COMP5310: Principles of Data Science

W7: Association Rule Mining

Presented by

Maryam Khanian

School of Computer Science

Based on slides by previous lecturers of this unit of study

THE UNIVERSITY OF SYDNEY

The University of Sydney

Last week: Hypothesis testing and evaluation

Objective

- Overview of experimental design and learn Python tools for hypothesis testing and classifier evaluation.

Lecture

- **Significance**
  - Pairwise t-tests.
  - Non-normal data.
- **Evaluation**
  - Classifier evaluation.
  - User evaluation.

Readings

- Data Science from Scratch: Ch 7.
- Model evaluation (sklearn doco).
  - http://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation
- Hypothesis testing (scipy lectures).
  - http://www.scipy-lectures.org/packages/statistics/index.html#hypothesis- testing-comparing-two-groups

Exercises

- Scipy: statistical tests.
- Sklearn: evaluation metrics.

Machine Learning

What is machine learning?

- Creating and using models that are learned from data
  - Predicting whether an email is spam or not
  - Discovering hidden rules in complex datasets
  - Predicting whether a credit card transaction is fraudulent
  - Predicting tumour cells as benign or malignant

Machine Learning Problems

- Prediction
  - Classification and Regression
- Clustering, segmentation and association rules
  - Find patterns in the data
- Outlier/anomaly detection
  - Find unusual patterns
- Reinforcement learning
  - Learn from rewards (like babies)

Supervised vs. Unsupervised Learning

- Supervised learning (e.g. classification and regression)
  - Supervision: The training data are accompanied by **labels** indicating the class of the observations
- Unsupervised learning (e.g. clustering and association rules)
  - The class labels of training data is **unknown**
  - Given a set of measurements, observations, etc. with the aim of
    - Establishing the existence of classes or clusters in the data
    - Discovering hidden patterns or rules

Unsupervised Learning

- We'll focus on unsupervised machine learning techniques
  - ***Association rule mining***
  - Clustering
  - Dimensionality reduction
  - Outlier detection
  - Etc.

Association Rule Mining

Association Analysis

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

How can businesses improve sales by analysing customer purchase data?

Slides adapted from Tan et al. Introduction to data mining.
http://www-users.cs.umn.edu/~kumar/dmbook/
http://www-users.cs.umn.edu/~kumar/dmbook/dmslides/chap6_basic_association_analysis.pdf

Association Rule Mining

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

– Predict occurrence of an item based on other items in the transaction, eg:

```
{Diaper} → {Beer}
{Milk,Bread} → {Eggs,Coke}
{Beer,Bread} → {Milk}
```

– Note that arrows indicate co-occurrence, not causality

Definition: Itemset

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

– An **itemset** is a collection of one or more items
  `{Milk,Bread,Diaper}`

– A **k-itemset** is an itemset containing k items

Definition: Frequent Itemset

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

- **Support count** (σ) is the itemset frequency

  $\sigma(\{\text{Milk}, \text{Diaper}, \text{Beer}\}) = 2$

- **Support** (s) is the normalised itemset frequency

  $$s = \frac{\sigma(\{\text{Milk}, \text{Diaper}, \text{Beer}\})}{|T|} = \frac{2}{5}$$

- A **frequent itemset** has $s \geq min\_support$

Definition: Association Rule

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

- An **association rule** is an implication of the form X→Y where X and Y are itemsets
  `{Milk,Diaper}→{Beer}`
- **Confidence** (c) measures how often Y occurs in transactions with X

$$c = \frac{\sigma(\{\text{Milk,Diaper,Beer}\})}{\sigma(\{\text{Milk,Diaper}\})} = \frac{2}{3}$$

- An **association rule** has c ≥ min_conf

Association Rule Mining Task

- Given a set of transactions T, the goal of association rule mining is to find all rules having
  - support ≥ *min_sup* threshold
  - confidence ≥ *min_conf* threshold

