Product Recommender Engine

Use Case: 'The MovieLens 10M dataset'

Aurélien-Morgan

2019-02-27

Harvard Executive Education – Data Science Professional Certification Program

– Capstone Project 1 / 2 - mandatory topic -

Table of Content

List of Figures

List of Tables

0 Preamble

Widespread across the internet, Recommender Agents have the ability to effectively learn user's interests and help these potential consumers find their way to an ideal product among the vast variety of an offer. Service providers, often marketplaces, rely on them heavily.

Their main intended goal is to convert a "browser" into a "buyer" via assisted navigation, ease of use and enhanced user experience. This all brings to higher user retention. Recommending systems also allow detecting opportunities for cross sales and/or suggesting package deals, increasing the value of the average shopping basket.

Recommending Systems are used by platforms, brands and advertisement agencies to generate improved overall user satisfaction, by making it more likely for each user to find quickly what he/she's interested in. They are already applied to a variety of industries, ranging from retail and accommodation booking to video streaming, news feeding and even social networks [1]. Recommendation systems have been around for more than 25 years [2]. Ever since, there have been many developments in the field, as for instance contemplated in this contemporary paper : [3]. As the **Microsoft Research Lab – Asia** puts it, many efforts are of course continuously being made worldwide to even further develop them [4] and those are watched closely by marketing and sales professionals eager to remain on top of the wolf pack. Recommender Engines have indeed proven to be very effective Marketing / Targeting tools.

There are inherent complexities induced by the usage of Recommending Systems. The exciting topic of the ubiquity of recommender systems has for instance been covered by Ms. Lusi Li in her PhD thesis [5] in the context of a dominant e-commerce platform that sells competing products from different manufacturers while simultaneously recommending a subset of these products. Ms. Li therein explores the intra-product-category competition as a function of products complementarity as well as the potential strategic price responses from manufacturers. Her observations are very interesting. You should check them out ! They can obviously be transposed to any industry.

1 Executive Summary

From 2006 to 2009, Netflix^{TM} sponsored a competition, offering a grand prize of $1,000,000 to the team that could take an offered dataset of over 100 million movie ratings and return recommendations that were 10% more accurate than those offered by the company's existing recommender system. The recommendation system relied on data consisting in a set of users having allotted a rating of up to 5 stars to each movie they had respectively watched. This competition energized the search for new and more accurate algorithms.

The Netflix^{TM} challenge winners were evaluated based on the ***R**oot **M**ean **S**quare **E**rror* of their model (***RMSE***)[1]. For a set of "N" ratings, if we define "y_u, i" as the rating for movie "i" by user "u" and denote our prediction with "$\hat{y}_u, i$", then it can be written as follows[2] :

$$RMSE = \sqrt{\frac{1}{N}\sum_{u,i}(y_{u,i} - \hat{y}_{u,i})^2}$$

On 21 September 2009, achieving a Test RMSE of 0.8567 stars, accounting for a 10.06% improvement over the original system, gave the victory to the "***BellKor's Pragmatic Chaos***" team.
A good summary of how the winning algorithm was put together can be read here : [6] and a more detailed explanation here : [7].

This will be the focus of the present short paper : recommendation systems which, in order to make specific recommendations to users, use ratings that users have given items. Since the original data is not publicly available [8], we here are going to work on data provided by the online movie recommender service "MovieLens" ('https://movielens.org/'). The entire latest 'MovieLens' dataset can be found online [9]. To make the computation a little easier, we will use the "10M" version of the 'MovieLens' dataset [10], which consists of *"only"* 10 million ratings applied to 10,000 movies by 70,000 users and which was released in January 2009[3].

The brief introduction to Recommender Engines in the herein report will adopt characteristics of the model developed by the Netflix^{TM} challenge winning team, such as their original data analysis strategies. We'll also use the RMSE as our loss function[4].
The Netflix^{TM} challenge winners implemented two general classes of models. One was similar to "k-nearest neighbors", where they found "movies" that were similar to each other and "users" that were similar to each other. The other one was based on an approach called "matrix factorization". We'll cover a little of all that in the present document.

[1] the square root of mean error of the ***L**east **S**quares **E**stimates* (LSE)
[2] variables that are estimates are marked with an hat "̂".
[3] [11]
[4] meaning, we'll also develop a model so that it minimizes the RMSE.

2 A dive into the 'hows'

We'll start by developing a **linear model** encompassing elements accounting for two different effects. These simple contributions to each individual user/movie rating consist in the fact that :

- on average, to a measurable extend, each given user rates above (or below) the average of users ; a.k.a the "**user effect**"
- on average, to a measurable extend, each given movie is rated above (or below) the average of movies ; a.k.a the "**movie effect**"

in that model, we'll employ **regularization** techniques (think "Bayesian adjustments") by optimizing a select group of parameters of our model against the "Penalized Least Square". in layman's terms, we'll penalize (correct against) overly high ratings for movies that have thus far only collected so few ratings that these can not be considered as that representative (and we'll do the same for users which have provided few ratings to date).

We'll then **model the residuals** of that linear model thru reliance over movies **k nearest neighbors** (knn). There are two important aspects to this statement :

- *Nearest neighbors* models work based on *distances/similarities*[5]. In our case here, for each user, we'll assign ratings to movies based on ratings of *similarly rated* movies. We'll measure "similarity" depending on how movies "compare" considering ratings distribution. For instance, to simplify, if movie "A" has been rated *3 stars* by users "1", "2", "3" and "4", then the knn algorithm will predict that same rating to user "5" if all 5 users have rated movie "B" the same (say, *4 stars*).
- *Residuals* are the *errors* of the linear model, i.e. the difference between the predictions of that model and the actual ratings. if a user rates a movie *3.5 stars* and if the linear model has predicted *3 stars*, then *0.5 stars* is the *residual* that will be passed on for the knn model to predict.

Together combined, these two models will constitute our first **stacked ensemble model**. What we're doing here is similar in spirit to *boosting* as it indeed consists in having a subsequent model focus its prediction on the error (shortcomings) of a prior model. But similarities end there[6], [7].

We'll also apply a **matrix factorization** technique to the residuals of the initial regularized linear model. Together combined, those two will constitute our second stacked ensemble model.
Matrix factorization is the mathematical orthogonal transformation of the matrix of ratings (with movies as columns and users as rows). We'll use *Primary Components Analysis (pca)* to decompose that matrix in order to achieve **Dimensionality reduction**. Pca allows us to summarize info relative to correlated users (think 'clusters') with a limited number of uncorrelated factors (called "Primary Components"). As an example, lets consider 2 highly correlated users. Those 'uncorrelated factors' ; those "Primary Components" can for instance be thought of as characteristics of clusters of movies such as for example :

- whether or not a user appreciates movies directed by Luc BESSON[15].
- whether or not a user appreciates movies starring Jean DUJARDiN[16].
- whether or not a user appreciates movies involving a mix of romance and action.
- etc.

Matrix factorization algorithms permits us to detect patterns between clusters of users and clusters of movies, explaining for the variation without losing too much info (only 'noise') in the true ratings.

[5] the similarity measure adopted by the Netflix^{TM} challenge winning team was cosine. For details see the knn treaining section.

[6] most boosting algorithms consist of iteratively learning weak supervised models and adding them to finally constitute a strong supervised model. When they are added, they are typically weighted in some way that is usually related to the weak learners' accuracy [12].

[7] model stacking introduced by Kaggle Competitions Grandmaster Marios MICHAILIDIS (KazAnova) : [13][14]

At this stage, we'll end up with the two below stacked model instances :

- Regularized linear + knn
- Regularized linear + pca

They will each provide predictions with a certain level of accuracy. To achieve an even better performance globally, we'll apply **bagging ensemble** with these two. Bagging[8] involves averaging predictions in order to counterbalance individual models weaknesses ; achieve better predictive power ; converge towards true ratings.

To train all these models, we'll employ k-fold cross-validation[17] as a way to keep us safe from over-fitting[18] (low bias, high variance). Overfitting is this tendency models can have to reproduce training data too well, follow even variation induced by randomness in the training data and have no predictive power.

---

[8] historically, Bagging came from the abbreviation of Bootstrap AGGregatING

3 For the little geek in you

Lets get a little technical. its gonna be brief and informative, I promise. I'll try to always remain engaging but if you have no interest in unveiling how to execute, even partly, the prediction procedure here depicted, you can simply skip over this section. There's much fun in reading the rest anyway !

The intent was to provide people with procedures and results that could fully be reproduced by anyone at home without any monetary investment and reliance over cloud computing, as long as they own a relatively modern PC.

The *R* and *Python* languages have that in common that objects they are using are hold in virtual memory. They both struggle at dealing natively with large ones. For that reason, the *movielens* dataset that we employ here, with its 10 million records, is large enough that it requires particular measures to be taken for it to be manageable on a single laptop.
At our disposal, a machine hosting a Windows 10 Operating System with an “8th Gen. intel® Core™ i7 processor”[9] (6 cores ; 12 logical processors), 256 GB of SSD storage on the same disk as the one hosting the O.S. and 16 GB of memory[10]. We won't speak of other characteristics such as GPU as it is not exploited in the use case that concerns us here.

