Data Mining 2 23-24

Notes

University of Pisa

M.Sc. in Data Science and Business Informatics

Contents

1 Rule Based Models **3**

1.1 How Rule-Based Models Work 4

1.2 Properties of a Rule Set 4

1.3 Building a Rule Set 5

1.3.1 Direct Methods for Rule Extraction 6

1.3.2 Indirect Methods for Rule Extraction 8

1.4 Characteristics of Rule Based Models 9

2 Sequential Pattern Mining **10**

2.1 Time Constraints 13

2.2 Generalized Sequential Patterns Algorithm 14

2.2.1 Candidate generation 14

2.2.2 Candidate Pruning 15

2.2.3 Support Counting 16

2.3 Generalized Sequential Patterns and Time Constraints 16

3 Transactional Clustering **17**

3.1 K-Modes 18

3.2 ROCK (RObust Clustering using linK) 18

3.3 CLOPE (Clustering with sLOPE) 20

3.4 TX-Means 21

4 Time Series **23**

4.1 Similarity Between Time Series 24

4.1.1 Structural-based Similarities 24

4.1.2 Shape-based Similarities 24

5 Time Series: Clustering and Classification **32**

5.1 Clustering 32

5.2 Motif and Discord Discovery 33

5.2.1 Matrix Profile . . . 33
5.3 Classification . . . 35
5.3.1 Shapelet Extraction . . . 36

6 Imbalanced Learning **38**
6.1 Balancing the Training Set . . . 38
6.1.1 Undersampling . . . 38
6.1.2 Oversampling . . . 39
6.2 Balancing at the Algorithm Level . . . 40

7 Dimensionality Reduction **42**
7.1 Feature Selection . . . 42
7.2 Feature Projection . . . 43
7.2.1 Principal Component Analysis (PCA) . . . 43
7.2.2 Multi-Dimensional Scaling (MDS) . . . 44

Chapter 1

Rule Based Models

A rule based classifier is a model that uses a **rule set** of "if-then" rules to classify instances. Each rule is expressed in the form:

$$r_i : (Cond_i) \rightarrow y_i.$$

The left side contains a conjunction of attribute test conditions, and is called **antecedent** or **precondition**, while the right side represents the predicted class, and is called the **consequent**. Each condition is defined by a set of k attribute-value pairs, such that:

$$Cond_i = (A_1 \ op \ v_1) \wedge (A_2 \ op \ v_2) \wedge \cdots \wedge (A_k \ op \ v_k) \ ,$$

where op is a comparison operator. Each attribute test is also known as a **conjunct**.

A rule r **covers** an instance x if the attributes of the instance satisfy the antecedent of the rule. Consider the following dataset:

Name	Can Fly	Gives Birth	Blood Type
Bat	Y	Y	W
Owl	Y	N	W
Crocodile	N	N	C
Platypus	N	N	W

Table 1.1: Small example dataset.

The rule (Can Fly $= Y$) $\wedge$ (Gives Birth $= N$) $\rightarrow$ Bird covers the instance "Owl".

The **coverage** of a rule is the fraction of records in the whole dataset that are covered by it. The **accuracy** (sometimes called **precision**) of a rule is the fraction of records in the dataset that satisfy the antecedent that also satisfy the consequent.

Coverage and Accuracy

$$Coverage(r) = \frac{|A|}{|D|}$$

$$Accuracy(r) = \frac{|A \cap y|}{|A|}$$

1.1 How Rule-Based Models Work

A rule based classifier classifies a test instance based on the rule triggered by the instance. Looking at the dataset pictured in Table 1.1, assume we obtained the following rule set from a training set:

$$r_1 : (\text{Can Fly} = Y) \rightarrow \text{Bird}$$
$$r_2 : (\text{Gives Birth} = Y) \wedge (\text{Blood Type} = W) \rightarrow \text{Mammal}$$
$$r_3 : (\text{Blood Type} = C) \rightarrow \text{Reptile}$$

The instance Owl triggers the first rule, and is therefore classified as a Bird. The Bat triggers both the first and the second rule, which produce conflicting outcomes. None of the rules cover the example Platypus, so there's no immediate way to assign a class to this animal. The following section will explain how these issues can be solved.

1.2 Properties of a Rule Set

The rule set generated by the model can be characterized by the following two properties:

Mutually Exclusive Rule Set

The rules in a rule set R are mutually exclusive if no two rules in R are triggered by the same instance; this property guarantees that each instance is covered by at most one rule in R.

Exhaustive Rule Set

A rule set R is exhaustive if each combination of attribute values is covered by at least one rule.

Unfortunately, many rule based classifiers do not have such properties. If the rule set is not exhaustive, a default rule with an empty antecedent can be added to classify all instances that are not covered by any other rule.

If the rule set is not mutually exclusive, the rules can be organized into an **ordered rule set** (also known as **decision list**).

Ordered Rule Set

The rules in an ordered rule set R are ranked in decreasing order of priority.

The rank of the rule can be defined via either **rule-based ordering** (rules are ranked based on their quality, e.g., their accuracy) or **class-based ordering** (all rules that have the same consequent appear together). When a test instance is presented to the model, it is compared with the rules starting from the one at the top of the ranking, and the prediction will be the one appearing as the consequent of the highest ranking rule that covers the instance. If none of the rules are triggered, the default rule is reached, classifying the instance as the default class.

Another approach is to use a **voting scheme**, where, for each test instance, votes are accumulated for each class assigned to it by the rules it triggers. The prediction will correspond to the class with the highest number of votes, and votes may also be weighted depending on the rule that is producing it (for example, rules with lower accuracy will produce votes with lower weight).

The advantage of using an unordered rule set is that they're less susceptible to errors, since they are not biased by the chosen ordering. Model building is also less expensive, since the rules don't have to be sorted. On the other hand, classification can be more costly, since the same instance must be first compared to all the rules in the rule set before evaluating the votes.

1.3 Building a Rule Set

Rule extraction methods can be either:

- **Direct**, if the rules are extracted from the data itself;
- **Indirect**, if the rules are extracted from some other model (e.g., Decision Trees).

1.3.1 Direct Methods for Rule Extraction

To illustrate how direct methods work, we'll consider a widely-used algorithm called **RIPPER** (Repeated Incremental Pruning to Produce Error Reduction). This algorithm scales almost linearly with the number of training examples, and is particularly suited for datasets with imbalanced class distributions. It also works well with noisy data, since it uses a validation set to prevent overfitting.

RIPPER uses the **sequential covering** algorithm to extract rules from data. This algorithm uses a greedy strategy to build rules, one class at a time. For binary problems, the majority class is chosen as the default, and the algorithm learns the rules to detect only the minority class. For multiclass problems, the classes are first ordered by prevalence in the dataset; then, starting from the least prevalent class y_1, all elements belonging to it are labeled as positive, while all the rest, belonging to $y_2, y_3, \ldots y_c$, are labeled as negative. The sequential covering algorithm learns a set rules that discriminates between these positive and negative classes. Next, all instances in y_2 (the second least prevalent class) are labeled as positive, while all instances belonging to $y_3, y_4, \ldots y_c$ are labeled as negative, and a new rule set is constructed. This process is repeated until only one class remains, y_c, which is designated as the default one.

Algorithm 1 Sequential covering algorithm.

1: E = TR instances, A = set of attribute-value pairs
2: $Y_\sigma = \{y_1, y_2, \ldots, y_k\}$
3: $R = \{\}$
4: **for** each $y \in Y_\sigma - \{y_k\}$ **do**
5: **while** stopping cond is False **do**
6: $r \leftarrow$ Learn-One-Rule(E, A, y)
7: Remove TR instances from E that are covered by r.
8: $R \leftarrow R \vee r$
9: **end while**
10: **end for**
11: Insert default rule: $R \leftarrow R \vee (\{\} \rightarrow y_k)$

The algorithm always starts with an empty decision list, R, and extracts rules for each class following the ordering specified by their prevalence. The Learn-One-Rule function iteratively extracts all rules for the current class, and all training instances

covered by each rule found is removed from E. The rule is then added to the bottom to the rule list, and the loop repeats until the specified stopping criterion is met.

Rule Evaluation

In the Learn-One-Rule function, the algorithm must search for an optimal rule by growing one in a greedy fashion. It starts with a rule with an empty antecedent, $r : \{\} \rightarrow +$. Then, new conjuncts are gradually added to the antecedent in order to improve the rule's accuracy.

RIPPER uses the **FOIL's First Order Inductive Learner) information gain** as the measure to choose which conjunctive to add to the rule's antecedent.

FOIL's Information Gain

Given p_0 and n_0 the number of positive and negative examples covered by the original rule, and p_1 and n_1 the number of positive and negative examples covered by the new rule, the FOIL's information gain is defined as:

$$\textit{FOIL's inf. gain} = p_1 \times \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

RIPPER starts with a rule $r : A \rightarrow +$. It then adds one conjunct, B, generating the rule $r : A \wedge B \rightarrow +$, and the information gain is calculated for this addition. This step is repeated for different conjuncts, and the rule with the highest information gain is chosen to replace the original rule. The function stops once there's no additions that improve the information gain, so the rule covers only positive instances. Additionally, all instances covered by the rule are removed from the training set.

RIPPER also performs pruning of the rules to improve the generalization error based on their performance on a validation set. After generating a rule, the following metric is computed:

$$v = \frac{(p - n)}{(p + n)} \ ,$$

where p/n are the number of positive/negative validation instances covered by that rule. If this measure improves after removing a conjunct, the latter is permanently pruned, and the measure is again evaluated for the next conjunct. The check follows the reverse order to the one established by the insertion of conjuncts during generation. Note that a pruned rule may cover both positive and negative examples of the training set; this means that the rule is less adapted to the training data, but performs better on unseen examples.

