Data Mining 2 23-24

Notes

University of Pisa

M.Sc. in Data Science and Business Informatics

Contents

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 of 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: $R \leftarrow R \vee r$ (Add the rule to the rule set)
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:

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

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 $r' : A' \rightarrow y$ 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.

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 $1 \leq i_1 < i_2 < \dots < i_n \leq m$ 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 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 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(F_{k-1})
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 ones. 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:

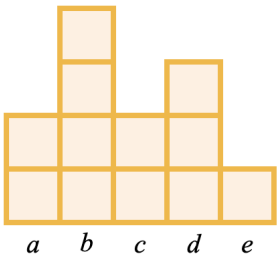
$$g(C_i, C_j) = \frac{link(C_i, C_j)}{(n_i + n_j)^{1+2f(\theta)} - n_i^{1+2f(\theta)} - n_j^{1+2f(\theta)}},$$

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 $r2$
    $r1_{new} =$ `getRepr`$(C1)$
    $r2_{new} =$ `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 be 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.

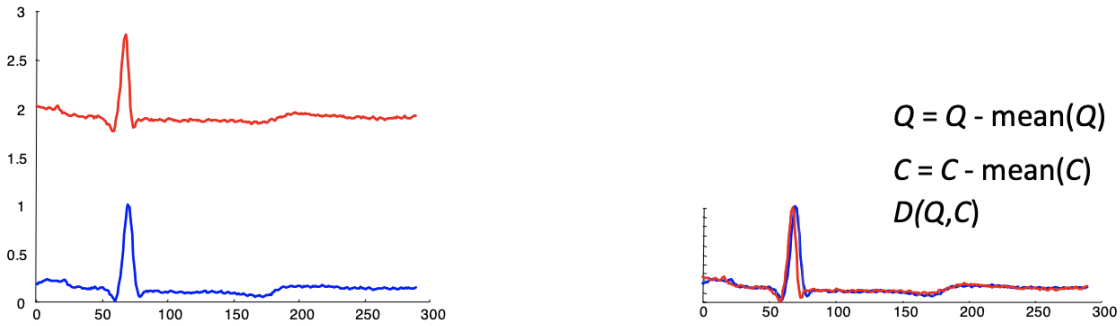


Figure 4.1: Offset translation transformation.

- **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).

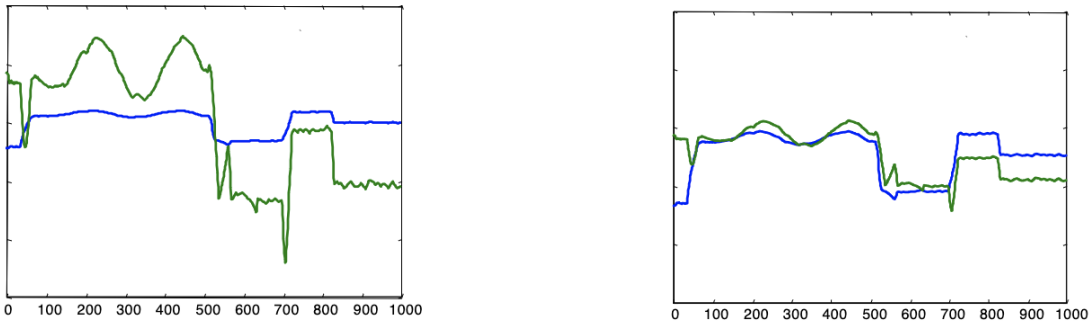


Figure 4.2: Amplitude scaling transformation.

- **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.

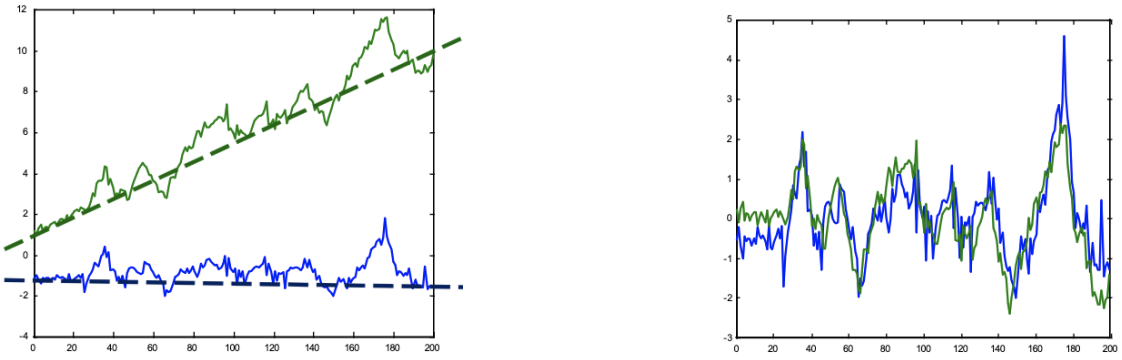


Figure 4.3: Linear trend transformation.

- **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.

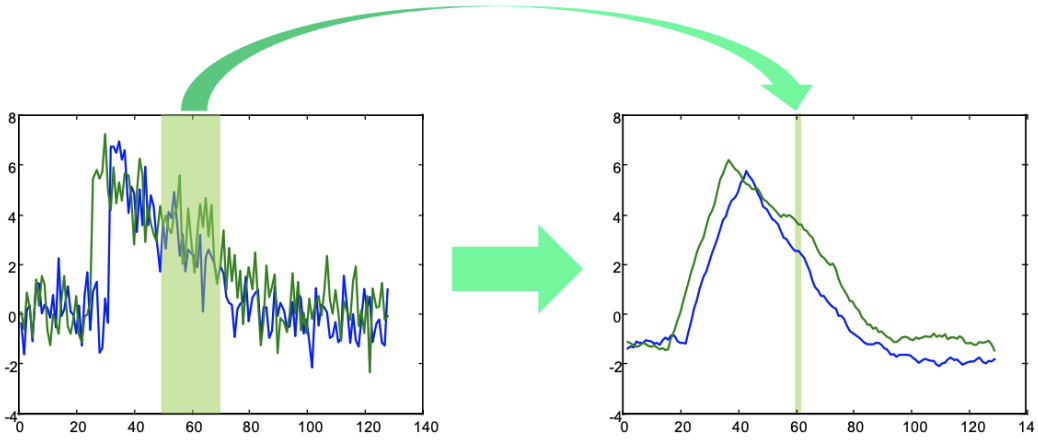


Figure 4.4: Noise transformation.

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.

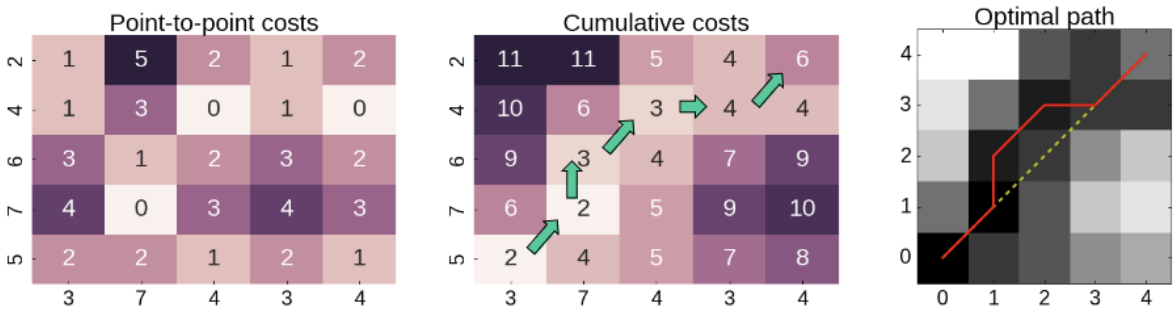


Figure 4.5: How Dynamic Time Warping is calculated.

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.

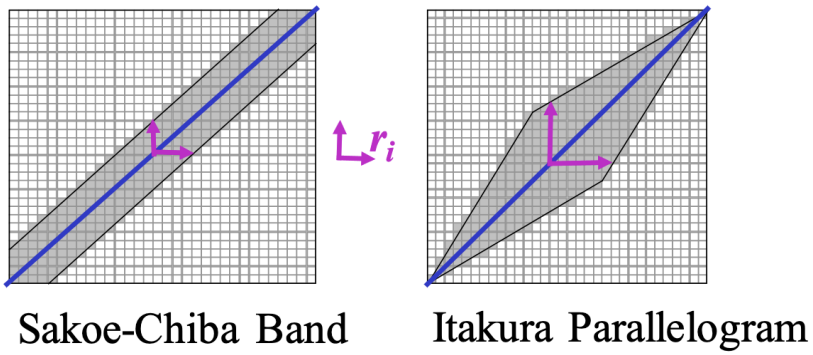


Figure 4.6: Global constraints.

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 $N = n$, the transformed representation is identical to the original series; if $N = 1$, 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 is **segmented** using a set of segments of predefined length w; 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.

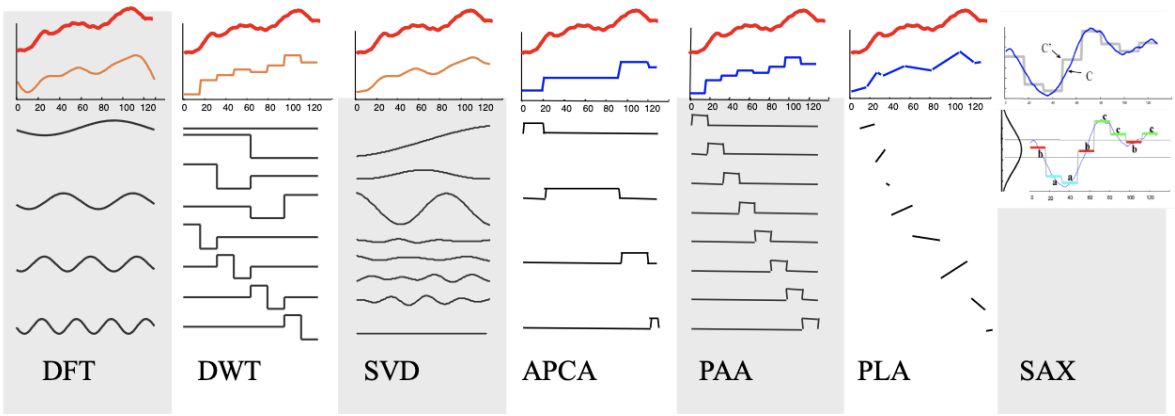


Figure 4.7: The most common time series approximations.

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

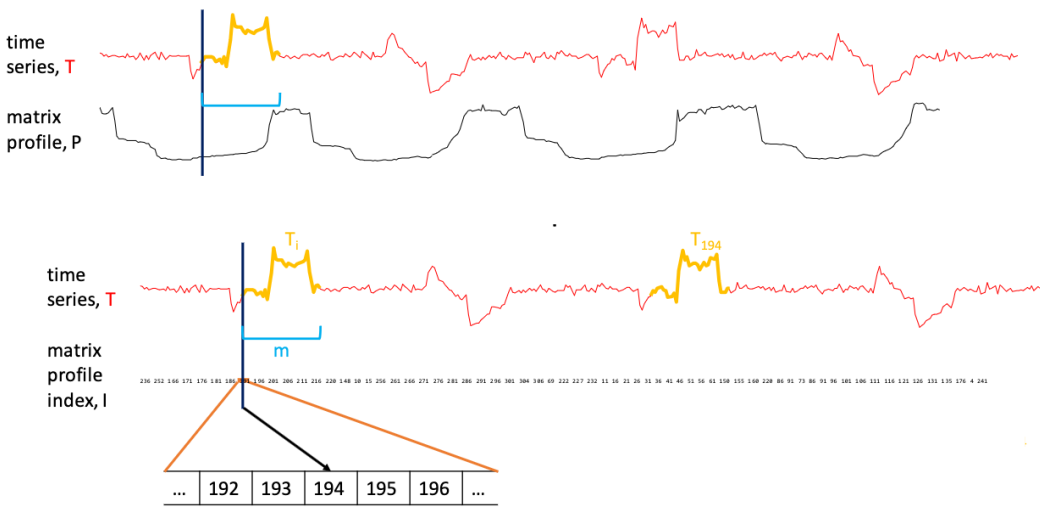


Figure 5.1: Matrix Profile (on top) and Matrix Profile Index (on the bottom).

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

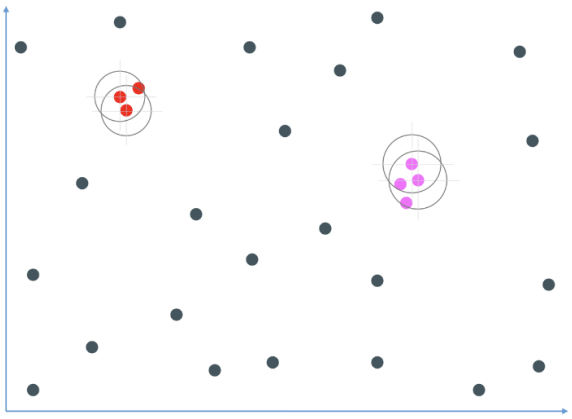


Figure 5.2: A graphical interpretation of how top k-motifs are found. The red dots are the top 1-motif, and the magenta ones are the top-2 motifs.

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.

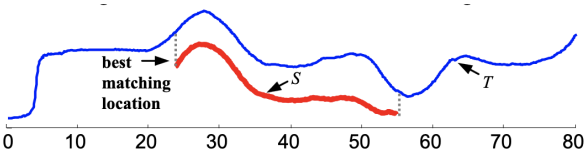


Figure 5.3: The best matching location of S over T.

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, (always with replacement, of course).

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 $\bar{Y}_i$ is the sample mean over the i^{th} group, n_i is the number of observations in the i^{th} 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:

- Random Subspace Projection;
- 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.

Random Subspace Projection is a very simple approach: given the number of attributes n in the dataset $D \in \mathbb{R}^{n \times l}$, and the number of dimensions we want to reduce to k, a matrix $M \in \mathbb{R}^{k \times n}$ is randomly generated (such that its columns have unit length), and so the transformed data is $D' = MD$. It preserves the structure of the data, and is computationally cheap.

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 subtracted from the respective 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. The matrix C can be written as:

$$C = USV^T,$$

where S is the diagonal of C, 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 goal is to find a mapping such that $d_{ij} = \|x_i - x_j\|$ for all points in D. 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, its k 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 means that u and v are nearest neighbors, and 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 high, and vice-versa. This probability is calculated as:

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

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 perplexity is set as a hyperparameter, so the sigma of each p is adjusted to match it. The final distribution over pairs is symmetrized: $p_{ij} = (p_{ij} + p_{ji})/2N$.

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, and is used to define the cost function:

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

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 Student's t-distribution with one degree of freedom, which is a heavy tailed distribution:

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

This distribution helps to model dissimilarity among distant objects.

