Algorithms for Massive Datasets Project

An APriori Algorithm Implementation

Elisabetta Rocchetti (965762)

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Contents

1 Dataset

The dataset "*Old Newspapers*" comes from a public source, namely Kaggle[1]. It contains a subset of raw data taken from a larger corpus, the *HC Corpus*.
Each item in this dataset represents a sentence sampled from an old newspaper. In particular, the dataset's attributes are the following:

- Language: the language in which the sentence is expressed. It can be one of the 67 languages listed in the dataset's description on Kaggle's website[2].
- Source: each sentence is sampled from an old newspaper. This field refers to which newspaper the text is taken from.
- Date: this attribute gives information about the article's release date. The sentence inherits its article's date.
- Text: this field contains the sentence or the paragraph sampled from the newspaper's article.

The *Old Newspapers* dataset includes 16806041 sentences (or paragraphs) structured as described above, constituting a 6.02 GB `.tsv` file: this fact suggests that the result of any operation considering pairs of sentences could not be stored in an average-sized RAM. This problem can be tackled using algorithms for massive datasets.
In order to be able to test (in a reasonable amount of time) any code performing operations on this dataset, a subset including 500 Italian sentences is selected and treated as if it was a massive dataset indeed; thus, any line of code is written implementing highly-scalable techniques and methods.

2 Data organization

The first step to perform is to download the *Old Newspapers* dataset. Kaggle's website offers the direct download option via command line. Specifically, the command `kaggle datasets download` needs the exact name of the resource to dowload. To get this information it is useful to run the command

```
kaggle datasets list --tags business,internet,news,history,linguistics
```

which searches for datasets in Kaggle having the tags business, internet, news, history, linguistics, thus retrieving the reference name `alvations/old-newspapers`.
Then, the dataset can be downloaded executing

```
kaggle datasets download alvations/old-newspapers --unzip
```

Developing the `python` code in *Google Colaboratory* environment, the dataset is downloaded and stored at `/content/old-newspaper.tsv`: thus, using `pandas`' `read_csv` it is possible to fetch the dataset and to store it in a local variable.

```
data = pd.read_csv("/content/old-newspaper.tsv", sep ="\t")
```

[1] urlhttps://www.kaggle.com
[2] `https://www.kaggle.com/alvations/old-newspapers`

For testing purposes (in particular, to avoid to download every time the whole dataset), the Italian subset of the dataset is stored permanently on my personal Google Drive.

```
data_italian = data.loc[data['Language'] == "Italian", "Text"]
path = '/content/drive/My Drive/data_italian.csv'
with open(path, 'w', encoding = 'utf-8-sig') as f:
  data_italian.to_csv(f, index = False)
```

Future runs of the project's code can be done retrieving the dataset in the following way.

```
data_italian = pd.read_csv('/content/drive/My Drive/data_italian.csv')
```

As stated in Section 1, only 500 sentences (*Text* attribute) from the Italian subset are taken into account for the rest of the project.

```
data = data_italian["Text"][:500]
```

3 Pre-processing

Proper pre-processing is crucial to be able to work on the dataset in an efficient and correct manner: in particular, in this project, it is necessary to act and to take decisions as if the dataset was a massive dataset. For this reason, the first step to process the data is to make it a Spark's *Resilient Distributed Dataset* (RDD), which allows parallel computation and multiple-nodes data partition, which are essential in a Distributed File System environment (which is assumed in this project). This step can be done using Spark's `parallelize` method, which has to be called from a Spark context (`sc`).

```
rdd_dataset = sc.parallelize(data)
```

Once the RDD version of the dataset is created, it is possible to continue the pre-processing implementing a text tokenizer to extract single words out of each sentence. This is done by instantiating a *tokenizer* using NLTK's `RegexpTokenizer`, which takes a regular expression in input and returns a tokenizer working with that regular expression. The regular expression `\w+` matches one or more word characters.
To apply the tokenization to each element in the RDD dataset, the `map` function is used, passing as argument another function which tokenizes, lowers and strips the text. Notice that each sentence is managed as a set: this means that there are not word repetitions.

```
custom_tokenizer = RegexpTokenizer('\w+')
rdd_dataset = (rdd_dataset.map(lambda txt: list(set(custom_tokenizer
                                                    .tokenize(txt.lower().strip())))))
```

4 Algorithm and implementation

This Section shows the first implementation of *Apriori* algorithm and the final version of it. These two differ a bit because the former is a real first approach, which is coded without having the whole picture of the process in mind. The latter version of the code is written having a more assessed vision of how the implemented algorithm should behave.