At its peak, the algorithm put together to generate this report requires a little over 60 gigabytes of memory. To circumvent that constraint, we need to extend the memory capacity for R and this is feasible by first increasing the O.S. paging size as explained here : [19]&[20]. if you can, set the maximum memory allocation for R at up to 128 GB. In addition to the above, it must also be informed at the 'R session' level with the below line of code :

```
memory.limit( size = 128000 )
```

On such a large dataset, in order to keep training duration at a manageable level on a laptop as the one depicted above, one has to draw on multi-threading/parallel processing whenever possible too. For matrix arithmetics, a first great level of gain can be achieved via switching from *standard R* to *Microsoft R Open* and *intel MKL*[21][22][23][24][25].
It will however not save the day on all circumstances. When looking for performance optimization in R, options generally are 'vectorization' versus 'apply functions' and/or 'for loops' [26]. *Parallel 'foreach' looping*[11] has for instance been investigated in the context of knn training which was by far the most computationally demanding part of the entire algorithm. However benefitial when working with ditributed systems, such parallel processing requires objects to be duplicated in memory on each thread it is running on, which means memory overload and, mostly, much time spent on reconciliating all sub-results at the end of the line[28][29][30]. On non-ditributed systems, this can quickly become an overkill[31][32].

Only when *Rcpp* has been considered is it that a reasonnable processing time could be achieved for knn training. It litterally permitted to turn hours of training time into minutes. The Rcpp package provides an API on top of R, permitting direct interchange of R objects between R and C++[12]. That way, access is given to efficient memory pre-allocation to save from the burden of repetitive objects resizing as well as to the usage of pointers and references (addresses) to save from the hussle of unnecessarily overcrowding the memory. It's fairly easy to jumpstart Rcpp coding thanks to the excellent “**Rcpp for everyone**” by Masaki E. TSUDA[33]. These *documentation* pages put together by the initial and main author of the package, Mr. Dirk EDDELBUETTE, can also be of some assistance : [34].
For the Rcpp package to be usable, a working C++ compiler is also needed in order to build binary packages. On Windows, you can install “Rtools” ('https://cran.r-project.org/bin/windows/Rtools/'). To check wether or not you have a working version of rtools associated to your R session, you can utilize the below R line of code :

[9] All product and company names are trademarks™ or registered® trademarks of their respective holders.

[10] Gaming laptops are strong and make great Machine Learning platforms.

[11] packages used in that context were *foreach*, *parallel*, *doParallel* and *doSNOW*[27]

[12] to abide by the requirements of this report, the project had to be made up of only 2 R files : one for the report generation itself and one single file for the entire model. The *filecontents* package from *MiKTeX* has thus been used to embed the bibliography at the end of this report. Comparatively, the *inline* R package could be used to embed Rcpp source code inside the model R source file itself. Here, the *Rcpp::sourceCpp* function sufficed.

```
devtools::find_rtools( debug = TRUE )
#Either a visible TRUE if rtools is found, or an invisible FALSE with a diagnostic message.
```

In addition to the above mentionned arrangments, many tricks in the books had to be used to manipulate/operate on large objects efficiently (low memory consumption, fast execution times). For that reason, the entire algorithm has been developped from scratch with two exceptions : where available librairies already provided hard-to-beat performances, namely the *coop* package for *cosine similarity matrix computation* and the *stat* package for *pca primary components decomposition* :

- the coop package[13] is the only one that could be identified as being able to provide a stable cosine similarity matrix computation for cases with *missing values*, i.e. cases like the one we have with the movielens dataset where not all user/movie pairs is assigned a rating. As a side advantage, it is really fast. For the readers eager to learn about any such algorithm inner workings, refer to the excellent post by Rebecca BARTER where we can find a native R version of a "pairwise.complete" matrix computation : [35][14].
- the *stat* package performs really great in conjunction with *Microsoft R Open* and *intel MKL*. if you're nevertheless interested in investigating the possibility of developping your own implementation, as a starting point, you can refer to Day 92 of Tomáš Bouda's 100 days of algorithm and his pca algorithm in R : [36]

Of course, easilly available solutions such as the ones offered by the *recommenderlab* package already exist. They are however very much craving in terms of memory and, very demanding in terms of processing power (which often translates in unbearably long training times).

Further enhancements can of course still be brought to the algorithm as delivered. It would for instance be an easy win to further extend the resort to Rcpp, or even relate to the *RcppParallel* package. An other area that has not been explored but which could present advantages relies with *Microsoft MPI*[15] in conjunction with the *Rmpi* package[38].
An even much larger gain could indeed be achieved if all these solutions were used in conjunction with the usage of GPU.

A detailed trace generated during the training of a fully cross-validated final model instance can be found Appendix C. It shows progress information together with timing measures.

[13] by order of increasing performance, also considered packages for that job have been *lsa* and *proxy*.

[14] Rebecca BARTER introduces this similarity measure in the context of *Natural Language Processing (NLP)* as *documents classification* is an other area in which reliance over cosine similarity is widespread.

[15] a Microsoft implementation of the Message Passing interface standard[37] for developing and running parallel applications on a Windows platform.

4 Analysis

In the following section we're gonna explain the process and techniques used, such as data exploration and visualization, insights gained, and our modeling approach.

So let's start building this model, shall we ?

4.1 K-fold cross validation

We can not go any further in this report without dealing first with the concept of **cross validation**.

imagine that a user is in a particulary bad mood. He/she's had a very bad day. Then he/she rates a movie. The rating provided is well below the one that would have been given in any other circumstance.
It's for the trained model not to be too influenced by such individual anomalies (outliers) that modern machine learning systematically involves cross-validation via a resampling method such as k-fold[39].

Lets picture our dataset for a moment as a french baguette[40]. We start by putting aside a slice representing 10% of that baguette for latter validation of our final model instance (see Results section). With the remaining 90%, lets cut k equal size slices. in our case, as schematized figure 1 below, we'll make k = 10 ; employ 10-folds in our model cross-validation.
We'll train a model instance on 9 slices of *training data*, optimize its hyperparameters against the 10th slice which is used as our *test data*. We do this 10 times, using each time a different slice as our test dataset.

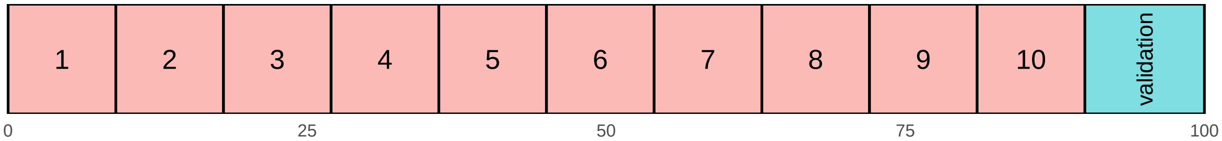

Figure 1: Dataset splitting

> Please do kindly bear with me. *Hyperparameters optimization* simply consists in measuring an optimization metric, in our case the RMSE, for different possible values of a parameter to be tuned in value. Each such value gives a different model RMSE value. The **optimum** value for that parameter is the one which gives the lowest RMSE for our model. We'll come to that in details in the following sections of the report.

4.2 Regularization

4.2.1 user effect and movie effect

Figure 2 below shows that (a) not all users rate on average the same and that (b) not all movies get rated on average the same.

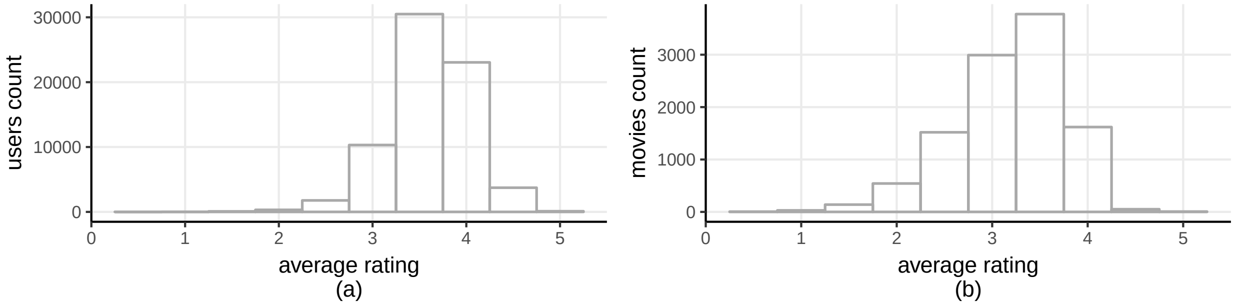

Figure 2: User & Movie effects

Our linear model of the users/movies ratings allocation can be written down as follows :

$$y_{\widehat{i,u}} = \hat{\mu} + \hat{b_i} + \hat{b_u}$$

where μ is the average all all ratings, b_i stands for the movie effect for movie "i" (e.g. the *"bias"* for movie "i") and b_u stands for the user effect for user "u" (e.g. the *"bias"* for user "u"). With such a model, all in all, estimates for the terms $\mu + b_i + b_u$ equates to our prediction $y_{\widehat{u,i}}$.

And here's how the RMSE equation translates :

$$\begin{aligned} RMSE &= \sqrt{\frac{1}{N}\sum_{u,i}(y_{u,i} - (\hat{\mu} + \hat{b_i} + \hat{b_u}))^2} \\ &= \sqrt{\frac{1}{N}\sum_{u,i}\epsilon_{u,i}^2} \end{aligned}$$

Where $\epsilon_{u,i}$ are the *residuals* to our linear model, i.e. the respective differences between each *true rating* and our corresponding *prediction*.

