Universitat Oberta de Catalunya

Artificial Intelligence - CET1

Recommenders and clustering

Juan José Rodríguez Aldavero

March 29, 2020

Contents

1 Exercises

1.a Exercise 1

In this exercise we have programmed some functions to analyze the dataset. These functions will serve to find the video game and the user with the highest and lowest number of ratings, as well as the average ratings per video game and user.

A. From the code sol1a.py we see how the video game with the least number of ratings has been V022, with 95 ratings; and the video game with the most ratings has been V096, with 127 ratings. The average number of ratings per videogame was 110.
B. From the code sol1b.py we see how the user with the least number of ratings has been U028, with 35 ratings; and the user with the highest number has been U059, with 73 ratings. The average number of ratings per user was 55.
C. In my opinion, it is very important to pre-process the data before adjusting models such as the predictors discussed in this exercise because this will significantly improve the quality of the predictions. For example, in this dataset there are differences between the number of ratings given by each user: The user with more ratings has more than twice as many ratings as the one with less. This lack of standardization means that, when calculating similarities, one does not have all the precision one wants. For example, it is very important to eliminate outliers from the dataset, as well as to statistically standardize factors such as changes of units, heterocedasticity, etc.
D. It is well known that real data is much noisier than synthetic generated data, as well as having more errors and failures in formatting. This links directly to the previous topic, as all these errors contribute decisively to worsen the accuracy of the models, and therefore a pre-processing is necessary.

1.b Exercise 2

In this exercise we have programmed a set of functions to find the Euclidean distance and Pearson correlation between any two examples of the set. We have also studied the difference between using all the ratings or only those that are shared by the pair of examples. Euclidean distance is defined by the expression:

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

while Pearson's is:

$$r_{xy} = \frac{\sum_{l=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

A. From the code sol2a.py the Euclidean distance between any two points has been calculated taking into account only the shared examples. Using this function, we see that the two most similar users are U008 and U132, with a similarity $s = 0.14827$.

B. From the code sol2b.py we calculate the euclidean distance but in this case taking into account all the ratings of each user and only discarding those video games that have not been rated by either user. The two most similar users are U005 and U063, with a similarity $s = 0.07537$. We can see how it is much lower than the previous similarity.
C. From the code sol2c.py the correlation of Pearson between any two points has been calculated taking into account only the shared examples. Using this function, we see that the two most similar users are U144 and U180, with a correlation $p = 0.66381$.
D. From the code sol2d.py we calculate the Pearson correlation but in this case taking into account all the ratings of each user and only discarding those video games that have not been rated by either user. The two most similar users are U068 and U190, with a correlation $p = 0.36879$.
E. We see how the results are substantially better for cases where we only use shared examples. This is reasonable since the interpolation of considering an intermediate value of 2.5 for the non-shared examples is somewhat arbitrary and distorts the data. We can also see how the results are different for Euclidean similarity and for Pearson's correlation, since they are not magnitudes that can be compared to each other: Euclidean similarity is very sensitive to changes of scale and number of dimensions, while Pearson's correlation measures the distribution of data with respect to the best linear fit between them, and is independent of displacements and changes of scale between data.

1.c Exercise 3

In this exercise we have used the Surprise library to create a predictor using different algorithms.

A. From the code sol3a.py, we calculate a prediction of the score that the user U098 makes of the video game V077, using four algorithms: SVD, KNNBasic, KNNWithMeans and NormalPredictor. The result is the following:

Algorithm	Real value	Predicted value	Difference	Difference (%)
SVD	4	3.44702	-0.55298	-14%
KNNBasic	4	2.67478	-1.32522	-33%
KNNWithMeans	4	2.96901	-1.03099	-26%
NormalPredictor	4	5.00000	1.00000	25%

We see that the most accurate of the four algorithms is SVD, while the least accurate is KNNBasic. This does not have to imply that the SVD algorithm is the most accurate, as we will see later because we have to take into account different factors such as that the initial parameters of each algorithm are initialized randomly, and the optimization of these parameters follows an optimization procedure that will only be optimal for large datasets, much larger than the 200×100 dataset used here.