The Apriori algorithm finds the most frequent itemsets: this problem is often linked to select association rules with high interest and confidence. Association rules are represented as $I \rightarrow j$, where I is a set of items and j is one item. The implication $\rightarrow$ represent the statement "if all of the items in I appear in some basket, then j is likely to appear in that basket as well", where baskets are sets of items. The "likeliness" is defined by the concept of *confidence*, which is the ratio of baskets containing I that also contain j. To measure how much the presence of I determines also the presence of j, the *interest* is computed as the difference between the confidence and the fraction of baskets containing j. These measures need the computation of the frequencies of itemsets. One notorious application of these concepts is the *market-basket analysis*.

4.1 First implementation

To develop this first implementation attempt, *Mining of Massive Datasets* textbook written by A. Rajaraman and J. Ullman (chapter 6) is analysed and literally transformed into code.
In Section 2 it is explained how the sentences are processed: the result of that pre-processing is a list of lists containing as many strings as the number of unique words in the sentences.

4.1.1 First pass

The first pass in Apriori consists in creating two tables: one serves as a "dictionary" and the other stores counts of words. The former table is useful to have a word-to-number correspondence to be able to encode each word with the respective integer count. The latter table counts the frequency of singletons, thus each cell contains the number of occurrencies of the respective item.
The "dictionary" is created as follows.

```
rdd = (rdd_dataset.flatMap(lambda txt: txt)
                  .map(lambda word: (word,1))
                  .reduceByKey(lambda w1, w2: 1)
                  .map(lambda x: x[0])
                  .collect())
dictionary = dict(zip(rdd, range(len(rdd))))
```

First, the dataset is flattened (the lists representing sentences are merged into one single list); then, each word is mapped into a key-value pair `(word, 1)`: this translation is useful only to be able to use `reduceByKey`, which collapses the items having the same key, and, in this particular instance, does not aggregate them with any operation. The final `map` picks only the keys in the key-value pair structures. Now that the data structure `rdd` contains unique items from all the baskets (thus being a lot smaller than the previous `rdd_dataset`), it is zipped with a vector containing numbers from 0 to `len(rdd)`, thus the two structures together form the "dictionary", mapping each word to an integer number. This whole set of operations could be done using only the `set` python data structure, but it is important to keep in mind that `rdd_dataset` is a massive dataset, thus map and reduce functions have to be implemented.

The count table is created as follows.

```
def getIndex(txt):
  word_list = []
```

```
  for word in txt:
    i = dictionary[word]
    if i not in word_list:
      word_list = word_list + [i]
  return word_list

rdd_coded_flattened = rdd_dataset.flatMap(getIndex)
count_singletons = (rdd_coded_flattened.map(lambda x: (x, 1))
                                       .reduceByKey(lambda x1, x2: x1+x2))
```

Function `getIndex` translates each word element into the respective integer number as defined in the "dictionary" data structure. From now on, it will be less expensive in terms of memory to work with "coded" datasets (thus having integers instead of strings). The first step is, indeed, the translation from word to integers and stores everything in an unique flattened data structure. The second step computes the frequencies of singletons: first, the `map` function maps all the elements in a key-value pair structure, with all the keys having 1 as value. This step prepares the items making them "countable". Lastly, the `reduceByKey` function aggregates all the items having the same key (thus the same word as key value) and computes the sum across all the values, obtaining the total number of occurrences per word.

4.1.2 Between the passes

In this step prepares the input for the second step, selecting only the frequent singletons to be evaluated in the second pass: this "in-between" step implements what is permitted by the monotonicity property, which states "if a set I of items is frequent, then so is every subset of I".
This step creates the *frequent-items table*, which re-numbers the m frequent singletons from a $1-n$ structure to a $1-m$ structure (with $m \leq n$). This is implemented as follows.

```
s = len(dictionary) * 0.01
count_frequent_singletons = count_singletons.filter(lambda x: x[1] > s).collect()
frequent_items_table = np.zeros(len(dictionary))
for c, item in enumerate(count_frequent_singletons):
  frequent_items_table[item[0]] = c+1
```

After setting the threshold `s` which determines how many occurrencies a singleton must have to be considered "frequent", the `filter` function is used to filter in the frequent singletons. As last step, each frequent singleton is mapped in a new data structure, the *frequent-items table*, which has either 0 if the corresponding singleton is not frequent or the new numbering (a value between 1 and m) if it is frequent.

