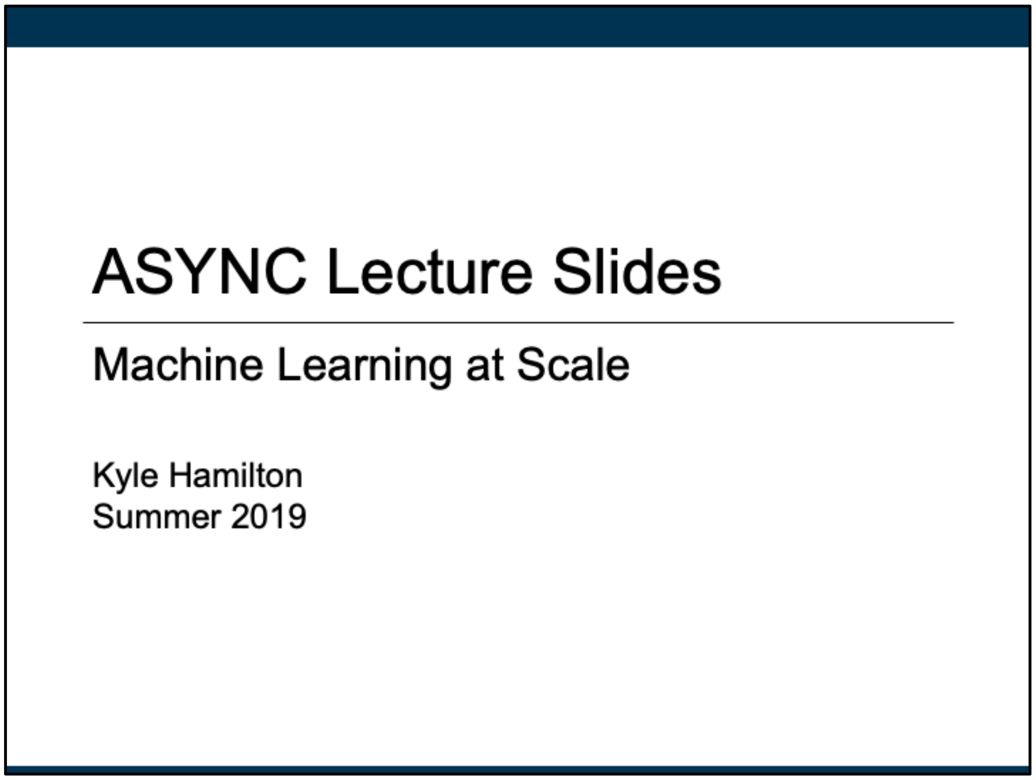
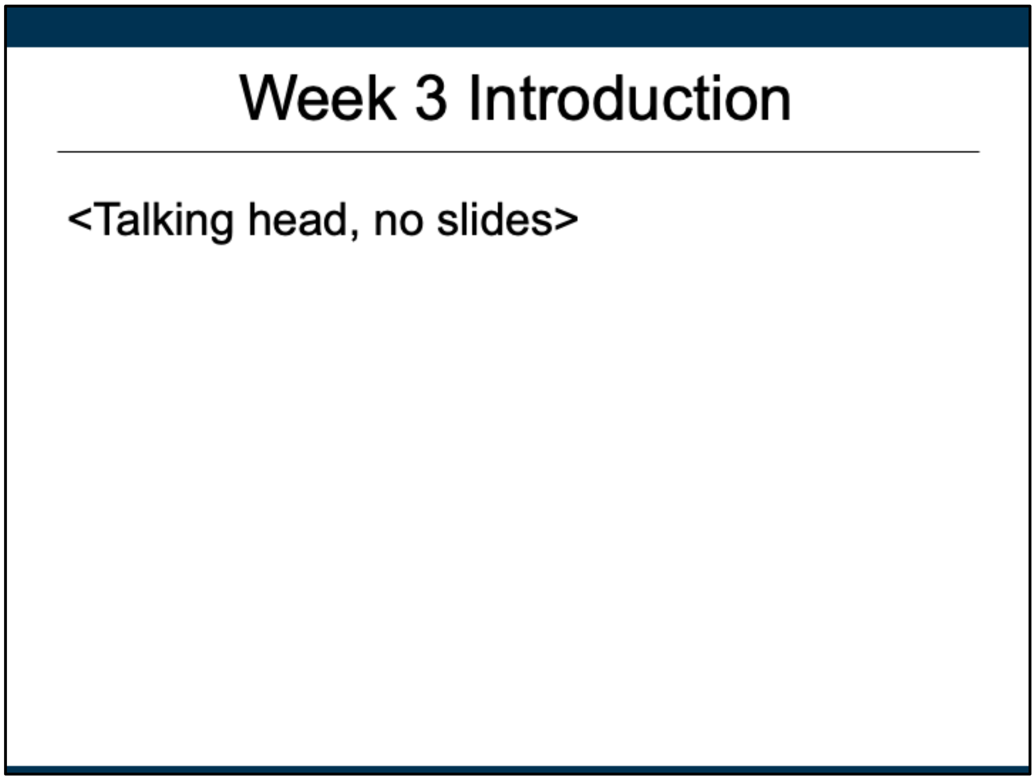