B. From the code sol3b.py, we calculated a cross-validation of the four algorithms for five partitions, using as a metric RMSE. The results are as follows:

Algorithm	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Mean RMSE
SVD	1,69505	1,66527	1,69146	1,65396	1,70016	1,68118
KNNBasic	1,65021	1,61678	1,64567	1,63915	1,60664	1,63169
KNNWithMeans	1,66111	1,64897	1,63393	1,64716	1,66958	1,65215
NormalPredictor	2,05727	2,01679	2,06329	2,05699	2,03378	2,04562

If we compare these results with those obtained in the previous section, we see how surprisingly the algorithm that worst predicts the previous example (KNNBasic) has the lowest average RMSE value for the cross-validation, while the algorithm with the best prediction (SVD) presents the third best result for RMSE. This is because we have predicted a single rating before, while RMSE takes into account all ratings. In other words, looking only at the difference for one rating is not sufficient to decide the accuracy of the algorithm, while the RMSE measure is more reasonable. It should also be noted that the predictions have a certain randomness, because the initial parameters of the algorithm are initialized at random, and therefore each execution is different.

C. Since there is a random variability in the predictions due to the initialization of the parameters of the algorithms, it is interesting to make a brief statistical analysis by performing several executions for each algorithm and checking their statistics such as the mean and variance. This will allow us to check without any doubt which algorithm has a better performance for this dataset, despite the random fluctuations. This has been done by performing $n = 10$ runs for each algorithm, and the results are presented in the following boxplot:

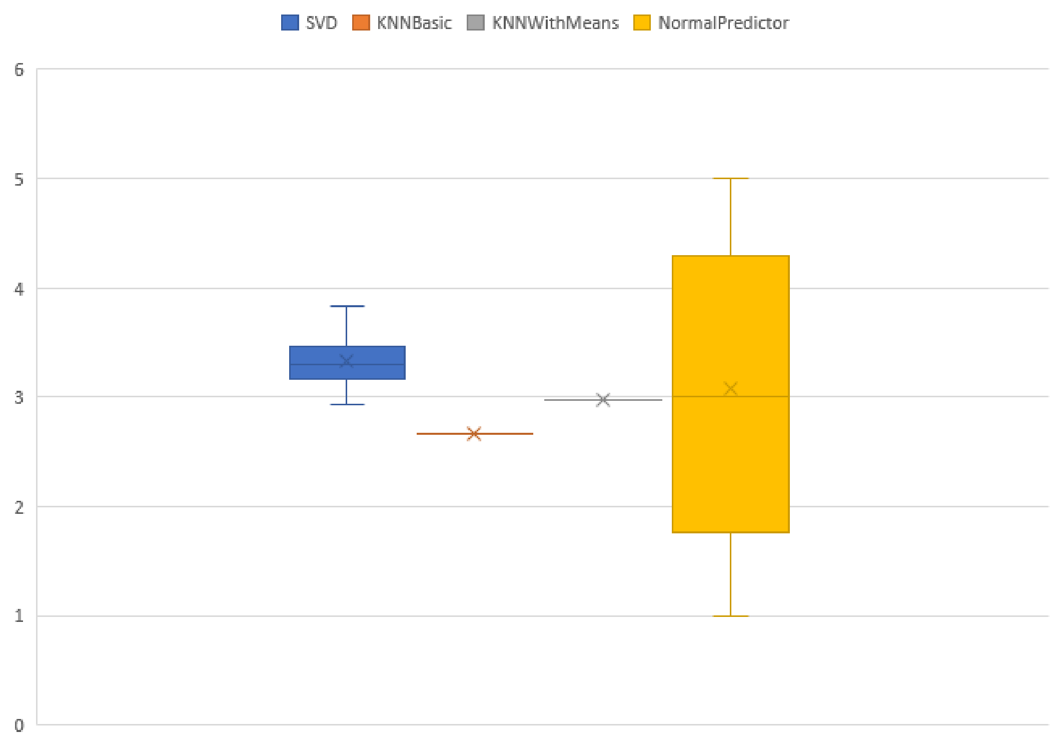

Figure 1.1: *Boxplot of the predicted values for the different algorithms*

We can see how the two KNN algorithms have absolutely no variance, while the other two havw quite a bit of variance. If we observe the statistics, we find the following table:

	SVD	KNNBasic	KNNWithMeans	NormalPredictor
Mean:	3,330	2,670	2,970	3,087
Variance:	0,078	0,000	0,000	1,987

