main

September 11, 2023

1 Association rule mining

In this notebook, you'll implement the basic pairwise association rule mining algorithm.

To keep the implementation simple, you will apply your implementation to a simplified dataset, namely, letters ("items") in words ("receipts" or "baskets"). Having finished that code, you will then apply that code to some grocery store market basket data. If you write the code well, it will not be difficult to reuse building blocks from the letter case in the basket data case.

```
In [1]: import sys
        print(f"=== Python version ===\n{sys.version}")
```

```
=== Python version ===
3.8.7 (default, Jan 25 2021, 11:14:52)
[GCC 5.5.0 20171010]
```

1.1 Problem definition

Let's say you have a fragment of text in some language. You wish to know whether there are association rules among the letters that appear in a word. In this problem:

- Words are "receipts"
- Letters within a word are "items"

You want to know whether there are *association rules* of the form, $a \implies b$, where a and b are letters. You will write code to do that by calculating for each rule its *confidence*, $\text{conf}(a \implies b)$. "Confidence" will be another name for an estimate of the conditional probability of b given a, or $\Pr[b \mid a]$.

1.2 Sample text input

Let's carry out this analysis on a "dummy" text fragment, which graphic designers refer to as the *lorem ipsum*:

```
In [2]: latin_text = """
        Sed ut perspiciatis, unde omnis iste natus error sit
        voluptatem accusantium doloremque laudantium, totam
        rem aperiam eaque ipsa, quae ab illo inventore
```

```
veritatis et quasi architecto beatae vitae dicta
sunt, explicabo. Nemo enim ipsam voluptatem, quia
voluptas sit, aspernatur aut odit aut fugit, sed
quia consequuntur magni dolores eos, qui ratione
voluptatem sequi nesciunt, neque porro quisquam est,
qui dolorem ipsum, quia dolor sit amet consectetur
adipisci[ng] velit, sed quia non numquam [do] eius
modi tempora inci[di]dunt, ut labore et dolore
magnam aliquam quaerat voluptatem. Ut enim ad minima
veniam, quis nostrum exercitationem ullam corporis
suscipit laboriosam, nisi ut aliquid ex ea commodi
consequatur? Quis autem vel eum iure reprehenderit,
qui in ea voluptate velit esse, quam nihil molestiae
consequatur, vel illum, qui dolorem eum fugiat, quo
voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos
ducimus, qui blanditiis praesentium voluptatum
deleniti atque corrupti, quos dolores et quas
molestias excepturi sint, obcaecati cupiditate non
provident, similique sunt in culpa, qui officia
deserunt mollitia animi, id est laborum et dolorum
fuga. Et harum quidem rerum facilis est et expedita
distinctio. Nam libero tempore, cum soluta nobis est
eligendi optio, cumque nihil impedit, quo minus id,
quod maxime placeat, facere possimus, omnis voluptas
assumenda est, omnis dolor repellendus. Temporibus
autem quibusdam et aut officiis debitis aut rerum
necessitatibus saepe eveniet, ut et voluptates
repudiandae sint et molestiae non recusandae. Itaque
earum rerum hic tenetur a sapiente delectus, ut aut
reiciendis voluptatibus maiores alias consequatur
aut perferendis doloribus asperiores repellat.
"""
```

```
print("First 100 characters:\n  {} ...".format(latin_text[:100]))
```

```
First 100 characters:

Sed ut perspiciatis, unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium,  ...
```

Exercise 0 (ungraded). Look up and read the translation of *lorem ipsum*!

Data cleaning. Like most data in the real world, this dataset is noisy. It has both uppercase and lowercase letters, words have repeated letters, and there are all sorts of non-alphabetic characters. For our analysis, we should keep all the letters and spaces (so we can identify distinct words), but we should ignore case and ignore repetition within a word.

For example, the eighth word of this text is "error." As an *itemset*, it consists of the three unique letters, $\{e, o, r\}$. That is, treat the word as a set, meaning you only keep the unique letters.

This itemset has three possible *itempairs*: $\{e, o\}$, $\{e, r\}$, and $\{o, r\}$.

> Since sets are unordered, note that we would regard $\{e, o\} = \{o, e\}$, which is why we say there are only three itempairs, rather than six.

Start by writing some code to help "clean up" the input.

Exercise 1 (2 points). Complete the following function, `normalize_string(s)`. The input `s` is a string (`str` object). The function should return a new string with (a) all characters converted to lowercase and (b) all non-alphabetic, non-whitespace characters removed.

> *Clarification*. Scanning the sample text, `latin_text`, you may see things that look like special cases. For instance, `inci[di]dunt` and `[do]`. For these, simply remove the non-alphabetic characters and only separate the words if there is explicit whitespace.
>
> For instance, `inci[di]dunt` would become `incididunt` (as a single word) and `[do]` would become `do` as a standalone word because the original string has whitespace on either side. A period or comma without whitespace would, similarly, just be treated as a non-alphabetic character inside a word *unless* there is explicit whitespace. So `e pluribus.unum basium` would become `e pluribusunum basium` even though your common-sense understanding might separate `pluribus` and `unum`.
>
> *Hint*. Regard as a whitespace character anything "whitespace-like." That is, consider not just regular spaces, but also tabs, newlines, and perhaps others. To detect whitespaces easily, look for a "high-level" function that can help you do so rather than checking for literal space characters.

```
In [3]: ### Define demo inputs
        demo_s_ex1 = latin_text[:100]
        print(demo_s_ex1)
```

```
Sed ut perspiciatis, unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium,
```