Chapter 8

Outlier Detection

An **outlier** (anomaly) is an observation that doesn't fit the distribution of the data for normal instances, i.e., is unlikely under the distribution of the majority of instances. Outlier (anomaly) detection has the goal of finding these anomalies in datasets. Although outliers are by definition unusual, their discovery and analysis may provide critical insights on the data, in applications like fraud detection, intruder detection, ecosystem disturbances, medicine and public health, aviation safety, and so on. In all these fields, finding exceptional events or objects is often the main focus of analysis: in fraud detection we're interested in finding anomalous credit card transactions; in intrusion detection attacks can be identified by noticing weird behaviour in systems and networks.

Outliers can be caused by natural variation in values or data belonging to a minority class, or may be caused by errors. In comparison, **noise** is a random error addition to data typically caused by imprecision during measurements; it does not necessarily produce unusual values or objects, and is normally not interesting per se. Also, data objects may be outliers only for some attributes, or considering all of them. Identifying outliers in a multivariate setting can be challenging, especially when the dimensionality is high.

There are two ways in which an outlier detection method can be used. In the first one, given a dataset with both normal and anomalous instances, we are required to identify these anomalies. In the second one, we're also provided a test set of instances which we want to classify as either outliers or normal data objects. All the techniques presented in the next section can operate in the first way, and most of them (with a few exceptions) can operate in the second way.

8.1 Characteristics of Outlier Detection Methods

- **Model-based vs. Model-free**: many approaches build actual models that can be used to identify whether a test instance is anomalous or not. Most **model-based** techniques construct a model trained on the normal class and identify any data point that does not fit the model. Alternatively, a model can be trained on both normal and anomalous data, and outliers are identified as those data points which are more likely to belong to the anomalous class. Both cases don't necessarily need label data (i.e., models can be trained in an unsupervised way), since they can make assumptions about the nature of the anomalous class.

  On the other hand, **model-free** approaches don't explicitly construct a model that characterizes the distribution of the anomalous and/or normal class. Instead, they directly identify instances as outliers without the need for any training, using a simpler calculation.

  If the ground truth of anomalies is available, we could define a classification problem to find outliers and solve it using popular machine learning models (ensemble models, support vector machines, deep neural networks,...). The dataset would often be unbalanced, so ad-hoc formulations should be adopted.

- **Global vs. Local**: some approaches consider the global context, building a model or computing an output considering the entire dataset, or by only considering the local context of each data instance; specifically, an outlier detection approach is local if its output on a given instance does not change if instances outside the local neighborhood of that point are modified or removed.

  Some approaches may be both, either because the reference set for an instance varies automatically during execution, or because it can be set by an hyperparameter.

- **Label vs. Score**: different approaches produce outputs in different formats. Some produce a binary **anomaly label**, and each object is identified as either an outlier or a normal instance. Others produce an **anomaly score**, indicating how strongly an instance is likely to be an anomaly. After calculating the score of a set of instances, a ranking can be obtained to identify the top-most scoring anomalies. Optionally, a cut-off threshold can be set so that all instances with a score above that are definitely classified as outliers, while the ones below are normal instances.

A very simple visual technique to find outliers is using boxplots and scatterplots. However, they do not return explicit values, and are highly subjective, since they cannot

find points that are outliers when considering all dimensions. An alternative is calculating the **Histogram-based Outlier Score**, which is done by building histograms for each feature. These histograms are then normalized to the $[0, 1]$ range, and the **HBOS** for each record p is computed as a product of the inverse of the estimated density (according to the histograms):

$$HBOS(p) = \sum_{i=0}^{d} \log(\frac{1}{hist_i(p)})$$

This approach assumes that features are independent.

8.2 Statistical Approaches

Statistical approaches use probability distributions to model the normal class, associating a probability value to each data instance indicating how likely it is for that instance to be generated from that distribution. Outliers will be all instances with an associated low probability.

In practice, it applies a statistical test that depends on data distribution, parameters of the distribution (mean, variance, etc.), and the number of expected outliers (a confidence limit). One of the main issues in these approaches lies in identifying the distribution of a dataset (which is not known a priori), considering that often datasets have a mixture of distributions. These approaches are also influenced by the number of attributes in the data.

Grubb's Test detects outliers in univariate data. It assumes a normal distribution. One outlier is detected at a time and removed from the dataset, using a statistical test: the null hypothesis H_0 is *"there is no outlier in the dataset"*, while the alternative hypothesis H_A is *"there is at least one outlier in the dataset"*. The Grubb's test statistic is:

$$G = \frac{\max |X - \bar{X}|}{s}$$

with $\bar{X}$ being the sample mean and s the sample standard deviation of the data. The null hypothesis is rejected at significance level α if:

$$G > \frac{N-1}{\sqrt{N}} \sqrt{\frac{t^2_{\alpha/N,N-2}}{N-2+t^2_{\alpha/N,N-2}}}$$

where $t_{alpha/N,N-2}$ is the upper critical value of the t-distribution with $N-2$ degrees of freedom and a significance level of α/N. This is a one-sided test, but it can be defined as a two-sided test by using $\alpha/2N$.

Another approach is based on **likelihood**. Assume the dataset contains samples from a mixture of two probability distribution, M (majority) and A (anomalous), such that $D = (1 - \lambda)M + \lambda A$. Initially, all points are assumed to belong to M. Let $L_t(D)$ be the log likelihood of the dataset at time t; for each data point x_t that belongs to M, it is moved to A, and the following check is done:

- Calculate $L_{t+1}(D)$, the new log-likelihood of the dataset.
- Compute $\Delta = L_t(D) - L_{t+1}(D)$.
- If $\Delta > c$ (c is a user-defined threshold), then x_t is declared an outlier and permanently moved to A. Otherwise, it is moved back to M.

Pros statistical approaches have a firm mathematical foundation, can be very efficient, and produce good results if the data distribution is known.

Cons in many cases, the data distribution is unknown, and for high-dimensional data estimating it may be difficult, therefore the assumptions done by these approaches may be completely incorrect. Also, these are global approaches, and anomalies distort the parameters of the distribution.

8.3 Deviation-based Approaches

Given a set of data points, outliers are those which do not fit to the general characteristics of that set, i.e, the variance of the set is minimized if those outliers are removed from it. This idea is at the basis of deviation-based approaches, which assume that outliers are the outermost points.

An example of such approach is the one proposed in [1], where the problem is defined as follows. Given:

- A set of items I;
- A dissimilarity function D;
- A cardinality function C, such that $I_1 \subset I_2 \implies C(I_1) < C(I_2)$ for all $I_1, I_2 \subseteq I$;

Define for each $I_j \subset I$ the **smoothing factor**:

$$SF(I_j) = C(I - I_j) * (D(I) - D(I - I_j))$$

Then, we say that $I_x \subset I$ is an **exception set** with respect to D and C if

$$SF(I_x) \geq SF(I_j) \forall I_j \subset I$$

Intuitively, the exception set contains all instances that contribute the most to the dissimilarity of the itemset I with the least number of elements; in other words, its outliers. The smoothing factor indicates how much the dissimilarity can be reduced by removing the subset I_j from I.

This approaches is similar to statistical-based ones, although it does not depend from a chosen distribution. The naive solution is in $O(2n)$ for n data objects, and can be applicable to any data type. It often applies an heuristic such as random sampling or best first search. It outputs a labeling.

8.4 Depth-based Approaches

Depth-based approaches search for outliers at the border of the data space. Instances are first organized in convex hull layers, and outliers are points that lie in the outermost layers, while normal points are those which lie in the center of the data space, therefore in the innermost layers.

The figure below shows an example dataset.

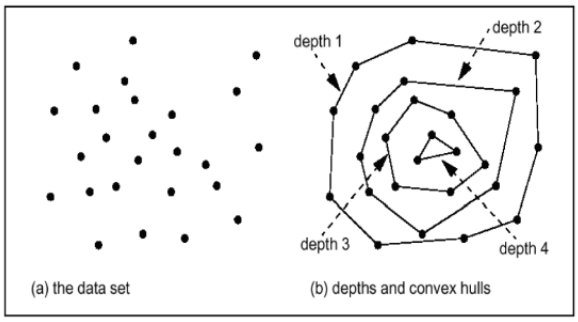


Figure 8.1: Example of depth-based outlier detection.

The points on the convex hull of the full dataset have depth 1. The points in the convex hull of the dataset obtained by removing the points of depth 1 have depth 2, and so on. Once a certain number of convex hulls have been calculated, all points with a depth smaller than some hyperparameter k are reported as outliers. By default it produces a label, but it can be modified so that it returns the depth of each point instead of a 0/1 classification. This algorithm, however, is typically only efficient for two- or three-dimensional spaces.

Another depth-based algorithm is **Elliptic Envelope**. It finds the center of the data samples and draws an ellipsoid around it, creating an imaginary elliptical area around the dataset. Instances that fall inside the envelope are considered normal, and

anything outside is an outlier. This algorithm works best for data with a Gaussian distribution.

8.5 Proximity-based Approaches

8.5.1 Distance-based

A simple way to define the anomaly score of an instance x is to use its distance to the k^{th} nearest neighbor, $dist(x, k)$. If an instance has a lot of close points (so it belongs to the group of normal data), this distance will be small, otherwise the point will be far away from the rest and have a very high distance compared to the rest. Outlier detection algorithms which use k-NN distances to score data follow two approaches: either they have a simple **nested-loop** structure, where a sequential scan computes the k-NNs of each instance, or they are **partition-based**, i.e., they first partition data into micro clusters, and information is aggregated for each partition. This way, micro clusters that cannot qualify when searching for the k-NNs of a point can be pruned.

Another common algorithm uses two parameters, a radius ε and a percentage π, such that a point p is considered an outlier if at most π percent of all other points have a distance to p less than ε (p is close to very few points). Formally, the set of outliers is computed as:

$$OutlierSet(\varepsilon, \pi) = \left\{ p | \frac{\#\{q \in D | dist(p, q) < \epsilon\}}{\#D} \leq \pi \right\}$$

Finally, outliers can be identified using the **in-degree number**. Given a dataset, the k-NN graph is constructed: each vertex is a data point, and each directed edge between two vertices p and q means that q is one of p's k-nearest neighbors. The in-degree of a point is calculated as the number of reverse k-NNs (Rk-NN), i.e., the number of points who have it as a neighbor. If a point has an in-degree lower than some user-defined threshold, it is declared an outlier. This algorithm outputs an outlier label.

Pros Distance-based approaches are very simple to implement. They do not require any training, since they're model-free.

Cons Since the outcome depends on distances, they tend to be slow, and they're not as accurate in high-dimensional spaces. Also, they're sensitive to the chosen hyperparameters and to variations in density.

8.5.2 Density-based

The density around an instance can be defined as $n/V(d)$, where n is the number of instances within a given distance, and $V(d)$ is the volume of the neighborhood. Since $V(d)$ is constant for a given d, the density is often only represented using n after fixing a value of d. Outliers can be defined as points that are in regions of low density, and their anomaly score as the inverse of the density around them (or the inverse of distance to the k^{th} neighbor, or the inverse of the average distance to k neighbors). However, if the dataset has regions of different density, this approach can present problems (as seen when discussing DBSCAN).

To fix this problem, the **average relative density** can be used instead. Given a point x and a number of neighbors k, it is calculated as:

$$avg.\ relative\ density(x,k) = \frac{density(x,k)}{\sum_{y \in N(x,k)} density(y,k)/|N(x,k)|}$$

where $N(x,k)$ is the k-neighborhood of x. This way, the density around a point depends on that of its nearest neighbors. If the dataset has areas with differing densities, points in lower density areas will not be classified as outliers. If a point is so faraway that its normal density is low, this relative density will also be low. However, if a point is an outlier only for a very dense cluster, it may not be identified (such as the situation in the figure below).

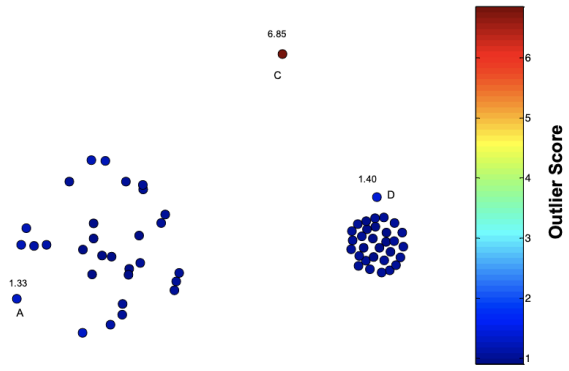


Figure 8.2: An example in which average relative density fails. Notice how point D is an outlier for the denser cluster, but it is treated as a normal point.

Local Outlier Factor (LOF)

An alternative is using the **Local Outlier Factor** (**LOF**). First, for each pair of points p and o in the dataset, their **reachability distance** is calculated, identical to that used

in OPTICS:

$$\textit{reachability-dist}_k(p, o) = \max\{\textit{k-dist}(o), \textit{dist}(p, o)\}$$

then, each point is associated with its **local reachability distance**:

$$LRD_k(p) = 1/\frac{\sum_{o \in N_k(p)} \textit{reach.-dist}_k(p, o)}{k}$$

Finally, the LOF is calculated as:

$$LOF_k(p) = \frac{\sum_{o \in N_{k(p)}} LRD_{k(o)}/LRD_{k(p)}}{k}$$

If a point's LOF is close to 1, then the point is inside a cluster, and if it is much greater than 1, it is an outlier. The result is influenced by the chosen k, but an increase in k does not necessarily mean an increase in LOF.

Connectivity-based Outlier Factor (COF)

Sometimes, in regions of low density, LOF may fail to detect outliers, unless we use a small k. However, using a too small k may include outliers as normal points. A solution is using the **Connectivity-based Outlier Factor**. It uses **chaining distance** to calculate a point's nearest neighbor. The average chaining distance is calculated as:

$$\textit{ac-dist}_{N_{k(p)}}(p) = \sum_{i=1}^{k-1} \frac{2(r-i)}{r(r-1)} CDS_i$$

where k is the number of neighbors in p's k-neighborhood, and CDS_i is the cost description sequence of removing the i^{th} neighbor. The chaining distance for a point can be seen as the minimum of the total sum of the distances linking all neighbors. In practice, it is calculated with a graph-like structure.
The COF is then calculated as:

$$COF_k(p) = \frac{|N_k(p)|\textit{ac-dist}_{N_{k(p)}}}{\sum_{o \in N_{k(p)}} \textit{ac-dist}_{N_{k(o)}}(o)}$$

INFLuenced Outlierness (INFLO)

Another issue that the previous methods fail to address is if clusters of different densities are not well separated. The idea behind INFLO is to take symmetric neighborhood relationships into account. In other words, it considers both nearest neighbors as well as the reverse nearest neighbors (that is, the points having p as a neighbor), defining the **influence space** of a point p (indicated with $kIS(p)$):

$$kIS(p) = \textit{k-NN}(p) \cup \textit{Rk-NN}(p)$$

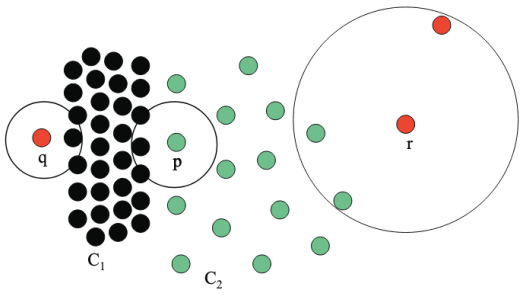


Figure 8.3: In this example, points q and r have a lower LOF compared to p, despite it not being an outlier for the sparse cluster on the right.

