CS 696 - Notes Oct. 22, 2025

1 Apriori Algorithm Example

1.1 The Setup

Here are the 5 shopping transactions, now including Veggies:

- T1: {Bread, Milk}
- T2: {Bread, Diapers, Veggies, Eggs}
- T3: {Milk, Diapers, Veggies, Cola}
- T4: {Bread, Milk, Diapers, Veggies}
- T5: {Bread, Milk, Diapers, Cola}

Our `min_support = 3` (meaning an itemset must appear in at least 3 transactions).

1.2 Step 1: Find Frequent 1-Itemsets (L1)

First, we count every individual item.

Item	**Count (Support)**
{Bread}	4
{Milk}	4
{Diapers}	4
{Veggies}	3
{Eggs}	1
{Cola}	2

Now, we prune the list, keeping only those with a support of 3 or more.
Frequent 1-Itemsets (L1):

- {Bread}: 4
- {Milk}: 4
- {Diapers}: 4
- {Veggies}: 3

(We remove {Eggs} and {Cola}.)

1.3 Step 2: Find Frequent 2-Itemsets (L2)

1. **Generate:** We create candidate pairs (C2) using only the frequent items from L1.
   - {Bread, Milk}
   - {Bread, Diapers}
   - {Bread, Veggies}
   - {Milk, Diapers}
   - {Milk, Veggies}
   - {Diapers, Veggies}
2. **Count:** We scan the original 5 transactions again to count these pairs.
   - {Bread, Milk}: 3 (T1, T4, T5)
   - {Bread, Diapers}: 3 (T2, T4, T5)
   - {Bread, Veggies}: 2 (T2, T4)
   - {Milk, Diapers}: 3 (T3, T4, T5)
   - {Milk, Veggies}: 2 (T3, T4)
   - {Diapers, Veggies}: 3 (T2, T3, T4)
3. **Prune:** We keep only the pairs with `min_support = 3`.

Frequent 2-Itemsets (L2):

- {Bread, Milk}: 3
- {Bread, Diapers}: 3
- {Milk, Diapers}: 3
- {Diapers, Veggies}: 3

(We remove {Bread, Veggies} and {Milk, Veggies}.)

1.4 Step 3: Find Frequent 3-Itemsets (L3)

1. **Generate:** We create candidate 3-itemsets (C3) by combining L2 sets.
   - {Bread, Milk} + {Bread, Diapers} $\rightarrow$ {Bread, Milk, Diapers}
   - {Milk, Diapers} + {Diapers, Veggies} $\rightarrow$ {Milk, Diapers, Veggies}
2. **Prune (The "Apriori Property"):** We check if all subsets of a candidate are also in L2.
   - **Candidate: {Bread, Milk, Diapers}**
     - Subsets: {Bread, Milk}, {Bread, Diapers}, {Milk, Diapers}
     - Are all these in L2? **Yes.** $\rightarrow$ We keep this candidate for counting.
   - **Candidate: {Milk, Diapers, Veggies}**

  - Subsets: {Milk, Diapers}, {Milk, Veggies}, {Diapers, Veggies}
  - Are all these in L2? **No.** The subset {Milk, Veggies} was *not* in L2 (it only had a support of 2).
  - We prune {Milk, Diapers, Veggies} without even counting it.

3. **Count:** We only need to count the one remaining candidate.
   - {Bread, Milk, Diapers}: 2 (T4, T5)
4. **Prune:** The count (2) is less than our `min_support` (3). We remove it.

Frequent 3-Itemsets (L3):

- {} (Empty)

1.5 Final Result

The algorithm stops because L3 is empty. The **final frequent itemsets** found by the algorithm are:

- {Bread}
- {Milk}
- {Diapers}
- {Veggies}
- {Bread, Milk}
- {Bread, Diapers}
- {Milk, Diapers}
- {Diapers, Veggies}

2 Rule-Based Classification

Here's a simple example of rule-based classification.

2.1 The Goal

Let's predict whether a person will buy a new video game (Yes or No) based on a few factors. This "Yes" or "No" is our target class.

