The Apriori Algorithm

Association rule learning, the Apriori algorithm and it's implementation

tommyod @ github

Presentation: github.com/tommyod/Efficient-Apriori/blob/master/docs/presentation/apriori.pdf

December 28, 2018

Table of contents

A problem: learning association rules

Motivating example

Example (Learning from transactions)

Consider the following set of *transactions*.

$$\begin{aligned}\{\text{eggs}, \text{bread}, \text{jam}, \text{bacon}\}\\ \{\text{apples}, \text{eggs}, \text{bacon}\}\\ \{\text{bacon}, \text{bread}\}\\ \{\text{ice cream}, \text{bread}, \text{bacon}\}\end{aligned}$$

What interesting information can we infer from this data?
Examples:

- The itemsets $\{\text{bacon}, \text{bread}\}$ and $\{\text{bacon}, \text{eggs}\}$ often appear in the transactions, with counts 3 and 2, respectively.
- The rule $\{\text{bread}\} \Rightarrow \{\text{bacon}\}$ is meaningful in the sense that $P(\text{bacon}|\text{bread}) = 1$.

Formal problem statement

Problem

Given a database $T = \{t_1, t_2, \ldots, t_m\}$*, where the* t_i *are transactions, and a set of items* $I = \{i_1, i_2, \ldots, i_n\}$*, learn meaningful rules* $X \Rightarrow Y$*, where* $X, Y \subset I$*.*

To accomplish this, we need measures of the *meaningfulness* of association rules.

Properties of association rules

Definition (Support)

The *support* of an association rule $X \Rightarrow Y$ is the frequency of which $X \cup Y$ appears in the transactions T, i.e. support$(X \Rightarrow Y) := P(X, Y)$.

- No reason to distinguish between the support of an itemset, and the support of an association rule, i.e. support$(X \Rightarrow Y) = $ support$(X \cup Y)$.
- An important property of support is that support$(\{\text{eggs}, \text{bacon}\}) \leq$ support$(\{\text{bacon}\})$.

More formally, we observe that:

Theorem (Downward closure property of sets)

If $s \subset S$, then support$(s) \geq$ support(S).

Properties of association rules

Definition (Confidence)

The confidence of the association rule $X \Rightarrow Y$ is given by

$$\text{confidence}(X \Rightarrow Y) = P(Y|X) = \frac{P(X, Y)}{P(X)} = \frac{\text{support}(X \Rightarrow Y)}{\text{support}(X)}.$$

Notice the following interesting property.

Example

The confidence of $\{A, B\} \Rightarrow \{C\}$ will always be greater than, or equal to, $\{A\} \Rightarrow \{B, C\}$). By definition we have

$$\frac{\text{support}(\{A, B\} \Rightarrow \{C\})}{\text{support}(\{A, B\})} \geq \frac{\text{support}(\{A\} \Rightarrow \{B, C\})}{\text{support}(\{A\})},$$

where the numerator is equal, and $\text{support}(\{A\}) > \text{support}(\{A, B\})$

Properties of association rules

Definition (Confidence)

The confidence of the association rule $X \Rightarrow Y$ is given by

$$\text{confidence}(X \Rightarrow Y) = P(Y|X) = \frac{P(X,Y)}{P(X)} = \frac{\text{support}(X \Rightarrow Y)}{\text{support}(X)}.$$

Theorem (Downward closure property of rules)

Consider the rule $(X - y) \Rightarrow y$ and $(X - Y) \Rightarrow Y$, where $y \subset Y$. Then

$$\text{confidence}\left((X - y) \Rightarrow y\right) \geq \text{confidence}\left((X - Y) \Rightarrow Y\right)$$

Proof. The numerator is identical, but the denominator has support$(X - y) \leq$ support$(X - Y)$ by the downward closure property of sets.