Finding Frequent Itemsets

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

Support {Bread → Milk}

- Let *min_support* = *50%*
- Freq. 1-itemsets:
  - Bread:4(80%); Milk:4(80%); Diaper:4(80%); Beer:3(60%);
- Freq. 2-itemsets:
  - {Bread, Milk}:3(60%)
    {Bread, Diaper}:3(60%)
    {Milk, Diaper}:3(60%)
    {Diaper, Beer}:3(60%)

Finding Association Rules

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Market-basket transactions

TID: Transaction Identifier

Items: Transaction item set

- Let *min_support* = 50%, min_conf = 50%
- Freq. Itemsets:
  - Bread:4; Milk:4; Diaper:4; Beer:3; {Bread, Milk}:3; {Bread, Diaper}:3; {Milk, Diaper}:3; {Diaper, Beer}:3;
- Association rules:
  - Beer → Diaper (100%)
  - Diaper → Beer (75%)
  - …

c(Beer → Diaper) = 3/3

3/4

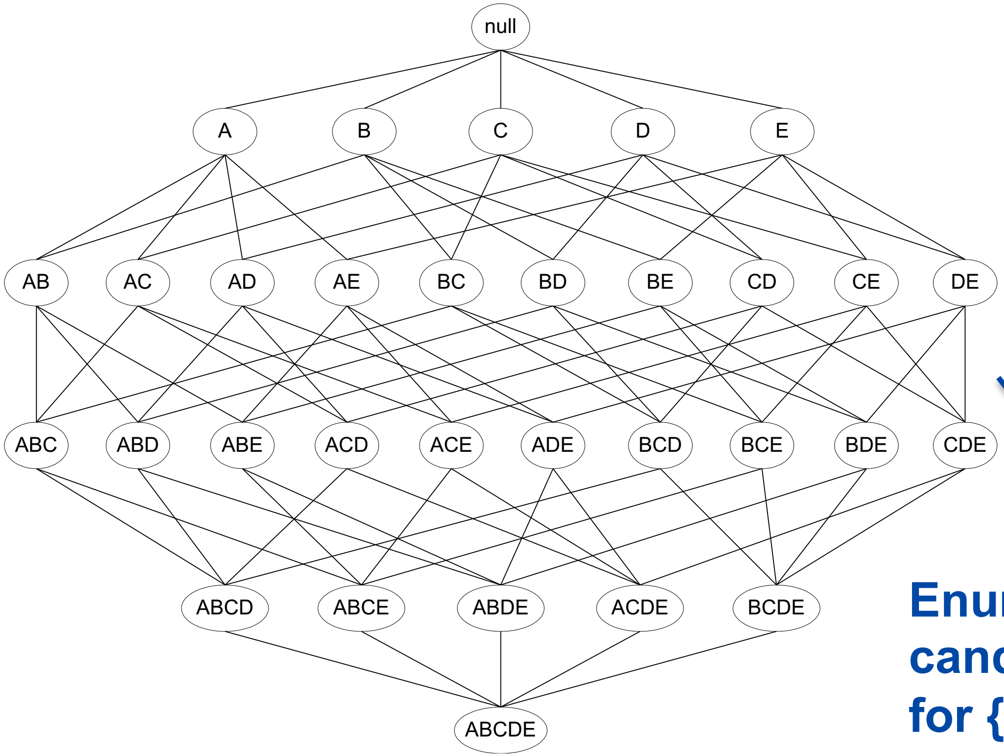
Association Rule Mining with Apriori Principle

Two-step approach

1. **Frequent itemset generation**
   - Generate all itemsets with $s \geq min\_support$
2. **Rule generation**
   - Generate high-confidence rules from each frequent itemset
   - Each rule is a binary partitioning of a frequent itemset

Easy! But brute force enumeration is computationally prohibitive.

There are 2^d candidate itemsets!



