Sommelier: A Recommender System

By Peter Chamberlin
May 4, 2013

7500 words.
BSc Information Systems and Management Project Report,
Birkbeck College, University of London.

Abstract

In this project I implement a recommender system as a web service for wine recommendations. As a starting point the I use the wines, authors and ratings data from wine website Decanter.com, and implement a RESTful web API for accessing wine and author information, augmenting the results with recommendations for other wines or authors. The system is written in the Python language, using a MySQL database. The API is built using the Flask framework. Implementing a primitive collaborative filtering system but finding my source data to be exceptionally sparse for recommendation, I explore and implement imputation techniques using matrix factorization and singular value decomposition in an attempt to boost the system's ability to make good recommendations. I find that both matrix factorization and singular value decomposition improve the recommendation quality of the system, but I am unable to satisfactorily test these systems.

Contents

1 Introduction

1.1 Online Recommender Systems

Since their origin in the mid-1990s with systems such as Tapestry [17] and GroupLens [38], recommender systems have become ubiquitous on the World Wide Web, being employed by some of the worlds largest online businesses as core parts of their offering.

The growth of the Web has given companies the ability to gather unprecedented amounts of data about their users' preferences, both explicitly collected and inferred from their behaviour, while at the same time enabling them to reach users for less cost more often than ever before.

Amazon's system of product recommendations using item-to-item collaborative filtering is regarded as a "killer feature"[15], and is one of the defining features of the Amazon website. The importance of recommendations to Amazon is reflected in their stated mission, "to delight our customers by allowing them to serendipitously discover great products"[15].

Another company which, like Amazon, is synonymous with recommender systems is Netflix, an "Internet television network" [24]. In October 2006 Netflix launched "The Netflix Prize", a competition with a $1,000,000 Grand Prize on offer for any team which could beat their own Cinematch recommender system by "at least 10%" accuracy over a fixed set of data [26]. In 2009 the prize was awarded to the BellKor's Pragmatic Chaos team, who had improved on Netflix's own system by 10.06% [25].

1.2 Aims and Objectives

My interest in recommender systems is founded in their variety and ubiquity. It occurred to me that I encounter, and am the subject of, dozens of these systems in my everyday life. Whether it's Twitter or Facebook recommending interesting interesting people or content to me, Amazon recommending me a book or film, or even a supermarket targeting special offers to me, I interact with recommender systems all the time. I am fascinated both by how these systems work theoretically and by how they are implementated in practice.

I have kindly been permitted to freely use data from Decanter.com's wine reviews database [10] for my project. The sphere of wine recommendations is particularly interesting; wine is at first glance a narrow subject, but it is a nuanced one. Among oenophiles there is an emphasis on personal taste and a strong tradition of rating and grading.

In this project I aim to build a recommender system based on Decanter.com's wine database, identifying and overcoming the challenges associated with the implementation of a real-world recommender system.

Recommendation quality is important to any recommender system, and I intend to focus strongly on it, but I aim to produce a system satisfactory in both its performance in delivering recommendations and its performance as a web service, with a robust and elegant implementation in code.

My aim is not to build any kind of graphical interface for the system, but instead to provide a machine readable service API. Where necessary I will also produce batch scripts and command line tools for preparing and manipulating data.

2 Literature Review and Context

2.1 Recommender Systems

Although the term *recommender system* was not coined until 1997 by Resnick and Varian (Resnick and Varian, 1997 [39]), the Tapestry system of 1992 (Goldberg et al., 1992 [17]) is widely recognised as the first of the kind (Su and Khoshoftaar, 2009 [45]). The creators of Tapestry coined the term *collaborative filtering* to describe their method of recommendation, which is based on the principle that if two users rate a number of the same items in a similar manner, then it can be assumed that they will rate other new items similarly (Su and Koshgoftaar, 2009[45]).

Su and Khoshgoftaar (2009 [45]) point out that although collaborative filtering has been widely adopted as a general term to describe systems making recommendations, many such systems do not explicitly collaborate with users or exclusively filter items for recommendation. In fact the term recommender system itself was coined by Resnick and Varian in response to the inadequacy of collaborative filtering for describing the plurality of techniques that were beginning to become associated with it. Recommender system is intentionally a broader term, describing any system that "assists and augments [the] natural social process" of recommendation (Resnick and Varian, 1997 [39]).

In his 2002 survey of the state of the art in recommender systems, Robin Burke presents five categories of recommender (Burke, 2002 [5]). I have reproduced his table of recommendation techniques in Table 1. Burke presents five main categories of filtering technique: collaborative, *content-based*, *demographic*, *utility-based* and *knowledge-based* (Burke, 2002 [5]).

These five kinds of system are classified using three properties: *background data*, *input data* and *process* (Burke, 2002 [5]). Background data is that which exists before and independant of the recommendation, such as previous stated preferences of a group of users U for a set of items I. Input data is that which is considered by the system when making recommendations, such as the ratings of an individual u of items in I. Process is the method by which recommendations are arrived at by application of the input data and the background data (Burke, 2002 [5]). These three aspects provide a good lens through which to compare the different approaches.

2.1.1 Collaborative Filtering

Collaborative filtering is "the technique of using peer opinions to predict the interest of others" (Claypool et al., 1999 [7]), and uses the ratings of a set of users U over a set of items I as background data, and the ratings of each individual user u of items in I as input data. The process of recommendation is to identify similar users to u in U, and then to infer their preferences for items in I based on the preferences of those similar users (Burke, 2002 [5]).

In 2002 Burke described collaborative filtering as the most widely used and mature of these types (Burke, 2002 [5]), citing GroupLens (Resnick, 1994 [38]) and Tapestry (Goldberg, 1992 [17]) as important examples of such systems. From my observations of more recent literature that remains the case. Collaborative filtering is still certainly among the most widely used of these techniques, with Su and Khoshgoftaar describing a large number of collaborative filtering-based systems in their 2009 survey (Su and Koshgoftaar, 2009 [45]).

Table 1: Recommendation Techniques, reproduced from Burke, 2002 [5]

Technique	Backgroud	Input	Process
Collaborative	Ratings from U of items in I.	Ratings from u of items in I.	Identify users in U similar to u, and extrapolate from their ratings of i.
Content-based	Features of items in I.	u's ratings of items in I.	Generate a classifier that fits u's rating behaviour and use it on i.
Demographic	Demographic information about U and their ratings of items in I.	Demographic information about u.	Identify users that are demographically similar to u, and extrapolate from their ratings of i.
Utility-based	Features of items in I.	A utility function over items in I that describes u's preferences.	Apply the function to the items and determine i's rank.
Knowledge-based	Features of items in I. Knowledge of how these items meet a user's needs.	A description of u's needs or interests.	Infer a match between i and u's need.

Even in its most basic form there are many potential methods for measuring similarity between users in a collaborative filtering system. The most simple are such distance metrics such as Manhattan distance or Euclidean distance (Segaran, 2007 Ch.2 [41]). One of the most commonly used and relatively simple measures of similarity is the Pearson correlation coefficient, which is even used in quite advanced systems (Segaran, 2007 Ch.2 [41]).

There are a number of issues associated with the application of pure collaborative filtering, however, which hamper its application in many instances:

- Early rater problem, whereby an item entering the system with no ratings has no chance of being recommended (Claypool et al., 1999 [7]).
- Sparsity problem. Where there is a high ratio of items to ratings it may be difficult to find items which have been rated by enough users to use as the basis for recommendation (Claypool et al., 1999 [7])(Su and Koshgoftaar, 2009 [45]).
- Grey sheep, which are users who neither conform nor disagree with any other group in a significant way, making it very difficult to recommend items for them (Claypool et al., 1999 [7])(Su and Koshgoftaar, 2009 [45]).
- Synonymy, whereby identical items have different names or entries. In this case the collaborative filtering systems are unable to detect that they are the same item (Su and Koshgoftaar, 2009 [45]).

- Vulnerability to shilling, where a user may submit a very large number of ratings to manipulate the recommendation of items (Su and Koshgoftaar, 2009 [45]).

Much of the variation between collaborative filtering techniques described by Su and Khoshgoftaar (2009 [45]) can be attributed to efforts by system developers to minimise the impact of one or more of these problems by introducing auxiliary methods.

2.1.2 Content-based

In content-based filtering systems the features of items in I form the background data, and the user u's ratings serve as the input data. The process of recommendation depends on building a classifier that can predict u's rating behaviour in respect of an item i based on u's previous ratings of items in I (Burke, 2002 [5]).

Content-based, like collaborative filtering, builds up a long term profile of a user's interests and preferences (Burke, 2002 [5]). Researchers have developed systems which successfully combined content-based and collaborative filtering systems to produce better recommendations. Claypool et al. describe such a system which predicts ratings based on a weighted average of results from each system, with the weighting varying per user in order to achieve optimal results (Claypool et al., 1999 [7]).

2.1.3 Demographic

Demographic recommender systems use demographic information about users U and their ratings in I as background data, with demographic information about u as the input data. The recommendation process depends on matching u with other demographically similar users in U (Burke, 2002 [5]).

2.1.4 Utility-based

Utility-based systems use features of items in I as their background data, and depend on a utility function representing u's preferences in order to arrive at recommendations. The process is the application of the function for u to the items I (Burke, 2002 [5]).

2.1.5 Knowledge-based

Knowledge-based systems, like utility-based systems, draw on the features of items in I as their background data. As input data they require information about u's needs. The process is to infer a need for one or more items in I (Burke, 2002 [5]).

Knowledge-based systems can also benefit from being combined with collaborative filtering systems, in a way that may improve an under-performing knowledge-based system, but without realising pure collaborative filtering's ability to identify niche groups (Burke, 1999 [3]).

2.2 Recommending Wines.

Recommender systems for wines are not a new idea, being typical of the kind of item many systems are designed to recommend. Burke developed the VintageExchange FindMe recommender system in 1999 (Burke, 1999 [2]), and there is at least one patent pending with the WIPO for a wine recommender system as an aid to salespeople or waiting staff (Ward et al., 2012 [48]).

Burke's FindMe, a knowledge-based recommender system, "required approximately one person-month of knowledge engineering effort" (Burke, 1999b [3]) in order to perform well. Such knowledge-based systems are required to recognise the importance of given product features, and so require a great deal of priming (Burke, 1999b [3]).

Another wine recommender system is the Tetherless World Wine Agent (TWWA) (Patton, 2010 [28]). The TWWA project is primarily concerned with knowledge representaion and the Semantic Web, presenting a common and collaborative ontology for wine with which users can share wine recommendations across their social networks (TWWA, 2013 [46]). The system does not automatically tailor recommendations to users, although this is stated as a target for future work (TWWA, 2013 [46]).

2.3 Evaluating Recommender Systems

Shani and Gunawardana (2011 [42]) describe several approaches to the evaluation of recommender systems, including different experimental settings and a number of different statistical methods. They highlight the different aspects of recommender systems that can be evaluated, including prediction accuracy; how accurately does the system predict ratings or preferences, item-space coverage; what proportion of the items in the system are recommendable, and user-space coverage; the proportion of users or interactions the system can provide recommendations for (Shani and Gunawardana, 2011 [42]).

In terms of prediction accuracy mean absolute error (MAE), and root mean squared error (RMSE) are the most popular measures (Shani and Gunawardana, 2011 [42]), which Su and Khoshgoftaar (2009 [45]) call "predictive accuracy metrics".

There is some criticism of accuract metrics applied to recommender systems. McNee, Riedl and Konstan (2006 [20]) cite Amazon.com as a case in point, where on the page for a book by a given author you will find recommendations for other of the same author's books. They argue that this is not interesting for the user, and that there is a need for recommender systems developers to look beyond simply the ratability of systems, concluding, "It is now time to also study recommenders from a user-centric perspective to make them not only accurate and helpful, but also a pleasure to use" (McNee et al., 2006 [20]).

2.4 Web Services and Service Oriented Architecture

Service oriented architecture (SOA) is an increasingly widely used paradigm in enterprise applications, enabling such benefits as modularity, distribution and reuse of services (Sheikh et al., 2011 [43]). SOA systems are loosely coupled, and lend themselves to supporting heterogeneous applications (Benatallah and Nezhad, 2008 [1]), such as would be the case for a service providing data for a variety of websites

or mobile applications. Web SOAs encompass traditional WS-* web services approaches, such as SOAP and XML, as well as the more recently emerging RESTful (REpresentational State Transfer) approach, which takes advantage of the existing communications protocol of HTTP, and associated protocols such as SSL (Benatallah and Nezhad, 2008 [1]). RESTful and WS-* have their benefits and weaknesses, and the choice of either would be dependant on the needs of any given service.

Pautasso et al. (2008 [29]) critically compare WS-* and RESTful services, concluding that although REST is limited by the contraints of the HTTP protocol, its restictive nature and architecture are also a strength, "choosing REST removes the need for making a series of further architectural decisions related to the various layers of the WS-* stack and makes such complexity appear as superfluous" (Pautasso et al., 2008 [29]).

2.5 Implications

There is extensive literature on recommender systems, so much so that it is very difficult to assess the pros and cons of any given approach in respect of my project. One theme that is very strong in recommender systems is that of collaborative filtering, which is central to the majority of systems I have looked at. In many cases collaborative filtering is used in conjunction with other methods that boost or tune the results. For that reason I intend to pursue collaborative filtering in my system, and will incorporate strategies to improve recommendations iteratively.

With regard to the web application component of my system, it seems clear that a RESTful web service would be suitable. My system will have a small number of API endpoints, so I will be able to reap the benefits the simplicity of the RESTful web stack without suffering the difficulties associated with its limited customisability.

3 Method

3.1 Methodology

Given the exploratory nature of this project I elected to take an incremental and iterative approach, developing small parts of the system at any time (increments), and iterating over those parts with improvements as necessity dicated and time allowed. This approach is influenced by that laid out by Cockburn (2008 [8]).

3.1.1 Phases

The main phases of development will be:

1. Clean up and migrate data
2. Create initial API app
3. Connect API with database
4. Implement routes for API access to wines and authors
5. Augment API routes for wines and authors with recommendations
6. Iterate on recommendation methods, evaluating and improving quality

3.1.2 Minimum Viable Product

It was clear that I would be doing a large amount of experimental programming, and given my lack of prior experience in the problem domain I felt it inappropriate to attempt to quantify my expectations for the system in terms of detailed requirements. Nevertheless there were very clear minimum objectives for the system, without which it would not be possible to claim any degree of success.

The system should at least:

- Provide an HTTP API for accessing wine and user information from the Decanter.com tastings database.
- Augment the API results for wines and authors with appropriate recommendations of other similar or interesting wines and author.
- Provide API results suitable for machine interpretation by web or mobile applications.
- Provide a mechanism by which to evaluate recommendation quality.

These requirements in the least should be fulfilled by the system. With this having been done the focus of the project will be on maximising the quality of recommendations.

3.2 Technologies and Tools

3.2.1 Python

As the main progamming language for my project I chose to use Python. There were several candidate languages, not least Java, but I decided on Python because it has a number of attributes which lent themselves particularly to this project:

- Extensive mathematical and scientific libraries, such as NumPy (NumPy, 2013 [27]) and SciPy (SciPy, 2013 [40]).
- Extensive detailed documentation (Python Documentation, 2013 [33]).
- Widely used in web development, such as by Google and YouTube (Python Quotes, 2013 [35]).
- Interactive interpreter, allowing command line interaction and supporting scripting on Unix-like systems (Python, Interpreter, 2013 [32]).

One deciding factor was that my first enquiry into recommender systems was reading Segaran's code examples in Chapter 2 of Collective Intelligence (2007, Ch.2 [41]), where the language he uses for his code examples is Python.

In addition to its suitability for tasks around recommender systems, Python has a solid heritage of web application frameworks, such as Django (Django Project, 2013 [11]) and Flask (Flask, 2013 [12]). Django is a fully featured website building framework, and as such carries many features unnecessary for my project, whereas Flask, a "micro-framework" (Flask, 2013 [12]), appeared to be more lightweight and simple to implement. Therefore I chose to implement my API using Flask.

For the most part I considered that my system would suit the stateless, non-persistent nature of a Python web application. The only concern in this regard would be that I would need to recreate objects in memory from scratch with each request rather than persist them as I might using another language, such as using Java with the JPA (Java EE 6 Tutorial, The JPA [18]). It was reasonable to suppose that in generating recommendations I would potentially be creating large objects in memory, and that there may be a performance deficit incurred by having to rebuild such objects on a per request basis. I resolved that should the lack of persistence prove problematic down the line I would be able to use a persistence mechanism such as Memcached (Python-Memcached, 2013 [34]) to serve this purpose, and found that there is wide support for such a solution using Python and Flask (Flask Documentation: Caching, 2013 [14]).