Examples of support and confidence

Example (Support and confidence of a rule)

Consider again the following set of transactions.

$$\{\text{eggs}, \text{bread}, \text{jam}, \text{bacon}\}$$
$$\{\text{apples}, \text{eggs}, \text{bacon}\}$$
$$\{\text{bacon}, \text{bread}\}$$
$$\{\text{ice cream}, \text{bread}, \text{bacon}\}$$

- The rule $\{\text{bread}\} \Rightarrow \{\text{bacon}\}$ has support $3/4$, confidence 1.
  - Support $3/4$ since $\{\text{bread}, \text{bacon}\}$ appears in 3 of the transactions.
  - Confidence 1 since $\{\text{bread}\}$ appears 3 times, and in 3 of those $\{\text{bacon}\}$ also appears.

A naive algorithm

Example (Naive algorithm for learning rules)

for subsets of every size $k = 1, \ldots, |I|$
    for every subset of size k
        for every split of this subset into $\{X\} \Rightarrow \{Y\}$
            compute support and confidence of the rule
            by counting the support in the transactions

- Fantastic staring point for an algorithm, since it (1) clearly terminates in finite time, (2) is simple to implement and (3) will run reasonably fast on small problem instances.
- Terribly slow on realistic problem instances, since it must check every possible itemset against every transaction.

A solution: the Apriori algorithm

Overview of apriori

- Split the problem into two distinct phases.
  - Finding meaningful (high support) itemsets.
  - Generating meaningful (high confidence) rules.
- **Phase 1**
  - The user specifies a desired *minimum support*.
  - The algorithm exploits the downward closure property, i.e. $\text{support}(S) \leq \text{support}(s)$ if $s \subset S$.
    - No reason to check S if s has low support.
  - Bottom-up approach to subset generation.
- **Phase 2**
  - The user specifies a desired *minimum confidence*.
  - Also exploits the above downward closure property.
  - Bottom-up approach to rule generation.

Phase 1: Generating itemsets (example 1)

Example (Itemset generation via Apriori)

Consider again the following set of transactions.

$$\{\text{eggs}, \text{bread}, \text{jam}, \text{bacon}\}$$
$$\{\text{apples}, \text{eggs}, \text{bacon}\}$$
$$\{\text{bacon}, \text{bread}\}$$
$$\{\text{ice cream}, \text{bread}, \text{bacon}\}$$

- We set the minimum confidence to 50 %.
  - Itemsets of size 1 with desired confidence are $\{\text{bacon}\}$, $\{\text{bread}\}$ and $\{\text{eggs}\}$. They are called *large itemsets* of size 1.
  - From these, we can form $\{\text{bacon}, \text{bread}\}$, $\{\text{bacon}, \text{eggs}\}$ and $\{\text{bread}, \text{eggs}\}$. These are *candidate itemsets* of size 2.
  - Large itemsets of size 2: $\{\text{bacon}, \text{bread}\}$ and $\{\text{bacon}, \text{eggs}\}$.

Phase 1: Generating itemsets (example 2)

Example

Transactions

$\{1, 2, 7, 4\}$
$\{2, 3, 4\}$
$\{1, 6, 3\}$
$\{1, 2, 4, 5\}$

Iteration 1

- Running the algorithm with minimum support 50 %.
- Candidate itemsets of size 1:
  - $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
- Large itemsets of size 1:
  - $\{1\}, \{2\}, \{3\}, \{4\}$

Phase 1: Generating itemsets (example 2)

Example

Transactions

$\{1, 2, 7, 4\}$

$\{2, 3, 4\}$

$\{1, 6, 3\}$

$\{1, 2, 4, 5\}$

Iteration 2

- Running the algorithm with minimum support 50 %.
- Candidate itemsets of size 2:
  - $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$
- Large itemsets of size 2:
  - $\{1, 2\}, \{1, 4\}, \{2, 4\}$

Phase 1: Generating itemsets (example 2)

Example

Transactions

$$\{1,2,7,4\}$$
$$\{2,3,4\}$$
$$\{1,6,3\}$$
$$\{1,2,4,5\}$$

Iteration 3

- Running the algorithm with minimum support 50 %.
- Candidate itemsets of size 3:
  - $\{1,2,4\}$
- Large itemsets of size 3:
  - $\{1,2,4\}$

Phase 1: Pseudocode

Algorithm sketch

Create L_1, a set of large itemsets of size 1

$j = 1$

while L_j is not empty do:

    create every candidate set C_{j+1} from L_j

    prune candidates a priori C_{j+1} (every subset must be in L_j)

    for every transaction $t_i \in T$ do:

        count occurrences of every set in C_{j+1} in t_i

    $j = j + 1$

Iterating through the transactions checking for every possible candidate in C_{j+1} is expensive. Optimizations: choosing good data structures, pruning transactions.