Enumeration of 2^5-1 candidate itemsets for {A,B,C,D,E}

Reducing the number of candidates

- **Apriori Principle**
  If an itemset is frequent, then all of its subsets must also be frequent

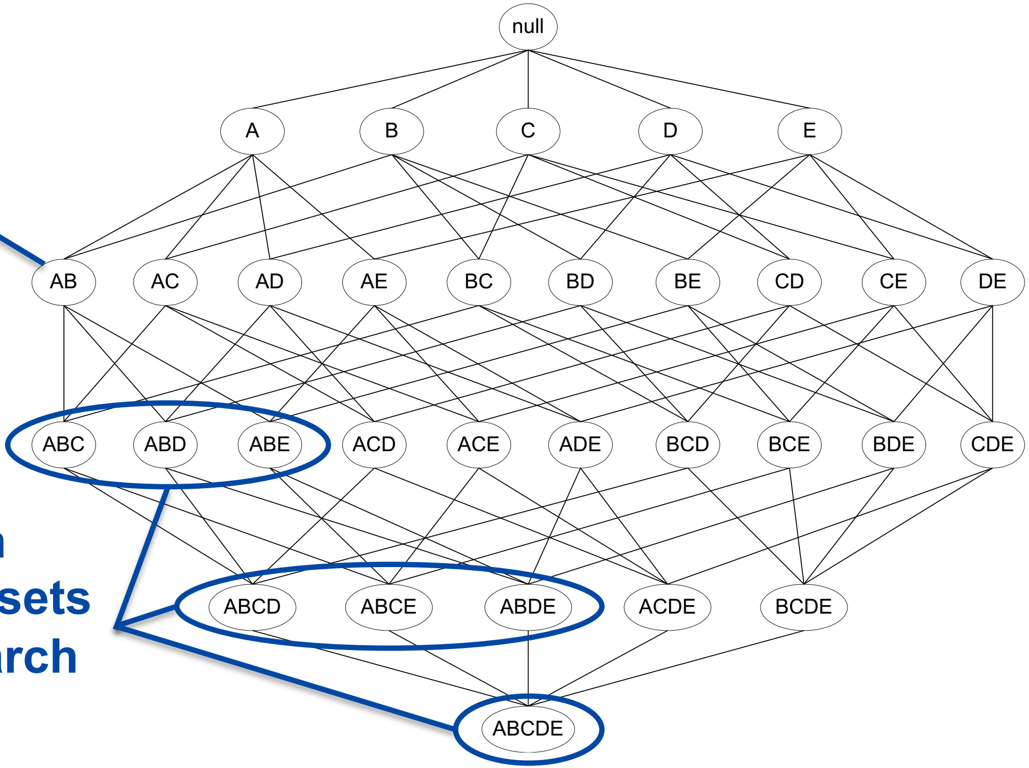
  ?

- **Conversely**
  ?
  If an itemset is infrequent, then its supersets are also infrequent
  - The support of an itemset never exceeds the support of its subsets
    - s({Bread}) $\geq$ s({Bread, Beer})
    - s({Milk}) $\geq$ s({Bread, Milk})
  - This is known as the anti-monotone property of support

Pruning the 2^d candidate itemsets

If *s({A,B})* < *min_support*

Then we can prune supersets from the search space



Apriori algorithm for generating frequent itemsets

While the list of (k-1)-itemsets is non-empty:

  Generate candidate k-itemsets

  Identify and keep frequent k-itemsets

The Apriori Algorithm—An Example

Min_sup = 2

Database TDB

Tid	Items
1	A, C, D
2	B, C, E
3	A, B, C, E
4	B, E

1st scan

C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

2nd scan

C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_3

Itemset	sup
{B, C, E}	2

L_3

Itemset	sup
{B, C, E}	2

Create initial 1-itemsets

Add each item to the initial list of candidate itemsets

```
def createC1(dataset):
    "Create a list of candidate item sets of size one."
    c1 = []
    for transaction in dataset:
        for item in transaction:
            if not [item] in c1:
                c1.append([item])
    c1.sort()
    #frozenset because it will be a key of a dictionary.
    return list(map(frozenset, c1))
```

