Aprendizado Descritivo

1 Definição

Aprendizado descritivo se enquadra na seguinte classificação dos métodos de aprendizado de máquina:

Aprendizado preditivo: supervisionado ou não, tem como objetivo predizer dada característica (*target*).

Aprendizado descritivo: supervisionado ou não, tem como objetivo obter uma descrição para os dados.

Contrastando, o aprendizado preditivo pretende prever dados futuros ou desconhecidos, enquando o aprendizado descritivo prentende trazer uma instrospecção sobre os dados. Tal diferença pode se apresentar de forma tênue. Um exemplo são os algorítmos de agrupamento, que podem ser utilizados em ambas formas. Particularmente, o algorítmo *K-means* se apresenta:

Preditivo: *K-means*: $P \to C$

Descritivo: *K-means*: $A \to C$

Em aprendizado preditivo, o *K-means* possui como domínio toda a população, ainda que o treinamento seja realizado apenas com uma amostra. Já no aprendizado descritivo, nos limitamos à amostra, afinal ela constitui o alvo total de análise.

2 Mineração de itens frequentes

Define-se como itens frequentes um conjunto itens que são recorrentes em um conjunto de transações. Considera-se nos conjuntos de dados:

1. Os itens são os elementos do conjunto de variáveis de análise I.
2. Um conjunto $X \subseteq I$ é denominado *itemset*.
3. O conjunto de todos *k-itemsets* é denotado port $I^{(k)}$.
4. A amostra de transações é denominada por T.
5. Cada transação é identificada unicamente por um `tid`.
6. Um conjunto $Y \subseteq T$ é denominado *tidset*.
7. Cada transação consiste de um identificador, e um conjunto de itens: $(\text{tid}, X),\ X \subseteq I$

Formalmente, um conjunto de dados será uma tripla (T, I, D), onde $D \subseteq T \times I$ é uma relação binária:

$$(t, i) \in D \iff \left[i \in X \text{ na transação } (t, X)\right]$$

Uma transação pode **conter** um *itemset*, e tal relação é definida da seguinte forma:

$$X \subseteq t \iff \forall\, i \in X : (t, i) \in D$$

O conjunto de transações que contém um *itemset* X é denominado **extensão** ou **cobertura** de X.
Tal conjunto é definido pela seguinte operação:

$$c : \mathcal{P}(I) \to \mathcal{P}(T)$$
$$c(X) = \big\{ t \in T \mid \forall\, i \in X : (t, i) \in D \big\}$$

Analogamente, o maior conjunto de itens comuns à um *tidset* Y é chamado de **intensão** de Y.

$$i : \mathcal{P}(T) \to \mathcal{P}(I)$$
$$i(Y) = \big\{ x \in I \mid \forall\, t \in Y : (t, x) \in D \big\}$$

Desta forma, podemos representar um conjunto de dados de duas formas:

Horizontal: o conjunto de transações e suas intensões: $\big\{ \big(t, i(\{t\})\big) \mid t \in T \big\}$

Vertical : o conjunto de itens e suas coberturas: $\big\{ \big(x, c(\{x\})\big) \mid x \in I \big\}$

2.1 Metodologia

A identificação de regras de associação, em geral, envolve duas etapas:

1. Mineração de conjuntos de itens frequentes
2. Descoberta de regras de associação

Devido à natureza computacionalmente intensa da primeira etapa, nossos estudos a focam.

O limiar que separa os itens frequentes dos infrequentes é chamado de **suporte mínimo**.
O suporte de um *itemset* é o tamanho de sua cobertura:

$$\sup(X) = \big|c(X)\big|$$

Admite-se também a definição de **suporte relativo**:

$$\mathrm{rsup}(X) = \frac{\big|c(X)\big|}{|T|}$$

2.2 Algoritmos

O espaço de busca do problema é o conjunto potência do conjunto de itens. Se considerarmos a relação de subconjuntos como uma relação de ordem parcial, temos que o espaço de busca é estruturado como um reticulado. Este reticulado pode ser visualizado como um grafo, onde somente as relações diretas são representadas.

$\rightarrow$ Se $A \subseteq B \wedge |A| = |B| - 1$, então existe uma aresta de A para B.

Assim, a mineração de conjunto de itens frequentes é resolvida por uma busca neste reticulado, seja em largura ou em profundidade. De fato, existem abordagens baseadas em ambos métodos.