ASYNC Lecture Slides

Machine Learning at Scale

Kyle Hamilton
Summer 2019



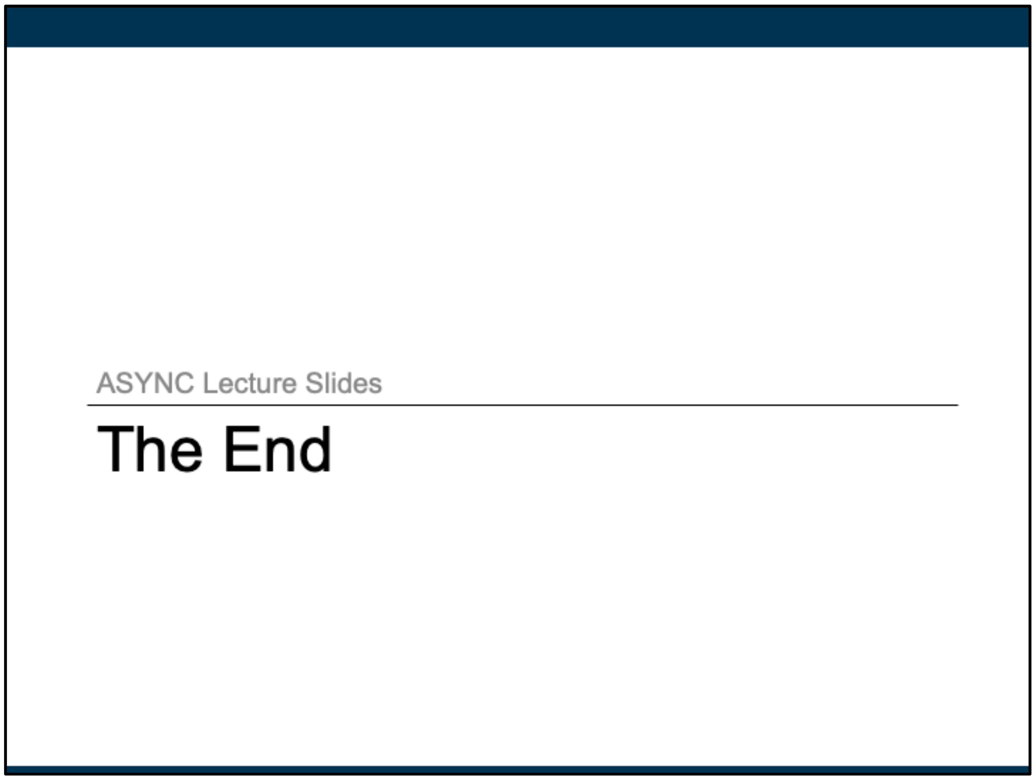
Week 3 Introduction

<Talking head, no slides>

A large part of the power of the MapReduce framework comes from its simplicity. The contract with the programmer is as follows. The program provides the data, specifies the mapper function, the reducer function, and then has the option to specify a combiner and also a partitioner. All the other aspects of execution are handled by the execution framework transparently.
The purpose of this lecture is to provide, primarily through example, a guide to MapReduce algorithm design. The examples that we'll cover in this lecture can be thought of as design patterns for MapReduce. They will help us design our mappers, our reducers, and see if we want to have combiners as part of that arrangement. They will also help us understand when to partition our data and how to partition our data. And these patterns will serve us well as we look at all aspects of machine learning in terms of algorithmic design.

