Itemset Mining Documentation

Release 0.2.2

Zax Rosenberg

Apr 23, 2022

CONTENTS

Implements itemset mining algorithms.

CHAPTER

ONE

ALGORITHMS

1.1 High-utility itemset mining (HUIM)

HUIM generalizes the problem of frequent itemset mining (FIM) by considering item values and weights. A popular application of HUIM is to discover all sets of items purchased together by customers that yield a high profit for the retailer. In such a case, item values would show not just that a load of bread is in a basket, but how many there are; and weights would include the profit from a loaf of bread.

More technically, HUIM requires transactions in the transactions "database" to have internal utilities (i.e. item values) associated with each item in each transaction and a "database" of external utilities for each item (i.e. weights).

Algorithm	Class	How to Cite
	itemset_mining.two_phase_huim.TwoPhase	

- Includes max length support

CHAPTER

TWO

ROADMAP (HIGH TO LOW PRIORITY):

- **Address low-correlation HUIs with one of bond, all-confidence, or affinity.** Itemsets that are high utility, but where the items aren't correlated can be misleading for making marketing decisions. E.g. if an itemset of a TV and a pen is a HUI, it's likely just because the TV is expensive, not because it's an interesting pattern.
- **Add *average* utility measure support**, for easier, more intuitive minutil
- **Support discount strategies** via a discount strategy table and upgraded external utilities table.
- **Add top-k HUI support.**
- **Support identifying periodic high utility itemsets.** This allows detection of purchase patterns among high-utility itemsets to allow cross-promotions to customers who buy sets of items periodically.
- **Support items' on-shelf time.** Ignmoring on-shelf time will biat toward items that have more shelf time, since they have more chance to generate higher utility.
- **Allow incremental transaction updates** without rerunning everything.
- **Support concise HUI itemsets, specifically closed form.** This allows the algorithm to be more efficient, only showing longer itemsets, which may be the most interesting ones (correlation issues aside).

2.1 Installation:

```
pip install itemset-mining
```

2.2 Example:

```
>>> from operator import attrgetter
>>> from itemset_mining.two_phase_huim import TwoPhase
>>> transactions = [
...     [("Coke 12oz", 6), ("Chips", 2), ("Dip", 1)],
...     [("Coke 12oz", 1)],
...     [("Coke 12oz", 2), ("Chips", 1), ("Filet Mignon 1lb", 1)],
...     [("Chips", 1)],
...     [("Chips", 2)],
...     [("Coke 12oz", 6), ("Chips", 1)]
... ]
>>> # ARP for each item
>>> external_utilities = {
...     "Coke 12oz": 1.29,
...     "Chips": 2.99,
```

(continues on next page)

(continued from previous page)

```
...     "Dip": 3.49,
...     "Filet Mignon 1lb": 22.99
... }
>>> # Minimum dollar value generated by an itemset we care about across all␣
↪transactions
>>> minutil = 20.00

>>> hui = TwoPhase(transactions, external_utilities, minutil)
>>> result = hui.get_hui()
>>> sorted(result, key=attrgetter('itemset_utility'), reverse=True)
...
[HUIRecord(items=('Chips', 'Coke 12oz'), itemset_utility=30.02),
 HUIRecord(items=('Chips', 'Coke 12oz', 'Filet Mignon 1lb'), itemset_utility=28.56),
 HUIRecord(items=('Chips', 'Filet Mignon 1lb'), itemset_utility=25.979999999999997),
 HUIRecord(items=('Coke 12oz', 'Filet Mignon 1lb'), itemset_utility=25.57),
 HUIRecord(items=('Filet Mignon 1lb',), itemset_utility=22.99),
 HUIRecord(items=('Chips',), itemset_utility=20.93)]
```

2.2.1 Changelog

Changed

- Deduplicate `HUIRecords` in results when itemset length >= 3

Changed

- Hotfix missing VERSION file preventing library import

Added

- Support for generator transaction inputs to Two-Phase HUIM algorithm **Changed**
- Fix examples to pass doctest

Added

- Added Two-Phase HUIM algorithm

2.2.2 Installation

```
pip install itemset-mining
```

Note that the latest version of the library will only work toward being compatible with newer version of Python. This is to reduce the technical debt of maintaining the project and Python itself. The support policty will roughly follow Numpy's community version support recommendation in NEP 29. See the table below for officially supported versions:

Bulwark	Python
0.1.0	>=3.6

2.2.3 API

itemset_mining.two_phase_huim

itemset_mining.two_phase_huim

Classes

`CandidateHUIRecord`	alias of `itemset_mining.two_phase_huim.HUIRecord`
`HUIRecord`(items, itemset_utility)	
`TwoPhase`(transactions, external_utilities, ...)	