The density of each ponint is simply calculated as the inverse of its k-distance. The INFLO is then calculated as:

$$INFLO_k(p) = \frac{\sum_{o \in kIS(p)} \frac{dens(o)}{\#kIS(p)}}{dens(p)}$$

Similarly to LOF, if it is close to 1 the point is in a cluster, if instead it is much greater than 1, the point is an outlier.

Pros Density-based approaches are, like distance based ones, very simple to implement.

Cons They're computationally expensive since there is a need to find the distance between all points in the dataset, and densities are less meaningful as the dimensionality of the data increases. They are sensitive to the chosen hyperparameters.

8.6 Clustering-based Approaches

Cluster-based outliers are points that do not fit strongly into any cluster. For prototype-based clusters, it means they are far away from it; for density-based clusters, it means the object has a too low density around it; for graph-based clusters, it means the point in not well connected.

Some clustering algorithms are already capable of separating points belonging to clusters from outliers (DBSCAN, OPTICS). An idea would be to simply execute those algorithms feeding the dataset and choosing the appropriate hyperparameters, and analyze the set of "noise" points returned by them. However, these are first and foremost

clustering algorithms: they are not optimized to find outliers and sets of abnormal data objects may be recognized as a cluster instead of outliers.

Just as we have seen for density-based outliers, we can use **Cluster-Based Local Outlier Factor** (**CBLOF**). To calculate it, first a clustering algorithm is used to find clusters in the dataset. Then, each cluster is classified as either a **small cluster** (**SC**), or **large cluster** (**LC**), using two parameters, α and β. Specifically, given the clustering $C = \{C_1, C_2, \ldots, C_k\}$ such that $|C_1| \geq |C_2| \geq \cdots \geq |C_k|$, a boundary b is found if one of the following inequalities hold:

$$|C_1| + |C_2| + \cdots + |C_b| \geq |D| * \alpha$$
$$|C_b|/|C_{b+1}| \geq \beta$$

This means that all clusters up until the b^{th} contain $\alpha\%$ of the data points (first formula), or that the smallest of the large clusters must be at least β times larger than the larger of the small clusters (second formula).

The CBLOF is calculated depending on the size of the cluster the point belongs to, as well as the distance to the nearest large cluster. If the point is in a LC, the outlier score is the product between the size of the cluster and the distance with the cluster center. If it is in a SC, the score is the product between the size of the cluster and the distance to the center of the closest LC.

$$CBLOF(p) = \begin{cases} |C_i| * \min(d(p, C_j)), p \in C_i \wedge C_j \in LC & \text{if } C_i \in SC \\ |C_i| * d(p, C_i) & \text{if } C_i \in LC \end{cases}$$

Pros Cluster-based approaches are simple to implement and a variety of different algorithms exist.

Cons A main issue is that the presence of outliers itself will distort the clustering. Also, it may be difficult choosing the appropriate hyperparameters and number of clusters when we don't have any knowledge about the structure of the data.

8.7 High-dimensional Approaches

All the approaches seen until now have one big problem in common: they are affected by the curse of dimensionality, and so they don't work well in high-dimensional spaces. An alternative to considering distances is instead considering angles, with **angle-based outlier detection**. The idea is to compare the angles between pairs of distance vectors that start from the same point. If a point is inside a cluster of data, most angles forming

between distance vectors connecting that point and all other points will differ widely. For outliers, instead, all those angles will be similar to each other. For each point p, the

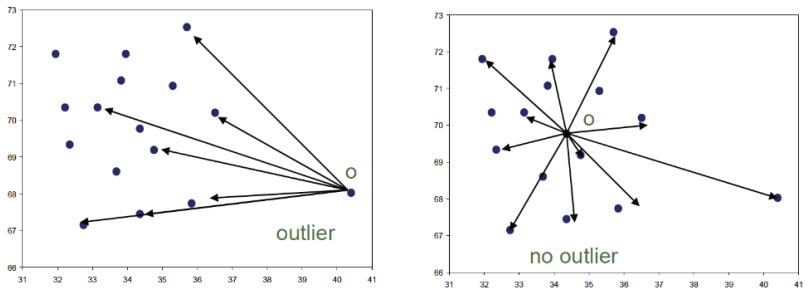


Figure 8.4: Intuition behind angle-based outlier detection.

angle between any two other instances is calculated. The spectrum of all these angles is computed, and depending on its broadness, a score can be assigned to p: a small spectrum assigns an high outlier score, a large spectrum assign a small outlier score. Specifically, the **angle-based outlier degree** (**ABOD**) is assigned to p, calculated as the variance of the angle spectrum weighted by the corresponding distances:

$$ABOD(p) = \underset{x,y \in D}{\text{Var}} \left(\frac{< \vec{xp}, \vec{yp} >}{\|\vec{xp}\|^2 \cdot \|\vec{yp}\|^2} \right)$$

Note that the lower the ABOD, the more likely a point is to be an outlier.

The naive algorithm that calculates these degrees has a cost of $O(n^3)$, but it can be approximated by using random sampling to mine the top-n outliers; ABOD is calculated only on the pairs of points in the sample, finding a lower bound of the real ABOD, and filtering out points that have a high ABOD lower bound. The real ABOD is only calculated for a small number of points.

A second approach is called **grid-based subspace outlier detection**. The data space of k dimensions is partitioned into an equi-depth grid of Φ cells. The **sparsity coefficient** of a grid cell C is:

$$S(C) = \frac{count(C) - n(1/\Phi)^k}{\sqrt{n(1/\Phi)^k(1 - (1/\Phi)^k)}}$$

where $count(C)$ is the number of data objects in C. If $S(C) < 0$, then $count(C)$ is lower than expected: all points in those cells are labeled as outliers. This algorithm has a cost of $O(\Phi k)$.

This is a very coarse model: all the points within a cell with low sparsity are automatically outliers, without doing any further analysis on those points, so the quality of the result depends on the grid resolution and position.

8.8 Ensemble-based Approaches

Ensemble-based means that a group of methods is used on the same dataset, and the final result will be a combination of their singular outcomes. For outlier detection, **feature bagging** (**FeaBag**) is used: a set of different outlier detection methods is selected, and each of them is applied on a random set of features selected from the original feature space. Each method will identify different outliers, assigning them outlier scores. The average of those scores is then returned for each point.

An extension of histogram-based outlier score is **lightweight on-line detector of anomalies (LODA)**. It is especially useful in real-time scenarios where a big amount of records must be processed. LODA approximates the joint probability using a collection of one-dimensional histograms, each constructed on an input space projected onto a randomly generated vector. Even thoug one-dimensional histograms are weak outlier detection methods by themselves, their collection yields good results.

8.9 Model-based Approaches

Isolation forests are a model specialized for outlier detection. First, a set of trees is initialized. Then, each tree receives a different sample of the data. The construction of the tree is done by repeating the following steps until the entire structure is complete, isolating each point in a leaf:

1. Pick a random dimension;
2. Pick a random value on that dimension;
3. Draw a separating hyperplane at that value, and split the data in the two sides the hyperplanes separates them.

Outliers will tend to be separated in few steps (so the leaves containing them will have low depth), while normal points will be separated after a long time (so they will be found in the deeper leaves).
Given a dataset of size n, the outlier score of a point x is:

$$s(x, n) = 2^{\frac{-\mathrm{E}[h(x)]}{c(n)}}$$

where $h(x)$ represents the path length from the root to x, and $c(n)$ is the average $h(x)$ given n, calculated as:

$$c(n) = \begin{cases} 1 & n = 2 \\ 2 * H(n-1) - \dfrac{2(n-1)}{n} & n > 2 \\ 0 & \text{else} \end{cases}$$

$H(i) = \ln(i) + \gamma, \gamma \approx 0.57$ is the i^{th} harmonic number. Isolation forests are com-

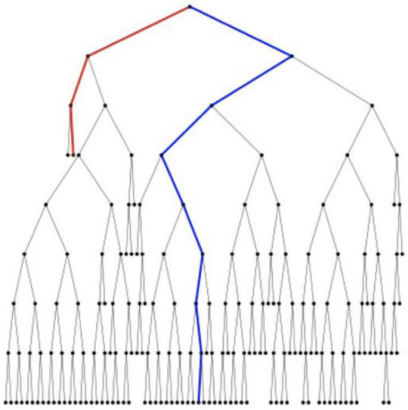


Figure 8.5: An example of tree: in red the path from the root to an anomaly, in blue the path to a normal point.

putationally efficient, paralellizable, and can handle high dimensional data. However, inconsistent scoring can be observed since the space is always split across one dimension at a time. The **extended isolation forest** model solves this problem: instead of choosing a dimension, each tree randomly selects both a normal vector and an intercept describing an angled hyperplane.

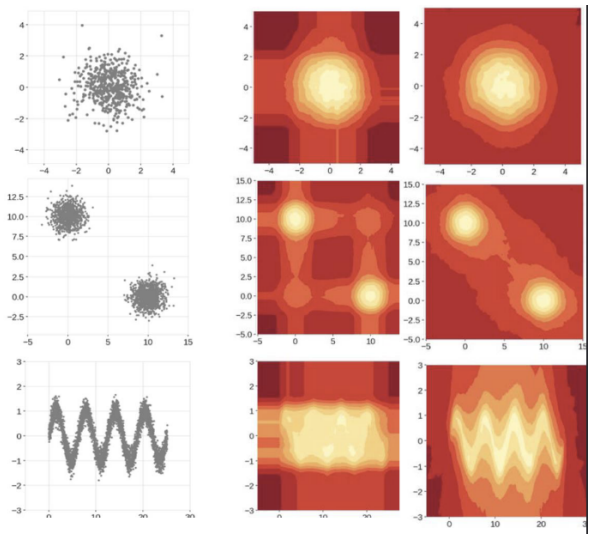


Figure 8.6: Isolation forests and Extended isolation forests, compared.

Chapter 9

Logistic Regression

Logistic regression is the task of finding a model that represents the log-odds of an event, used for binary classification problems. The function used by the model is called **logistic function**, which converts the log-odds of a point to the probability of it being labeled 0 or 1. The model is fitted to the data by maximum likelihood estimation.

The idea is to model the probability using a linear function, written as:

$$f(x_i) = \beta_1 x_i + \beta_0$$

where β_0 and β_1 are regression coefficients. Since the value returned by this function for each x_i is its corresponding log-odds, we can write the following equation:

$$\ln\left(\frac{p}{1-p}\right) = \beta_1 x_i + \beta_0$$
$$\Longleftrightarrow$$
$$\frac{p}{1-p} = e^{\beta_1 x_i + \beta_0}$$
$$\Longleftrightarrow$$
$$p = e^{\beta_1 x_i + \beta_0} - pe^{\beta_1 x_i + \beta_0}$$
$$\Longleftrightarrow$$
$$p + pe^{\beta_1 x_i + \beta_0} = e^{\beta_1 x_i + \beta_0}$$
$$\Longleftrightarrow$$
$$p(1 + e^{\beta_1 x_i + \beta_0}) = e^{\beta_1 x_i + \beta_0}$$
$$\Longleftrightarrow$$
$$\boxed{p = \frac{e^{\beta_1 x_i + \beta_0}}{1 + e^{\beta_1 x_i + \beta_0}} = \frac{1}{1 + e^{-(\beta_1 x_i + \beta_0)}}}$$

So, for a set of given values for the model's parameters, its **likelihood** is calculated as:

$$\mathcal{L}_n(\beta_0, \beta_1; x) = \prod_{i:y_i=1} f(x_i) * \prod_{j:y_j=0} 1 - f(x_j)$$

i.e., it's the product of the likelihoods of each data point. In practice, the **log-likelihood** is used instead:

$$l(\beta_0, \beta_1; x) = \sum_{i:y_i=1} \ln(f(x_i)) + \sum_{j:y_j=0} \ln(1 - f(x_j))$$

The likelihood is calculated for different parameters, gradually increasing it until it finds the optimal fit.

Chapter 10

Support Vector Machines

Support Vector Machines (SVMs) are a type of model used for both classification and regression. They used to be the most popular approach for supervised learning when there's little to no domain knowledge for the data, but they've been mostly replaced by neural networks and random forests.

10.1 Hard Margin SVM

Assume a classification task. Given a training set $TR =< x_i, d_i >$, $i = 1 \dots N$, we want to find an hyperplane of equation $w^T x + b = 0$ to separate the examples; specifically, we want:

$$w^T x_i + b \geq 0 \text{ for } d_i = +1$$
$$w^T x_i + b < 0 \text{ for } d_i = -1$$

$g(x) = w^T x + b$ is called the **discriminant function**, and $h(x) = sign(g(x))$ is the hypothesis. Note that here the bias is referred to as b instead of w_0. To find the hyperplane that separates all the points, instead of simply minimizing the empirical risk, the SVM minimizes the expected generalization error by finding the separator that is farthest away from all the examples in the dataset. It also establishes a margin (ρ) around it, whose width is exactly twice the distance between the hyperplane and the closest data point(s). The optimal hyperplane is the one that maximizes this margin ρ: $w_o^T x + b_o = 0$, where $\rho = 2/\|w_o\|$.

The two terms w and b can be rescaled so that the closest points to the separating hyperplane satisfy $\|g(x)\| = 1$, so we can write:

$$w^T x_i + b \geq 1 \text{ for } d_i = +1$$
$$w^T x_i + b \leq -1 \text{ for } d_i = -1\,,$$

or, in a compact form,

$$d_i(w^T x_i + b) \geq 1\,.$$

Any x_i that satisfies this equation is called a **support vector**, and is referred to as $x^{(s)}$. Let's denote the distance between the optimal hyperplane and a point x as r, such that $x = x_p + r\frac{w_o}{\|w_o\|}$ (where x_p is a point on the hyperplane). Evaluating $g(x)$ we obtain:

$$\begin{aligned} g(x) = g(x_p + r\frac{w_o}{\|w_o\|}) = \\ = w_o^T x_p + b_o + w_o^T r\frac{w_o}{\|w_o\|} = \\ = g(x_p) + w_o^T r\frac{w_o}{\|w_o\|} = \\ = 0 + r\frac{\|w_o\|^2}{\|w_o\|} = r\|w_o\|\,, \end{aligned}$$

thus, $r = \dfrac{g(x)}{\|w_o\|}$.
Consider the distance between the hyperplane and a positive support vector $x^{(s)}$, then r is calculated as:

$$r = \frac{g(x^{(s)})}{\|w_o\|} = \frac{1}{\|w_o\|} = \frac{\rho}{2}\,,$$

therefore, $\rho = \dfrac{2}{\|w_o\|}$.