4.1.3 Second pass

This step counts the occurrencies of pairs of frequent singletons, thus allowing for another selection (which depends on the threshold `s`) filtering in the frequent pairs. The implementation of this step is shown in the following.

```
def filterFrequent(words):
  r = []
```

```
  for w in words:
    if frequent_items_table[w] != 0:
      r = r + [w]
  return r

def generatePairs(words):
  r = []
  for i in range(len(words)):
    for j in range(i+1, len(words)):
      r = r + [((words[i], words[j]),1)]
  return r

rdd_coded = rdd_dataset.map(getIndex)
count_pairs = (rdd_coded.map(filterFrequent)
                    .filter(lambda x : len(x) > 1)
                    .flatMap(generatePairs)
                    .reduceByKey(lambda x1, x2 : x1 + x2))
count_frequent_pairs = count_pairs.filter(lambda x: x[1] > s).collect()
```

Function `filterFrequent` filter out the non-frequent items in each basket; function `generatePairs` generates a key-value pair where the key is formed by each 2-items subset obtainable from the filtered basket, and puts the pair in a key-value structure where the value is always 1 (this will facilitate the frequency computation).
After coding each basket's items into numbers, the frequency of pairs can be computed following these steps:

- filtering out the non-frequent items from each basket (using function `filterFrequent`);
- ignoring the baskets containing less than two items, because a pair could not be formed by only one item;
- generating pairs in each basket using the items that remains after previous filters (computing function `generatePairs`);
- thanks to the implicit transformation computed in `generatePairs`, it is possible to aggregate any item with the same key and to sum the respective value, which is always 1: this allows an easy frequency computation;
- eventually, only the pairs having a frequency greater than the threshold are kept and stored in `count_frequent_pairs`.

4.2 Final implementation

After understanding in a precise manner which steps to perform to obtain the final result, it is possible to re-write the code in a compact way.
First, it is useful to explain all the functions and classes used to make the final `apriori` function more readable.

```
def createDictionary(rdd_dataset, tokenizer):
  rdd = (rdd_dataset.flatMap(lambda txt: txt)
```

```
                .map(lambda word: (word,1))
                .reduceByKey(lambda w1, w2: 1)
                .map(lambda x: x[0])
                .collect())
  dictionary = dict(zip(rdd, range(len(rdd))))
  return dictionary
```

Function `createDictionary` is the same adopted in Section 4.1, thus no more explanations are needed in this case.

```
class Indexer():
  def __init__(self, tokenizer, dictionary):
    self.tokenizer = tokenizer
    self.dictionary = dictionary
    self.reverse_dictionary = dict((y,x) for x,y in dictionary.items())

  def getIndex(self, txt):
    word_list = []
    for word in txt:
      i = self.dictionary[word]
      if i not in word_list:
        word_list = word_list + [i]
    return word_list

  def getWord(self, coded):
    new_k = []
    if type(coded[0]) != int:
      for k in coded[0]:
        new_k = new_k + [self.reverse_dictionary[k]]
      return (tuple(new_k), coded[1])
    else: return (self.reverse_dictionary[coded[0]], coded[1])
```

Class `Indexer` has a two-fold scope: first, it translates words to integers, following the proper coding imposed by the dictionary (function `getIndex`); second, it translates integers to words (function `getWord`), using the "reverse dictionary" (which is equal to the dictionary structure but has the key-value pairs inverted) as reference. `getIndex` takes a basket as input and retrieves the coded version of that basket as output. `getWord` takes key-value pairs as input (where the key can be either a singleton or a k-itemset) and returns the same object but with the numeric key value translated into words.

```
class FrequentItemSets():
  def __init__(self, dict_size):
    self.frequent_itemsets = np.zeros(dict_size)
    self.dictionary_size = dict_size
```

```
  def filterFrequent(self, words):
    r = []
    for w in words:
      if self.frequent_itemsets[w] != 0:
        r = r + [w]
    return r

  def getFrequentItemSets(self):
    return self.frequent_itemsets

  def setFrequentItemSets(self, frequent_itemsets):
    self.frequent_itemsets = np.zeros(self.dictionary_size)
    i = 0
    for item in frequent_itemsets:
      if type(item[0]) == tuple:
        for el in item[0]:
          i = i+1
          self.frequent_itemsets[el] = i
      else:
        i = i+1
        self.frequent_itemsets[item[0]] = i
```

