Data Mining Methods for Recommender Systems

Xavier Amatriain, Alejandro Jaimes, Nuria Oliver, and Josep M. Pujol

Abstract In this chapter, we give an overview of the main Data Mining techniques that are applied in the context of Recommender Systems. We first describe common preprocessing methods such as sampling or dimensionality reduction. Next, we review a the most important classification techniques, including Bayesian Networks and Support Vector Machines. We describe the so popular *k*-means clustering algorithm and discuss several alternatives. We also present association rules and present algorithms for an efficient training process. In addition to introducing these techniques, we survey their uses in Recommender Systems and present cases where they have been successfully applied.

1 Introduction

Recommender Systems (RS) typically apply techniques and methodologies from other neighboring areas – such as Human Computer Interaction (HCI) or Information Retrieval (IR). However, most of these systems bear in their core an algorithm that can be understood as a Data Mining (DM) technique. In fact, most of the challenges in Data Mining [65] are also challenges in Recommender Systems [1]:

Xavier Amatriain
Telefonica Research, Via Augusta, 122, Barcelona 08021, e-mail: xar@tid.es

Alejandro Jaimes
Telefonica Research, Emilio Vargas, 6, Madrid 28043 e-mail: ajaimes@tid.es

Nuria Oliver
Telefonica Research, Via Augusta, 122, Barcelona 08021 e-mail: nuriao@tid.es

Josep M. Pujol
Telefonica Research, Via Augusta, 122, Barcelona 08021 e-mail: jmps@tid.es

[1] with the important exception of those related to user interface design

Scalability, Dimensionality, Complex and Heterogeneous Data, Data Quality, Data Ownership and Distribution, Privacy Preservation, and Streaming Data.

There are many definitions for Data Mining. In the context of this chapter, we will define Data Mining as the "non-trivial extraction of meaningful information from large amounts of data by automatic or semi-automatic means". Data Mining uses methods and techniques drawn from machine learning, artificial intelligence, statistics, and database systems. However most of these "traditional" techniques need to be adapted to account for the high dimensionality and heterogeneity of data that is pervasive in Data Mining problems.

The process of data mining typically consists of 3 steps, carried out in succession: *Data Preprocessing* [53], *Data Analysis*, and *Result Interpretation* (see Figure 1).

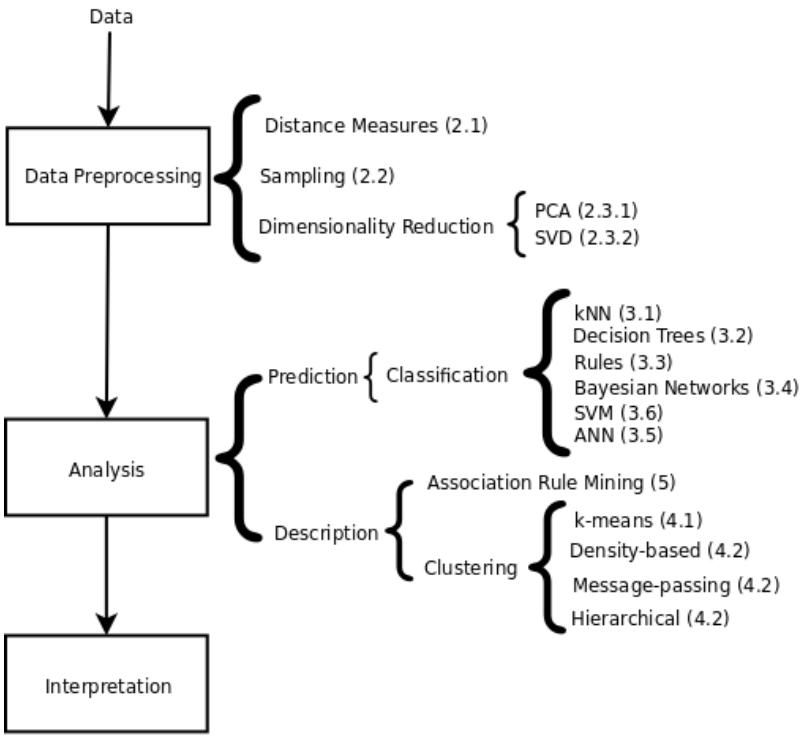


Fig. 1: Main steps and methods in a Data Mining problem, with their correspondence to chapter sections.

We will analyze some of the most important methods for data preprocessing in Section 2. In particular, we will focus on sampling, dimensionality reduction, and the use of distance functions because of their significance and their role in recommender systems.

We usually distinguish two kinds of methods in the analysis step: *predictive* and *descriptive*. Predictive methods use a set of observed variables to predict future or unknown values of other variables. Prediction methods include *classification*, *regression* and *deviation detection*. Descriptive methods focus on finding meaningful patterns that help understand and interpret the data. These include *clustering*, *association rule discovery* and *pattern discovery*. Both kinds of methods can be used in the context of a recommender system.

In Sections 3 through 5, we provide an overview introduction to the analysis methods that are most commonly used in Recommender Systems: classification, clustering and association rule discovery (see Figure 1 for a detailed view of the different topics covered in the chapter).

Note that this chapter does not intend to give a thorough review of Data Mining methods, but rather to highlight the impact that Data Mining algorithms have in the Recommender Systems field, and to provide an overview of the key Data Mining techniques that have been successfully used. We shall direct the interested reader to Data Mining textbooks (see [25, 65], for example) or the more focused references that are provided throughout the chapter. Most of the algorithms and techniques presented in this chapter are also implemented in general purpose machine learning frameworks such as Weka [66] or Torch [17], or even mathematics and statistical packages such as Matlab ® [64] or Octave [22].

2 Data Preprocessing

We define *data* as a collection of *objects* and their *attributes*, where an attribute is defined as a property or characteristic of an object. Other names for object include *record*, *item*, *point*, *sample*, *observation*, or *instance*. An attribute might be also be referred to as a *variable*, *field*, *characteristic*, or *feature*.

There are different types of data with attributes of varied nature. In addition, real-life data typically needs to be *preprocessed* (*e.g.* cleansed, filtered, transformed) in order to be used by the machine learning techniques in the analysis step. There might be missing points, duplicated data, or noise, for instance.

In this section, we focus on three issues that are of particular importance when designing a recommender system. First, we review different similarity or distance measures between data points or collections of data points. Next, we discuss the issue of sampling as a way to reduce the number of items in very large collections while preserving its main characteristics, or as a way to separate a *training* and *testing* data set. Finally, we describe the most common techniques to reduce the dimensionality of the data.

2.1 Similarity Measures

We define *similarity* as a numerical measure – often falling in the $[0,1]$ range – of how alike two items are. Having an appropriate similarity function is a key issue for many data mining algorithms. We usually refer to the *distance* function, d, as a numerical measure of how different two items are.

The most common distance measure is the Euclidean distance:

$$d(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2} \tag{1}$$

where n is the number of dimensions (attributes) and x_k and y_k are the k^{th} attributes (components) of data objects x and y, respectively. Note that in order to compute the Euclidean distance, it is necessary to normalize the data if scales differ.

The Minkowski Distance is a generalization of Euclidean Distance:

$$d(x,y) = (\sum_{k=1}^{n}|x_k - y_k|^r)^{\frac{1}{r}} \tag{2}$$

where r is the degree of the distance. Depending on the value of r, the generic Minkowski distance is known with specific names: For $r = 1$, the *city block*, (*Manhattan, taxicab* or *L1 norm*) distance; For $r = 2$, the *Euclidean* distance; For $r \to \infty$, the *supremum* (L_{max} *norm* or L_∞ *norm*) distance, which corresponds to computing the maximum difference between any dimension of the data objects.

The Mahalanobis distance is defined as:

$$d(x,y) = \sqrt{(x-y)\sigma^{-1}(x-y)^T} \tag{3}$$

where σ is the covariance matrix of the data.

Another very common approach is to consider items as document vectors of an n-dimensional space and compute their similarity as the cosine of the angle that they form:

$$cos(x,y) = \frac{(x \bullet y)}{||x||||y||} \tag{4}$$

where $\bullet$ indicates vector dot product and $||x||$ is the norm of vector x. This similarity is known as the *cosine similarity* or the *L2 Norm*.

The similarity between items can also be given by their *correlation* which measures the linear relationship between objects. While there are several correlation coefficients that may be applied, the *Pearson correlation* is the most commonly used:

$$Pearson(x,y) = \frac{\Sigma(x,y)}{\sigma_x \times \sigma_y} \tag{5}$$

, where Σ is the covariance of data points x and y and σ is their standard deviation.

Finally, several similarity measures have been proposed in the case of items that only have binary attributes. First, the following quantities are computed: *M01* = the

number of attributes where x was 0 and y was 1, $M10$ = the number of attributes where x was 1 and y was 0, $M00$ = the number of attributes where x was 0 and y was 0, $M11$ = the number of attributes where x was 1 and y was 1.

From those quantities we can compute:

1. The *Simple Matching* coefficient (SMC):

$$SMC = \frac{numberofmatches}{numberofattributes} = \frac{M11 + M00}{M01 + M10 + M00 + M11} \tag{6}$$

2. The *Jaccard* coefficient (JC):

$$JC = \frac{M11}{M01 + M10 + M11} \tag{7}$$

3. The *Extended Jaccard (*Tanimoto*) coefficient (EJC): It is a variation of JC for continuous or count attributes.*

$$d = \frac{x \bullet y}{||x||^2 + ||x||^2 - x \bullet y} \tag{8}$$

2.1.1 Similarity Measures in Recommender Systems

The most common approach to collaborative filtering in Recommender Systems is to use the *k*NN classifier that will be described in Section 3.1. This classification method – as most classifiers and clustering techniques – is highly dependent on defining an appropriate similarity measure.

Recommender Systems have traditionally used either the cosine similarity (see Eq. 4) or the Pearson correlation (see Eq. 5) – or one of their many variations through, for instance, weighting schemes – as the similarity measure. However, most of the other distance measures previously reviewed are possible in this context. Spertus *et al.* [62] did a large-scale study to evaluate six different similarity measures in the context of the Orkut social network. Although their results might be biased by the particular setting of their experiment, it is interesting to note that the best response to recommendations were to those generated using the cosine similarity. Lathia *et al.* [44] also carried out a study of several similarity measures where they concluded that, in the general case, the prediction accuracy of a recommender system was *not* affected by the choice of the similarity measure. As a matter of fact and in the context of their work, using a random similarity measure sometimes yielded better results than using any of the well-known approaches.

2.2 Sampling

Sampling is the main technique used in data mining for selecting a subset of relevant data from a large data set. It is used both in the preprocessing and final data

interpretation steps. Sampling may be used because processing the entire data set is computationally too expensive. It can also be used to create *training* and *testing* datasets. In this case, the training dataset is used to learn the parameters or configure the algorithms used in the analysis step, while the testing dataset is used to evaluate the model or configuration obtained in the training phase, making sure that it performs well (*i.e. generalizes*) with previously unseen data.

The key issue to sampling is finding a subset of the original data set that is *representative – i.e.* it has approximately the same property of interest – of the entire set. The simplest sampling technique is *random sampling*, where there is an equal probability of selecting any item. However more sophisticated approaches are possible. For instance, in *stratified sampling* the data is split into several partitions based on a particular feature, followed by random sampling on each partition independently.

The most common approach to sampling consists of using sampling *without replacement*: When an item is selected, it is removed from the population. However, it is also possible to perform sampling *with replacement*, where items are not removed from the population once they have been selected, allowing for the same sample to be selected more than once.

It is common practice to use standard random sampling without replacement with an $80/20$ proportion when separating the training and testing data sets. This means that we use random sampling without replacement to select 20% of the instances for the testing set and leave the remaining 80% for training. Note that the $80/20$ proportion should be taken as a rule of thumb: It is generally the case that any value over $2/3$ for the training set is appropriate.

Sampling can lead to an over-specialization to the particular division of the training and testing data sets. For this reason, the training process is repeated K times as follows: the training and test sets are created from the original data set, the model is trained using the training data and tested with the examples in the test set. Next, different training/test data sets are selected to start the training/testing process again that is repeated K times. Finally, the *average* performance of the K learned models is reported.

This process is known as *cross-validation*. There are several cross-validation techniques. In *repeated random sampling*, a standard random sampling process is carried out n times. In *n-Fold cross validation*, the data set is divided into n folds. One of the folds is used for testing the model and the remaining $n-1$ folds are used for training. The cross validation process is then repeated n times with each of the n subsamples used exactly once as validation data. Finally, the *leave-one-out (LOO)* approach can be seen as an extreme case of n-Fold cross validation where n is set to the number of items in the data set. Therefore, the algorithms are run as many times as data points using only one of them as a test each time. It should be noted, though, that as Isaksson *et al.* discuss in [40], cross-validation may be unreliable unless the data set is sufficiently large.