10.1.1 Primal Problem

The optimum hyperplane will maximize ρ and minimize $\|w\|$. One approach to find it is to perform a gradient descent to find the w and b that minimize/maximizes them, but there's another approach, that is solving a **quadratic optimization problem**.

Quadratic optimization problem (primal form)

Given the training examples $TR =< x_i, d_i >$, find the optimal values of w and b which minimize

$$\Psi(w) = \frac{1}{2} w^T w$$

satisfying the constraints

$$d_i(w^T x_i + b) \geq 1\,.$$

The objective function $\Psi(w)$ is quadratic and convex in w. The constraints are linear in w, and solving the problem scales with the size of the input space m.

To solve this problem, the **Lagrangian multipliers method** is used. The Lagrangian function corresponding to the quadratic optimization problem is constructed:

$$J(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i (d_i(w^T x_i + b) - 1)\,,$$

where $\alpha_i \geq 0$ are the **Lagrangian multipliers**. Each term in the sum corresponds to a constraint of the primal problem; J must be minimized with respect to w and b and maximized with respect to α. The solution will correspond to a saddle point of J.
If we minimize J with respect to w, then:

$$\frac{\partial J}{\partial w} = \frac{2}{2} w - \sum_{i=1}^{N} \alpha_i (d_i(1 * x_i + 0) + 0) =$$

$$= w - \sum_{i=1}^{N} \alpha_i (d_i(x_i)) = 0$$

so

$$w = \sum_{i=1}^{N} \alpha_i (d_i(x_i))\,.$$

Thus the optimal hyperplane is expressed as:

$$g(x) = w_o^T x + b_o = 0$$

$$\Longleftrightarrow$$

$$\sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x + b_o = 0\,.$$

If instead we minimize it with respect to b:

$$\frac{\partial J}{\partial b} = 0 - \sum_{i=1}^{N} \alpha_i d_i = 0\,.$$

These can be substituted in J to study the dual form of the problem. To calculate b_o, we know that given a positive support vector $x^{(s)}$, it holds that:

$$w_o^T x^{(s)} + b_o = 1\,,$$

therefore

$$b_o = 1 - \sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x^{(s)}\,.$$

From the **Kuhn-Tucker Conditions**, it follows that

$$\alpha_i(d_i(w^T x_i + b) - 1) = 0\,, \forall i = 1, \ldots, N$$

in the saddle point of J. If $\alpha_i > 0$, then $(d_i(w^T x_i + b) = 1$, and x_i is a support vector. If x_i is not a support vector, then $\alpha_i = 0$. Hence we can restrict the computation to $N_s : w_o = \sum_{i=1}^{N_s} \alpha_{o,i} d_i x_i$. The hyperplane depends only on support vectors.

10.1.2 Dual Problem

To obtain the Lagrangian multipliers α_i, we solve the problem in its dual form:

Quadratic optimization problem (dual form)

Given the training examples $TR =< x_i, d_i >$, find the optimal values of α_i which maximize

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j x_i^T x_j$$

satisfying the constraints

$$\alpha_i \geq 0\,, \forall i = 1, \ldots, N\,,$$

$$\sum_{i=1}^{N} \alpha_i d_i = 0\,.$$

The value of α_i can be found by solving the quadratic programming (QP) problem, or by more recent and efficient approaches (such as sequential minimal optimization (SMO)). Solving this problem scales with the number of training examples, less with the dimensionality. We don't actually need to explicitly know w_o. All we need are calculating the Lagrangian multipliers by solving the dual problem, and then calculating

b_o. So, given the input pattern x, we compute $g(x) = \sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x + b_o$, and classify it as $h(x) = sign(g(x))$.

This approach of finding optimal separating hyperplane maximizing the margin also provides:

- An unique solution with zero errors for the binary classifier;
- An automatized approach to Structural Risk Minimization that minimizes the VC-confidence without having to deal with hyperparameters;
- The use of a solver in the class of constrained quadratic programming (instead of gradient descent) with a nice dual form;
- A solution focused on a selection of training data points: the support vectors.

But what about noisy or non linearly separable data?

10.2 Soft Margin SVM

In realistic datasets, the training set is not going to be perfect. It will contain noisy points and outliers that make the problem not linearly separable. The solution is to find a separator with a soft margin, that is, a margin that allows some errors within it. Because the margin can allow points to fall inside of it, the support vectors are no longer going to be the closest points to the margin.

We introduce what are called **slack variables**, which are non negative scalar variables:

$$\xi_i \geq 0 \, , \forall i = 1, \ldots, N$$
$$d_i(w^T x_i + b) \geq 1 - \xi_i \, , \forall i = 1, \ldots, N$$

If $x^{(s)}$ is a support vector, it will satisfy the equation:

$$d_i(w^T x^{(s)} + b) = 1 - \xi_i$$

The problem can be then rewritten to admit points in the margin.

Quadratic optimization problem with Soft Margin (primal form)

Given the training examples $TR =< x_i, d_i >$, find the optimal values of w and b which minimize

$$\Psi(w, \xi) = \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i$$

satisfying the constraints

$$d_i(w^T x_i + b) \geq 1 - \xi_i \,, \forall i = 1, \ldots N$$
$$\xi_i \geq 0 \,, \forall i = 1, \ldots N$$

The term C is a user-defined regularization hyperparameter; so this version of SVM is no longer "automatic" like hard margin SVM was. C is found as the trade-off between empirical risk minimization and capacity term (VC-confidence). If C is too low, we allow many training errors, leading to underfitting. If C is too high, we don't let any training error, leading to overfitting.

Quadratic optimization problem with Soft Margin (dual form)

Given the training examples $TR =< x_i, d_i >$, find the optimal values of α_i which maximize

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j x_i^T x_j$$

satisfying the constraints

$$0 \leq \alpha_i \leq C \,, \forall i = 1, \ldots, N$$
$$\sum_{i=1}^{N} \alpha_i d_i = 0$$

The Kuhn-Tucker conditions can be redefined as:

$$\alpha_i (d_i(w^T x_i + b) + \xi_i - 1) = 0 \,, \forall i = 1, \ldots, N$$
$$\mu_i \xi_i = 0 \,, \forall i = 1, \ldots, N \,,$$

where μ_i are Lagrange multipliers introduced to enforce non-negativity of the slack variables in the primal function. If $0 < \alpha_i < C$, then $\xi_i = 0$. If $\alpha = C$, then $\xi_i \geq 0$. To solve the problem, we again solve the dual problem with respect to α_i, and calculate w_o and b_o exactly as before.

10.3 Mapping To a High-dimensional Space

If the TR set represents a non-linearly separable problem, the data points can be mapped from the input space to a high-dimensional **feature space**, where they are linearly separable. The approach we follow is analogous to the LBE for linear models. We define some function $\phi : \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}$. Finding this function, however, is not always easy, unless we have some prior knowledge allowing us to select the proper feature space. Also, using high dimensional feature spaces can lead to overfitting.

Given this function, we map all points to the new feature space: $x \mapsto \phi(x)$. The problem is formulated as before, with a new training set $TR =< \phi(x_i), d_i >$, and a new hyperplane $g(x) = w^T\phi(x) + b = 0$. The notation used here incorporates the bias in the weight vector, with $w_0 = b$ and $\phi_0(x) = 1$:

$$\phi(x) = (\phi_0(x) = 1, \phi_1(x), \ldots, \phi_{m_1}(x))^T$$

The weight vector is now a linear combination of the feature vectors:

$$w = \sum_{i=1}^{N} \alpha_i d_i \phi(x_i)$$

and the hyperplane equation can be written as:

$$g(x) = \sum_{i=1}^{N} \alpha_i d_i \phi^T(x_i)\phi(x) = 0$$

Evaluating $\phi(x)$ may be intractable. Fortunately, under certain conditions we do not need to evaluate it directly, or even know the feature space itself. This is possible with a so-called **kernel trick**, i.e. by using a function k to directly compute the dot products $\phi^T(x_i)\phi(x)$ in the feature space:

$$k : \mathbb{R}^{m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}$$

This function is known as the **inner product kernel function**. It's also a symmetric function, so $k(x_i, x) = k(x, x_i)$. Consider the function $\phi(x) = \phi((x_1, x_2)^T) =$

$(x_1^2, \sqrt{2}x_1x_2, x_2^2)^T$. Given $x = (x_1, x_2)^T$ and $y = (y_1, y_2)^T$ in $\mathbb{R}^2$, we compute $\phi^T(x)\phi(y)$ in this

$$\phi^T(x)\phi(y) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)(y_1^2, \sqrt{2}y_1x_2, y_2^2)^T =$$
$$x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 = (x_1y_1 + x_2y_2)^2 =$$
$$((x_1, x_2)(y_1, y_2)^T)^2 = (x^Ty)^2 = k(x, y)$$

We can arrange the dot products in the feature space between the image of the input training patterns in a N by N matrix, called **kernel matrix**:

$$K = \begin{pmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ k(x_2, x_1) & \dots & k(x_2, x_N) \\ \vdots & & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{pmatrix}, K = \{k(x_i, x_j)\}_{(i,j)=1)}^N$$

The kernel matrix is symmetrical, as the inner product kernel is symmetrical. Not every kernel function computes the inner product in a feature space; this property holds only for kernels gaining positive semi-definite kernel matrices. This is related to the matrix having non negative Eigenvalues.
Given k_1 and k_2 both kernels over $\mathbb{R}^{m_0} \times \mathbb{R}^{m_0}$. The following are also kernel functions:

- $k_1(x, y) + k_2(x, y)$;
- $\alpha k_1(x, y) \, \forall \alpha \in \mathbb{R}_+$;
- $k_1(x, y)k_2(x, y)$.

The problem is reformulated in both forms as follows.

Quadratic optimization problem in feature space (primal form)

Given the training examples $TR =< \phi(x_i), d_i >$, find the optimal values of w which minimizes

$$\Psi(w, \xi) = \frac{1}{2}w^Tw + C\sum_{i=1}^{N}\xi_i$$

satisfying the constraints

$$d_i(w^T\phi(x_i)) \geq 1 - \xi_i \, , \forall i = 1, \dots N$$
$$\xi_i \geq 0 \, , \forall i = 1, \dots N$$

Quadratic optimization problem in feature space (dual form)

Given the training examples $TR =< \phi(x_i), d_i >$, find the optimal values of α_i which maximize

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j d_i d_j k(x_i, x_j)$$

satisfying the constraints

$$\sum_{i=1}^{N} \alpha_i d_i = 0$$

$$0 \leq \alpha_i \leq C \, , \forall i = 1, \ldots N$$

To classify an unseen input pattern x using the machine we trained, we compute $\sum_i \alpha_i d_i k(x, x_i)$, and then classify x as $h(x) = sign(\sum_{i=1}^{n} \alpha_i d_i k(x, x_i))$. Note: we have to memorize the x_i in the training set for the test phase.

Some examples of commonly used kernels include:

- **Polynomial Learning Machine**: $k(x, x_i) = (x^T x_i + 1)^p$ (where p is a user-specified parameter);
- **Radial Basis Function Net**: $k(x, x_i) = e^{-\frac{\|x - x_i\|^2}{2\sigma^2}}$ (where σ is a user-specified parameter);
- **Two-layer Perceptron**: $k(x, x_i) = tanh(\beta_0 x^T x_i + \beta_1)$ (where $\beta_0 > 0$ and $\beta_1 < 0$ are user-specified parameters).

Using an RBF always leads to a feature space with an infinite number of dimensions.

10.4 Pros and Cons of SVM

Pros:

- The regularization is embedded within the optimization problem, so it does not need any external hyperparameter for it;
- It automatically approximates SRM by finding the hypothesis with the best possible VC-dim;

- It's a convex problem, so the global minimum can always be found;
- Features are implicitly transformed through kernels;
- Linear model with bound of the complexity that depends on the margin (which is optimized);
- Rich set of non-linear decision functions in the input space via kernels.

Cons:

- Kernel and kernel parameters must be chosen explicitly;
- It uses a batch algorithm to update the weights, so no chance to parallelize;
- Very large problems were computationally intractable (although nowadays many efficient solutions have been proposed, including gradient descent ones);
- For soft margin SVM, since we have to select the C parameter and the kernel function, we have no guarantee that the final model will have high accuracy.

Note that for the last point about soft margin, the VC-dim of the model is controlled by the width of the margin. If we choose the smallest possible margin (with just one support vector), we can classify an arbitrarily large number of training points correctly, thus the VC-dim will be infinite. If instead we have a very large width, we end up in a situation where all points in the TR set are used as support vectors, therefore leading to a low VC-dim. This is analogous to using a k-NN classifier with $k = 1$ and $k = l$, respectively. Also, the hyperparameter C used for regularization must be chosen appropriately for the kernel hyperparameters, so cross-validation is needed.

Chapter 11

Neural Networks

(Artificial) neural networks are a flexible machine learning tool, encompassing a wide set of models capable of approximating functions. They can learn from examples, are universal approximators (Cybenko's/Universal approximation Theorem), can deal with noisy and/or incomplete data, and can handle both continuous real and discrete data.

11.1 Artificial Neurons

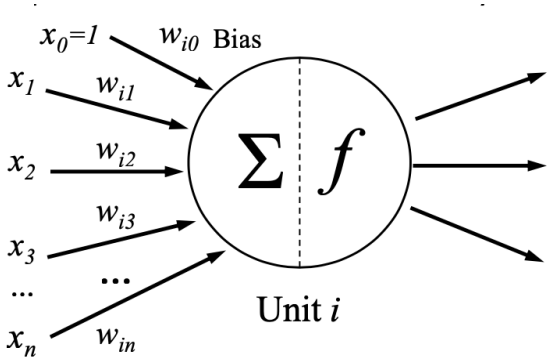
Artificial neural networks are made up of several **nodes** (also called artificial neurons, or units) connected in a net capable of solving artificial intelligence problems. They are heavily inspired by biological neural networks, down to how the single units work and communicate with each other to learn a certain function.
The main ideas that these models are based on are:

- **Strength reinforcement**: stimuli "reinforce" the weights;
- **Plasticity**: the nervous system is highly capable of adapting.

Each neuron i has a number of **inputs** coming from external sources or other units, and corresponding **weights**, the free parameters that can be modified during the learning phase. Each neuron calculates its **net input** (weighted sum of inputs) and its output as follows:

$$\begin{cases} net_i(x) = \sum_j w_{ij} x_j \\ o_i(x) = f(net_i(x)), \end{cases}$$

where f is the unit's **activation function**. Note that w_{ij} is the weight of the input coming from the node j and going into the node i; some books/libraries/simulators, etc. may use the opposite notation.