Class `FrequentItemSets` manages the structure `frequent_items_table` seen in Section 4.1.2. In particular, it offers three functions: `filterFrequent`, `getFrequentItemSets` and `setFrequentItemSets`. The first function adopts the same behaviour seen in Section 4.1.3. The second one returns the frequent items table. The latter takes as input a key-value pair list structure and, for each key value, updates the `frequent_itemsets` structure in the same way as explained for the `frequent_items_table` structure in Section 4.1.2.

```
class SubSetter():
  def __init__(self, n = 2):
    self.subset_size = n
  def findSubSets(self, words):
    return list(itertools.combinations(words, self.subset_size))
  def generateSubSets(self, words):
    l = self.findSubSets(words)
    r = [(item,1) for item in l]
    return r
```

Class `SubSetter` generates all the k-subsets from a set of words, and it does this by using the function `itertools.combinations`. It then prepares each subset in a key-value pair, with value always 1, to facilitate the frequency computation (this class has the same scope as function `generatePairs` in Section 4.1.3).

```
1  def apriori(dataset, itemset_size, tokenizer, threshold, coded = False):
2    rdd_dataset = sc.parallelize(dataset)
```

```
rdd_dataset = (rdd_dataset.map(lambda txt: list(set(tokenizer
                                                    .tokenize(txt
                                                              .lower()
                                                              .strip()))))
dictionary = createDictionary(rdd_dataset, tokenizer)
threshold = len(dictionary) * threshold
indexer = Indexer(tokenizer, dictionary)
frequentItemsets = FrequentItemSets(len(dictionary))
results = []

rdd_coded = rdd_dataset.map(indexer.getIndex)
rdd_coded_flattened = rdd_dataset.flatMap(indexer.getIndex)

i = 1
iterate = True
while(iterate):
  if i == 1:
    count_frequent_singletons = (rdd_coded_flattened
                                 .map(lambda x: (x, 1))
                                 .reduceByKey(lambda x1, x2: x1+x2)
                                 .filter(lambda x: x[1] > threshold))
    frequentItemsets.setFrequentItemSets(count_frequent_singletons.collect())
    if coded:
      results.append(count_frequent_singletons.map(indexer.getWord))
    else: results.append(count_frequent_singletons)

  else:
    subsetter = SubSetter(i)
    count_frequent_kItemSets = (rdd_coded.map(frequentItemsets.filterFrequent)
                                 .filter(lambda x : len(x)>=i)
                                 .flatMap(subsetter.generateSubSets)
                                 .reduceByKey(lambda x1, x2 : x1 + x2)
                                 .filter(lambda x: x[1] > threshold))
    frequentItemsets.setFrequentItemSets(count_frequent_kItemSets.collect())
    if coded:
      results.append(count_frequent_kItemSets.map(indexer.getWord))
    else: results.append(count_frequent_kItemSets)

  i = i+1
  if ((itemset_size == "no-stop" and len(results[-1].collect()) == 0) or
      (itemset_size != "no-stop" and i > itemset_size)):
    iterate = False
```

```
45
46     return results
```

`apriori` function puts together all the steps seen in Section 4.1 and allows the generation of k-frequent itemsets with k (possibly) greater than 2. This function's structure is divided in 3 blocks:

1. preparation step (lines 2-14): this step prepares the RDD, other data structures (such as dictionary, indexer and frequent itemsets instances) and parameters (threshold) to be used in following steps. Moreover, it prepares 2 version of the same coded dataset: the former is structured in baskets, the latter is flattened. This is useful because `rdd_coded_flattened` is used when creating singletons, and `rdd_coded` is used elsewhere;

2. frequent singletons (lines 20-24): this step is detached from the others in Apriori algorithm because there are major differences to assess, thus it is more convenient to separate it. The main differences are that this step does not rely on a previously-computed frequent items table and it does not need to generate subsets. This step's output are frequent singletons counts and the very first update on the frequent items table;

3. frequent k-itemsets, $k > 1$ (lines 30-36): this step computes all the frequent k-itemsets with $k > 1$. It filters out infrequent items from the previous iteration, filters out baskets which do not have enough items to produce a k-subset, it generates all the possible k-subsets on the remaining items in the baskets, it computes the frequencies, it filters out those k-subsets that are not frequent enough and, eventually, it updates the frequent items table.