Sort and return as list of sets

Identify itemsets that meet the support threshold

```
def scanD(dataset, candidates, min_support):
    "Returns all candidates that meet a minimum support level"
    sscnt = {}
    for tid in dataset:
        for can in candidates:
            if can.issubset(tid):
                sscnt.setdefault(can, 0)
                sscnt[can] += 1

    num_items = float(len(dataset))
    retlist = []
    support_data = {}
    for key in sscnt:
        support = sscnt[key] / num_items
        if support >= min_support:
            retlist.append(key)
            support_data[key] = support
    return retlist, support_data
```

Calculate support counts for each candidate

Check whether candidates meet threshold

Generate the next list of candidates

(k-1)-itemsets

Iterate through all pairs of itemsets

Check whether pairs differ by a single item

A|B returns the union of A and B

```python
def aprioriGen(freq_sets, k):
    "Generate the joint transactions from candidate sets"
    retList = []
    lenLk = len(freq_sets)
    for i in range(lenLk):
        for j in range(i + 1, lenLk):
            L1 = list(freq_sets[i])[:k - 2]
            L2 = list(freq_sets[j])[:k - 2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                retList.append(freq_sets[i] | freq_sets[j]) # | is set union
    return retList
```

Generate all Frequent Itemsets

```python
def apriori(dataset, min_support=0.5):
    "Generate a list of candidate item sets"
    C1 = createC1(dataset)
    D = list(map(set, dataset))
    L1, support_data = scanD(D, C1, min_support)
    L = [L1]
    k = 2
    while len(L[k - 2]) > 0:
        Ck = aprioriGen(L[k - 2], k)
        Lk, supK = scanD(D, Ck, min_support)
        support_data.update(supK)
        L.append(Lk)
        k += 1

    return L, support_data
```

Initialise L with frequent 1-itemsets

While the list of (k-1)-itemsets is non-empty:

Generate candidate k-itemsets

Identify frequent k-itemsets

Keep frequent k-itemsets

Rule Generation

- Given a frequent itemset **L**, find all non-empty subsets $\mathbf{x} \subset \mathbf{L}$ such that $\mathbf{x} \rightarrow \mathbf{L} - \mathbf{x}$ satisfies the minimum confidence requirement
  - If {A,B,C,D} is a frequent itemset, candidate rules are:

    A →BCD, B →ACD, C →ABD, D →ABC,

    AB →CD, AC → BD, AD → BC, BC →AD, BD →AC, CD →AB,

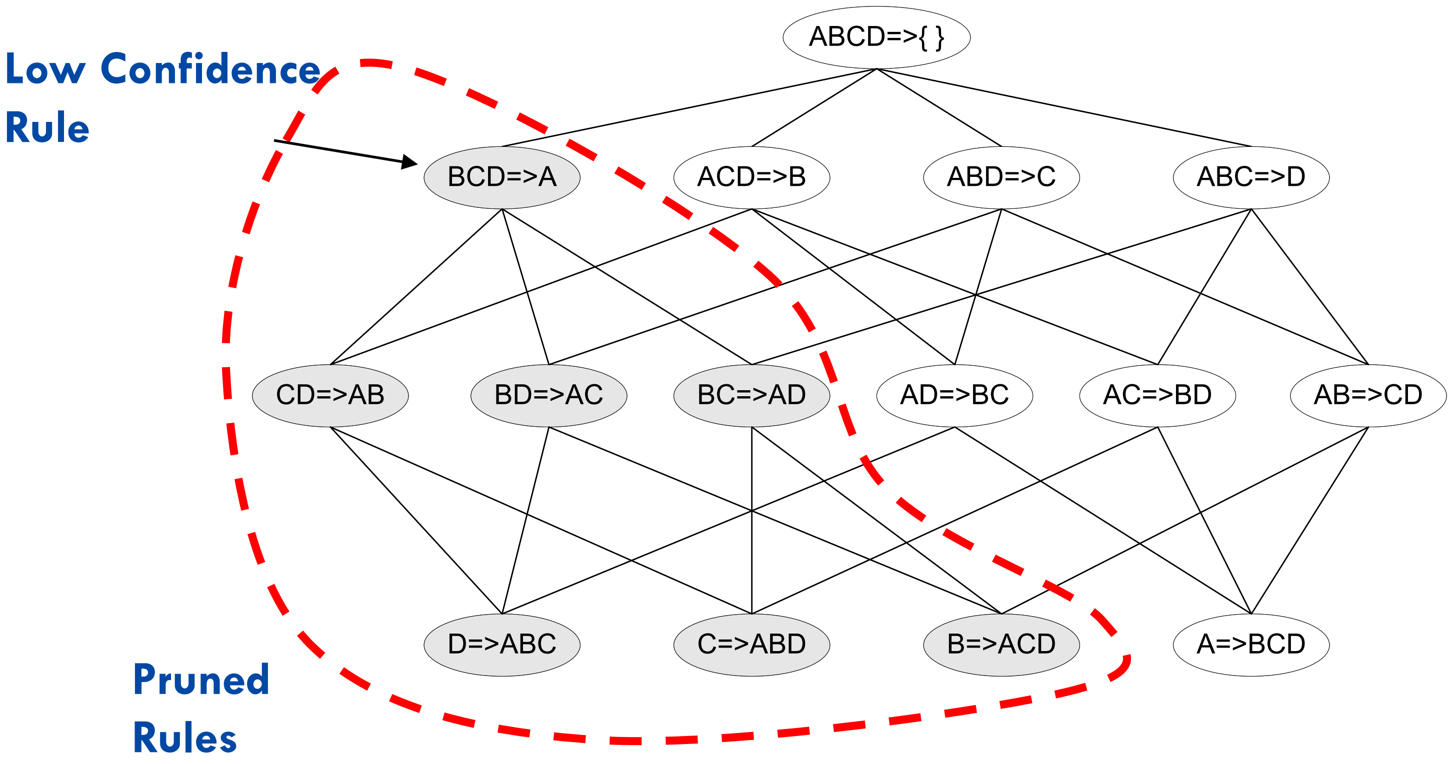
    ABC →D, ABD →C, ACD →B, BCD →A,

- If $|L| = k$, then there are $2^k - 2$ candidate association rules

Efficient Rule generation

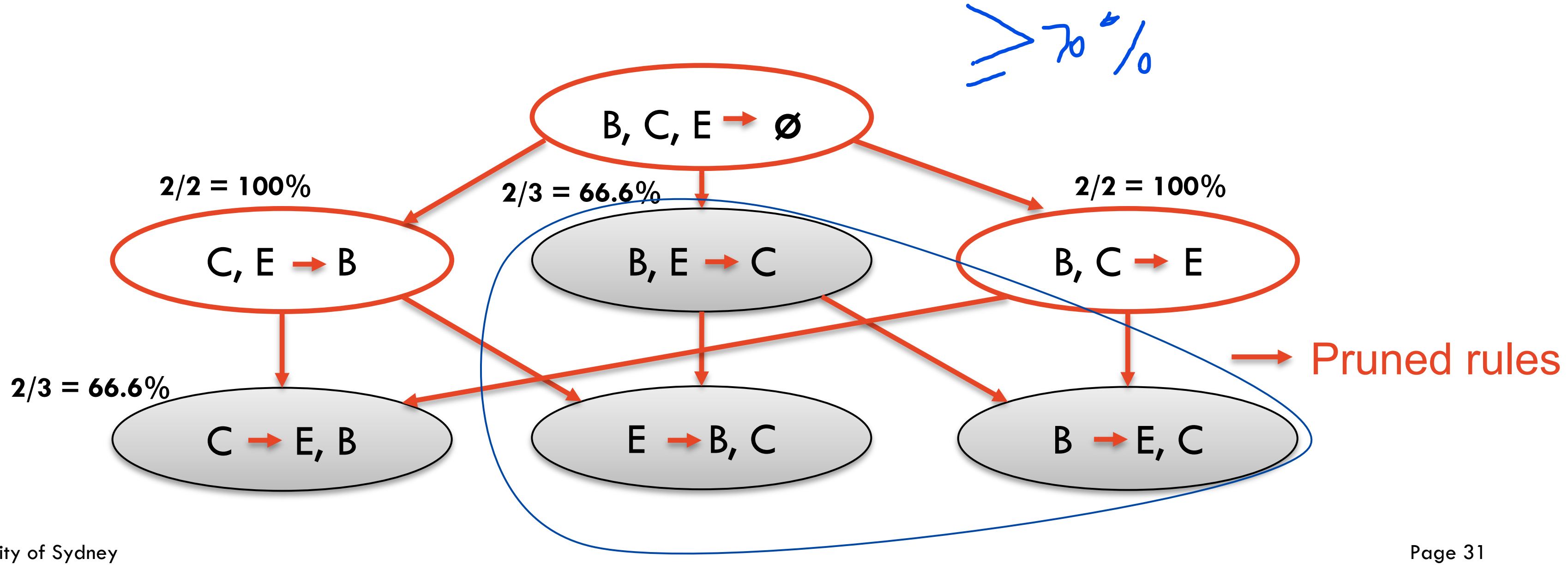
- How to efficiently generate rules from frequent an itemset?
- Similar to support, the confidence of rules generated from the same itemset has an anti-monotone property
- Example: $X = \{A,B,C,D\}$:
  - $c(ABC \rightarrow D) \geq c(AB \rightarrow CD) \geq c(A \rightarrow BCD)$
- Confidence is anti-monotone w.r.t. number of items on the RHS of the rule