2.3 Reducing Dimensionality

It is common in Recommender Systems to have not only a data set with features that define a high-dimensional space, but also very sparse information in that space – *i.e.* there are values for a limited number of features per object. The notions of density and distance between points, which are critical for clustering and outlier detection, become less meaningful in highly dimensional spaces. This is known as the *Curse of Dimensionality*. Dimensionality reduction techniques, as the ones reviewed in this section, help overcome this problem by converting the original high-dimensional space to a lower-dimensionality space. In addition, some algorithms not only address the problems of data sparsity, but they also bring in welcomed side-effects such a reduction in the noise or improved computational efficiency.

In the following, we summarize the two most relevant dimensionality reduction algorithms in the context of Recommender Systems: *Principal Component Analysis (PCA)* and *Singular Value Decomposition (SVD)*. These techniques have become so popular (see 2.3.3) that they are considered as independent approaches to Recommender Systems in themselves. However, they can be used as a preprocessing step for any of the other techniques that will be reviewed in this chapter.

2.3.1 Principal Component Analysis

Principal Component Analysis (PCA) [41] is a classical statistical method to find patterns in high dimensionality data sets. PCA allows to obtain an ordered list of components that account for the largest amount of the variance from the data in terms of least square errors: The amount of variance captured by the first component is larger than the amount of variance on the second component and so on. We can reduce the dimensionality of the data by neglecting those components with a small contribution to the variance.

Given a data matrix $A_{n\times m}$ of n samples with m attributes (dimensions) we can perform the principal component analysis using the algorithm in listing 1.

Figure 2 shows the PCA analysis to a two-dimensional point cloud generated by a combination of Gaussians. After the data is centered, the principal components are obtained and denoted by u_1 and u_2. Note that the length of the new coordinates is relative to the energy contained in their eigenvectors. Therefore, for the particular example depicted in Fig 2, the first component u_1 accounts for 83.5% of the energy, which means that removing the second component u_2 would imply losing only 16.5% of the information.

The rule of thumb is to choose m' so that the cumulative energy is above a certain threshold, typically 90%. PCA allows us to retrieve the original data matrix by projecting the data onto the new coordinate system $X'_{n\times m'} = X_{n\times m}W'm \times m'$. The new data matrix X' contains most of the information of the original X with a dimensionality reduction of $m - m'$.

Although PCA is a powerful technique, it does have important limitations. PCA relies on the empirical data set to be a linear combination of a certain basis. For non-

Algorithm 1 PCA algorithm

1: *Substract the mean*. For PCA to work properly, the mean of each of the data dimensions must be zero. Thus the mean is subtracted from each attribute (column): $A_{.j} = A_{.j} - \frac{1}{n}\sum_i A_{ij}, \forall j \in \{1..m\}$
2: *Covariance matrix*. Compute the covariance matrix of data A, centered at the origin as $C = \frac{1}{n-1}A^T A$. The covariance matrix C will be a square matrix of dimensionality m.
3: *Calculate Eigenvectors and Eigenvalues*. Compute the matrix V of eigenvectors which diagonalize the covariance matrix C as $V^{-1}CV = D$, where V contains the eigenvector and the diagonal of D contains the eigenvalues ($\{\lambda_1 ... \lambda_m\}$).
4: *Rearrange eigenvectors and eigenvalues*. Once the eigenvectors are computed – which are the principal components of the analysis, they are sorted in decreasing value of their eigenvalues and arranged in a matrix W of dimensionality m: The first principal component – which captures most of the data variation – is the eigenvector with the highest eigenvalue.
5: *Compress the data*. The dimensionality of the principal component matrix W can be reduced by keeping only the first m' eigenvectors (W'). The loss of information by discarding an eigenvector j is the fraction of the eigenvector's energy that is $\frac{\lambda_j}{\sum_i \lambda_i}$, where λ_j is the eigenvalue of the j-th eigenvector.

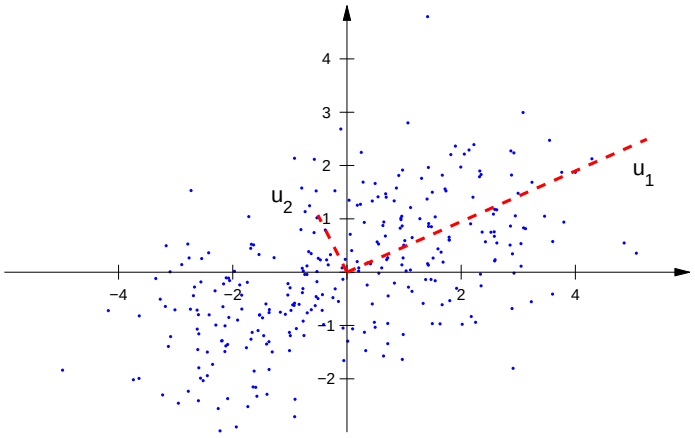


Fig. 2: PCA analysis of a two-dimensional point cloud from a combination of Gaussians. The principal components derived using PCS are u_1 and u_2, whose length is relative to the energy contained in the components.

linear data, generalizations of PCA have been proposed, such as Kernel PCA [**?**]. Another important assumption of PCA is that the original data set has been drawn from a Gaussian distribution. When this assumption does not hold true, as it is the case of multi-modal Gaussian or non-Gaussian distributions, there is no warranty that the principal components are meaningful.

2.3.2 Singular Value Decomposition

Singular Value Decomposition [34] is a powerful technique for dimensionality reduction that is related to PCA. The key issue in an SVD decomposition is to find a lower dimensional feature space where the new features represent "concepts" and the strength of each concept in the context of the collection is computable. Because SVD allows to automatically derive semantic "concepts" in a low dimensional space, it is the basis of *latent-semantic analysis* [57], a very popular technique for text classification in Information Retrieval .

The core of the SVD algorithm lies in the following theorem: It is always possible to decompose a given matrix A into $A = U\lambda V^T$. Given the $n \times m$ matrix data A (n items, m features), we can obtain an $n \times r$ matrix U (n items, r concepts), an $r \times r$ diagonal matrix λ (strength of each concept), and an $m \times r$ matrix V (m features, r concepts). Figure 3 illustrates this idea.

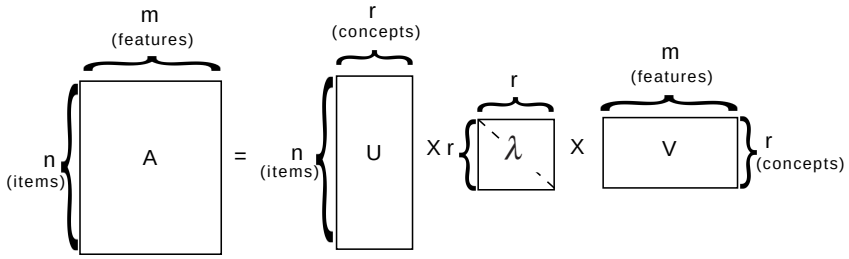


Fig. 3: Illustrating the basic Singular Value Decomposition Theorem: an item $\times$ features matrix can be decomposed into three different ones: an item $\times$ concepts, a concept strength, and a concept $\times$ features.

The λ diagonal matrix contains the *singular values*, which will always be positive and sorted in decreasing order. The U matrix is interpreted as the "item-to-concept" similarity matrix, while the V matrix is the "term-to-concept" similarity matrix.

In order to compute the SVD of a rectangular matrix A, we consider AA^T and A^TA. The columns of U are the eigenvectors of AA^T, and the columns of V are the eigenvectors of A^TA. The singular values on the diagonal of λ are the positive square roots of the nonzero eigenvalues of both AA^T and A^TA. Therefore, in order to compute the SVD of matrix A we first compute T as AA^T and D as A^TA and then compute the eigenvectors and eigenvalues for T and D.

The r eigenvalues in λ are ordered in decreasing magnitude. Therefore, the original matrix A can be approximated by simply truncating the eigenvalues at a given k. The truncated SVD creates a rank-k approximation to A so that $A_k = U_k \lambda_k V_k^T$. A_k is the *closest* rank-k matrix to A. The term "closest" means that A_k minimizes the sum of the squares of the differences of the elements of A and A_k. The truncated SVD is a representation of the underlying latent structure in a reduced k-dimensional space, which generally means that the noise in the features is reduced.

2.3.3 Dimensionality Reduction in Recommender Systems

Sparsity and the *curse of dimensionality* are recurring problems in Recommender Systems. Even in the simplest setting, we are likely to have a sparse matrix with thousands of rows and columns (*i.e.* users and items), most of which are zeros. Therefore, dimensionality reduction comes in naturally. Applying dimensionality reduction makes such a difference and its results are so directly applicable to the computation of the predicted value, that this is now considered to be an approach to Recommender Systems design, rather than a preprocessing technique. As a matter of fact, the two preferred approaches to collaborative filtering are nowadays standard kNN and its many variations, and dimensionality reduction via SVD [].

Earlier works, however, used PCA as a way to reduce dimensionality in a collaborative filtering setting. Goldberg *et al.* proposed an approach to use PCA in the context of an online joke recommendation system [33]. Their system, known as Eigentaste [2], starts from a standard matrix of user ratings to items. They then select their *gauge* set by choosing the subset of items for which all users had a rating. This new matrix is then used to compute the global correlation matrix where a standard 2-dimensional PCA is applied.

The use of SVD as tool to improve collaborative filtering has been known for some time. Sarwar *et al.* [60] describe two different ways to use SVD in this context. First, SVD can be used to uncover latent relations between customers and products. In order to accomplish this goal, they first fill the zeros in the user-item matrix with the item average rating and then normalize by subtracting the user average. This matrix is then factored using SVD and the resulting decomposition can be used – after some trivial operations – directly to compute the predictions. The other approach is to use the low-dimensional space resulting from the SVD to improve neighborhood formation for later use in a kNN approach.

As described by Sarwar *et al.* [59], one of the big advantages of SVD is that there are incremental algorithms to compute an approximated decomposition. This allows to accept new users or ratings without having to recompute the model that had been built from previously existing data. The same idea was later extended and formalized by Brand [11] into an online SVD model. The use of incremental SVD methods has recently become a commonly accepted approach after its success in

[2] http://eigentaste.berkeley.edu

the Netflix Prize [3]. The publication of Simon Funk's simplified incremental SVD method [31] marked an inflection point in the contest. Since its publication, several improvements to SVD have been proposed in this same context (see Paterek's ensembles of SVD methods [50] or Kurucz *et al.* evaluation of SVD parameters [43]).

Finally, it should be noted that different variants of Matrix Factorization (MF) methods such as the Non-negative Matrix Factorization (NNMF) have also been used [67]. These algorithms are, in essence, similar to SVD. The basic idea is to decompose the ratings matrix into two matrices, one of which contains features that describe the users and the other contains features describing the items. Matrix Factorization methods are better than SVD at handling the missing values by introducing a bias term to the model. However, this can also be handled in the SVD preprocessing step by replacing zeroes with the item average. Note that both SVD and MF are prone to overfitting. However, there exist MF variants, such as the Regularized Kernel Matrix Factorization [], that can avoid the issue efficiently. The main issue with MF – and SVD – methods is that it is unpractical to recompute the factorization every time the matrix is updated because of computational complexity. However, Rendle and Schmidt-Thieme [56] propose an online method that allows to update the factorized approximation without recomputing the entire model.

3 Classification

A classifier is a mapping between a feature space and a label space, where the features represent characteristics of the elements to classify and the labels represent the classes. A restaurant recommender system, for example, can be implemented by a classifier that classifies restaurants into one of two categories (good, bad) based on a number of features that describe it (*e.g.*, quality of food on a scale from 1 to 10, atmosphere on a scale from 1 to 10, etc.). A particular restaurant *R* will be represented by a feature vector $FV_r =< fv_1, fv_2, , fv_n >$. In this particular example, the classifier is binary because it produces only two labels: good or bad.

There are many types of classifiers, but in general they will either be *supervised* or *unsupervised*.

- In supervised classification, a set of labels or categories is known in advance (e.g., we know there are two types of restaurants, good and bad) and we have a set of labeled examples which constitute a training set (we know in advance which restaurants are good and which are bad). The task is then to learn a mapping (boundary, or function) that can separate the instances (good from bad restaurants) so that if a new unseen instance (restaurant) is presented to the classifier it can predict its category (good, bad).
- In unsupervised classification, the labels or categories are unknown in advance and the task is to suitably (according to some criteria) organize the elements at hand (e.g., given a list of restaurants, put them into groups considering all or

[3] http://www.netflixprize.com

some of their characteristics: quality of food, price, location, etc.). Following this example, an unsupervised learning algorithm might discover two groups of restaurants in a list where it might turn out that one group has only French restaurants and the other one only American restaurants although the labels "French" and "American" did not exist in the feature vectors. Unsupervised classification is accomplished by means of *clustering* algorithms, which will be covered in section 4.