3.2.2 MySQL

Originally I had envisaged a system backed by a PostgreSQL (PostgreSQL, 2013 [30]) RDBMS, but having received the Decanter.com data as a MySQL (MySQL, 2013 [23]) database it did not seem, comparing the two systems, that there would be any significant benefit migrating the data to PostgreSQL. Both are widely used in production, and have similar feature sets. For a short time I considered using a NoSQL database such as MongoDB (MongoDB, 2013 [21]) for my project, but decided against such a solution, recognising that such document-oriented systems are not ideal when joining between tables in the way that I would need to for

my wines and tasting notes. It seemed that an RDBMS was ideally suited to the purpose, and there was no reason why that shouldn't be MySQL.

3.2.3 GitHub

Given the iterative nature of my development process I envisaged a need to be able to easily version my source code, possibly running several different versions at once, with the ability to revert changes back to any previous state. I also wanted a remote backup of my system in case of problems with my own development computer. In order to do be able to do these things I chose to store my code as a project in GitHub (GitHub, 2013 [16]), which is a web service providing Git version control. I chose to use GitHub for my notes and project files also, so that my entire project was stored, versioned and backed up together.

3.3 Approach to Recommendations

Wines are considered to be highly specific items. Even though properties such as region or grape variety might be the same between two wines, they may be significantly different to one other. For this reason it is very difficult to conceive of a recommender system for wines based on content attributes that would not be stuck recommending items in the near neighbourhood of their subject. In such a system a user that likes one Chianti might simply be recommended more Chiantis, because they all have the same region and grape variety, roughly the same alcohol, similar residual sugar etc..

In that respect wines are very much like movies. Just because a user likes one comedy film, it doesn't mean they will like all comedy films. What may give wines the edge over movies is the lexicon of their tasting notes. It would be interesting to see if wine tasting notes could be used to aid interesting recommendations. Primarily though, I will seek to apply collaborative-filtering techniques to the recommendation of wines in the Decanter.com database, pursuing data imputation techniques to predict ratings for users on wines.

4 System

4.1 Development Environment

I have developed my project to run on a Linux platform, and have used version 12.04 of the Ubuntu operating system during the system's development (Ubuntu, 2013 [47]). Interaction with the system is via the standard terminal. Appendix A details installation instructions for the necessary software packages and libraries for the project. The instructions assume the user is using an Internet-connected computer running a recent version of the Ubuntu operating system (v12.04 or above).

System-level software requirements include:

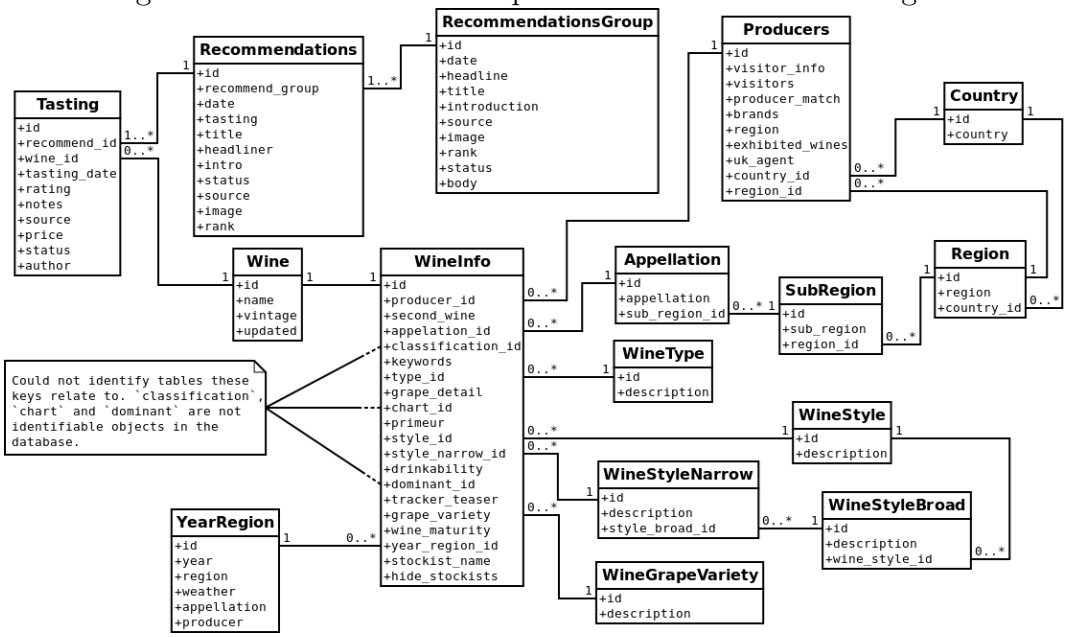
- MySQL v5.5, Python v2.7, Virtualenv, NumPy, SciPy

Other dependencies are installed locally to the project.

4.2 Data Migration

4.2.1 The Original Decanter Wines Database

Figure 1: Wines Database Represented as UML Class Diagram



The data source I have used for my project is the wines database belonging to Decanter.com [10]. The database contains nearly 40,000 professional ratings and tasting notes for wines from as far back as 1986, featuring vintages as far back as 1917.

I had hoped that the data would be fairly usable as it was received, as the data is currently used on the Decanter.com website, but on examination it was clear that there would be quite a lot of work to do in order to repurpose the data for my system.

I have modelled the database as a class diagram (Figure 1). This diagram represents the 16 tables in the original database, the columns present in those tables, and the relationships between them.

One thing immediately apparent about the database is that there are circular relationship between some of the tables. For example, the table WineInfo contains foreign key fields for both WineStyleNarrow and WineStyle, but it is the case that a association with WineStyleNarrow already encompasses an association with WineStyle, via WineStyleNarrow's association with WineStyleBroad, which associates with WineStyle. Duplicating the association in this way makes the database difficult to maintain as there is a dependency between the two foreign keys style_id and style_narrow_id on the WineInfo table which means that both must be updated any time that one is. There is a similar situation between the keys appellation_id and producer_id in the same table.

In addition to unnecessary duplication of associations in the WineInfo table there are foreign keys for missing, or unidentifiable, tables: classification_id, chart_id and dominant_id. It is likely that these keys relate to tables removed from the database at one time or other.

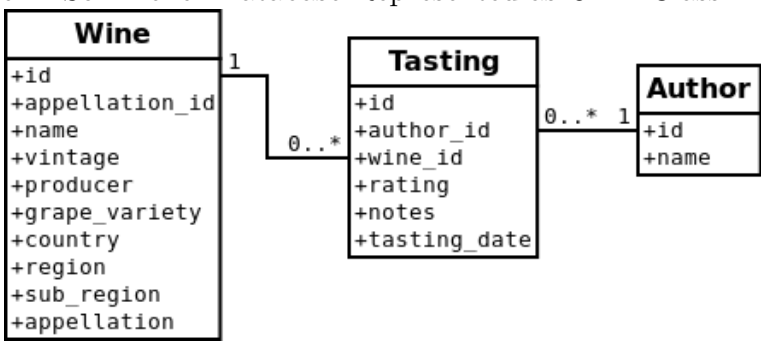
The WineInfo table is a mixture of foreign keys joining to very small tables, such as type_id joining to WineType, where WineType is a table holding only a single non-key field. This approach, of factoring out any non-integer fields to separate tables, is inconsistent with the fact that the same table also has several text fields, including second_wine and tracker_teaser. second_wine only holds data in 450 of the 38762 entries in the table, and is an empty string by default in all others.

The problems with the database structure are consistent with the fact that the database has been developed over a long period of time in an ad hoc fashion. There have probably been many different developers, and the database, being many years old, has probably supported many incarnations of Decanter's website.

Despite the problems with the database I considered there to be a great deal of useful and interesting information in the database, with it to contain usable ratings and/or tastings for over 33,000 wines, but decided that it would be best to port it to a much more simple schema, and to strip out as much redundant data as possible, for the purposes of my project.

4.2.2 The Cleaned-up Sommelier Development Database

Figure 2: Sommelier Database Represented as UML Class Diagram



For the new Sommelier database (Figure 2), I decided to minimise the complexity of the data, denormalizing the tables to make querying it as simple as possible. This would make updates to the data more complex, as any update to a field shared among many wines, for example a producer's name, would need to be replicated across a high number of records. As I did not intend to perform any writes on the data as

part of my project I felt able to overlook this shortcoming of the design. If this system were to be applied in the real world, then there may be a need to refactor the database so that updates may be made more easily.

As can be seen in Figure 2, I disregarded entirely much of the data from the original database. In many cases this was because I did not feel the data would be beneficial to my system. The Recommendations and RecommendationsGroup tables, for example, may have been useful, but the data in the tables was often incomplete and I did not reckon them to enrich the tasting notes enough to warrant migration. The fact that belonging to a certain group of recommendations may be used to infer similarities between wines seemed a tenuous reason to retain data in my development database.

The tables WineStyleNarrow and WineStyleBroad contained generic text descriptions for wine, such as "rich and creamy" and "crisp and tangy". I initially considered this to have potential for migration into tag data which I could reuse as part of my filtering. Unfortunately less than 6435 of the records in WineInfo had non-null values for their 'style_narrow_id' field, and only 3397 of these had corresponding records in the Tasting table. This figure was only around 10% of the number of wines I expected the Sommelier database to contain so I decided that the WineStyleNarrow and WineStyleBroad tables were not worth migrating.

The WineType table was ignored because no wines corresponded to it, there were values for 'type_id' in some WineInfo records, but none of those values corresponded with values in the 'id' field in WineType.

Whatever the situation with the data in other tables, it was clear that the Tastings table would need to be at the centre of my system, as it is where user, item and rating are associated. Those are the core data points for most recommender systems, so I resolved that I would migrate all tastings which had both their 'notes' and 'rating' field populated, along with the associated authors and wines, to a new database using my simplified schema.

4.2.3 The Author Problem

The biggest shortcoming of the dataset is that the author of a tasting note is often not recorded. The number of tastings with known authors is only 1411, with there being only 18 named authors on the system. There are 25812 tastings with no author associated. I had reckoned on there being a much higher number of wine tastings with identifiable authors, given that each note and rating was always by a particular person. It appeared that this information had either not been entered, or had been lost.

Table 2 shows the distribution of tastings amongst authors, only 5 of which have tasted and rated more than 100 wines in the database. The table also shows how many of the wines tasted by each author have also been tasted and rated by other authors. Table 4 shows the number of wines tasted in common for each of the authors (authors who have not tasted any wines in common with another author are omitted). It is clear that the data is extremely sparse. When migrating the data I was careful to ensure that I maintained all author to tasting associations, but rather than an oversight during migration, this shortcoming is inherent in the data.

Measuring sparsity as the percentage of empty values in a matrix $\mathbf{M}$ of size $|A| \times |W|$, where A is the vector of authors and W is the vector of wines, we find that the database is 94% sparse (Appendix B.2). This is a worryingly sparse

Table 2: Number of wines rated by each author

Author	Wines tasted	Wines also tasted by other authors
Amy Wislocki	28	1
Andrew Jefford	105	38
Beverley Blanning MW	13	-
Carolyn Holmes	1	-
Christelle Guibert	119	9
Clive Coates MW	6	-
David Peppercorn	44	-
Gerald D Boyd	7	-
Harriet Waugh	250	23
James Lawther MW	226	21
John Radford	2	-
Josephine Butchart	24	1
Norm Roby	4	-
Rosemary George MW	6	-
Serena Sutcliffe	31	15
Stephen Brook	19	3
Steven Spurrier	497	53

dataset for collaborative filtering. Su (2006 [44]) reports a "fast degradation of performance" for the collaborative filtering algorithms they tested when the rate of sparsity exceeded 90%. In addition there is the problem that ratings are not evenly distributed among authors, with a very small proportion of the authors accounting for the majority of ratings.

In a few cases there are tastings with no author associated where an author's initials or full name are recorded within the text of a tasting note, but unfortunately extracting and making use of these initials has been impractical given the time constraints of this project.

4.3 Experimentation

4.3.1 Recommending Wines With Python

In Chapter 2 of Collective Intelligence (Segaran, 2007 [41]), Segaran details basic methods for user- and item-based collaborative filtering. Following the guidelines from this chapter I recreated Segaran's recommendation methods and applied them to my dataset, utilizing the Python command line interpreter (See code in Appendix D.1).

Table 4 shows the Pearson correlation between the authors. In cases where there are fewer than 3 items rated in common it is not possible for Pearson correlation to produce a useful result, and for such relationships my code returns a value of 0.0. Table 4 shows that Pearson correlation was only able to produce similarity scores among five of the authors. For 18 authors in the system, recommendations could not be made at all for 13. This performance was consistent with my expectations given the sparseness of the data.

Table 3: Matrix of authors with wines tasted in common

Author	SS	JL	JB	SB	CG	SS	HW	AJ	AW
Steven Spurrier (SS)	-	6	1	1	7	0	15	30	1
James Lawther MW (JL)	6	-	0	0	0	15	0	0	0
Josephine Butchart (JB)	1	0	-	0	0	0	0	0	0
Stephen Brook (SB)	1	0	0	-	0	0	1	1	0
Christelle Guibert (CG)	7	0	0	0	-	0	1	5	0
Serena Sutcliffe (SS)	0	15	0	0	0	-	0	0	0
Harriet Waugh (HW)	15	0	0	1	1	0	-	10	0
Andrew Jefford (AJ)	30	0	0	1	5	0	10	-	0
Amy Wislocki (AW)	1	0	0	0	0	0	0	0	-

Table 4: Pearson Similarity of Authors (Appendix B.3)

Author	SS	JL	JB	SB	CG	SS	HW	AJ	AW
Steven Spurrier (SS)	-	0.58	0.0	0.0	0.0	-	0.67	0.49	0.0
James Lawther MW (JL)	0.58	-	-	-	-	0.22	-	-	-
Josephine Butchart (JB)	0.0	-	-	-	-	-	-	-	-
Stephen Brook (SB)	0.0	-	-	-	-	-	0.0	0.0	-
Christelle Guibert (CG)	0.0	-	-	-	-	-	0.0	0.0	-
Serena Sutcliffe (SS)	-	0.22	-	-	-	-	-	-	-
Harriet Waugh (HW)	0.67	-	-	0.0	0.0	-	-	0.71	-
Andrew Jefford (AJ)	0.49	-	-	0.0	0.0	-	0.71	-	-
Amy Wislocki (AW)	0.0	-	-	-	-	-	-	-	-

4.4 API Interface

4.4.1 Interchange Format

I chose to use JSON (JavaScript Object Notation) as my response format, which is a commonly used by RESTful APIs. It is designed to be natively handled by Javascript, but also has a sophisticated level of support in Python and other languages.

4.4.2 Routes

For the API I conceived a very simple set of routes, based on providing accesss to two the different object types, authors and wines. with an additional index route at the root to aid discoverability. Figure 3 shows the routing structure for the API.

Each route has a very simple url structure, with the plural item type and page number accessing a paged list of items, and the item type and id accessing any individual item.

Figure 3: API Routes. See code in Appendix E.1

```
/
/author/<id>
/authors/<page number>
/wine/<id>
/wine/<page number>
```

4.4.3 Response Specification

In principle RESTful services should favour discoverability, so as such the reponses contain links to other documents available on the API which are relevant to them. The objects are also designed to be consistent with each other as far as possible, using the same attribute for comparable properties wherever possible.

Every response is an object with the attributes "type" and "self". The "type" attribute is to aid the client in interpreting the object and for this API can take the values "author", "wine", or "list". "self" is a sub-object containing at least the fields "title" and "link", with additional fields depending on the type of the object.

"wine"- and "author"-type objects have the attribute "related_content", which is where different kinds of recommended content will be inserted by the API. The "list"-type objects have the attribute "list", which is a list of items with "title" and "link" attributes.

By structuring the responses this way I hope to make it as simple as possible for any client of the system to handle the different routes and navigate the API, as they should be able to ascertain the expected format of any object very quickly.