Rule Generation for Apriori Algorithm



Association Rule

- From the example before, we had a frequent itemset: {B, C, E}
- From this frequent itemset {B, C, E}, we can have the following association rules



Identify rules that meet the confidence threshold

Frequent itemset (rule components)

Possible consequences (RHS of rule)

Rule accumulator

```
def calc_confidence(freqSet, H, support_data, rules, min_confidence=0.7):
    "Evaluate the rule generated"
    pruned_H = []
    for conseq in H:
        conf = support_data[freqSet] / support_data[freqSet - conseq]
        if conf >= min_confidence:
            #print(freqSet - conseq, '--->', conseq, 'conf:', conf)
            rules.append((freqSet - conseq, conseq, conf))
            pruned_H.append(conseq)
    return pruned_H
```

Calculate confidence

Add rule to accumulator if *conf* ≥ *min_confidence*

Return consequences that pass the confidence threshold

Recursively Evaluate Rules

Need at least 1 item for LHS

Generate candidate consequence itemsets

Update rules and return consequences that pass confidence threshold

```python
def rules_from_conseq(freqSet, H, support_data, rules, min_confidence=0.7):
    "Generate a set of candidate rules"
    m = len(H[0])
    pruned_H = calc_confidence(freqSet, H, support_data, rules, min_confidence)
    if len(freqSet) > (m + 1):
        Hmp = aprioriGen(pruned_H, m + 1)
        if len(Hmp) >= 1:
            rules_from_conseq(freqSet, Hmp, support_data, rules, min_confidence)
```

Recurse with new consequence candidates

Mine all Association Rules

For each k
For each k-itemset

```
def generateRules(L, support_data, min_confidence=0.7):
    """Create the association rules
    L: list of frequent itemsets
    support_data: support data for those itemsets
    min_confidence: minimum confidence threshold
    """
    rules = []
    for i in range(1, len(L)):
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            rules_from_conseq(freqSet, H1, support_data, rules, min_confidence)
    return rules
```

Initial consequence candidates

Recursively evaluate rules

Association Rule Mining with FP-Growth

Performance Bottlenecks of Apriori

- Bottlenecks of *Apriori*:
  - Candidate generation:
    - Generate huge candidate sets
  - Multiple scans of database

Overview of FP-Growth:

- Compress a large database into a compact, *Frequent-Pattern tree* (FP-tree) structure
  - Highly compacted, but complete for frequent pattern mining
  - Avoid costly repeated database scans
- Develop an efficient, FP-tree-based frequent pattern mining method (FP-growth)
  - A divide-and-conquer methodology: decompose mining tasks into smaller ones
  - Avoid candidate generation

Construct FP-tree

Two Steps:

1. Scan the transaction DB for the first time, find frequent items (single item patterns) and order them into a list L in frequency descending order.