In essence, then, classifiers try to find boundary functions to separate or group elements into either known categories or into groups of similar elements. In this section we describe several algorithms to learn supervised classifiers and will be covering unsupervised classification in section 4.

3.1 Nearest Neighbors

Instance-based classifiers work by storing training records and using them to predict the class label of unseen cases. A trivial example is the so-called *rote-learner*. This classifier memorizes the entire training set and classifies only if the attributes of the new record match one of the training examples exactly.

A more elaborate, and far more popular, instance-based classifier is the *Nearest neighbor classifier* (*k*NN) [19]. Given a point to be classified, the *k*NN classifier finds the *k* closest points (*nearest neighbors*) from the training records. It then assigns the class label according to the class labels of its *nearest-neighbors*. The underlying idea is that if a record falls in a particular neighborhood where a class label is predominant it is because the record is likely to belong to that very same class.

Given a query point q for which we want to know its class l, and a training set $X = \{\{x_1, l_1\}...\{x_n\}\}$, where x_j is the j-th element and l_j is its class label, the k-nearest neighbors will find a subset $Y = \{\{y_1, l_1\}...\{y_k\}\}$ such that $Y \in X$ and $\sum_1^k d(q, y_k)$ is minimal. Y contains the k points in X which are closest to the query point q. Then, the class label of q is $l = f(\{l_1...l_k\})$.

The distance measure d is usually the Euclidian distance. However, other measures, such as the ones reviewed in 2.1, can be applied depending on the data. In order to prevent the distance measure from being dominated by some of the attributes, it is common-practice to scale attributes. Furthermore, and in order to avoid counter-intuitive results, we sometimes normalize vectors to unit length.

There are different candidates for the function f by which the new class label is assigned. The most widely used is the majority vote rule with ties broken at random. With majority vote, the query point q is assigned to the most common label of its nearest neighbors. A variation of the majority vote is to weight the votes according to the distance between the training points y_k and the query point q; the vote of the closest neighbor counts more than the vote of the furthest – this is one of preferred approach when using *k*NN in a collaborative filtering setting. Another strategy is to use the consensus rule. Unlike the majority vote rule, consensus only assigns the label if and only if all k neighbors have the same class label. This technique is useful

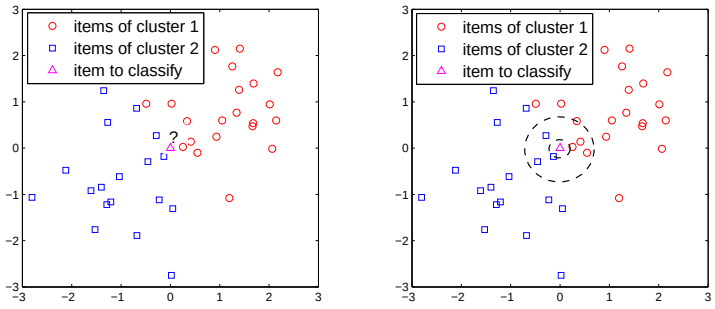


Fig. 4: Example of k-Nearest Neighbors. The left subfigure shows the training points with two class labels (circles and squares) and the query point (as a triangle). The right sub-figure illustrates closest neighborhood for $k = 1$ and $k = 7$. The query point would be classified as square for $k = 1$, and as a circle for $k = 5$ according to the simple majority vote rule. Note that the query points was just on the boundary between the two clusters.

to discriminate the classification in terms of *certainty*, however, many query points might remain unclassified due to the strictiveness of this criterium.

Perhaps the most challenging issue in *k*NN is how to choose the value of *k*. If *k* is too small, the classifier will be sensitive to noise points. But if k is too large, the neighborhood might include too many points from other classes. The right plot in Fig. 4 shows how different *k* yields different class label for the query point, if $k = 1$ the class label would be *circle* whereas $k = 7$ classifies it as *square*. Note that the query point from the example is on the boundary of two cluster, and therefore, it is difficult to classify.

*k*NN classifiers are amongst the simplest of all machine learning algorithms. Since *k*NN does not build models explicitly it is considered a *lazy learner*. Unlike eager learners such as decision trees or rule-based systems, *k*NN classifiers leave many decisions to the classification step. Therefore, classifying unknown records is relatively expensive.

3.1.1 Nearest Neighbors in Recommender Systems

Nearest Neighbor is one of the most common approaches to collaborative filtering (and therefore to designing a recommender systems). As a matter of fact, any overview on Recommender Systems – such as the one by Adomavicius and Tuzhilin [1] – will include an introduction to the use of nearest neighbors in this context.

One of the advantages of this classifier is that it is conceptually very much related to the idea of collaborative filtering: Finding like-minded users (or similar items) is essentially equivalent to finding neighbors for a given user or an item.

The other advantage is that, being the *k*NN classifier a lazy learner, it does not require to learn and maintain a given model. Therefore, in principle, the system can adapt to rapid changes in the user ratings matrix. Unfortunately, this comes at the cost of recomputing the neighborhoods and therefore the similarity matrix.

The *k*NN approach, although simple and intuitive, has shown good accuracy results and is very amenable to improvements. As a matter of fact, its supremacy as the *de facto* standard for collaborative filtering recommendation has only been challenged recently by approaches based on dimensionality reduction such as the ones reviewed in Section 2.3.

The general *k*NN approach to collaborative filtering has experienced improvements in several directions. For instance, in the context of the Netflix Prize, Bell and Koren [7] propose a method to remove *global effects* such as the fact that some items may attract users that consistently rate lower. They also propose an optimization method for computing interpolating weights once the neighborhood is created.

3.2 Decision Trees

Decision trees [55] are classifiers on a target attribute (or class) in the form of a tree structure. The observations (or items) to classify are composed of attributes and their target value. The nodes of the tree can be: a) *decision nodes*, in these nodes a single attribute-value is tested to determine to which branch of the subtree applies. Or b) *leaf nodes* which indicate the value of the target attribute.

Figure 5 is a decision tree of the data contained in Table 1. In this toy example, the goal is to classify potential pizza-lovers as a function of three attributes (marital status, annual income and interest in sports). The tree will then be used to predict the risk of future borrowers based on historic data.

sub:similarity

Sports' fan	*Marital Status*	*Annual income*	*Likes Pizza*
Yes	Divorced	90K	Yes
No	Single	125K	No
Yes	Married	100K	No
Yes	Married	60K	No
Yes	Married	75K	No
Yes	Single	105K	No
Yes	Single	85K	Yes
Yes	Single	90K	Yes
No	Divorced	220K	No
No	Married	120K	No

Table 1: Attributes and target attribute from the observations

There are many algorithms for decision tree induction: Hunts Algorithm, CART, ID3, C4.5, SLIQ, SPRINT to mention the most common. We briefly describe Hunt's

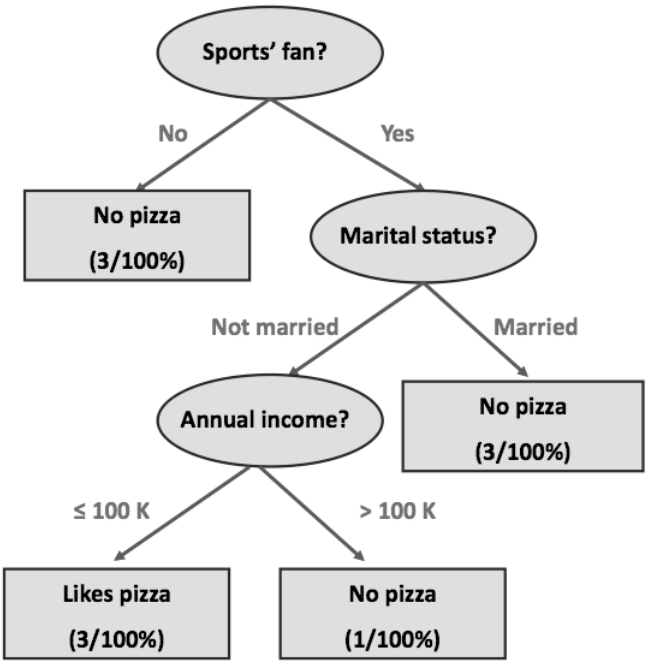


Fig. 5: Example of a Decision Tree for the data summarized in Table 1

Algorithm – used in the toy example – for it is one of the earliest and easiest to understand. The recursive Hunt algorithm is described in Listing 2.

The algorithm relies on the *test condition* applied to a given attribute that discriminates the observations by their target values. Once the partition induced by the test condition has been found, the algorithm is recursively repeated until a partition is empty or all the observations have the same target value. In Fig. 5 there are three test conditions, one for each attributed in the data. The first *test condition* was to ask for sport's supporters since there is no observation of a non sports's fan liking pizza. Applying the condition will create the split of the data in two new nodes: a) non sports' fans with three instances and b) sports' fans with seven instances. Why was sports chosen to do the split instead of another attribute? Because the partition yielded by sports maximized the information gain, defined as follows,

$$\Delta_i = I(parent) - \sum_{j=1}^{k_i} \frac{N(v_j)I(v_j)}{N} \tag{9}$$

where k_i are values of the attribute i, N is the number of observations, v_j is the j-th partition of the observations according to the values of attribute i. Finally, I is a function that measures node *impurity*. There are different measures of impu-

Algorithm 2 Hunt algorithm

1: $D_t = \{(X_{i1},...,X_{ip},Y_i), \forall i \in N\}$
2: (the N observations to classify in D_t, Y_i is the target attribute of the i-th instance or observation)
3: c (the current node)
4:
5: **procedure** Hunt(D_t, c)
6:
7: **if** same value for all Y_i in D_t **then**
8: mark c as leaf node with value Y_i
9: **else**
10: use *test condition* to split D_t in Q different sets of observations $D_{t1}..D_{tq}$
11: according the values of an attribute j X_{ij}
12: **for** $i \in Q$ **do**
13: Hunt($D_t i$,i)
14: **end for**
15: **end if**
16:
17: **end procedure**

rity: Gini Index, Entropy and misclassification error are the most common in the literature. We used misclassification to build up the example depicted in Fig. 5. We computed the information Δ for each attribute and selected sports since it maximized the information gain. Then, the original observations are split into two new nodes and the process is repeated.

Note that the *test condition* selection process uses a greedy hill-climbing strategy. Decision trees can, therefore, get stuck in a local optimal classification. The *test condition* is also sensitive to the attribute type (i.e. nominal, ordinal, continuous...) and whether we decide to do a 2-way split or a multi-way split. However, these two issues are not uncorrelated. For instance, if we base our partition on nominal attributes we will favor a multi-way split using as many partitions as distinct values. We have different ways to handle splitting conditions based on continuous attributes. We can discretize any continuous attribute to form an ordinal attribute following either a static or dynamic approach. In the static, we discretize the attribute once at the beginning. In the dynamic, ranges can be found by equal interval bucketing, equal frequency bucketing, or clustering.

We already mentioned that the decision tree stops once all observations belong to the same class (or the same range in the case of continuous attributes). This implies that the impurity of the leaf nodes is zero. For practical reasons, however, most decision trees implementations use pruning by which a node is no further split if its impurity measure or the number of observations in the node are below a certain threshold. This early termination criteria are used to improve the efficiency of the algorithm. Early termination avoids too fine-grained splits that might be irrelevant in the prediction stage or could be over-fitting to the training data.

The main advantages of building a classifier using a decision tree is that it is inexpensive to construct and it is extremely fast at classifying unknown instances.

Another appreciated aspect of decision tree is that they can be used to produce a set of rules that are easy to interpret (see section 3.3) while maintaining an accuracy comparable to other basic classification techniques.

3.2.1 Decision Trees in Recommender Systems

Decision trees may be used in a model-based approach for a recommender system. One possibility is to use content features to build a decision tree that models all the variables involved in the user preferences. Bouza *et al.* [9] use this idea to construct a Decision Tree using semantic information available for the items. The tree is built after the user has rated only two items. The features for each of the items are used to build a model that explains the user ratings. They use the information gain of every feature as the splitting criteria. It should be noted that although this approach is interesting from a theoretical perspective, the precision they report on their system is worse than that of recommending the average rating.

As it could be expected, it is very difficult and unpractical to build a decision tree that tries to explain all the variables involved in the decision making process. Decision trees, however, may also be used in order to model a particular part of the system. Cho *et al.* [14], for instance, present a Recommender System for online purchases that combines the use of Association Rules (see Section 5) and Decision Trees. The Decision Tree is used as a filter to select which users should be targeted with recommendations. In order to build the model they create a candidate user set by selecting those users that have chosen products from a given category during a given time frame. In their case, the dependent variable for building the decision tree is chosen as whether the customer is likely to buy new products in that same category.