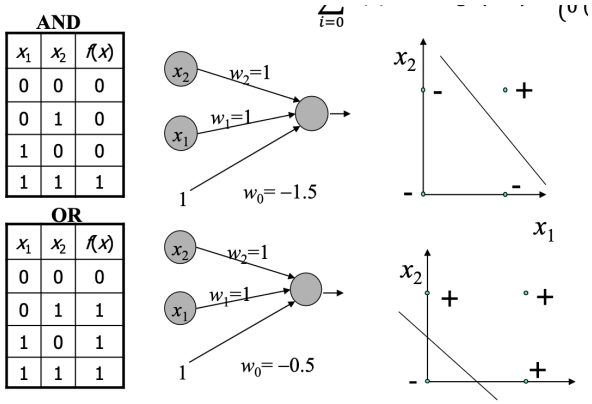


11.1.1 Perceptrons

The perceptron was proposed and implemented in 1958 by Frank Rosenblatt. The first perceptron was an actual physical machine designed for image recognition, and not a program. It had a few hundreds of photocells, randomly connected to a layer of neurons. Multiple perceptrons can be composed and connected to build a network. This is called a **multi-layer perceptron neural network (MLP NN)**.

McCulloch and Pitts had already proposed a neural network model in 1943. In this model, each neuron is in one of two possible states: firing (1), or not firing (0). All synapses (connections) are equivalent and characterized by a weight w_i which is positive for "excitatory" connections, and negative for "inhibitory" connections. A neuron i becomes active when the sum of the connections coming from other active neurons and the bias is larger than 0. Both inputs and outputs are binary, so it can implement binary classification tasks.

The following pictures illustrate how this model can be used to represent boolean functions AND and OR by using a single neuron. Each neuron has two inputs, plus a bias (w_0). Both problems are linearly separable, so if the inputs are represented graphically on a plane, we can draw a single line (described by the net function) that separates all inputs that produce a 0 from all the ones that produce a 1.



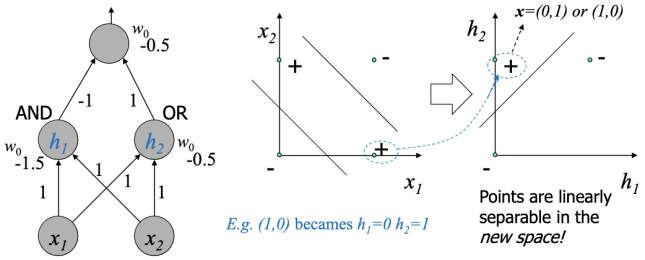
If we instead consider the XOR operator, it corresponds to a non-linearly separable problem, so we can't use a single neuron. The solution is to use a two layer network. The operation can be rewritten as follows:

$$x_1 \oplus x_2 = x_1 \cdot \bar{x_2} + \bar{x_1} \cdot x_2$$

then we have:

$$x_1 \oplus x_2 = \bar{h_1} \cdot h_2$$

So the XOR operation is moved to a new space that represents the problem as linearly separable:



This type of composition can be used to perform more complex tasks, extending the network to many layers of abstraction. In NN, this internal representation can be learned.

11.2 Learning Algorithms For One Unit Models

There's two kinds of learning algorithms:

- **Perceptron**: can only be used for classification.
- **ADALINE (Adaptive Linear Neuron)**: can use LMS with either SVD or gradient descent algorithm. This approach an be generalized to multi-level perceptron NNs.

11.2.1 Perceptron Learning Algorithm

The goal of the algorithm is to minimize the number of misclassified patterns; so it must find w such that $sign(w^T x) = d$. This is an on-line algorithm, so one step can be done for each input pattern. The algorithm can be summarized as follows:

1. Initialize the weights (either to 0 or a small random value);

2. Pick a learning rate η;

3. For each training pattern $< x, d >$, where d is either $+1$ or -1, compute $out = sign(w^T x)$; if $out = d$, don't change the weights, otherwise modify the weights as:

$$w_{new} = w + \eta dx$$

or, in a different form,

$$w_{new} = w + \frac{1}{2}\eta(d - out)x$$

Looking at the problem from a geometrical point of view, it's as if we modified the weight vector w by summing the vector ηdx, where d indicates the direction of the vector with respect to that x: if positive, the addition will move w "towards" the point, if negative, it will move it "away" from it.
The form $w_{new} = w + \eta dx$ is in the form of Hebbian learning, while $w_{new} = w + \eta(d - out)x = w + \eta\delta x$ is in the form of error-correcting learning.

11.2.2 Differences Between LMS and Perceptron Learning Algorithm

They are apparently very similar; they both calculate the new value of w by adding a δ multiplied by the learning rate η, along with the input x for the Perceptron Learning Algorithm. The δ for LMS is calculated as $(d - w^T x)$, while for the perceptron learning algorithm it's $(d - sign(w^T x))$. However similar they are, there a few important differences:

- LMS does not necessarily minimize the number of training examples misclassified by the LTU, since it changes the weights for both misclassified and correctly classified ones;
- The perceptron learning algorithm always converges for a linear separable problem to a perfect classifier, while LMS has asymptotic convergence (also for non linearly separable problems);
- The perceptron learning algorithm is difficult to extend to a NN, while the LMS can be extended to a NN by using the gradient based approach.

11.3 Activation Functions

As seen before, the possible activation functions can be a linear function, a threshold function (perceptron/LTU), or a non-linear function such as the **sigmoidal logistic**

function. The latter is a function that assumes a continuous range of values in the bounded interval $[0, 1]$. It has the important property of being a smoothed differentiable function. The slope of the sigmoid function is defined by the parameter a. Some common examples of functions are:

- **Logistic function**: $f_\sigma(x) = \dfrac{1}{1 + e^{-ax}}$ (with output in the range $[0, 1]$)
- **tanh function**: $f_{tanh}(\frac{x}{2}) = 2f_\sigma(x) - 1$ (with output in the range $[-1, +1]$)
- **Radial basis function**: $f(x) = e^{-ax^2}$
- **Rectified linear unit (ReLU)**: $f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
- **Softplus function**: $f(x) = ln(1 + e^x)$

The derivative of the identity function is 1. The derivative of the threshold function is not defined, which is why it's not used in LMS. As for sigmoid functions:

$$\frac{df_\sigma(x)}{dx} = f_\sigma(x)(1 - f_\sigma(x))$$

$$\frac{df_{tanh}(x)}{dx} = 1 - f_{tanh}(x)^2$$

11.4 LMS With Sigmoidal Function

Since the sigmoidal logistic function is differentiable, we can derive a LMS algorithm by computing the gradient of the mean square loss function as for the linear units. The output of a neuron is calculated as:

$$out(x) = f_\sigma(x^T w)$$

The objective of the algorithm is to find $w = argmin_w E(w) = \sum_p (d_p - out(x_p))^2 = \sum_p (d_p - f_\sigma(x_p^T w))^2$, so the weights that minimize the residual sum of squares. The gradient descent algorithm uses the new delta rule: $w_{new} = w + \eta \delta_p x_p$, where $\delta_p = (d_p - out(x_p)) f'_\sigma$. Additionally, the parameter a, the slope of the function f_σ, can affect the step of the gradient descent.

The max of f' corresponds to net inputs close to 0, while the minimum of f' corresponds to **saturated cases**, as in where the function f goes to either 0 or 1 asymptotically.

11.5 Multi Layer Perceptrons

A MLP can be seen in two possible ways: either as a network of units, or as a flexible function. As a network, a MLP contains a number of units connected by **weighted links**. The units are organized in layers: the first layer that loads the input is called the **input layer**; the layer that produces the final output is called the **output layer**; all other inbetween layers are called **hidden layers**. The notation used to refer to networks will be the following:

- the index t denotes a generic unit, while k denotes an output unit;
- the index u denotes a generic input component;
- x is a generic input from an external source (if it's an input vector) or from another unit;
- if the pattern x is loaded in the input layer, the notation o can be used for both inputs and hidden layer inputs, so, inside the network, the input to each unit t from any source u is simply denoted as o_u.

As a flexible function, the MLP can be written as a function in the following form:

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} f_i(\dots))),$$

where f_k is a sigmoid activation function.

11.5.1 Architecture

The architecture of a NN defines the topology of the connections between units. A **feedforward architecture** describes a network that operates as follows. For each input pattern x, do:

1. load the input in the input layer;
2. compute the output of all the units in the first hidden layer;
3. compute the output of all the units in the second hidden layer;
4. ...
5. compute the output of all the units in the output layer;
6. compute the error (delta) at the output level.

A **recurrent architecture** describes a network with feedback loops, which allow the output of units to go "back" in the network towards units in previous layers.

11.6 Flexibility of NNs

The hypothesis space of a NN is the continuous space of all the functions that can be represented by assigning the weight values of the given architecture. Depending on the class of values produced as output, the model can deal with both regression and classification tasks, by using, respectively, a linear function or a sigmoidal one. It can also implement multi-regression and multi-class classifiers by defining multiple output units.
When we consider a NN as a function, so in the form

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} f_i(\dots))),$$

each f_j can be seen as computed by an independent unit, or a special kind of ϕ of Linear Basis Expansion. Additionally, $h(x)$ is non-linear in the parameters w. So, we can reformulate the function as follows:

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} x_i)) = f(\sum_j w_j \phi_j(x, w))$$

The main difference with LBE is that the ϕ are adapted to data by fitting the w.

11.7 SGD and Backpropagation Learning Algorithm

The universal approximation theorem declares that single hidden-layer network (with logistic activation functions) can approximate (arbitrarily well) every continuous function, given enough units; a MLP network can approximate (arbitrarily well) every input-output mapping (provided enough units in the hidden layers), as well. Note that this theorem does not say which learning algorithm to use, nor the number of units needed. The expressive power of NN is strongly influenced by two aspects: the number of units (and in turn, the number of weights w), and the architecture. Another important aspect is the number of layers to use in the network. The universal approximation theorem tells us that 1 layer is sufficient to approximate any function, yet it gives no indication on the number of units to use, which may be incredibly high. So, instead of using only one layer, the network is split into multiple layers, each with a limited number of units.

The most common learning algorithm used for NN is Stochastic Gradient Descent (LMS approach). The goal is to adapt the free parameters w in order to obtain the best approximation of the target function. This is normally done by checking the value of the error (or loss) function calculated on the training set. In the case of NNs, however, this

is not as straightforward: the network has multiple units across layers, so the learning algorithm must first decide how much credit to give to the hidden units in causing the error to increase.

The so called **loading problem** is formulated as follows: given a network and a set of examples, is there a set of weights so that the network will be consistent with the examples? The problem is NP-complete, and while networks can in practice be trained in a reasonable amount of time, an optimal solution in not guaranteed.

The idea behind the backpropagation algorithm solution is to extend the gradient descent approach, so it can be used with MLP networks as well. The things we need are:

- differentiable loss;
- differentiable activation functions;
- a network to follow the information flow.

What we want to find is w obtained by computing the gradient of the error function. The advantages of the backpropagation algorithm are:

- it's easy because of the compositional form of the model;
- it keeps track of the quantities local to each unit;
- it's efficient, since it's $O(\#W)$ rather than $O(\#W^2)$
- supposedly, the brain's learning "algorithm" is a local sub optimal approximation of backpropagation (although this debate remains controversial).

Other than SGD, other commonly used optimization algorithms are:

- RMSprop, which adapts the learning rate by reducing it using a moving average of the squared gradient;
- Adagrad, similar to RMSProp but with element-wise scaling of the gradient;
- Adam, which adds an exponentially decaying average of past gradients (like momentum in SGD).

11.7.1 Issues in Training NNs

The resulting model is often over-parameterized; additionally, the optimization problem is no longer convex, and is potentially unstable.

Some problems in training NNs involve:

- **Hyperparameters**: starting values, choosing between on-line/batch, learning rate, number of hidden units in the network;
- **Multiple minima**: the loss is no longer a simple convex function, so finding a minimum needs specific techniques;
- **Stopping criteria**;
- **Overfitting and regularization**;
- **Input scaling/output representation**.

Hyperparameters

Starting values The weights are normally initialized with random values near 0; we want to avoid a completely null weight vector, too high values, or all components equal to each other, since these can hamper training. We can also consider the **fan-in**, i.e., the number of inputs to a hidden unit: $range * 2/fan - in$. This should be avoided if the fan-in is too large, or if the unit in question is an output unit, since the δ would be too close to 0 (since we're using backpropagation, those δ values would be propagated to previous layers, and cause all others to decrease as well).

On-line or Batch gradient descent The batch version of the algorithm calculates all the gradients of each pattern over an epoch, then it updates the weights. The stochastic version instead updates the weights after calculating the gradient on one pattern at a time; it also needs a smaller learning rate η compared to the batch version, since a value too high may produce a training that's too "chaotic".

In terms of MLP with backpropagation, the way the gradient is calculated is:

$$-\frac{\partial E(w)}{\partial w_{tu}} = -\sum_{p=1}^{l} \frac{\partial E_p(w)}{\partial w_{tu}} = \sum_{p=1}^{l} \delta_{p,t} o_{p,u}$$

for the batch version, and:

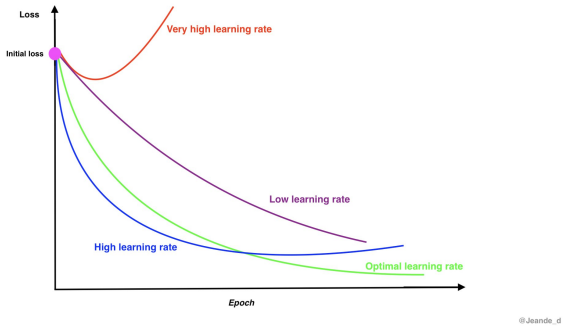
$$-\frac{\partial E(w)}{\partial w_{tu}} = \delta_{p,t} o_{p,u}$$

for the on-line version.

The batch version produces a more accurate estimation of the gradient, but the on-line version can help avoid local minima while not being as accurate. Additionally, when using the on-line version, a random shuffling of the patterns should be done before each epoch, in order to avoid any bias in the gradient descending.

A variant of the on-line version is the **minibatch (MB)**. In this version, the epochs are divided in parts; the gradient is summed up to mb patterns (mb = the size of a part) before updating the weights, instead of updating them after every single calculation. Indeed, the batch an on-line versions can be seen as extreme cases of the minibatch: the first uses a part that's exactly the same size as the dataset, the other uses parts of size = 1. A commonly used value of mb is 100, since it's the ideal partitioning of the dataset that exploits GPU memory parallelism the best.

Learning rate An higher value of η causes the descent to be faster but more unstable; a lower value of η causes the descent to be slower but more stable. A way to monitor the behavior of the model (and adjust the learning rate as desired) is to plot the **learning curve**, which is obtained by plotting the error against the number of epochs. Depending on the value of η, the curve may have a different appearance. In a practical



approach, it's useful to consider the mean of the gradients over the epoch, in order to have a uniform approach with respect to the number of input data. When using LMS, dividing by l is equivalent to using η/l as our learning rate. As a way to improve the choice of learning rate, the following approaches may be considered:

- Using **momentum**: by adding momentum, Δw is now calculated as:

$$\Delta w = -\eta \frac{\partial E(w)}{\partial w} + \alpha \Delta w_{old}\,,$$

  so we introduce a new term that depends on the previous value of Δw. α is a value between 0 and 1. It's commonly assumed to help with batch mode more, but can also be used with on-line; in this case, Δw_{old} is the Δw_{p-1} (as in, of the previous example).