This function returns a list of RDDs containing the apriori results in each step (e.g. frequent singletons, frequent pairs, ...). It is also possible to decide whether to translate integer codes into words to have higher result-readability. Lastly, it is possible either to define a maximum itemset size or to let the algorithm run until there are not enough frequent items to produce further greater itemsets.

5 Scalability

Coding choices and solutions are implemented having always in mind scalability, since this algorithm is in real-world scenarios in which datasets are considered big data datasets. Both pre-processing and Apriori are implemented with a great use of Spark's `rdd` data structure, `map` and `reduce` functions: this means parallel and distributed computation, thus scalability. This code-style together with the nature of Apriori (which works iteratively with a decreasing input size, due to the filters applied over non-frequent itemsets) allows for an even higher scalability.
Spark's solutions are not used only when the data structure on which the execution is currently working on is (possibly) small enough to be stored in a standard-sized RAM: this happens, for instance, when dealing with the dictionary (which indeed is not stored in a `rdd` data structure) and the frequent items table (which is surely small enough to avoid using scalable solutions).

6 Experiments

A typical result from an execution of `apriori` is depicted in Figure 1.

```
[(('il', 'l', 'la', 'e'), 134),
(('il', 'l', 'la', 'per'), 104),
(('il', 'l', 'e', 'per'), 107),
(('il', 'la', 'e', 'per'), 133),
(('il', 'a', 'di', 'e'), 184),
(('il', 'a', 'di', 'per'), 143),
(('il', 'a', 'che', 'e'), 158),
(('il', 'a', 'che', 'per'), 121),
(('il', 'di', 'che', 'e'), 159),
(('il', 'di', 'che', 'per'), 122)]
```

Figure 1: **Result obtained from an execution of `apriori`**. This Figure depicts the output resulting from the execution of the following code: `apriori(data, 4, RegexpTokenizer('\w+'), 0.01, True)`. Only 10 items belonging to the last RDD stored in the result are shown.

Experiments are done on different itemsets sizes and thresholds, but keeping the dataset size constant and the same tokenizer. Execution times [3] (cumulative) are stored and displayed in Figure 2.

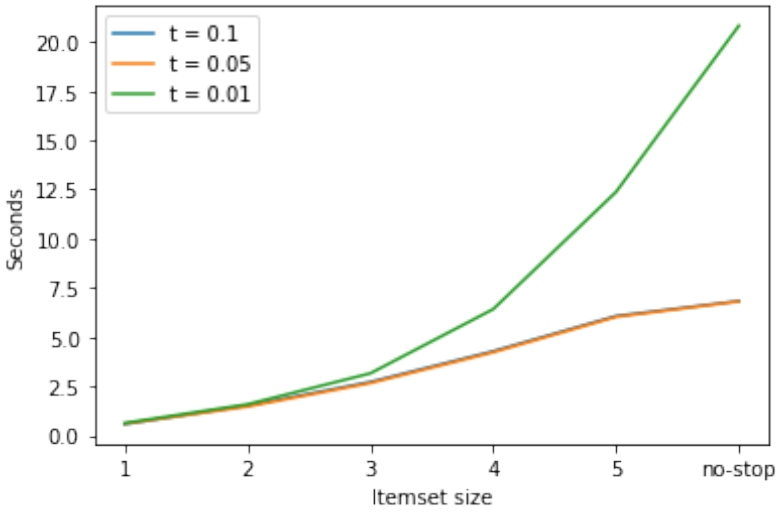

Figure 2: **Different experiments on `apriori`**. This Figure depicts a few `apriori` execution times (cumulative, in seconds) measured considering different itemset sizes (from singletons to no maximum size) and different threshold levels (0.1, 0.05, 0.01). Dataset size and tokenizer are kept constant.