3.3 Ruled-based Classifiers

Rule-based classifiers classify data by using a collection of "**if** ... **then** ..." rules. The rule *antecedent* or condition is an expression made of attribute conjunctions. The rule *consequent* is a positive or negative classification.

We say that a rule *r covers* a given instance *x* if the attributes of the instance satisfy the rule condition. We define the *coverage* of a rule as the fraction of records that satisfy its antecedent. On the other hand, we define its *accuracy* as the fraction of records that satisfy both the antecedent and the consequent. We say that a classifier contains *mutually exclusive rules* if the rules are independent of each other – *i.e.* every record is covered by at most one rule. Finally we say that the classifier has *exhaustive rules* if they account for every possible combination of attribute values –i.e. each record is covered by at least one rule.

In order to build a rule-based classifier we can follow a direct method to extract rules directly from data. Examples of such methods are RIPPER, or CN2. On the

other hand, it is common to follow an indirect method and extract rules from other classification models such as decision trees or neural networks.

For instance the rules derived from applying a decision tree 5 to the data of Table 1 would be:

1. *IF NOT* Sports' fan *THEN NOT* Pizza (coverage 30%, accuracy 100%)
2. *IF* Sports' fan *AND* Married *THEN NOT* Pizza (coverage 30%, accuracy 100%)
3. *IF* Sports' fan *AND NOT* Married *THEN* Pizza (coverage 40%, accuracy 75%)

Note that we excluded the annual income attribute for the sake of illustrating the coverage and accuracy. The advantages of rule-based classifiers are that they are extremely expressive since they are symbolic and operate with the attributes of the data without any transformation. Rule-based classifiers, and by extension decision trees, are easy to interpret, easy to generate and they can classify new instances efficiently.

3.3.1 Rule-based Classifiers in Recommender Systems

In a similar way to Decision Tress, it is very difficult to build a complete recommender model based on rules. As a matter of fact, this method is not very popular in the context of recommender systems because deriving a rule-based system means that we either have some explicit prior knowledge of the decision making process or that we derive the rules from another model such a decision tree. However a rule-based dystem can be used to improve the performance of a recommender system by injecting partial domain knowledge or business rules.

Anderson *et al.* [3], for instance, implemented a collaborative filtering music recommender system that improves its performance by applying a rule-based system to the results of the collaborative filtering process. If a user rates an album by a given artist high, for instance, predicted ratings for all other albums by this artist will be increased.

Gutta *et al.* [26] implemented a rule-based recommender system for TV content. In order to do, so they first derived a C4.5 Decision Tree that is then decomposed into rules for classifying the programs.

Basu *et al.* [5] followed an inductive approach using the *Ripper* [16] system to learn rules from data. They report slightly better results when using hybrid content and collaborative data to learn rules than when following a pure collaborative filtering approach.

3.4 Bayesian Classifiers

A Bayes classifier [30] is a probabilistic framework for solving classification problems. It is based on the definition of conditional probability and the Bayes theorem. The Bayesian school of statistics uses probability to represent uncertainty about the

relationships learned from the data. In addition, the concept of *priors* is very important as they represent our expectations or prior knowledge about what the true relationship might be. In particular, the probability of a model given the data (*posterior*) is proportional to the product of the *likelihood* times the *prior probability* (or prior). The likelihood component includes the effect of the data while the prior specifies the belief in the model before the data was observed.

Bayesian classifiers make use of Bayes' theorem, that relates all the previous concpts, and is given by:

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)} \tag{10}$$

where M is a model (or hypothesis) and D is the data. $P(M)$ is the prior probability of M, *i.e.* the probability that M is correct before the data D is observed; $P(D|M)$ is the *conditional* probability of seeing data D given that model M is true. This is called the *likelihood* of the data; $P(D)$ is the *marginal* probability of the data and $P(M|D)$ is the *posterior* probability, *i.e.* the probability that model M is true, given the data.

If we assume an exhaustive set of mutually exclusive models M_i, we obtain:

$$P(D) = \sum_i P(D, M_i) = \sum_i P(D|M_i)P(M_i) \tag{11}$$

Note that $P(D)$ in Equation 10 is a normalizing constant that only depends on the data and in most cases does not need to be computed explicitly. As a result, Bayes' theorem is typically simplified to $P(M|D) \propto P(D|M)P(M)$.

Bayesian classifiers consider each attribute and class label as (continuous or discrete) random variables. Given a record with N attributes $(A_1, A_2, ..., A_N)$, the goal is to predict class C_k by finding the value of C_k that maximizes the posterior probability of the class given the data $P(C_k|A_1, A_2, ..., A_N)$. Applying Bayes' theorem,

$$P(C_k|A_1, A_2, ..., A_N) \propto P(A_1, A_2, ..., A_N|C_k)P(C_k) \tag{12}$$

A particular but very common Bayesian classifier is the *Naive Bayes Classifier*. In order to estimate the conditional probability, $P(A_1, A_2, ..., A_N|C_k)$, a Naive Bayes Classifier assumes the probabilistic *independence* of the attributes – *i.e.* the presence or absence of a particular attribute is unrelated to the presence or absence of any other. This assumption leads to

$$P(A_1, A_2, ..., A_N|C_k) = P(A_1|C_k)P(A_2|C_k)...P(A_N|C_k) \tag{13}$$

If conditional probabilities are zero, then the entire expression becomes zero so the Naive Bayes Classifier will not be able to classify the instance. In this case, we can use the *m-estimate* approach for estimating conditional probabilities:

$$P(A_i|C_k) = \frac{n_c + mp}{n + m} \tag{14}$$

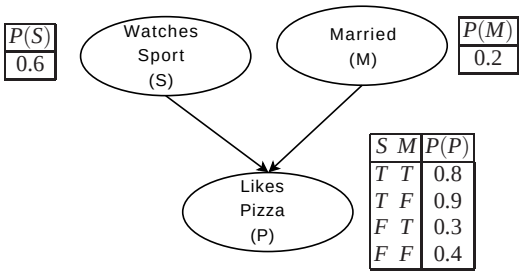


Fig. 6: Example of a Bayesian Belief Network

where n is the number of training instances in class C, n_c is the number of training instances belonging to class C with attribute A_i, p is the prior estimation of the probability (usually set to one over the number of values of the attribute we are considering), and m is a parameter known as the *equivalent sample size*.

Another issue with Bayesian classifiers is that the computation of each $P(A_i|C_k)$ depends on the nature of the attribute that we are dealing with. In the case of discrete attributes, $P(A_i|C_k) = \frac{|A_i^k|}{Nc}$, where $|A_i^k|$ is number of instances that have attribute A_i and belong to class C_k. Continuous attributes are typically discretized.

The main benefits of Naive Bayes classifiers are that they are robust to isolated noise points and irrelevant attributes, and they handle missing values by ignoring the instance during probability estimate calculations.

However, the independence assumption may not hold for some attributes as they might be correlated. In this case, the usual approach is to use the so-called *Bayesian Belief Networks (BBN)* (or Bayesian Networks, for short). BBN's use an acyclic graph to encode the dependence between attributes and a probability table that associates each node to its immediate parents (see Fig. 6). If a node A does not have any parent, the table contains only prior probability $P(A)$; if the node has only one parent B, the table contains the conditional probability $P(A|B)$; and if the node has multiple parents, the table contains the conditional probability $P(A|B_1, B_2, ..., B_3)$. BBN's provide a way to capture prior knowledge in a domain using a graphical model. And, although constructing the model is non-trivial, once the structure of the network is determined it is quite easy to add a new variable. In a similar way to Naive Bayes classifiers, BBN's handle incomplete data well and they are quite robust to model overfitting.

3.4.1 Bayesian Classifiers in Recommender Systems

Bayesian classifiers are particularly popular for model-based recommender systems. They are often used to derive a model for content-based recommender systems. However, they have also been used in a collaborative filtering setting.

Ghani and Fano [32], for instance, use a Naive Bayes classifier to implement a content-based recommender system. The use of this model allows for recommending products from unrelated categories in the context of a department store.

Miyahara and Pazzani [48] implement a recommender system based on a Naive Bayes classifier. In order to do so, they define two classes: *like* and *don't like*. In this context they propose two ways of using the Naive Bayesian Classifier: The *Transformed Data Model* assumes that all features are completely independent, and feature selection is implemented as a preprocessing step. On the other hand, the *Sparse Data Model* assumes that only known features are informative for classification. Furthermore, it only makes use of data which both users rated in common when estimating probabilities. Experiments show both models to perform better than a correlation-based collaborative filtering.

Pronk *et al.* [52] use a Bayesian Naive Classifier as the base for incorporating user control and improving performance, especially in cold-start situations. In order to do so they propose to maintain two profiles for each user: one learned from the rating history, and the other explicitly created by the user. The blending of both classifiers can be controlled in such a way that the user-defined profile is favored at early stages, when there is not too much rating history, and the learned classifier takes over at later stages.

In the previous section we mentioned that Gutta *et al.* [26] implemented a rule-based approach in a TV content recommender system. Another of the approaches they tested was a Bayesian classifier. They define a two-class classifier, where the classes are *watched*/*not watched*. The user profile is then a collection of attributes together with the number of times they occur in positive and negative examples. This is used to compute prior probabilities that a show belongs to a particular class and the conditional probability that a given feature will be present if a show is either positive or negative. It must be noted that features are, in this case, related to both content –*i.e.* genre – and contexts –*i.e.* time of the day. The posteriori probabilities for a new show are then computed from these.

Breese *et al.* [12] implement a Bayesian Network where each node corresponds to each item. The states correspond to each possible vote value. In the network, each item will have a set of parent items that are its best predictors. The conditional probability tables are represented by decision trees. The authors report better results for this model than for several nearest-neighbors implementations over several datasets.

Hierarchical Bayesian Networks have also been used in several settings as a way to add domain-knowledge for information filtering [71]. One of the issues with hierarchical Bayesian networks, however, is that it is very expensive to learn and update the model when there are many users in it. Zhang and Koren [72] propose a variation over the standard Expectation-Maximization (EM) model in order to speed up this process in the scenario of a content-based recommender system.

3.5 Artificial Neural Networks

The Artificial Neural Network (ANN) model [74] is an assembly of inter-connected nodes and weighted links that is inspired in the architecture of the biological brain. Nodes in an ANN are called *neurons* as an analogy with biological neurons. These simple functional units are composed into networks that have the ability to learn a classification problem after they are trained with sufficient data.

The simplest case of an ANN is the *perceptron* model, illustrated in figure 7.

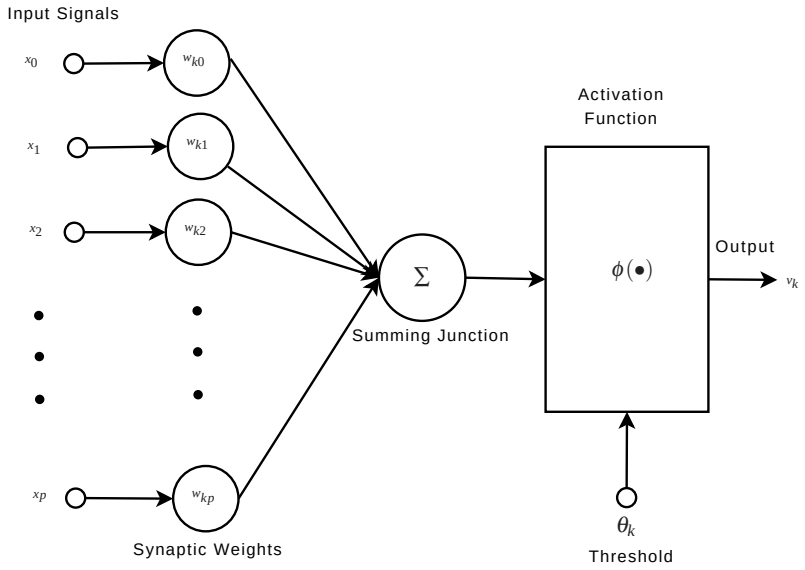


Fig. 7: Perceptron model

If we particularize the *activation function* ϕ to be the simple Threshold Function , the output is obtained by summing up each of its input value according to the weights of its links and comparing its output against some threshold θ_k. The output function can be expressed using Eq. 15. The perceptron model is a linear classifier that has a simple and efficient learning algorithm summarized in Listing 3.

$$y_k = \begin{cases} 1, & \text{if } \sum x_i w_{ki} \geq \theta_k \\ 0, & \text{if } \sum x_i w_{ki} < \theta_k \end{cases} \tag{15}$$

Besides the simple Threshold Function used in the Perceptron model, there are several other common choices for the activation function such as sigmoid, tanh, or step functions.

