# Association Rules Mining

Jakramate Bootkrajang

September 14, 2017

Based on materials by David Corne:macs.hw.ac.uk

# Outlines

- Basic Concept
- Apriori Algorithm

# Assoc. Rule: Introduction

- In this chapter we will study basket data.
- A basket is a collection of items purchased in a single transaction.
- We'd like to find *interesting* association between items.
  - We want to know what items are usually purchased together.
- Once we know it we can exploit such knowledge in our marketing strategy.

# One example of basket data

| ID | apples, | beer, | cheese, | dates, | eggs, | fish, | glue, | honey, | ice-cream |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | 1 | | | 1 | 1 | |
| 2 | | | 1 | 1 | 1 | | | | |
| 3 | | 1 | 1 | | | 1 | | | |
| 4 | | 1 | | | | 1 | | | 1 |
| 5 | | | | | 1 | | 1 | | |
| 6 | | | | | | 1 | | | 1 |
| 7 | 1 | | | 1 | | | | 1 | |
| 8 | | | | | | 1 | | | 1 |
| 9 | | | 1 | | 1 | | | | |
| 10 | | 1 | | | | | 1 | | |
| 11 | | | | | 1 | | 1 | | |
| 12 | 1 | | | | | | | | |
| 13 | | | 1 | | | 1 | | | |
| 14 | | | 1 | | | 1 | | | |
| 15 | | | | | | | | 1 | 1 |
| 16 | | | | 1 | | | | | |
| 17 | 1 | | | | | 1 | | | |
| 18 | 1 | 1 | 1 | 1 | | | | 1 | |
| 19 | 1 | 1 | | 1 | | | 1 | 1 | |
| 20 | | | | | 1 | | | | |

# Our data

- Our example DB has 20 records (baskets) of supermarket transactions, from a supermarket that only sells 9 things.
- In reality, one month in a large supermarket with five stores spread around a reasonably sized city might easily yield a DB of 20,000,000 baskets, each containing a set of products from a pool of around 1,000.

# An association rule

- A 'rule' is something like this:

  *If a basket contains apples and cheese, then it also contains beer*

- Any such rule has two associated measures:
  1. **Confidence** – when the 'if' part is true, how often is the 'then' bit true? This is the same as accuracy.
  2. **Support** – how much of the database contains the 'if' part?

# A rule from our example basket data

- What is the confidence and support of:

  *If a basket contains beer and cheese, then it also contains honey*

- 2/20 of the records contain both beer and cheese, so support is 10%
- Of these 2, 1 contains honey, so confidence is 50%

# Interesting/Useful rules

- Statistically, anything that is interesting is something that happens significantly more than you would expect by chance.
- For example, basic statistical analysis of basket data may show that 10% of baskets contain bread, and 4% of baskets contain washing-up powder
- So if you choose a basket at random
  - There is a probability of 0.1 that it contains bread.
  - There is a probability of 0.04 that it contains washing-up powder.

# Bread and Washing-up powder

- What is the probability of a basket containing both bread and washing-up powder?
- If these two things are independent, chance is $0.1 \times 0.04 = 0.004$
- That is, we would expect 0.4% of baskets to contain both bread and washing up powder

# Interesting means Surprising

- We therefore have a prior expectation that just 4 in 1,000 baskets should contain both bread and washing up powder.
- If we investigate, and discover that really it is 20 in 1,000 baskets, then we will be very surprised. It tells us that:
  - Something is going on in shoppers minds: bread and washing-up powder are connected in some way.
  - There may be ways to exploit this discovery... HOW?

# Finding surprising rules

- So what is the most surprising rule in this database?
- This would be a rule whose confidence is more different from its expected confidence than any others.
- Moreover, it also has to have a suitable level of support.