Now, they will also cause us to think very differently about the relationship between memory, disk space, and network bandwidth. More concretely, we're going to look at the relationship between memory, disk, and bandwidth. We'll look at various data structures that will be leveraged in the MapReduce framework. And in particular, we're going to look at the Hadoop shuffle. This is the backbone of Hadoop, as we shall learn today.
We'll look at design patterns around local aggregation. And we'll look at order inversion as another design pattern that will be actually leveraged later for naive Bayes classification learning. We'll look at secondary sorting and how that plays a key

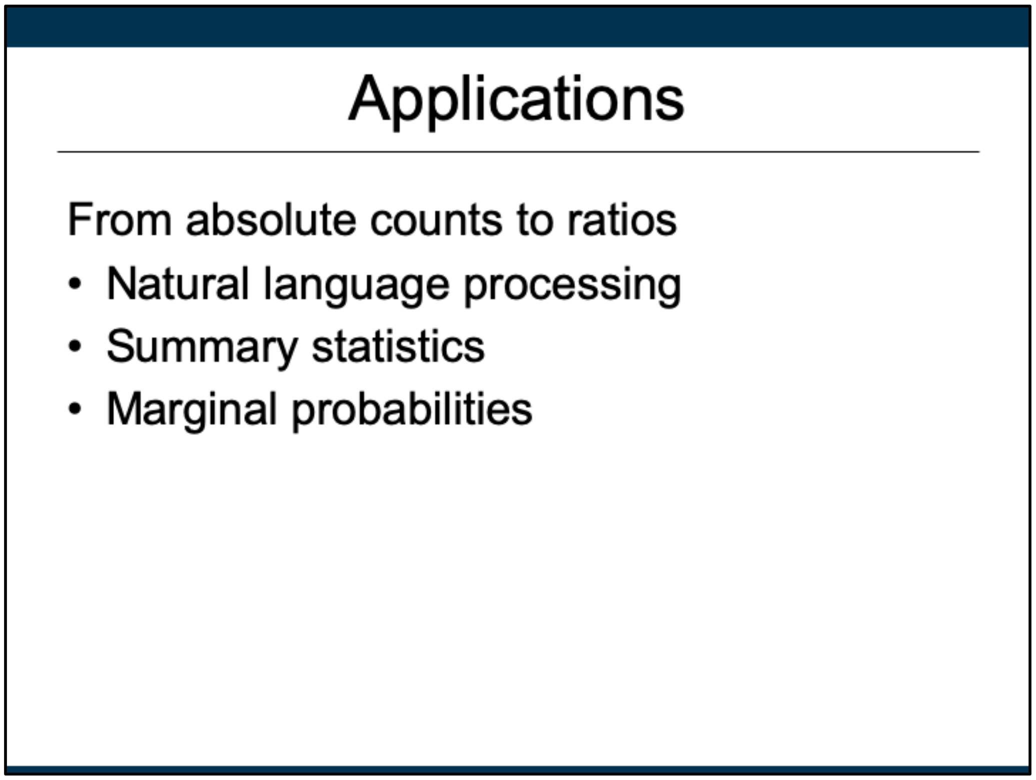
role in a lot of the algorithms that we'll discuss in future classes. And we'll also talk about some other scaling tricks that will be useful in future as well.