This example represents the template of an object with the minimum required fields, where %s represents a placeholder for a string value:

```
{
    ''type'': %s
    ''self'': {
        ''title'': %s
        ''link'': %s
    }
}
```

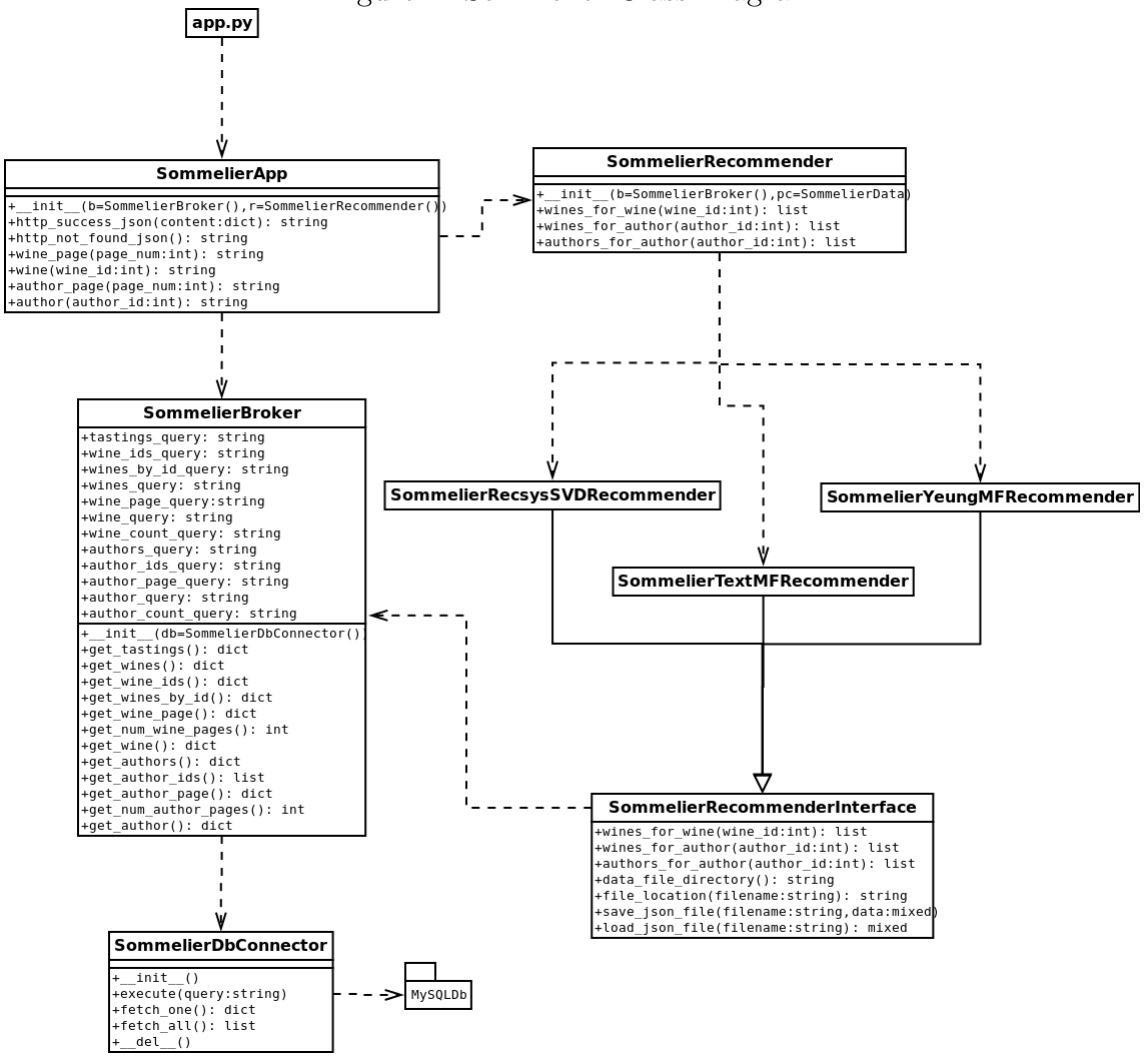
Full example responses for each content type are shown in Appendices C.1.1, C.1.2, C.1.3, C.1.4, and C.1.5. Responses are in the JSON format as specified by JSON.org (2013 [19]).

4.5 System Implementation

Figure 4 is a UML class diagram representing the complete Sommelier system.

The system is comprised of five main classes: Sommelier, SommelierBroker, SommelierDbConnector, SommelierRecommender, and SommelierRecommenderInterface. Additionally there are classes for different recommender implementations, and interactions with the files app.py and the sommelier_precompute.py are indicated.

Figure 4: Sommelier Class Diagram



4.5.1 Flask

The Flask framework is initialized by the file app.py, and this file is where all HTTP requests to the API are handled. For each route there is configuration in this file that calls the corresponding method on Sommelier. In this way the workings of the Sommelier application are completely unknown to Flask, and similarly Sommelier does not need to manage requests or routing in any way. Flask simply receives a request and routes it appropriately, passing Sommelier's return values to its Response class, which formats the HTTP response in a standard manner.

The following example is the routing in app.py for the route /authors, and shows how the only coupling between the Flask and Sommelier systems is the one-to-one relationship of Flask routes to methods on Sommelier (line 6):

```
@app.route('/authors', defaults = {'page_num': 1}, methods = ['GET'])
@app.route('/authors/<int:page_num>', methods = ['GET'])
def sommelier_authors(page_num):
    response_body, keyed_args_dict = sommelier.author_page(page_num)
    return Response(response_body, **keyed_args_dict)
```

Flask has a built-in development server, which can be launched from app.py by

calling app.run(). I do so in the main method, enabling debugging mode as a keyed argument:

```
if _name_ ==  '_main_':
    app.run(debug=True)
```

By default this launches the server on port 5000 of localhost. Neither debug mode, not the development server are suitable for production environments, for which there are a number of other more robust options (Flask Deployment Options, 2013 [13]).

4.5.2 Database Access

Maintaining a connection to MySQL-python comes with the attendant complexity of managing of the lifecycle of a database connection object and cursor. To avoid having to recreate these objects many times in the code I wrote the wrapper SommelierDbConnector (see Figure 4), which manages the connection and cursor in the minimum fashion, while exposing the minimum functionality to the application for executing on and fetching from the database. The full code is in Appendix **??**.

To further abstract the database from the application logic of the system I created a broker class, SommelierBroker, which holds a hard coded set of queries that are performed on the database. I a new query is to be made on the system it is to be written into the broker. This approach has serious drawbacks, such as that it would be potentially unwieldy to scale. However, as I developed the system I found that there was only a very small set of queries that I was using, and decided that it was easier in the short term for one broker to own all of them.

4.5.3 Recommenders

The system gets recommendations via the facade class SommelierRecommender (see Figure 4. This pattern abstracts away the complexity of making recommendations from the rest of the system, offering a simple interface with which to query whichever of the recommenders is instantiated.

This pattern was very helpful when it came to developing the recommenders, as it makes it trivial to swap one out for another in the system.

4.5.4 Precomputation

For various recommendation approaches it is necessary to precompute data. I have not modelled this behaviour in Figure 4 as there are many potential ways of doing so. I have assumed that arbitrary scripts would be created and run on a scheduled basis to carry out precomputation for methods such as matrix factorization or SVD, where imputing matrices cannot practically be done at runtime.

5 Testing and Evaluation

5.1 Testing the Code

5.1.1 Unit Tests

For the main components of the system I have produced unit tests (see Appendix G.1) which are written using Python's standard, JUniti-inspired test framework, unittest (unittest framework, 2013 [36]). These tests cover the classes Sommelier, SommelierBroker, SommelierRecommenderBase, and various parts of recommender classes SommelierPearsonCFRecommender and SommelierYeungMFRecommender.

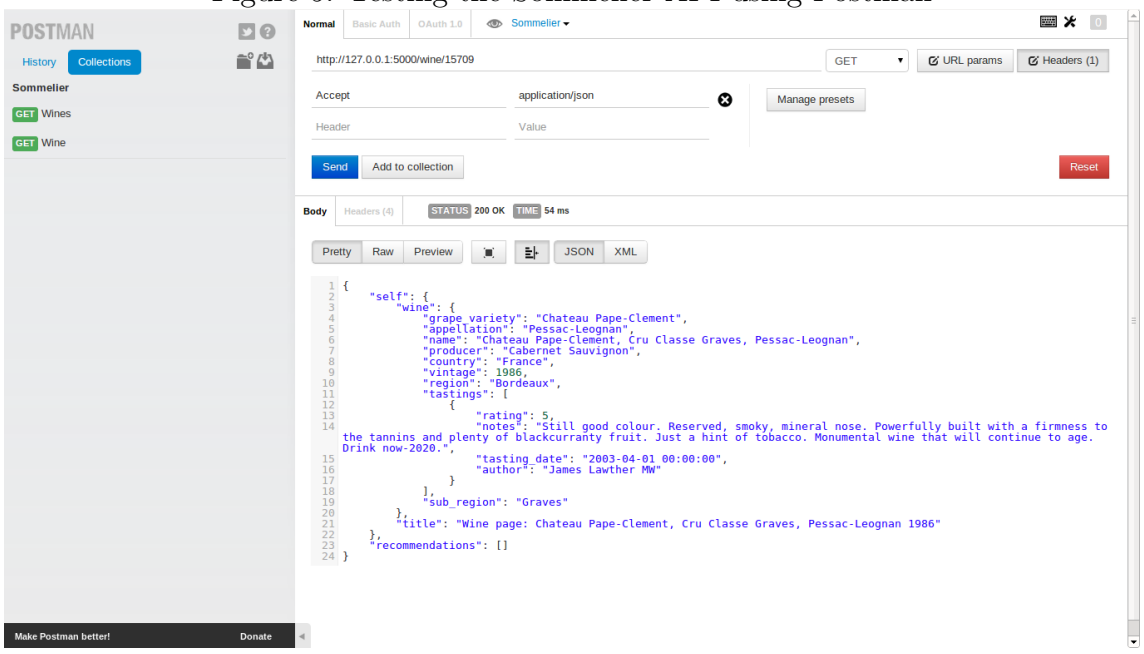
In each case I have attempted to ensure that I test key units of work, using the python's mock library to factor out dependencies. This way the unit tests for one class will not fail because of a problem in another class, making maintanance of both the tests and the application code easier.

I have used the nose module as a test runner (Testing with nose, 2013 [37]), which automatically locates and executes test files with the project. Automatically discovering and running all tests with nose is trivial:

```
cd /path/to/sommelier
./flask/bin/nosetests
```

5.1.2 Functional Testing

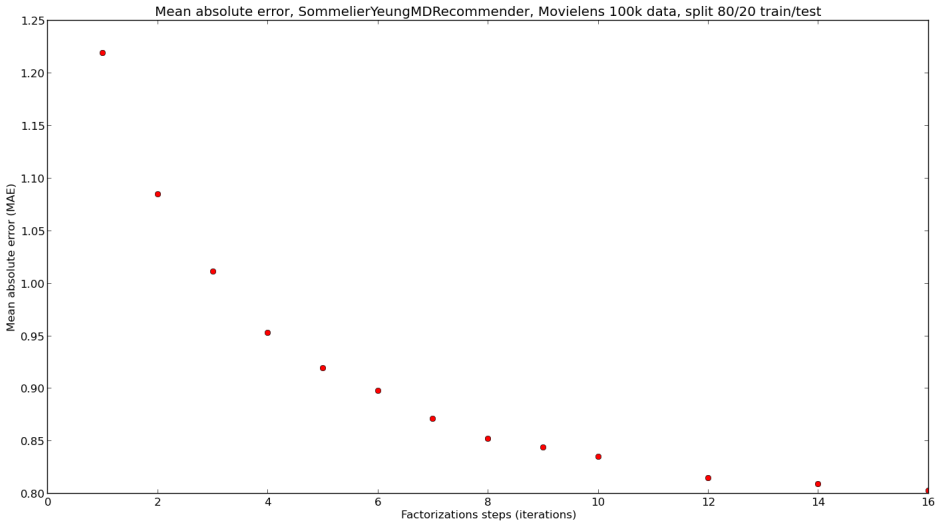
Figure 5: Testing the Sommelier API using Postman



I have not implemented any automated functional testing, instead relying on manual testing to ensure that the API functions as it should. For this purpose I have used Postman (Postman website, 2013 [31]), which is an extension for the Google Chrome web browser (Chrome Extension [6]) designed as a testing client for REST APIs. Figure 5 shows the interface for Postman in Google Chrome.

5.2 Testing Recommendations

Figure 6: Testing the SommelierYeungMFRecommender with Movielens Data



When I came to test my recommendations, the most developed recommender implementation in my system was the SommelierYeungMFRecommender, based on a matrix factorization algorithm published on a blog by Albert Yeung (Quuxlabs, Matrix Factorization: A Simple Tutorial and Implementation in Python, 2010 [49]). This system imputes a matrix of predicted values by using a gradient descent algorithm to iteratively improved its similarity to the original data.

Testing recommendations made by the system has been challenging. Of the three metrics put forward by Shani and Gunawardana (2011 [42]) that I raised in the literature review: prediction accuracy, item-space coverage and user-space coverage, I was only succesful in attempting to test one: prediction accuracy.

The metric I prefered for the purpose of evaluation was Mean Absolute Error (MAE), which is a commonly used measure (Su and Khoshgoftaar, 2009 [45]).

I wrote a method, split_data_evaluation(), in the class SommelierYeungMFRecommender. This splits the tastings into two sets according to a percentage split, one for testing and one for training. The training data used to impute a matrix of predicted ratings, including ratings for items in the test set which are unseen by the system. Once the imputation is complete the predicted ratings are compared with the true ratings in the test set and the MAE is calculated, along with the standard deviation of the error (see Appendix F.3.5).

I did not have a great deal of time to carry out my tests, having estimated a week for this task. Unfortunately I underestimated the time cost of running many iterations of a computationally intensive process, finding that each iteration of tests and improvements took several hours. By the time I had a satisfactory testing system there was no more development time left available.

I did produce one good set of data supporting the effectiveness of Yeung's algorithm for collaborative filtering, which was conducted using test data from the Movielens 100k data set (GroupLens, MovieLens Data Sets, 2011 [22]). Figure 6

shows Yeung's algorithm successfully reducing the MAE of the MovieLens data as iterations increase. When casually compared (they are not directly comparable) to the MAE results reported by Su and Khoshgoftaar (Su and Khoshgoftaar, 2006 [44]), where the best MAE was 0.769 for the tests they undertook, the MAE of just over 0.8 for the SommelierYeungMFRecommender over 16 iterations looks quite promising. To gather this data I implemented the function split_data_evaluate_movielens_file() on the SommelierYeungMFRecommender class.

I did not satisfactorily test the system against the Decanter.com database. The sparsity of that data does not necessarily preclude successful results, but it was not until late in the project that I recognised the correct way to evaluate the system. I already spent a certain amount of effort testing the Decanter.com data in an invalid manner. Implementing the train/test system with the MovieLens data was the last thing I had time to do, unfortunately. A few more days would have seen the results for the Decanter.com data.

6 Conclusion

In some ways this project can be considered successful. Some objectives have been achieved well, such as that I have developed an API for the Decanter.com wine tasting data, and augmented the responses with recommendations.

The components of the system are fairly decoupled. I have successfully abstracted away a great deal of complexity, so that between classes there is little data shared, and the system's components operate largely independently of each other. In general I am very happy with the system's architecture.

On the other hand, I have run out of time, and am forced to submit the project with failing unit tests, and without having implemented the recommendation methods for my best recommender. Those are critical failures.

It was always my goal to be able to assert that the recommendations are good, which I am not able to do, as recommendations are successfully made with only the weakest system, raw collaborative filtering. Additionally, I have not succeeded in exploiting anything specific to wines when making my recommendations. The recommender I have produced is no different to one that might produce movie recommendations, or recommendations for any other generic rated item.

There is the foundation of a good system, but more work is needed in order for it to fulfil its potential.

I don't think that I managed my time as well as I could, and the project is less successful as a result.

7 Reflection

I have mixed feelings about this project. On the one hand I implemented a functioning system which, in many ways, satisfied my original objectives: the API works reliably, with a response format I am happy with, the RESTful pattern employed in the API layer is good, and Flask helped make a success of the RESTful API - it was a good choice of framework. The architecture of the Sommelier system is fairly sound, and the system is capable of making recommendations.

One the other hand, I am disappointed that I was not able to complete a better recommender. It may be that the ad-hoc matrix factorization-based recommender I built in imitation of Albert Yeung's work (Quuxlabs, Matrix Factorization: A Simple Tutorial and Implementation in Python, 2010 [49]) is an excellent recommender, but I have only produced a single, small set of test results to suggest that that may be the case, and in any case I haven't wired it into the Sommelier API.

My failure to satisfactorily architect the task of pre-imputing data for imputing ratings was a mistake. I should have planned an architecture around running such imputation tasks.Likewise, it was late in the project I introduced benchmarking of recommendation quality. I should have planned to do this from the beginning and prepared for it. The lack of foresight in this regard means I am unable to accurately the quality of the system's recommendations.

Overall I feel a lack of up-front planning damaged my project. I should have spent more time analysing the practicalities of implementing systems using complex mathematical techniques such as matrix factorization. I may have had more success taking a less hands on approach to the mathematics of the recommendation and had more success implementing a library such like recsys-svd.

Finally it is a disappointment that I did not produce a recommender that offers anything unique to the wine domain. In part I was hampered by poor quality source data, which took far longer to disentangle and migrate than I had anticipated it would, but had I planned better I would have had time to try to make something of the written tasting notes.

Nevertheless I have learnt a great deal. Prior to this project I had written very little Python, never used Flask, and never attempted to implement a recommender system. Despite my mixed success in the latter I am pleased with my achievements in the former. What I have to show for my effort is, I hope, good in parts. If I were to do it all again I would do it differently, and hopefully better.