4.2.2 regularization

As can be seen figure 3(a) and (b) below, on the left part of the abscissa axises (x), many users provided little ratings and many movies have few ratings.

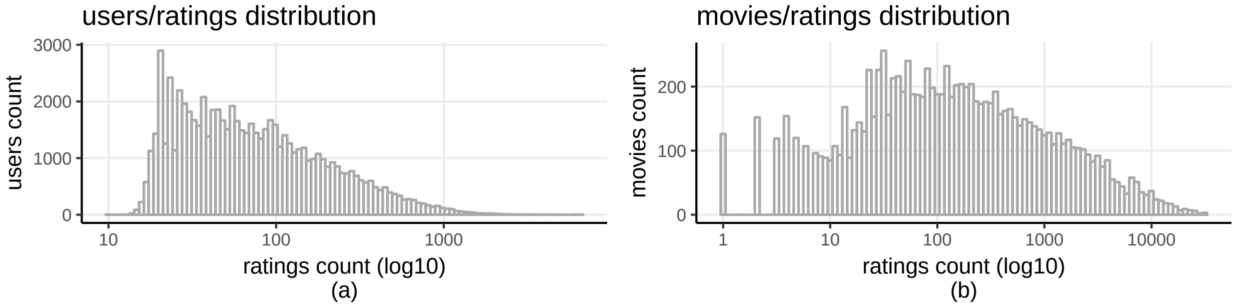

Figure 3: Distribution of ratings

From the two above mentioned observations, one can conclude that some ratings are to be considered as more significant/representative than others. To the RMSE equation, we can thus add a term as a penalty that gets larger whit large b_i and/or b_u. By doing so, we shrink the coefficient estimates towards zero (no user bias, no movie bias, for non statistically significant records).

$$RMSE_{penalized} = \sqrt{\frac{1}{N}\sum_{u,i}(y_{u,i} - (\hat{\mu} + b_i + b_u))^2 \quad + \quad \lambda(\sum_i b_i^2 + \sum_u b_u^2)}$$

Using calculus it can be shown that the values of b_i (and b_u) that minimize this equation are for cases with the number of movies "i" per user is larger than λ (and the number of users "u" per movie is larger than λ), where λ is the *penalization term* ; the hyperparameter to be optimized during model training.
For each of the model instances trained during k-fold cross validation, we'll establish the optimum value of λ ; the one minimizing $RMSE_{(\lambda)}$. We'll finally apply the average over these "k" different values for λ to our final model. Stay tuned !

4.3 K Nearest Neighbors

The final result doesn't look anything like it but this August 2015 **datascience.stackexchange** post by Bartłomiej TWARDOWSKi served as the initial inspiration for the *knn training* part of the algorithm developped in the frame of the herein report :[41].

There are several popular means of comparing items via measuring distances/similarities[42]. The Netflix^{TM} challenge winning team went for the **cosine similarity**. The Cosine Similarity between two movies is the computation of the cosine of the angle between two "movie" vectors in the "movies" space :

$$\begin{aligned} similarity(\overrightarrow{A}, \overrightarrow{B}) &= cos(\Theta) \\ &= \frac{\overrightarrow{A} \cdot \overrightarrow{B}}{||\overrightarrow{A}|| \cdot ||\overrightarrow{B}||} \\ &= \frac{\sum_{i=1}^{N} A_i B_i}{\sqrt{\left(\sum_{i=1}^{N} A_i^2\right)\left(\sum_{i=1}^{N} B_i^2\right)}} \end{aligned}$$

In a 3D environment (N = 3), it is fairly easy to compute. Both the **A** and **B** vectors have three coordinates. In our case of computing rating similarities among movies, we however are in an environment of **10,677** dimensions (N = 10,677 different movies)[16].
This is for instance why we choose to go with "movies" knn here and not "users" knn : our dataset englobes **69,878** different users. This would have made computation of the similarities matrix way too long[17].

Similarity ranges from -1 (exactly opposite), to 1 (exactly the same), with 0 indicating orthogonality or decorrelation, while in-between values indicate intermediate similarity or dissimilarity.

For instance, for the movie **"Scout, The (1994)"**, figure **4** shows the distribution of similarities with each of the 10,677 movies of our dataset[18]. Be reminded that we consider residuals from regularized linear modelling of ratings here.

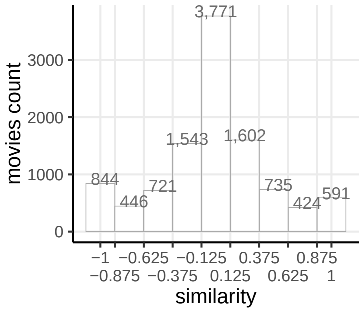

Figure 4: Cosine similarity of a movie against all the others

[16] this well known fact is called **curse of dimensionnality**. A very educational explaination of that phenomenon can be found there : [43].

[17] it is interesting to note however that, the higher the number of dimensions, the better the accuracy of the knn predictions, since discriminating between user rating profiles is more precise the more different users there are (e.g. the more users there are, the more similar to a particular user are its k closest neighbors). Going with "users" knn modeling would have brought to an increased accuracy.

[18] we consider pairwise similarities (see Little geek section).

4.4 Primary Components Analysis

Matrix factorization is a mathematical method that makes it possible to decompose a matrix into two matrices of lower dimensions[19]. The variability of the ratings (with users and movies) can thus be decomposed in two matrices : the **weights** matrix and the **pattern** matrix.
Included figure 6, we can see a graphical representation of a matrix factorization of a sample[20] from our dataset.

$$residuals\_matrix = weights\_matrix * pattern\_matrix^T$$

The principle consists in considering the right amount of variability in the training dataset, meaning isolating and ignoring the part in the variability in the training data that relates purely to randomness so as to ignore it in future predictions. Taking into account the right amount of primary components brings the best ability to predict future observation. This amount also is an hyperparameter to be optimized during model training.

4.4.1 The "weights" matrix

The **weights** matrix has *"users count"* rows and is made up of uncorrelated **primary component** columns, each of which have a decreasing standard deviation (e.g. a decreasing variability). To put it simply, **PC1**, the first primary component, the one with the highest standard deviation, translates the movie characteristic that divides the most among our set of users. Table 1 below shows the respective standard deviation of the 15 first primary components of the entire dataset.

Table 1: Primary compnents standard deviation

rank	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sdev.percent	1.11	0.88	0.82	0.72	0.70	0.65	0.64	0.60	0.60	0.58	0.56	0.54	0.54	0.54	0.53

4.4.2 The "pattern" matrix

The **pattern** matrix has *"movies count"* rows and encompasses the relation between each movie and each primary component. It's in the *pattern* matrix that we can identify clusters of movies. For instance, looking at *PC1*, the first primary component, the one explaining for most of the user ratings variability, we can guesstimate what the most important characteristic within our set of movies is.

Table 2 lists the 10 movies that are on one extreme end of PC1 and table 3 the 10 movies that are on the other. From there, we can interject that it separates thrillers from disaster films[21]

Table 2: Top 10 PC1 movies

title	PC1
Independence Day (a.k.a. ID...	0.2063
Twister (1996)	0.1188
Armageddon (1998)	0.1182
Jurassic Park (1993)	0.1169
Forrest Gump (1994)	0.1131
Pretty Woman (1990)	0.1093
Titanic (1997)	0.1080
Speed (1994)	0.1068
Star Wars: Episode I - The ...	0.1061
Ghost (1990)	0.1028

Table 3: Reverse top 10 PC1 movies

title	PC1
Pulp Fiction (1994)	-0.2395
Fargo (1996)	-0.1360
Clockwork Orange, A (1971)	-0.1107
2001: A Space Odyssey (1968)	-0.1013
12 Monkeys (Twelve Monkeys)...	-0.1002
Natural Born Killers (1994)	-0.0964
American Beauty (1999)	-0.0949
Being John Malkovich (1999)	-0.0898
Godfather, The (1972)	-0.0837
Taxi Driver (1976)	-0.0836

Comparatively, with table 4 and 5, we can do the same for **PC2**, the second primary component, which then seems to translate the dichotomy in the tastes of our set of users between 'science fiction / superheros' movies and drama.

[19] an other interesting approach than the one adopted in this report to introduce matrix factorization can be found there :[44], with further insights there :[45].

[20] a random sample of ratings for 80 users and 40 movies, where only the first 20 primary components are shown.

[21] subject to interpretation, primary components are obviously varying with different datasets.

Table 4: Top 10 PC2 movies

title	PC2
Congo (1995)	0.0551
Johnny Mnemonic (1995)	0.0476
Judge Dredd (1995)	0.0476
Net, The (1995)	0.0469
Natural Born Killers (1994)	0.0449
Beverly Hills Cop III (1994)	0.0411
Wild Wild West (1999)	0.0388
Robin Hood: Men in Tights (...	0.0353
Specialist, The (1994)	0.0352
Tank Girl (1995)	0.0347

Table 5: Reverse top 10 PC2 movies

title	PC2
Star Wars: Episode IV - A N...	-0.2446
Star Wars: Episode V - The ...	-0.2059
Jurassic Park (1993)	-0.2035
Forrest Gump (1994)	-0.1869
Star Wars: Episode VI - Ret...	-0.1794
Braveheart (1995)	-0.1702
Raiders of the Lost Ark (In...	-0.1624
Terminator 2: Judgment Day ...	-0.1612
Toy Story (1995)	-0.1594
Silence of the Lambs, The (...	-0.1416

4.4.3 Variance explained

The *variance explained* is the cummulative sum of the standard deviation of all the primary components. For instance, we can see figure 5(a) that it takes the first 1,397 primary components to account for 80% of the variability in our dataset.
This figure shows **variance explained** over **rank** for each of the pca model instances of the whole k-fold training process. We can notice there that all 12 curves overlay at this magnification. The differences between each of them is not enormous.

