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:

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

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 $(CanFly = Y) \land (GivesBirth = N) \rightarrow Bird$ 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:

$$\begin{aligned} r_1 &: (CanFly = Y) \rightarrow Bird \\ r_2 &: (GivesBirth = Y) \wedge (BloodType = W) \rightarrow Mammal \\ r_3 &: (BloodType = C) \rightarrow Reptile \end{aligned}$$

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

1.2 Properties of a Rule Set

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

Mutually Exclusive Rule Set

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

Exhaustive Rule Set

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

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

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

Ordered Rule Set

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

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

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

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

1.3 Building a Rule Set

Rule extraction methods can be either:

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

1.3.1 Direct Methods for Rule Extraction

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

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

Algorithm 1 Sequential covering algorithm.

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

The algorithm always starts with an empty decision list, R, and extracts rules for each class following the ordering specified by their prevalence. The Learn-One-Rule function iteratively extracts all rules for the current class, and all training instances covered by each rule found is removed from E. The rule is then added to the bottom to the rule list, and the loop repeats until the specified stopping criterion is met.

Rule Evaluation

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

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

FOIL's Information Gain

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.2 Indirect Methods for Rule Extraction

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

A rule is generated from each path in the tree. For each rule $r : A \rightarrow y$ in the rule set, alternative rules $r' : A' \leftarrow y$ are considered, where A' is obtained by removing one of the conjuncts in A. The simplified rule with the lowest pessimistic error rate is retained as a replacement if the error rate is also lower than that of the original rule. Eventual duplicates of the new rule are eliminated from the rule set.

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

1.4 Characteristics of Rule Based Models

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

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

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

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

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

Chapter 2

Sequential Pattern Mining

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

Sequence, element, event

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

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

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

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

Subsequence

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

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

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

Support

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

Support Count

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

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

Sequential Pattern Mining

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

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

$$\begin{gathered}
\langle\{i_1\}\{i_1\}\rangle, \langle\{i_1\}\{i_2\}\rangle, \langle\{i_1\}\{i_3\}\rangle, \langle\{i_1 i_2\}\rangle, \langle\{i_1 i_3\}\rangle \\
\langle\{i_2\}\{i_1\}\rangle, \langle\{i_2\}\{i_2\}\rangle, \langle\{i_2\}\{i_3\}\rangle, \langle\{i_2 i_3\}\rangle \\
\langle\{i_3\}\{i_1\}\rangle, \langle\{i_3\}\{i_2\}\rangle, \langle\{i_3\}\{i_3\}\rangle \,,
\end{gathered}$$

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

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

Anti-monotone property

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

Apriori principle

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

2.1 Time Constraints

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

maxspan

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

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

maxgap and mingap

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

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

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

Contiguous Subsequence

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

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

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

Modified Apriori Principle

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

2.2 Generalized Sequential Patterns Algorithm

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

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

2.2.1 Candidate generation

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

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

Algorithm 2 Generalized Sequential Patterns pseudocode.

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

- 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.

Bibliography

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