References

[1] Benatallah, B., Nezhad, M., Hamid, R.. (2008). Service Oriented Architecture: Overview and Directions. In: B“orger, Egon and Cisternino, Antonio Advances in Software Engineering. Berlin, Heidelberg: Springer, Verlag. 116-130.

[2] Burke, R.. (1999). *The Wasabi Personal Shopper: A Case-Based Recommender System*. Submitted to the 11th Annual Conference on Innovative Applications of Artificial Intelligence.

[3] Burke, R.. (1999). *Integrating Knowledge-Based and Collaborative-Filtering Recommender Systems*. In: Artificial Intelligence for Electronic Commerce: Papers from the AAAI Workshop (AAAI Technical Report WS-99-0 1), pp.69-72.

[4] Burke, R.. (2000). *Knowledge-Based Recommender Systems*, Encyclopedia of Library and Information Systems. Marcel Dekker.

[5] Burke, R.. (2002). *Hybrid Recommender Systems: Survey and Experiments*, User Modeling and User-Adapted Interaction, Volume 12 Issue 4, November 2002, Pages 331 - 370. Kluwer Academic Publishers: Hingham, MA, USA

[6] Chrome Web Store. *Postman - REST client*. Retrieved May 4, 2013 from https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm

[7] Claypool, M., Gokhale, A., Miranda, T., Murnikov, P., Netes, D., Sartin, M.. (1999). *Combining Content-Based and Collaborative Filters in an Online Newspaper*, Recommender Systems - Implementation and Evaluation, ACM SIGIR Workshop On, August 1999.

[8] Cockburn, A.. (2008). *Using both incremental and iterative development* STSC CrossTalk, Vol. 21, No. 5.

[9] Debnath, S., Ganguly, N., Mitra, P.. (2008). *Feature weighting in content based recommendation system using social network analysis*, Proceedings of the 17th international conference on World Wide Web, WWW '08, 2008, Beijing, China, Pages 1041 - 1042. ACM: New York, NY, USA,

[10] Decanter.com, *Wine Reviews*. Retrieved May 4, 2013 from http://www.decanter.com/wine/reviews/1

[11] Django Project Homepage. Retrieved May 4, 2013 from http://www.djangoproject.com

[12] Flask (A Python Microframework), *Welcome*. Retrieved May 4, 2013 from http://flask.pocoo.org/

[13] Flask, *Deployment Options*. Retrieved May 4, 2013 from http://flask.pocoo.org/docs/deploying/

[14] Flask Documentation *Caching*. Retrieved May 4, 2013 from http://flask.pocoo.org/docs/patterns/caching/

[15] Mangalindan, J. P.. (2012). *Amazon's Recommendation Secret*, July 2012. Retrieved May 4, 2013 from http://tech.fortune.cnn.com/2012/07/30/amazon-5/

[16] Github Website, *About*. Retrieved May 4, 2013 from https://github.com/about

[17] Goldberg, D. Nichols, D., Oki, B. M., and Terry, D.. (1992). *Using collaborative filtering to weave an information tapestry*, Commun. ACM 35, 12 (Dec. 1992), 61–70.

[18] The Java EE 6 Tutorial, *Introduction to the Java Persistence API*. Retrieved May 4, 2013 from http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

[19] JSON Homepage. Retrived May 4, 2013 from http://www.json.org

[20] McNee, S. M., Riedl, J., Konstan, J. A.. (2006). *Accuracy is not always good: how accuracy metrics have hurt recommender systems*, Extended Abstracts of the 2006 ACM Conference on Human Factors in Computing Systems CHI 2006

[21] MongoDB. Retrieved May 4, 2013 from http://www.mongodb.org/

[22] GroupLens, MovieLens Data Sets. Retrieved May 4, 2013 from http://www.grouplens.org/node/73

[23] MySQL, *The world's most popular open source database*. Retrieved May 4, 2013 from http://www.mysql.com/

[24] Netflix, *About us*. Retrieved May 4, 2013 from https://signup.netflix.com/MediaCenter

[25] Netflix Prize Website, *Index*. Retrieved May 4, 2013 from http://www.netflixprize.com//index

[26] Netflix Prize Website, *Rules*. Retrieved May 4, 2013 from http://www.netflixprize.com//rules

[27] Numpy, *Scientific Computing Tools for Python*. Retrieved May 4, 2013 from http://www.numpy.org/

[28] Patton, E., McGuinness, D.. (2010). *Scaling the Wall: Experiences Adapting a Semantic Web Application to Utilize Social Networks on Mobile Devices*, 2010. In: Proceedings of the WebSci10: Extending the Frontiers of Society On-Line, April 26-27th, 2010, Raleigh, NC: US.

[29] Pautasso, C., Zimmermann, O., Leymann, F.. (2008). *RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision*, 17th International World Wide Web Conference (WWW2008). SOA4All, Enabling the SOA Revolution on a World Wide Scale. Proceedings of the 2nd IEEE International Conference on Semantic Computing ICSCIEEE Computer Society. IEEE 2008

[30] PostgreSQL *The world's most advanced open source database*. Retrieved May 4, 2013 from http://www.postgresql.org/

[31] Postman website. Retrieved May 4, 2013 from http://www.getpostman.com/

[32] Python Documentation, *Using the Python Interpreter*. Retrieved May 4, 2013 from http://docs.python.org/2/tutorial/interpreter.html

[33] Python Documentation, *Overview*. Retrieved May 4, 2013 from http://docs.python.org/2/index.html

[34] Python Website: *python-memcached*. Retrieved May 4, 2013 from https://pypi.python.org/pypi/python-memcached/

[35] Python Website: *Quotes about Python*. Retrieved May 4, 2013 from http://www.python.org/about/quotes/

[36] Python Unittest. Retrieved May 4, 2013 from http://docs.python.org/2/library/unittest.html

[37] nose, *Testing With nose*. Retrieved May 4, 2013 from http://nose.readthedocs.org/en/latest/testing.html

[38] Resnick, P., Iacovou, N., Sushak, M., Bergstrom, P., Riedl, J.. (1994). *GroupLens: An open architecture for collaborative filtering of netnews*, 1994 ACM Conference on Computer Supported Collaborative Work, 1994. Association of Computing Machinery, Chapel Hill, NC.

[39] Resnick, P., Varian, H. R.. (1997). *Recommender Systems*, 1997. Communications of the ACM, 40 (3), 56-58. Association of Computing Machinery, Chapel Hill, NC.

[40] Scipy, - -. Retrieved May 4, 2013 from http://www.scipy.org/

[41] Segaran, T.. (2007). *Programming Collective Intelligence*. O'Reilly, CA, US.

[42] Shani, G., Gunawardana, A.. (2011). *Evaluating Recommender Systems*, Recommender Systems Handbook, pp257-297. Springer, US

[43] Sheikh, M. A. A., Aboalsamh, H. A., Albarrak, A.. (2011). *Migration of legacy applications and services to Service-Oriented Architecture (SOA)*, Current Trends in Information Technology (CTIT), 2011 International Conference and Workshop on, pp.137,142, 26-27 Oct. 2011

[44] Su, X., Khoshgoftaar, T.M.. (2006). *Collaborative Filtering for Multi-class Data Using Belief Nets Algorithms*, Tools with Artificial Intelligence, 2006. ICTAI '06. 18th IEEE International Conference on, pp.497,504, Nov. 2006

[45] Su, X., Khoshgoftaar, T. M.. (2009). *A Survey of Collaborative Filtering Techniques*, Advances in Artificial Intelligence, vol. 2009, Article ID 421425, 19 pages, 2009.

[46] http://wineagent.tw.rpi.edu/index.php

[47] Ubuntu 12.04.2 LTS (Precise Pangolin). Retrieved May 4, 2013 from http://releases.ubuntu.com/precise/

[48] Ward, R., Towne, D., Stannard, D. H., LaChappelle, P.. (2012). *Wine Recommendation System and Method*. International Application Number PCT/US2012/046480, filed 12/07/2012. Retrieved May 4, 2013 from http://patentscope.wipo.int/search/en/detail.jsf?docId=WO2013009990

[49] Yeung, A., (2010). *Matrix Factorization: A Simple Tutorial and Implementation in Python*. Retrieved May 4, 2013 from http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/

A Installation

These dependecies should be installed from the command line, having navigated to the project directory.

```
# We need pip to be installed to manage python packages
sudo apt-get install python-pip
pip install -U pip

# vitualenv enables us to install modules and packages local to our
# project, so we dont need to expose our system-level python
# installation to incompatible or otherwise obnoxious packages
# that might destabilize our other project, or OS in general
pip install virtualenv
virtualenv flask

# MySQL-python will enable us to query data (very useful!)
# Advice retrieved from http://codeinthehole.com/writing/how-to-set-up-mysql-for-python-on-ubuntu/
# there are some additional dependencies with MySQL-python that need
# to be installed at a system level
sudo apt-get install libmysqlclient-dev python-dev
# Documentation on working with MySQLdb is at: http://mysql-python.sourceforge.net/MySQLdb.html

# now we can install MySQL-python in our virtualenv using a local pip
./flask/bin/pip install MySQL-python

# flask itself is our web framework of choice
./flask/bin/pip install flask

# mock will be useful for unit testing
./flask/bin/pip install mock

# nose will serve as our test runner
# https://nose.readthedocs.org/en/latest/ Source: https://github.com/nose-devs/nose
./flask/bin/pip install nose

# gonna need numpy a lot probably!
# scipy is a pain. The following installation method is courtesy of:
# http://www.scipy.org/Installing_SciPy/Linux#head-d437bf93b9d428c6efeb08575f631ddf398374ea
# This installs rather a lot of stuff :-|
sudo apt-get build-dep python-numpy
# the following command does a big build and throws all sorts of errors, which are apparently fine to ignore.
sudo apt-get -b source python-numpy
./flask/bin/pip install scipy

# install dependencies for python-recsys
flask/bin/pip install csc-pysparse networkx divisi2
# clone python-recsys from Git and set it up in virtualenv
git clone http://github.com/ocelma/python-recsys
cd python-recsys
../flask/bin/python setup.py install

# install matplotlib for graphing test results
# n.b. this is not subsequently installed in the virtualenv, but
# that isnt a problem as the graphing can be done without
# invoking any of our virtualenv
sudo apt-get install python-matplotlib
```

B Source Data

B.1 Migration

The notes in this section include the commands that recreate the migration of data from the original Decanter.com wines database to the Sommelier database.

B.1.1 Convert to UTF-8

```
## Conversion from latin1 to utf8:

Based on advice from: http://en.gentoo-wiki.com/wiki/Convert_latin1_to_UTF-8_in_MySQL

From the Bash shell:
```

```
$ mysqldump -uroot -p -hlocalhost --default-character-set=latin1 -c --insert-ignore --skip-set-charset -r wine_dump.sql w
$ file wine_dump.sql
> wine_dump.sql: Non-ISO extended-ASCII English text, with very long lines
$ iconv -f ISO8859-1 -t UTF-8 wine_dump.sql > wine_dump_utf8.sql
$ sed -i 's/latin1/utf8/g' wine_dump_utf8.sql

Now, from the MySQL command line:
mysql> CREATE DATABASE sommelier CHARACTER SET utf8 COLLATE utf8_general_ci;

And finally, back in the Bash shell:
$ mysql -uroot --max_allowed_packet=16M -p --default-character-set=utf8 sommelier < wine_dump_utf8.sql
```

B.1.2 Create Sommelier Tables

This is the code for creating the new Sommelier database tables. The code is executed on a copy of the original database which has been duplicated with a new name. After creating the new tables, the old tables are deleted and the new tables are renamed to the old names (wine, author, tasting).

Content for sommelier.wine:

```
CREATE TABLE `sommelier_wine` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL DEFAULT '',
  `vintage` int(4) NOT NULL DEFAULT '0',
  `grape_variety` varchar(255) NOT NULL DEFAULT '',
  `producer` varchar(255) NOT NULL DEFAULT '',
  `country` varchar(255) NOT NULL DEFAULT '',
  `region` varchar(255) NOT NULL DEFAULT '',
  `sub_region` varchar(255) NOT NULL DEFAULT '',
  `appellation` varchar(255) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

INSERT INTO sommelier_wine SELECT
  w.id,
  w.name AS name,
  w.vintage AS vintage,
  p.producer_match AS producer,
  gv.description AS description,
  c.country AS country,
  r.region AS region,
  sr.sub_region AS sub_region,
  a.appellation AS appellation
FROM
  wine w
JOIN wine_info wi ON w.id = wi.id
LEFT JOIN producers p ON p.id = wi.producer_id
LEFT JOIN wine_grape_variety gv ON gv.id = wi.grape_variety
LEFT JOIN appellation a ON a.id = wi.appellation_id
LEFT JOIN sub_region sr ON sr.id = a.sub_region_id
LEFT JOIN region r ON r.id = sr.region_id
LEFT JOIN country c ON c.id = r.country_id
ORDER BY w.id ASC;

CREATE TABLE `sommelier_author` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

INSERT INTO sommelier_author SELECT DISTINCT
  NULL,
  t.author as name
FROM
  tasting t
WHERE t.author <> '';

CREATE TABLE `sommelier_tasting` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `wine_id` int(11) NOT NULL,
  `author_id` int(11) NOT NULL,
  `rating` int(11) NOT NULL,
```

```
  `notes` TEXT NOT NULL,
  `tasting_date` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  PRIMARY KEY (`id`),
  KEY `wine_idx` (`wine_id`),
  KEY `author_idx` (`author_id`)
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

INSERT INTO sommelier_tasting SELECT
  NULL,
  t.wine_id AS wine_id,
  a.id AS author_id,
  t.rating AS rating,
  t.notes AS notes,
  t.tasting_date AS tasting_date
FROM
  tasting t
JOIN wine w ON w.id = t.wine_id
JOIN wine_info wi ON w.id = wi.id
LEFT JOIN sommelier_author a ON t.author = a.name
WHERE t.rating > 0
  AND t.notes <> ''
ORDER BY w.id ASC;

Finally, delete all wines without tasting records:

DELETE FROM sommelier_wine WHERE id NOT IN ( SELECT wine_id FROM sommelier_tasting );

DROP TABLE wine;
DROP TABLE tasting;
DROP TABLE author;

RENAME TABLE sommelier_wine TO wine;
RENAME TABLE sommelier_tasting TO tasting;
RENAME TABLE sommelier_author TO author;
```

B.2 Sparsity

Sparsity of 94% calculated by:
Distinct number of known authors in tasting table: 18
Total number of tastings by known authors: 1411
Distinct number of wine ids in tasting table with known author: 1307
Sparsity percentage $= 100 - ((1411 \div (1307 \times 18)) \times 100)$

B.3 Author Similarity

Table data obtained in the Python interactive interpreter by the following commands (see also D.1):

```
import recommendations
recommendations.getAuthorSimilarities()
```

C API Design

C.1 API Routes

C.1.1 API Response: Index

```
{
    ``type'': ``list'',
    ``self'': {
        ``title'': ``Sommelier API'',
        ``link'': ``/''
    },
    ``list'': [
        {
        ``title'': ``All Authors'',
        ``link'': ``/authors/1''
        },
        {
        ``title'': ``All Wines'',
        ``link'': ``/wines/1''
```

```
        }
    ]
}
```

C.1.2 API Response: Authors

```
{
    ‘‘type’’: ‘‘list’’,
    ‘‘self’’: {
        ‘‘title’’: ‘‘Authors, Page 1’’
        ‘‘link’’: ‘‘/authors/1’’
    },
    ‘‘list’’: [
        /* maximum 50 links per page */
        {
        ‘‘title’’: ‘‘Mr. Author’’,
        ‘‘link’’: ‘‘/author/123’’
        },
        {
        ‘‘title’’: ‘‘A.N. Other’’,
        ‘‘link’’: ‘‘/author/234’’
        }
    }
}
```

C.1.3 API Response: Author

```
{
    ‘‘type’’: ‘‘author’’,
    ‘‘self’’: {
        ‘‘title’’: ‘‘Mr. Author’’,
        ‘‘name’’: ‘‘Mr. Author’’,
        ‘‘tastings’’: [
            {
            ‘‘rating’’: 5,
            ‘‘notes’’: ‘‘Tasting notes about the wine’’,
            ‘‘tasting_date’’: ‘‘2003-04-01 00:00:00’’,
            ‘‘wine’’: {
                ‘‘title’’: ‘‘Wine Name 1990’’,
                ‘‘link’’: ‘‘/wine/123’’
            }
        ],
        ‘‘link’’: ‘‘/author/123’’
    },
    ‘‘related_content’’: {
        /* maximum of 5 wines */
        ‘‘recommended_wines’’: [
            {
            ‘‘title’’: ‘‘Chateau du Vin 1996’’,
            ‘‘link’’: ‘‘/wine/234’’
            }
        ],
        /* maximum of 5 other authors */
        ‘‘similar_authors’’: [
            {
            ‘‘title’’: ‘‘A.N. Other’’,
            ‘‘link’’: ‘‘/author/234’’
            }
        ]
    }
}
```