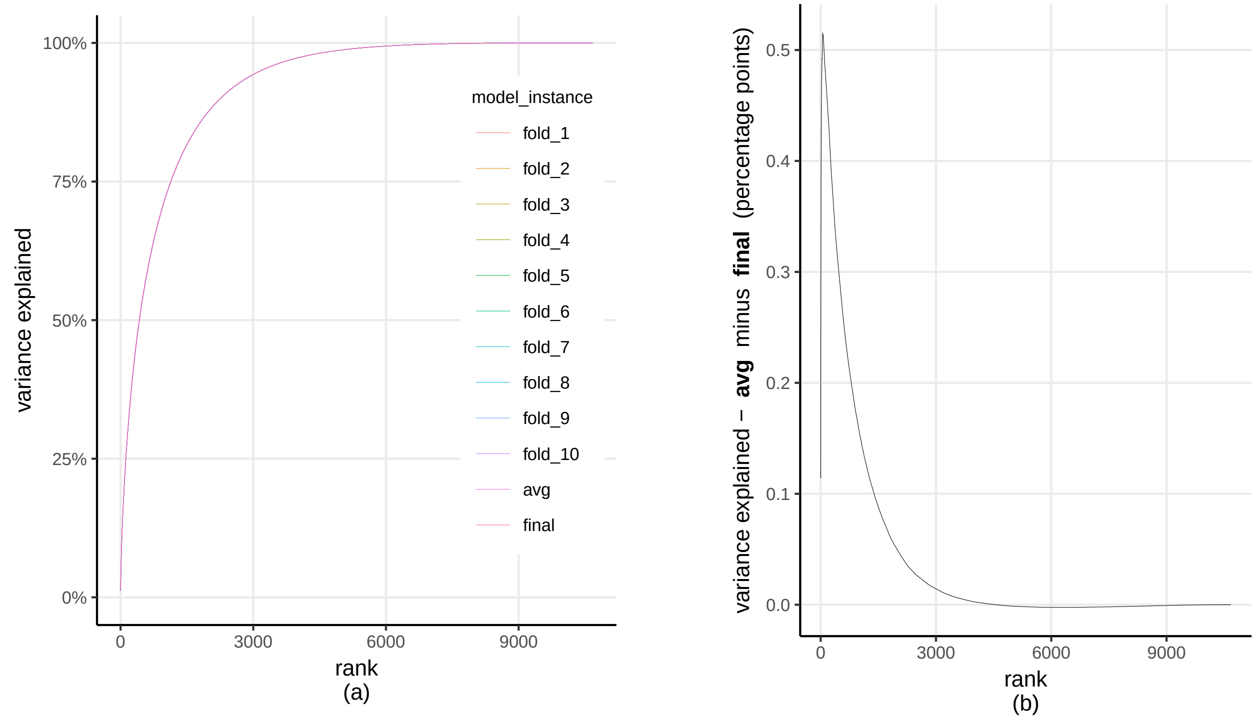

Figure 5: Variance explained

Figure 5(b) shows the spread of *variance explained* between the *average of all cross-validation folds* (in our case here, 10 folds) and our *final trained pca model instance*. However not huge, this spread is not negligible. This is to illustrate a mistake commonly made when using cross-validation. Averaging *results* from cross-validation model instances does not equate to actually training a model on the entire training dataset using the average of k-fold *optimal hyperparameters values*.

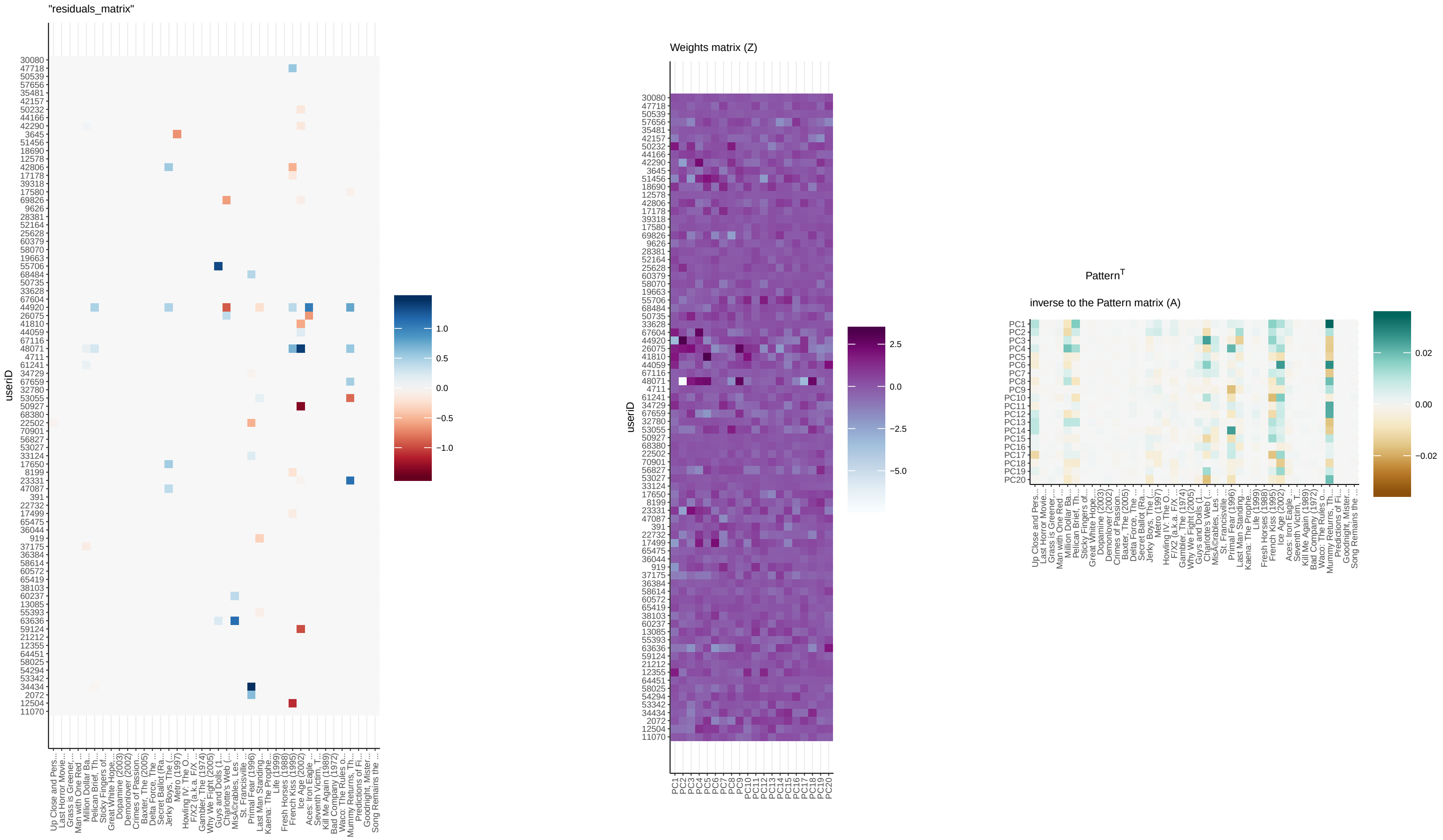

Figure 6: Matrix factorization

5 Results

In this section, we'll go over the respective results of each of the pieces constitutive to our final fully trained model instance.

5.1 Regularization

Figure 7 shows the different values of the RMSE performance metric for each value of the **lambda** hyperparameter, the penalty term from the regularization algorithm, against which it has been measured at training time.

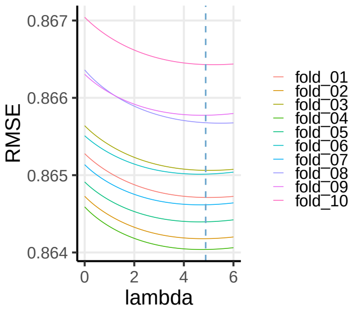

Figure 7: Regularization optimization

The "non-regularized" linear model corresponds to RMSE for *lambda = 0*. For each of the 10 cross-validation folds, we can observe U curves bottoming out at the optimum lambda value (e.g. the value corresponding to the lowest RMSE).

Table 6 shows these values of the lambda hyperparameter. It also shows the RMSE measures for linear models and regularized linear models, showing how much is gained with optimum regularization.