# The same basket

| ID | apples, | beer, | cheese, | dates, | eggs, | fish, | glue, | honey, | ice-cream |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | 1 | | | 1 | 1 | |
| 2 | | | 1 | 1 | 1 | | | | |
| 3 | | 1 | 1 | | | 1 | | | |
| 4 | | 1 | | | | 1 | | | 1 |
| 5 | | | | | 1 | | 1 | | |
| 6 | | | | | | 1 | | | 1 |
| 7 | 1 | | | 1 | | | | 1 | |
| 8 | | | | | | 1 | | | 1 |
| 9 | | | 1 | | 1 | | | | |
| 10 | | 1 | | | | | 1 | | |
| 11 | | | | | 1 | | 1 | | |
| 12 | 1 | | | | | | | | |
| 13 | | | 1 | | | 1 | | | |
| 14 | | | 1 | | | 1 | | | |
| 15 | | | | | | | | 1 | 1 |
| 16 | | | | 1 | | | | | |
| 17 | 1 | | | | | 1 | | | |
| 18 | 1 | 1 | 1 | 1 | | | | 1 | |
| 19 | 1 | 1 | | 1 | | | 1 | 1 | |
| 20 | | | | | 1 | | | | |

# Here are some interesting rules in our mini DB

- If a basket contains glue, then it also contains either beer or eggs
  confidence: 100% ; support 25%

- If a basket contains apples and dates, then it also contains honey
  confidence 100% ; support 20%

# The Apriori algorithm

- A simple, fast, and very good algorithm at finding interesting rules.
- It is used a lot in the R&D departments of retailers in the industry.
- Invented by Rakesh Agrawal and Ramakrishnan Srikant. (Their paper is on the web)

# Finding rules in two steps

1. Find all itemsets with a specified minimal support.
   An itemset is just a specific set of items, e.g. apples, cheese. The Apriori algorithm can efficiently find all itemsets whose coverage is above a given threshold.

2. Use these itemsets to help generate intereresting rules.
   Having done stage 1, we have considerably narrowed down the possibilities, and can do reasonably fast processing of the large itemsets to generate candidate rules.

# Terminology 1

- K-itemset : a set of k items.
  - {beer, cheese, eggs} is a 3-itemset
  - {cheese} is a 1-itemset
  - {honey, ice-cream} is a 2-itemset
- Support: an itemset has support s% if s% of the records in the DB contain that itemset.
- Minimum support: the Apriori algorithm starts with the specification of a minimum level of support, and will focus on itemsets with this level or above.

# Terminology 2

- Large itemset: An itemset whose support is at least minimum support. (also called frequent itemset)
- $L_k$ : the set of all large k-itemsets in the DB.
- $C_k$ : a set of candidate large k-itemsets. In the apriori algorithm will generates this set, which contains all the k-itemsets that might be large, and will then eventually generate the $L_k$ set.

# Terminology 3

- Sets:

  Let A be a set, A = {cat, dog}

  Let B be a set, B = {dog, eel, rat}

  Let C = {eel, rat}

⋆ '$A \cup B$' means A union B. So $A \cup B$ = {cat, dog, eel. rat}

⋆ When X is a subset of Y,

⋆ $Y - X$ means the set of things in Y which are not in X.

⋆ So, $B - C$ = {dog}

# The basket revisited (initials are used instead of full names)

| ID | a, | b, | c, | d, | e, | f, | g, | h, | i |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | 1 | | | 1 | 1 | |
| 2 | | | 1 | 1 | 1 | | | | |
| 3 | | 1 | 1 | | | 1 | | | |
| 4 | | 1 | | | | 1 | | | 1 |
| 5 | | | | | 1 | | 1 | | |
| 6 | | | | | | 1 | | | 1 |
| 7 | 1 | | | 1 | | | | 1 | |
| 8 | | | | | | 1 | | | 1 |
| 9 | | | 1 | | 1 | | | | |
| 10 | | 1 | | | | | 1 | | |
| 11 | | | | | 1 | | 1 | | |
| 12 | 1 | | | | | | | | |
| 13 | | | 1 | | | 1 | | | |
| 14 | | | 1 | | | 1 | | | |
| 15 | | | | | | | | 1 | 1 |
| 16 | | | | 1 | | | | | |
| 17 | 1 | | | | | 1 | | | |
| 18 | 1 | 1 | 1 | 1 | | | | 1 | |
| 19 | 1 | 1 | | 1 | | | 1 | 1 | |
| 20 | | | | | 1 | | | | |

E.g.
3-itemset {a,b,h}
has support 15%

2-itemset {a, i}
has support 0%

4-itemset {b, c, d, h}
has support 5%

If minimum support is 10%, then {b} is a large itemset, but {b, c, d, h} Is a *small* itemset!

