



Knowledge Discovery in Databases with Exercises Summer Semester 2025

Submission 1: Frequent Patterns

About this Assignment

Throughout the course of this assignment, you will independently implement the two methods, **Apriori** (Task 1) and **FP-growth** (Task 2). For this purpose, a basic code skeleton, several helper classes, and some test cases are provided to you.

Key Data

- **Max. Group Size:** 3
- **Max. Points:** 40
- **Estimated Workload:** 4 - 6 hours

How to Work on the Assignment

To start working on the assignment, you'll need to accept the assignment via GitHub Classroom by clicking the provided link. This will set up a new GitHub repository for your group, packed with all the necessary files for the assignment. If you're joining an existing group, it'll add you to that group's repository.¹

Once that's done, you have two main options for working on your assignment. You can clone the repository² to your local machine by navigating to **Code** → **Local**, which allows you to work directly from your computer. Alternatively, you might prefer using GitHub Codespaces by selecting **Code** → **Codespaces** for a virtual online environment, complete with the ability to run Python through the **Terminal** provided.

Whichever method you choose, it's crucial to commit and push your changes back to the repository to submit your solution². After your submission, GitHub Actions takes over to automatically grade your solution and provide feedback. You'll find this feedback in the **Actions** tab of your repository. If you didn't receive full points, you can improve your solution and push the changes back to the repository to trigger a reevaluation.

¹Each student must join individually. You can join groups while accepting an assignment.

²If you're unfamiliar with Git or GitHub, check out this helpful guide: <https://github.com/git-guides/>

How to Prepare the Transfer the Points to StudOn

In addition to joining the GitHub Classroom, you also need to register your GitHub username on StudOn. This is necessary to transfer the points you've earned on GitHub to StudOn. To do this, enter your GitHub username in **Submission 1 - GitHub Username**. Make sure to enter your username correctly, as otherwise, the points cannot be transferred.

After submission deadline, the points you've earned on GitHub will be transferred to StudOn. This process is not immediate and may take a few days. If you have any questions or issues, please contact us via the StudOn forum.

Restrictions

Within the scope of your implementation, you are not permitted to modify the helper classes, the test cases, or the provided GitHub Actions.

This will be checked on a random basis, and any attempt to do so will result in zero points for the involved group, similar to the consequences of plagiarism.

Task 1: Apriori

Apriori is a classic algorithm for frequent itemset mining over transactional databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger itemsets as long as those itemsets appear sufficiently often in the database.

Task 1.1

At the beginning of Apriori, the identification of 1-itemsets is paramount.

Open `apriori.py` in your repository and implement the `_generate_one_itemsets`, which generates all 1-itemsets for a given dataset:

```
1 def _generate_one_itemsets(self, dataset: Dataset) -> Set[Itemset]:
2     """
3     Generate all 1-itemsets for the given dataset.
4
5     Parameters:
6     dataset (Dataset): The dataset for which the 1-itemsets should be generated.
7
8     Returns:
9     Set[Itemset]: A set containing all 1-itemsets that are contained in the dataset.
10    """
11    # TODO
```

Make sure that you expect a `Dataset` and return a `Set[Itemset]`³.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_generate_one_itemsets.py
```

Task 1.2

After the 1-itemsets have been identified, the next step is to count the occurrences of these itemsets in the dataset.

Complete the function `_count_occurrences_of_itemsets`, which counts the occurrences of all given itemsets in the dataset:

```
1 def _count_occurrences_of_itemsets(
2     self, dataset: Dataset, itemsets: Set[Itemset]
3 ) -> ItemsetsWithOccurrenceCounts:
4     """
5     Count the occurrences of the given itemsets in the dataset.
6
7     Parameters:
8     dataset (Dataset): The dataset for which the itemset occurrences should be counted.
9     itemsets (Set[Itemset]): The itemsets for which the occurrences should be counted.
10    The itemsets do not need to be present in the dataset.
11
12    Returns:
13    ItemsetsWithOccurrenceCounts: A dictionary containing the itemsets as keys and
14    their occurrence counts as values.
15    """
16    # TODO
```

³Hint: `Itemset` and `Database` are helper classes that can be found in the `classes/` folder.

Expect that the input consists of a `Dataset` and a `Set[Itemset]`. The method should return an instance of `ItemsetsWithOccurrenceCounts`.

Also be aware that the method should be able to count the occurrences of itemsets with any length, not just 1-itemsets.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_count_occurrences_of_itemsets.py
```

Task 1.3

After counting occurrences, it is necessary in Apriori to prune all itemsets falling below the minimum support threshold.

Complete the function `_prune_itemsets_below_min_support`, which prunes all itemsets that do not meet the minimum support threshold:

```
1 def _prune_itemsets_below_min_support(  
2     self,  
3     itemsets_with_occurrence_counts: ItemsetsWithOccurrenceCounts,  
4 ) -> Set[Itemset]:  
5     """  
6     Prune itemsets that are below the minimum support threshold.  
7  
8     Parameters:  
9     itemsets_with_occurrence_counts (ItemsetsWithOccurrenceCounts): A dictionary containing  
10    the itemsets as keys and their occurrence counts as values.  
11  
12    Returns:  
13    Set[Itemset]: A set containing all itemsets that are considered frequent.  
14    """  
15    # TODO
```

The input consists of an `ItemsetsWithOccurrenceCounts`. The (absolute) minimum support is a member variable of the Apriori object and can therefore be accessed via `self.min_support`. You have to return a `Set[Itemset]`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_prune_itemsets_below_min_support.py
```

Task 1.4

The last missing step in the Apriori algorithm is to generate the candidate itemsets for the next iteration.

Complete the function `_generate_candidate_itemsets`, which generates the candidate itemsets for the next iteration:

```

1 def _generate_candidate_itemsets(
2     self, frequent_itemsets: Set[Itemset]
3 ) -> Set[Itemset]:
4     """
5     Generate length-k+1 candidate itemsets based on the given frequent itemsets.
6     k is the length of the longest frequent itemset.
7
8     Parameters:
9     frequent_itemsets (Set[Itemset]): A set containing all frequent itemsets.
10
11     Returns:
12     Set[Itemset]: A set containing all length-k+1 candidate itemsets.
13     """
14
15     # If there are no frequent itemsets, return an empty set
16     if not frequent_itemsets:
17         return set()
18
19     # TODO

```

The input consists of a `Set[Itemset]` containing all frequent itemsets. The method should return a `Set[Itemset]` containing all candidate itemsets for the next iteration.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_generate_candidate_itemsets.py
```

Task 1.5

All previous steps can be combined into a single algorithm: Apriori.

Complete the function `fit`, which implements the Apriori algorithm:

```

1 def fit(self, dataset: Dataset):
2     """
3     Use the Apriori algorithm to find all frequent itemsets in the given dataset.
4     Saves the frequent itemsets in the frequent_itemsets attribute.
5
6     Parameters:
7     dataset (Dataset): The dataset to which the Apriori algorithm should be fitted.
8     """
9
10    # Reset the set of frequent itemsets
11    self.frequent_itemsets = set()
12
13    # TODO

```

The input consists of a `Dataset`. The method should not return anything but save the frequent itemsets in the `self.frequent_itemsets` attribute of the Apriori object.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/apriori/test_fit.py
```

Task 2: FP-growth

While Apriori represents a very simple approach to mining frequent itemsets, there are alternative methods available. An interesting method is FP-growth, which necessitates only two passes on the original dataset. This is achieved through the utilization of the so-called FP-trees.

Task 2.1

The first step in FP-growth is to find all frequent 1-itemsets. At the same time, it is beneficial not to immediately discard the occurrence counts of the frequent 1-itemsets.

In `fpgrowth.py` implement `_generate_frequent_one_itemsets_with_occurrence_counts`, which generates all 1-itemsets together with their occurrence counts for a given dataset:

```
1 def _generate_frequent_one_itemsets_with_occurrence_counts(  
2     self, dataset: Dataset  
3 ) -> ItemsetsWithOccurrenceCounts:  
4     """  
5     Generate all frequent 1-itemsets for the given dataset.  
6  
7     Parameters:  
8     dataset (Dataset): The dataset for which the frequent 1-itemsets should be generated.  
9  
10    Returns:  
11    ItemsetsWithOccurrenceCounts: A dictionary containing the frequent 1-itemsets as keys  
12    and their occurrence counts as values.  
13    """  
14    # TODO
```

Expect a `Dataset` as input and return an `ItemsetsWithOccurrenceCounts`. Remember that you did do a similar task in Apriori.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_generate_frequent_one_itemsets_with_occurrence_counts.py
```

Task 2.2

After identifying the frequent 1-itemsets, the f-list can be generated. This is where the occurrence counts of the frequent 1-itemsets come into play.

Complete `_generate_f_list`:

```
1 def _generate_f_list(  
2     self, frequent_one_itemsets: ItemsetsWithOccurrenceCounts  
3 ) -> List[Itemset]:  
4     """  
5     Generate the f-list for the given frequent 1-itemsets.  
6  
7     Parameters:  
8     frequent_one_itemsets (ItemsetsWithOccurrenceCounts): The frequent 1-itemsets with  
9     their occurrence counts for which the F-list should be generated.  
10  
11    Returns:  
12    List[Itemset]: A f-list containing the frequent 1-itemsets sorted by decreasing  
13    occurrence count.  
14    """  
15    # TODO
```

The input consists of an `ItemsetsWithOccurrenceCounts`. The return value should be a `List[Itemset]` containing the frequent 1-itemsets sorted by decreasing occurrence count

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_generate_f_list.py
```

Task 2.3

After generating the f-list, the dataset can be sorted according to the f-list. This is necessary to build the FP-tree.

Complete the function `_sort_dataset_accoring_to_f_list`, which sorts the dataset according to the f-list:

```
1 def _sort_dataset_according_to_f_list(  
2     self, dataset: Dataset, f_list: List[Itemset]  
3 ) -> SortedDataset:  
4     """  
5     Sort the dataset according to the given f-list.  
6  
7     Parameters:  
8     dataset (Dataset): The dataset to be sorted.  
9     f_list (List[Itemset]): The f-list according to which the dataset should be sorted.  
10  
11     Returns:  
12     SortedDataset: The sorted dataset.  
13     """  
14     # TODO
```

The input consists of a `Dataset` and a `List[Itemset]`. The method should return a `SortedDataset`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_sort_dataset_according_to_f_list.py
```

Task 2.4

With the sorted dataset, the FP-tree can be built.

Complete the function `_construct_initial_fp_tree`, which builds the initial FP-tree:

```
1 def _construct_initial_fp_tree(self, sorted_dataset: SortedDataset) -> FPTree:  
2     """  
3     Construct the initial FP-tree from the given sorted dataset.  
4  
5     Parameters:  
6     sorted_dataset (SortedDataset): The sorted dataset from which the initial  
7     FP-tree should be constructed.  
8  
9     Returns:  
10    FPTree: The initial FP-tree.  
11    """  
12    # TODO
```

The input consists of a `SortedDataset`. The method should return an `FPTree`.