As expected, execution time typically increases as the threshold decreases: indeed, it is expected that more itemsets are included in the candidate set to be evaluated in subsequent steps.

6.1 Confidence and Interest

As explained at the beginning of Section 4, Apriori can be used in market-basket analysis computation, in particular it is really useful to obtain confidence and interest measures. This project contains also a short investigation on the computation, via `apriori` function, of confidence and interest.

[3] These execution times have to be taken keeping in mind that they are not real algorithm execution times, but they are obtained as the median of timings measured on multiple repetitions of the same experiment: this is necessary because time is taken from the system, thus it is not measuring only the execution time, but also other non-measurable processes. These test are not performed on a distributed file system.

```
def containsI(elem):
  for i in I:
    if i not in elem[0]:
      return False
  return True

def containsIj(elem):
  for i in (I+[j]):
    if i not in elem[0]:
      return False
  return True
```

Functions `containsI` and `containsIj` check whether or not a set of elements is included in another set of elements. These functions will be useful to select itemsets containing I and $I \cup j$.

```
def computeConfidenceAndInterest():
    k = len(I)
    result = apriori(data, k+1, tokenizer, threshold, True)
    number_I, number_Ij, number_j = 0, 0, 0
    try:
      number_I = list(dict(result[-2].filter(containsI).collect()).values())[0]
    except IndexError:
        return "No itemset containing I"
    try:
      number_Ij = list(dict(result[-1].filter(containsIj).collect()).values())[0]
    except IndexError:
      pass
    try:
      number_j = list(dict(result[0].filter(lambda x: x[0]==j).collect()).values())[0]
    except IndexError:
      pass
    confidence = number_Ij/number_I
    interest = confidence - (number_j/len(data))
    return (confidence, interest)
```

Function `computeConfidenceAndInterest` computes `apriori` setting the maximum itemsets size to $|I \cup j|$; then it obtains the number of baskets containing I, the number of baskets containing $I \cup j$ and the number of baskets containing j. Lastly, if possible, it computes the confidence and the intervals measures as explained in Section 4.
For instance, setting `I = ['di', 'e', 'che']`, `j = 'per'`, `tokenizer = RegexpTokenizer('\w+')` and `threshold = 0.01`, the result obtained from `computeConfidenceAndInterest` is:

- Confidence: 0.68
- Interest: 0.19

This means that it is likely to find the word "per" together with the words "di", "e", "che" and the fraction of sentences containing "di", "e", "che" and "per" is higher than the fraction of sentences containing only "per", thus one can suspect "di", "e" and "che" to influence the presence of "per".

7 Comments and discussions

The main purpose of this project is to apply *Apriori* algorithm in order to make it work on a textual dataset, namely the *Old Newspaper* dataset.
The *Old Newspaper* dataset is a notorious corpus containing sentences and paragraphs samples from newspapers. In this application, this dataset's sentences serve as a base to build baskets, which are sets of words.
The dataset is downloaded directly from Kaggle's APIs and, possibly, stored "locally" in a Google Drive folder. Given the *Old Newspaper*'s large size, a subset of 500 Italian sentences is taken from the corpus, allowing for a faster computation and testing of further steps.
A crucial step to perform to run `apriori` is to pre-process the corpus, tokenizing each sentence and obtaining lists of tokens representing unique words in sententeces. This way, baskets are available and ready to be the input for `apriori`.
An initial explorative phase is done to have a complete picture of how `apriori` code should behave, then the final implementation is presented as the union of three major blocks: preparation step, frequent singletons step and frequent k-itemsets step. This implementation makes a great use of Spark classes, functions and data structure as well as personalised classes made to achieve a better readability and organization. Descriptions, explanations and coding choices are made to adhere to the theoretical version given by *Mining of Massive Datasets* textbook.
Scalability represents a key factor concerning coding choices, which consider the input data size as if it was big enough to be considered "big data". Map and reduce functions are used whenever the conditions and the nature of the data structure to manage are suitable for distributed and parallel computing.
Experiments are performed both to evaluate the effectiveness of the algorithm implementation per se and to check how suitable it is when used to compute confidence and interest metrics for market-basket analysis. In both cases, `apriori` behaves as expected.
Future works concerning the exploration of Apriori variants should consider testing procedures performed on a distributed file system on which it is possible to have a more precise estimate of how scalable the proposed solutions are.