2. For each transaction, order its frequent items according to the order in L; Scan DB the second time, construct FP-tree by putting each frequency ordered transaction onto it.

FP-tree Example: step 1

Step 1: Scan DB for the first time to generate L

Min_sup = 2

TID	Items bought
1	*A, C, D*
2	*B, C, E*
3	*A, B, C, E*
4	*B, E*

Item	Frequency
A	2
B	3
C	3
D	1
E	3

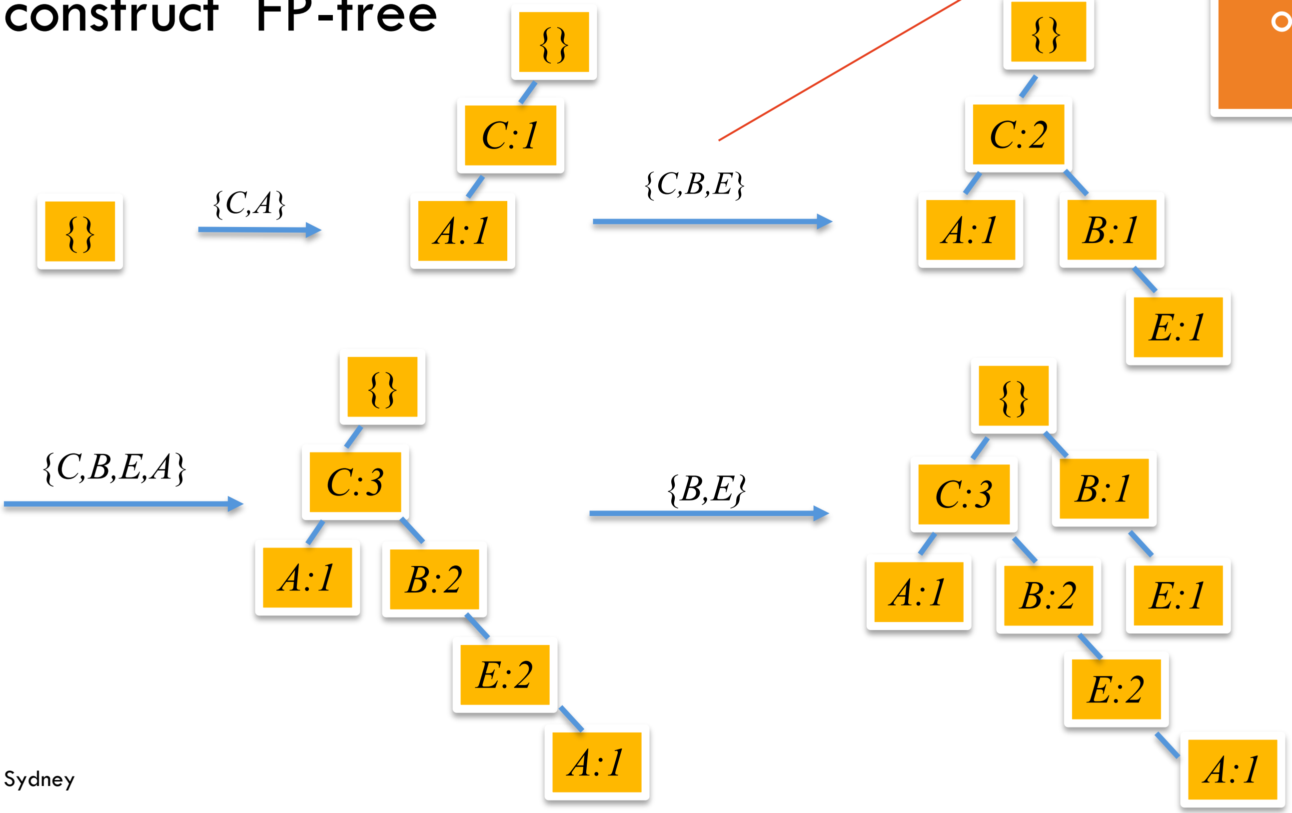
FP-tree Example: step 2

Step 2: scan the DB for the second time, order frequent items in each transaction