The generation of rules is interrupted once a stopping condition is verified, such that the complexity of the model is high enough to generalize well, but not so high that it overfits the training data. Some common stopping conditions are evaluated based on the **Minimum Description Length** (**MDL**). The MDL measures the cost of a model as:

$$Cost(M, D) = Cost(D|M) + \alpha \times Cost(M) \ ,$$

where M and D are the model and the data, respectively, and α is a tuning hyperparameter (usually set to 0.5). The first term of the addition encodes the misclassification error, while the second term uses node encoding (number of children) plus encoding of the splitting condition. The cost is evaluated in terms of how many bits are needed to encode the rule set: if the addition of a rule would increase the length of the set by at least d bits, then RIPPER stops adding rules (by default, d is 64 bits). This is a form of Pessimistic Error Estimate, since it evaluates the generalization error of the model as:

$$R(T) = R_{emp} + \Omega \times \frac{k}{l} \ ,$$

where $R_{emp}(T)$ is the training error, Ω is a trade-off hyperparameter that represents the cost of adding a new rule, k is the size of the rule set, and l is the number of training instances.

RIPPER also performs additional optimization steps to determine whether the rules in the set can be replaced by better alternatives. For each rule r, two new rules are considered as replacement:

- A replacement rule r^*: a new rule is grown from scratch;
- A revised rule r': conjuncts are added to the rule r to extend it.

The rule set for r is compared with the rule sets for r^* and r', choosing the rule that minimizes the MDL.

1.3.2 Indirect Methods for Rule Extraction

Indirect methods generate a rule set by using the output of some other model, typically an unpruned decision tree. In a decision tree, each path connecting the root to a leaf can be expressed as a classification rule, where each attribute test condition encountered on the path is a different conjunct of the antecedent, and the (majority) class in the leaf node is the consequent. This section will focus on the approach followed by the algorithm C4.5rules.

A rule is generated from each path in the tree. For each rule $r : A \rightarrow y$ in the rule set, alternative rules $r' : A' \leftarrow y$ are considered, where A' is obtained by removing

one of the conjuncts in A. The simplified rule with the lowest pessimistic error rate is retained as a replacement if the error rate is also lower than that of the original rule. Eventual duplicates of the new rule are eliminated from the rule set.

After generating the rule set, C4.5rules uses a class-based ordering to rearrange the rules, so that all rules predicting the same class appear close together in the same subset. The description length of each subset is calculated, and the classes are arranged in increasing order of their total description length. This way, the subset with the lowest description length is given priority over the others, since it is assumed to contain the best set of rules.

1.4 Characteristics of Rule Based Models

Rule based classifiers are very similar to decision trees, and have about the same expressiveness. Both models construct rectilinear decision boundaries in the input space, and assign a class to each partition. Rule based classifiers, however, can allow multiple rules to be triggered for the same instance, while in decision trees, each instance can only follow one specific path. Because of this, rule based models can approximate more complex functions.

Like decision trees, they can handle different types of attributes, both continuous and categorical, and can work for both binary and multiclass classification tasks. Additionally, rule based classifiers often produce models that are easier to interpret but have comparable performance to decision trees.

They can also handle redundant attributes, since if two or more highly correlated, only one of them is chosen to be added as a conjunct. Since irrelevant attributes will show poor information gain, rule based models will tend to avoid choosing them as conjuncts. Still, as seen for decision trees, if the problem is sufficiently complex, sometimes irrelevant attributes may be chosen over other more relevant ones that show poor information gain individually, but would be useful when interacting with others.

They cannot handle missing values in the test set, as the positioning of the rules in a rule set follows a specific ordering strategy, so if a test instance is covered by multiple rules they may produce conflicting outputs.

Since RIPPER uses a class-based ordering strategy, emphasizing classes with fewer instances, these models are very well suited for imbalanced class distributions.

Chapter 2

Sequential Pattern Mining

Sequential pattern mining is the discovery of subsequences that frequently appear in a sequential dataset, i.e., finding all the subsequences whose number of occurrences is greater or equal than a user-defined threshold (*minsup*). These frequent subsequences are also called **sequential patterns**. Unlike frequent itemset mining, sequences also contain spatio-temporal information that specifies when certain transactions happen. Common examples of sequential data may be the purchase history of customers in a supermarket, genome sequences, or web browsing history.

Sequence, element, event

A sequence s is an **ordered** list of elements $s = \langle e_1 e_2 \dots e_n \rangle$. Each element (or "transaction") e_j is an **ordered** list of one or more events (or "items") $e_j = \{i_1, i_2, \dots, i_m\}$. Each event is a literal.

Each event/item can occur only once in an element/transaction, but may occur multiple times in separate elements/transactions. Events in the same element appear according to lexicographical ordering. An example of a sequence is the following:

$$s = \langle \{1, 2, 3\}\{1\}\{1, 3\} \rangle .$$

The whole sequence is delimited by angle brackets $<, >$, and each element is delimited by curly brackets $\{, \}$. This sequence contains three elements, three unique events, and a total of 6 events. The **length** of a sequence ($|s|$) is the number of its elements. The **size** of the sequence is the total number of its events. A sequence of size k is also known as a k**-sequence**.

Subsequence

A sequence $s = \langle s_1 s_2 \dots s_n \rangle$ is a subsequence of a sequence $t = \langle t_1 t_2 \dots t_m \rangle$ if there exist integers $1 \leq i_1 < i_2 < \dots < i_k \leq n$ such that $s_1 \subseteq t_{i_1}, s_2 \subseteq t_{i_2}, \dots, s_n \subseteq t_{i_k}$.

If s is a subsequence of t, then s is **contained** in t.

Let D be a dataset of one or more sequences, called data-sequences. Each data-sequence consists in a list of elements, ordered by increasing time. Each element is associated with a sequence-id, a timestamp, and a list of the events it contains. For simplicity, we will assume that elements occur at regular intervals and never overlap. For each pattern, we can calculate its **support** and **support count**, defined the same as they were defined for itemsets in association analysis.

Support

The support of a sequence s is the fraction of data-sequences in a dataset D that contain s.

Support Count

The support count of a sequence s is the absolute number of data-sequences in a dataset D that contain s.

For the purpose of sequential pattern mining, only a single valid occurrence of a sequence in a data-sequence is considered towards computing support, so even if a subsequence appears in n different ways within the same data-sequence, its support count will only be increased by 1. We can now formally define sequential pattern mining as follows:

Sequential Pattern Mining

Given D a dataset of data-sequences, and *minsup* a user-defined minimum support threshold, the problem of mining sequential patterns is to find all sequences whose support $\geq$ *minsup*; each such sequence is a **sequential pattern**, also called frequent sequence.

Discovering all frequent sequences in a dataset is a computationally challenging task. The most basic algorithm that solves the problem uses a brute-force approach: generate all possible k-sequences for $k = 1, 2, 3 \ldots$, and compute support for every single one of them. The ones whose support is greater or equal than a *minsup* threshold are declared frequent. However, the set of all possible candidate sequences is exponentially large and difficult to enumerate, even more than what was seen in association analysis. An event can appear multiple times in different elements within the same sequence, and elements arranged in different orders correspond to different sequences. This means that even when considering a relatively small set of events, the algorithm generates a large set of candidates; e.g., with only three unique events, the candidates generated for size $k = 2$ would be:

$$\langle\{i_1\}\{i_1\}\rangle, \langle\{i_1\}\{i_2\}\rangle, \langle\{i_1\}\{i_3\}\rangle, \langle\{i_1 i_2\}\rangle, \langle\{i_1 i_3\}\rangle$$
$$\langle\{i_2\}\{i_1\}\rangle, \langle\{i_2\}\{i_2\}\rangle, \langle\{i_2\}\{i_3\}\rangle, \langle\{i_2 i_3\}\rangle$$
$$\langle\{i_3\}\{i_1\}\rangle, \langle\{i_3\}\{i_2\}\rangle, \langle\{i_3\}\{i_3\}\rangle\,,$$

for a total of 12 candidates. As the number of items increases (it can easily be in the order of the hundreds, thousands, or more), the number of candidates would explode beyond what could be analyzed in appropriate time. Even if we were to generate candidates from input sequences, removing one item at a time and calculating support, we would still have a disproportionate amount of sequences to check.

One approach to solve the problem efficiently is to exploit the anti-monotonicity property of support and the Apriori property, already used for frequent itemset mining. As a reminder:

Anti-monotone property

A measure f possesses the anti-monotone property if for every itemset X that is a proper subset of an itemset Y, it holds that $f(Y) \leq f(X)$.

Apriori principle

If a k-sequence is frequent, then all of its $(k-1)$-subsequences must also be frequent.

2.1 Time Constraints

Time constraints control how support is calculated by considering the time elapsed between elements of a sequence. For example, consider a dataset that represents market basket data: each product is an event, each individual purchase is an element, and each data-sequence is a set of purchases made by a customer within some interval of time (months or years). If we're interested in finding a correlation between certain products, we may want to limit the time passed between transactions: if a customer bought product A, and then bought product B several months after, then the sequence $\langle\{A\}\{B\}\rangle$ is not significant for the purposes of our analysis. The time constraints are three: **maxspan**, **maxgap**, and **mingap**.

maxspan

The *maxspan* constraint specifies the maximum time passed between the first and last element of a sequence. If $t_{i,i+1}$ is the time passed between consecutive elements i and $i+1$ of a data-sequence, then the following inequality must hold true:

$$\sum t_{i,i+1} \leq maxspan$$

maxgap and mingap

The *maxgap* and *mingap* constraints specify the maximum time and minimum time passed between two consecutive elements of a sequence, respectively. If $t_{i,i+1}$ is the time passed between consecutive elements i and $i+1$ of a data-sequence of length n, then the following inequality must hold true for $i = 1 \ldots (n-1)$:

$$mingap < t_{i,i+1} \leq maxgap$$

The *maxgap* constraint violates the Apriori principle. A modification of this principle is used instead, which refers to **contiguous subsequences**:

Contiguous Subsequence

Given a sequence $s = \langle s_1 s_2 \dots s_n \rangle$, a sequence t is a contiguous subsequence of s if:

- t is obtained by dropping an event from either s_1 or s_n;
- t is obtained by dropping one event from any element s_i that contains more than one event;
- t is a contiguous subsequence of w, and w is a contiguous subsequence of s.

The Apriori principle can then be modified in the following way:

Modified Apriori Principle

If a k-sequence is frequent, then all of its **contiguous** $(k-1)$-subsequences must also be frequent.