We should not confuse this result with the fact that the KNN algorithms are characterised by low bias and high variance, because this refers to the results of the whole dataset. For example, if we perform a statistical analysis with respect to the RMSE magnitude for cross-validation, with 10 folds:

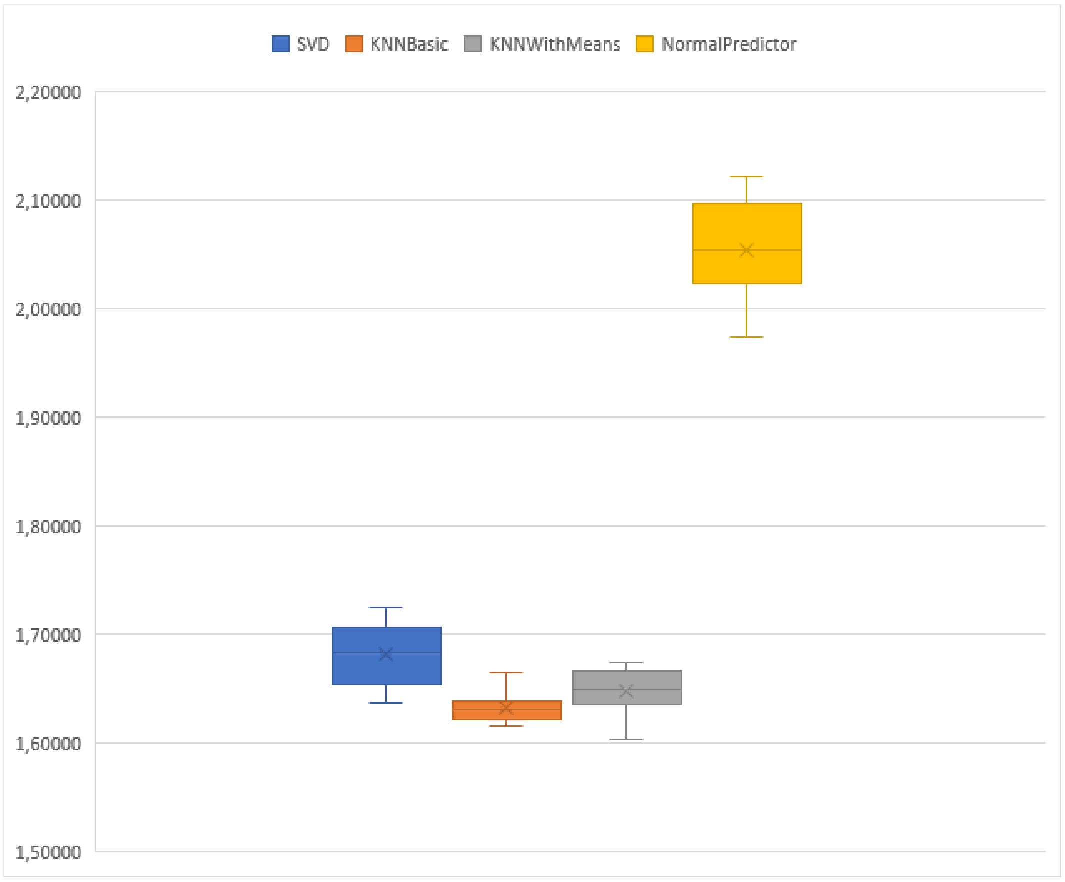

Figure 1.2: *Boxplot of the RMSE for the different algorithms*

We see how this boxplot follows the structure of the previous one, but in this case all algorithms have some degree of variance. We see how the KNN algorithms have a lower degree of variance than the other two, and the algorithm with the lowest RMSE mean is the KNNBasic as we found out before. On the other hand, we see that the NormalPredictor algorithm is the one with the worst performance by far, as opposed to the result of the previous boxplot when where referred only to one prediction.

1.d Exercise 4

In this exercise we make a brief analysis of the methods used from a mathematical and conceptual point of view. In particular, we will see how the K-Nearest-Neighbours algorithms work, as well as the similarity measures used on them. We begin by briefly describing the four algorithms:

A. KNNBasic: This algorithm is the most basic version of the KNN. What it does is to calculate a weighted average of the similarities of the K-Nearest-Neighbors, where the weights of each term correspond to the score given to the predicted video game, i, by each user, v.

