Assignment 6: Latice exploration for functional dependencies

Robin Pourtaud DS
https://github.com/RobinPourtaud/DDDI-Assignment6

The algorithm

Test Functional Dependency function

The test functional dependency function is a function that takes a pandas dataframe and X and Y such that X->Y is a functional dependency to test.

```
def test_fd(df,X, Y):
        return df.groupby(X)[Y].nunique().le(1).all().all()
```

This function group by X and check if the number of unique values of Y is less or equal to 1. If it is, then X->Y is a functional dependency.

For example, if we have the following dataframe:

A	B	C
1	1	1
1	1	2
1	2	1

If we test the functional dependency B->C, we will get the following result:

```
df = pd.DataFrame({'A': [1,1,1], 'B': [1,1,2], 'C': [1,2,1]})
df.groupby('B')['C'].nunique()
```

B	C
1	2
2	1

All values of C are not less or equal to 1, so B->C is not a functional dependency.

Why le(1) and not eq(1)? Because we can have null values in the dataframe.

Explore Latice function

In this project, we are looking to find a smart way to explore the latice of combinations. That being said, I propose to compute the minimal functional dependencies. Since all dependencies can be derived from the minimal ones, we don't need to go accross the whole latice.

Smart exploration of the latice seems to have been studied a lot. I found easily some paper like : https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2dfe42d8987c9f64456aee9d60aa392a3a6d5099 that answer the question for all functional dependencies. This is why I tried to focus on the minimal functional dependencies.

By minimal functional dependencies, I mean that if A -> B, then Ax -> B for all x. So we only need to find the keys. This thinking will cut a lot of useless checks ! (I hope).

To compare the speed of the algorithm, I propose to compare the speed time of the naive approach with the optimized one.

```
def latice_exploration_naïve(df):
    """Explore the latice of combinations to find functionnal dependencies.

    Args:
        df (pandas.DataFrame): Dataframe with the data to explore
    Returns:
        list: List of functionnal dependencies
    """
    def test_fd(X, Y):
        return df.groupby(X)[Y].nunique().le(1).all().all()
    rules = []
    for y in df.columns:
        for n in range(1, len(df.columns)):
            for X in itertools.combinations(set(df.columns) - {y}, n):
                if test_fd(list(X), y):
                    rules.append([list(X), [y]])
    return rules
```

Here is the pseudo code of the optimized algorithm:

Algorithm

1. Initial candidates : $\{(A), (B), ...\}$ (all variables)
2. Check $\forall x, y \in candidates$ if $x \rightarrow y$ is a functional dependency
3. If it is, add $x \rightarrow y$ to the list of FDs and remove x from candidates. We don't care about x and its supersets anymore.
4. If it's not, then add x to the temporary candidates list.
5. Create the new candidates by a union of the temporary candidates. We keep only the union that are relevant. For example {(A,B), (A,C)} will not create {(A,B,C)} because we need to have {(A,B), (A,C), (B,C)} to create it! We ignore the rest.
6. Repeat 2 to 5 until candidates is empty.
7. Return the list of FDs.

The source code of the algorithm is on the notebook on github.
https://github.com/RobinPourtaud/DDDI-Assignment6

In the future, it could be interesting inspire ourselves from algorithms like FD_Mine or TANE.

Results

> The results for naive can be different from https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html#c168191 because for effeciency reasons, I didn't group rules. Like A->B and A->C count for 2 in my case, but only 1 on the website because A->B,C.

> The number of results can"t be compared between the two algorithms because the naive check every dependencies, while the optimized one only check the minimal ones. The number is obviously different.

> You can check results on the notebook if you want. You can also see directly the results using the function `latice_exploration` and the args `prettyPrint` set to True.

If ">2000s" is written, it means that the algorithm took more than 2000 seconds to run. I considered that it was too long to wait for a results -> Time out. The results are set by default to 0.

Filename	TimeOpti	TimeNaive	ResOpti	ResNaive
abalone.csv	1.1113157272338867	3.112067222595215	34	783
adult.csv	425.96985125541687	>2000s	17	0
balance-scale.csv	0.02011585235595703	0.02033233642578125	1	1
breast-cancer-wisconsin.csv	1.5519657135009766	5.027352333068848	25	1713
bridges.csv	2.7577059268951416	22.932160139083862	69	37926
chess.csv	0.8559975624084473	1.4479122161865234	1	1
echocardiogram.csv	2.5449798107147217	21.139954328536987	393	44583
flight_1k.csv	>2000s	>2000s	0	0
hepatitis.csv	>2000s	>2000s	0	0
horse.csv	>2000s	>2000s	0	0
iris.csv	0.026702404022216797	0.022899627685546875	4	5
letter.csv	>2000s	>2000s	0	0
nursery.csv	2.5882556438446045	5.732923984527588	0	1
plista_1k.csv	>2000s	>2000s	0	0

We can see that for very small values, like iris, the naive approach can be a little faster, but for every other cases, the optimized approach is way faster. A further statistical analysis would not be necessary because the comparison between results is clear but the comparison between algorithm is not very fair.