No entanto, a maioria das abordagens compartilham a mesma estrutura de busca:

1. Identificam candidatos navegando o espaço de busca
2. Computam o suporte desses candidatos, descartando os infrequentes

2.2.1 Algoritmo ingênuo

Um algorítmo ingênuo é definido: enumerar cada *itemset* possível, e verificar no conjunto de dados quais transações contêm esse *itemset*.

- A computação do suporte de um *itemset* requer uma passada sobre o conjunto de dados: $\mathcal{O}(|T|)$
- Verificar se uma dada transação contém um *itemset*: $\mathcal{O}(|I|)$
- Portanto, o custo total de computação do suporte é $\mathcal{O}(|I| \cdot |T|)$
- O espaço de busca, por sua vez, é o conjunto potência de I.

Logo, a complexidade do algoritmo ingênuo é $\mathcal{O}\big(2^{|I|} \cdot |I| \cdot |T|\big)$.

Vale notar que, devido aos seus tamanhos, a memória principal tipicamente não comporta o conjunto de dados. Tal característica agrava fortemente a ineficiência deste algorítimo, onde o componente $\mathcal{O}(|I| \cdot |T|)$ corresponde à passadas no conjunto de dados.

2.2.2 Apriori

O algoritmo Apriori é viabilizado pela propriedade de **anti-monotonicidade** da função suporte:

$$A \subseteq B \implies \sup(A) \geq \sup(B)$$

O Apriori utiliza busca em largura para minerar os padrões. A busca inicia com a identificação dos itens frequentes. Depois, os conjuntos de tamanho k são explorados antes dos imediatamente maiores. Assim como o ingênuo, ele também opera em duas etapas:

1. Geração de candidatos
2. Cálculo do suporte e eliminação dos infrequentes.

Candidatos que diferem em apenas um item são combinados para gerar os próximos candidatos, de tamanho $k + 1$. Imediatamente, os que possuirem algum subconjunto infrequente são descartados. Utilizando este método, os suportes dos candidatos são atualizados com uma única passada no conjunto de dados.

No total, o número de passadas é drasticamente reduzido: $\mathcal{O}(|I|)$

Apesar disso, o algoritmo ainda apresenta problemas:

- Nem sempre a memória primária comporta todos candidatos de um nível, demandados para busca em largura.
- As operações de poda e cálculo do suporte podem ser consideravelmente custosas, mas podem ser atenuadas com estruturas de dados apropriadas.
- A redução do suporte mínimo implica um grande impacto no custo computacional, pois quanto mais profundo o nível, seu tamanho cresce exponencialmente.
- A densidade da base de dados também decorre em um custo maior: transações com mais itens implicam *itemsets* maiores, mais subconjuntos são gerados para a contagem do suporte.

2.2.3 Equivalence Class Transformation

O Eclat tem como proposta eliminar a necessidade de passadas no conjunto de dados para computar o suporte. Para isso, utiliza-se uma representação vertical dos dados, e o fato de que a cobertura da união de dois *itemsets* é a interseção de suas coberturas.

De forma equivalente, a ideia central do algoritmo é manter os *tidsets* em memória principal para computar o suporte dos *itemsets* através de interseções desses conjuntos. Contudo, a memória principal pode não comportar todos os *tidsets*. Assim, é necessário algum mecanismo que possibilite a divisão do espaço de busca em subproblemas independentes.

Esta divisão pode ser feita conforme uma relação de equivalência estabelecida sobre os candidatos. Seja $p : \mathcal{P}(I) \times N \to \mathcal{P}(I)$ uma função prefixo. A seguinte relação é uma relação de equivalência:

$$\theta_k \subseteq \mathcal{P}(I) \times \mathcal{P}(I)$$
$$A \, \theta_k \, B \equiv p(A, k) = p(B, k)$$

Dessa forma, induz-se uma partição dos conjuntos de itens em classes de equivalência, onde todos os elementos compartilham um certo prefixo.

Durante a busca em profundidade, o algoritmo particiona os conjuntos de itens conforme a relação de equivalência e o nível da árvore. O cálculo do suporte no algoritmo se restringe a calcular o tamanho do *tidset*.

Apesar disso, o algoritmo ainda apresenta problemas:

- O tempo de execução depende do cálculo da interseção dos *tidsets*.
- O custo computacional do algoritmo está diretamente relacionado ao tamanho dos *tidsets*.
- O custo de espaço também depende do tamanho. Quanto mais denso o conjunto de dados, mais largos serão os *tidsets*.