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k(u)} \text{sim}(u,v) \cdot r_{vi}}{\sum_{v \in N_i^k(u)} \text{sim}(u,v)}$$

This algorithm could be suitable for simple problems where there is no a priori knowledge of the data and therefore no interest in proposing any specific algorithm, as it is an algorithm caracterised by low bias but high variance (tending to overfit but without infering information about the system).

B. KNNWithMeans: This algorithm is similar to the previous case, with the difference that the weights, r_{vi}, are corrected by their mean scores, which means that we only take into account the deviations from the mean, $r_{vi} - \mu_v$. Finally, to correct this we add the mean of the predicted user, μ_u.

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - \mu_v)}{\sum_{v \in N_i^k(u)} \text{sim}(u,v)}$$

This algorithm may be appropriate when we want to take into account only the deviations from the average of each user. For example, if a user considers a score of 3 out of 5 to be very low, while another user considers it to be high, it may be interesting to consider only the deviations from the average to find out if a rating is really good or bad. I suppose that, because we are adding a new parameter to the data which is the mean μ, we are infering additional information about the system and we are adding bias to the algorithm in exchange for variance.

C. KNNWithZScore: This algorithm is similar to KNNWithMeans, with the only difference that the weights are also standardized by dividing by the variance σ_v. This makes the weights correspond to the standardized deviations (zero mean and unit variance).

$$\hat{r}_{ui} = \mu_u + \sigma_u \frac{\sum_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - \mu_v) / \sigma_v}{\sum_{v \in N_i^k(u)} \text{sim}(u,v)}$$

It can be used when we want to consider only the standardized deviations. For example, when the dataset is large and its data is very diverse so that we only keep the statistically significant deviations. Same as before, because we are adding new information in the form of the σ parameter, this model will be more biased than the normal KNN at the expense of lower varianace.

D. KNNBaseline: This algorithm is similar to KNNWithMeans, but instead of subtracting the mean from each score, we subtract a special parameter called baseline that standardizes each of the weights.

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - b_{vi})}{\sum_{v \in N_i^k(u)} \text{sim}(u,v)}$$

The Surprise library also incorporates measures of similarity between variables, similar to the Euclidean similarity or Pearson's correlation used during the exercise. The most important similarity measures in the library are described below:

A. cosine: Measure of similarity where the cosine is taken between two measurement vectors, u and v. This is a measure of the projection of one vector on the other, since the cosine can be interpreted as the scalar normalized product of both vectors.

$$\text{cosinesim}(u,v) = \frac{\sum_{i \in I_{uv}} r_{ui} \cdot r_{vi}}{\sqrt{\sum_{i \in I_{uv}} r_{ui}^2} \cdot \sqrt{\sum_{i \in I_{uv}} r_{vi}^2}}$$

We can use this magnitude when we assume that the data is correctly represented in Euclidean space, and when we have no more assumptions about the structure of the space. This can be incorrect when, for example, there are internal correlations between different values of space, in which case it may be more convenient to use Pearson correlations or the Mahalanobis metric.

B. MSD: Similarity measure equivalent to the Euclidean similarity used in the exercise. Only common values are summed up, as we have done in the code sol2a.py.

$$\text{msdsim}(u,v) = \frac{1}{\text{msd}(u,v)+1}, \qquad \text{msd}(u,v) = \frac{1}{|I_{uv}|} \cdot \sum_{i \in I_{uv}} (r_{ui} - r_{vi})^2$$

We can use this metric under the same assumptions as for the previous one, since they are inverse (adding a +1 to not divide by zero).

C. pearson: Similarity measure is equivalent to the Pearson correlation used in the problem. Again, only common examples are added, as we have done in the code sol2c.py.

$$\text{pearsonsim}(u,v) = \frac{\sum_{i \in I_{uv}} (r_{ui} - \mu_u) \cdot (r_{vi} - \mu_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \mu_u)^2} \cdot \sqrt{\sum_{i \in I_{uv}} (r_{vi} - \mu_v)^2}}$$

D. pearson baseline: Measure of similarity corresponding to a normalized pearson correlation, where the baseline parameter is subtracted from each of the weights.

$$\text{pearsonbaselinesim}(u,v) = \frac{\sum_{i \in I_{uv}} (r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum_{i \in I_{uv}} (r_{vi} - b_{vi})^2}}$$