# The Apriori Algorithm

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. {  $C_k$ = `apriori-gen`($L_{k-1}$)
4.  For each c in $C_k$, initialise c.count to zero
5.  For all records r in the DB
6.  {$C_r$ = `subset`($C_k$, r); For each c in $C_r$ , c.count++ }
7.  Set $L_k$ := all c in $C_k$ whose count $\geq$ minsup
8. } // end, and return all of the $L_k$ sets.

# The Apriori Algorithm explained

1. Find all large 1-itemsets

To start off, we find all of the large 1-itemsets. This is done by a basic scan of the DB. We take each item in turn, and count the number of times that item appears in a basket. In our running example, suppose minimum support was 30%, then the only large 1-itemsets would be: {a}, {b}, {c}, {d} and {f}. So we get $L_1 = \{\{a\}, \{b\}, \{c\}, \{d\}, \{f\}\}$

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)

We already have $L_1$. This next bit just means that the remainder of the algorithm generates $L_2$, $L_3$ , and so on until we get to an $L_k$ that's empty.

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. $\{$ $C_k = \texttt{apriori-gen}(L_{k-1})$

Given the large (k-1)-itemsets, this step generates some candidate k-itemsets that might be large. Because of how `apriori-gen` works, the set $C_k$ is guaranteed to contain all the large k-itemsets, but also contains some that will turn out not to be 'large'.

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. {  $C_k$ = `apriori-gen`($L_{k-1}$)
4.  For each c in $C_k$, initialise c.count to zero

We will find the support for each of the candidate k-itemsets in $C_k$, by counting how many times each of these itemsets appears in the DB. this step starts us off by initialising these counts to zero.

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. {  $C_k$ = `apriori-gen`($L_{k-1}$)
4.  For each c in $C_k$, initialise c.count to zero
5.  For all records r in the DB
6.  {$C_r$ = `subset`($C_k$, r); For each c in $C_r$ , c.count++ }

We now take each record r in the DB and do this: get all the candidate k-itemsets from $C_k$ that are contained in r. For each of these, update its count.

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. {  $C_k$ = `apriori-gen`($L_{k-1}$)
4.  For each c in $C_k$, initialise c.count to zero
5.  For all records r in the DB
6.  {$C_r$ = `subset`($C_k$, r); For each c in $C_r$ , c.count++ }
7.  Set $L_k$ := all c in $C_k$ whose count $\geq$ minsup

Now we have the count for every candidate. Those whose count is big enough are valid large itemsets of the right size. We therefore now have $L_k$, We now go back into the for loop of line 2 and start working towards finding $L_{k+1}$

# The Apriori Algorithm explained

1. Find all large 1-itemsets
2. For (k = 2 ; while $L_{k-1}$ is non-empty; k++)
3. { $C_k$ = `apriori-gen`($L_{k-1}$)
4. For each c in $C_k$, initialise c.count to zero
5. For all records r in the DB
6. {$C_r$ = `subset`($C_k$, r); For each c in $C_r$ , c.count++ }
7. Set $L_k$ := all c in $C_k$ whose count $\geq$ minsup
8. } // end, and return all of the $L_k$ sets.

We finish at the point where we get an empty $L_k$ . The algorithm returns all of the (non-empty) $L_k$ sets, which gives us an excellent start in finding interesting rules (although the large itemsets themselves will usually be very interesting and useful.

# apriori-gen explained

- Suppose we have worked out that the large 2-itemsets are:

  $L_2 = \{\{\text{milk, noodles}\}, \{\text{milk, shoes}\}, \{\text{noodles, wine}\}\}$
- `apriori-gen` now generates 3-itemsets that all may be large.
- An obvious way to do this would be to generate all of the possible 3-itemsets that you can make from {milk, noodles, shoes, wine}.
- But this would include, for example, {milk, shoes, wine}. Now, if this really was a large 3-itemset, that would mean the number of records containing all three is $\geq$ minsup;
- This means it would have to be true that the number of records containing {shoes, wine} is $\geq$ minsup. But, it can't be, because this is not one of the large 2-itemsets.

# apriori-gen: the join step

- apriori-gen is clever in generating not too many candidate large itemsets, but making sure to not lose any that do turn out to be large.
- To explain it, we assume that is always an ordering of the items, e.g., alphabetical order, and that the data structures used always keep members of a set in this order. a $<$ b will mean that a comes before b in alphabetical order.
- Suppose we have $L_k$ and wish to generate $C_{k+1}$

  First we take every distinct pair of sets in $L_k$

  $\{a_1, a_2, \ldots, a_k\}$ and $\{b_1, b_2, \ldots, b_k\}$, and do this:

  in all cases where $\{a_1, a_2, \ldots, a_{k-1}\} = \{b_1, b_2, \ldots, b_{k-1}\}$, and $a_k < b_k$, $\{a_1, a_2, \ldots, a_k, b_k\}$ is a candidate (k+1)-itemset.