  A variant of momentum is the **Nesterov momentum**, where the gradient is evaluated after the momentum is applied; so we first calculate $w = w + \alpha \Delta w_{old}$, the we evaluate the gradient on this new w. This variant has been shown to

improve the rate of convergence for the batch mode, but not for the stochastic mode.

- Variable learning rate (start high, then decrease): using minibatch, the gradient does not reach 0 even when close to a minimum (as exact gradient can do), hence a fixed learning rate should be avoided. We can linearly decay η for each step until iteration τ, using $\alpha = step/\tau$:

  $$\eta_s = (1 - \alpha)\eta_0 + \alpha\eta_\tau$$

  and then stop for some iteration, from which we can use a fixed small value of η. Ideally, the final value of η should be $\sim 1\%$ of η_0, so it takes a few hundred steps to reach it.

- Adaptive learning rates (changed during training and for each w): automatically adapt the learning rate during training, avoiding or reducing the fine tuning phase via hyperparameter selection. Some popular ones include AdaGrad, RMSProp, Adam. Can be combined with momentum.

- Varying in NNs with many layers (higher for deep layers, or higher for units with few input connections).

Number of units The number of units controls the complexity of the NN. This choice is in general a model selection issue, so it's selected by a cross-validation. Few units typically lead to underfitting, while too many lead to overfitting, but we can have NNs with many units that don't overfit if regularization is used. We can follow two approaches; either a **constructive**, incremental, one, in which the learning algorithm decides the (small) starting number of units and then adds more as needed, or a **pruning** one, where we start with a large network and progressively eliminate weights or units.

Multiple Minima

Loss in no longer convex. The function may have multiple minima, as well as maxima. The final result depends a lot on the starting weight values. Ideally, we should try a number of random starting configurations (trials), and then take the mean result (as in, the mean of errors) and check the variance in order to evaluate the model. Then, we can either pick the solution that produces the lowest/median validation error, or we can consider the mean of the outputs.

It's worth noting that finding a local minima with too high error that stops training is not a big issue, since we can always check the final training error (and restart if

needed). In general, we don't need to find the global minimum, a "good" local minima is sufficient. This is because the minimum we find is of R_{emp}, not R (which is what we want to approximate). Also, instead of finding a point corresponding to the null gradient, we may want to stop at a point that has sufficiently small gradient.

Another aspects to consider is that the NN builds a variable size hypothesis space, and tends to increase the VC-dim during training. As the VC-dim increases, the R_{emp} decreases towards the global minimum, but we'll likely incur into overfitting, so stopping before we find the minimum might actually produce a better approximation of the target function.

Stopping Criteria

The basic stopping criteria is to check the error (mean or max error E), however, we may not always have enough information to set a tolerance threshold for the error. We may want to use an internal criterion: we stop if the improvement of the error (e.g., less than 0.1%), or the changes to the weights (e.g. norm of gradient $< \eta$) are negligible.

We must not stop at an arbitrary number of steps; if it's too small, then it may be too early and cause underfitting, while if it is too large, we may incur into overfitting. Also, if we use K-Fold cross validation, the number of ideal steps for each fold may vary: how do we choose it for the final model, trained over all data? We can consider the number of epochs as a hyperparameter, and select its value as the mean across the folds; however, changing the data size at the end, adding all the records of validation and test set, the stop point may be different, leading to underfitting.

Overfitting and Regularization

As said before, we typically don't want the global minimizer of R_{emp}, since it would be an overfitting solution. The control of complexity requires some form of regularization. This can be achieved by introducing a penalty term, or indirectly by early stopping. Another important step is to perform cross-validation on empirical data to find the best trade-off.

In NNs, learning normally starts by setting the weights to small, random values, and the complexity is low. As optimization proceeds, hidden units tend to saturate, increasing the number of free parameters, hence increasing the complexity of the model. So, how do we choose when to stop this optimization?

- **Early stopping**: using a validation set to determine when to stop. We ideally want to consider multiple epochs to estimate the error. Since the effective number

of parameters grows during the course of training, stopping means limiting the complexity.

- **Regularization**: we can use a regularization related to Tikhonov theory, applied to the loss; a penalty term is added, such that the loss will be calculated as:

$$Loss(w) = \sum_p (d_p - o(x_p))^2 + \lambda \|w\|^2$$

This is a form of weight decay, since the new values of the weights during gradient descent will be calculated by adding the term $-\lambda w$, which causes it to decrease even when Δw is 0. The regularization parameter λ is generally a low value, and is selected in the model selection phase.

Remember that, when using regularization, the loss is used during model training, while the error (or risk) for the "data term" is used during model evaluation, since it only measures how different the output of the hypothesis is from the correct label. A common misconception is that regularization helps convergence stability, but it does not. It only controls the complexity.

Also, early stopping and regularization can be used together. The difference is that early stopping is an **empirical approach**, that requires a VL set to decide a stopping point, while regularization is a **principled approach**, and allows the VL curve to follow the TR curve.

Note, also, that the bias w_0 is often omitted from the regularizer, since its inclusion causes the results to not be independent from target shift or scaling. If it's included, it has its own regularization term.

Input Scaling/Output Representation

Preprocessing of the data can have a large effect on the result of training. Data should be normalized via either standardization (each feature is modified so that mean is 0 and standard deviation is 1), or rescaling (the range of values is restricted to [0,1]).

For regression, there's one or more output linear units; for classification, there's either a singular binary output unit, or there's multiple binary units. In the latter case, we can use a sigmoid to choose the threshold to assign the class, a rejection zone, or 1-to-K encoding to choose the "winner" class. Often, the symmetric logistic function learns faster. To avoid asymptotic convergence, 0.9 and 0.1 can be used instead of 1 an 0 as a label value (for the logistic function, for the tanh function -0.9 is used instead of -1). In any case, the target range must be in the output range of units. If targets are

0/1, it's common to use the Softmax function:

$$o_k(x) = \frac{e^{(-net_k)}}{\sum_{j=1}^{K} e^{(-net_j)}}$$

Or the (binary) cross entropy:

$$E(w) = -\sum_{i \in TR} \{d_i \log(out(x_i)) + (1 - d_i) \log(1 - out(x_i))\}$$

11.8 When To Consider NNs

- The input is high dimensional discrete or real-valued, both for regression and classification tasks;
- The dataset possibly contains noise;
- The form of the target function is unknown;
- Human readability of the output is not critical;
- Training time is not critical;
- The computation of the output itself has to be fast.

Chapter 12

Deep Learning

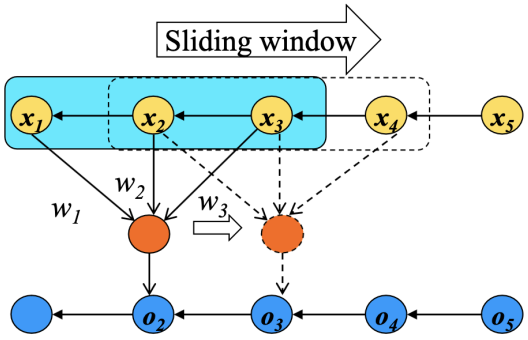
12.1 Convolutional Neural Networks

Convolutional Neural Networks are a specialized kind of neural network for processing data with a known, grid-like topology, such as 2D images. The name "convolutional" refers to the mathematical operation called **convolution**, indicated by an asterisk ($*$). This is an operation of two functions of a real valued argument, defined as follows:

$$s(t) = (f * g)(t) \stackrel{def}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau\,,$$

The idea behind this operator is that we want to calculate the average of f weighted by another function g moved over time ("sliding"), calculated for a certain t. In convolutional network terminology, the first argument (here, f) is referred to as the **input**, and the second argument (g) as the **kernel**. The output is called **feature map**.

This operator can be applied to neural networks as well. Consider a simple network with one hidden layer:



Here, the output of each node in the hidden layer is calculated as $out_t = \sum_{i=1}^{3} w_i x_{t+i-2}$. In other words, the weights assigned to the inputs "slide" across the hidden layer. Weights are tuned as usual by learning.

12.1.1 2D Convolution

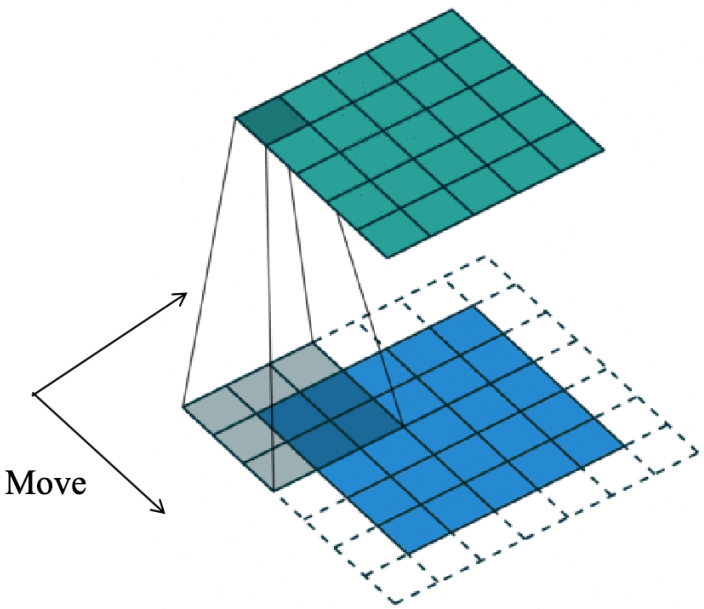
Discrete convolution can be seen as a multiplication by a matrix, with several entries constrained to be equal to other entries. The convolution over a 2D image I with a kernel K can be expressed as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)\,,$$

or, as expressed by many libraries, as the **cross-correlation function**:

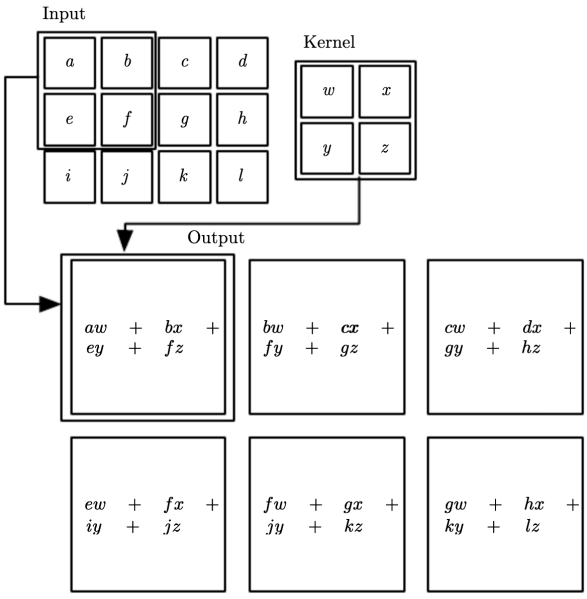
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)\,.$$

The example below shows a 2D image with 25 pixels, and a 3x3 kernel (unit local receptive field) with a stride equal to 1 (i.e., the kernel moves across the image 1 pixel at a time; by choosing the stride we choose the size of the feature map). The image also has padding added to its edge.



Once the kernel reaches the end of the first "row" of pixels, it restarts from the position it started from shifted one pixel below. The full movement of the kernel over the image produces the feature map. The next image better shows the matrix multiplication interpretation of convolution; the image is 4x3 and the kernel is 2x2. The stride is again equal to 1.

Each unit's weights are a **filter** trained to detect some specific feature or pattern in the image. Each filter produces the strongest response to a spatially local input pattern, and then the filters obtained by training are applied to the whole (global) image, so



that features and patterns can be identified regardless of where they are positioned on the image.
The size of the feature map can be reduced via pooling. Some examples of pooling are:

- **Subsampling**, using a stride greater than 1;
- **Average Pooling** (normal or weighted average);
- **Max Pooling** (most common option).

This way, instead of producing a value for every single pixel of the original image, we get a smaller set of pixels where each value is obtained by considering the values of neighboring ones (calculating the mean or max value). Pooling also helps to make the representation approximately invariant to small translations of the input, since the mean or max value of a neighborhood of points is unlikely to be affected by a small translation of the pixels in the image.

CNN exploits **weight sharing**, where the number of connections in the network is kept the same while reducing the number of actual free parameters. The produced sliding window of units is applied over a segment of the input, and reapplied multiple times to produce various layers of feature maps. Training of the weights is usually done via backpropagation. Since these networks tend to be big and deal with large amount of data, many hyperparameters are fixed by experience or by suggestions of experts, since it would be too expensive to run cross-validation.

12.2 Deep Learning

The Deep Learning framework includes many different models, such as:

- Deep Neural Networks;
- Convolutional Neural Networks;
- Deep belief Networks;
- Recurrent and Recursive Neural Networks.

They differ from "shallow" models in that they have a big amount of layers.

The core concept at the base of deep learning is increasing the level of abstraction of the data through the use of several layers; for example, an image can be gradually abstracted on each layer, first as a vector of intensity values per pixel, then a set of edges, then regions of a particular shape, and so on. We've already mentioned this idea with CNNs: the original complicated mapping is broken down into its simple elements, by gradually calculating simpler mapping at each layer. A series of hidden layers extracts increasingly abstract features from a set of example images. Additionally, these abstract features, once learned by the units, can also be combined together to generalize on examples that were never seen during training.

In general, deeper networks are often able to use less units per layer, thus less free parameters as well and less training data required to achieve a good generalization. Still, many layers may be harder to be trained, so there's a need to improve the techniques we know regarding gradient descent, regularization, and data exploitation.

12.2.1 Insights

Why So Many Layers?

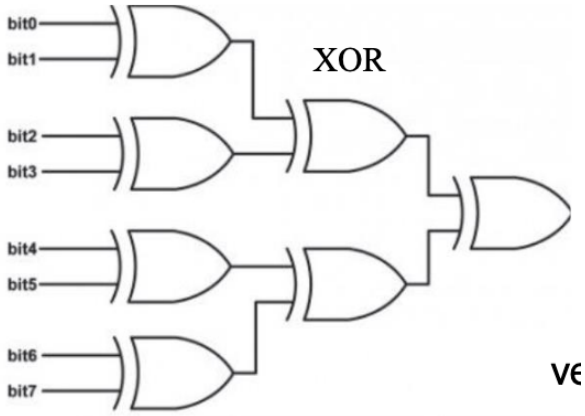
Imagine a two-layer circuit of logic gates, which can represent any Boolean function. Any Boolean function can indeed be written as a sum of products (i.e., in disjunctive normal form). With logical circuits of depth two, the number of logic gates required to represent most Boolean functions is exponential w.r.t. input size.

