part0

September 4, 2019

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.

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 [1]: 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 (normalize_string_test: 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 [2]: import re
        def normalize_string(s):
            assert type (s) is str
            s=s.lower()
            s=re.sub('[^a-z \n]','',s)
            return s
        # Demo:
        print("First 100 characters:\n  {} ...",normalize_string(format(latin_text[:100])))
```

```
First 100 characters:
  {} ...
sed ut perspiciatis unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium
```

```
In [3]: # `normalize_string_test`: Test cell
        norm_latin_text = normalize_string(latin_text)

        assert type(norm_latin_text) is str
        assert len(norm_latin_text) == 1694
        assert all([c.isalpha() or c.isspace() for c in norm_latin_text])
        assert norm_latin_text == norm_latin_text.lower()

        print("\n(Passed!)")
```

```
(Passed!)
```

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

For example, get_normalized_words(latin_text) will return the following:

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

(Only the first five entries of the output are shown.)

```
In [4]: def get_normalized_words (s):
            assert type(s) is str
            s=normalize_string(s)
            q=s.split()
            return q
        # Demo:
        print("First five words:\n{}".format(get_normalized_words(latin_text)[:5]))
```

```
First five words:
['sed', 'ut', 'perspiciatis', 'unde', 'omnis']
```

```
In [5]: # `get_normalized_words_test`: Test cell
        norm_latin_words = get_normalized_words(norm_latin_text)

        assert len(norm_latin_words) == 250
        for i, w in [(20, 'illo'), (73, 'eius'), (144, 'deleniti'), (248, 'asperiores')]:
            assert norm_latin_words[i] == w

        print ("\n(Passed.)")
```

```

(Passed.)
```

Exercise 3 (make_itemsets_test: 2 points). Implement a function, make_itemsets(words). The input, words, is a list of strings. Your function should convert the characters of each string 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.

For example, consider the following:

```
make_itemsets(['sed', 'ut', 'perspiciatis', 'unde', 'omnis'])
```

This would return the list,

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