Example

2.2.4 How to Contribute

First off, thank you for considering contributing! It's thanks to people like you that we continue to have a high-quality, updated and documented tool.

There are a few key ways to contribute:

1. Writing new code
2. Writing tests
3. Writing documentation
4. Supporting fellow developers on StackOverflow.com.

No contribution is too small! Please submit as many fixes for typos and grammar bloopers as you can!

Regardless of which of these options you choose, this document is meant to make contribution more accessible by codifying tribal knowledge and expectations. Don't be afraid to ask questions if something is unclear!

Workflow

1. Set up Git and a GitHub account
2. Fork and clone the repo. Read more about forking workflows.
3. Set up a development environment.
4. Create a feature branch. Pull requests should be limited to one change only, where possible. Contributing through short-lived feature branches ensures contributions can get merged quickly and easily.
5. Rebase on master and squash any unnecessary commits. We do not squash on merge, because we trust our contributors to decide which commits within a feature are worth breaking out.
6. Always add tests and docs for your code. This is a hard rule; contributions with missing tests or documentation can't be merged.
7. Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.

8. Once you've addressed review feedback, make sure to bump the pull request with a short note, so we know you're done.

Each of these abbreviated workflow steps has additional instructions in sections below.

Development Practices and Standards

- Obey follow PEP-8 and Google's docstring format.
  - The only exception to PEP-8 is that line length can be up to 100 characters.
- Use underscores to separate words in non-class names. E.g. `n_samples` rather than `nsamples`.
- Don't ever use wildcard imports (`from module import *`). It's considered to be a bad practice by the official Python recommendations. The reasons it's undesirable are that it pollutes the namespace, makes it harder to identify the origin of code, and, most importantly, prevents using a static analysis tool like pyflakes to automatically find bugs.
- Any new module, class, or function requires units tests and a docstring. Test-Driven Development (TDD) is encouraged.
- Don't break backward compatibility. In the event that an interface needs redesign to add capability, a deprecation warning should be raised in future minor versions, and the change will only be merged into the next major version release.
- Semantic line breaks are encouraged.

Set up a Development Environment

For your local development version of Python it's recommended to use the lowest version this library supports. to ensure newer features aren't accidentally used.

Within your virtual environment, you can easily install an editable version of this library along with its tests and docs requirements with:

```
pip install -e '.[dev]'
```

At this point you should be able to run/pass tests and build the docs:

```
python -m pytest

cd docs
make html
```

To avoid committing code that violates our style guide, we strongly advise you to install pre-commit hooks, which will cause your local commit to fail if our style guide was violated:

```
pre-commit install
```

You can also run them anytime (as our tox does) using:

```
pre-commit run --all-files
```

You can also use tox to run CI in all of the appropriate environments locally, as our cloud CI will:

```
tox
# or, use the -e flag for a specific environment. For example:
tox -e py36
```

Rebase on Master and Squash

If you are new to rebase, there are many useful tutorials online, such as Atlassian's. Feel free to follow your own workflow, though if you have an default git editor set up, interactive rebasing is an easy way to go about it:

```
git checkout feature/<feature_name_in_snake_case>
git rebase -i master
```

Create a Pull Request to the master branch

Create a pull request to the master branch. Tests will be be triggered to run via Travis CI. Check that your PR passes CI, since it won't be reviewed for inclusion until it passes all steps.

For Maintainers

Steps for maintainers are largely the same, with a few additional steps before releasing a new version:

- Update version in itemset_mining/VERSION, which updates three spots: setup.py, itemset*mining/*_init__.py, and docs/conf.py.
- Update the CHANGELOG.md and the main README.md (as appropriate).
- Rebuild the docs in your local version to verify how they render using:

```
pip install -e ".[dev]"
cd docs
make html
```

- Test distribution using TestPyPI with Twine:

```
# Installation
python3 -m pip install --user --upgrade setuptools wheel
python3 -m pip install --user --upgrade twine

# Build/Upload dist and install library
python3 setup.py sdist bdist_wheel
python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
pip install itemset-mining --index-url https://test.pypi.org/simple/
```

- Releases are indicated using git tags. Create a tag locally for the appropriate commit in master, and push that tag to GitHub. Travis's CD is triggered on tags within master:

```
git tag -a v<#.#.#> <SHA-goes-here> -m "itemset-mining version <#.#.#>"
git push origin --tags
```

PYTHON MODULE INDEX

i

INDEX

I