Efficient-Apriori Documentation

Release 2.0.3

tommyod

Feb 02, 2023

CONTENTS

An efficient pure Python implementation of the Apriori algorithm.

CHAPTER ONE

OVERVIEW

An efficient pure Python implementation of the Apriori algorithm. Created for Python 3.6 and 3.7.

The apriori algorithm uncovers hidden structures in categorical data. The classical example is a database containing purchases from a supermarket. Every purchase has a number of items associated with it. We would like to uncover association rules such as *{bread, eggs}* -> *{bacon}* from the data. This is the goal of association rule learning, and the Apriori algorithm is arguably the most famous algorithm for this problem. This project contains an efficient, well-tested implementation of the apriori algorithm as descriped in the original paper by Agrawal et al, published in 1994.

CHAPTER TWO

INSTALLATION

The package is distributed on PyPI. From your terminal, simply run the following command to install the package.

```
$ pip install efficient-apriori
```

Notice that the name of the package is `efficient-apriori` on PyPI, while it's imported as `import efficient_apriori`.

CHAPTER THREE

A MINIMAL WORKING EXAMPLE

Here's a minimal working example. Notice that in every transaction with *eggs* present, *bacon* is present too. Therefore, the rule *{eggs}* -> *{bacon}* is returned with 100 % confidence.

```python
from efficient_apriori import apriori
transactions = [('eggs', 'bacon', 'soup'),
                ('eggs', 'bacon', 'apple'),
                ('soup', 'bacon', 'banana')]
itemsets, rules = apriori(transactions, min_support=0.5,  min_confidence=1)
print(rules)  # [{eggs} -> {bacon}, {soup} -> {bacon}]
```

See the API documentation for the full signature of `apriori()`. More examples are included below.

CHAPTER

FOUR

MORE EXAMPLES

4.1 Filtering and sorting association rules

It's possible to filter and sort the returned list of association rules.

```python
from efficient_apriori import apriori
transactions = [('eggs', 'bacon', 'soup'),
                ('eggs', 'bacon', 'apple'),
                ('soup', 'bacon', 'banana')]
itemsets, rules = apriori(transactions, min_support=0.2,  min_confidence=1)

# Print out every rule with 2 items on the left hand side,
# 1 item on the right hand side, sorted by lift
rules_rhs = filter(lambda rule: len(rule.lhs) == 2 and len(rule.rhs) == 1, rules)
for rule in sorted(rules_rhs, key=lambda rule: rule.lift):
  print(rule) # Prints the rule and its confidence, support, lift, ...
```

CHAPTER

FIVE

CONTRIBUTING

You are very welcome to scrutinize the code and make pull requests if you have suggestions for improvements. Your submitted code must be PEP8 compliant, and all tests must pass. See tommyod/Efficient-Apriori on GitHub for more information.

CHAPTER

SIX

API DOCUMENTATION

Although the Apriori algorithm uses many sub-functions, only three functions are likely of interest to the reader. The `apriori()` returns both the itemsets and the association rules, which is obtained by calling `itemsets_from_transactions()` and `generate_rules_apriori()`, respectively. The rules are returned as instances of the `Rule` class, so reading up on it's basic methods might be useful.

6.1 Apriori function

`efficient_apriori.`**`apriori`**(*transactions: Iterable[set | tuple | list]*, *min_support: float = 0.5*, *min_confidence: float = 0.5*, *max_length: int = 8*, *verbosity: int = 0*, *output_transaction_ids: bool = False*)

The classic apriori algorithm as described in 1994 by Agrawal et al.

The Apriori algorithm works in two phases. Phase 1 iterates over the transactions several times to build up itemsets of the desired support level. Phase 2 builds association rules of the desired confidence given the itemsets found in Phase 1. Both of these phases may be correctly implemented by exhausting the search space, i.e. generating every possible itemset and checking it's support. The Apriori prunes the search space efficiently by deciding apriori if an itemset possibly has the desired support, before iterating over the entire dataset and checking.

Parameters

- **`transactions`** (*`list of transactions (sets/tuples/lists). Each element in`*) – the transactions must be hashable.
- **`min_support`** (*`float`*) – The minimum support of the rules returned. The support is frequency of which the items in the rule appear together in the data set.
- **`min_confidence`** (*`float`*) – The minimum confidence of the rules returned. Given a rule X -> Y, the confidence is the probability of Y, given X, i.e. P(Y|X) = conf(X -> Y)
- **`max_length`** (*`int`*) – The maximum length of the itemsets and the rules.
- **`verbosity`** (*`int`*) – The level of detail printing when the algorithm runs. Either 0, 1 or 2.
- **`output_transaction_ids`** (*`bool`*) – If set to true, the output contains the ids of transactions that contain a frequent itemset. The ids are the enumeration of the transactions in the sequence they appear.