C.1.4 API Response: Wines

```
{
    ‘‘type’’: ‘‘list’’
    ‘‘self’’: {
        ‘‘title’’: ‘‘Wines, Page 1’’
        ‘‘link’’: ‘‘/wines/1’’
    },
    ‘‘list’’: [
        /* maximum 50 links per page */
```

```
        {
        ‘‘title’’: ‘‘Wine Name 1990’’,
        ‘‘link’’: ‘‘/wine/123’’
        },
        {
        ‘‘title’’: ‘‘Chateau du Vin 1996’’,
        ‘‘link’’: ‘‘/wine/234’’
        }
    }
}
```

C.1.5 API Response: Wine

```
{
    ‘‘type’’: ‘‘wine’’,
    ‘‘self’’: {
        ‘‘title’’: ‘‘Wine Name 1990’’,
        ‘‘name’’: ‘‘Wine Name’’,
        ‘‘vintage’’: 1990,
        ‘‘producer’’: ‘‘Wine Producer Name’’,
        ‘‘grape_variety’’: ‘‘Cabernet Sauvignon’’,
        ‘‘appellation’’: ‘‘Pessac-Leognan’’,
        ‘‘country’’: ‘‘France’’,
        ‘‘region’’: ‘‘Bordeaux’’,
        ‘‘sub_region’’: ‘‘Graves’’
        ‘‘tastings’’: [
            {
            ‘‘rating’’: 5,
            ‘‘notes’’: ‘‘Tasting notes about the wine’’,
            ‘‘tasting_date’’: ‘‘2003-04-01 00:00:00’’,
            ‘‘author’’: {
                ‘‘title’’: ‘‘Mr. Author’’,
                ‘‘link’’: ‘‘/author/123’’
            }
        ],
        ‘‘link’’: ‘‘/wine/123’’
    },
    ‘‘related_content’’: {
        /* maximum of 5 wines */
        ‘‘similar_wines’’: [
            {
            ‘‘title’’: ‘‘Chateau du Vin 1996’’,
            ‘‘link’’: ‘‘/wine/2345’’
            }
        ]
    }
}
```

D Experimental Code

D.1 Code derived from Segaran, 2007

This code was copied largely from Segaran's exercises in Ch.2 of Collective Intelligence (2007). I have truncated part of the data fixture for brevity.
I have adapted Segaran's code, implementing new methods to query data from the Sommelier database and apply his methods to that data.

```
# a dictionary of critics and their ratings of a small
# set of movies
# copied from Segaran: Collective Intelligence (2006) Ch.2
critics={
    'Lisa Rose': {
        'Lady in the Water': 2.5,
        'Snakes on a Plane': 3.5,
        'Just My Luck': 3.0,
        'Superman Returns': 3.5,
        'You, Me and Dupree': 2.5,
        'The Night Listener': 3.0
    },
#### TRUNCATED ####
    'Toby': {
```

```
        'Snakes on a Plane': 4.5,
        'Superman Returns': 4.0,
        'You, Me and Dupree': 1.0
    }
}

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
from math import sqrt
def sim_distance(prefs,person1,person2):
    si={}
    for item in prefs[person1]:
        if item in prefs[person2]:
            si[item]=1
    if len(si)==0: return 0
    sum_of_squares=sum([pow(prefs[person1][item]-prefs[person2][item],2)
    return 1/(1+sum_of_squares)

# This method is equivalent to sim_distance() above, uses scipy's sqeuclidean method
import scipy.spatial
def euclidean_distance(prefs,person1,person2):
    vector1=[]
    vector2=[]
    for item in prefs[person1]:
        if item in prefs[person2]:
            vector1.append(prefs[person1][item])
            vector2.append(prefs[person2][item])
    if len(vector1)==0: return 0
    euclidean_distance=scipy.spatial.distance.sqeuclidean(vector1,vector2)
    return 1 / (1 + euclidean_distance)

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def sim_pearson(prefs,p1,p2):
    si={}
    for item in prefs[p1]:
        if item in prefs[p2]: si[item]=1
    n=len(si)
    if n==0: return 0
    sum1=sum([prefs[p1][it] for it in si])
    sum2=sum([prefs[p2][it] for it in si])
    sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
    sum2Sq=sum([pow(prefs[p2][it],2) for it in si])
    pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])
    # calculate Pearson score:
    num=pSum-(sum1*sum2/n)
    den=sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
    if den==0: return 0
    r=num/den
    return r

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def topMatches(prefs,person,n=5,similarity=sim_pearson):
    scores=[(similarity(prefs,person,other), other)
            for other in prefs if other!=person]
    scores.sort()
    scores.reverse()
    return scores[0:n]

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
# Gets recommendations for a person by using weighted average
# of every other user's rankings
def getRecommendations(prefs,person,similarity=sim_pearson):
    totals={}
    simSums={}
    for other in prefs:
        if other==person: continue
        sim=similarity(prefs,person,other)
        if sim<=0: continue
        for item in prefs[other]:
            # only score movies 'person' hasn't seen
            if item not in prefs[person] or prefs[person][item]==0:
                # similarity*score
                totals.setdefault(item,0)
                totals[item]+=prefs[other][item]*sim
```

```
                # sum of similarities
                simSums.setdefault(item,0)
                simSums[item]+=sim
    rankings=[(total/simSums[item],item) for item,total in totals.items()]
    rankings.sort()
    rankings.reverse()
    return rankings

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def transformPrefs(prefs):
    result={}
    for person in prefs:
        for item in prefs[person]:
            result.setdefault(item,{})
            result[item][person]=prefs[person][item]
    return result

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def calculateSimilarItems(prefs,n=10,similarity=sim_distance):
    result={}
    itemPrefs=transformPrefs(prefs)
    c=0
    for item in itemPrefs:
        c+=1
        if c%100==0: print "%d / %d" % (c,len(itemPrefs))
        scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
        result[item]=scores
    return result

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def getRecommendedItems(prefs,itemMatch,user):
    userRatings=prefs[user]
    scores={}
    totalSim={}
    for (item,rating) in userRatings.items():
        for (similarity, item2) in itemMatch[item]:
            if item2 in userRatings: continue
            # Weighted sum of rating times similarity
            scores.setdefault(item2,0)
            scores[item2]+=similarity*rating
            # Sum of all the similarities
            totalSim.setdefault(item2,0)
            totalSim[item2]+=similarity
    # Divide each total score by total weighting to give an average
    rankings=[(score/totalSim[item],item) for item,score in scores.items()]
    rankings.sort()
    rankings.reverse()
    return rankings

# Method copied from Segaran: Collective Intelligence (2006) Ch.2
def loadMovieLens(path='../data/ml-100k'):
    movies={}
    for line in open(path+'/u.item'):
        (id,title)=line.split('|')[0:2]
        movies[id]=title
    prefs={}
    for line in open(path+'/u.data'):
        (user,movieid,rating,ts)=line.split('\t')
        prefs.setdefault(user,{})
        prefs[user][movies[movieid]]=float(rating)
    return prefs

def loadSommelierWines(comparator='rating'):
    import MySQLdb
    from MySQLdb.constants import FIELD_TYPE
    from MySQLdb.cursors import DictCursor
    converter = { FIELD_TYPE.LONG: int }
    connection = MySQLdb.connect(user="sommelier",db="sommelier",passwd="vinorosso",conv=converter)
    connection.set_character_set('utf8')
    cursor = connection.cursor(DictCursor)
    cursor.execute('SET NAMES utf8;')
    cursor.execute('SET CHARACTER SET utf8;')
    cursor.execute('SET character_set_connection=utf8;')
```

```
    cursor.execute("""
select w.name as wine, w.vintage, a.name as author, t.rating, t.notes
from wine w join tasting t on t.wine_id = w.id join author a on a.id = t.author_id
    """)
    results = cursor.fetchall()
    prefs={}
    for row in results:
        user = row['author']
        wine = row['wine']
        vintage = row['vintage']
        rating = row['rating']
        notes = row['notes']
        prefs.setdefault(user,{})
        if comparator == 'notes':
            comp = row['notes']
        else:
            comp = row['rating'] + 0.0
        prefs[user][''.join([wine,str(vintage)])] = comp
    cursor.close()
    connection.close()
    return prefs

def loadSommelierAuthors():
    import MySQLdb
    from MySQLdb.constants import FIELD_TYPE
    from MySQLdb.cursors import DictCursor
    converter = { FIELD_TYPE.LONG: int }
    connection = MySQLdb.connect(user="sommelier",db="sommelier",passwd="vinorosso",conv=converter)
    connection.set_character_set('utf8')
    cursor = connection.cursor(DictCursor)
    cursor.execute('SET NAMES utf8;')
    cursor.execute('SET CHARACTER SET utf8;')
    cursor.execute('SET character_set_connection=utf8;')
    cursor.execute("""
select w.name as wine, w.vintage as vintage, a.name as author, t.rating as rating from wine w join tasting t on t.wine_id
    """)
    results = cursor.fetchall()
    authors = {}
    for row in results:
        author = row['author']
        wine = ' '.join([row['wine'], str(row['vintage'])])
        rating = row['rating']
        authors.setdefault(author,{})
        authors[author][wine] = rating;
    cursor.close()
    connection.close()
    return authors

def getAuthorSimilarities(similarity=sim_pearson):
    authors = loadSommelierAuthors()
    sims = {}
    for author1 in authors.keys():
        sims.setdefault(author1, {})
        for author2 in authors.keys():
            if author1 == author2:
                continue
            sim = similarity(authors, author1, author2)
            if sim != 0:
                sims[author1][author2] = sim
    return sims
```

E Flask

E.1 App.py

```
#!python

# import and intialize Flask
from flask import Flask, Response
app = Flask(__name__)

# import and initialize Sommelier
```

```
from src.sommelier import Sommelier
sommelier = Sommelier()

@app.route('/')
def sommelier_index():
    response_body, keyed_args_dict = sommelier.index()
    return Response(response_body, **keyed_args_dict)

@app.route('/wines', defaults = {'page_num': 1}, methods = ['GET'])
@app.route('/wines/<int:page_num>', methods = ['GET'])
def sommelier_wines(page_num):
    response_body, keyed_args_dict = sommelier.wine_page(page_num)
    return Response(response_body, **keyed_args_dict)

@app.route('/wine/<wine_id>', methods = ['GET'])
def sommelier_wine(wine_id):
    response_body, keyed_args_dict = sommelier.wine(wine_id)
    return Response(response_body, **keyed_args_dict)

@app.route('/authors', defaults = {'page_num': 1}, methods = ['GET'])
@app.route('/authors/<int:page_num>', methods = ['GET'])
def sommelier_authors(page_num):
    response_body, keyed_args_dict = sommelier.author_page(page_num)
    return Response(response_body, **keyed_args_dict)

@app.route('/author/<author_id>', methods = ['GET'])
def sommelier_author(author_id):
    response_body, keyed_args_dict = sommelier.author(author_id)
    return Response(response_body, **keyed_args_dict)

if __name__ == '__main__':
    app.run(debug=True)
```

F Sommelier Application

F.1 Sommelier Database Connector

A class managing database queries for the Sommelier application. This class implements only the minimum interface with the MySQLDB library, simply managing the lifecycle of a single cursor and exposing the cursor's execute(), fetchone() and fetchall() methods.

```
#!python

import math
import MySQLdb
from MySQLdb.constants import FIELD_TYPE
from MySQLdb.cursors import DictCursor

class SommelierDbConnector:

    cursor = None
    connection = None

    def __init__(self):
        converter = { FIELD_TYPE.LONG: int }
        self.connection = MySQLdb.connect(
                            user="sommelier",
                            db="sommelier",
                            passwd="vinorosso",
                            conv=converter)
        self.connection.set_character_set('utf8')
        self.cursor = self.connection.cursor(DictCursor)
        self.cursor.execute('SET NAMES utf8;')
        self.cursor.execute('SET CHARACTER SET utf8;')
        self.cursor.execute('SET character_set_connection=utf8;')

    def execute(self, query):
        return self.cursor.execute(query)

    def fetch_one(self):
        return self.cursor.fetchone()
```

```
    def fetch_all(self):
        return self.cursor.fetchall()

    def __del__(self):
        if self.cursor is not None:
            self.cursor.close()
        if self.connection is not None:
            self.connection.close()
```

F.2 Sommelier

```
#!python

# import useful Flask libs
from flask import request, Response, jsonify, json

# import broker libs that will interface with the DB
from broker import SommelierBroker

# import the Sommelier recommender
from recommender import SommelierPearsonCFRecommender, SommelierRecsysSVDRecommender

class Sommelier():

    def __init__(self, b=SommelierBroker(), r=SommelierRecsysSVDRecommender()):
        self.broker = b
        self.recommender = r

    # Takes dict of content and generates JSON response with
    # appropriate MIME type, HTTP status code etc.
    def http_success_json(self, content):
        response = json.dumps(content, encoding="utf-8")
        response = ''.join(response.decode('unicode-escape').splitlines())
        return response, { 'status': 200, 'mimetype': 'application/json; charset=utf-8' }

    def http_not_found_json(self):
        response = json.dumps("404: Not found", encoding="utf-8")
        return response, { 'status': 404, 'mimetype': 'application/json; charset=utf-8' }

    def index(self):
        return self.http_success_json({
            'type': 'list',
            'self': {
                'title': 'Sommelier',
                'link': '/'
            },
            'list': [
                {
                'title': 'All Wines',
                'link': '/wines/1'
                },
                {
                'title': 'All Authors',
                'link': '/authors/1'
                }
            ]
        })

    def wine_page(self, page_num):
        records = self.broker.get_wine_page(page_num)
        if not records:
            return self.http_not_found_json()
        num_pages = self.broker.get_num_wine_pages()
        wines_list = []
        for wine in records:
            wines_list.append({
                'title': '{} {}'.format(wine['name'], wine['vintage']),
                'link': '/wine/{}'.format(wine['id'])
            })
        return self.http_success_json({
            'type': 'list',
            'self': {
```

```
            'title': 'Wines, Page {}'.format(page_num),
            'link': '/wines/{}'.format(page_num)
        },
        'list': wines_list
    })

def wine(self, wine_id):
    record = self.broker.get_wine(wine_id)
    if not record:
        return self.http_not_found_json()
    tastings_list = []
    for tasting in record['tastings']:
        tastings_list.append({
            'rating': tasting['rating'],
            'notes': tasting['notes'],
            'tasting_date': tasting['tasting_date'],
            'author': {
                'title': tasting['author'],
                'link': '/author/{}'.format(tasting['author_id'])
            }
        })
    wines = self.recommender.wines_for_wine(wine_id)
    wines_list = []
    for wine in wines:
        wines_list.append({
            'title': '{} {}'.format(wine['name'], wine['vintage']),
            'link': '/wine/{}'.format(wine['id'])
        })
    return self.http_success_json({
        'type': 'wine',
        'self': {
            'title': '{} {}'.format(record['name'], record['vintage']),
            'name': record['name'],
            'vintage': record['vintage'],
            'grape_variety': record['grape_variety'],
            'appellation': record['appellation'],
            'sub_region': record['sub_region'],
            'region': record['region'],
            'country': record['country'],
            'producer': record['producer'],
            'tastings': tastings_list,
            'link': '/wine/{}'.format(record['id'])
        },
        'related_content': {
            'similar_wines': wines_list
        }
    })

def author_page(self, page_num):
    records = self.broker.get_author_page(page_num)
    if not records:
        return self.http_not_found_json()
    num_pages = self.broker.get_num_author_pages()
    authors_list = []
    for author in records:
        authors_list.append({
            'title': author['name'],
            'link': '/author/{}'.format(author['id'])
        })
    return self.http_success_json({
        'type': 'list',
        'self': {
            'title': 'Authors, Page {}'.format(page_num),
            'link': '/authors/{}'.format(page_num)
        },
        'list': authors_list
    })

def author(self, author_id):
    author_id = int(author_id)
    record = self.broker.get_author(author_id)
    if not record:
        return self.http_not_found_json()
```

```
        tastings_list = []
        if 'tastings' in record:
            for tasting in record['tastings']:
                tastings_list.append({
                    'rating': tasting['rating'],
                    'notes': tasting['notes'],
                    'tasting_date': tasting['tasting_date'],
                    'wine': {
                        'title': '{} {}'.format(tasting['wine'], tasting['vintage']),
                        'link': '/wine/{}'.format(tasting['wine_id'])
                    }
                })
        wines = self.recommender.wines_for_author(author_id)
        wines_list = []
        for wine in wines:
            wines_list.append({
                'title': '{} {}'.format(wine['name'], wine['vintage']),
                'link': '/wine/{}'.format(wine['id'])
            })
        authors = self.recommender.authors_for_author(author_id)
        authors_list = []
        for author in authors:
            authors_list.append({
                'title': author['name'],
                'link': '/author/{}'.format(author['id'])
            })
        return self.http_success_json({
            'type': 'author',
            'self': {
                'title': record['name'],
                'name': record['name'],
                'tastings': tastings_list,
                'link': '/author/{}'.format(record['id'])
            },
            'related_content': {
                'recommended_wines': wines_list,
                'similar_authors': authors_list
            }
        })
```