2.2 The Data

We have a small dataset of past customer behavior:

Customer	Owns Console?	Is Pre-order?	Is Game on Sale?	Bought Game?
1	Yes	Yes	No	Yes
2	Yes	No	Yes	Yes
3	No	Yes	No	No
44	Yes	No	No	No
5	No	No	Yes	No
6	Yes	Yes	Yes	Yes

2.3 The "IF-THEN" Rules

A rule-based classifier (like a decision tree or algorithms like RIPPER) scans this data to generate a set of simple, human-readable rules. The goal is to create the smallest, most accurate set of rules possible.

After "learning" from the data, the algorithm might generate a rule set like this:

- **Rule 1:** IF (Is Pre-order? = Yes) AND (Owns Console? = Yes) THEN Bought Game? = Yes
- **Rule 2:** IF (Is Game on Sale? = Yes) AND (Owns Console? = Yes) THEN Bought Game? = Yes
- **Default Rule:** OTHERWISE, Bought Game? = No

This rule set is the "model" itself.

2.4 How to Classify New Data

Now, let's get two new customers and use our rules to classify them. The rules are applied in order.

New Customer A:

- Owns Console? = Yes
- Is Pre-order? = No
- Is Game on Sale? = Yes

Classification Process:

1. **Check Rule 1:** (Is Pre-order? = Yes) is **False**. Rule does not apply.
2. **Check Rule 2:** (Is Game on Sale? = Yes) is **True** AND (Owns Console? = Yes) is **True**.
    - The rule applies.
    - **Prediction: Bought Game? = Yes**

New Customer B:

- Owns Console? = No
- Is Pre-order? = No
- Is Game on Sale? = Yes

Classification Process:

1. **Check Rule 1:** (Is Pre-order? = Yes) is **False**. Rule does not apply.
2. **Check Rule 2:** (Is Game on Sale? = Yes) is **True** BUT (Owns Console? = Yes) is **False**.
    - The rule does not apply.
3. **Check Default Rule:** No other rules matched.
    - The default rule applies.
    - **Prediction: Bought Game? = No**

3 Stochastic Gradient Descent

Let's use a simple analogy: You are at the top of a large, foggy mountain, and you need to find the lowest point (the valley). Your "model" is your set of instructions for which way to step, and the "error" (or loss function) is your current altitude. Your goal is to minimize your altitude.

3.1 The Goal: Find the Best "Weight"

- **Task:** Predict a student's final **Exam Score** (Y) based on how many **Hours they Studied** (X).
- **Model:** A simple line: `Predicted_Score = Weight * Hours_Studied`
- **Goal:** Find the *single best value* for `Weight` that makes the most accurate predictions.

Let's say we have 10,000 students in our dataset. We'll start with a random guess: `Weight = 5`.

3.2 Method 1: "Batch" Gradient Descent (The Slow, Careful Way)

This is the non-stochastic way. "Batch" means "all at once."

1. **Look Everywhere:** You ask *all 10,000 students* to stand on the mountainside.
2. **Calculate Average Slope:** You calculate the error (prediction vs. actual score) for *every single student*. Then, you average all 10,000 of these errors to find the "true" average downhill slope (the gradient).
3. **Take One Step:** Based on this perfect average, you take *one* careful step downhill. (e.g., You update `Weight` from 5 to 5.1).
4. **Repeat:** You must now ask all 10,000 students again for the next single step.

Problem: This is incredibly slow. You do 10,000 calculations just to make *one* tiny update to your `Weight`.

3.3 Method 2: Stochastic Gradient Descent (SGD) (The Fast, Noisy Way)

"Stochastic" means "random," or in this case, "one at a time."

1. **Shuffle Data:** First, you randomly shuffle your 10,000 students.
2. **Pick One Student:** You look at *only the first student* in the shuffled list.
   - **Data:** Student 1 studied **4 hours** and got a **90**.
3. **Calculate Slope for One:** You calculate the error *just for this one student.*

- **Prediction:** `Predicted_Score = 5 * 4 = 20`
- **Error:** This prediction (20) is *way* too low! The actual score was 90.