2.2.4 dEclat

De forma a atacar o problema de espaço do Eclat, podemos substituir os *tidsets* pela diferença entre os mesmos e os prefixos que os definem para os membros de cada classe. Tal conjunto é denomidado **diffset**. Para um prefixo P e um *itemset* PX, o *diffset* de X é

$$d(PX) = c(P) - c(X)$$

Somente os *diffsets* são armazenados, portanto o suporte não é mais obtido como a cardinalidade desse conjunto. Calculamos o suporte de um *itemset* PXY, obtido a partir de PX e PY, da seguinte forma:

$$\sup(PXY) = \sup(PX) - |d(PXY)|$$

Tal solução passa por computar o diffset de PXY:

$$d(PXY) = d(PY) - d(PX)$$

Em outras palavras, podemos usar os diffsets dos conjuntos base para calcular o diffset do novo candidato.

Essa abordagem se mostra muito eficiente para conjuntos densos. Porém, em conjuntos esparsos, o algoritmo original é a melhor opção.

2.2.5 FP-Growth

O algoritmo FP-Growth procura atacar dois problemas presentes nas abordagens anteriores:

- Repetidas passadas sobre a base de dados.
- Geração de candidatos.

Para isso:

1. Adota-se uma estratégia de busca em profundidade.
2. Adota-se projeções dos dados para mantê-los em memória principal.
3. Utiliza-se uma árvore especial de prefixos denominada FP-Tree.
4. Busca-se os padrões inteiramente através desta árvore, sem necessidade de retorno aos dados.

A FP-Tree constitui uma árvore de prefixos tradicional, adicionada de uma tabela auxiliar de localização. Cada nó contém um item e sua frequência **naquele prefixo**. Portanto, um mesmo item pode constar em vários nós, caso ocorra em prefixos distintos. Já a tabela possui as seguintes colunas:

1. Item: identificador do item, índice da tabela.

2. Frequência: frequência individual de cada item (denota a ordem dos itens).

3. Nós: coleção de referências para os nós da árvore referentes ao item.

Sua construção se dá em duas fases:

1. Computa-se a frequêcia individual dos itens: (Uma passada nos dados)
Itens infrequentes são descartados, pois não podem formar padrões frequentes.

2. Insere-se cada transação, através dos seus itens ordenados: (Outra passada)
Itens são ordenados em ordem decrescente de frequência, sendo os infrequentes filtrados.

Com a FP-Tree construída, inicia a mineração recursiva de padrões.

Para cada item em ordem **crescente**:

1. Constrói-se uma **nova FP-Tree** a partir dos prefixos deste item, desconsiderando o item em questão. É importante calcular as frequências apenas nos prefixos considerados.
2. Desta, descarta-se os itens cuja frequência na árvore é inferior ao suporte mínimo.
3. Caso a árvore constitua um ramo, considera-se este ramo um conjunto, e deste extrai-se o conjunto potência. Caso contrário, repete-se o processo recursivamente para cada item da árvore.
4. Finalmente, acresce-se aos conjuntos extraídos o item da iteração.

Exemplo: (suporte mínimo = 2)

Tabela 1: Base de dados

tid	a	b	c	d	e	f
1	✓		✓	✓		✓
2		✓	✓			
3			✓		✓	✓
4	✓			✓		
5		✓	✓			✓
6	✓		✓			✓

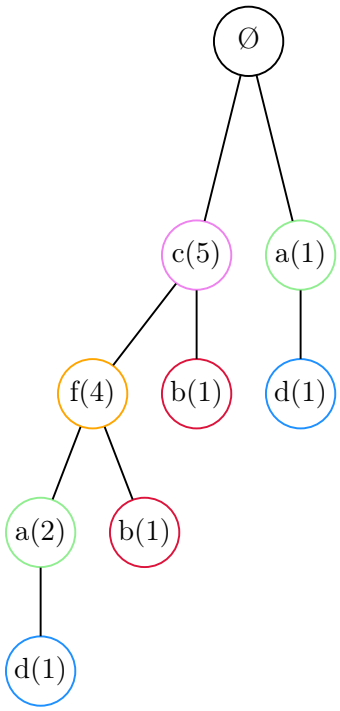
Construção da FP-Tree:

1. Cálculo da frequência:

Item	Frequência	Nós
c	5	Ø
f	4	Ø
a	3	Ø
b	2	Ø
d	2	Ø

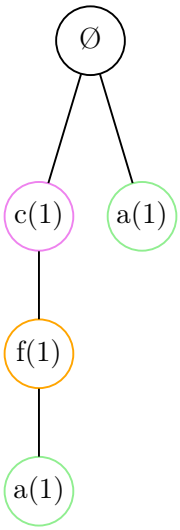
2. Inserção das transações:

Item	Frequência	Nós
c	5	1
f	4	1
a	3	2
b	2	2
d	2	2



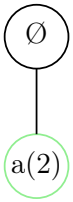
Mineração de padrões a partir do item d:

1. Projeção dos prefixos:



Item	Frequência	Nós
a	2	2
c	1	1
f	1	1

2. Remoção dos infrequentes:

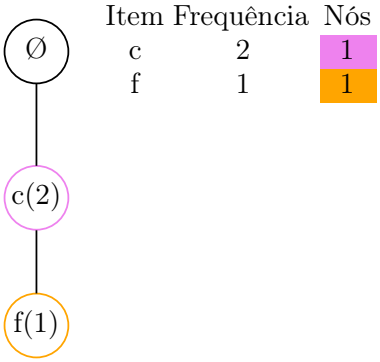


Item	Frequência	Nós
a	2	1

Padrões extraídos: $\{d\}, \{a, d\}$

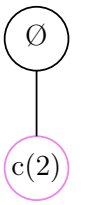
Mineração de padrões a partir do item b:

1. Projeção dos prefixos:



Item	Frequência	Nós
c	2	1
f	1	1

2. Remoção dos infrequentes:

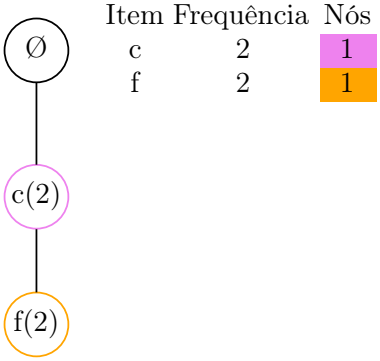


Item	Frequência	Nós
c	2	1

Padrões extraídos: $\{b\}, \{c, b\}$

Mineração de padrões a partir do item a:

1. Projeção dos prefixos:



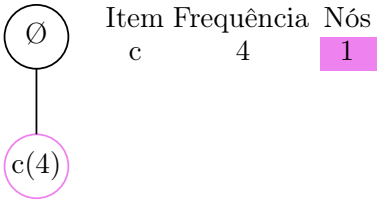
Item	Frequência	Nós
c	2	1
f	2	1

2. Não há infrequentes.

Padrões extraídos: $\{a\}, \{c, a\}, \{a, f\}, \{c, f, a\}$

Mineração de padrões a partir do item f:

1. Projeção dos prefixos:



Item	Frequência	Nós
c	4	1

2. Não há infrequentes.

Padrões extraídos: $\{f\}, \{c, f\}$

Mineração de padrões a partir do item f:

1. Projeção dos prefixos:

Item	Frequência	Nós
-	-	-

2. Não há infrequentes.

Padrões extraídos: $\{c\}$

2.3 Representações compactas

Considerande uma base de dados contendo apenas duas transações:

$$\{ \\ (0, a_1, \ldots, a_{50}), \\ (1, a_1, \ldots, a_{100}) \\ \}$$

Ao considerar um suporte mínimo igual a 2, essa base apresenta uma quantidade exorbitante de *itemsets* frequentes:

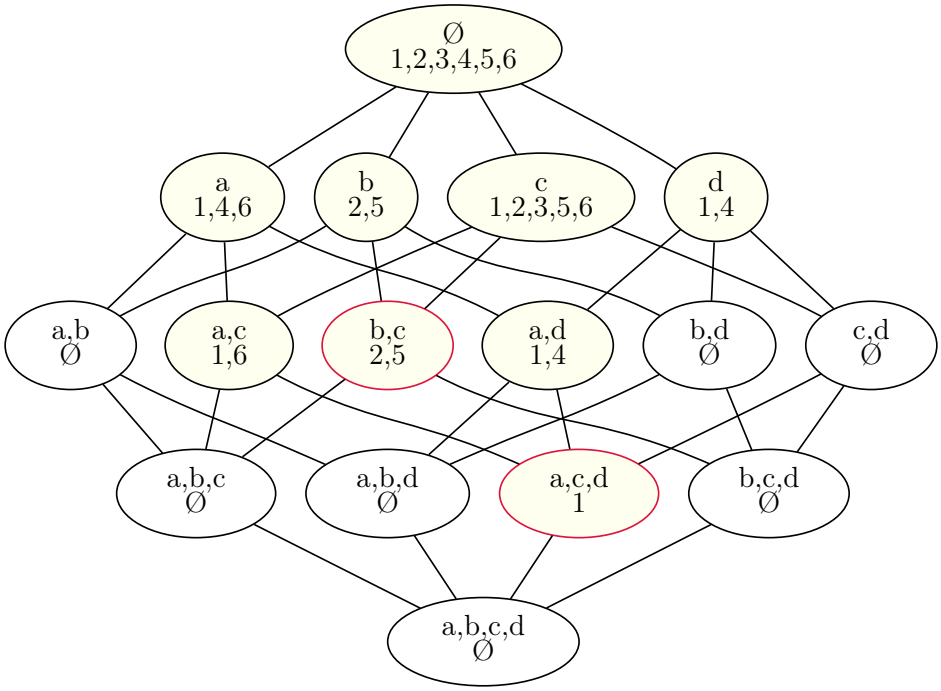
$$\binom{100}{1} + \ldots + \binom{100}{100} = 2^{100} - 1$$

Apesar de ser um caso extremo, é comum encontrar pequenas ocorrências deste padrão em bases de dados maiores. Isso inviabiliza a computação utilizando os métodos descritos anteriormente.

De forma a mitigar este problema, utilizamos representações compactas dos *itemsets* frequentes.

2.3.1 Fronteira da frequência

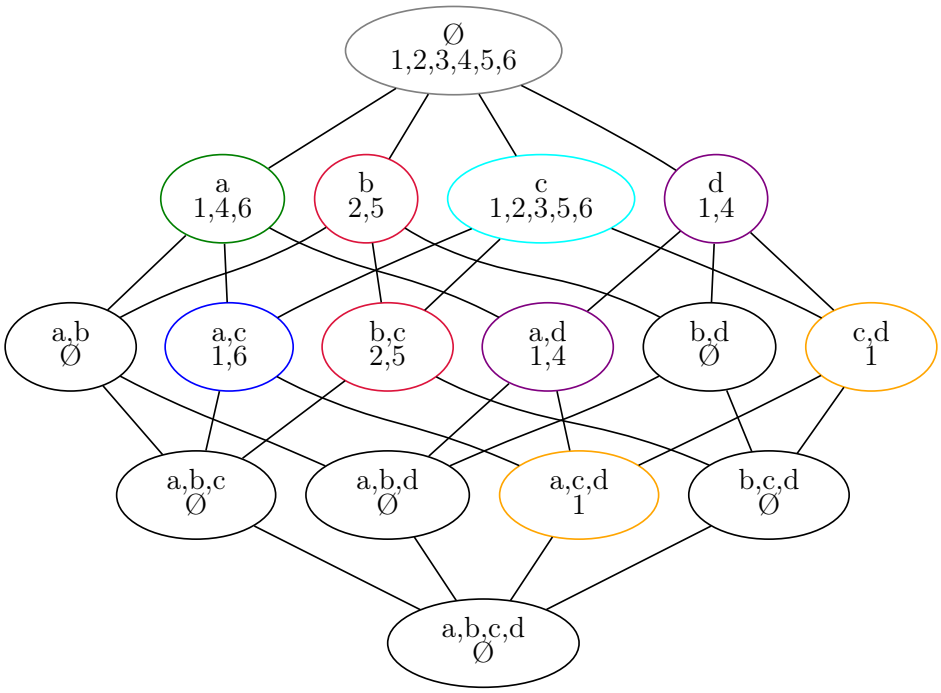
Uma forma de representação compacta denominada *Itemsets* máximos é obtida ao se considerar o limiar entre os *itemsets* frequentes e infrequentes. Os *Itemsets* máximos são os *itemsets* frequentes que não possuem superconjuntos frequentes.



Apesar de ser uma represetação bastante compacta, o cálculo do suporte para os *itemsets* derivados dessa representação não é viável de forma direta, sendo necessária uma nova passada na base de dados.