F.3 Recommender

F.3.1 Dependencies

These are all the dependencies loaded up for the recommender module

```
#!python

# I load a whole load of dependencies at the top here.
# Obviously there is overhead involved in calling in all this stuff
# but for the sake manageability I'm putting it all up here ;-)

# general
from __future__ import division
import os
import collections
import json
import time
import random
import math

# text/language processing
import re
import nltk
from nltk.corpus import stopwords

# maths!
import numpy
from numpy import random
import scipy
```

```
# recsys-svd
import recsys.algorithm
from recsys.algorithm.factorize import SVD
from recsys.evaluation.prediction import MAE, RMSE

# Sommelier libs
from broker import SommelierBroker
```

F.3.2 SommelierRecommenderBase

```
# Abstract / base class to hold methods shared between all recommenders
class SommelierRecommenderBase:


    def __init__(self, b=SommelierBroker()):
        self.broker = b

    def data_file_directory(self):
        return "".join([os.getcwd(), '/data/'])

    def file_location(self, filename):
        return "".join([self.data_file_directory(), filename])

    def save_json_file(self, filename, data):
        thefile = "".join([self.data_file_directory(), filename, '.json'])
        with open(thefile, 'w') as outfile:
            json.dump(data, outfile)

    def load_json_file(self, filename):
        print "".join(["Loading ", self.data_file_directory(), filename, '.json'])
        return json.loads(open("".join([self.data_file_directory(), filename])).read())

    # Preferences are returned as a dictionary of indexed lists of ratings, accompanied
    # by a list of column ids which correspond to the ratings lists
    #
    # preferences = {
    #   row_id: [ rating_1, rating_2, .. rating_N ]
    # }
    #
    # column_ids = [ itemid_1, itemid_2, .. itemid_N ]
    #
    # @returns ( {} preferences, [] item_ids )
    def preferences(self, tastings, row_key, column_key, rating_key='rating'):
        if len(tastings) == 0:
            # if there aren't any tastings then bail out...
            return {}, []
        preferences = {}
        column_data = {}
        column_ids = []
        for tasting in tastings:
            # tastings should be ordered by author and wine
            preferences.setdefault(tasting[row_key], [])
            column_data.setdefault(tasting[column_key], {})
            column_data[tasting[column_key]].setdefault(tasting[row_key], tasting[rating_key])
        for column in column_data:
            for item in preferences.keys():
                if item in column_data[column].keys():
                    preferences[item].append(column_data[column][item])
                else:
                    preferences[item].append(0)
                if column not in column_ids:
                    column_ids.append(column)
        return preferences, column_ids

    # Open a given file which is in Movielens format
    # and convert to tastings as used by Sommelier
    def load_movielens_to_tastings(self, filename):
        tastings = []
        print "Loading Movielens data... (this may take a while)"
        for line in open("".join([self.data_file_directory(), filename]), 'r'):
            tastings.append(self.movielens_line_to_tasting(line))
        print "Done."
        return tastings
```

```
# Converts a live from a Movielens file (as a string)
# to a tasting dict. This is for use in benchmarking and
# evaluation so that Movielens data can be used to
# test the system. Movielens file format:
# UserId::ItemId::Rating::Timestamp
def movielens_line_to_tasting(self, line):
    if line.find("::") != -1:
        # this is double-colon separated
        user_id, item_id, rating, timestamp = line.split("::")
    elif line.find("\t") != -1:
        # presumably this is tab-separated
        user_id, item_id, rating, timestamp = line.split('\t')
    else:
        raise Exception("Cannot decode input string: {}".format(line))
    tasting = {
        'author_id': int(user_id),
        'wine_id': int(item_id),
        'rating': int(rating),
        'tasting_date': str(time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(int(timestamp))))
    }
    return tasting

def tastings_to_movielens_format(self, tastings, separator="::"):
    # make a list of strings which will be the lines in our
    # Movielens format data file. The format is:
    # AuthorId::WineId::Rating::Timestamp
    lines = []
    for tasting in tastings:
        # if the date is not valid set to 0, otherwise convert to Unix epoch
        date = '0'
        if 'tasting_date' in tasting and tasting['tasting_date'] != '0000-00-00 00:00:00':
            date = str(time.mktime(time.strptime(tasting['tasting_date'], "%Y-%m-%d %H:%M:%S"))).encode('utf-8')
        lines.append("".join([
            str(tasting['author_id']).encode('utf-8'), separator,
            str(tasting['wine_id']).encode('utf-8'), separator,
            str(tasting['rating']).encode('utf-8'), separator,
            date, '']))
    return lines

def pearson_r(self, preferences, key_a, key_b):
    # get mutually rated items into two lists of ratings
    items_a = []
    items_b = []
    # iterate over all items
    for i in range(0, len(preferences[key_a])):
        # if both a and b have rated the item then add it to their item lists
        if preferences[key_a][i] != 0 and preferences[key_b][i] != 0:
            items_a.append(preferences[key_a][i])
            items_b.append(preferences[key_b][i])
    if len(set(items_a)) <= 1 or len(set(items_b)) <= 1:
        # one of the sets has standard deviation of 0 so pearson_r
        # will be NaN - in this case return 0.0
        return 0.0
    if len(items_a) < 3:
        # no items in common = no correlation
        # less than 3 items in common = problematic for comparison
        return 0.0
    # we now how two lists of ratings for the same items
    # these can be fed into the scipy.stats pearson r method
    # we only want the first value [0] from the method as we
    # don't need the 2-tail p value.
    return scipy.stats.pearsonr(items_a, items_b)[0]

# Calculates mean absolute error for each row in the matrix
# using the MAE() class from python-recsys
# python-recsys evaluation documentation:
# http://ocelma.net/software/python-recsys/build/html/evaluation.html
def evaluate_matrices_mae(self, original_matrix, imputed_matrix):
    return self.evaluate_matrices(original_matrix, imputed_matrix, evaluator=MAE())

def evaluate_matrices_rmse(self, original_matrix, imputed_matrix):
    return self.evaluate_matrices(original_matrix, imputed_matrix, evaluator=RMSE())
```

```
    def recsys_evaluate_matrices(self, original_matrix, imputed_matrix, evaluator=MAE()):
        total_error = 0
        total_rows = 0
        errors = ()
        for row_i, row in enumerate(original_matrix):
            # For each row build its list of non-zero values
            # Build a corresponding list of values for the imputed matrix
            row_values = []
            imputed_values = []
            for col_i, col in enumerate(row):
                if row[col_i] > 0:
                    row_values.append(col)
                    imputed_values.append(imputed_matrix[row_i][col_i])
            if len(row_values) == 0 or len(imputed_values) == 0:
                continue
            evaluator.load_ground_truth(row_values)
            evaluator.load_test(imputed_values)
            row_error = evaluator.compute()
            errors = errors + (row_error, )
            total_error += row_error
            total_rows += 1
        mean_total_error = 0.0
        if total_rows > 0.0:
            mean_total_error = total_error / total_rows
        return errors, mean_total_error
```

F.3.3 SommelierPearsonCFRecommender

```
# This class implements recommendations largely based on the
# basic user-user, user-item and item-item methods
# detailed by Segaran (2007, Ch.2)
class SommelierPearsonCFRecommender(SommelierRecommenderBase):

    def __init__(self, b=SommelierBroker()):
        self.broker = b

    # using a weighted average
    def wines_for_author(self, author_id, max_items=5):
        author_id = int(author_id)
        preferences, wine_ids = self.author_preferences(author_id)
        rankings = self.sorted_rankings(author_id, preferences, max_items)
        recommended_wine_ids = [ wine_ids[i] for r, i in rankings ]
        if len(recommended_wine_ids) > 0:
            return self.broker.get_wines_by_id(recommended_wine_ids)
        # if we couldn't find any wines to recommend, return empty
        return []


    def authors_for_author(self, author_id, max_items=5):
        author_id = int(author_id)
        preferences, wine_ids = self.author_preferences(author_id)
        return self.sorted_similarities(author_id, preferences)

    def wines_for_wine(self, wine_id, max_items=5):
        wine_id = int(wine_id)
        preferences, wine_ids = self.wine_preferences(wine_id)
        rankings = self.sorted_rankings(wine_id, preferences, max_items=5)
        recommended_wine_ids = [ wine_ids[i] for r, i in rankings ]
        if len(recommended_wine_ids) > 0:
            return self.broker.get_wines_by_id(recommended_wine_ids)
        # if we couldn't find any wines to recommend, return empty
        return []

    def author_preferences(self, author_id):
        tastings = self.broker.get_comparable_author_tastings(author_id)
        return self.preferences(tastings, 'author_id', 'wine_id')

    def wine_preferences(self, wine_id):
        tastings = self.broker.get_comparable_wine_tastings(wine_id)
        return self.preferences(tastings, 'wine_id', 'author_id')

    def sorted_similarities(self, subject_id, preferences, max_items=5):
```

```
        similarities = []
        for item_id in preferences.keys():
            if item_id == subject_id:
                # we don't need to compare our subject with itself
                continue
            similarity = self.pearson_r(preferences, subject_id, item_id)
            # only return positive correlations
            if similarity > 0:
                similarities.append((item_id, similarity))
        sorted_similarities = sorted(similarities, key=lambda sim: sim[1], reverse=True)
        return sorted_similarities[0:max_items]

    def sorted_rankings(self, subject_id, preferences, max_items=5):
        totals = {}
        similarity_sums = {}
        if subject_id not in preferences:
            # we can't recommend for this subject
            return []
        subject_preferences = preferences[subject_id]
        for other in preferences.keys():
            if other == subject_id:
                # don't compare subject to themself
                continue
            similarity = self.pearson_r(preferences, subject_id, other)
            if similarity <= 0:
                # no similarity is a waste of time
                continue
            for i in range(0, len(preferences[other])):
                if preferences[subject_id][i] == 0 and preferences[other][i] > 0:
                    totals.setdefault(i, 0)
                    totals[i] += preferences[other][i] * similarity
                    similarity_sums.setdefault(i, 0)
                    similarity_sums[i] += similarity
        if not totals:
            # there are no recommendations to be made :-(
            return ()
        rankings = [(total/similarity_sums[item], item) for item, total in totals.items()]
        return sorted(rankings, key=lambda sim: sim[0], reverse=True )[0:max_items]
```

F.3.4 SommelierRecsysSVDRecommender

```
# Implements SommelierRecommender using
# python-recsys SVD library to make recommendations
class SommelierRecsysSVDRecommender(SommelierRecommenderBase):

    # filename for raw tasting data in format used by MovieLens
    # format (rows): UserId::ItemId::Rating::UnixTime
    tastings_movielens_format = 'tastings_movielens_format'

    # filename for python-recsys zip outfile
    tastings_recsys_svd = 'recsys_svd_data'

    def __init__(self, b=SommelierBroker()):
        self.broker = b

    def wines_for_author(self, author_id):
        svd = self.load_recsys_svd()
        # there may not be recommendations for this author, which would
        # raise a KeyError. We don't want a KeyError, an empty list is fine!
        try:
            recommendations = svd.recommend(int(author_id), is_row=False)
        except:
            recommendations = []
        wine_ids = []
        for recommendation in recommendations:
            wine_ids.append(recommendation[0])
        recommended_wines = []
        if len(wine_ids) > 0:
            wines = self.broker.get_wines_by_id(wine_ids)
            for wine in wines:
                recommended_wines.append({
                    'name': wine['name'],
                    'vintage': wine['vintage'],
```

```
                'id': wine['id']
            })
        return recommended_wines

    def authors_for_author(self, author_id):
        return []

    def wines_for_wine(self, wine_id):
        svd = self.load_recsys_svd()
        # there may not be recommendations for this author, which would
        # raise a KeyError. We don't want a KeyError, an empty list is fine!
        try:
            # get similar wines, but pop() the first item off as it is the current wine
            recommendations = svd.similar(int(wine_id))
            recommendations.pop(0)
        except:
            recommendations = []
        wine_ids = []
        for recommendation in recommendations:
            wine_ids.append(recommendation[0])
        similar_wines = []
        if len(wine_ids) > 0:
            wines = self.broker.get_wines_by_id(wine_ids)
            for wine in wines:
                similar_wines.append({
                    'name': wine['name'],
                    'vintage': wine['vintage'],
                    'id': wine['id']
                })
        return similar_wines

    # Decomposes the matrix of tastings data using recsys-svd's SVD()
    # and saves the output matices etc. to a zip file using SVD()'s built-in method
    def impute_to_file(self, tastings, k=100, min_values=2, verbose=True):
        # create a data file in Movielens format with the tastings data
        self.save_tastings_to_movielens_format_file(tastings)
        # for logging/testing purposes we may like this verbose
        if verbose:
            recsys.algorithm.VERBOSE = True
        svd = SVD()
        # load source data, perform SVD, save to zip file
        source_file = self.file_location(self.tastings_movielens_format)
        svd.load_data(filename=source_file, sep='::', format={'col':0, 'row':1, 'value':2, 'ids': int})
        outfile = self.file_location(self.tastings_recsys_svd)
        svd.compute(k=k, min_values=min_values, pre_normalize=None, mean_center=True, post_normalize=True, savefile=outfi
        return svd

    def save_tastings_to_movielens_format_file(self, tastings):
        movielens_lines = self.tastings_to_movielens_format(tastings)
        outfile = self.file_location(self.tastings_movielens_format)
        with open(outfile, 'w') as datafile:
            for line in movielens_lines:
                datafile.write(line)

    # loads a recsys-svd data file stored to disk by the impute_to_file() method
    def load_recsys_svd(self):
        from recsys.algorithm.factorize import SVD
        svd = []
        # if there's an svd file, load it - otherwise we're out of luck as
        # we don't want to build these matrices at runtime!
        tastings_svd_file = self.file_location(self.tastings_recsys_svd)
        if os.path.isfile(tastings_svd_file):
            svd = SVD(tastings_svd_file)
        # return the recsys SVD object, ready to make some recommendations...
        return svd
```