`FPTree` implements a method `add_items_to_tree`, which might be helpful for this task.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_construct_initial_fp_tree.py
```

Task 2.5

In FP-growth, besides the initial FP-tree, the so-called conditional FP-trees also play a crucial role. To be able to build these, the conditional pattern base must be generated.

Complete the function `_get_conditional_pattern_base`:

```
1 def _get_conditional_pattern_base(  
2     self, item: Item, fp_tree: FPTree  
3 ) -> ConditionalPatternBase:  
4     """  
5     Get the conditional pattern base for the given item in the FP-tree.  
6  
7     Parameters:  
8     item (Item): The item for which the conditional pattern base should be generated.  
9     fp_tree (FPTree): The FP-tree from which the conditional pattern base should  
10        be extracted.  
11  
12     Returns:  
13     ConditionalPatternBase: The conditional pattern base for the given item.  
14     """  
15     # TODO
```

The input consists of an `Item` and an `FPTree`. The output is a `ConditionalPatternBase`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_get_conditional_pattern_base.py
```

Task 2.6

With the conditional pattern base, the conditional FP-tree can be built.

Complete the function `_construct_conditional_fp_tree`:

```
1 def _construct_conditional_fp_tree(  
2     self, conditional_pattern_base: ConditionalPatternBase  
3 ) -> FPTree:  
4     """  
5     Construct a conditional FP-tree from the given sorted dataset.  
6  
7     Parameters:  
8     conditional_pattern_base (ConditionalPatternBase): The conditional pattern base  
9     for which the conditional FP-tree should be constructed.  
10  
11     Returns:  
12     FPTree: The conditional FP-tree.  
13     """  
14     # TODO
```


The input consists of a `ConditionalPatternBase`. The method should return an `FPTree`.

There are a lot of similarities between this function and `_construct_initial_fp_tree`.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_construct_conditional_fp_tree.py
```

Task 2.7

The last missing step in FP-growth is to recursively mine the frequent itemsets.

Complete `fit`, which implements the FP-growth algorithm:

```
1 def fit(self, dataset: Dataset):
2     """
3     Use the FP-growth algorithm to find all frequent itemsets in the given dataset.
4     Saves the frequent itemsets in the frequent_itemsets attribute.
5
6     Parameters:
7     dataset (Dataset): The dataset to which the FP-growth algorithm should be fitted.
8     """
9     # TODO
```

The input consists of a `Dataset`. The method should not return anything but save the frequent itemsets in the `self.frequent_itemsets` attribute of the FP-growth object.

You are free to implement some extra methods if you think they are necessary.

You can test whether your implementation is correct by executing the following command in the console:

```
1 pytest tests/fpgrowth/test_fit.py
```

Appendices

In [Task 1](#) and [Task 2](#) test cases are provided and used to grade the submission.

The most test cases are based on the following data sets:

Small Fruit Dataset

All test cases starting with the prefix `test_with_small_fruit_dataset` are based on a small transactional dataset regarding fruits.

The dataset is structured as follows:

TID	Items
1	Apple, Banana, Cherry
2	Banana, Cherry
3	Cherry, Apple
4	Dragonfruit, Apple, Cherry
5	Apple, Dragonfruit

Table 1: Small Fruit Dataset

Large Book Dataset

All test cases starting with the prefix `test_with_large_book_dataset` are based on a large(r) ⁴ transactional dataset.

The dataset is structured as follows:

TID	Books	Book	Title
1	Book 1, Book 2, Book 3	Book 1	The Shadows of Tomorrow
2	Book 2, Book 4, Book 5	Book 2	Echoes of a Forgotten Realm
3	Book 3, Book 6, Book 7	Book 3	Whispers of the Ancient World
4	Book 4, Book 8, Book 9	Book 4	Chronicles of the Unseen
5	Book 1, Book 5, Book 10	Book 5	Legends of the Fallen Skies
6	Book 6, Book 7, Book 8	Book 6	Tales of the Crimson Dawn
7	Book 9, Book 10, Book 2	Book 7	Secrets of the Silent Ocean
8	Book 3, Book 4, Book 5	Book 8	Memories of the Last Horizon
9	Book 6, Book 8, Book 1	Book 9	Dreams of the Distant Stars
10	Book 7, Book 9, Book 10	Book 10	Visions of the Lost Empire

Table 2: Large Book Dataset

⁴The term "large" is, of course, somewhat exaggerated. However, the datasets should still be comprehensible by humans, which is why this is the largest dataset we use for testing.