An example of such function is the parity function: it returns 1 if there is an odd number of 1s over N binary inputs (i.e., N bits), 0 otherwise. If we were to implement this function with logic gates, assuming N inputs, we would need:

$$\frac{2^N}{2} + 1 = 2^{N-1} + 1$$

gates, since we have to perform 1 OR and exactly 2^{N-1} ANDs.

We can propose an alternative solution with a polynomial number of gates, by increasing the number of layers to $log(N)$. The solution for $N = 8$ is shown below (in a simplified view where each XOR corresponds to 2 AND gates and 1 OR gate). In this solution, the number of gates is greatly reduced from $2^7 + 1 = 129$ to only $7 * 3 = 21$.



The universal approximation theorem states that even with 1 hidden layer, a NN can approximate any possible function, however it does not specify the number of units needed. For some families of functions a boundary on this number can be found, but as seen in the example above, the bound may be exponential w.r.t. the dimension of the input. Also, there exist families of functions which can be approximated efficiently by a NN with depth greater than some value d, but which require a much larger model if depth is restricted to be less or equal than d.

This theorem also implies that regardless of what function we are trying to learn, a large MLP is able to represent the function, but there's no guarantee that that the training algorithm will be able to learn that function, either because it can't find the value of the parameters that correspond to the desired function, or because it might choose the wrong function due to overfitting. So another advantage to using multiple layers is ensuring that the learning algorithm can actually properly learn.

The inductive bias is: choosing a deep model encodes the (very general) belief that the function we want to learn should involve composition of several simple functions. If our task actually matches our bias, then the deep shape of the learner is suitable, and those deeper models also perform better than shallow ones. Typically, these tasks are the ones that involve images or language, but for other tasks that deal with different data this deep structure may not be appropriate.

12.2.2 Techniques

When implementing a deep neural network, there's many technical aspects to consider. As already stated before, the deeper the network, the less units are needed for each layer, and therefore there will be less parameters to train, but some layers may be difficult to train. This section will focus on the methods that need improvement and their issued when used with deep NNs.

Batch normalization is a method that normalizes each batch by calculating each individual batch statistics such as mean and variance for each layer. Each matrix (batch x activation of units) is normalized with mean and variance, shifting the values to zero-mean and unit variance. This technique helps to keep the normalization of the input across all layers of the network. It also achieves a faster learning and higher accuracy for Deep Learning.

Dropout is a method that makes bagging more practical for large neural networks. Dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, we can effectively remove a unit from a network by multiplying its output value by 0. Then, a minibatch-based learning algorithm is used to train one working sub-network at a time. The sub-network is selected at random, with each binary mask to apply to the original network having a different probability set as an hyperparameter (e.g., 0.8 for input units and 0.5 for hidden units). The sub-networks inevitably share weights, since they're obtained from the same base network; this causes the training of the sub-networks to find good settings for the parameters.

Dropout has a regularization effect as well: it avoids to train all units on all training data and reduces unit interactions. It also reduces variance without affecting bias just like bagging does. It even regularizes singular hidden units to be not just good features, but features that can be good in different contexts (different sub-networks). It can be used for any model that uses distributed representation and SGD training.

12.3 Recurrent Neural Networks

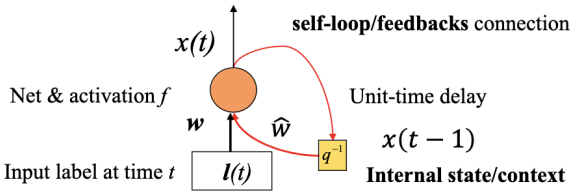
Up until now, we considered feedforward neural networks. The input is read from the first layer, then traverses a number of hidden layers, and the prediction is finally produced by the output layer. Recurrent neural networks are a different category of architecture, based on the addition of feedback loops to the network topology. These self loops provide the network with dynamical properties, introducing the ability of holding a memory (state) of past computations of the model.

RNNs have been the reference approach for sequence processing, especially for

speech and text recognition, processing, and generation. The type of data handled by these models is structured; it is usually an ordered set of sequences of vectors.

12.4 Memory

The introduction of feedback loops allows the network to hold a memory about past computations. This memory is needed because the output of a certain input depends on previous outputs produced by the same network.



The output of the node is calculated for a time t recursively, as:

$$x(t) = \begin{cases} 0 & t = 0 \\ \tau(i(t), x(t-1)) = f(w^T i(t) + \hat{w} x(t-1) + \theta) & t \geq 0 \end{cases}$$

Here, f is the activation function of the unit, $i(t)$ is the input label at time t, $\hat{w}$ is the recurrent weight (the one that's coming from a feedback), and θ is the bias. The internal state summarizes the past information, and changes each time the unit produces a new output. The encoding of the past memory is also adaptive. τ is the state transition function realized by the NN. $x(t)$ here refers only to one state, but it can also include a set of states.

12.5 Properties

Many RNN architectures are possible, but even a simple one with a few nodes each with its own feedback loop is already incredibly powerful. They are universal approximators of non-linear dynamic systems, and are Turing equivalent (they can simulate any automata).

RNN models are based on the following assumptions:

- **Causality**: a system is causal if the output at time t_0 only depends on inputs at time $t < t_0$ (necessary and sufficient for internal state);
- **Stationarity**: time invariance after model training. The state transition function τ is independent on node v of the sequence.

- **Adaptivity**: transition functions are realized by NN with free parameters, so they are learned from data.

RNNs can also be **unfolded**. Unfolding a RNN means representing it as a graph with a repetitive structure corresponding to a chain of events (so it represents how the same model behaves through time). Unfolding is associated with weight sharing between unfolded layers.

Chapter 13

Ensemble Learning

Ensemble methods (also known as classifier combination methods) are used to improve classification accuracy by aggregating the predictions of multiple classifiers. An ensemble method constructs a set of **base classifiers** trained on the training set (sampled differently depending on the specific technique), and then predicts the output on instances by taking the majority vote.

The key idea that justifies the use of ensemble methods is the so-called **wisdom of the crowds**: the collective knowledge of a diverse and independent group of people usually exceeds that of a single individual. According to Surowiecki, there are five elements required to form a wise crowd: diversity of opinion, independence, decentralization, aggregation, and trust. These same elements can be applied to machine learning, where each "individual" in the crowd is a model.

To illustrate how a classifier's performance can be improved, consider the following example.
We have an ensemble of 25 binary classifiers, each with the same error rate $\varepsilon = 0.35$. The ensemble predicts the class label of a test instance by taking the majority vote on the predictions of the single classifiers. If these classifiers are perfectly identical, then they will all answer the same way on any test input, so the error rate of the ensemble will also be ε.

If instead the base classifiers are independent (their errors are uncorrelated), the ensemble makes a wrong prediction only if more than half of the base classifiers make a mistake; so the error rate of the ensemble is:

$$e = \sum_{i=13}^{2} 5 \binom{25}{i} \varepsilon^i (1-\varepsilon)^{25-i} = 0.06$$

which is considerably lower than ε.

Ensemble classifiers can be constructed in many ways:

- By manipulating the training set (bagging, boosting);
- By manipulating the input features (random forests);
- By manipulating the class labels. (error-correcting output coding).

13.1 Bagging

Bagging, which stands for Bootstrap AGGregatING, is a technique that samples with replacement from a data set according to a probability distribution. Given a dataset $X = \{x_1, \ldots, x_n\}$, m datasets o size n are sampled from it, such that each record x_i has $\frac{1}{n}$ probability of being extracted. Since replacement is used, the same record may appear multiple times in the same sample, while some records may not appear at all.

Algorithm 6 Bagging algorithm.

1: Let k be the number of bootstrap samples.
2: **for** $i = 1$ to k **do**
3: D_i = Sample of size N.
4: Train a base classifier C_i on D_i.
5: **end for**
6: $C^*(x) = \arg\max_y \sum_i \delta(C_i(x) = y)$

In the above pseudocode, $\delta()$ is a function that returns 1 if its argument is true, 0 else.

Bagging improves generalization error by reducing the variance of the base classifiers. The performance of bagging depends on the stability of the base classifier: if a base classifier is unstable, bagging helps to reduce the errors associated with fluctuations in the training data, and if it is stable, the error of the ensemble will be mainly caused by bias of the base classifier.

13.2 Boosting

Boosting is an iterative procedure used to change the distribution of training examples for base classifiers, increasing weights for instances that are harder to learn. Initially, all instances in the original dataset are assigned a weight equal to $\frac{1}{n}$; a sample is selected (the same way bagging does, with replacement), and a classifier is built on it. All records that are incorrectly classified by this base classifier have their weight increased, while the ones that are correctly classified have theirs decreased. For the next step,

another sample is obtained considering the new weights, and used to train a second model. After classifying those instances, their weights are updated, and this goes on until the desired number of boosting rounds is reached.

13.2.1 AdaBoost

In the AdaBoost algorithm, each base classifier C_i is assigned an **importance**. It depends on the classifier's error rate, defined as:

$$\varepsilon_i = \frac{1}{l} \sum_{j=1}^{l} w_j \delta(C_i(x_j) \neq y_i)$$

where $\delta()$ is the same function seen in the bagging pseudocode. The importance is then calculated as:

$$\alpha_i = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_i}{\varepsilon_i} \right)$$

When the error is close to 0, the importance is a large positive value, while when the error is close to 1, the importance is a large negative value.

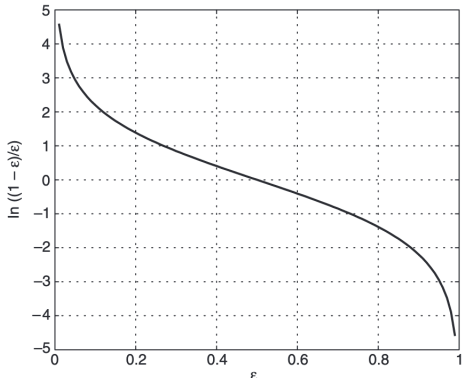


Figure 13.1: α as a function of the error rate.

The importance is used to update the weights of the training examples, as well. Given an example x_i, its corresponding weight at boosting round $j + 1$ is:

$$w_i^{(j+1)} = \frac{w_i^{(j)}}{Z_j} \times \begin{cases} e^{-\alpha_j} & C_j(x_i) \neq y_i \\ e^{\alpha_j} & C_j(x_i) \neq y_i \end{cases}$$

Z_j is the normalization factor, used to ensure that $\sum_i w_i^{(j+1)} = 1$. Using this formula, the weights of examples are increased for those incorrectly classified and decreased for those correctly classified, but the update is influenced by how "good" the classifier is.

The classification is also influenced by the importance of the classifier:

$$C*(x) = \arg\max_y \sum_{j=1}^{T} \alpha_j \delta(C_j(x) = y)$$

This approach penalizes models with poor accuracy; additionally, if any intermediate boosting round produces an error rate higher than 50%, the weights of the examples are reverted to their original value $\frac{1}{l}$, and the resampling procedure is repeated.

A default choice of models used by AdaBoost is **decision stumps**, a decision tree with only the root and two leaf nodes. By themselves, they are very weak learners, but together in an ensemble, they can be very powerful.

13.3 Random Forest

Random forests improve the performances of classifiers by constructing an ensemble of decorrelated decision trees. A key feature of this method is that each base model receives a sample that only selects a subset of the original attributes, usually of dimension $m' \approx \sqrt{m}$, or $m' \approx \log(m)$.

The decision trees used in a random forest are unpruned, as they are allowed to grow to their largest possible size until every leaf is pure. Hence, the base classifiers have lo bias but high variance.

This technique is one of the most accurate learning algorithms available, and is also efficient even on large datasets with thousands of input variables with no need for explicit variable deletion. It also provides an estimate of which variables are most important in classification, and generates an internal unbiased estimate of the generalization error as the forest building progresses.

13.4 Gradient Boosting

The goal of gradient boosting is to find the best hypothesis by expanding a starting function. In the case of regression, the functions are:

$$F_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma) = \mathrm{E}[y]$$

$$F_m(x) = F_{m-1}(x) + \left(\arg\min_{h_m \in H} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right)(x)$$

where h_m is a base learner function, and the loss function is usually Squared Error Loss.

Choosing the best function at every step is a difficult optimization problem, so a steepest descent approach is used. After calculating $F_0(x)$ (which is simply the mean of all the target variable values), the following steps are repeated:

1. The **pseudo-residuals** are calculated as:

$$r_{im} = \left[\frac{\partial L(y_i, F(x_i))}{\partial F_{m-1}(x_i)}\right] \ \forall i = 1, \ldots, n$$

   (that is, the gradient).

2. A base learner (usually a weak learner, e.g., a tree) is fit to the data, using the training set $\{x_i, r_{im}\}_{i=1}^{n}$. This creates a set of **terminal regions** R_{jm} (in the case of trees, these regions correspond to the different leaves).

3. The multiplier values γ_{jm} are calculated as:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{i:x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

   These are the values that minimize the loss function within each leaf.

4. The m^{th} function is updated:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

   where ν is a learning rate, and $1(\cdot)$ is an indicator function.

When using gradient boosting for classification, the predictions are calculated as the log odds for the positive class. To transform it into a probability, the logistic function is used:

$$\frac{e^{log(p)}}{1 + e^{log(p)}}$$

Pseudo-residuals are then calculated as the difference between the observed value and the predicted value. Since leaves will contain a mix of different values, the γ_{jm} values are calculated as:

$$\gamma_{jm} = \frac{\sum_i Residual_i}{\sum_i [p_i(1 - p_i)]}$$

13.4.1 XGBoost

XGBoost (eXtreme Gradient Boosting) is an implementation of regularized Gradient Boosting. The first step in the training phase is to choose a starting value; it can be anything, but by default it is 0.5 (regardless of wheter the problem is of regression or classification). Unlike Gradient Boosting which uses regular regression trees, XGBoost uses a special kind of model that is built as follows:

1. The tree is initialized with a single leaf, containing all the residuals for all data points.

2. For, regression, the **similarity score** is calculated as:

$$Similarity\ Score = \frac{(\sum_i Residual_i)^2}{l + \lambda}$$

   where l is the number of residuals in the leaf, and λ is a regularization term. For classification, the similarity score is calculated as:

$$Similarity\ Score = \frac{(\sum_i Residual_i)^2}{\sum_i (p_i(1 - p_i)) + \lambda}$$

3. Different splits are evaluated, and their similarity score is calculated: the one which improves the gain in similairity is chosen. This continues for each leaf until the desired tree depth is reached (by default, it is 6). Typically, the higher λ is, the lower the similarity scores will be.

4. Finally, the tree is pruned to avoid overfitting. A parameter γ is set as a threshold, such that if a leaf's similarity is less than gamma, it is cut.

13.4.2 LightGBM

LightGBM is another implementation of Gradient Boosting, which has faster training speeds than XGBoost, as well as higher accuracy. It uses a technique called **Histogram-based Gradient Boosting**, which discretizes continuous features into discrete bins (this approach also makes it so that LightGBM requires less memory). It also supports parallel and GPU learning.

When looking at the trees produced by this framework, they tend to show a lot of vertical growth, as opposed to those produced by XGBoost, which instead tend to grow horizontally.