F.3.5 SommelierYeungMFRecommender

```
# Implements SommelierRecommender using
# Albert Yeung's example Matrix Factorization code
# combined with basic CF techniques outlined in
# Segaran's "Collective Intelligence" (2007, Ch. 2)
class SommelierYeungMFRecommender(SommelierRecommenderBase):
```

```
# filename for user/item matrix in lists format
# format: [[1,2,3][4,5,6]..]
original_matrix = "tastings_yeung_matrix"

# filename for user/item matrix in lists format
# format: [[1,2,3][4,5,6]..]
test_data_matrix = "tastings_yeung_test_matrix_{}percent"

# filename for factored and reconstructed matrix, using Yeung's simple MF algorithm
# format: [[1,2,3][4,5,6]..]
reconstructed_matrix = "reconstructed_yeung_matrix_k{}_steps{}"

factors_matrix = "imputed_yeung_factors_k{}_steps{}"

weights_matrix = "imputed_yeung_weights_k{}_steps{}"

multiple_factorization_metas = "multiple_factorization_metas_k{}_steps{}"

def __init__(self, b=SommelierBroker()):
    self.broker = b

def wines_for_author(self, author_id):
    return []

def authors_for_author(self, author_id):
    return []

def wines_for_wine(self, item_id):
    return []

def impute_to_file(self, tastings):
    matrix = self.generate_lists_ui_matrix(tastings, self.original_matrix)
    factored_matrix = self.yeung_factor_matrix(matrix, steps=1000, factors=10, evaluator='MAE')[0]
    return []

# load the sparse ui matrix from disk
def load_lists_ui_matrix(self):
    matrix_data = self.load_json_file(self.original_matrix)
    matrix = matrix_data["ratings"]
    author_ids = matrix_data["author_ids"]
    wine_ids = matrix_data["wine_ids"]
    return matrix, author_ids, wine_ids

def generate_lists_ui_matrix(self, tastings={}, outfile=''):
    if not tastings:
        # get all the tastings from the database
        tastings = self.broker.get_tastings()
    # make a dict with an entry for each author, with wines and ratings:
    # { author: { wine_id: rating, wine_id: rating, ... } ... }
    author_ratings = {}
    wine_ids = []
    for tasting in tastings:
        author_ratings.setdefault(tasting['author_id'], {})
        author_ratings[tasting['author_id']][tasting['wine_id']] = tasting['rating']
        if tasting['wine_id'] not in wine_ids:
            wine_ids.append(tasting['wine_id'])
    # for each author iterate over wines and make a tuple with ratings for each wine, or 0.0
    lists_matrix = []
    author_ids = []
    for author_id in author_ratings:
        author = author_ratings[author_id]
        author_ids.append(author_id)
        author_wine_list = []
        for wine_id in wine_ids:
            # if there is a key in the author dict for this wine, take the rating from that
            if wine_id in author:
                author_wine_list.append(float(author[wine_id]))
            # otherwise append a 0.0
            else:
                author_wine_list.append(0.0)
        lists_matrix.append(author_wine_list)
    if outfile:
```

```
        self.save_json_file(outfile, { "ratings": lists_matrix, "author_ids": author_ids, "wine_ids": wine_ids })
    return lists_matrix, author_ids, wine_ids

# Copied from Albert Yeung: http://www.quuxlabs.com/wp-content/uploads/2010/09/mf.py_.txt
# This method factors the given matrix into
def yeung_factor_matrix(self, matrix=[], steps=5000, factors=10, evaluator=MAE(), verbose=True):
    if not matrix:
        matrix = self.load_lists_ui_matrix()
    R = numpy.array(matrix)
    N = len(R)
    M = len(R[0])
    K = factors
    P = numpy.random.rand(N, K)
    Q = numpy.random.rand(M, K)
    nP, nQ, e = self.yeung_matrix_factorization(R, P, Q, K, steps, verbose=verbose)
    if verbose: print "Final error: {}".format(e)
    self.save_json_file(self.factors_matrix.format(K, steps), nP.tolist())
    self.save_json_file(self.weights_matrix.format(K, steps), nQ.tolist())
    nR = numpy.dot(nP, nQ.T)
    if verbose: print "Saving to JSON file..."
    self.save_json_file(self.reconstructed_matrix.format(K, steps), nR.tolist())
    if verbose: print "Evaluation using {}...".format(evaluator.__class__.__name__)
    errors, mean_total_error = self.recsys_evaluate_matrices(R, nR, evaluator)
    if verbose: print "Mean total error: {}".format(mean_total_error)
    return nR, nP, nQ, errors, mean_total_error

# Copied from Albert Yeung: http://www.quuxlabs.com/wp-content/uploads/2010/09/mf.py_.txt
def yeung_matrix_factorization(self, R, P, Q, K, steps=5000, alpha=0.0002, beta=0.02, verbose=True):
    if verbose:
        print "Matrix Factorization"
        print "Steps: {}".format(steps)
        print "Factors: {}".format(K)
    Q = Q.T
    s = 0
    for step in xrange(steps):
        s+=1
        if verbose: print "Step {} / {}".format(s,steps)
        for i in xrange(len(R)):
            for j in xrange(len(R[i])):
                if R[i][j] > 0:
                    eij = R[i][j] - numpy.dot(P[i,:],Q[:,j])
                    for k in xrange(K):
                        P[i][k] = P[i][k] + alpha * (2 * eij * Q[k][j] - beta * P[i][k])
                        Q[k][j] = Q[k][j] + alpha * (2 * eij * P[i][k] - beta * Q[k][j])
        eR = numpy.dot(P,Q)
        e = 0
        for i in xrange(len(R)):
            for j in xrange(len(R[i])):
                if R[i][j] > 0:
                    e = e + pow(R[i][j] - numpy.dot(P[i,:],Q[:,j]), 2)
                    for k in xrange(K):
                        e = e + (beta/2) * ( pow(P[i][k],2) + pow(Q[k][j],2) )
        if verbose: print "Error: {}".format(e)
        if e < 0.001:
            break
    return P, Q.T, e

# Will run any number of Yeung factorizations of a matrix, iterating over
# a list of configuration argument dicts to be passed to the factorization method
def multiple_factorizations(self, matrix, config_args):
    sparsity = self.sparsity_percent(matrix)
    print "Sparsity of matrix for multiple factorizations: {}%".format(sparsity)
    metas = []
    for args in config_args:
        start = int(time.time())
        fm = self.yeung_factor_matrix(matrix, **args)
        end = int(time.time())
        # get metas from result
        ue = fm[3]
        te = fm[4]
        t = end - start
        metas.append({"args": args, "user_errors": ue, "total_errors": te, "execution_time_seconds": t})
        self.save_json_file(self.multiple_factorization_metas.format(args["factors"], args["steps"]), metas)
```

```
    return metas

def predict_rating(self, imputed_matrix, authors, wines, author_id, wine_id):
    if author_id in authors:
        author_idx = authors.index(author_id)
        if wine_id in wines:
            wine_idx = wines.index(wine_id)
            if author_idx < len(imputed_matrix):
                if wine_idx < len(imputed_matrix[author_idx]):
                    return imputed_matrix[author_idx][wine_idx]
                raise Exception("Wine index out of bounds for matrix")
            raise Exception("Author index out of bounds for matrix")
        raise Exception("Wine index not found for imputed matrix")
    raise Exception("Author index not found for imputed matrix")

# Get all tastings, randomly split into train and test portions
# Generate a matrix for the test portion of the data
# Factor this matrix using the **config_args passed into the method
# Iterate over the training portion of the data, checking the
# real ratings against those imputed by the factorization
# Record MAE for each author, standard deviation of error for
# each author, total MAE, total standard deviation of error,
# and finally a normalised MAE for good measure...
def split_data_evaluation(self, config_args, matrix_file=[], tastings=[], percent_train=80):
    print "Test/train split: {}/{}".format(percent_train, 100-percent_train)
    print "Randomly splitting tastings for testing..."
    train_tastings, test_tastings = self.split_train_test_tastings(tastings, percent_train)
    print "Generating matrix for testing..."
    train_matrix, author_ids, wine_ids = self.generate_lists_ui_matrix(train_tastings, self.test_data_matrix)
    print "num authors {}".format(len(train_matrix))
    print "num items {}".format(len(train_matrix[0]))
    for args in config_args:
        print "Evaluation for args: {}".format(args)
        imputed_matrix = self.yeung_factor_matrix(train_matrix, **args)[0]
        total_error = 0.0
        num_tastings = 0
        errors = []
        author_errors = {}
        missing_authors = 0
        missing_wines = 0
        for tasting in test_tastings:
            if tasting["author_id"] not in author_ids:
                # there were no items for this author in the test data
                missing_authors +=1
                continue
            if tasting["wine_id"] not in wine_ids:
                # there were no items for this wine in the test data
                missing_wines += 1
                continue
            prediction = self.predict_rating(imputed_matrix, author_ids, wine_ids, tasting["author_id"], tasting["wir
            rating = float(tasting['rating'])
            error = abs(rating - prediction)
            author_errors.setdefault(tasting['author_id'], [])
            author_errors[tasting['author_id']].append(error)
            errors.append(error)
            total_error += error
            num_tastings += 1
        print "Missing authors: {}, missing wines: {}".format(missing_authors, missing_wines)
        author_stds = {}
        for author_id in author_errors:
            # author standard deviation
            author_stds.setdefault(author_id, 0.0)
            author_stds[author_id] = numpy.std(author_errors[author_id])
        mae = total_error / num_tastings
        # we can get the mae as a normalised value (between 0 and 1) by dividing by the difference between the
        # highest and lowest rating which we know is (5 - 0) = 5
        nmae = mae / 5
        # total standard deviation
        total_std = numpy.std(errors)
        print "NMAE {}".format(nmae)
        print "MAE {}".format(mae)
        print "Total SD {}".format(total_std)
        print "Author SDs {}".format(author_stds)
```

```
    def split_data_evaluate_movielens_file(self, filepath, config_args, percent_train=80):
        tastings = self.load_movielens_to_tastings(filepath)
        print "Number of tastings: {}".format(len(tastings))
        self.split_data_evaluation(config_args, tastings=tastings, percent_train=percent_train)

    def split_train_test_tastings(self, tastings, percent_train=80):
        if percent_train > 100:
            raise Exception("percent_test must be 100 or less")
        num_tastings = len(tastings)
        num_train = math.ceil(num_tastings * ( percent_train / 100 ))
        # now create an array of length num_tastings, with num_test amount of 1s
        # randomly distributed within it
        ones = numpy.ones((num_train)).tolist()
        zeros = numpy.zeros((num_tastings - num_train)).tolist()
        ones_and_zeros = ones + zeros
        random.shuffle(ones_and_zeros)
        test_tastings = []
        train_tastings = []
        for tasting in tastings:
            if ones_and_zeros.pop() == 1:
                train_tastings.append(tasting)
            else:
                test_tastings.append(tasting)
        return train_tastings, test_tastings

    def sparsity_percent(self, matrix):
        zero_count = 0.0
        total_count = 0.0
        for row in matrix:
            for col in row:
                total_count += 1.0
                if col == 0.0:
                    zero_count += 1.0
        return ( zero_count / total_count ) * 100.0
```

F.3.6 SommelierTextMFRecommender

```
# Implements SommelierRecommender performing
# matrix decomposition based around word frequency to
# generate topics which can be used to predict similarities
# between wines based on the language used about them, and
# between authors based on the language they use. Similar to
# the techniques outlined by Segaran in "Collective
# Intelligence" (2007, Ch. 10), but applied in a different
# context.
#
## UNIMPLEMENTED: THIS CLASS DOES NOT WORK ... YET
class SommelierTextMFRecommender(SommelierYeungMFRecommender):

    # filename for user/item matrix in lists format
    # format: [[1,2,3][4,5,6]..]
    original_matrix = "tastings_text_mf_matrix"

    # filename for factored and reconstructed matrix, using Yeung's simple MF algorithm
    # format: [[1,2,3][4,5,6]..]
    reconstructed_matrix = "reconstructed_text_mf_matrix"

    factors_matrix = "imputed_text_mf_factors"

    weights_matrix = "imputed_text_mf_weights"

    def __init__(self, b=SommelierBroker()):
        self.broker = b

    def wines_for_author(self, author_id):
        return []

    def authors_for_author(self, author_id):
        return []

    def wines_for_wine(self, item_id):
        return []
```

```
    def impute_to_file(self, tastings, steps=5000):
        matrix = self.get_tastings_word_matrix(tastings)
        self.save_json_file(self.original_matrix, matrix)
        self.yeung_factor_matrix(matrix=matrix, steps=steps)
        return matrix

    # one row for each tasting, one column for each word
    # with counts of occurrences of the word in the tasting note
    def get_tastings_word_matrix(self, tastings):
        words = self.get_words(tastings)
        rows = []
        for t in tastings:
            row = []
            t_words = re.split('\W+', t['notes'])
            for w in words:
                row.append(t_words.count(w))
            rows.append(row)
        return rows

    def get_words(self, tastings):
        words = []
        dist = self.tastings_frequency_distribution(tastings, min_values=4)
        for w in dist:
            words.append(w)
        return words

    # gets all words metioned >= min_values times, excluding stopwords
    def tastings_frequency_distribution(self, tastings, min_values=4):
        words = []
        for tasting in tastings:
            words += re.split('\W+', tasting['notes'])
        filtered_words = [w.lower() for w in words if not w in stopwords.words('english')]
        dist = nltk.FreqDist(filtered_words)
        words = dist.samples()
        for w in words:
            if dist[w] < min_values:
                dist.pop(w)
        return dist
```

F.3.7 SommelierRecommender

```
# Facade class for recommenders. May have specific recommender implementation
# injected into it or depend on default defined in its signature.
class SommelierRecommender:

    def __init__(self, b=SommelierBroker(), r=SommelierPearsonCFRecommender()):
        self.broker = b
        self.recommender = r

    def wines_for_author(self, author_id):
        return self.recommender.wines_for_author(author_id)

    def authors_for_author(self, author_id):
        return self.recommender.authors_for_author(author_id)

    def wines_for_wine(self, wine_id):
        return self.recommender.wines_for_wine(wine_id)
```

F.4 Sommelier Broker

```
#!python
import math
from dbconnector import SommelierDbConnector

class SommelierBroker:

    tastings_query = """
        SELECT *
        FROM tasting t
        WHERE author_id <> 0
```

```
"""
wine_ids_query = """
    SELECT id
    FROM wine w
    ORDER BY id ASC
    """
wines_by_id_query = """
    SELECT *
    FROM wine w
    WHERE w.id
    IN ({})
"""
wines_query = """
    SELECT *
    FROM wine w
    ORDER BY id ASC
    """
wine_page_query = """
    SELECT *
    FROM wine w
    LIMIT {} OFFSET {}
    """
wine_query  = """
    SELECT w.*, t.*, a.name AS author FROM wine w
    LEFT JOIN tasting t ON w.id = t.wine_id
    LEFT JOIN author a ON t.author_id = a.id
    WHERE w.id = {}
    """
wine_count_query = """
    SELECT COUNT(*) AS count FROM wine
    """
authors_query = """
    SELECT *
    FROM author a
    ORDER BY id ASC
    """
author_ids_query = """
    SELECT id
    FROM author a
    """
author_page_query = """
    SELECT *
    FROM author a
    LIMIT {} OFFSET {}
    """
author_query  = """
    SELECT a.*, t.*, w.name as wine FROM author a
    JOIN tasting t ON t.author_id = a.id
    JOIN wine w ON t.wine_id = w.id
    WHERE a.id = {}
    """
author_count_query = """
    SELECT COUNT(*) AS count FROM author a
    """
# scalability issues w/ multiple sub-queries ?
comparable_author_tastings_query = """
    SELECT t.*, a.*
    FROM tasting t
    JOIN author a ON t.author_id = a.id
    WHERE t.author_id <> 0
    AND t.author_id IN (
        SELECT DISTINCT t2.author_id
        FROM tasting t2
        WHERE t2.wine_id IN (
            SELECT t3.wine_id
            FROM tasting t3
            WHERE t3.author_id = {}
        )
    )
    ORDER BY t.author_id ASC, t.wine_id ASC
"""
# scalability issues w/ multiple sub-queries ?
comparable_wine_tastings_query = """
```

```
    SELECT t.*, a.*
    FROM tasting t
    JOIN author a ON t.author_id = a.id
    AND t.wine_id IN (
        SELECT DISTINCT t2.wine_id
        FROM tasting t2
        WHERE t2.author_id IN (
            SELECT t3.author_id
            FROM tasting t3
            WHERE t3.wine_id = {}
        )
    )
    ORDER BY t.wine_id ASC, t.author_id ASC
"""
page_size = 50

def __init__(self, db=SommelierDbConnector()):
    self.db = db

def get_tastings(self):
    self.db.execute(self.tastings_query)
    return self.db.fetch_all()

def get_wines(self):
    self.db.execute(self.wines_query)
    return self.db.fetch_all()

def get_wine_ids(self):
    self.db.execute(self.wine_ids_query)
    return self.db.fetch_all()

def get_wines_by_id(self, wine_ids):
    self.db.execute(self.wines_by_id_query.format(",".join(map(str, wine_ids))))
    return self.db.fetch_all()

def get_wine_page(self, pagenum=1):
    pageparams = self.page_size, self.page_size * (pagenum - 1)
    self.db.execute(self.wine_page_query.format(*pageparams))
    return self.db.fetch_all()

def get_num_wine_pages(self):
    self.db.execute(self.wine_count_query)
    result = self.db.fetch_one()
    count = float( result['count'] );
    return int( math.ceil( count / self.page_size ) )

def get_wine(self, wine_id):
    self.db.execute(self.wine_query.format(wine_id))
    result = self.db.fetch_one()
    if result is None: return {}
    wine = {
        'name': result['name'],
        'vintage': result['vintage'],
        'grape_variety': result['grape_variety'],
        'appellation': result['appellation'],
        'sub_region': result['sub_region'],
        'region': result['region'],
        'country': result['country'],
        'producer': result['producer'],
        'tastings': []
    }
    if result['author'] is not None:
        wine['tastings'].append({
            'author': result['author'],
            'notes': result['notes'],
            'rating': result['rating'],
            'tasting_date': result['tasting_date']
        })
        results = self.db.fetch_all()
        for row in results:
            wine['tastings'].append({
                'author': row['author'],
                'notes': row['notes'],
```

```
                'rating': row['rating'],
                'tasting_date': row['tasting_date']
            })
        return wine

    def get_authors(self):
        self.db.execute(self.authors_query)
        return self.db.fetch_all()

    # returns a vector of ids
    def get_author_ids(self):
        self.db.execute(self.author_ids_query)
        results = self.db.fetch_all()
        ids = []
        for row in results:
            ids.append(row['id'])
        return ids

    def get_author_page(self, pagenum=1):
        pageparams = self.page_size, self.page_size * (pagenum - 1)
        self.db.execute(self.author_page_query.format(*pageparams))
        return self.db.fetch_all()

    def get_num_author_pages(self):
        self.db.execute(self.author_count_query)
        result = self.db.fetch_one()
        count = float( result['count'] );
        return int( math.ceil( count / self.page_size ) )

    def get_author(self, authorid):
        self.db.execute(self.author_query.format(authorid))
        result = self.db.fetch_one()
        if result is None: return {}
        author = {
            'name': result['name'],
            'tastings': []
        }
        if result['wine'] is not None:
            author['tastings'].append({
                'wine': result['wine'],
                'notes': result['notes'],
                'rating': result['rating']
            })
            results = self.db.fetch_all()
            for row in results:
                author['tastings'].append({
                    'wine': row['wine'],
                    'notes': row['notes'],
                    'rating': row['rating']
                })
        return author

    def get_comparable_author_tastings(self, author_id):
        self.db.execute(self.comparable_author_tastings_query.format(author_id))
        results = self.db.fetch_all()
        return results

    def get_comparable_wine_tastings(self, wine_id):
        self.db.execute(self.comparable_wine_tastings_query.format(wine_id))
        results = self.db.fetch_all()
        return results
```