The demo included in the solution cell below should display the following output:

```
sed ut perspiciatis unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium
```

```
In [4]: def normalize_string(s):
            assert type (s) is str
            ###
            ### YOUR CODE HERE
            new_str = [c for c in s.lower() if c.isalpha() or c.isspace()]
        #    print(new_str)
            new_str2 = "".join(new_str)
        #    print(new_str2)
```

```
    return(new_str2)
    ###

# Demo:
print(normalize_string(demo_s_ex1))
```

```
sed ut perspiciatis unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium
```

The cell below will test your solution for Exercise 1. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [5]:
```
### test_cell_ex1
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_1',
    'func': normalize_string, # replace this with the function defined above
    'inputs':{ # input config dict. keys are parameter names
        's':{
            'dtype':'str', # data type of param.
            'check_modified':False,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'str',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': True, # Ignored if dtype is not df
            'check_row_order': True, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        }
    }
}
tester = Tester(conf, key=b's63L2lglDfBJpcKzxpwcyy61HyKnJNBOJXl9BMyWhyo=', path='resou
for _ in range(70):
    try:
        tester.run_test()
```

```
            (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
        except:
            (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tes
            raise
    print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 2 (1 point). Implement the following function, `get_normalized_words(s)`. It takes as input a string `s` (i.e., a `str` object). It should normalize `s` and then return a list of its words. (That is, the function should not assume that the input `s` is normalized yet.)

```
In [6]: ### Define demo inputs

        demo_s_ex2 = latin_text[:33]
        demo_s_ex2
```

```
Out[6]: '\nSed ut perspiciatis, unde omnis '
```

The demo included in the solution cell below should display the following output:

```
['sed', 'ut', 'perspiciatis', 'unde', 'omnis']
```

```
In [7]: def get_normalized_words (s):
            assert type(s) is str
            ###
            ### YOUR CODE HERE
            normalized_words = ""
            normalized_words = normalize_string(s)
        #    print(normalized_words)
            normalized_list = []
            normalized_list = normalized_words.split()
        #    print(normalized_list)
            return normalized_list
            ###

        # Demo:
        print(get_normalized_words(demo_s_ex2))
```

```
['sed', 'ut', 'perspiciatis', 'unde', 'omnis']
```

The cell below will test your solution for Exercise 2. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [8]:
```
### test_cell_ex2

from tester_fw.testers import Tester

conf = {
    'case_file':'tc_2',
    'func': get_normalized_words, # replace this with the function defined above
    'inputs':{ # input config dict. keys are parameter names
        's':{
            'dtype':'str', # data type of param.
            'check_modified':False,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': True, # Ignored if dtype is not df
            'check_row_order': True, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        }
    }
}
tester = Tester(conf, key=b's63L2lglDfBJpcKzxpwcyy61HyKnJNBOJXl9BMyWhyo=', path='resou
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_vars) = te
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars) = te
        raise

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 3 (2 points). Implement a function, `make_itemsets_unstructured_text(text)`. The input, `text`, is a string of unstructured text (like the `latin_text` example above). Your function should get the normalized words from the text, convert the characters of each word into an itemset and then return the list of all itemsets. These output itemsets should appear in the same order as their corresponding words in the input. You may find it helpful to call `get_normalized_words` in your solution.

In [9]:
```
### Define demo inputs
demo_text_ex3 = 'sed \tut, perspiciatis\n und.e omnis'
```

The demo included in the solution cell below should display the following output:

```
[{'d', 'e', 's'},
 {'t', 'u'},
 {'a', 'c', 'e', 'i', 'p', 'r', 's', 't'},
 {'d', 'e', 'n', 'u'},
 {'i', 'm', 'n', 'o', 's'}]
```

> Because sets are unordered, different versions of Python may produce sets with whose element-ordering differs from what you see above. However, the sets themselves should be in this order in the output list, since that is the order in which the corresponding words were given.

```
In [10]: def make_itemsets_unstructured_text(text):
             ###
             ### YOUR CODE HERE
             normalized_words = []
             normalized_words = get_normalized_words(text)
         #    print(normalized_words)

             return [set(normalized_set) for normalized_set in normalized_words]
             ###

         make_itemsets_unstructured_text(demo_text_ex3)
```

```
Out[10]: [{'d', 'e', 's'},
          {'t', 'u'},
          {'a', 'c', 'e', 'i', 'p', 'r', 's', 't'},
          {'d', 'e', 'n', 'u'},
          {'i', 'm', 'n', 'o', 's'}]
```

The cell below will test your solution for Exercise 3. The testing variables will be available for debugging under the following names in a dictionary format. - `input_vars` - Input variables for your solution. - `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution. - `returned_output_vars` - Outputs returned by your solution. - `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [11]: ### test_cell_ex3

         from tester_fw.testers import Tester

         conf = {
             'case_file':'tc_3',
             'func': make_itemsets_unstructured_text, # replace this with the function defined
             'inputs':{ # input config dict. keys are parameter names
                 'text':{
                     'dtype':'list', # data type of param.
```