2.3.2 Classes de equivalência

Uma outra forma de obter representações compactas é particionar os *itemsets* em classes de equivalência, definidas pela cobertura.



Tal consideração permite duas formas de representações compactas:

***Itemsets* fechados:** são os **maiores** *itemsets* das classes de equivalência.
No exemplo:

- $\{a\}$
- $\{c\}$
- $\{a, d\}$
- $\{b, c\}$
- $\{a, c\}$
- $\{a, c, d\}$

***Itemsets* geradores mínimos:** são os **menores** *itemsets* das classes de equivalência.
No exemplo:

- $\{a\}$
- $\{b\}$
- $\{c\}$
- $\{d\}$
- $\{a, c\}$
- $\{c, d\}$

Enquanto os *itemsets* fechados são únicos, os geradores mínimos não necessariamente são. Portanto, a representação utilizando geradores mínimos pode apresentar redundância, não sendo muito compacta em casos extremos.

2.3.3 Algoritmos

Apesar de ser possível embutir tais representações nos algoritmos descritos anteriormente, isso apenas auxiliaria na digestão dos resultados, não apresentado ganho de desempenho computacional algum. De forma a explorar tal possibilidade de ganho computacional, algoritmos específicos se fazem necessários.

MAFIA: *Maximal Frequent Itemset Algorithm*

O algoritmo utiliza uma abordagem *best-first/branch-and-bound* para navegar o reticulado, adotando uma representação vertical dos dados. Além disso, explora a ordem lexicográfica dos itens, bem como a relação de ordem parcial de subconjuntos entre os *itemsets*.

Duas podas são realizadas:

- Reordenamento dinâmico: Ao visitar cada nó na busca em profundidade, seus filhos imediatos são verificados, e os itens que causam infrequência são desconsiderados para a subárvore atual.
- *Parent Equivalence Pruning* (PEP): Caso a cobertura de um dos itens candidatos seja superior à do *itemset* atual, ele é imediatamente adicionado a este *itemset*, restringindo buscas adicionais relativas à este item.

DCI Closed:

O algoritmo utiliza uma abordagem de divisão e conquista no reticulado, adotano uma representação vertical dos dados. Tal abordagem se mostra extremamente interessante na medida que permite a paralelização da execução do algoritmo. Além disso, também explora uma ordem lexicográfica sobre os itens, e sua extensão sobre os *itemsets*.

A ideia central do algoritmo é escalar o reticulado, visitando cada classe de equivalência uma única vez. Somente um candidato de cada classe é avaliado, e novos candidatos são gerados estendendo os conjuntos fechados com itens ainda não investigados.

3 Mineração de sequências frequentes

Define-se sequências como uma lista ordenada de *itemsets* **entre** transações. Desta forma, enquanto *itemsets* são padrões intra-transações, sequências são padrões inter-transações.

Inicialmente, considera-se o conjunto de dados como um conjunto de transações compostas por:

- Identificador de cliente
- *Timestamp* da transação
- Conjunto de itens

Uma sequência de *itemsets* $\alpha = \langle a_1, \ldots, a_n \rangle$ é subsequência de $\beta = \langle b_1, \ldots, b_m \rangle$, denotado por $\alpha \sqsubseteq \beta$, se:

1. Existe uma mapeamento $\varphi : \alpha \to \beta$ — Todos *itemsets* de α estão associados à *itemsets* de β.
2. $\forall\, a \in \alpha, b = \varphi(a) : a \subseteq b$ — Tal associação apresenta uma relação de subconjunto.
3. $\forall\, i < j, b_k = \varphi(a_i), b_l = \varphi(b_j) : k < l$ — A ordem relativa entre α e β é mantida.

A partir da definição de sequência, redefine-se a base de dados como um conjunto dos clientes e suas respectivas sequências de transações. Tal definição constitui uma representação horizontal da base de dados.

$$\mathcal{D} : \{C \times S\}$$

Cliente	Transações
c_1	$s_1 = \langle t_{1,1}, \ldots, t_{1,i} \rangle$
$\vdots$	$\vdots$
c_n	$s_n = \langle t_{n,1}, \ldots, t_{n,j} \rangle$

Onde $t_{k,i}$ é o *itemset* da i-ésima transação do cliente k.

Vale notar que tal representação abdica da informação precisa do *timestamp* de cada transação, mantendo apenas a ordem temporal relativa dos *itemsets*.