Using neurons as atomic functional units, there are many possible architectures to put them together in a network. But, by far, the most common approach is to use

Algorithm 3 Perceptron Learning algorithm

1: Let $D = (x_i, y_i)|i = 1, 2, \cdots, N$ be the set of training examples
2: Initialize the weight vector with random values, w^0
3: **repeat**
4: **for** each training example $(x_i, y_i) \in D$ do **do**
5: Compute the predicted output $\hat{y}_i^k$
6: **for** each weight w_j **do**
7: Update the weight $w_j^{k+1} = w_j^k + \lambda(y_i - \hat{y}_i^k)x_{ij}$
8: **end for**
9: **end for**
10: **until** stopping condition is met

the *feed-forward ANN* (see figure 8). In this case, signals are strictly progated in one way: from input to output.

An ANN can have any number of layers. The simple feedforward network in figure 8 has three layers. On the other hand, the perceptron in figure 7 is a single-layer feed-forward ANN. Layers in an ANN are classified into three types: input, hidden, and output. Units in the input layer respond to data that is fed into the network. Hidden units receive the weighted output from the input units. And the output units respond to the weighted output from the hidden units and generate the final output of the network.

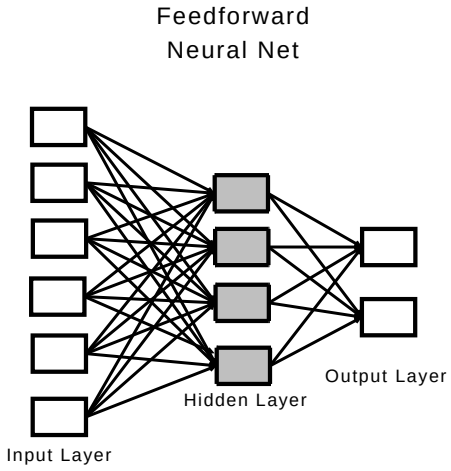


Fig. 8: Example of a simple feed-forward ANN with one hidden layer

The learning algorithm in listing 3 is only valid for the simple Perceptron model. There exist *supervised*, *unsupervised*, and *reinforcement* learning algorithms for the general case of multilayer networks. The generic algorithm for learning ANN in a supervised way, for instance, is summarized in figure 9. The most common concrete

algorithm for learning ANN's is the so-called *back-propagation* algorithm, based on the computation of the error derivative of the weights. However, it is beyond the scope of this chapter to go into the details of this algorithm.

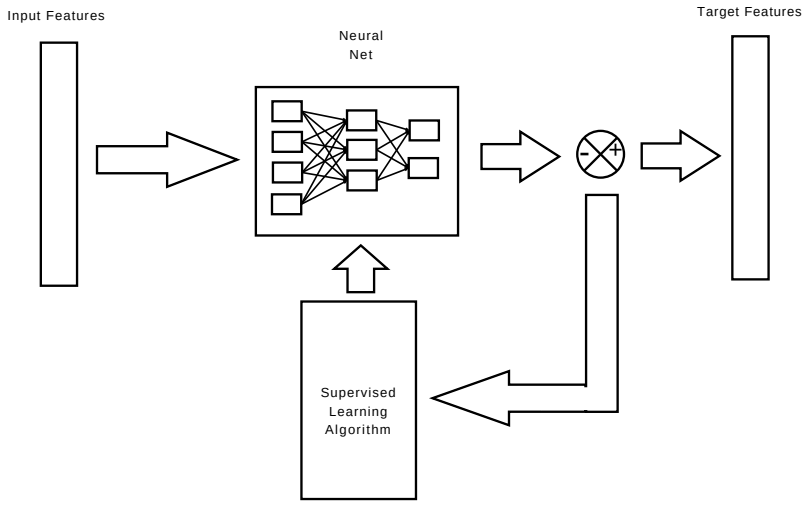


Fig. 9: Supervised learning process for learning an ANN

The main advantages of ANN are that – depending on the activation function – they can perform non-linear classification tasks, and that, due to their parallel nature, they can be efficient and even operate if part of the network fails. The main disadvantage is that it is hard to come up with the ideal network topology for a given problem and once the topology is decided this will act as a lower bound for the classification error. ANN's belong to the class of *sub-symbolic* classifiers, which means that they provide no semantics for inferring knowledge – *i.e.* they promote a kind of *black-box* approach.

3.5.1 Artificial Neural Networks in Recommender Systems

ANN's can be used in a similar way as Bayesian Networks to construct model-based recommender systems. However, there is no conclusive study to whether ANN introduce any performance gain. As a matter of fact, Pazzani and Billsus [51] did a comprehensive experimental study on the use of several machine learning algorithms for web site recommendation. Their main goal was to compare the simple naive Bayesian Classifier with computationally more expensive alternatives such as Decision Trees and Neural Networks. Their experimental results show that Decision Trees perform significantly worse. On the other hand ANN and the Bayesian

classifier perform similarly. They conclude that there does not seem to be a need for nonlinear classifiers such as the ANN.

ANN can be used to combine (or hybridize) the input from several recommendation modules or data sources. Hsu *et al.* [27], for instance, build a TV recommender by importing data from four different sources: user profiles and stereotypes; viewing communities; program metadata; and viewing context. They use the backpropagation algorithm to train a three-layered neural network.

Berka *et al.* [28] used ANN to build an URL recommender system for web navigation. They implemented a content-independent system based exclusively on *trails* – *i.e.* associating pairs of domain names with the number of people who traversed them. In order to do so they used feed-forward Multilayer Perceptrons trained with the Backpropagation algorithm.

Christakou and Stafylopatis [15] build a hybrid content-based collaborative filtering recommender system. The content-based recommender is implemented using three neural networks per user, each of them corresponding to one of the following features: "kinds", "stars", and "synopsis". They trained the ANN using the Resilient Backpropagation method.

3.6 Support Vector Machines

The goal of a Support Vector Machine (SVM) classifier [20] is to find a linear hyperplane (decision boundary) that separates the data in such a way that the margin is maximized. For instance, if we look at a two class separation problem in two dimensions like the one illustrated in figure 10, we can easily observe that there are many possible boundary lines to separate the two classes. Each boundary has an associated margin. The rationale behind SVM's is that if we choose the one that maximizes the margin we are less likely to missclassify unknown items in the future.

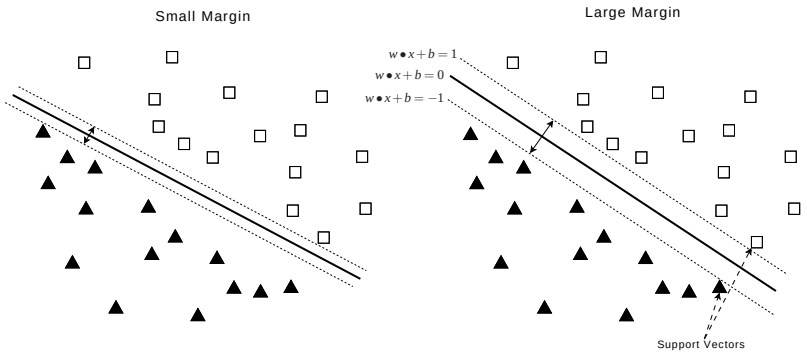


Fig. 10: Different boundary decisions are possible to separate two classes in two dimensions. Each boundary has an associated margin.

A linear separation between two classes is accomplished through the following function:

$$w \bullet x + b = 0 \tag{16}$$

We define a function that can classify items of being of class +1 or -1 as long as they are separated by some minimum distance from the class separation function previously defined. The function is given by Eq. 17

$$f(x) = \begin{cases} 1, & \text{if } w \bullet x + b \geq 1 \\ -1, & \text{if } w \bullet x + b \leq -1 \end{cases} \tag{17}$$

$$Margin = \frac{2}{\|w\|^2} \tag{18}$$

Following the main rationale for SVM's, we would like to maximize the margin between the two classes, given by equation 18. This is in fact equivalent to minimizing the inverse value $L(w) = \frac{\|w\|^2}{2}$ but subjected to the constraints given by $f(x)$. This is a constrained optimization problem and there are numerical approaches to solve it (e.g., quadratic programming).

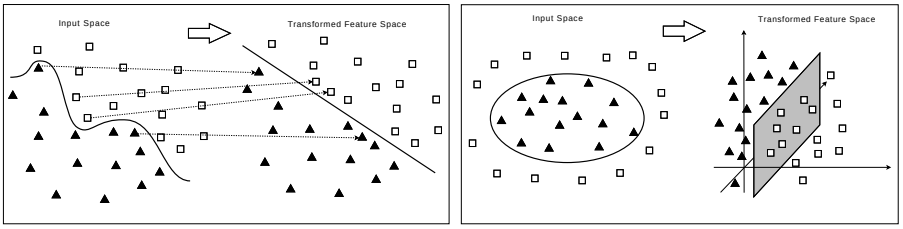
If the items are not linearly separable we can decide to turn the svm into a *soft margin* classifier by introducing a *slack variable*. In this case the formula to minimize is given by equation 19 subject to the new definition of $f(x)$ in equation 20. Note that the constant C in equation 19 allows to define the cost of a constraint violation.

$$L(w) = \frac{\|w\|^2}{2} + C\sum_{i=1}^{N} \varepsilon \tag{19}$$

$$f(x) = \begin{cases} 1, & \text{if } w \bullet x + b \geq 1 - \varepsilon \\ -1, & \text{if } w \bullet x + b \leq -1 + \varepsilon \end{cases} \tag{20}$$

On the other hand, if the decision boundary is not linear we need to transform data into a higher dimensional space (see figure 11). This is accomplished thanks to a mathematical transformation known as the *kernel trick*. The basic idea is to replace the dot products in equation 17 by a *kernel* function. There are many different possible choices for the kernel function such as Polynomial or Sigmoid. But by far the most common kernel function are the family of Radial Basis Function (RBF). The formulation for a Gaussian RBF, for instance, is given by Eq. 21.

$$k(x,x') = exp(-\gamma||x - x'||^2) \tag{21}$$



(a) (b)

Fig. 11: Mapping input data into a different feature space where problem will be linearly separable

3.6.1 Support Vector Machines in Recommender Systems

Support Vector Machines have recently gained popularity for their performance and efficiency in many settings. SVM's have also shown promising recent results in recommender systems.

Kang and Yoo [42], for instance, report on an experimental study that aims at selecting the best preprocessing technique for predicting missing values for an SVM-based recommender system. In particular, they use SVD and Support Vector Regression. The Support Vector Machine recommender system is built by first binarizing the 80 levels of available user preference data. They experiment with several settings and report best results for a threshold of 32 – *i.e.* a value of 32 and less is classified as *prefer* and a higher value as *do not prefer*. The user id is used as the class label and the positive and negative values are expressed as preference values 1 and 2.

Xu and Araki [69] used SVM to build a TV program recommender system. They use information from the Electronic Program Guide (EPG) as features. But in order to reduce features they removed words with lowest frequencies. Furthermore, and in order to evaluate different approaches, they used both the Boolean and the *Term frequency - inverse document frequency* (TFIDF) weighting schemes for features. In the former, 0 and 1 are used to represent absence or presence of a term on the content. In the latter, this is turned into the TFIDF numerical value.

Xia *et al.* [68] present different approaches to using SVM's for recommender systems in a collaborative filtering setting. They explore the use of Smoothing Support Vector Machines (SSVM). They also introduce a SSVM-based heuristic (SSVMBH) to iteratively estimate missing elements in the user-item matrix. They compute predictions by creating a classifier for each user. Their experimental results report best results for the SSVMBH as compared to both SSVM's and traditional user-based and item-based collaborative filtering.

Oku *et al.* [24] propose the use of Context-Aware Vector Machines (C-SVM) for context-aware recommender systems. They compare the use of standard SVM, C-SVM and an extension that uses collaborative filtering as well as C-SVM. Their

results show the effectiveness of the context-aware methods for restaurant recommendations.

3.7 Ensembles of Classifiers

The basic idea behind the use of *ensembles* of classifiers is to construct a set of classifiers from the training data and predict class label of previously unseen records by aggregating their predictions.

Ensembles of classifiers work whenever we can assume that the classifiers are independent. In this case we can ensure that the ensemble will produce results that are in the worst case as bad as the worst classifier in the ensemble. Therefore, combining classifiers of a similar classification error will only improve results.

