Association Rule Mining

Assignment 3 - Extracting associating rules using Apriori Algorithm

Team Members

Prakhar Srivastav (ps2894)

Running the program

On CLIC

To run on the **CLIC** machines, just run the following set of commands.

```
$ cd /home/ps2894/ps2894-proj3
$ ./apriori.py
```

Locally

```
$ cd <project_folder>
$ ./apriori.py
```

Dataset

Description

The dataset that I've used to derive the association rules from is the NYC School Report 2009-2010 dataset which has progress report results for all schools in the city of New York. The schools reported in this dataset are across levels - Elementary, Middle & High School. Associated with each row, is a school `DBN` number that acts as a primary key. The dataset also contains the school name and principal for each school. The dataset contains, among other things, for each school its grades on environment, progress, performance and an overall grade. Each of these grades columns are categorical values ranging from A to F. Alongside each grade column is its numerical score from which the grade is derived. The raw dataset has a total of approximately 1600 schools a few of which have missing data on a few columns.

A list of all columns is indicated below, the columns marked in **bold** have been used in generating the association rules in this program.

- DBN
- District
- School Name
- Principal

- Progress Report Type
- School level
- Peer index
- **2009-10 Overall Grade**
- 2009-10 Overall Score
- **2009-10 Environment Grade**
- 2009-10 Environment Category Score
- **2009-10 Performance Grade**
- 2009-10 Performance Category Score
- **2009-10 Progress Grade**
- 2009-10 Progress Category Score
- 2008-09 Progress Report Grade

Transformation

The steps involved in building the INTEGRATED-DATASET are listed below -

1. Remove all the columns that have scores.
2. Filter out the rows that have null values in any of the "grade" columns.
3. Transform the values in each grade column using the following algorithm -
    - Assign the number `i` to each column, with `Overall Grade` being 1, `Environment grade` being 2 and so on.
    - For each column, append the column index, to the value. Hence, a value of `A` in `Performance Grade` column becomes `A3` and an `F` in `Overall Grade` becomes `F1`.

The total number of rows in the INTEGRATED-DATASET is 1483.

Why Interesting?

What makes this data interesting is that it categorizes evaluation of schools in New York based on various metrics. In most evaluation schemes, the relationships and interplay between metrics is not evident, hence using an algorithm like association rule mining can be useful in deriving interesting relationships.

Internal Design

The key overarching design philosophy in the program has been **to keep the implementation as data-agnostic as possible**. This has the benefit of trying out the implementation with various datasets with minimal changes in code.

Apriori Algorithm

The algorithm used in this implementation is kept as close to the one described in the paper as possible. In particular, the algorithm for generating frequent-itemsets is identical to the one described in 2.1.1. The implementation of this algorithm can be seen in the `getNextCandidates` function. This function is responsible for generating the next set of candidates of size n, given a set of frequent item sets of size $n-1$. Using the terminology used in the paper, the function takes as input L_{n-1} and generates C_n.

```
def getNextCandidates(self, candidates, size=2):
    """
    Algorithm: As explained in section 2.2.1
    JOIN STEP:
        1. Generate all 2-combinations of Ln-1
        2. Merge these sets together
        3. Validate these if these are of the correct size
           and do not have multiple entries of the same column
        4. Return all the valid ones
    PRUNE STEP:
        1. For each the valid subsets of size n
        2. Generate all subsets of size n - 1
        3. If ANY of these subsets are NOT in Ln-1, reject this set
    """
    possibleCombs = map(self.convertToSet, combinations(candidates, 2))
    validSets = [c for c in possibleCombs if self.hasUniqueCategories(c) and len(c) == size]
    newCandidates = []
    for s in validSets: # s is a set
        subsets = combinations(s, len(s)-1)
        isValid = True
        for x in subsets:
            if set(x) not in candidates:
                isValid = False
                continue
        if isValid and s not in newCandidates:
            newCandidates.append(s)
    return newCandidates
```

Once the candidate sets C_n are generated, the `getFrequentItemSets` validates the count of these sets and generates L_n.

```
def getFrequentItemSets(self, candidateSets):
    # generates the frequent item set for the current candidate set
    # From the paper, this returns Ln for Cn
    frequentSets = []
    for s in candidateSets:
        count = self.getCount(tuple(s))[0]
        if count >= self.support:
            frequentSets.append(s)
    return frequentSets
```

Lastly, the `generateFrequentItemSets` function, ties these two functions together and keeps generating frequent items of larger sizes as long C_n at a particular iteration is not empty.

```
def generateFrequentItemSets(self):
    # generates all sets of frequent itemsets from the dataset
    candidateSet = map(lambda x: set([x]), self.valueMap.keys())
    currentSize = 2
    while len(candidateSet):
        frequentSet = self.getFrequentItemSets(candidateSet)
        for s in frequentSet:
```

```
            t = tuple(s)
            self.frequentSets[t] = self.getCount(t)[0]
        candidateSet = self.getNextCandidates(frequentSet, currentSize)
        currentSize += 1
```

SQLite

SQLite is a popular embeddable, file-based, server-less database engine which is used by this program under the hood. Since the dataset is relational, loading into a simple database that provides SQL-like querying abilities greatly simplifies the design. Before every execution, the program loads the `csv` file into `data.db` which is used subsequently for counting item sets. A few helper functions defined in `apriori.py` help in generating SQL queries on the fly.

```
def generateQuery(self, **kwargs):
    # utility method that generates a SQL query based on the keyword arguments provided
    clause = " and ".join(["%s = '%s'" % (k, v) for (k, v) in kwargs.iteritems()])
    return "select count(*) from %s where %s" % (self.dbname, clause)
```

Lastly, the program uses a third-party library Tabulate that is used for pretty-printing the tabular results on the screen and the file.

Sample Run

An interesting set of association rules can be generated with a support of 10% and confidence of 70%. The results of a run with these values is stored in `example_run.txt`. A few interesting set of rules are

```
perf_grade = C,overall_grade = B  => progress_grade = B [72% conf]
env_grade = A,progress_grade = A  => overall_grade = A  [93% conf]
```

File Structure

```
|-- INTEGRATED-DATASET.csv
|-- apriori.py
|-- data
|-- |-- original_data.csv
|-- dataloader.py
|-- example-run.txt
|-- readme.md
|-- tabulate.py
```