LightGBM uses **Gradient-Based One-Side Sampling** (**GOSS**), which retains instances with larger gradients and samples those with lower gradients (basically focusing less on instances which have low training error). Another technique used is **Exclusive Feature Bundling** (**EFB**), a near lossless method to reduce the number of features, bundling features that are mutually exclusive (i.e., they never take zero values simultaneously) together.

13.4.3 CatBoost

CatBoost (Categorical Boosting) is a library that provides a gradient boosting framework. It uses **target encoding**, replacing each category of a variable with a number

calculated from the distribution of the target labels for that category (e.g., the mean value). It uses **Symmetric Decision Trees**, which are balanced trees with the same splitting condition for all nodes at the same level.

13.4.4 Explainable Boosting Machines

Explainable Boosting Machines (EBM) are a type of Generalized Additive Model (GAM). The model trains one feature at a time, in a round-robin fashion at each iteration. They produce exact explanations about the model's predictions.

Chapter 14

Explainability

Interpretability (or **explainability**) is the degree to which a human can understand the cause of a decision. The higher the interpretability of a model, the easier it is for a person to understand why certain decisions or precisions have been made. Interpretability can be used not only to understand why a model gives a specific prediction, but also to highlight biases in the model or the data.

A **black box** is a model whose internals are either unknown to the observer, or they are known but uninterpretable by humans. Some examples of black box models are deep neural networks, support vector machines, and ensemble methods. Other models are instead interpretable by design, such as decision trees, linear regression, logistic regression, and rule-based models.

Machine learning interpretability can be classified according to various criteria:

- **Intrinsic or post-hoc**: intrinsic interpretability refers to models that are considered interpretable because of their simple structure, such as short decision trees or sparse linear models. Post-hoc interpretability refers to the application of interpretation methods after model training. Post.hoc methods can also be applied to intrinsecally interpretable models.

- **Model-specific or model-agnostic**: model-specific interpretation methods are limited to specific model classes (e.g., the interpretation of regression weights in a linear model is model-specific). Model-agnostic methods can be applied on any machine learning model after the model has been trained. The latter work by analyzing the input and output pairs, since bydefinition they do not have access to the model internals.

- **Local or global**: global methods explain the entire method's behaviour, while local ones explain the prediction of single instances.

Intrinsic interpretability methods are always both global and model specific.

Explanations are provided in different formats depending on the input data type: tabular data uses decision trees, feature importance; image data uses saliency maps, contingency maps; text data uses sentence higlighting, attention-based methods. For all of them, prototypes and counter-exemplars can be used, as well.

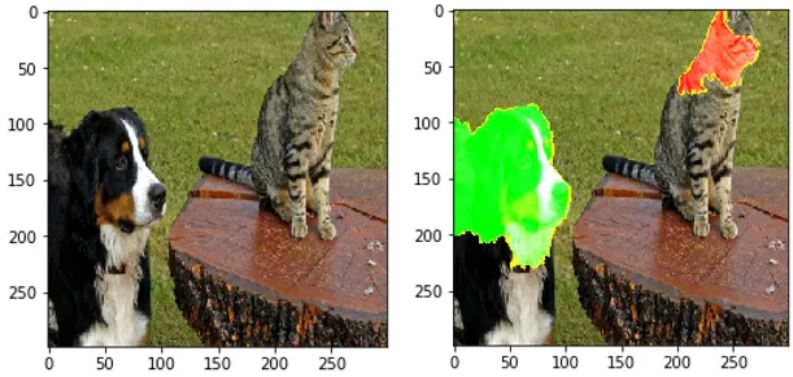


Figure 14.1: Saliency map for a photo of a cat and a dog. The areas highlighted in green represent those which contribute positively to classifying it as "Bernese dog", while the red areas are those which contribute negatively.

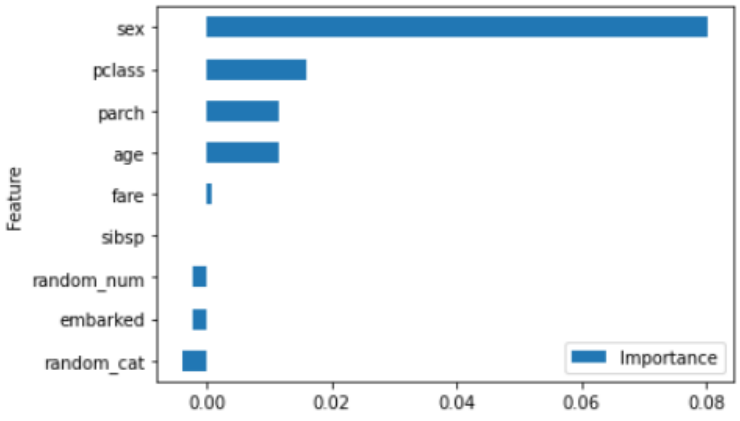


Figure 14.2: A plot showing feature importance for a dataset. A positive importance means the feature strongly contributes to the prediction, while a negative importance means the feature is negatively correlated with the prediction.

14.1 Explanation Methods

14.1.1 TREPAN

TREPAN is a global explainer first developed to extract symbolic representations from trained neural networks (although they are usable on any kind of black box model). TREPAN queries a neural network and builds a decision tree that approximates the network's behaviour using m-of-n rules: m-of-n expressions are Boolean expressions that are specified by an integer threshold m, and a set of n Boolean conditions, and is satisfied if at least m of its n conditions are satisfied. A limitation of regular decision trees is that at lower depths, splits are less significative since they are based on few training instances. TREPAN instead can select as many instances as it wants to build a split condition rule. It learns to predict the label returned by the black box, not the original one.

When selecting a split at a given node, the oracle is given the list of all the previously selected splits on the path from the root to that node. This information is needed to restrict the feature values to consider to build the rule.

14.1.2 LIME

LIME (Local Interpretable Model-agnostic Explanations) is a local explainer which implements surrogate models (a logistic regressor, usually with LASSO or Ridge regularization), trained to approximate the predictions of underlying black box model. LIME works by perturbing the data instance, generating a new dataset composed of perturbed instances and their corresponding predictions by the black box model. An interpretable model is trained on this new dataset, weighted by the proximity of the sampled instances to the instance of interest. This surrogate should be a good approximation of the original model locally, but not globally: this kind of "accuracy" is called **local fidelity**.

When using tabular data, instances are perturbed by changing values of the features, drawing from a normal distribution with mean and standard deviation taken from the feature itself. Each feature will be assigned an importance, describing how much each of them contributes (positively or negatively) to the prediction. For image or text data, the solution is to exclude certain pixels or words. Specifically, images are transformed into vectors of interpretable superpixels expressing presence or absence; a synthetic neighborhood is obtained perturbing these vectors, and the surrogate ends up assigning weights to each superpixel. This is done because perturbing single pixels is not enough to change predictions by much.

14.1.3 LORE

LORE (LOcal Rule-based Explainer) extends LIME using a decision tree as a surrogate, and generating synthetic instances using a genetic procedure that takes into account for instances with the same labels and those with different ones. Local explanations are expressed in the form of pairs, composed by a logic rule, describing a path in the decision tree that explains why the instance has been classified as such, and a set of **counterfacual rules**, explaining which conditions should be changed to obtain a different prediciton. It can be generalized to work on text and image data, using the same data representation as LIME.

14.1.4 SHAP

Shapley values are a method from coalitional game theory which can be used to find how to evenly distribute the "payout" (the prediction) across the "players" (the features). The goal is to explain the difference between the prediction and the average prediction for all instances. The Shapley value is the average marginal contribution of a feature across all possible coalitions. A coalition between features is simulated by randomly selecting another instance from the dataset and using its values for the features not in the coalition, obtaining a new instance. Since this procedure is highly computationally expensive, so an alternative is to compute contributions for only a few samples of the possible coalitions.

SHAP (SHapley Additive exPlanations) is a model-agnostic method based on Shapley values. Its goal is to explain the prediction of a given instance by compuiting the contribution of each feature to the prediction calculating their Shapley values. Players can also be groups of features instead of individual ones. In SHAP, Shapley value explanation is represented as an additive feature attribution method, a linear model:

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i$$

where g is the explanation model, z' the coalition vector, M the maximum coalition size, and ϕ_i is the Shapley value of feature i.

14.1.5 Integrated Gradients

Integrated gradients is a method that computes the gradients of all the points between the input instance and the baseline input (for images, this could be a black image; for text, the zero embedding vector): the intgrated gradients are the cumulation of these

gradients. The result can be visualized using the appropriate format (e.g., saliency maps for images).

14.1.6 Instance-Based Explanations

Instance-based explanation methods select particular instances or generate synthetic instances to explain black box model behaviour. They are mainly local explainers. These methods make sense if the input can be represented in a human understandable way, such as tabular data with few features, images, or short text. Instance-based methods can be divided mainly into two groups:

- **Prototypes**: a selection of representative instances with the same class as the instance under analysis, among which we can distinguish **criticisms** (instances that are not well represented by prototypes), **influential instances** (training points that were the most influential for the training of the black-box model or for the prediction);
- **Counterfactuals**: a selection of represetative instances with a different class than the instance under analysis. A counterfactual explanation represents the smallest change to the feature values that changes the prediction to a specific class (or simply the other class, in a binary task).

Generating Counterfactuals

A naive approach is to use trial-and-error, randomly changing feature values of the instance and stopping when the desired output is obtained. A more sophisticated approach is to define a loss function that considers the instance of interest, the counterfactual, and the desired outcome: the counterfactual explanation that minimizes this loss can be found using an optimization algorithm. There are many different methods, each using a different loss function and optimization algorithm.

One such method defines the loss as:

$$L(x, x', y', \lambda) = \lambda \cdot (\hat{f}(x') - y')^2 + d(x, x')$$

The first term is the squared difference between the prediction of the counterfactual x' and the desired outcome y', and the second term is the distance between the instance under analysis and the counterfactual, defined as a Manhattan distance weighted by the median absolute devation (MAD). λ is a regularization term. By minimizing this loss, we find the closest counterfactual to the instance that changes the prediction. Instead

of selecting a specific value for λ, a tolerance can be set instead, such that the following constraint must be satisfied:

$$|\hat{f}(x') - y'| \leq \epsilon$$

The steps taken to generate counterfactuals are:

- An instance x, a desired outcome y', a tolerance ϵ, and a regularization term λ are chosen;
- A random instance is sampled as initial counterfactual;
- The loss is optiimzed, using that counterfactual as the starting point;
- While $| \cap f(x') - y'| > \epsilon$, λ is increased and the loss is optimized again. The counterfactual that minimizes the loss is returned.
- Steps 2-4 are repeated and a list of counterfactuals is returned (or only the one that minimizes the loss).

Another method, called **DICE** (**Diverse Counterfactual Explanations**), solves an optimization problem with penalization terms that ensure plausibility by similarity and diversity; it returns a set of k plausible and different counterfactuals are calculated as:

$$C(x) = \arg\min_{c_1,\ldots,c_k} \frac{1}{k}\sum_{i=1}^{k} L(f(c_i), y') + \frac{\lambda_1}{k}\sum_{i=1}^{k} d(c_i, x) - \lambda_2 \textit{dpp-diversity}(c_1, \ldots, c_k)$$

where *dpp-diversity* is "diversity via determinantal point processes", a measure of diversity between the counterfactuals.

Finally, counterfactuals can be found using **heuristic strategies**. These strategies are usually more efficient than optimization algorithms, but do not necessarily find solutions that are optimal. The search strategy is designed so that at each iteration, x' is updated with the objective of minimizing a cost function.

An example of a heuristic strategy is **GSG** (**Growing Spheres Generation**), which relies on a generative approach growing a sphere of syntetic instances around x to find the closest counterfactual: it ignores the direction towards which the closest classification boundary might be, so the solution is not always optimal. The algorithm generates observations in spherical layers around the instance, until a potential counterfactual is found.

Another example is **NNCE** (**Nearest-Neighbor Counterfactual Explainer**), which selects counterfactuals as the instances x' most similar to x and with a different label, using a nearest-neighbor search. Candidate counterfactuals are sorted with respect to the distance between x, and the k most similar ones are selected.

Finally, **CBCE** (**Case-Based Counterfactual Explainer**) is a refinement of NNCE, which adopts the notion of **explanation case** (xc). Given a dataset X, an xc is a couple of instances (x, x') such that x and x' are the two most similar instances in X, and they have different labels.

Appendix A

Maximum Likelihood Estimation

Maximum Likelihood Estimation is a method used to determine values for the parameters of a model, such that they maximize the likelihood that the process described by the model produces the observed data. This is done by maximizing a likelihood function, so that the observed data is most probable.

The dataset is modeled as a random sample $x = [x_1, x_2, \ldots, x_n]$ of i.i.d. points, taken from an unknown joint probability distribution, expressed in terms of parameters $\theta = [\theta_1, \theta_2, \ldots, \theta_k]^T$, so that the distribution falls within the set $\{f(\cdot;\theta)|\theta \in \Theta\}$, where Θ is the parameter space. Evaluating the joint density at x is

$$\mathcal{L}_n(\theta) = \mathcal{L}_n(\theta; x) = f_n(x;\theta) = \prod_{i=1}^{n} f_n(x_i;\theta)$$

called the likelihood function. The goal is to find the θ which maximizes it:

$$\theta = \arg\max_{\theta\in\Theta} f_n(x;\theta)$$

Since this maximum is found by differentiation, it is often convenient to use the natural logarithm of the likelihood function, called **log-likelihood**:

$$l(\theta; x) = \ln \mathcal{L}_n(\theta; x)$$

Sometimes there's known estimators for the parameters; for example, when assuming a Gaussian distribution, the μ parameter is estimated as the arithmetic mean of the available observations.

Appendix B

Odds and Log Odds

Given an event with probability p, the **odds** of that even occurring are:

$$odds = \frac{p}{1-p}$$

The **log-odds** of that event occurring are:

$$\ln\left(\frac{p}{1-p}\right)$$

An **odds ratio** is a statistic that quantifies the strength of the association between two events x and y, and is calculated as the ratio of the odds of the to events:

$$odds\ ratio = \frac{\left(\frac{p_x}{1-p_x}\right)}{\left(\frac{p_y}{1-p_y}\right)}$$

For example, consider the following contingency table:

		Has Cancer	
		Yes	No
Has Mutation	Yes	23	117
	No	6	210

We want to know if there is a relationship between the presence of a mutated gene and cancer. The odds ratio is:

$$\frac{\frac{23}{117}}{\frac{6}{210}} = \frac{0.2}{0.03} = 6.88$$

The result tells us that if someone has a mutated gene, the odds of having cancer are 6.88 higher than for those who do not have a mutation.

Bibliography

[1] Andreas Arning, Rakesh Agrawal, and Prabhakar Raghavan. A linear method for deviation detection in large databases. In *KDD*, volume 1141, pages 972–981, 1996.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Christoph Molnar. Interpretable machine learning. `https://christophm.github.io/interpretable-ml-book/`.