Table 6: Cross-validation folds - regularized RMSE

	fold_01	fold_02	fold_03	fold_04	fold_05	fold_06	fold_07	fold_08	fold_09	fold_10
non-regularized	0.8653	0.8647	0.8656	0.8646	0.8649	0.8655	0.8651	0.8664	0.8663	0.867
regularized	0.8647	0.8642	0.8651	0.864	0.8644	0.865	0.8646	0.8657	0.8658	0.8664
lambda	*5*	*4.8*	*5*	*4.7*	*4.6*	*4.6*	*4.7*	*5.5*	*4.7*	*5.2*

From there, we can draw that the optimum **lambda** value to be used for our final trained model is the average value of **4.88** !

5.2 K Nearest Neighbors

Comparatively to regularization optimization, with knn optimization, we end up getting the results reflected figure 8 and table 7.
We can then observe the gains from stacking knn modelling over reularized linear modelling with the corresponding enhancement as observable on the RMSE performance metric.
For our final trained model, **k** (the amount of nearest neighbors to be considered for each movie) takes the optimum value averaged from cross validation of **2,158**.

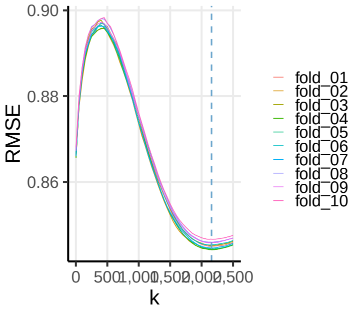

Figure 8: KNN optimization

Table 7: Cross-validation folds - knn RMSE

	fold_01	fold_02	fold_03	fold_04	fold_05	fold_06	fold_07	fold_08	fold_09	fold_10
regularized	0.8647	0.8642	0.8651	0.864	0.8644	0.865	0.8646	0.8657	0.8658	0.8664
regularized + knn	0.845	0.8443	0.8452	0.8442	0.8444	0.8453	0.8446	0.8459	0.8458	0.8466
k	*2,120*	*2,160*	*2,150*	*2,170*	*2,200*	*2,130*	*2,170*	*2,160*	*2,140*	*2,180*

5.3 Primary Components Analysis

The optimum value for the *primary component rank* (the hyperparameter **prcomp_rank**) for our final model is **53**. Interestingly, looking back at figure 5, we see that it accounts for only 16% of the variability in the residuals to our regularized linear model[22].

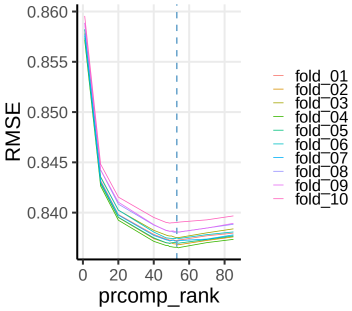

Figure 9: PCA optimization

Table 8: Cross-validation folds - pca RMSE

	fold_01	fold_02	fold_03	fold_04	fold_05	fold_06	fold_07	fold_08	fold_09	fold_10
regularized	0.8647	0.8642	0.8651	0.864	0.8644	0.865	0.8646	0.8657	0.8658	0.8664
regularized + pca	0.8372	0.8368	0.8375	0.8365	0.8369	0.8374	0.8372	0.838	0.8381	0.839
prcomp_rank	*53*	*54*	*53*	*54*	*49*	*52*	*52*	*52*	*54*	*49*

5.4 Bagging

As can be observed on below table 9, combining *regularization + knn* on one hand and *regularization + pca* on the other always brings to enhanced results compared to the ones obtained with either of both alone.
Optimized during cross validation, the relative weights each model instance takes in the mix is characterized by the **rel_prop** hyperparameter.
In our final model, when the result of the *regularization + knn* model is assigned a weight of **1**, the result of the *regularization + pca* model is assigned a weight of **rel_prop = 2.9**.

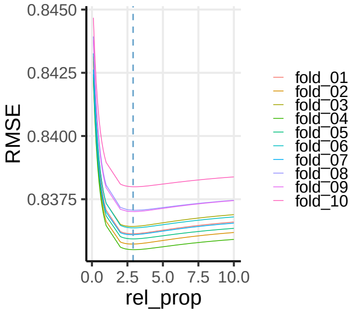

Figure 10: Bagging optimization

Table 9: Cross-validation folds - bagging RMSE

	fold_01	fold_02	fold_03	fold_04	fold_05	fold_06	fold_07	fold_08	fold_09	fold_10
regularized + knn	0.845	0.8443	0.8452	0.8442	0.8444	0.8453	0.8446	0.8459	0.8458	0.8466
regularized + pca	0.8372	0.8368	0.8375	0.8365	0.8369	0.8374	0.8372	0.838	0.8381	0.839
bagging	0.8361	0.8357	0.8364	0.8355	0.8359	0.8364	0.8361	0.8371	0.837	0.838
rel_prop	*2.8*	*2.8*	*2.8*	*3*	*2.9*	*2.9*	*2.8*	*3*	*3*	*3*

[22] such a low coverage for the variability of the data could be attributed to the fact that, contrarily to what we've done with the knn modelling, with pca modelling, to missing values (user/movie pairs with no rating) we've attributed a **0 (zero)** residual value (e.g. an average between all ratings from that user and all ratings for that movie), which is a stance / a parti pris.

5.5 Validation dataset

The overall objective to training a machine learning algorithm of course resides in being able to generate predictions.

On the validation dataset[23], we managed to reach an RMSE of 0.833018.

Table 10: final model instance - RMSE on validation dataset

regularized + knn	regularized + pca	weighted
0.8438	0.8336	0.833

We did cover many aspects of the algorithm that won the Grand Prize from the Netflix^{TM} challenge. For simplicity, many other (secondary) characteristics have been left aside, for instance no consideration has been given to time variations such as the fact that users become harsher critics over time.

> DISCLAIMER: In the context of the Netflix^{TM} challenge, the final RMSE was higher than the one we obtain here despite the fact that that other dataset had 10 times more records than us but, no conclusion can be drawn as long as we don't have accesss to the original dataset.

[23] we saved 10% of the source dataset for validation purposes. The Model has been developed (trained/tested) on the remaining 90% (see the cross-validation section).

6 Conclusion / Discussion

From its open Grand Contest, Netflix^{TM} did adopt one solution which brought a 8.43% improvement in RMSE, but not the Grand Prize winner, which delivered over 10% improvement over the existing in-house solution as required.
Simply put, the additional 1.57% difference was not deemed meaningful enough, given the additional engineering work it required to put it in production. This brought some quite intense badmouthing on crowded R&D in the media[46], but there's good to take in everything and Netflix gained much from this experiment of theirs : the collectively spent hours amounted to much more than the awarded monetary prize AND, they learnt a lot[47].

Netflix^{TM} had launched the challenge in 2006 and the prize was awarded when a team finally reached the challenged critera of 10% improvement in RMSE three years later in 2009. By then, Netflix^{TM} had moved away from DVD renting to enter the movie streaming buisiness.
This move changed the way recommendations should be delivered and how user preferences were estimated. With DVD, they could primarily rely on explicit ratings users gave. With streaming, they could use click/stream data[48] and other online behavior to figure out what users actually liked, and what they didn't. This change of business model also implied that they needed a system more responsive to user real time interactions. To abide by these constraints, several hybrid systems emerged, notably LCARS (or Location-Content-Aware Recommender System)[49] and other context-aware systems.

That being said, collaborative filtering recommender engines like the one covered in the herein report are still a very relevant and predominant part of the mix. Personnalizations based on preferences is a modern golden mine. They are the reason why data is so valuable in the modern age. KYC (Know Your Customer) is any startup's obsession nowadays and is at the center of most companies' digital transformation.
In an era where everything that is Machine Learning related changes so quickly and where new technological trends emerge every so often, the challenges recommendation systems have to face are however numerous :

- Trust – A recommender system is of little value for a user if the user does not trust the system. Trust can be built by a recommender system by explaining how it generates recommendations, and why it recommends an item.
  At the moment, **models explainability** is of the essence. Machine Learning algorithms currently suffer from an "acceptability" problem. They are often perceived as black boxes, which is one of the main reason why they don't really take off in companies with an already established business. Decision makers require/need transparency in order to rely on a solution. That's why some serious educational wisdom needs to be dropped into the communication of machine learning practioners who need not be put on a pedestal.

- Cold Start - a typical recommendation system dilema is associated to the early situation when the system cannot draw any inference for a user or item about which it has not yet gathered sufficient information. For a user that has never before rated any movie, user demographics is for instance often used as a fallback to identify cluster of users to which he/she belongs[50].

- Scalability - bringing an algorithm into production conditions, with dataset made up of billions of records sometimes increasing at millions-a-day pace, is a reccurent real-world constraint on adoption. Cloud computing service providers dependency and the effect of related invoices on a business bottom line are both increasingly scrutinized.

- Privacy - The topic of users/consumers data privacy did not await the 21st century to emerge, with for example the creation by France in January 1978 of a dedicated surveillance commission : the "*Commission nationale de l'informatique et des libertés*" (CNIL)[51][52][53]. The issue of course slipped rapidly into recommendation systems when they emerged[54], [55]. Europe has been pioneering on the legislative front with major concerns such as **data ownership** and **data protection**, with the adoption in April 2016 of the "*EU General Data Protection Regulation*" (GDPR)[56]. Countries outside the E.U. show growing awareness and sensitivity towards these matters, like for instance the U.S. which are yet to pass such a law but the state of California went ahead in June 2018 by passing into law the so called "*California Consumer Privacy Act*"[57][58].