2.2 Generalized Sequential Patterns Algorithm

the Generalized Sequential Patterns (GSP) algorithm is an efficient algorithm that uses the anti-monotonicity of support to extract sequential patterns; it also supports time constraints. It is very similar to the Apriori algorithm, with the same exact basic structure. The pseudocode of the algorithm is presented in the next pseudocode block.

The algorithm does a first pass over the dataset and computes support for all unique events, determining which 1-sequences (sequences with only a 1-event element) are frequent. The main loop of the algorithm has a candidate generation phase, a candidate pruning phase, and finally a support counting phase.

2.2.1 Candidate generation

This phase generates new candidate k-sequences by merging together the frequent $(k-1)$-sequences found in the previous iteration. There's two possible cases:

- For $k = 2$, all frequent 1-sequences are merged with each other (including with themselves). For each couple of events i_1 and i_2, the generated candidates will be: $\langle \{i_1\}\{i_2\} \rangle$, $\langle \{i_2\}\{i_1\} \rangle$, and $\langle \{i_1 i_2\} \rangle$, if $i_1 \neq i_2$; only $\langle \{i_1\}\{i_2\} \rangle$, if $i_1 = i_2$.

Algorithm 2 Generalized Sequential Patterns pseudocode.

```
1: k = 1.
2: F_k = {i : i ∈ I ∧ s(i) >= minsup} # find all frequent 1-sequences
3: repeat
4:     k = k + 1
5:     C_k = candidate-gen
6:     C_k = candidate-prune(C_k, F_{k-1})
7:     for all t ∈ T do
8:         C_t = subsequences(C_k, t)
9:         for all c ∈ C_t do
10:            σ(c) = σ(c) + 1
11:        end for
12:    end for
13:    F_k = {c|c ∈ C_k ∧ s(c) ≥ minsup} # find all frequent k-sequences
14: until F_k = ∅
```

- For $k > 2$, two frequent $(k-1)$-sequences s_1 and s_2 are merged only if the subsequence obtained by dropping the first event from s_1 is the same as the one obtained by dropping the last event from s_2. Then, the candidate can be generated in two ways.

  If the last element of s_2 has only one event, append that last element to s_1 and obtain the merged sequence.

  If the last element of s_2 has more than one event, append the last event of that last element to the last element of s_1 and obtain the merged sequence.

Note that a sequence can, in some cases, be merged with itself, as long as the conditions described above hold true. Also, this procedure is both complete and generates no duplicates.

2.2.2 Candidate Pruning

A k-candidate can be pruned if at least one of its $(k-1)$-subsequences is infrequent, since support shows anti-monotone property, and therefore its support can only be less-or-equal-than any of its subsequences. Pruning is done by dropping one event at a time from the k-candidate, and checking if the resulting $(k-1)$-sequence is contained in the frequent ones found in the previous iteration. If any of them are not frequent, the candidate can be discarded.

2.2.3 Support Counting

After the candidate set is pruned, the algorithm iterates over the data-sequences, and for each of them finds which k-candidates it contains, increasing their support count accordingly. At the end, all k-candidates whose support is less than *minsup* are discarded, while the rest form the set of frequent k-sequences.

2.3 Generalized Sequential Patterns and Time Constraints

Introducing the *mingap*, *maxgap*, and *maxspan* time constraints requires the support counting and candidate pruning procedures to be modified. Support counting must now consider the time gap between consecutive elements (for the *mingap* and *maxgap* constraints) and the overall span of the sequence (for the *maxspan* constraint) when determining if a candidate is contained in a sequence. This means that the procedure can't simply determine the first occurrence of a candidate within a data-sequence, but must keep searching for an occurrence that satisfies all three constraints at once, such that $mingap < t_{i,i+1} \leq maxgap$, and $\sum t_{i,i+1} \leq maxspan$, where $t_{i,i+1}$ is the gap between consecutive elements in a data-sequence.

As for candidate pruning, the Apriori principle and anti-monotonicity of support no longer hold true because of the *maxgap* constraint. If a candidate c is being pruned, and all of its subsequences are checked, some of them may be infrequent, even though the candidate is actually a frequent sequence.

For example, let sequence $s = \langle\{1,2\}\{2\}\{3\}\{4\}\rangle$ be a data-sequence, and sequence $c = \langle\{1\}\{2\}\{4\}\rangle$ a candidate sequence. If $maxgap = 2$, then c is contained in s, but the subsequence $c' = \langle\{1\}\{4\}\rangle$ is not, because the gap between $\{1\}$ and $\{4\}$ is 3.

Therefore, the procedure must check all of a candidate's contiguous subsequences, removing one event at a time only from the elements that contain two events or more. If at least one of the contiguous subsequences of a candidate is infrequent, the candidate can be pruned. This new pruning strategy ensures that no candidate frequent sequences are accidentally discarded, since it skips all the elements that, when removed, could produce a sequence that contains a gap between consecutive elements that violates the *maxgap* constraint. However, this strategy inevitably decreases the effect that pruning has on the overall execution.

Chapter 3

Transactional Clustering

Clustering is a task whose objective is to find grouping of objects of a dataset into sets, called clusters, where each member of a cluster is more similar to all other elements in the same cluster than it is to elements in different clusters. For numerical data, proximity between objects is measured using distances, typically Euclidean or Manhattan, since records with numerical attributes only can be interpreted as points in an n dimensional space. However, these measures are not appropriate to carry clustering tasks on categorical attributes, which are used to represent transactional data. The biggest issue is that the way two transactions are deemed "near" each other does not correspond to geometrical proximity.

This issue can be illustrated with a simple example. Assume we have 4 transactions:

$$T1 = \{1, 2, 3, 4\}$$
$$T2 = \{1, 2, 4\}$$
$$T3 = \{3\}$$
$$T4 = \{4\}$$

Usually, transactional data can be represented with a set of boolean attributes, one for each item, where a value of "1" indicates the presence of the corresponding item, while a "0" indicates its' absence. In this case, the dataset will be represented as:

$$P1 = \{1, 1, 1, 1\}$$
$$P2 = \{1, 1, 0, 1\}$$
$$P3 = \{0, 0, 1, 0\}$$
$$P4 = \{0, 0, 0, 1\}$$

If Euclidean distance were used to calculate how far or close there transactions are to each other, we would find out that the distance between transactions 3 and 4 is:

$$d(P3, P4) = \sqrt{(0)^2 + (0)^2 + (1)^2 + (-1)^2} = \sqrt{2}$$

However, the two transactions don't share any item, making them completely different.

This chapter will present four different algorithms that can cluster categorical data, each of which define proximity between transactions in different ways.

3.1 K-Modes

K-modes is similar to K-means, but can be used for categorical attributes. It starts by randomly choosing k data points at random to elect to representative points. The algorithm then enters a loop, in which it assigns all points to the closest mode, re-computes the mode of each cluster, and repeats the past two steps until no object changes assignment between two successive iterations (or until some stopping criterion is verified).

The distance between two records is calculated as the number of mismatches between their attributes:

$$d(X, Y) = \sum_i \delta(x_i, y_i)$$

$$\delta(x, y) = \begin{cases} 0 & (x = y) \\ 1 & else \end{cases}$$

The representative object of a cluster is calculated as the mode of the objects in the same cluster.

This algorithm minimizes the function:

$$P(W, Q) = \sum_i^k \sum_j^n w_{i,j} d(x_i, Q_i)\,,$$

where $w_{i,j}$ is 1 if object i belongs to cluster j, 0 otherwise, and Q is the set of the modes of each cluster.

3.2 ROCK (RObust Clustering using linK)

ROCK is a hierarchical algorithm that uses **neighborhoods** and **links to clusters** to define closeness between two transactions. Neighborhoods are calculated locally, while links are calculated globally. The algorithm can be split into three main parts:

1. **A random sample is drawn from the dataset**. A sample of points is uniformly extracted, using it to form clusters instead of the entire data. This ensures the algorithm can be used even on very large dataset while still producing an accurate enough clustering.

2. **An agglomerative hierarchical clustering algorithm is performed on the sample**. ROCK follows the same steps as other hierarchical agglomerative algorithms: it starts by assigning each point to a singleton cluster, then computes the similarity measure for all pairs of clusters, and merges the two "closest" objects. These steps repeat until some stopping condition is met (usually, until k clusters are formed).

   Two objects A and B are **neighbors** if their similarity is greater or equal than some hyperparameter threshold θ, chosen between 0 and 1:

   $$A \in N_B \wedge B \in N_A \iff sim(A, B) \geq \theta \,,$$

   where similarity is calculated with Jaccard's coefficient (the ratio of the number of matching items between the objects and the total number of distinct items between the two):

   $$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

   A point is also considered a neighbor of itself.

   A **link** is calculated as the number of common neighbors between two objects:

   $$link(A, B) = |N_A \cap N_B|$$

   Higher values of link means that there's a higher probability that the two objects belong to the same group (since they share neighbors).

3. **The entire dataset is labeled by assigning each object to a cluster**. A random sample is selected from each cluster, and each point p in the original dataset is assigned to the cluster i such that p has the maximum number of neighbors in the corresponding sample.

The best clusters are the ones that maximize the criterion function of the algorithm:

$$E_l = \sum_{i=1}^{k} n_i \sum_{p_q, p_r \in C_i} \frac{link(p_q, p_r)}{n_i^{1+2f(\theta)}} \,,$$

$$f(\theta) = \frac{1-\theta}{1+\theta}$$

where n_i is the size of cluster C_i. This function penalizes clusters that present very few links compared to the expected number of links in the entire cluster, so that we avoid that objects with a low number of links are assigned to the same cluster. At each merging step of the algorithm, the two clusters that are merged are the ones that maximize the goodness measure:

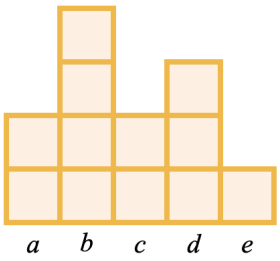
$$g(C_i, C_j) = \frac{link(C_i, C_j)}{(n_i + n_j)^{1+2f(\theta)} - n_i^{1+2f(\theta)} - n_j^{1+2f(\theta)}},$$

where the numerator is the number of cross-links between the two clusters, and the denominator is the expected number of them.