TID	Items bought	Ordered frequent items
1	A, C, D	C, A
2	B, C, E	C, B, E
3	A, B, C, E	C, B, E, A
4	B, E	B, E

FP-tree Example: step 2

Step 2: construct FP-tree



NOTE: Each transaction corresponds to one path in the FP-tree

Mining Frequent Patterns Using FP-tree

Starting the processing from the end of list L:

Step 1:

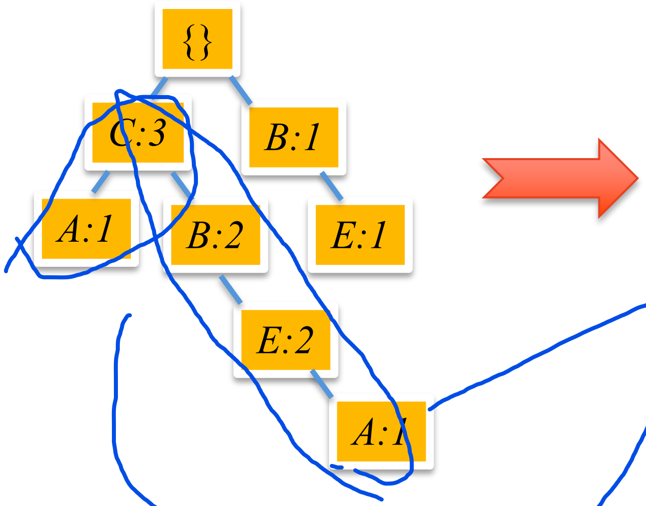
Construct **conditional pattern base** for each item in L

Step 2:

Construct **conditional FP-tree** from each conditional pattern base

Step 1: Construct Conditional Pattern Base

- Traverse the FP-tree by following the link of each frequent item
- Accumulate all of **transformed prefix paths** of that item to form a **conditional pattern base**



***Conditional* pattern bases**

item	cond. pattern base
A	EBC:1, C:1
E	BC:2, B:1
B	C:2
C	{}

Step 2: Construct Conditional FP-tree

- For each pattern base
  - Accumulate the count for each item in the base
  - Construct the conditional FP-tree for the frequent items of the pattern base

***A**-cond.* **pattern base:**		***A**-conditional* **FP-tree**	**Frequent patterns ending with *A*:**
EBC:1, C:1	→	**{ (C:2) } | A**	**CA:2**

Step 2: Construct Conditional FP-tree

E-*cond.* **pattern base:**

BC:2, B:1

→

E-*conditional* **FP-tree**

{ (B:3, C:2, BC:2) } | E

Frequent patterns ending with *E*:

BE:3, CE:2, BCE:2

B-*cond.* **pattern base:**

C:2

→

B-*conditional* **FP-tree**

{ (C:2) } | B

Frequent patterns ending with *B*:

CB:2

All Frequent Itemsets: { A:2, E:3, B:3, C:3, BC:2, EC:2, AC:2, EB:3, BCE:2 }

Review

W7 review: Association Rule Mining

Objective

Learn techniques for unsupervised learning, with tools in Python.

Lecture

- Association rule mining

Readings

- Intro to Data Mining, Ch. 5
  https://www-users.cse.umn.edu/~kumar001/dmbook/ch5_association_analysis.pdf
- Data Science from Scratch, Ch. 11

Exercises

- Associations from scratch
- Associations using mlxtend
- Associations using pyfpgrowth