- Diversity - Last but not least, the ethical question of **exposure diversity** commands itself. Should recommendation systems only care about people's past behavior and popular demand ? Aren't content quality and diversity also essential ? These issues shall always remain central to the debate on building a better tomorrow[59][60].

As already contemplated earlier in this report, the strength of recommendation are numerous ; it builds up user retention, allows room for cross sales with paired recommendations, etc[61]. There have also been many studies on the economical effect of recommender engines, such as the one reflected in this engaging research paper by the Department of Economics from YALE University on buyers segmentation based on willingness to pay less for uncertainty in trying a new product: [62].

A/B testing is super trendy as a seamless method used to introduce new functionalities and improve the user experience. It consists in steering different variations of a feature towards subsets of users and then measuring which of these variations brings better performance[24]. At the age of the Internet, companies continuously more try to experiment with flexibility, adopt a "trial and error" attitude, fail faster and rebound at lower cost but staying safe from the fear to innovate.

Recommender engines are not immune from that agile philosophy. We can only welcome such a development.

[24] which performance has often has to do with user conversion rate (from "browser" to "buyer").

Appendices

Appendix A

Provided with the algorithm put together in order to generate this report is a set of fully working examples. Placed within ***## Not run:*** / ***## End(Not run)*** tags, they allow for fully running code lines using the functions described Appendix B.

One can for instance get a single instance from one cross-validation fold, one can call the **get_model_instance** function

```
a_model <- get_model_instance(
  dataset = edx
  , dataset_test_idx = folds[[ f ]]
  , lambdas = seq( 0, 6, .1 )
  , ks = c( 2, seq( 50, 2000, by = 50 )
            , seq( 2000, 2500, by = 10 ) )
  , prcomp_ranks = c( 1, 10, 20, 40
                      , seq( 47, 55, 1 ), 70, 85 )
  , rel_prop = c( seq( .1, 1, by = .1 )
                  , seq( 1.5, 6, by = .5 )
                  , seq( 7, 10 ) )
  , print_comments = TRUE )
```

To get a fully cross-validated final model instance, one can make a call to the **get_cv_model_instance** function :

```
trained_model_instance <- get_cv_model_instance(
  dataset = edx
  , cv_folds_count = 10
  , lambdas = seq( 0, 6, .1 )
  , ks = c( 2, seq( 50, 2000, by = 50 )
            , seq( 2000, 2500, by = 10 ) )
  , prcomp_ranks = c( 1, 10, 20, 40
                      , seq( 47, 55, 1 ), 70, 85 )
  , rel_props = c( seq( .1, 1, by = .1 )
                   , seq( 2, 6, by = .5 ), seq( 7, 10 ) )
  , print_comments = TRUE )
```

A detailed trace as generated by such a call is provided Appendix C. On a computer comparable to the one described in the Little geek section, it take a little under 40 hours to train.

To generate predictions, a user can then make a call to the **model_predict** function :

```
predictions <-
  model_predict( trained_model_instance =
                    trained_model_instance
                 , validation_set =
                    validation[ , c( "userId", "movieId" ) ]
                 , true_ratings =
                    validation[ , c( "rating" ) ]
                 , print_comments = TRUE )
```

Alternatively, the same can also be done on a small subset of the data, called **dev_subset** (and its associated **dev_validation**). This subset, also provided, corresponds to a sample piece of the entire dataset, made up of *"only"* ratings from a select 500 users.
On a computer comparable to the one described in the Little geek section, it take a little over 1 hour to train a fully cross-validated model using the "lambdas" "ks", "prcomp_ranks" and "rel_props" ranges provided in the corresponding example sections.

Appendix B

List of functions developped to constitute the algorithm that comes along with the herein report.

amc_pdf_print

```
## ###############################################
##         'amc_pdf_print' function             ##
##################################################
# convenience method to print the pdf report     #
# with (optional) input parameters.              #
# In addition, allows to access                  #
# 'global environment' variables from within     #
# the 'report printing' session (thus avoiding   #
# to train a model each time).                   #
##################################################
```

RMSE

```
## ########################################
## 'RMSE' function                       ##
###########################################
# a function that computes the "RMSE"     #
# for vectors of "ratings"                #
# and their corresponding "predictions" : #
# REMINDER: we're optimizing our model    #
#           against the RMSE metric.      #
###########################################
```

to_rating

```
## ##################################################
## 'to_rating' function                            ##
#####################################################
# converts continuous values to 'stars rating' unit #
# movilens ratings ranging from "0.5" to "5" stars  #
#####################################################
```

duration_string

```
## #######################################################
## 'duration_string' function                           ##
##########################################################
# convenience method to custom-format duration strings   #
##########################################################
```

get_regularization_optimization

```
## #######################################################
## 'get_regularization_optimization' function           ##
##########################################################
# optimization procedure for the hyperparameter "lambda" #
# (regularization penalty term)                          #
##########################################################
# inputs :                                               #
#    - "train_set" - the source data :                   #
#      user/movie ratings in tidy format                 #
#      with the following column names :                 #
#          o "userId", "movieId", "title", "rating"      #
#    - "test_set" data points used                       #
#      to generate predictions (to be compared           #
#      against 'true ratings'                            #
#    - "lambdas" list of different values                #
#      to be considered for "lambda"                     #
#      (primary components count)                        #
#    - "print_comments" do (or not) show                 #
#      progress info on the console                      #
##########################################################
# resultset (list of objects) :                          #
#    - "mu_hat", "movie_avgs" and "user_avgs"            #
#      regularization parameters                         #
#    - "lambda" - the optimum parameter value            #
#    - "lambda_optimization"                             #
#      the "RMSE" versus "k" dataset                     #
##########################################################
```

get_regularization_residuals_matrix

```
## #################################################
## 'get_regularization_residuals_matrix' function ##
####################################################
# inputs :                                         #
#    - "train_set" - the source data :             #
#      user/movie ratings in tidy format           #
#      with the following column names :           #
#          o "userId", "movieId", "rating"         #
#    - "mu_hat", "movie_avgs" and "user_avgs" ;    #
#      the Regularization parameters               #
#    - "print_comments" do (or not) show progress  #
#      info on the console                         #
####################################################
# resultset : a matrix made up of the user/movie   #
# rating residuals (e.g. the error/loss from       #
#  the Regularization)                             #
####################################################
```

get_knn_predictions

```
## ####################################################
## 'get_knn_predictions' function                    ##
#######################################################
#                 Rcpp implementation                 #
# for each "userId"/movieId" pair of the test domain, #
# returns one rating prediction                       #
# per value of the "ks" vector                        #
#       (Rcpp function called inside                  #
#        the R 'get_knn_optimization' function)       #
#######################################################
# inputs :                                            #
#    - "ks" list of different values                  #
#      to be considered for "k" (neighbors count)     #
#    - "test_sim_matrix" similarity matrix            #
#      for the "test" movies (one per column)         #
#    - "test_ratings_matrix" rating matrix            #
#      for the "test" users, all movies included      #
#      (from which "neighbor" ratings are picked)     #
#    - "test_domain" list of "userId"/movieId" pairs  #
#      (for which a prediction is expected)           #
#    - "print_comments" do (or not) show              #
#      progress info on the console                   #
#######################################################
# resultset (tidy format ; 4 columns) :               #
#    - "userId"                                       #
#    - "movieId"                                      #
#    - "k"                                            #
#    - "prediction"                                   #
#######################################################
```

get_knn_optimization

```
## #######################################################
## 'get_knn_optimization' function                      ##
##########################################################
#    optimization procedure for the hyperparameter "k"   #
##########################################################
# inputs :                                               #
#    - "train_rating_residuals_matrix" similarity matrix #
#      for the "residual ratings"                        #
#      (remainder after "Regularization")                #
#      on the "training" dataset                         #
#    - "ks" list of different values                     #
#      to be considered for "k" (neighbors count)        #
#    - "test_set" data points used                       #
#      to compare predictions against 'true ratings'     #
#    - "mu_hat", "movie_avgs" and "user_avgs"            #
#      regularization parameters                         #
#    - "print_comments" do (or not) show                 #
#      progress info on the console                      #
##########################################################
# resultset (list of objects) :                          #
#    - "similarity_matrix"                             #
#     the movies cosine similarity matrix                #
#    - "k" - the optimum parameter value                 #
#    - "k_optimization" - the "RMSE" versus "k" dataset  #
#    - "predicted_ratings"                               #
#      the optimized predictions in tidy format          #
#      with colnames "userId", "movieId" and "pred"      #
##########################################################
```

get_pca_optimization

```
## ############################################################
## 'get_pca_optimization' function                           ##
```

```
###############################################################
# optimization procedure for the hyperparameter "prcomp_rank" #
###############################################################
# inputs :                                                    #
#    - "train_rating_residuals_matrix" similarity matrix      #
#      for the "residual ratings"                             #
#      (remainder after "Regularization")                     #
#      on the "training" dataset                              #
#    - "prcomp_ranks" list of different values                #
#      to be considered for "prcomp_ranks"                    #
#      (primary components count)                             #
#    - "test_set" data points used                            #
#      to compare predictions against 'true ratings'          #
#    - "mu_hat", "movie_avgs" and "user_avgs"                 #
#      regularization parameters                              #
#    - "print_comments" do (or not) show                      #
#      progress info on the console                           #
###############################################################
# resultset (list of objects) :                               #
#    - "pca" - the primary components decomposition object    #
#      (of class "prcomp")                                    #
#    - "prcomp_rank" - the optimum parameter value            #
#    - "prcomp_rank_optimization"                             #
#      the "RMSE" versus "prcomp_rank" dataset                #
#    - "predicted_ratings"                                    #
#      the optimized predictions in tidy format               #
#      with colnames "userId", "movieId" and "pred"           #
###############################################################
```