Phase 1: Pseudocode - Details on candidates and pruning

create every candidate set C_{j+1} from L_j
prune candidates a priori C_{j+1} (every subset must be in L_j)

Example Given large itemsets of size 3
$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}$.

- Naive candidates are
  $\{2, 3, 4, 5\}, \{1, 3, 4, 5\}, \{1, 2, 4, 5\}, \{1, 2, 3, 5\}, \{1, 2, 3, 4\}$.
- Apriori-gen candidates are $\{1, 2, 3, 4\}, \{1, 3, 4, 5\}$. Generated efficiently by keeping the itemsets sorted.
- While the itemset $\{1, 2, 3, 4\}$ is kept, $\{1, 3, 4, 5\}$ is discarded since the subset $\{1, 3, 5\} \subset \{1, 3, 4, 5\}$ is not among the large itemsets of size 3 .

The example above is from page 4 in the referenced paper.

Phase 1: Pseudocode - Details on counting occurences

for every transaction $t_i \in T$ do:
  count occurrences of every set in C_{j+1} in t_i

Example

Check if $A = \{1, 3, 7\}$ is a subset of $B = \{1, 2, 3, 5, 7, 9\}$.

- A naive computation checks if every element of A is found in B. This has computational complexity $\mathcal{O}(|A||B|)$, where $|A|$ is the size of A.
- A better approach is to use binary search when B is sorted. The computational complexity becomes $\mathcal{O}(|A| \log_2 |B|)$.
- Using hash tables (e.g. the built-in `set.issubset` in Python), the computational complexity is down to $\mathcal{O}(|A|)$.

For the given example, this resolves to approximately 18, 8 and 3 operations.

Phase 2: Building association rules (example)

- In practice this step is much faster than Phase 1.
- The efficient algorithm exploits the downward closure property.

Example

Consider rules made from $ABCD$. First the algorithm tries to move itemsets of size 1 to the right hand side, i.e. one of $\{\{A\}, \{B\}, \{C\}, \{D\}\}$.

$$\begin{array}{cc} BCD \Rightarrow A & ACD \Rightarrow B \\ ABD \Rightarrow C & ABC \Rightarrow D \end{array}$$

Assume that only $ABC \Rightarrow D$ and $ACD \Rightarrow B$ had high enough confidence. Then the only rule created from $ABCD$ with a size 2 itemset on the right hand side worth considering is $AC \Rightarrow BD$. This is a direct result of the downward closure property.

Recursive function which is not very easy to explain in detail.

The Apriori algorithm on real data

Consider the following data set, with 32.561 rows.

Education	Marital-status	Relationship	Race	Sex	Income	Age
Bachelors	Never-married	Not-in-family	White	Male	$\leq$50K	middle-aged
Bachelors	Married-civ-spouse	Husband	White	Male	$\leq$50K	old
HS-grad	Divorced	Not-in-family	White	Male	$\leq$50K	middle-aged
11th	Married-civ-spouse	Husband	Black	Male	$\leq$50K	old
Bachelors	Married-civ-spouse	Wife	Black	Female	$\leq$50K	young
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Masters	Married-civ-spouse	Wife	White	Female	$\leq$50K	middle-aged
9th	Married-spouse-absent	Not-in-family	Black	Female	$\leq$50K	middle-aged
HS-grad	Married-civ-spouse	Husband	White	Male	$>$50K	old
Masters	Never-married	Not-in-family	White	Female	$>$50K	middle-aged

The data may be found at `https://archive.ics.uci.edu/ml/datasets/adult`.