G Tests

G.1 Unit Tests

G.1.1 Sommelier Tests

```
#!python
import unittest
from mock import Mock, MagicMock
import sommelier
```

```
from sommelier import Sommelier

# Tests for the sommelier class, making sure it returns the right content in the
# right format for each request, and that its auxiliary methods, like
# http_success_json() work properly

class SommelierTest(unittest.TestCase):

    def setUp(self):
        mock_broker = MagicMock()
        mock_broker.get_wine_page = MagicMock(return_value={'items':['item','item']})
        mock_broker.get_num_wine_pages = MagicMock(return_value=23)
        mock_broker.get_author_page = MagicMock(return_value=[{'name':'wine','vintage':1234,'id':123}])
        mock_broker.get_num_author_pages = MagicMock(return_value=23)
        mock_broker.get_wine = MagicMock(return_value=[{'name':'wine','vintage':1234,'id':123}])
        mock_broker.get_author = MagicMock(return_value=[{'name':'an author','id':123}])
        mock_recommender = MagicMock()
        mock_recommender.wines_for_wine = MagicMock(return_value=[{'name':'wine','vintage':1234,'id':123}])
        mock_recommender.wines_for_author = MagicMock(return_value=[{'name':'wine','vintage':1234,'id':123}])
        mock_recommender.authors_for_author = MagicMock(return_value=[{'name':'author name','id':123}])
        self.sommelier = Sommelier(b=mock_broker, r=mock_recommender)

    def test_http_success_json(self):
        # method takes any dict as input
        test_content = { 'test': 'test content' }
        # should convert that dict into a UTF-8 JSON string, returned as first item of tuple
        expected_content = u'{"test": "test content"}'
        # also expected to return a dict with HTTP status and MIME Type parameters
        # - these will be converted to keyed arguments to Response() later using **
        expected_keyed_args_dict = { 'status': 200, 'mimetype': 'application/json; charset=utf-8' }
        response_body, keyed_args_dict = self.sommelier.http_success_json(test_content)
        self.assertEqual(expected_content, response_body)
        self.assertEqual(expected_keyed_args_dict, keyed_args_dict)

    def test_wines_page(self):
        expected_response_body = u'{"num_pages": 23, "wines": {"items": ["item", "item"]}}'
        response_body, keyed_args_dict = self.sommelier.wine_page(1)
        self.assertEqual(expected_response_body, response_body)

    def test_wine(self):
        expected_response_body = u'{"recommendations": ["wine", "wine", "wine"], "wine": {"wine": "a wine"}}'
        response_body, keyed_args_dict = self.sommelier.wine(123)
        self.assertEqual(expected_response_body, response_body)

    def test_authors_page(self):
        expected_response_body = u'{"num_pages": 23, "authors": {"items": ["item", "item"]}}'
        response_body, keyed_args_dict = self.sommelier.author_page(1)
        self.assertEqual(expected_response_body, response_body)

    def test_author(self):
        expected_response_body = u'{"recommendations": {"authors": ["authors for author", "author", "author"], "wines": [
        response_body, keyed_args_dict = self.sommelier.author(123)
        self.assertEqual(expected_response_body, response_body)
```

G.1.2 Recommender Tests

```
#!python
import unittest
from mock import Mock, MagicMock

# import all our recommenders...
from recommender import SommelierRecommenderBase, SommelierPearsonCFRecommender, SommelierYeungMFRecommender, SommelierRe

# Tests for the sommelier recommender class
# This does not test some very small methods, such as
# those for saving and retrieving JSON files
class RecommenderTest(unittest.TestCase):

    dummy_tastings = [
        { "author_id": 1, "wine_id": 5, "rating": 9 },
        { "author_id": 2, "wine_id": 6, "rating": 10 },
        { "author_id": 3, "wine_id": 7, "rating": 11 },
```

```
    { "author_id": 4, "wine_id": 8, "rating": 12 }]

expected_preferences_1 = {
    1: [ 0, 9, 0, 0 ],
    2: [ 0, 0, 10, 0 ],
    3: [ 0, 0, 0, 11 ],
    4: [ 12, 0, 0, 0 ]
    }, [8, 5, 6, 7]

expected_preferences_2 = {
    5: [ 9, 0, 0, 0 ],
    6: [ 0, 10, 0, 0 ],
    7: [ 0, 0, 11, 0 ],
    8: [ 0, 0, 0, 12 ]
    }, [1, 2, 3, 4]

pearson_r_test_preferences = {
    "row_1": [ 1, 2, 3, 4, 5 ],
    "row_2": [ 2, 3, 4, 5, 6 ],
    "row_3": [ 5, 4, 3, 2, 1 ],
    "row_4": [ 3, 3, 3, 3, 3 ],
    "row_5": [ 1, 2 ],
    "row_6": [ 3, 1 ]}

dummy_line_tab_separated = "1\t2\t3\t4"

dummy_line_colon_separated = "1::2::3::4"

dummy_line_invalid_separator = "1BAZ2BAZ3BAZ4"

dummy_line_invalid_values = "Should::NOT::be:Strings"

expected_tasting = {'author_id': 1, 'rating': 3, 'tasting_date': '1970-01-01 01:00:04', 'wine_id': 2}

expected_movielens_lines_colon = [ "1::5::9::0", "2::6::10::0", "3::7::11::0", "4::8::12::0" ]

expected_movielens_lines_tab = [ "1\t5\t9\t0", "2\t6\t10\t0", "3\t7\t11\t0", "4\t8\t12\t0" ]

dummy_matrix_a = [[1,1,1,1],[2,2,2,2],[3,3,3,3]]

dummy_matrix_b = [[1,1,1,1],[2,4,4,2],[3,3,3,1]]

dummy_matrix_c = [[0,2,0,2],[0,2,0,2],[2,2,2,2]]

expected_mae_a_b = ((0.0,1.0,0.5),0.5)

expected_mae_a_c = ((1.0,1.0,1.0),1.0)

def setUp(self):
    mock_broker = MagicMock()
    self.recommender = SommelierRecommenderBase(b=mock_broker)

def test_pearson_r(self):
    # rows 1 and 2 have 100% positive correlation
    self.assertEqual(1.0, self.recommender.pearson_r(self.pearson_r_test_preferences, 'row_1', 'row_2'))

    # rows 1 and 3 have 100% negative correlation
    self.assertEqual(-1.0, self.recommender.pearson_r(self.pearson_r_test_preferences, 'row_1', 'row_3'))

    # row 4 has a standard deviation of 0, which we need to deal with to avoid a division by zero error
    # when the covariance is divided by the product of the standard deviations
    # in this case the method returns 0.0, on the basis that a user submitting the same rating for every
    # item is not expressing any preference at all, so a neutral similarity score is appropriate
    self.assertEqual(0.0, self.recommender.pearson_r(self.pearson_r_test_preferences, 'row_1', 'row_4'))

    # cases where there are two or less items for comparison are problematic for pearson_r; any two
    # lists with 2 items will always result in either 1.0 or -1.0. That is not a useful score, as
    # there may be no similarity in the ratings at all, so we return 0.0 if there are < 3 items
    self.assertEqual(0.0, self.recommender.pearson_r(self.pearson_r_test_preferences, 'row_5', 'row_6'))

def test_preferences(self):
    # preferences formatting for author rows / wine columns
    self.assertEqual(self.expected_preferences_1, self.recommender.preferences(self.dummy_tastings, 'author_id', 'wir
```

```
        # preferences formatting for wine rows / author columns
        self.assertEqual(self.expected_preferences_2, self.recommender.preferences(self.dummy_tastings, 'wine_id', 'autho

    # Movielens data can be encoded with either tabs or double colons (::), so test for both and neither...
    def test_movielens_line_to_tasting(self):
        self.assertEqual(self.expected_tasting, self.recommender.movielens_line_to_tasting(self.dummy_line_tab_separated)
        self.assertEqual(self.expected_tasting, self.recommender.movielens_line_to_tasting(self.dummy_line_colon_separate
        self.assertRaises(Exception, lambda _: self.recommender.movielens_line_to_tasting(self.dummy_line_invalid_separat
        self.assertRaises(Exception, lambda _: self.recommender.movielens_line_to_tasting(self.dummy_line_invalid_values)

    # tastings_to_movielens_format() should use double colon (::) separator by default
    def test_tastings_to_movielens_format(self):
        self.assertEqual(self.expected_movielens_lines_colon, self.recommender.tastings_to_movielens_format(self.dummy_ta
        self.assertEqual(self.expected_movielens_lines_colon, self.recommender.tastings_to_movielens_format(self.dummy_ta
        self.assertEqual(self.expected_movielens_lines_tab, self.recommender.tastings_to_movielens_format(self.dummy_tast

    def test_evaluate_matrices(self):
        self.assertEquals(self.expected_mae_a_b, self.recommender.recsys_evaluate_matrices(self.dummy_matrix_a, self.dumm
        self.assertEquals(self.expected_mae_a_c, self.recommender.recsys_evaluate_matrices(self.dummy_matrix_a, self.dumm

#
class PearsonCFRecommenderTest(unittest.TestCase):

    def setUp(self):
        mock_broker = MagicMock()
        self.recommender = SommelierPearsonCFRecommender(b=mock_broker)

    # test args for both sorted_rankings and sorted_similarities
    # these consist of an id and a dict of user preferences of the format: { id: [ list, of, ratings ] }
    # In this test data user 1 has 3 ratings in common with the others, the first three (0, 1 and 2).
    # Users 2, 3 and 4 have each rated the last two items (3 and 4), whereas user 1 has not.
    # Items 3 and 4 have been given identical rankings by users 2, 3 and 4 ...
    test_args = ( 1, { 1: [ 1, 2, 3, 0, 0 ], 2: [ 1, 2, 3, 3, 4 ], 3: [ 3, 2, 1, 3, 4 ], 4: [ 1, 2, 3, 3, 4 ] } )

    # ... so the rankings should be 4.0 for item 4 and 3.0 for item 3, with item 4 first in the list
    # as it has the higher rating
    sorted_rankings_expected = [(4.0, 4), (3.0, 3)]

    # covers recommender.sorted_rankings
    def test_sorted_rankings(self):
        self.assertEqual(self.sorted_rankings_expected, self.recommender.sorted_rankings(*self.test_args))

    # We can use the same arguments from above to test the sorted_similarities() method
    # this method will look at the pearson score for each of the users. The ratings in this
    # case are loaded so that users 2 and 4 have identical ratings as 1, so will be recommended
    # in that order. User 3 has inverse ratings to user 1, so they will score -1.0 and not
    # be returned by the method
    sorted_similarities_expected = [(2, 1.0), (4, 1.0)]

    # covers recommender.sorted_similarities
    def test_sorted_similarities(self):
        self.assertEqual(self.sorted_similarities_expected, self.recommender.sorted_similarities(*self.test_args))

class YeungMFRecommenderTest(unittest.TestCase):

    dummy_tastings = [
        { "author_id": 1, "wine_id": 1, "rating": 1 },
        { "author_id": 2, "wine_id": 2, "rating": 2 },
        { "author_id": 3, "wine_id": 3, "rating": 3 },
        { "author_id": 4, "wine_id": 4, "rating": 4 }]

    dummy_wine_ids = [
        { 'id': 1 },
        { 'id': 2 },
        { 'id': 3 },
        { 'id': 4 }]

    expected_lists_matrix = [
        [1.0, 0.0, 0.0, 0.0],
        [0.0, 2.0, 0.0, 0.0],
        [0.0, 0.0, 3.0, 0.0],
        [0.0, 0.0, 0.0, 4.0]]
```

```
    def setUp(self):
        mock_broker = MagicMock()
        mock_broker.get_tastings = MagicMock(return_value=self.dummy_tastings)
        mock_broker.get_wine_ids = MagicMock(return_value=self.dummy_wine_ids)
        self.recommender = SommelierYeungMFRecommender(b=mock_broker)

    def test_generate_lists_ui_matrix(self):
        generated_matrix, foo, bar = self.recommender.generate_lists_ui_matrix()
        self.assertEqual(self.expected_lists_matrix, generated_matrix)
```

G.1.3 Broker Tests

```
#!python
import unittest
from mock import Mock, MagicMock
from brokers import SommelierBroker

# Simple tests to make sure that the brokers call the db how we want them to, i.e.
# using and templating their queries as we expect

class BrokersTest(unittest.TestCase):

    def setUp(self):
        # instantiate broker for testing
        self.sommelier_broker = SommelierBroker()
        # create mock DB
        db = MagicMock()
        db.execute = MagicMock()
        db.fetch_all = MagicMock()
        db.fetch_one = MagicMock()
        # inject mock db into broker
        self.sommelier_broker.db = db

    def test_get_authors(self):
        authors = self.sommelier_broker.get_authors()
        expected_query = self.sommelier_broker.authors_query
        self.sommelier_broker.db.execute.asset_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_all.assert_called_once()

    def test_get_wines(self):
        authors = self.sommelier_broker.get_wines()
        expected_query = self.sommelier_broker.wines_query
        self.sommelier_broker.db.execute.asset_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_all.assert_called_once()

    def test_get_wine_page(self):
        page = self.sommelier_broker.get_wine_page(1)
        expected_query = self.sommelier_broker.wine_page_query.format(50, 0)
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_all.assert_called_once_with()

    def test_get_wine_num_pages(self):
        numpages = self.sommelier_broker.get_num_wine_pages()
        expected_query = self.sommelier_broker.wine_count_query
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_one.assert_called_once_with()

    def test_get_wine(self):
        wine = self.sommelier_broker.get_wine(123)
        expected_query = self.sommelier_broker.wine_query.format(123)
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_one.assert_called_once_with()

    def test_get_author_ids(self):
        ids = self.sommelier_broker.get_author_ids()
        expected_query = self.sommelier_broker.author_ids_query
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_all.assert_called_once_with()

    def test_get_author_page(self):
        page = self.sommelier_broker.get_author_page(1)
        expected_query = self.sommelier_broker.author_page_query.format(50, 0)
```

```
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_all.assert_called_once_with()

    def test_get_author_num_pages(self):
        numpages = self.sommelier_broker.get_num_author_pages()
        expected_query = self.sommelier_broker.author_count_query
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_one.assert_called_once_with()

    def test_get_author(self):
        author = self.sommelier_broker.get_author(123)
        expected_query = self.sommelier_broker.author_query.format(123)
        self.sommelier_broker.db.execute.assert_called_once_with(expected_query)
        self.sommelier_broker.db.fetch_one.assert_called_once_with()

if __name__ == '__main__':
    unittest.main()
```

G.2 MovieLens Test Runner

Script: eval_train_movielens.py

```
#!flask/bin/python
from src.recommender import SommelierYeungMFRecommender, SommelierRecommender
y = SommelierYeungMFRecommender()
y.split_data_evaluate_movielens_file('ml-100k/u.data', [
    {"steps":1,  "factors":10, "verbose":True},
    {"steps":2,  "factors":10, "verbose":True},
    {"steps":3,  "factors":10, "verbose":True},
    {"steps":4,  "factors":10, "verbose":True},
    {"steps":5,  "factors":10, "verbose":True},
    {"steps":6,  "factors":10, "verbose":True},
    {"steps":7,  "factors":10, "verbose":True},
    {"steps":8,  "factors":10, "verbose":True},
    {"steps":9,  "factors":10, "verbose":True},
    {"steps":10,  "factors":10, "verbose":True},
    {"steps":12,  "factors":10, "verbose":True},
    {"steps":14,  "factors":10, "verbose":True},
    {"steps":16,  "factors":10, "verbose":True},
], percent_train=80)

y.split_data_evaluate_movielens_file('ml-100k/u.data', [
    {"steps":1,  "factors":10, "verbose":True},
    {"steps":2,  "factors":10, "verbose":True},
    {"steps":3,  "factors":10, "verbose":True},
    {"steps":4,  "factors":10, "verbose":True},
    {"steps":5,  "factors":10, "verbose":True},
    {"steps":6,  "factors":10, "verbose":True},
    {"steps":7,  "factors":10, "verbose":True},
    {"steps":8,  "factors":10, "verbose":True},
    {"steps":9,  "factors":10, "verbose":True},
    {"steps":10,  "factors":10, "verbose":True},
    {"steps":12,  "factors":10, "verbose":True},
    {"steps":14,  "factors":10, "verbose":True},
    {"steps":16,  "factors":10, "verbose":True},
], percent_train=80)
```