3.3 CLOPE (Clustering with sLOPE)

CLOPE is a clustering algorithm that is efficient for high dimensional data. It uses an exclusively global criterion function that tries to increase the intra-cluster overlap of transactions. This is done by increasing the height-to-width ratio of the cluster histogram. It is especially suitable for big datasets with a high number of unique items, since it uses an array representation of the data instead of binary.

For each cluster, the width is calculated as the number of distinct items, while the height is calculated as the ratio between the total number of (non unique) items and the width. In the example below, the cluster has a width of 5 and a height of 2.4.



$$W(C_i) = 5$$
$$S(C_i) = 12$$
$$H(C_i) = \frac{12}{5} = 2.4$$

$$C_i = \{abcd, bcd, ab, bde\}$$

Higher ratios of height/width mean higher item overlapping.

The goodness of a clustering is calculated as the **gradient** of each cluster:

$$Profit_r(C) = \frac{\sum_{i=1}^{k} \frac{S(C_i)}{W(C_i)^r} \times |C_i|}{\sum_{i=1}^{k} |C_i|}$$

The hyperparameter r is called **repulsion**; for higher values of r, transactions within the same cluster must share a large portion of items, while for lower values transactions may share a lower amount of items, which can be useful for sparse databases.

The algorithm has two phases: first, each transaction is added to a new cluster or to an existing one such that the profit is maximized. Then, for each transaction, it is checked whether moving it to a different cluster improves profit, repeating this step until all transactions remain in the same cluster (no moves will improve the profit).

3.4 TX-Means

TX-Means is a parameter-free transactional clustering algorithm, and is useful to partition data obtained from a massive amount of different datasets. It finds a representative transaction for each cluster, which summarizes the pattern presented by the elements of that cluster. Like X-Means, it starts out with a cluster that contains all the objects in the dataset, and chooses how to recursively split it into subpartitions by using the Bayesian Information Criterion.

Algorithm 3 TX-Means pseudocode.

```
1: r = getRepr(B) # get representative basket of entire set of baskets
2: r is added to queue Q
3: while Q ≠ ∅ do
4:     C, r are extracted from Q
5:     Common items are removed from C and r
6:     C1, C2, r1, r2 = bisectBasket(C)
7:     if BIC(C1, C2, r1, r2) > BIC(C, r) then
8:         Add C1, C2, r1, r2 to Q
9:     else
10:        Add C, r to result
11:    end if
12: end while
13: Return result
```

The algorithm starts by finding a representative for all objects in the dataset, then enters a loop in which each cluster currently in the queue is split. Each split is either accepted, reinserting the new subpartitions in the queue, or rejected, adding the parent cluster to the result, depending on whether it produces an improvement in the BIC score. Below is the pseudocode for the functions `getRepr()` and `bisectBasket()`.

Algorithm 4 `getRepr` pseudocode.

I = set of items not shared among all baskets in B
r = set of items in common to all baskets in B
Calculate frequencies of items in I
$i = 0$, $d_0 = \text{inf}$
while $I \neq \emptyset$ **do**
    $i = i + 1$
    Add the items in I with maximum frequency to r
    Calculate the distance d_i between r and the baskets in B via Jaccard coefficient
    if $d_i \geq d_{i-1}$ **then**
        Return r
    else
        Remove from I items with maximum frequency
    end if
end while
Return r

Algorithm 5 `bisectBasket` pseudocode.

$SSE = \text{inf}$
Select two random baskets $r1, r2$
while True **do**
    $C1, C2$ = clusters obtained by assigning baskets to either $r1$ or $r2$
    $r1_{new} = \texttt{getRepr}(C1)$
    $r2_{new} = \texttt{getRepr}(C2)$
    $SSE_{new} = SSE(C1, C2, r1_{new}, r2_{new})$
    if $SSE_{new} \geq SSE$ **then**
        Return $C1, C2, r1, r2$
    end if
    $r1 = r1_{new}$
    $r2 = r2_{new}$
end while

TX-Means is also scalable thanks to the following sampling strategy. A random subset of transactions is chosen from the dataset, and TX-Means is run on that subset, returning a set of clusters and their respective representative transactions. Then, all the remaining transactions in the dataset are assigned to the clusters using a nearest neighbor approach with respect to the representatives found by the algorithm.

Chapter 4

Time Series

A **time series** is a collection of observations made sequentially in time, usually at constant time intervals. They can be constructed out of measurements of many different phenomenons, such as annual rainfall levels, earthquakes, fMRI data, quarterly earnings, audio/video data, and so on. Time series can analyzed through clustering, classification, motif discovery, rule discovery, forecasting, and trend/seasonality analysis. The key issues that arise when working with time series are:

- **The amount of data to work with can be incredibly large**: for datasets with several different sources and short intervals of time between measurements, the size can be very high.
- **Similarity is not easy to estimate**: since series are complex objects with several values each, defining how two series can be considered similar is not as easy as it can be for numerical data.
- **Different data formats**: different series in the same dataset may be represented using a different scale or format; e.g., atmospheric temperatures recorded in both $^{\circ}C$ and $^{\circ}F$.
- **Different sampling rates**: while usually it is assumed that series are recorded all with the same rate, in many real life cases this may not be true. Different series may have different lengths and different time intervals.
- **Noise, missing values, and other defects**: as with other types of data, time series can also present noisy information or missing values.

The following sections will explain how some of these issues can be dealt with.

4.1 Similarity Between Time Series

4.1.1 Structural-based Similarities

Especially when analyzing long time series, similarity is calculated on a structural level. This means that global features are extracted from the time series, creating a feature vector, and measuring similarity looking at those features. Some examples are the mean and variance, maximum/minimum, skewness, mean and variance of the 1^{st} derivative, and so on.

A measure used to calculate structural dissimilarity is **compression based dissimilarity**, which is calculated as:

$$d(x, y) = CDM(x, y) = \frac{C(x, y)}{C(x) + C(y)},$$

where C is a compression algorithm. The numerator is the compression of the concatenation of the two series, while the denominator is the sum of the compressions of the series done singularly: the Compression Dissimilarity Measure equal to 1 if the two series are unrelated, otherwise it is less than one. The smaller its value, the closer the series are to each other. CDM is never zero.

4.1.2 Shape-based Similarities

Shape-based similarities can be calculated using distance measures. Recall that distances have the following properties:

$$d(a, b) \geq 0, d(a, b) = 0 \iff a = b \text{ (Positivity)}$$
$$d(a, b) = d(b, a) \text{ (Symmetry)}$$
$$d(a, c) \leq d(a, b) + d(b, c) \text{(Triangle Inequality)}$$

One way to calculate shape-based dissimilarity is using Euclidean distance. Given two time series (with the same exact number of points/measurements), the Euclidean distance is calculated as if the series were points in an Euclidean space. However, this distance is very sensitive to distortions in the data, which should be removed in a preprocessing phase.
The most common distortions are:

- **Offset translation**: the series have different offsets on the y axis. Calculating the distance without addressing this problem would return a value that is drastically larger than the actual shape distance. This can be corrected by subtracting each series with its own mean value.

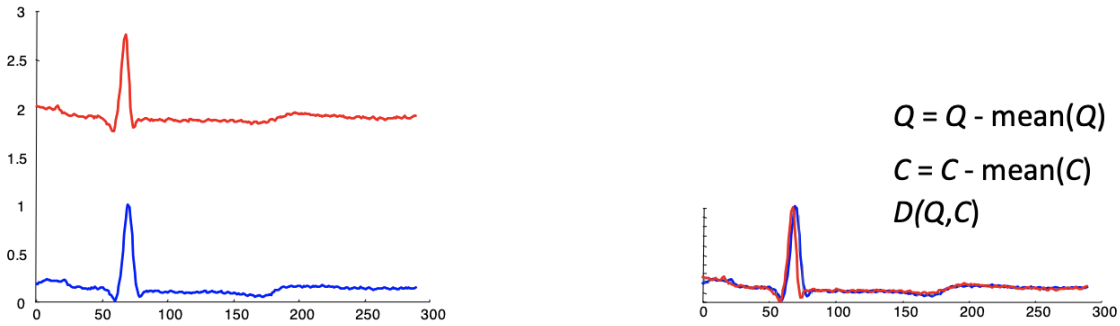


Figure 4.1: Offset translation transformation.

- **Amplitude scaling**: the series have the same shape, but are scaled differently on the y axis. This is corrected by applying Z-score normalization to the values of the series (subtracting the mean and dividing by the standard deviation).

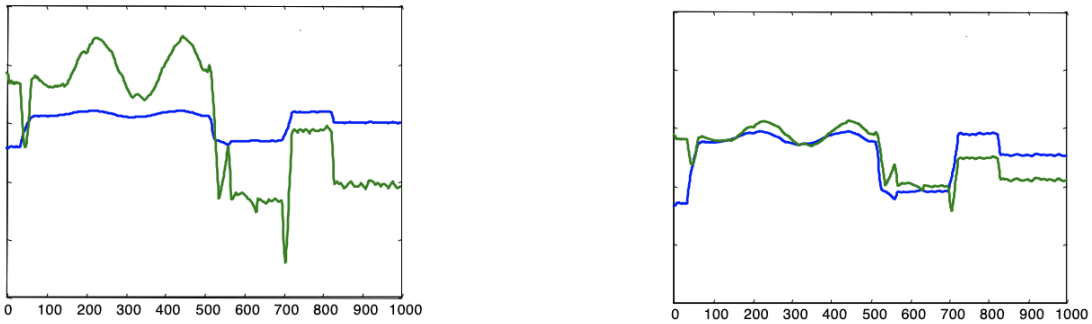


Figure 4.2: Amplitude scaling transformation.

- **Linear trend**: series can follow upwards or downwards linear trends, which means that as they progress they increase or decrease in level. This can be corrected by finding the best fitting line to a time series, and subtracting it from the time series itself.