4. **Take One Step:** The slope for *this one student* tells you to "increase `Weight` a lot!" So you immediately update your model. (e.g., `Weight` is now 5.8).

5. **Pick the Next Student:** You look at Student 2.
   - **Data:** Student 2 studied **2 hours** and got a **30**.

6. **Calculate Slope for One:** You use your *new* weight.
   - **Prediction:** `Predicted_Score = 5.8 * 2 = 11.6`
   - **Error:** This is also too low (actual was 30).

7. **Take Another Step:** The slope for *this student* tells you to "increase `Weight` again, but not as much." (e.g., `Weight` is now 6.1).

8. **Repeat...** You do this for all 10,000 students, one by one.

3.4 Summary: Batch vs. SGD

Imagine the valley is in the center of a map.

- **Batch Gradient Descent** calculates the *perfect* path downhill and takes one, slow, accurate step. It repeats this, drawing a smooth, direct line to the valley.
- **Stochastic Gradient Descent (SGD)** takes a "drunken walk." It looks at one person, takes a step. Looks at the next, takes a step. Sometimes a weird student (an outlier) will make it step in the wrong direction. But *on average*, it zig-zags its way to the valley much, much faster.

In the time it takes "Batch" to take **one** perfect step, SGD has already taken **10,000** small, wobbly steps and is probably already at the bottom of the valley. That's why it's the foundation for training almost all large models, like neural networks.

4 Bayes' Rule Example

$$P(A\&B) = P(A \mid B) \cdot P(B)$$

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

4.1 The Scenario

Imagine a rare disease that only **1%** of the population has. You take a test for this disease that is **99% accurate**.

This "99% accuracy" means two things:

1. **True Positive Rate:** If you *have* the disease, the test will be positive 99% of the time. (This is $P(+ \mid \text{Disease})$)
2. **True Negative Rate:** If you do *not* have the disease, the test will be negative 99% of the time.

This also means the test has a **1% False Positive Rate** (it will incorrectly say a healthy person is sick 1% of the time). This is $P(+ \mid \text{No Disease})$.

The Question: You test positive. What is the probability you *actually* have the disease?

Your intuition might say 99%, but let's use Bayes' rule. The easiest way is to think about a large group of people.

—

4.2 The Calculation (Using 10,000 People)

Let's test **10,000** people.

1. **How many are sick?**
   - The disease rate is 1%.
   - 1% of $10,000$ = **100 people** are actually sick.
2. **How many are healthy?**
   - 99% of $10,000$ = **9,900 people** are healthy.
3. **Now, let's find all the positive tests:**
   - **True Positives:** We test the 100 sick people. The test is 99% accurate.
     - 99% of 100 = **99 people** (Sick people who correctly test positive).
   - **False Positives:** We test the 9,900 healthy people. The test has a 1% false positive rate.
     - 1% of $9,900$ = **99 people** (Healthy people who incorrectly test positive).
4. **Find the Answer:**
   - Total people who tested positive = (True Positives) + (False Positives)

- Total Positive Tests $= 99 + 99 =$ **198 people**.
- Out of these 198 people who *tested* positive, only 99 *are* actually sick.

So, the probability that you *actually* have the disease given your positive test is:

$$P(\text{Disease} \mid +) = \frac{\text{True Positives}}{\text{Total Positives}} = \frac{99}{198} = 0.5$$

—

4.3 The Result

Even with a 99% accurate test, a positive result only means you have a **50% chance** of actually having the disease.

This surprising result happens because the **prior probability** (the base rate of the disease) was so low (1%). The number of healthy people getting false positives (99) was equal to the number of sick people getting true positives (99).

Bayes' rule is just the formal math for this exact logic:

$$P(D \mid +) = \frac{P(+ \mid D) \cdot P(D)}{P(+)} = \frac{0.99 \cdot 0.01}{(0.99 \cdot 0.01) + (0.01 \cdot 0.99)} = \frac{0.0099}{0.0099 + 0.0099} = \frac{0.0099}{0.0198} = 0.5$$