get_model_instance

```
## ##############################################
## 'get_model_instance' function               ##
#################################################
# optimization procedure for the set of         #
# hyperparameters of our recommender system,    #
# which consists in Regularization, KNN and PCA #
#  => "lambda", "k" and "prcomp_rank"           #
#################################################
# inputs :                                      #
#    - "dataset" - the source data :            #
#      user/movie ratings                       #
#      to be provided in tidy format            #
#      with the following column names :        #
#          o "userId", "movieId", "rating"      #
#    - "dataset_test_idx"                       #
#      row numbers of 'dataset'                 #
#      to be considered for testing             #
#      (the remainder for training)             #
#    - "lambdas" list of different values       #
#      to be considered for "lambda"            #
#      (penalty term)                           #
#    - "ks" list of different values            #
#      to be considered for "k"                 #
#      (neighbors count)                        #
#    - "prcomp_ranks" list of different values  #
#      to be considered for "prcomp_rank"       #
#      (primary components count)               #
#    - "rel_props" list of different values     #
#      to be considered for "rel_prop"          #
#      (relative proportion on the final result #
#       of the weight PCA predictions over      #
#       those of the KNN prediction)            #
#    - "print_comments" do (or not) show        #
#      progress info on the console             #
#################################################
# resultset (list of objects) :                 #
#                                               #
#    - "lambda" - the optimum parameter value   #
#    - "mu_hat", "movie_avgs" and "user_avgs"   #
#      the regularization parameters            #
#    - "residuals_matrix" - "residual ratings"  #
#      (remainder after "Regularization")       #
#                                               #
#    - "similarity_matrix"                    #
#      the movies cosine similarity matrix      #
#    - "k" - the optimum parameter value        #
#    - "k_optimization"                         #
#      the "RMSE" versus "k" dataset            #
#                                               #
#    - "pca" - the primary components           #
#      decomposition object                     #
#      (of class "prcomp")                      #
#    - "prcomp_rank"                            #
#      the optimum parameter value              #
#    - "prcomp_rank_optimization"               #
#      the "RMSE" versus "prcomp_rank" dataset  #
#                                               #
#    - "rel_prop"                               #
#      the optimum parameter value              #
#    - "rel_prop_optimization"                  #
#      the "RMSE" versus "rel_prop" dataset     #
#                                               #
#    - "predicted_ratings"                      #
#      the optimized predictions                #
#      in tidy format  with colnames            #
#      "userId", "movieId" and "pred"           #
#                                               #
#################################################
```

get_cv_model_instance

```
## ##############################################
## 'get_cv_model_instance' function           ##
#################################################
#   a "cross-validated" mixed model instance    #
#  (k-fold validation on "regularized knn/pca   #
#               weighted" modelling)           #
#################################################
# inputs :                                      #
#    - "dataset" - the source data :            #
#      user/movie ratings                       #
#      to be provided in tidy format            #
#      with the following column names :        #
#         o "userId", "movieId", "rating"       #
#    - "cv_folds_count"                         #
#      how many folds are to be used            #
#      for hyperparameters optimization         #
#      (the remainder for training)             #
#    - "lambdas" list of different values       #
#      to be considered for "lambda"            #
#      (penalty term)                           #
#    - "ks" list of different values            #
#      to be considered for "k"                 #
#      (neighbors count)                        #
#    - "prcomp_ranks" list of different values  #
#      to be considered for "prcomp_rank"       #
#      (primary components count)               #
#    - "rel_props" list of different values     #
#      to be considered for "rel_prop"          #
#      (relative proportion on the final result #
#       of the weight PCA predictions over      #
#       those of the KNN prediction)            #
#    - "print_comments" do (or not) show        #
#      progress info on the console             #
#################################################
# resultset (list of objects) :                 #
#                                               #
#    - "lambda" - the optimum parameter value   #
#    - "mu_hat", "movie_avgs" and "user_avgs"   #
#      the regularization parameters            #
#    - "residuals_matrix" - "residual ratings"  #
#      (remainder after "Regularization")       #
#                                               #
#    - "k" - the optimum parameter value        #
#    - "similarities_matrix"                    #
#      the movies cosine similarity matrix      #
#                                               #
#    - "prcomp_rank"                            #
#      the optimum parameter value              #
#    - "pca" - the primary components           #
#      decomposition object                     #
#      (of class "prcomp")                      #
#                                               #
#    - "rel_prop"                               #
#      the optimum parameter value              #
#                                               #
#    - "cv_k_models"                            #
#      a data.frame of "cv_folds_count" rows,   #
#      each corresponding to a "fold" model      #
#      (e.g. having the structure of an object  #
#       returned by the "get_model_instance"    #
#       function)                               #
#################################################
```

model_predict

```
## ######################################################
## 'model_predict' function                            ##
#########################################################
# inputs :                                              #
#    - "trained_model_instance" - a 'cross-validated'   #
#      trained model instance as generated              #
#      by the 'get_cv_model_instance' function          #
#    - "validation_set" - the 'validation'              #
#      dataset                                          #
#      to be provided in tidy format                    #
#      with the following column names :                #
#         o "userId", "movieId"                         #
#    - "true_ratings" ordered vector of 'true ratings'  #
#      for the "validation_set" list of observations    #
#      (optional, if RMSE measures are to be provided). #
#    - "print_comments" do (or not) show                #
#      progress info on the console                     #
#########################################################
# resultset : the predicted ratings                     #
#########################################################
```

Appendix C

Trace from a model training on the entire dataset

```
#
# 69,878 users ; 10,677 movies.
# ///////////////////////////////////////////////////////////////
#   TRAINING FOLD #1/10.....
```

[truncated]

```
#
# fold #9/10 training duration : 3h 48m 45s.
#
# ///////////////////////////////////////////////////////////////
#   TRAINING FOLD #10/10.....
# training set rows count: 8,100,064
# test set rows count: 899,991 (10.00% of the dataset)
# Regularization - computing "movie" & "user" effects:
#   |==================================================|
#   |--------------------------------------------------| done (3m 35s)
# Regularization - optimized "penalty" term : 4.7 - RMSE = 0.8658
# KNN - Computing the movies cosine similarity matrix.. done (18m 59s)
# Generating the knn predictions :
#   |=================================|
#   |---------------------------------| done: 31m 2s
# Movies neighbors optimization total duration : 52m 44s
# KNN - optimized "k" = 2,140 - RMSE = 0.8458
# PCA - performing the primary components decomposition.. done (33m 15s).
# intermediary housekeeping - garbage collection
# PCA - optimizing against the amount of Primary Components:
#   |=========| measurement 1/15
#   |---------| done (7m 6s).
#   |=========| measurement 2/15
#   |---------| done (8m 49s).
#   |=========| measurement 3/15
#   |---------| done (8m 13s).
#   |=========| measurement 4/15
#   |---------| done (10m 18s).
#   |=========| measurement 5/15
#   |---------| done (9m 27s).
#   |=========| measurement 6/15
#   |---------| done (10m 25s).
#   |=========| measurement 7/15
#   |---------| done (9m 24s).
#   |=========| measurement 8/15
#   |---------| done (9m 22s).
#   |=========| measurement 9/15
#   |---------| done (8m 49s).
#   |=========| measurement 10/15
#   |---------| done (9m 57s).
#   |=========| measurement 11/15
#   |---------| done (9m 56s).
#   |=========| measurement 12/15
#   |---------| done (10m 1s).
#   |=========| measurement 13/15
#   |---------| done (9m 54s).
#   |=========| measurement 14/15
#   |---------| done (9m 48s).
#   |=========| measurement 15/15
#   |---------| done (9m 48s).
# Primary components optimization total duration : 2h 54m 54s
# PCA - optimized "prcomp_rank" = 54 - RMSE = 0.8381
# housekeeping - garbage collection done.
# BAGGiNG - Optimizing against respective model instance weight :
#   |========================|
#   |------------------------| done (9s)
# BAGGiNG - optimized "rel_prop" = 3.0 - RMSE = 0.837
# fold #10/10 training duration : 3h 56m 14s.
#
# //////////////////////////////////////////////////////////////
#   total k-fold model instances training duration : 36h 24m 19s
# //////////////////////////////////////////////////////////////
#
#   Regularization - computing "movie" & "user" effects:
#   |=|
#   |-| done (7s)
# KNN - Computing the movies cosine similarity matrix.. done (19m 23s)
# PCA - performing the primary components decomposition.. done (32m 55s).
# serializing result..
```

References

[1] Supervised Random Walks - Predicting and Recommending Links in Social Networks. https://www-cs.stanford.edu/~jure/pubs/linkpred-wsdm11.pdf, 2011.

[2] Using collaborative filtering to weave an information tapestry. https://dl.acm.org/citation.cfm?doid=138859.138867, 1992.