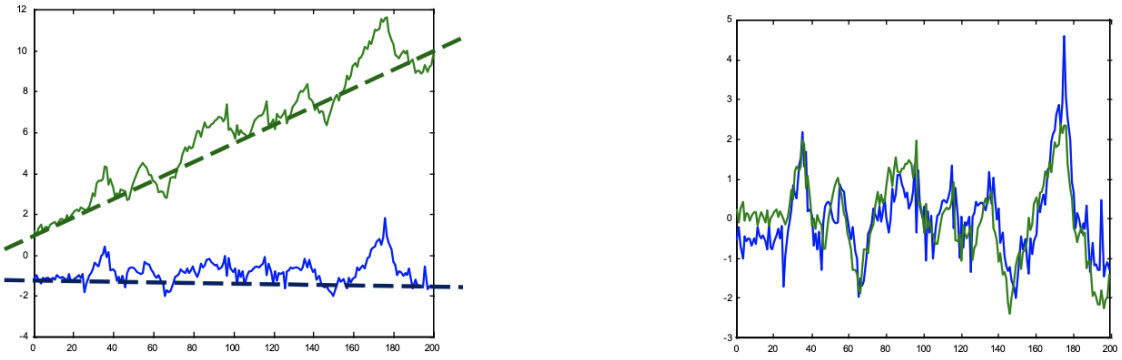


Figure 4.3: Linear trend transformation.

- **Noise**: noise is the presence of random error in the series. To remove noise, each data point can be replaced with the average value of its neighbors.

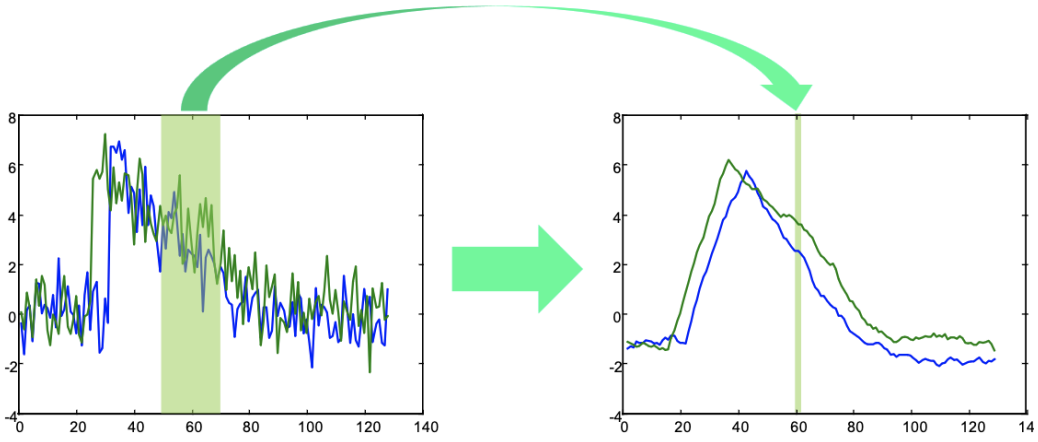


Figure 4.4: Noise transformation.

Noise can be removed by a **moving average** (**MA**): given a window of length w and a time series t, the MA is applied as:

$$t_i = \frac{1}{w} \sum_{j=i-w/2}^{w/2} t_j \; i = 1, \ldots, n$$

Dynamic Time Warping

It's often the case that two time series have approximately the same shape, but they do not line up on the x axis. The euclidean distance calculated between such series would be high, despite them being similar: this is because it considers a fixed time axis for both series. In order to find the similarity between them, the time axis must be "warped" for one or both time series to align them correctly.

In practice, DTW is calculated in three steps. First, given the series Q and C, a matrix of size $|Q| \times |C|$ is constructed, and each cell of index i, j contains the distance between the i^{th} component of Q and the j^{th} component of C. The diagonal of this matrix corresponds to the comparison done by the Euclidean distance. Every possible warping between two time series corresponds to a path from the bottom left corner $(0, 0)$ and the top right corner $(0, |C|)$ of the matrix. The DTW will be the best path, i.e., the one that yields the lowest sum of costs. The best path is found recursively, using the following formula:

$$\gamma(i, j) = d(q_i, c_j) + \min\{\gamma(i-1, j-1), \gamma(i-1, j), \gamma(i, j-1)\}$$

The dynamic programming approach to the problem starts by calculating the distance matrix for the two series; then, the matrix of cumulative costs for each path; finally, the path with the best alignment is found, connecting the cells with the lowest cost.

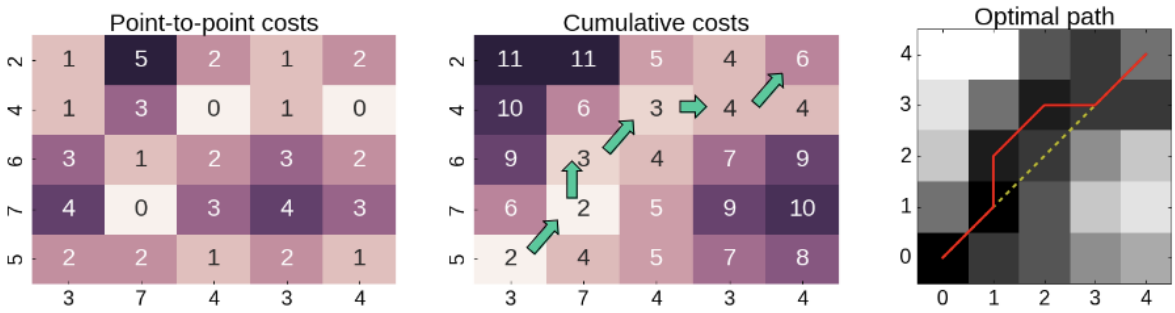


Figure 4.5: How Dynamic Time Warping is calculated.

When the performances of Euclidean distance and DTW are compared for classification tasks, the former leads to much lower accuracy than the latter; however, DTW is also two to three orders of magnitude slower than Euclidean distance, meaning that it is unsuitable for larger datasets if used as is. In order to speed up the calculation of DTW, different approaches can be used depending on the length of the time series.

- If the time series are short, then they can be **approximated** via compression or downsampling; alternatively, computation can be sped up by introducing **global constraints**.
- If the time series are long, compression/approximation-based dissimilarity can be used.

Global Constraints A global constraint limits the indices of the warping path $w_k = (i, j)_k$, such that $j - r \leq i \leq j + r$, where r is an integer that defines the allowed range of warping for a given point in the series. This restricts the computation of distances and cumulative costs to a small window described by r. Two types of global constraints are the **Sakoe-Chiba Band** (restriction around the diagonal) and the **Itakura Parallelogram** (greater restriction at extremes of series, less restriction in the middle). Empirical analysis shows that given a dataset, the increase in the value of r will rapidly improve performances, but after reaching a maximum they will decrease and then plateau.

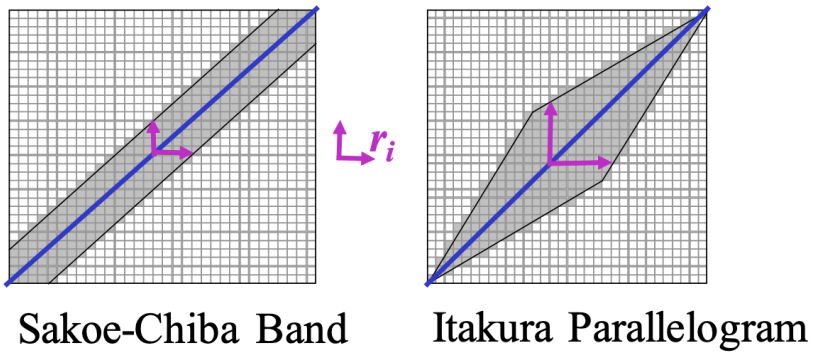


Figure 4.6: Global constraints.

Approximation Approximation is a form of Dimensionality Reduction designed for time series. Compared to compression, approximation maps the series to a smaller space that is understandable, while the compressed space is not necessarily understandable.

- **Discrete Fourier Transform (DFT)**: the time series is represented as a linear combination of sine and cosine functions, but only the first $n/2$ coefficients are kept. Each sine wave only requires 2 numbers: the phase and the amplitude. Many of the coefficients have very low amplitude, so they contribute very little to reconstructed signal: they can be discarded without significant loss of information, saving storage space.

  This approximation works great for most time series that represent natural signals, and many fast ($O(n \log(n))$) implementations exist. However, it struggles with series of different lengths and cannot support weighted distance measures. This approximation loses the temporal information of the original time series.

- **Discrete Wavelet Transform (DWT)**: the time series is represented as a linear combination of wavelet basis functions, but only the first n coefficients are kept. Wavelets represent data in terms of sum and difference of a prototype wavelet, called analyzing/mother wavelet; they are localized in time, so some of the wavelet coefficients will represent small subsections of the data (unlike Fourier coefficients which always represent global contribution to data).

  This approximation has a good ability to compress stationary signals, and many fast ($O(n)$) implementations exist. However, it can only be defined for sequences whose length is an integral power of two; wavelets tend to approximate the left side of the sequence at the expense of the right one; it cannot support weighted distance measures. This approximation loses the temporal information of the original time series.

- **Singular Value Decomposition (SVD)**: the time series is represented as a linear combination of eigenwaves, but only the first n coefficients are kept. This approach is similar to the previous two, but the eigenwaves are extracted from the data and are not a set of default waves.

  Time series can be thought of as points in a high-dimensional space, so the axes on which they are represented can be rotated so that axis 1 is aligned with the direction of maximum variance, axis 2 is aligned with the direction of maximum variance and is also orthogonal to axis 1, and so on, until the desired number of axes is obtained. Since the first eigenwaves will capture the most variance in the data, the rest can be truncated with little loss. This approximation loses the temporal information of the original time series.

- **Piecewise Linear Approximation (PLA)**: the time series is represented as a sequence of k straight lines, which can be either connected or disconnected. Each line is described by its length and the height of its leftmost point; the height of the rightmost one can be inferred from the next segment.

  This approximation requires to choose the "best" value of k, that is, the optimal number of segments used to represent the time series that corresponds to the best trade-off between accuracy and compactness; this problem has no general solution.

  This approximation can efficiently compress data and filters noise. It also supports non-euclidean similarity measures.