In order to generate ensembles, several approaches are possible. The two most common techniques are *Bagging* and *Boosting*. In Bagging, we perform sampling with replacement, building the classifier on each bootstrap sample. Therefore each sample has probability $(11/n)n$ of being selected. In Boosting we use an iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records. Initially, all records are assigned equal weights. But, unlike bagging, weights may change at the end of each boosting round: Records that are wrongly classified will have their weights increased while records that are classified correctly will have their weights decreased. An example of boosting is the AdaBoost algorithm.

3.7.1 Ensembles of Classifiers in Recommender Systems

The use of ensembles of classifiers is common practice in the recommender systems field. As a matter of fact, any *hybridation* technique [13] can be considered an ensemble as it combines in one way or another several classifiers.

Experimental results show that ensembles can produce better results than any classifier in isolation. Bell *et al.* [8], for instance, used a combination of 107 different methods in their progress prize winning solution to the Netflix challenge. They state that their findings show that it pays off more to find substantially different approaches rather than focusing on refining a particular technique. This is related to the property we highlighted before: if classifiers are uncorrelated, their combination can only improve results. In order to blend the results from the ensembles they use a linear regression approach. In order to derive weights for each classifier, they partition the test dataset into 15 different bins and derive unique coefficients for each of the bins.

3.8 Evaluating Classifiers

Learning algorithms and classifiers can be evaluated by multiple criteria. This includes how accurately they perform the classification, their computational complexity during training , complexity during classification, their sensitivity to noisy data, their scalability, and so on. In this section we will focus only on classification performance. In order to evaluate a model we usually take into account the following measures: **True Positives** (TP): number of instances classified as belonging to class A that truly belong to class A; **True Negatives** (TN): number of instances classified as not belonging to class A and that in fact do not belong to class A; **False Positives** (FP): number of instances classified as class A but that do not belong to class A; **False Negatives** (FN): instances not classified as belonging to class v but that in fact do belong to class A.

The most commonly used measure for model performance is its *Accuracy* defined as the ratio between the instances that have been correctly classified (as belonging or not to the given class) and the total number of instances.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) \tag{22}$$

However, accuracy might be misleading in many cases. Imagine a 2-class problem in which there are 99,900 samples of class A and 100 of class B. If a classifier simply predicts everything to be of class A, the computed accuracy would be of 99.9%. However, the model performance is questionable because it will never detect any class B examples.

One way to improve this evaluation is to define the cost matrix where we declare the cost of misclassifying class B examples as being of class A. In real world applications different types of errors may indeed have very different costs. For example, if the 100 samples above correspond to defective airplane parts in an assembly line, incorrectly rejecting a non-defective part (one of the 99,900 samples) has a negligible cost compared to the cost of mistakenly classifying a defective part as a good part.

Other common measures of model performance, particularly in Information Retrieval, are Precision and Recall.

$$P = TP/(TP + FP) \tag{23}$$

$$R = TP/(TP + FN) \tag{24}$$

Precision is a measure of how many errors we make in classifying samples as being of class A. Recall measures how good we are in not leaving out samples that should have been classified as belonging to the class. Note that these two measures are misleading when used in isolation in most cases. We could build a classifier of perfect precision by not classifying any sample as being of class A (therefore obtaining 0 TP but also 0 FP). Conversely, we could build a classifier of perfect recall by classifying all samples as belonging to class A.

As a matter of fact, there is a measure, called the F_1-measure that combines both Precision and Recall into a single measure as:

$$F_1 = \frac{2RP}{R+P} = \frac{2TP}{2TP+FN+FP} \tag{25}$$

Additional factors that impact performance include the class distribution and the size of the training and test sets. In order to address changes due to training sampling size, we can construct the so-called Learning Curve, which shows how accuracy changes with varying training sample size. This requires to decide on a sampling strategy in order to create the curve. These sampling strategies were reviewed in section 2.2

Sometimes we would like to compare several competing models rather than estimate their performance independently. In order to do so we use a technique developed in the 1950s for analysis of noisy signals: the Receiver Operating Characteristic (ROC) Curve. A ROC curve characterizes the relation between positive hits and false alarms. The performance of each classifier is represented as a point on the curve (see Fig. 12).

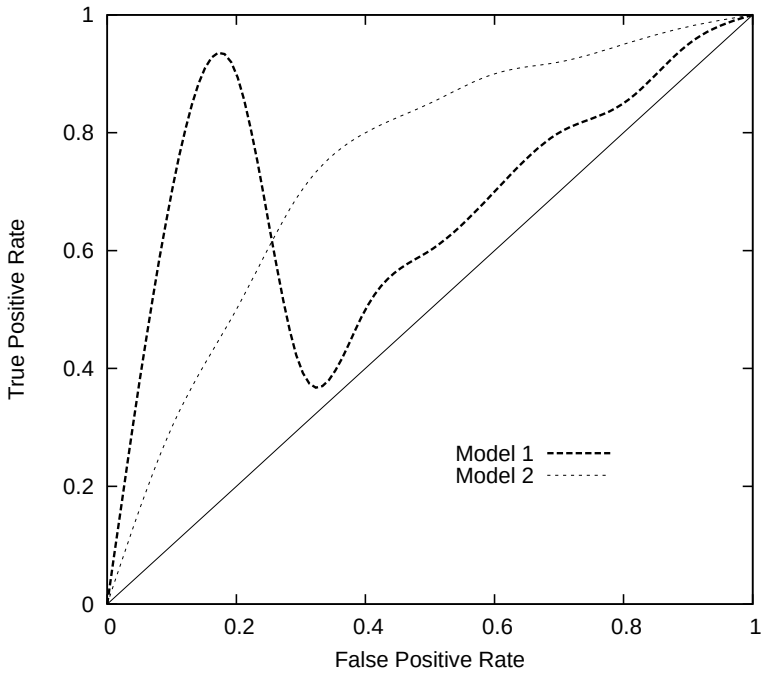


Fig. 12: Example of ROC Curve. Model 1 performs better for low False Positive Rates while Model 2 is fairly consistent throughout and outperforms Model 1 for False Positive Rates higher than 0.25

We plot the relation between TP and FP on a two-dimensional plot. Any point located at $x > y$ is classified as positive. Some points of interest in a ROC curve are $(TP,FP) = (0,0)$: declare everything to be negative class; $(1,1)$: declare everything to be positive class; $(1,0)$: ideal. The diagonal line means random guessing and below the diagonal line the prediction is opposite of the true class.

3.8.1 Evaluation of Classifiers in Recommender Systems

The most commonly accepted evaluation measure for recommender systems is the Mean Average Error (MAE) or Root Mean Squared Error (RMSE) of the predicted interest (or rating) and the measured one. These measures measure accuracy without any assumption on the purpose of the recommender system. However, as McNee et al. point out [47], there is much more than accuracy to deciding whether an item should be recommended. Herlocker et al. [38] provide a comprehensive review of algorithmic evaluation approaches to recommender systems. They suggest that some measures could potentially be more appropriate for some tasks. However, they are not able to validate the measures when evaluating the different approaches empirically on a class of recommendation algorithms and a single set of data.

A step forward is to consider that the purpose of a "real" recommender system is to produce a top-N list of recommendations and evaluate recommender systems depending on how well they can classify items as being *recommendable*. Ziegler *et al.* show [73] that evaluating recommender algorithms through top-N lists measures still does not map directly to the user's utility function. However, it does address some of the limitations of the more commonly accepted accuracy measures, such as MAE.

If we look at our recommendation as a classification problem, we can make use of well-known measures for classifier evaluation such as precision and recall. Basu *et al.* [6], for instance, use these measures by analyzing which of the items predicted in the top quartile of the rating scale were actually evaluated in the top quartile by the user.

McLaughlin and Herlocker [46] propose a *modified precision* measure in which non-rated items are counted as *not recommendable*. This precision measure in fact represents a lower-bound of the "real" precision.

Although the F-measure can be directly derived from the precision-recall values, it is not common to find it in recommender systems evaluations. Huang *et al.* [39] and Bozzon *et al.* [10], and Miyahara and Pazzani [48] are some of the few examples of the use of this measure.

ROC curves have also been used in evaluating recommender systems. Zhang *et al.* [58] use the value of the area under the ROC curve as their evaluation measure when comparing the performance of different algorithms under attack. Banerjee and Ramanathan [4] also use the ROC curves to compare the performance of different models.

It must be noted, though, that the choice of a good evaluation measure, even in the case of a top-N recommender system, is still a matter of discussion. Many

authors have proposed measures that are only indirectly related to these traditional evaluation schemes. Deshpande and Karypis [21], for instance, propose the use of the *hit rate* and the *average reciprocal hit-rank*. On the other hand, Breese *et al.* [12] define a measure of the utility of the recommendation in a ranked list as a function of the neutral vote.

4 Cluster Analysis

Clustering [37], also referred to as unsupervised learning, consists of assigning items to groups so that the items in the same groups are more similar than items in different groups: the goal is to discover natural (or meaningful) groups that exist in the data. Similarity is determined using a distance measure, such as the ones reviewed in 2.1, between the feature vectors that represent the items. The goal of a clustering algorithm is to minimize intra-cluster distances while maximizing inter-cluster distances because these constitute measures of the quality of a particular clustering. Intuitively this means that a good clustering of a set of data points shows clearly distinct groups. Note that in clustering there is no prior knowledge of class labels (as was the case in supervised learning). For example, a person who has an mp3 song collection may not have the songs in individual folders, but a good clustering algorithm might discover that the collection has mainly 3 groups (which could correspond, for example, to classical, heavy metal, and folk).

There are two main categories of clustering algorithms: hierarchical and partitional.

- Partitional clustering algorithms divide data items into non-overlapping clusters such that each data item is in exactly one cluster.
- Hierarchical clustering algorithms successively cluster items within found clusters, producing a set of nested cluster organized as a hierarchical tree.

Features used to represent an item play a crucial role in determining the clustering, as does the similarity metric used. In addition, items can be assigned membership values within a group (in fuzzy clustering an items membership in a group is assigned a value, commonly between 0 and 1), and it can be established that items have to belong to only one group or may belong to several.

Many clustering algorithms try to minimize a function that measures the quality of the clustering. Such a quality function is often referred to as the objective function, so clustering can be viewed as an optimization problem: the ideal clustering algorithm would consider all possible partitions of the data and output the partitioning that minimizes the quality function. But the corresponding optimization problem is NP hard, so many algorithms resort to heuristics (e.g., in the k-means algorithm using only local optimization procedures potentially ending in local minima). The main point is that clustering is a difficult problem for which finding optimal solutions is often not possible.

Selection of the particular clustering algorithm and its parameters (*e.g.*, similarity measure) depend on many factors, including the characteristics of the data. In the following sections we describe the *k*-means clustering algorithm and some of its alternatives.

4.1 k-*Means*

k-Means (also known as *k*-centers) clustering is a partitioning method. The function partitions the data set of N items into k disjoint subsets S_j that contain N_j items so that they are as close to each other as possible according a given distance measure. Each cluster in the partition is defined by its members N_j and by its centroid λ_j. The centroid for each cluster is the point to which the sum of distances from all items in that cluster is minimized. Thus, we can define the *k*-means algorithm as an iterative process to minimize

$$E = \sum_{1}^{k} \sum_{n \in S_j} d(x_n, \lambda_j) \tag{26}$$

where x_n is a vector representing the n-th item, λ_j is the centroid of the item in S_j and d is the distance measure. The *k*-means algorithm moves items between clusters until E cannot be decreased further. The algorithm is described in Listing 4.

Algorithm 4 *k*-means

Input
$X = x_1, ..., x_n$ (items to be clustered)
k (number of clusters)
Output
$\Lambda = \lambda_1, .., \lambda_k$ (cluster centroids)
$m : X \rightarrow C$ (cluster membership)

Set the centroids Λ to their initial value (e.g. random selection of k items in X)

for $x_i \in X$ **do**
    $m(x_i) = argmin_{j \in \{1..k\}} d(x_i, \lambda_j)$ (assign each item to the closest centroid)
end for

while m has changed **do**
    for $j \in \{1..k\}$ **do**
        Recalculate the centroid λ_j according to the items that belong to it $\{i | m(i) = j\}$
    end for
    for $x_i \in X$ **do**
        $m(x_i) = argmin_{j \in \{1..k\}} d(x_i, \lambda_j)$ (update the membership of x_i to the closest centroid)
    end for
end while

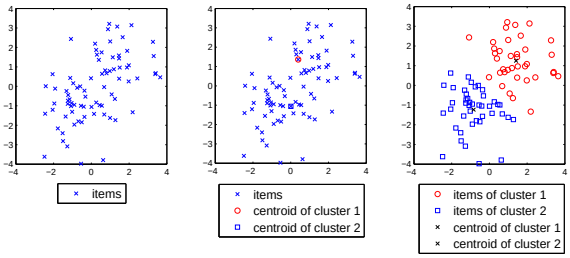


Fig. 13: Example of *k*-means, with k=2. From left to right, the data to be partitioned, the initial 2 centroids as two data items chosen at random, and finally, the final partition of the data in 2 clusters (with the centroid of the clusters as $\times$)

The algorithm works by randomly selecting k centroids. Then all items are assigned to the cluster whose centroid is the closest to them. The new cluster centroid needs to be updated to account for the items who have been added or removed from the cluster and the membership of the items to the cluster updated. This operation continues until there are no further items that change their cluster membership. Most of the convergence to the final partition takes place during the first iterations of the algorithm, and therefore, the stopping condition is often changed to "until relatively few points change clusters" in order to improve efficiency.