The Apriori algorithm on real data

Some rules are obvious in retrospect:

$$\begin{aligned}
\{\text{Husband}\} &\Rightarrow \{\text{Male}\} \\
\{\leq 50\text{K}, \text{Husband}\} &\Rightarrow \{\text{Male}\} \\
\{\text{Husband}, \text{middle-aged}\} &\Rightarrow \{\text{Male}, \text{Married-civ-spouse}\}
\end{aligned}$$

Some are more interesting:

$$\begin{aligned}
\{\text{HS-grad}\} &\Rightarrow \{\leq 50\text{K}\} \\
\{\leq 50\text{K}, \text{young}\} &\Rightarrow \{\text{Never-married}\} \\
\{\text{Husband}\} &\Rightarrow \{\text{Male}, \text{Married-civ-spouse}, \text{middle-aged}\}
\end{aligned}$$

The meaningfulness of a rule may be measured by *confidence*, *lift* and *conviction*.

A practical matter: writing a Python implementation

Overview of workflow

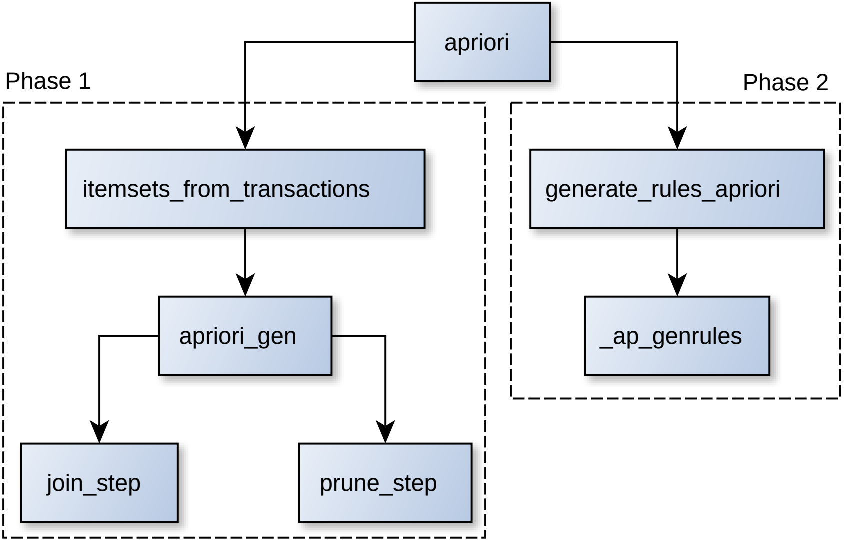
- Write simple functions first, i.e. the building blocks (e.g. pruning)
- Add doctests and unit tests (e.g. examples from paper)
- Implement a naive, but correct algorithm
- Implement an asymptotically fast algorithm
- Test the preceding two implementations against each other
- Optimize implementation by profiling the code (find bottlenecks)

Understand $\rightarrow$ Naive algorithm $\rightarrow$ Asymptotically fast $\rightarrow$ Further optimizations

Software testing

- Unit tests
  - Test a simple function $f(x_i) = y_i$ for known cases $i = 1, 2, \ldots$
  - Doubles as documentation when writing *doctests* in Python
- Property tests
  - Fix a property, i.e. $f(a, b) = f(b, a)$ for every a, b
  - Generate many random inputs a, b to make sure the property holds
- Testing against `R`, Wikipedia, etc
  - Generate some inputs and test against the `arules` package

Software structure



Software found at `https://github.com/tommyod/Efficient-Apriori`.

Summary and references

Summary and references

The Apriori algorithm discovers frequent itemsets in phase 1, and meaningful association rules in phase 2. Both phases employ clever bottom-up algorithms. By application of the downward closure property of itemsets (support) and rules (confidence), candidates may be pruned prior to expensive computations.

- The Python implementation
  - `github.com/tommyod/Efficient-Apriori`
- The original paper
  - Agrawal et al, *Fast Algorithms for Mining Association Rules*, 1994 `http://www.cse.msu.edu/~cse960/Papers/MiningAssoc-AgrawalAS-VLDB94.pdf`