# An illustration of the join step

- Suppose the 2-itemsets are: $L_2 = \{\{\text{milk, noodles}\}, \{\text{milk, shoes}\}, \{\text{noodles, wine}\}, \{\text{noodles, peas}\}, \{\text{noodles, shoes}\}\}$
- The pairs that satisfy: $\{a_1, a_2, \ldots a_{k-1}\} = \{b_1, b_2, \ldots, b_{k-1}\}$, and $a_k < b_k$, are:
  - {milk, noodles} and {milk, shoes}
  - {noodles, peas} and {noodles, wine}
  - {noodles, peas} and {noodles, shoes}
  - {noodles, wine} and {noodles, shoes}
- So the candidate 3-itemsets are: {milk, noodles, shoes}, {noodles, peas, wine} {noodles, peas, shoes}, {noodles, wine, shoes}

# apriori-gen: the pruning step (1/2)

- Now we have some candidate k+1 itemsets, and are guaranteed to have all of the ones that possibly could be large
- But we have the chance to maybe prune out some more before we enter the next stage of Apriori that counts their support (line 5-7).
- In the prune step, we take the candidate k+1 itemsets we have, and remove any for which some k-subset of it is not a large k-itemset.
- Such could not possibly be a large k+1 itemset. (Apriori property)

# apriori-gen: the pruning step (2/2)

- In the current example, we have
  - $L_2 = \{\{\text{milk, noodles}\},\{\text{milk, shoes}\},\{\text{noodles, wine}\}, \{\text{noodles, peas}\},\{\text{noodles, shoes}\}\}$
  - And candidate k+1 itemsets so far: {milk, noodles, shoes}, {noodles, peas, wine},{noodles, peas, shoes}, {noodles, wine, shoes}
- Now,
  - {peas, wine} is not a 2-itemset, so {noodles,peas,wine} is pruned.
  - {peas,shoes} is not a 2-itemset, so {noodles,peas,shoes} is pruned
  - {wine,shoes} is not a 2-itemset, so {noodles,wine,shoes} is pruned.
- After all of these we finally have

  $C_3 = \{\{\text{milk, noodles, shoes}\}\}$

# Making a rule (1/3)

- The Apriori algorithm finds interesting (i.e. frequent) itemsets.
- For example, it may find that {apples, bananas, milk} has support 30% — so 30% of transactions contain each of these three things.
- We can invent several potential rules, e.g.:
- IF basket contains apples and bananas, THEN it also contains milk.
- Suppose support of {apples, bananas} is 40%; what is the confidence of this rule?

# Making a rule (2/3)

- The confidence of a rule 'IF A THEN B' is:

  support(A $\cup$ B) / support(A).
- Suppose itemset A = {beer, cheese, eggs} has 30% support in the DB {beer, cheese} has 40%, {beer, eggs} has 30%, {cheese, eggs} has 50%, and each of beer, cheese, and eggs alone has 50% support.
- What is the confidence of:

  IF basket contains Beer and Cheese, THEN basket also contains Eggs?
- It's 30/40 = 0.75; this rule has 75% confidence.
- What is the confidence of:

  IF basket contains Beer, THEN basket also contains Cheese and Eggs ?
- It is 30/50 = 0.6; so this rule has 60% confidence.

# Making a rule (3/3)

- If the following rule has confidence c: If A then B
- And if support(A) = 2 * support(B),
- What can be said about the confidence of: If B then A
- Confidence c is support(A $\cup$ B) / support(A)
  = support(A $\cup$ B) / 2 * support(B)
- Let d be the confidence of "If B then A".
- d = support(A $\cup$ B) / support(B) – Clearly, d = 2c

# Summary

- We studied the Apriori algorithm for efficiently finding frequent large itemsets in large DBs.
- We learned the associated terminologies (support, confidence, k-itemset, large itemset).
- We learned how to construct rules from the frequent itemsets found.
- We worked out the confidence of a rule based on the support of its itemsets.