- **Piecewise Aggregate Approximation (PAA)**: the time series is represented as a sequence of N box basis functions, where each box has the same size. The mean value of the points in each segment is calculated, so that the approximation will be a series made up of several constant lines of fixed length. The dimensionality of the data is reduced from n to N (the number of segments). If $N = 1$, the transformed representation is identical to the original series; if $N = n$, the approximation is a single segment equal to the mean of the entire series.

  This approximation is fast to calculate, supports Euclidean and non-Euclidean distance measures, as well as weighted Euclidean distance.

- **Adaptive Piecewise Constant Approximation (APCA)**: this approximation was developed as an extension of PAA. It allows the segments in the approximation to have different lengths, so that each segment is represented by two values: the first one corresponds to the mean value of the points falling within the same segment, the second one is its length.

APCA can place bigger segments for areas of lower activity and many smaller segments for areas with higher activity (hence the "adaptive"). In general, finding the optimal piecewise polynomial representation of a time series requires a $O(Nn^2)$ dynamic programming algorithm; in practice, however, an optimal representation is not needed, so fast algorithms ($O(n \log(n))$) to calculate high quality approximations exist. It supports Euclidean and non-Euclidean distance measures, as well as weighted Euclidean distance.

- **Symbolic Aggregate Approximation (SAX)**: a time series can be **segmented** using a predefined length w, predefined number of segments k, or using change point detection methods. SAX converts a time series into a discrete sequence of symbols, using a small alphabet size. Every part of the representation contributes about the same amount of information about the shape of the time series. The series is first normalized, and then two steps of discretization are performed.

  First, the time series of length n is split into w equal-sized segments, and each segment is approximated by the mean of all the points falling in it. Aggregating the resulting w coefficients forms the PAA representation of the time series. Next, the breakpoints that divide the representation into α equiprobable regions are found, where α is the alphabet size (can be set by the user or derived from the Minimum Description Length). These breakpoints are chosen so that the probability of a segment falling into any of the regions is approximately the same: if the symbols were not equiprobable, some substrings would be more probable than others, introducing a probabilistic bias.

  Each region is assigned a different symbol from the chosen alphabet. The PAA coefficients can be mapped to the region to which they reside in a bottom-up fashion: the coefficients that fall in the lowest region are converted to a, the ones in the second lowest region to b, and so on. At the end, the representation will correspond to a sequence of symbols.

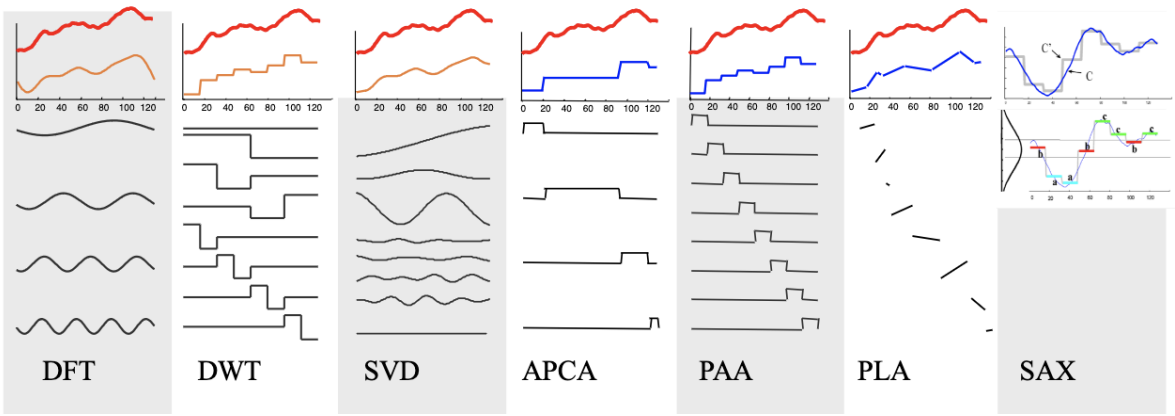


Figure 4.7: The most common time series approximations.

Chapter 5

Time Series: Clustering and Classification

5.1 Clustering

For time series, the most common methods are Partitional Clustering and Hierarchical Clustering. **Partitional Clustering** uses the K-means algorithm (or some variant) to optimize the objective function by minimizing the SSE. **Hierarchical Clustering** starts by computing pairwise distance between time series (using whatever appropriate distance measure is specified by the user), and then merges clusters in a bottom-up way, merging at each step the two closest clusters, until the cluster containing all data points is obtained. This approach is however limited to small datasets, since it is computationally complex.

Time series clustering can be of the following types:

- **Whole clustering**: the conventional type of clustering, where the goal is to assign each data object to a cluster.
- **Feature-based clustering**: features (or motifs, see next section) are extracted from the series, and then used to cluster.
- **Compression-based clustering**: compress time series and run clustering on the compressed versions.
- **Subsequence clustering**: subsequence clustering is done on the subsequences extracted from a single long time series using a sliding window.

5.2 Motif and Discord Discovery

Motifs are repeated patterns within a time series. **Discords** are exceptionally unusual patterns withing a time series. Motif discovery is a preprocessing phase for other analysis: mining association rules, where motifs are referred to as primitive shapes and frequent patterns, classification, which in some cases works by constructing typical prototypes (motifs) of each class, and anomaly detection, in which algorithms use motifs to model typical time series behaviour and detect future patterns that are dissimilar.

Given a predefined motif length m, a brute-force approach would simply search all possible motifs obtainable from all possible comparisons of subsequences of length m. This approach is clearly inefficient; the most commonly used algorithm was originally developed in bioinformatics, and is based on **random projections** and SAX.

First, an alphabet size α, SAX window size w, and motif length n are set. All approximated subsequences of length w are extracted via SAX from the time series by shifting forward the window one measurement at a time, for a total of $|T| - (n-1)$ subsequences. Then, a mask is randomly chosen to only select certain "columns" of the subsequences; i.e., if the mask $\{1, 3\}$ is chosen, only the first and third symbols in the representation are considered. Collisions between masked subsequences are recorded with a **collision matrix**, increasing the value in the corresponding cell. This step is repeated multiple times choosing different masks.

At the end of these random perturbations, motifs can be observed in the collision matrix, looking at the cells that have the highest values: their indexes are the positions in which the motifs start in the time series. The problem with this approach, however, is that it is highly dependent on the approximation technique used.

5.2.1 Matrix Profile

Given a time series T, and having calculated the pairwise distance among all the $|T| - (n-1)$ subsequences that can be extracted from T using a sliding window of length n, the **Matrix Profile** (**MP**) of a T is the vector that annotates the distance between each subsequence and its nearest neighbor. The index of the corresponding nearest neighbor of each subsequence is stored in a vector called **Matrix Profile Index**, which can be used to find the nearest neighbor in constant time. Pointers are not necessarily symmetric: if n is the nearest neighbor of m, the nearest neighbor of n is not necessarily m, but may be some other subsequence. However, for the two smallest values in the MP, the pointers of the corresponding subsequences must be mutual.

Low values in the matrix profile indicate that the corresponding subsequence has at least one similar subsequence in the data: these regions indicate motifs. Areas that

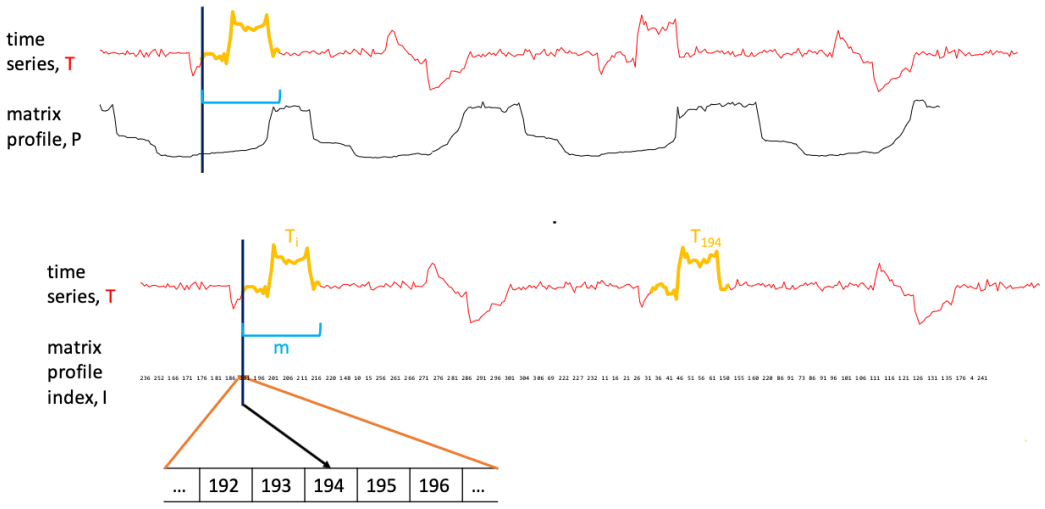


Figure 5.1: Matrix Profile (on top) and Matrix Profile Index (on the bottom).

instead have very high values indicate discords, since they are subsequences whose nearest neighbor is very distant.

To compute the matrix profile of a time series, the cells of the vector are initialized to ∞. Then, a random subsequence T_i is selected, and the distance with every other subsequence is stored in a different vector. This step has complexity $O(|T| \log(|T|))$. The matrix profile is updated, applying element-wise minimum to the two vectors (skipping the cell for T_i). A new subsequence is randomly selected, and the process is repeated, updating the matrix profile with the new minimum values. The algorithm stops once all subsequences have been selected. The total time complexity is $O(|T|^2 \log(|T|))$.

It may be useful to think of time series subsequences as points in an m dimensional space: subsequences that are very similar will be close together in denser areas, which will in turn correspond to regions in the MP with low values. To understand how the top-k motifs are extracted, we can consider this data-point interpretation. A parameter $R > 1$ is chosen, and the two nearest points are found, called the **motif pair**. Given the distance D_1 between these two points, a circle with radius $D_1 * R$ is drawn around each point: any point that falls within either circles are added to this motif: this is the top-1 motif. To find the top-2 motif, the next closest pair of points is found (excluding the ones in the previous motif), whose distance is $D_2 > D_1$. Again, a circle of radius $D_2 * R$ is drawn around each circle, and all points in this circle are added to the motif. To choose when to stop, i.e., to choose the value of k, we can either use a predefined value, or use the Minimum Description Length.