Em seguida, definse-se a função suporte, mantendo-se a propriedade do Apriori:

$$\begin{aligned} &\sup : S \to \mathbb{N} \\ &\sup(\alpha) = \left| \{\, c_k \mid \forall\, (c_k, s) \in \mathcal{D} : \alpha \sqsubseteq s \,\} \right| \end{aligned}$$

Ou seja, um cliente c_k suporta uma sequência α se α é uma subsequência da sequência de transações s_k.

Ademais, extende-se os conceitos que temos na mineração de itens frequentes:

Ordem: A ordem de uma sequência α é k (α é uma k-sequência) onde $k = \sum_{\forall a \in \alpha} |a|$

Frequente: Uma sequência α é frequente se seu suporte é maior ou igual ao suporte mínimo.

Máxima: Uma sequência frequente é máxima quando não há supersequências frequentes.

Fechada: Uma sequência frequente é fechada quando não há uma supersequência de mesmo suporte.

3.1 Algoritmos

Os algoritmos tradicionais para mineração de sequências frequentes se baseiam nos algoritmos vistos anteriormente, em particular o Apriori, Eclat e FP-Growth.

3.1.1 *Generalized Sequential Patterns*

Baseado no Apriori, o GSP adota a mesma estratégia de busca em largura, utilizando as sequências de ordem $k-1$ para gerar candidatas de tamanho k. Além disso, utiliza-se a propriedade de anti-monotonicidade da função suporte para executar podas no espaço de busca.

Bem como no Apriori, são executadas diversas passadas na base de dados:

1. Inicialmente, verifica-se o suporte das sequêcias unitárias, descartando-se as infrequentes.
2. Cada par $\langle x \rangle$ e $\langle y \rangle$ de 1-sequências dá origem a duas 2-supersequências:

   I. $\langle xy \rangle$
   II. $\langle x, y \rangle$

   Exceto quando $x = y$, onde apenas a segunda é considerada.
3. Para as sequências não unitárias, gera-se os candidatos da seguinte forma:

   1. Sejam $s_1 = \langle \{i \mid x\} \mid s \mid y \rangle$ e $s_2 = \langle x \mid s \mid \{y \mid j\} \rangle$ sequências de ordem $k-1$.
      Onde s é uma sequência em comum, possivelmente vazia, e x e y são *itemsets*, possivelmente vazios.

      De forma equivalente, s_1 e s_2 diferem respectivamente no primeiro item do primeiro elemento de s_1 e no último item do último elemento de s_2.

   2. Gera-se uma nova sequência: $\langle \{i \mid x\} \mid s \mid \{y \mid j\} \rangle$
4. Após a geração de novos candidatos, seus suportes são calculados e os infrequentes descartados.
5. O algoritmo encerra quando não há mais candidatos frequentes.

3.1.2 *Sequential Pattern Discorvery using Equivalence classes*

Baseado no Eclat, o Spade utiliza também uma representação vertical da base de dados, e opera dividindo o espaço de busca pelo prefixo das sequências.

Primeiramente, define-se como *index-set* de um item o conjunto de clientes e índices em que este item ocorre na respectiva sequência:

$$\mathcal{L}(i) = \left\{ (c_k, \text{indexes}(i, s_k)) \mid \forall\, (c_k, s_k) \in \mathcal{D} \right\}$$

A partir desta definição, apresenta-se uma representação vertical da base de dados como o conjunto de itens e suas *index-set*:

$$\left\{ (i, \mathcal{L}(i)) \mid \forall\, i \in I \right\}$$

Onde o suporte de um item é trivialmente calculado por:

$$\sup_{\text{item}}(i) = \left| \mathcal{L}(i) \right|$$

Define-se as classes de equivalência como as k-sequências que compartilham um prefixo de tamanho $k-1$.

A partir de tais definições, o algoritmo opera gerando novas sequências a partir da junção temporal de duas sequências que pertençam à mesma classe de equivalência.

O *index-set* de uma nova sequência pode ser convenientemente calculado como a interseção dos *index-sets* das sequências geradoras, e desta forma extende-se sua definição para sequências. Considerando que o suporte é definido como a cardinalidade do *index-set*, sua definição também é extendida por consequência.

Utilizando tal método de forma iterativa, o algoritmo encerra quando não houver mais sequências a serem geradas.