

H Alharthi, 'Natural Language Processing for Book Recommender Systems', 2019, PhD thesis, University of Ottawa

H Alharthi, D Inkpen, S Szpakowicz, 'A survey of book recommender systems', 2018, Journal of Intelligent Information Systems, 51(1), 139-160

X Amatriain, J Pujol, 'Data mining methods for recommender systems' (chapter 7 of 'Recommender Systems Handbook', 2nd edition, 2015, edited by F Ricci, L Rokach and B Shapira)

J Brooke, A Hammond, G Hirst, 'GutenTag: an NLP-driven tool for digital humanities research in the Project Gutenberg corpus', June 2015, proceedings of the 4th Workshop on Computational Linguistics for Literature

R Burke, 'Hybrid recommender systems: survey and experiments', 2002, User Modeling and User-Adapted Interaction, 12(4), 331-370

RJ Mooney, L Roy, 'Content-based book recommending using learning for text categorization', 2000, proceedings of the 5th ACM Conference on Digital Libraries, 195-204

X Ning, C Desrosiers, G Karypis, 'A comprehensive survey of neighborhood-based recommendation methods' (chapter 2 of 'Recommender Systems Handbook', 2nd edition, 2015, edited by F Ricci, L Rokach and B Shapira)

MS Pera, YK Ng, 'Automating readers' advisory to make book recommendations for K-12 readers', October 2014, proceedings of the 8th ACM Conference on Recommender systems, 9-16

P Vaz, D Martins de Matos, B Martins, P Calado, 'Improving a Hybrid Literary Book Recommendation System through Author Ranking', 2012, proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries

Zajac, Zygmunt, 'Goodbooks-10k: a new dataset for book recommendations', 2017, FastML

H Zhang, TWS Chow, QMJ Wu, 'Organizing books and authors by multilayer SOM', 2016, IEEE Transactions on Neural Networks and Learning Systems, 27(12), 2537-2550

CN Ziegler, SM McNee, JA Konstan, G Lausen, 'Improving recommendation lists through topic diversification', May 2005, proceedings of the 14th International Conference on the World Wide Web