Examples

```
>>> transactions = [('a', 'b', 'c'), ('a', 'b', 'd'), ('f', 'b', 'g')]
>>> itemsets, rules = apriori(transactions, min_confidence=1)
>>> rules
[{a} -> {b}]
```

6.2 Itemsets function

`efficient_apriori.`**`itemsets_from_transactions`**(*transactions: Iterable[set | tuple | list], min_support: float, max_length: int = 8, verbosity: int = 0, output_transaction_ids: bool = False*)

Compute itemsets from transactions by building the itemsets bottom up and iterating over the transactions to compute the support repedately. This is the heart of the Apriori algorithm by Agrawal et al. in the 1994 paper.

Parameters

- **`transactions`** (*`a list of itemsets (tuples/sets/lists with hashable entries)`*) –
- **`min_support`** (*`float`*) – The minimum support of the itemsets, i.e. the minimum frequency as a percentage.
- **`max_length`** (*`int`*) – The maximum length of the itemsets.
- **`verbosity`** (*`int`*) – The level of detail printing when the algorithm runs. Either 0, 1 or 2.
- **`output_transaction_ids`** (*`bool`*) – If set to true, the output contains the ids of transactions that contain a frequent itemset. The ids are the enumeration of the transactions in the sequence they appear.

Examples

```
>>> # This is an example from the 1994 paper by Agrawal et al.
>>> transactions = [(1, 3, 4), (2, 3, 5), (1, 2, 3, 5), (2, 5)]
>>> itemsets, _ = itemsets_from_transactions(transactions, min_support=2/5)
>>> itemsets[1] == {(1,): 2, (2,): 3, (3,): 3, (5,): 3}
True
>>> itemsets[2] == {(1, 3): 2, (2, 3): 2, (2, 5): 3, (3, 5): 2}
True
>>> itemsets[3] == {(2, 3, 5): 2}
True
```

6.3 Association rules function

`efficient_apriori.`**`generate_rules_apriori`**(*itemsets: Dict[int, Dict[tuple, int]]*, *min_confidence: float*, *num_transactions: int*, *verbosity: int = 0*)

Bottom up algorithm for generating association rules from itemsets, very similar to the fast algorithm proposed in the original 1994 paper by Agrawal et al.

The algorithm is based on the observation that for {a, b} -> {c, d} to hold, both {a, b, c} -> {d} and {a, b, d} -> {c} must hold, since in general conf({a, b, c} -> {d}) >= conf({a, b} -> {c, d}). In other words, if either of the two one-consequent rules do not hold, then there is no need to ever consider the two-consequent rule.

Parameters

- **`itemsets`** (*`dict of dicts`*) – The first level of the dictionary is of the form (length, dict of item sets). The second level is of the form (itemset, count_in_dataset)).
- **`min_confidence`** (*`float`*) – The minimum confidence required for the rule to be yielded.
- **`num_transactions`** (*`int`*) – The number of transactions in the data set.
- **`verbosity`** (*`int`*) – The level of detail printing when the algorithm runs. Either 0, 1 or 2.

Examples

```
>>> itemsets = {1: {('a',): 3, ('b',): 2, ('c',): 1},
...             2: {('a', 'b'): 2, ('a', 'c'): 1}}
>>> list(generate_rules_apriori(itemsets, 1.0, 3))
[{b} -> {a}, {c} -> {a}]
```

6.4 Rule class

class `efficient_apriori.`**`Rule`**(*lhs: tuple*, *rhs: tuple*, *count_full: int = 0*, *count_lhs: int = 0*, *count_rhs: int = 0*, *num_transactions: int = 0*)

A class for a rule.

Attributes

`confidence`
The confidence of a rule is the probability of the rhs given the lhs.

`conviction`
The conviction of a rule X -> Y is the ratio P(not Y) / P(not Y | X).

`lift`
The lift of a rule is the ratio of the observed support to the expected support if the lhs and rhs were independent.If X -> Y, then the lift is given by the fraction P(X and Y) / (P(X) * P(Y)).

`rpf`
The RPF (Rule Power Factor) is the confidence times the support.

`support`
The support of a rule is the frequency of which the lhs and rhs appear together in the dataset.

INDEX