The distance measure depends on the data we need to cluster. In the example in figure 13 we used the sum of square errors (SSE) ($d(x_i,\lambda_j) = \sum_k (x_{ik} - \lambda_{jk})^2$). However, other measures such as the Euclidian distance, Cosine similarity, Pearson correlation, Manhattan distance or Hamming distance are also widely used.

The basic *k*-means has several shortcomings: **(1)** it does assume prior knowledge of the data in order to choose the appropriate k. If the number of clusters is unknown it is advisable to run the algorithm for a range of k and then choose the partition whose in which the distance between the items of one cluster with respect to the items of the rest of the clusters is high. **(2)** The final clusters are very sensitive to the selection of the initial centroids. Thus, different runs of the algorithm could yield different clusters. To address this issue it has been proposed to run m replicas with different seeds – the initial centroid – and to return the one with the lowest value of E. More sophisticated techniques involve data sampling and using hierarchical clustering to determine the initial centroids. Another technique is to use the Bisecting *k*-means, which is a a variant of *k*-means that can produce a hierarchical clustering. The last shortcoming **(3)** is that the basic *k*-means can produce empty clusters, there are several strategies to deal with this issue, from treating it as an error to treat it as a singleton, creating a new cluster consisting of the one point furthest from its centroid.

k-means is an extremely simple and efficient algorithm but, apart from the issued just described it does have several limitations with regard to the data. *k*-means

has problems when clusters are of differing sizes, densities, non-globular shapes. *k*-means also has problems when the data contains outliers.

4.2 Alternatives to k*-means*

Although the *k*-means algorithm presents limitations, the truth is that it is very difficult to find practical alternatives. In the following paragraphs we will briefly review some of them.

Density-based clustering algorithms such as DBSCAN work by building up on the definition of density as the number of points within a specified radius. DBSCAN, for instance, defines three kinds of points: *core points* are those that have more than a specified number of neighbors within a given distance; *border points* have fewer than the specified number but belong to a *core point* neighborhood; and *noise points* are those that are neither core or border. The algorithm iteratively removes *noise points* and performs clustering on the remaining points.

Message-passing clustering algorithms are a very recent family of graph-based clustering methods. Instead of considering an initial subset of the points as centers and then iteratively adapt those, message-passing algorithms initially consider all points as centers – usually known as *exemplars* in this context. During the algorithm execution points, which are now considered nodes in a network, exchange messages until clusters gradually emerge. *Affinity Propagation* is an important representative of this family of algorithms [29] that works by defining two kinds of messages between nodes: "responsibility", which reflects how well-suited receiving point is to serve as exemplar of the point sending the message, taking into account other potential exemplars; and "availability", which is sent from candidate exemplar to the point and reflects how appropriate it would be for the point to choose the candidate as its exemplar, taking into account support from other points that are choosing that same exemplar. Affinity propagation has been applied, with very good results, to problems as different as DNA sequence clustering, face clustering in images, or text summarization.

Finally, **Hierarchical Clustering**, produces a set of nested clusters organized as a hierarchical tree (*dendogram*). Hierarchical Clustering does not have to assume a particular number of clusters in advanced. Also, any desired number of clusters can be obtained by selecting the tree at the proper level. Hierarchical clusters can also sometimes correspond to meaningful taxonomies. Traditional hierarchical algorithms use a similarity or distance matrix and merge or split one cluster at a time. There are two main approaches to hierarchical clustering. In *agglomerative* hierarchical clustering we start with the points as individual clusters and at each step, merge the closest pair of clusters until only one cluster (or k clusters) are left. In *divisive* hierarchical clustering we start with one, all-inclusive cluster, and at each step, split a cluster until each cluster contains a point (or there are k clusters).

4.3 Cluster Analysis in Recommender Systems

The main problem for scaling a collaborative filtering classifier is the amount of operations involved in computing distances – for finding the best *k*-nearest neighbors, for instance. A possible solution is, as we saw in section 2.3, to reduce dimensionality. But, even if we reduce dimensionality of features, we might still have many objects to compute the distance to. This is where clustering algorithms can come into play. The same is true for content-based recommender systems, where distances among objects are needed to retrieve similar ones.

Clustering is sure to improve efficiency because the number of operations is reduced. However, and unlike dimensionality reduction methods, it is unlikely that it can help improve accuracy. Therefore, clustering must be applied with care when designing a recommender system, measuring the compromise between improved efficiency and a possible decrease in accuracy.

We shall now review some known applications of clustering techniques in recommender systems.

Xue *et al.* [70] present a typical use of clustering in the context of a recommender systems by employing the k-*means* algorithm as a pre-processing step to help in neighborhood formation. They do not restrict the neighborhood to the cluster the user belongs to but rather use the distance from the user to different cluster centroids as a pre-selection step for the neighbors. They also implement a cluster-based smoothing technique in which missing values for users in a cluster are replaced by cluster representatives. Their method is reported to perform slightly better than standard *k*NN-based collaborative filtering.

In a similar way, Sarwar *et al.* [23] describe an approach to implement a scalable *k*NN classifier. They partition the user space by applying the *bisecting* k-*means* algorithm and then use those clusters as the base for neighborhood formation. They report a decrease in accuracy of around 5% as compared to standard *k*NN CF. However, their approach allows for a significant improvement in efficiency.

Connor and Herlocker [18] present a different approach in which, instead of users, they cluster items. Using the Pearson Correlation similarity measure they try out four different algorithms: average link hierarchichal agglomerative [35], robust clustering algorithm for categorical attributes (ROCK) [36], kMetis, and hMetis [4]. Although clustering did improve efficiency, all of their clustering techniques yielded worse accuracy and coverage than the non-partitioned baseline.

Li *et al.* [54] and Ungar and Foster [63] present a very similar approach for using k-*means* clustering for solving a probabilistic model interpretation of the recommender problem.

To the best of our knowledge, alternatives to *k*-means such as the ones presented in section 4.2 have not been applied to recommender systems. The simplicity and efficiency of the *k*-means algorithm shadows possible alternatives. It is not clear whether density-based or hierarchical clustering approaches have anything to offer in the recommender systems arena. On the other hand, message-passing algorithms

[4] http://www.cs.umn.edu/ karypis/metis

have been shown to be more efficient and their graph-based paradigm can be easily translated to the recommender systems problem. It is possible that we see applications of these algorithms in the coming years.

5 Association Analysis

Association Rule Mining focuses on finding rules that will predict the occurrence of an item based on the occurrences of other items in a transaction. The fact that two items are found to be related means co-occurrence but not causality.

We define an *itemset* as a collection of one or more items (e.g. (Milk, Beer, Diaper)). A *k-itemset* is an itemset that contains k items. The frequency of a given itemset is known as *support count* (e.g. (Milk, Beer, Diaper) = 131). And the *support* of the itemset is the fraction of transactions that contain it (e.g. (Milk, Beer, Diaper) = 0.12). A *frequent itemset* is an itemset with a support that is greater or equal to a *minsup* threshold.

An association rule is an expression of the form $X \Rightarrow Y$, where X and Y are itemsets. (e.g. $Milk, Diaper \Rightarrow Beer$). In this case the *support* of the association rule is the fraction of transactions that have both X and Y. On the other hand, the *confidence* of the rule is how often items in Y appear in transactions that contain X.

Given a set of transactions T, the goal of association rule mining is to find all rules having $support \geq minsupthreshold$ and $confidence \geq minconfthreshold$. The brute-force approach would be to list all possible association rules, compute the support and confidence for each rule and then prune rules that do not satisfy both conditions. This is, however, computationally very expensive.

For this reason, we take a two-step approach: (1) Generate all itemsets whose support $\geq$ minsup (**Frequent Itemset Generation**); (2) Generate high confidence rules from each frequent itemset (**Rule Generation**)

5.1 Frequent Itemset generation and the Apriori Principle

But if we follow a brute-force approach, frequent itemset generation is still computationally expensive. Each itemset in the lattice is a candidate frequent itemset and we have to count the support of each candidate by scanning the transaction database (i.e. match each transaction against every candidate).

Several techniques exist to optimize the generation of frequent itemsets. On a broad sense they can be classified into those that try to minimize the number of candidates (M), those that reduce the number of transactions (N), and those that reduce the number of comparisons (NM).

The most common approach though, is to reduce the number of candidates using the *Apriori principle*. This principle states that if an itemset is frequent, then all of its subsets must also be frequent. This is verified using the support measure because

the support of an itemset never exceeds that of its subsets. The Apriori Algorithm is a practical implementation of the principle. Its basic steps are illustrated in listing 5.

Algorithm 5 Apriori algorithm

```
1: Let k=1
2: Generate frequent itemsets of length 1
3: repeat
4:     Generate length (k+1) candidate itemsets from length k frequent itemsets
5:     Prune candidate itemsets containing subsets of length k that are infrequent
6:     Count the support of each candidate by scanning the DB
7:     Eliminate candidates that are infrequent, leaving only those that are frequent
8: until no new frequent itemsets are identified
```

Several implementation strategies are also possible to reduce the number of comparisons. However, no matter what strategies we adopt, we need to be aware of the factors that affect computational complexity. First, there is the minimum support threshold. A lower threshold may produce more and longer frequent itemsets. Another factor that affects is the number of items of the data set (dimensionality). More items means more space to store support count. The size of the transaction database might also affect algorithms such as the Apriori, which requires multiple passes.

5.2 Rule Generation

Given a frequent itemset L, the goal when generating rules is to find all non-empty subsets that satisfy the minimum confidence requirement.

If $|L| = k$, then there are $2k2$ candidate association rules. So, as in the frequent itemset generation, we need to find ways to generate rules efficiently.

For the Apriori Algorithm we can generate candidate rules by merging two rules that share the same prefix in the rule consequent. E.g. $join(CD \Rightarrow AB, BD \Rightarrow AC)$ would produce the candidate rule $D \Rightarrow ABC$. We could therefore prune rule $D \Rightarrow ABC$ provided that its subset $AD \Rightarrow BC$ does not have high confidence.

5.3 Association Rules in Recommender Systems

The effectiveness of association rule mining for uncovering patterns and driving personalized marketing decisions has been known for a some time [2]. However, and although there is a clear relation between this method and the goal of a recommender system, they have not become mainstream. The main reason is that this approach is similar to item-based collaborative filtering but is less flexible since it requires of an explicit notion of *transaction – e.g.* co-ocurrence of events in a given session. In the

next paragraphs we present some promising examples, some of which indicate that association rules still have not had their last word.

Mobasher *et al.* [49] present a system for web personalization based on association rules mining. Their system identifies association rules from pageviews co-occurrences based on users navigational patterns. Their approach outperforms a *k*NN-based recommendation system both in terms of precision and coverage.

Smyth *et al.* [61] present two different case studies of using association rules for recommender systems. In the first case they use the *a priori* algorithm to extract item association rules from user profiles in order to derive a better item-item similarity measure. In the second case, they apply association rule mining to a *conversational* recommender. The goal here is to find co-occurrent *critiques* – i.e. user indicating a preference over a particular feature of the recommended item.

Lin *et al.* [45] present a new association mining algorithm that adjusts the minimum support of the rules during mining in order to obtain an appropriate number of significant rule therefore addressing some of the shortcomings of previous algorithms such as the *a priori*. They mine both association rules between users and items. The measured accuracy outperforms previously reported values for correlation-based recommendation and is similar to the more elaborate approaches such as the combination of SVD and ANN.

As already mentioned in section 3.2.1, Cho *et al.* [14] combine Decision Trees and Association Rule Mining in a web shop recommender system. In their system, association rules are derived in order to link related items. The recommendation is then computed by intersecting association rules with user preferences. They look for association rules in different transaction sets such as purchases, basket placement, and click-through. They also use a heuristic for weighting rules coming from each of the transaction sets. Purchase association rules, for instance, are weighted higher than click-through association rules.

6 Conclusions

This chapter has introduced the main data mining methods and techniques that can be applied in the design of a recommender system. We have also surveyed their use in the literature and provided some rough guidelines on how and where they can be applied.