If we're interested in finding top-k discords instead, a parameter E is set to select how many subsequences will be excluded in the vicinity of the anomaly, and the subsequence

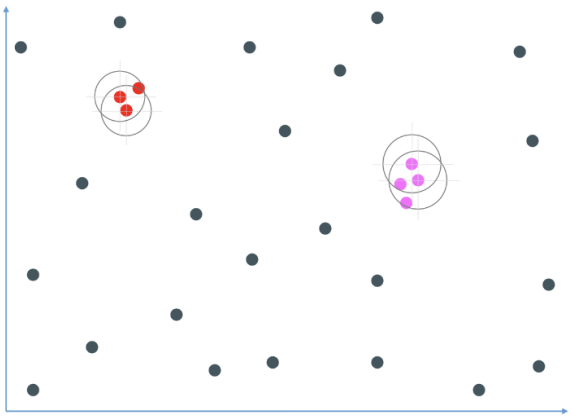


Figure 5.2: A graphical interpretation of how top k-motifs are found. The red dots are the top 1-motif, and the magenta ones are the top-2 motifs.

with the highest distance in the MP is found. The E closest subsequences to the anomaly are selected, and removed with the anomaly. Again, the value of k can be either set to a predefined value or chosen as the MDL.

5.3 Classification

For classification tasks, given a set of n time series all assumed of length m, each time series x_i is associated with a class label c_i. The objective is to find a target function f that maps all possible time series to the space of possible class labels. The most widely used algorithm for time series classification is K-NN, used in the raw data. It is simple, and can be used with Dynamic Time Warping. However, it is a lazy classifier, meaning that prediction is costly as it is, and DTW slows the execution even further. Additionally, K-NN based classification does not provide much insight into the data.

An alternative is shapelet-based classification. **Shapelets** are time series subsequences that are maximally representative of a class. Once extracted, shapelets can be used to transform the dataset so that it can be used as input for classifiers; additionally, they provide interpretable results, and can be incredibly accurate since they are local features, unlike most other time series classifiers which only consider global features. They also tend to be faster at classification compared to other methods, since the time complexity of the prediction phase is only $O(ml)$, where m is the length of the query time series, and l is the length of the shapelet. In contrast, DTW-based K-NN has a time complexity of $O(km^3)$, where k is the number of objects in the training set.

Since shapelets are much shorter than the time series they're extracted from, we need to define a measure to evaluate the similarity between a subsequence and a time

series. The distance between two time series T and S, with $|S| < |T|$, is calculated via:

$$SubsequenceDist(T, S) = \min(Dist(S, S')) \; \forall S' \in S_T^{|S|}$$

where $S_T^{|S|}$ is the set of all possible subsequences of T. This function returns the distance between S and the "best matching" location in T. Each time series in a dataset can be represented as a vector of distances with the shapelets extracted from them: if a given time series belongs to a certain class, it will be very close to the representative shapelet(s) of that specific class, and further away from shapelets that represent other classes.

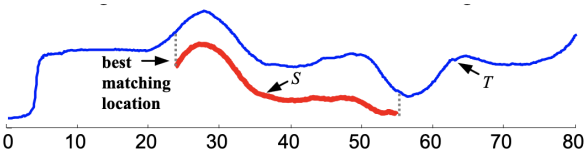


Figure 5.3: The best matching location of S over T.

5.3.1 Shapelet Extraction

The simplest way to extract shapelets is the brute-force approach, which generates all possible subsequences of all possible lengths from the time series in the dataset, adding them to the pool of candidates. Assume the time series are each assigned to one of two possible class labels. For each candidate, the algorithm must check how well it separates the objects of one class from the other, and choose the candidate that performs best. First, all time series are rearranged in the dataset based on the distance from the current candidate; then, the optimal split point that maximizes the **information gain** is found, similarly to how splits are chosen in Decision Tree training algorithms. After calculating the information gain of all candidates, the one with the highest value is selected.

A common measure used to evaluate information gain is entropy:

$$I(D) = -p(A)\log_2(p(A)) - p(B)\log_2(p(B))\,,$$

where $p(A)$ and $p(B)$ are the proportion of objects belonging to each respective class A and B. Given a strategy that divided the dataset D into two subsets D_1 and D_2, the information remaining in the dataset after the split is calculated as the weighted average entropy of each subset. If the fraction of objects in D_1 is $f(D_1)$, and the fraction of objects in D_2 is $f(D_2)$, the total entropy of the dataset after a split is calculated as:

$$\hat{I}(D) = f(D_1)I(D_1) + f(D_2)I(D_2)$$

The information gain for a splitting rule is given by the difference of the entropy before and after the split:

$$Gain(split) = I(D) - \hat{I}(D)$$

The total number of candidates generated by the brute-force approach amounts to:

$$\sum_{l=minlen}^{maxlen} \sum_{T_i \in D} (|T_i| - l + 1)$$

Then, for each of these candidates, the distance with every training time series must be calculated, as well as the entropy for each split. Obviously, this solution is highly space and time inefficient; two speedup methods are:

- **Distance Early Abandon**: in the brute-force approach, the distance from the time series T with the subsequence S is done by calculating the Euclidean distance between each subsequence of length $|S|$ in T and S, and choosing the minimum. This operation costs $O(|T|)$. However, the only distance we need is the one we keep, i.e., the minimum one. Instead of calculating all distances, the calculations can stop after the distance starts to increase compared to the smallest value recorded so far; this technique is known as early abandon.

- **Admissible Entropy Pruning**: we want to find only the best shapelet for each class. Obtaining the distance between a candidate and its nearest matching subsequence in each time series in the dataset is the most expensive operation. Instead of waiting until all distances from all time series are calculated, an upper bound of the information gain can be calculated based on the distances known so far. If at any point this upper bound cannot beat the best-so-far information gain, the distance calculations are stopped, pruning the candidate, since there's no way it could ever be better than the best candidate found so far. This is done by comparing the information gain of the best candidate with the information gain of the current one where all training instances that haven't been compared yet are assumed to be perfectly classified.

Chapter 6

Imbalanced Learning

Most classification tasks assume that classes are equally represented withing the training set. In reality, it's very common to have a majority (negative) class and a minority (positive) class, of which the latter contains only a small fraction of the training data, since it represents a set of more interesting events/objects. In order to handle imbalanced data, we can follow these approaches:

- **Balance the training set**: the training set is modified, using either under- or oversampling.
- **Balance at the algorithm level**: the algorithm is modified, adjusting the weight associated to the classes, using a different decision threshold, or using specific algorithms that perform well on imbalanced data.
- **Switch to Anomaly Detection**.

6.1 Balancing the Training Set

6.1.1 Undersampling

Undersampling is done on the majority class to reduce its size and make it comparable to that of the minority class. The simplest way is using **random undersampling**, with or without replacement, simply randomly selecting elements from the majority class until a set of the desired size is obtained.

Another technique is **Tomek Links**: it selects pairs of examples (a, b) that respect the following properties:

- a is the nearest neighbor of b;
- b is the nearest neighbor of a;

- a and b belong to different classes.

From this pair, the element belonging to the majority class is removed from the dataset. This way, all samples from the majority class that are very close to those in the minority class (i.e., they're harder to distinguish) can be excluded from the dataset.

Edited Nearest Neighbor works as follows: for each example, it finds its k-nearest neighbors, and checks the predicted class by that neighborhood; if this predicted class does not match the class of the example, it is removed along with the neighborhood. At the end, all examples that were too close to instances of the opposite class will have been removed from the dataset.

Condensed Nearest Neighbor performs a smart undersampling, constructing the subset of records which are able to correctly classify the original data using k-NN (typically, k is set to 1). The algorithm operates using these steps:

1. A random example is extracted and added to `STORE`. All other records are added to `GRABBAG`.

2. A loop iterates over all other samples in the dataset; for each example, it is classified via k-NN using the contents of `STORE`, and, if the prediction does not match the class label, it is moved to `STORE`. The loop continues until either `GRABBAG` becomes empty or a whole pass over it is done without no transfers to `STORE`.

3. Return `STORE`.

Finally, undersampling can be done by first performing centroid-based clustering on the data representing the majority class, and then using only the centroids instead of the entire data.

6.1.2 Oversampling

Oversampling the minority class increases the amount of examples belonging to it. Oversampling can also be done with random sampling, with or without replacement, as seen for undersampling.

SMOTE (Synthetic Minority Oversampling TEchnique) oversamples the minority class by adding data points via interpolation. It operates in the feature space, introducing synthetic examples along the segments that connect each sample with any/all of its minority class k nearest neighbors. Depending on the amount of oversampling needed, the value of k can be changed (by default, $k = 4$). In practice, interpolation is done by taking the difference between the current sample and one of its neighbors; this difference is multiplied by a random number between 0 and 1, and the result is

added to the values of the current sample. The final record represents a point that is somewhere along the segment that connects the sample and the neighbor.

An alternative to SMOTE is **ADASYN** (ADAptive SYNthetic). This algorithm operates as follows:

1. The ratio of min to maj class examples is calculated as $d = \#min/\#maj$.
2. The total number of synthetic minority class examples as $G = (\#maj - \#min)/\beta$, where β is the desired ratio of minority.
3. The k-NN of each minority sample is found, and the ratio of majority class examples is found for that neighborhood as $r_i = \#maj_i/k$; this ratio is then normalized by dividing it by the sum of all the r_i.
4. The number of synthetic samples to generate for each neighborhood is calculated as $G_i = G_{r_i}$.
5. G_i samples are generated for each neighborhood, taking two minority samples (x_i, y_i) within the neighborhood, and interpolating among them (as SMOTE does).

6.2 Balancing at the Algorithm Level

Another way to handle unbalanced data is to modify the behaviour of the training algorithm to take into account the disparity between the classes. There's two main ways this can be done: using class weights, or adjusting the decision threshold.