[3] Two Decades of Recommender Systems at Amazon.com. https://pdfs.semanticscholar.org/0f06/d328f6deb44e5e67408e0c16a8c7356330d1.pdf, 2017.

[4] Personalized Recommendation Systems - Five Hot Research Topics You Must Know. https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/personalized-recommendation-systems/, Novembre 2018.

[5] Economics of Recommender System in Online Marketplaces. http://libtreasures.utdallas.edu/xmlui/bitstream/handle/10735.1/5485/ETD-5608-7403.99.pdf?sequence=5, August 2017.

[6] Winning the Netflix Prize: A Summary. http://blog.echen.me/2011/10/24/winning-the-netflix-prize-a-summary/, September 2011.

[7] The BellKor Solution to the Netflix Grand Prize. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf, August 2009.

[8] Netflix Privacy Lawsuit. https://www.wired.com/2009/12/netflix-privacy-lawsuit/, 2009.

[9] Entire latest 'MovieLens' dataset ; Full 27,000,000 ratings as of End of 2018). https://grouplens.org/datasets/movielens/latest/, 2019.

[10] Dataset containing 10,000,054 ratings and 95,580 tags applied to 10,681 movies by 71,567 users of the online movie recommender service MovieLens.). https://grouplens.org/datasets/movielens/10m/, 2009.

[11] F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets - History and Context. ACM Transactions on interactive intelligent Systems (TiiS) 5, 4, Article 19, 19 pages. http://dx.doi.org/10.1145/2827872, December 2015.

[12] Boosting (machine learning). https://en.wikipedia.org/wiki/Boosting_(machine_learning).

[13] The BellKor Solution to the Netflix Grand Prize. http://blog.kaggle.com/2017/06/15/stacking-made-easy-an-introduction-to-stacknet-by-competitions-grandmaster-marios-michailidis-kazanova/, June 2017.

[14] Marios Michailidis: How to become a Kaggle nb. 1: An introduction to model stacking. https://youtu.be/9Vk1rXLhG48, May 2017.

[15] Luc BESSON - en.wikipedia). https://en.wikipedia.org/wiki/Luc_Besson.

[16] Jean DUJARDIN - en.wikipedia). https://en.wikipedia.org/wiki/Jean_Dujardin.

[17] Cross-Validation Explained). http://genome.tugraz.at/proclassify/help/pages/XV.html, 2006.

[18] Overfitting in Machine Learning). https://elitedatascience.com/overfitting-in-machine-learning, 2017.

[19] How to manage Windows 10 Virtual Memory. https://www.geeksinphoenix.com/blog/post/2016/05/10/how-to-manage-windows-10-virtual-memory.aspx, May 2016.

[20] How to manage Windows 10 Virtual Memory. https://www.easeus.com/partition-master/partition-windows-10-free.html, May 2016.

[21] Microsoft R Open. https://mran.microsoft.com/open.

[22] Multithreading on Windows. https://mran.microsoft.com/documents/rro/multithread#mt-windows.

[23] Why is R slow? https://www.r-bloggers.com/why-is-r-slow-some-explanations-and-mklopenblas-setup-to-try-to-fix-this/.

[24] Changing R versions for RStudio desktop. https://support.rstudio.com/hc/en-us/articles/200486138-Using-Different-Versions-of-R, March 2019.

[25] RStudio running on 32 bit on a 64 bit OS with 64 bit R installation. https://stackoverflow.com/questions/51704038/rstudio-running-on-32-bit-on-a-64-bit-os-with-64-bit-r-installation, August 2018.

[26] Day 92 of Tomáš Bouda's 100 days of algorithm - pca algorithm in R. http://clarkfitzg.github.io/2017/11/06/are-apply-functions-faster-than-for-loops/, June 2017.

[27] Progress bar when using dopar. https://stackoverflow.com/questions/10903787/how-can-i-print-when-using-dopar#15078540, February 2015.

[28] are apply functions faster than for loops? https://stackoverflow.com/questions/16963808/foreach-dopar-slower-than-for-loop#16964023, November 2017.

[29] R foreach with .combine=rbindlist. https://stackoverflow.com/questions/17411223/r-foreach-with-combine-rbindlist#17411848, July 2013.

[30] Why is rbindlist "better" than rbind? https://stackoverflow.com/questions/15673550/why-is-rbindlist-better-than-rbind#23983648, June 2014.

[31] doParallel error in R - error writing to connection. https://stackoverflow.com/questions/28503208/doparallel-error-in-r-error-in-serializedata-nodecon-error-writing-to-con, February 2015.

[32] doParallel error in R - error writing to connection 2. https://stackoverflow.com/questions/51054423/error-in-serializedata-nodecon-error-writing-to-connection, June 2018.

[33] Masaki E. TSUDA. Rcpp for everyone. https://teuder.github.io/rcpp4everyone_en/, February 2019.

[34] Dirk EDDELBUETTEL. Rcpp Version 1.0.1 Documentation. http://dirk.eddelbuettel.com/code/rcpp/html/dir_edae2c70406e9616e8689419d3d5106b.html.

[35] Rebecca BARTER. Visualising Word2Vec Word Vectors. https://rlbarter.github.io/superheat-examples/Word2Vec/#visualising-cosine-similarity-for-the-40-most-common-words, February 2019.

[36] Day 92 of Tomáš Bouda's 100 days of algorithm - pca algorithm in R. https://medium.com/100-days-of-algorithms/day-92-pca-bdb66840a8fb#dc37, June 2017.

[37] MPi Forum. https://www.mpi-forum.org/, May 2018.

[38] instructions to install and run Rmpi under Microsoft MPi. http://fisher.stats.uwo.ca/faculty/yu/Rmpi/, May 2018.

[39] Difference between bootstrapping and cross-validation. https://datascience.stackexchange.com/questions/32264/what-is-the-difference-between-bootstrapping-and-cross-validation#32276, May 2018.

[40] French Baguette. https://en.wikipedia.org/wiki/Baguette.

[41] User-based nearest neighbour implementation in R? https://datascience.stackexchange.com/questions/6660/user-based-nearest-neighbour-implementation-in-r#6690, August 2015.

[42] Distance and Similarity Measures. https://reference.wolfram.com/language/guide/DistanceAndSimilarityMeasures.html.

[43] Explain "Curse of dimensionality" to a child. https://stats.stackexchange.com/questions/169156/explain-curse-of-dimensionality-to-a-child#169170, August 2015.

[44] Practical Guide to Principal Component Analysis (PCA) in R and Python. https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis-python/, March 2016.

[45] Principal Component Analysis in R. https://www.r-bloggers.com/principal-component-analysis-in-r/, January 2017.

[46] What the Failed Netflix Prize Says About Business Advice. https://www.forbes.com/sites/ryanholiday/2012/04/16/what-the-failed-1m-netflix-prize-tells-us-about-business-advice/, April 2012.

[47] Matlab - The Netflix Prize and Production Machine Learning Systems - An Insider Look. https://blogs.mathworks.com/loren/2015/04/22/the-netflix-prize-and-production-machine-learning-systems-an-insider-look/#390faa49-a84f-4399-bb8e-ce3402b96044, April 2015.

[48] Building a Collaborative Filtering Recommender System with ClickStream Data. https://towardsdatascience.com/building-a-collaborative-filtering-recommender-system-with-clickstream-data-dffc86c8c65, April 2019.

[49] LCARS - A Location-Content-Aware Recommender System. https://www.cs.cmu.edu/~zhitingh/data/kdd13_lcars.pdf, 2013.

[50] Exploiting User Demographic Attributes for Solving Cold-Start Problem in Recommender System. http://www.lnse.org/papers/66-CA2016.pdf, August 2013.

[51] Commission nationale de l'informatique et des libertés - en.wikipedia. https://en.wikipedia.org/wiki/Commission_nationale_de_l%27informatique_et_des_libert%C3%A9s.

[52] CNIL - To protect personal data, support innovation, preserve individual liberties. https://www.cnil.fr/en/home.

[53] CNIL - English French glossary on data protection. https://www.cnil.fr/en/english-french-glossary-data-protection.

[54] Toward Privacy-Preserving Personalized Recommendation Services. https://www.sciencedirect.com/science/article/pii/S2095809917303855, August 2013.

[55] A Survey on Norwegian User's Perspective on Privacy in Recommender Systems. http://ceur-ws.org/Vol-2041/paper7.pdf, 2017.

[56] 2018 reform of EU data protection rules. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en, April 2016.

[57] Californians for Consumer Privacy. https://www.caprivacy.org/about, June 2018.

[58] State of California Department of Justice - California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa, June 2018.

[59] Exposure diversity as a design principle for recommender systems. https://www.ivir.nl/publicaties/download/ICS_2016.pdf, December 2016.

[60] Tech's ethical problem is also a diversity problem. https://www.fastcompany.com/90259821/the-answer-to-techs-ethical-problems-is-greater-diversity, November 2018.

[61] Empirical Analysis of the Business Value of Recommender Systems. http://blogs.darden.virginia.edu/venkatesanr/files/2012/07/empirical-value-of-recommender-system-JMiS.pdf, 2006.

[62] OPTiMAL PRICING WiTH RECOMMENDER SYSTEMS. https://cpb-us-w2.wpmucdn.com/campuspress.yale.edu/dist/3/352/files/2011/01/Paper19_p1177.pdf, 2006.