We started by reviewing techniques that can be applied in the pre-processing step. First, there is the choice of an appropriate distance measure, which is reviewed in Section 2.1. This is required by most of the methods in the following steps. The cosine similarity and Pearson correlation are commonly accepted as the best choice. Although there have been many efforts devoted to improving these distance measures, recent works seem to report that the choice of a distance function does not play such an important role. Then, in Section 2.2, we reviewed the basic sampling techniques that need to be applied in order to select a subset of an originaly large data set, or to separating a training and a testing set. Finally, we discussed the use of

dimensionality reduction techniques such as Principal Component Analysis and Singular Value Decomposition in Section 2.3. These techniques offer a dual advantage: On the one hand they reduce dimensionality and avoid the *curse of dimensionality* problem; on the other, they help reduce some of the noise in the original data set. We explained some success stories using dimensionality reduction tecniques, especially in the context of the Netflix prize.

In Section 3, we reviewed the main classification methods: namely, nearest-neighbors, decision trees, rule-based classifiers, bayesian networks, artificial neural networks, and support vector machines. We saw that, although *k*NN (see Section 3.1) collaborative filtering is the preferred approach, all those classifiers can be applied in different settings. Decision trees (see Section 3.2) can be used to derive a model based on the content of the items or to model a particular part of the system. Decision rules (see Section 3.3) can be derived from a pre-existing decision trees, or can also be used to introduce business or domain knowledge. Bayesian networks (see Section 3.4) are a popular approach to content-based recommendation, but can also be used to derive a model-based collaborative filtering system. In a similar way, artificial neural networks can be used to derive a model-based recommender. Finally, support vector machines (see Section 3.6) are gaining popularity also as a way to infer content-based classifications or derive a collaborative filtering model.

Choosing the right classifier for a recommender system is not easy and is in many senses task and data-dependent. In the case of collaborative filtering, some results seem to indicate that model-based approaches using classifiers such as the SVM or Bayesian Networks can slightly improve performance of the standard *k*NN classifier. However, those results are non-conclusive and hard to generalize. In the case of a content-based recommender system there is some evidence that in some cases Bayesian Networks will perform better than simpler methods such as decision trees. However, it is not clear that more complex non-linear classifiers such as the ANN or SVMs can perform better.

Therefore, the choice of the right classifier for a specific recommending task still has nowadays much of exploratory. A practical rule-of-thumb is to start with the simplest approach and only introduce complexity if the performance gain obtained justifies it. The performance gain should of course balance different dimensions such as prediction accuracy or computational efficiency.

We reviewed clustering algorithms in Section 4. Clustering is usually used in recommender systems to improve performance. A previous clustering step, either in the user of item space, reduces the number of distance computations we need to perform. However, this usually comes at the price of a lower accuracy so it should be handled with care. As a matter of fact, improving efficiency by using a dimensionality reduction technique such as SVD is probably a better choice in the general case. As opposed to what happens with classifiers, not so many clustering algorithms have been used in the context of recommender systems. The simpliciy and relative efficiency of the *k*-means algorithm (see Section 4.1) make it hard to find a practical alternative. We reviewed some of them such as Hierarchical Clustering or Message-passing algorithms in Section 4.2. Although these techniques have still

not been applied for recommender systems, they offer a promising avenue for future research.

Finally, in Section 5, we described association rules and surveyed their use in recommender systems. Association rules offer an intuititive framework for recommending items whenever there is an explicit or implicit notion of *transaction*. Although there exist efficient algorithms for computing association rules, and they have proved more accurate than standard *k*NN collaborative filtering, they are still not a favored approach.

The choice of the right data mining technique in designing a recommender system is a complex task that is bound by many problem-specific constraints. However, we hope that the short review of techniques and experiences included in this chapter can help the reader make a much more informed decision. Besides, we have also uncovered areas that are open to many further improvements, and where there is still much exciting and relevant research to be done in the coming years.

References

1. G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.
3. M. Anderson, M. Ball, H. Boley, S. Greene, N. Howse, D. Lemire, and S. McGrath. Racofi: A rule-applying collaborative filtering system. In *Proc. IEEE/WIC COLA'03*, 2003.
4. S. Banerjee and K. Ramanathan. Collaborative filtering on skewed datasets. In *Proc. of WWW '08*, 2008.
5. C. Basu, H. Hirsh, and W. Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 714–720. AAAI Press, 1998.
6. C. Basu, H. Hirsh, and W. Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *AAAI Workshop on Recommender Systems*, 1998.
7. R. Bell and Y. Koren. Improved neighborhood-based collaborative filtering. In *In proceedings of KDDCup '07*, 2007.
8. R. M. Bell, Y. Koren, and C. Volinsky. The bellkor solution to the netflix prize. Technical report, AT&T Labs Research, 2007.
9. A. Bouza, G. Reif, A. Bernstein, and H. Gall. Semtree: ontology-based decision tree algorithm for recommender systems. In *International Semantic Web Conference*, 2008.
10. A. Bozzon, G. Prandi, G. Valenzise, and M. Tagliasacchi. A music recommendation system based on semantic audio segments similarity. In *Proceeding of Internet and Multimedia Systems and Applications - 2008*, 2008.
11. M. Brand. Fast online svd revisions for lightweight recommender systems. In *SIAM International Conference on Data Mining (SDM)*, 2003.
12. J. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, page 4352, 1998.
13. R. Burke. Hybrid web recommender systems. pages 377–408. 2007.
14. Y. Cho, J. Kim, and S. Kim. A personalized recommender system based on web usage mining and decision tree induction. *Expert Systems with Applications*, (23), 2002.

15. C. Christakou and A. Stafylopatis. A hybrid movie recommender system based on neural networks. In *ISDA '05: Proceedings of the 5th International Conference on Intelligent Systems Design and Applications*, pages 500–505, 2005.
16. W. Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the 12th International Conference*, 1995.
17. R. Collobert and S. Bengio. Svmtorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 1:143–160, 2001.
18. M. Connor and J. Herlocker. Clustering items for collaborative filtering. In *SIGIR Workshop on Recommender Systems*, 2001.
19. T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.
20. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.
21. M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, 2004.
22. J. W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.
23. B. S. et al. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In *Proceedings of the Fifth International Conference on Computer and Information Technology*, 2002.
24. K. O. et al. Context-aware svm for context-dependent information recommendation. In *International Conference On Mobile Data Management*, 2006.
25. P. T. et al. *Introduction to Data Mining*. Addison Wesley, 2005.
26. S. G. et al. Tv content recommender system. In *AAAI/IAAI 2000*, 2000.
27. S. H. et al. Aimed- a personalized tv recommendation system. In *Interactive TV: a Shared Experience*, 2007.
28. T. B. et al. A trail based internet-domain recommender system using artificial neural networks. In *Proceedings of the Int. Conf. on Adaptive Hypermedia and Adaptive Web Based Systems*, 2002.
29. B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 307, 2007.
30. N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Mach. Learn.*, 29(2-3):131–163, 1997.
31. S. Funk. Netflix update: Try this at home, 2006.
32. R. Ghani and A. Fano. Building recommender systems using a knowledge base of product semantics. In *In 2nd International Conference on Adaptive Hypermedia and Adaptive Web Based Systems*, 2002.
33. K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Journal Information Retrieval*, 4(2):133–151, July 2001.
34. G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, April 1970.
35. E. Gose, R. Johnsonbaugh, and S. Jost. *Pattern Recognition and Image Analysis*. Prentice Hall, 1996.
36. S. Guha, R. Rastogi, and K. Shim. Rock: a robust clustering algorithm for categorical attributes. In *Proc. of the 15th Intl Conf. On Data Eng.*, 1999.
37. J. A. Hartigan. *Clustering Algorithms (Probability & Mathematical Statistics)*. John Wiley & Sons Inc.
38. J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, 2004.
39. Z. Huang, D. Zeng, and H. Chen. A link analysis approach to recommendation under sparse data. In *Proceedings of AMCIS 2004*, 2004.
40. A. Isaksson, M. Wallman, H. Göransson, and M. G. Gustafsson. Cross-validation and bootstrapping are unreliable in small sample classification. *Pattern Recognition Letters*, 29:1960–1965, 2008.

41. I. T. Jolliffe. *Principal Component Analysis*. Springer, 2002.
42. H. Kang and S. Yoo. Svm and collaborative filtering-based prediction of user preference for digital fashion recommendation systems. *IEICE Transactions on Inf & Syst*, 2007.
43. M. Kurucz, A. A. Benczur, and K. Csalogany. Methods for large scale svd with missing values. In *Proceedings of KDD Cup and Workshop 2007*, 2007.
44. N. Lathia, S. Hailes, and L. Capra. The effect of correlation coefficients on communities of recommenders. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2000–2005, New York, NY, USA, 2008. ACM.
45. W. Lin and S. Alvarez. Efficient adaptive-support association rule mining for recommender systems. *Data Mining and Knowledge Discovery Journal*, 6(1), 2004.
46. M. R. McLaughlin and J. L. Herlocker. A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *Proc. of SIGIR '04*, 2004.
47. S. M. McNee, J. Riedl, and J. A. Konstan. Being accurate is not enough: how accuracy metrics have hurt recommender systems. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 1097–1101, New York, NY, USA, 2006. ACM Press.
48. K. Miyahara and M. J. Pazzani. Collaborative filtering with the simple bayesian classifier. In *Pacific Rim International Conference on Artificial Intelligence*, 2000.
49. B. Mobasher, H. Dai, T. Luo, and M. Nakagawa. Effective personalization based on association rule discovery from web usage data. In *Workshop On Web Information And Data Management, WIDM '01*.
50. A. Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD Cup and Workshop 2007*, 2007.
51. M. J. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27(3):313–331, 1997.
52. V. Pronk, W. Verhaegh, A. Proidl, and M. Tiemann. Incorporating user control into recommender systems based on naive bayesian classification. In *RecSys '07: Proceedings of the 2007 ACM conference on Recommender systems*, pages 73–80, 2007.
53. D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann, second edition edition, 1999.
54. B. K. Q. Li. Clustering approach for hybrid recommender system. In *Web Intelligence 03*, 2003.
55. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, March 1986.
56. S. Rendle and L. Schmidt-Thieme. Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In *Recsys '08: Proceedings of the 2008 ACM conference on Recommender Systems*, 2008.
57. G. W. F. L. T. K. S. Deerwester, S. T. Dumais and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41, 1990.
58. J. F. S. Zhang, Y. Ouyang and F. Makedon. Analysis of a low-dimensional linear model under recommendation attacks. In *Proc. of SIGIR '06*, 2006.
59. B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Incremental svd-based algorithms for highly scalable recommender systems. In *5th International Conference on Computer and Information Technology (ICCIT)*, 2002.
60. B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl. Application of dimensionality reduction in recommender systemsa case study. In *ACM WebKDD Workshop*, 2000.
61. B. Smyth, K. McCarthy, J. Reilly, D. O'Sullivan, L. McGinty, and D. Wilson. Case studies in association rule mining for recommender systems. In *Proc. of International Conference on Artificial Intelligence (ICAI '05)*, 2005.
62. E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: A large-scale study in the orkut social network. In *Proceedings of the 2005 International Conference on Knowledge Discovery and Data Mining (KDD-05)*, 2005.
63. L. H. Ungar and D. P. Foster. Clustering methods for collaborative filtering. In *Proceedings of the Workshop on Recommendation Systems*, 2000.
64. A. R. M. W. L. Martinez. *Exploratory Data Analysis*. Chapman & Hall, 2004.
65. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition edition, 2005.

66. I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2nd edition edition, 2005.
67. M. Wu. Collaborative filtering via ensembles of matrix factorizations. In *Proceedings of KDD Cup and Workshop 2007*, 2007.
68. Z. Xia, Y. Dong, and G. Xing. Support vector machines for collaborative filtering. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 169–174, New York, NY, USA, 2006. ACM.
69. J. Xu and K. Araki. A svm-based personal recommendation system for tv programs. In *Multi-Media Modelling Conference Proceedings*.
70. G.-R. Xue, C. Lin, Q. Yang, W. Xi, H.-J. Zeng, Y. Yu, and Z. Chen. Scalable collaborative filtering using cluster-based smoothing. In *Proceedings of the 2005 SIGIR*, 2005.
71. K. Yu, V. Tresp, and S. Yu. A nonparametric hierarchical bayesian framework for information filtering. In *SIGIR '04*, 2004.
72. Y. Zhang and J. Koren. Efficient bayesian hierarchical user modeling for recommendation system. In *SIGIR 07*, 2007.
73. C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *Proc. of WWW '05*, 2005.
74. J. Zurada. *Introduction to artificial neural systems*. West Publishing Co., St. Paul, MN, USA, 1992.