The overall performances of a classifier can be summarized using a confusion matrix. A corresponding **cost matrix** can be constructed, associating a different cost (weight) to each "event".

	pred. +	pred. -
actual +	f_{++}	f_{+-}
actual -	f_{-+}	f_{--}

$C(i\|j)$	pred. +	pred. -
actual +	$C(+\|+)$	$C(-\|+)$
actual -	$C(+\|-)$	$C(-\|-)$

Figure 6.1: Confusion matrix on the left, cost matrix on the right.

The objective of the training algorithm is to minimize the total cost given by:

$$\sum_X C_X * freq_X$$

There's some classifiers, called **Meta-Cost Sensitive Classifiers**, that calculate the risk of classifying a certain instance x as class i:

$$R(i|x) = \sum_j P(j|x)C(i,j)$$

Many classifiers predict what class an instance belongs to by computing scores that represent the probability that the given instance belongs to a class. The score is usually the probability that the instance belongs to the positive/minority class. By default, if the score is greater than 50%, the instance is predicted as positive, otherwise it is predicted as negative. This schema can be generalized: if the score is greater than some threshold *THR* the instance is classified as positive, otherwise it is negative. For each possible threshold, we will get a different set of predictions, and all associated metrics change as well (accuracy, precision, recall, etc.).

To monitor how changing this threshold influences the behaviour of the model, it can be useful to study the **Receiving Operating Curve** of the model, which plots the True Positive Rate against the False Positive Rate for different values of the threshold.

Chapter 7

Dimensionality Reduction

Dimensionality reduction is the process of reducing the number of variables in the dataset, obtaining a set of **principal variables**. Approaches for dimensionality reduction can be divided into feature selection and feature projection.

7.1 Feature Selection

Feature selection extracts a subset of the variables via different strategies:

- **Filter strategy**: the relevance of each feature is evaluated using some appropriate measure (e.g., Information Gain), removing the ones whose corresponding value falls below a given threshold. Commonly, a variance threshold is used. By default, it removes all zero-variance features, meaning the ones that have a constant value. Another technique is Univariate Feature Selection, which selects the best features based on univariate statistical tests. An example of statistical test is the ANOVA F-Value between dependent and independent variables:

$$F\text{-}value = \frac{\sum_{i=1}^{K} n_i(\bar{Y}_i - \bar{Y})^2/(K-1)}{\sum_{i=1}^{K}\sum_{j=1}^{n_i}(Y_{ij} - \bar{Y}_i)^2/(N-K)},$$

  where Y_i is the sample mean over the i^{th} group, n_i is the number of observations in the ith group, $\bar{Y}$ is the overall mean, Y_{ij} is the j^{th} observation in the i^{th} out of K groups, and K and N are the number of groups and the sample size, respectively. The F-value is large when the numerator is large, which is unlikely if the population means of the groups have the same value;

- **Wrapper strategy**: the performances of different subsets of features are compared using a model (e.g. a classifier of which we evaluate accuracy), and the subset that yields the best result is selected. An example is Recursive Feature

Elimination (RFE), which selects features by recursively considering smaller and smaller sets of features: first, an estimator is trained on the data, and the "importance" of each feature is returned. The least important features are pruned from the current set in a recursive fashion, until the desired number of features to select is reached.

- **Embedded strategy**: feature selection is done at the algorithm level, where parameters are modified and appropriate weights are assigned to each feature.

7.2 Feature Projection

Feature projection transforms the features via either linear or non-linear transformations. The most common approaches include:

- Principal Component Analysis (PCA) and Singular Value Decomposition (SVD);
- Multidimensional Scaling (Sammon maps, ISOMAP, t-SNE);
- Non-negative Matrix Factorization (NMF);
- Linear Discriminant Analysis (LDA);
- Autoencoders.

7.2.1 Principal Component Analysis (PCA)

The goal of **Principal Component Analysis** is to find a new set of attributes that better capture the variability of the data; the first dimension is chosen as the one that captures as much of the variability as possible, the second one as the dimension that is orthogonal (uncorrelated) to the first and also captures the most of the remaining variability in the data, and so on until the number of desired dimensions is reached. These dimensions are obtained via linear transformations of the original attributes. The steps of PCA are the following:

1. The dataset X is standardized.
2. The mean value of data is calculated across each dimension.
3. The covariance matrix of all pairs of features is calculated; given a matrix of data X, the mean of each column is removed from the column vectors to get the centered matrix C, and so the covariance matrix of the row vectors of X is calculated as $\Sigma = C^T C$.

4. The eigenvalues and eigenvectors of Σ are found, and the k eigenvectors corresponding to the largest eigenvalues are chosen.

5. The original dataset is transformed using the eigenvectors found in the previous step as the new axes.

Once the dataset has been transformed, any reconstruction will have some error.

The covariance matrix is used to store the covariance between each pair of attributes in the dataset. The diagonal of the matrix contains the variance of each attribute (since the covariance of a variable with itself is its variance). Since this matrix is symmetric, its eigenvectors are guaranteed to be orthonormal.

In order to compute the eigenvalues and eigenvectors of the covariance matrix, **Singular Value Decomposition** is often used. A matrix $A \in R^{m \times n}$ can be written as:

$$A = U \Sigma V^T ,$$

where Σ is the diagonal of A, and the columns of V are the eigenvectors of the covariance matrix, sorted from largest to smallest eigenvalues.

7.2.2 Multi-Dimensional Scaling (MDS)

Given a pairwise dissimilarity matrix, the goal of MDS is to learn a mapping of data into a lower dimensional space such that the relative distances are preserved. These methods are used to map data to very low configurations (e.g., 2 or 3 dimensions).

The key steps of MDS are the following:

1. Given a pairwise dissimilarity matrix D, and the dimensionality k, find a mapping such that $d_{ij} = \|x_i - x_j\|$ for all points in D.

2. The function
$$J(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} d_1(d_{ij}, d_2(x_i, x_j))$$
is minimized, usually via a gradient descent algorithm, solving the associated optimization problem.

Depending on the distances used, the approach will return a different result. Classic MDS uses Euclidean distances for every calculation, Metric-MDS uses metrics as distances, and Non-Metric-MDS uses ranks of distances instead of their values.

Sammon Mapping is an algorithm that, similarly to MDS, maps the data to a lower dimensional space while trying to preserve relative distances. It can be seen as

a generalization of Metric-MDS. It introduces a weighting system that normalizes the squared errors in pairwise distances using the distance in the original space:

$$J(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{d_1(d_{ij}, d_2(x_i, x_j))}{d_{ij}}$$

As a consequence, Sammon Mapping preserves the small d_{ij}, making them more "important" in the fitting procedure than larger distances. In general, Sammon mapping better preserves inter-distances for smaller dissimilarities, and proportionally squeezes the inter-distances for larger dissimilarities.

ISOmetric Feature MAPping (**ISOMAP**) is used for dimensionality reduction when the data points have a complicated, non-linear relationship to one another; it preserves the intrinsic geometry of the data. The basic ISOMAP algorithm is:

1. For each data point, the nearest neighbors are found using Euclidean distance. These neighborhood relationships are represented as a weighted graph G, where each link between nodes u and v has a weight that corresponds to the distance between u and v.

2. The geodesic distance between all pairs of points on the data manifold is calculated by computing the shortest path distances on the graph G.

3. An embedding of the data in a k-dimensional Euclidean space is constructed. This embedding is the one that best preserves the manifold geometry.

t-Distributed Stochastic Neighbor Embedding (**t-SNE**) is mostly used for data visualization. While PCA tries to find a global structure, t-SNE tries to preserve local structure, i.e., distances and neighbors are preserved by the mapping. SNE encodes high dimensional neighborhood information as a distribution.

For each input data point x_i, imagine we have a Gaussian distribution centered around it. The probability that x_i chooses another data point x_j as its neighbor is proportional with the density under this Gaussian: if the standard deviation is high this probability is low, and vice-versa. This probability is calculated as:

$$p_{j|i} = \frac{e^{-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}}}{\sum_{k \neq i} e^{-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}}}$$

The final distribution over pairs is symmetrized: $p_{ij} = 1/2N(p_{ij} + p_{ji})$. For each distribution $p_{j|i}$, we define the **perplexity**:

$$perp(p_{j|i}) = 2^{H(p_{j|i})},$$

where $H(p)$ is the entropy. If the distribution is uniform over k elements, the perplexity is k; if the perplexity is low, the sigma is small, while if the perplexity is high the sigma is large.

The objective of SNE can be defined as: given a dataset $x_1, \ldots, x_n \in R^m$, defined the distribution p_{ij}, find a good embedding $y_1, \ldots, y_n \in R^k, k < m$, such that

$$q_{j|i} = \frac{e^{-\|y_i - y_j\|^2}}{\sum_{k \neq i} e^{-\|y_i - y_k\|^2}}$$

is optimized to be the closest possible to p, minimizing the **KL-divergence**. KL-divergence measures the distance between two distributions P and Q, as:

$$C = \sum_i KL(P_i|Q_i) = \sum_i \sum_j p_{j|i} \log(\frac{p_{j|i}}{q_{j|i}})$$

This is not a metric function, since it's not symmetric. It measures the penalty for using a wrong distribution, and is usually minimized using gradient descent. This is not a convex problem, so there's no guarantee an optimum will be reached, but multiple restarts can be done attempt at finding a good enough solution.

The peculiarity of t-SNE is that for Q, instead of using a Gaussian distribution, it uses a heavy tailed distribution:

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}$$

Bibliography

[1] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson, 2018.

[2] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. *Information systems*, 25(5):345–366, 2000.

[3] Yiling Yang, Xudong Guan, and Jinyuan You. Clope: a fast and effective clustering algorithm for transactional data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 682–687, 2002.

[4] Riccardo Guidotti, Anna Monreale, Mirco Nanni, Fosca Giannotti, and Dino Pedreschi. Clustering individual transactional data for masses of users. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 195–204, 2017.

[5] Chotirat Ann Ralanamahatana, Jessica Lin, Dimitrios Gunopulos, Eamonn Keogh, Michail Vlachos, and Gautam Das. Mining time series data. *Data mining and knowledge discovery handbook*, pages 1069–1103, 2005.

[6] Lexiang Ye and Eamonn Keogh. Time series shapelets: a new primitive for data mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 947–956, 2009.