ASYNC Lecture Slides

The End

Computing
Relative Frequencies

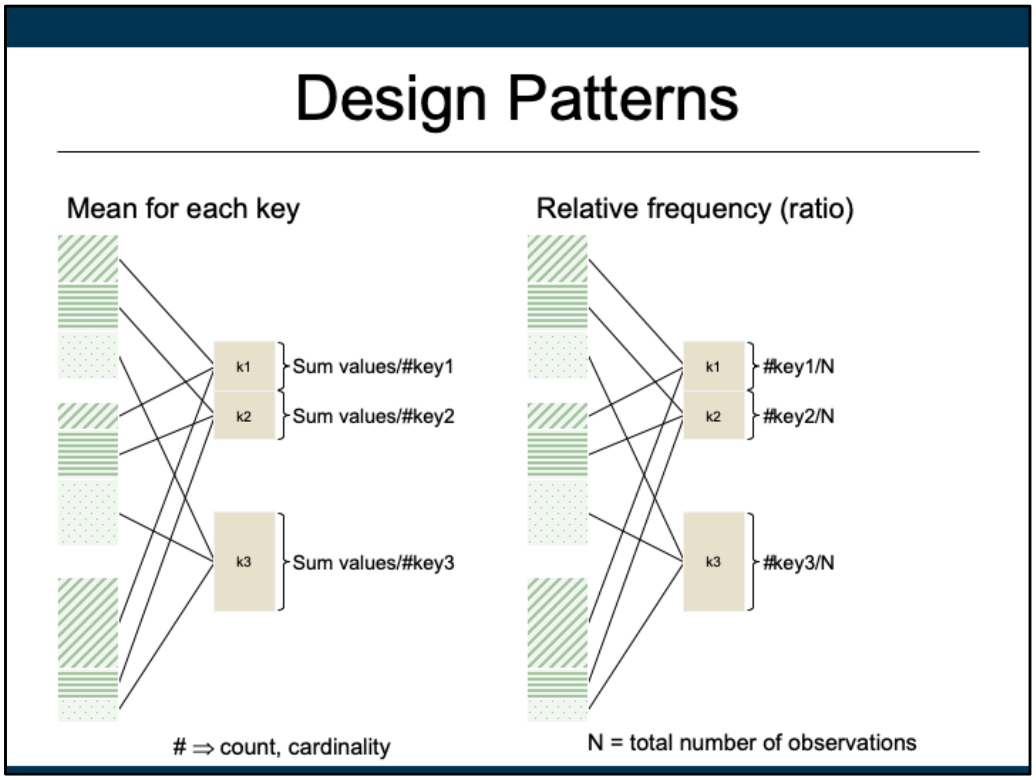


Applications

From absolute counts to ratios

- Natural language processing
- Summary statistics
- Marginal probabilities

Now that we are expert word counters at scale, what if we wanted to know not only the absolute counts, but also the ratios with which the words occur?
For example, if the word “China” occurs 10 times in a 100 word document, that’s no the same as if “China” occured 10 times in a 10,000 word document. Intuitively, we could assume that the shorter document is about “China”, but the longer one not necessarily so.
Even though we won’t be delving too deep into Natural Language Processing, we want to make sure that we know the fundamental design patterns that enable this kind of calculation at scale.
Other examples:
Natural Language Processing
Summary statistics
Marginal probabilities

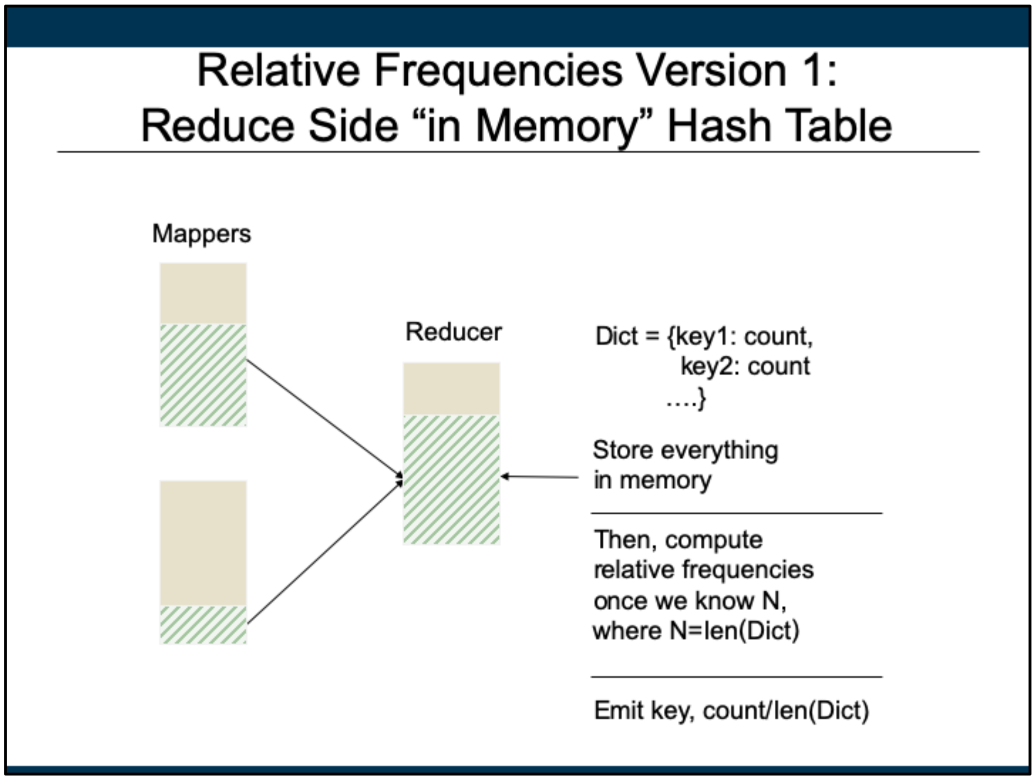


Design Patterns

Earlier we looked at an example where we calculated the mean of the values for a given key. For each key, we tallied the sum of the values and the count of the number of observations in the reducer, and when all values for a given key have been seen, we performed the division. In this section we'll see how to tally the counts of all observations in the dataset, not just the ones for a given key.

We'll take a look at a couple of ways to achieve this goal. Our final attempt will introduce the 'order inversion' pattern.
For now, we'll limit our discussion to a single reducer. Later we'll extend our solution to include multiple reducers when we introduce custom partitioning. But don't worry about that yet!

In general, this is a pattern for calculating results where the total number of observations comes into the equation.



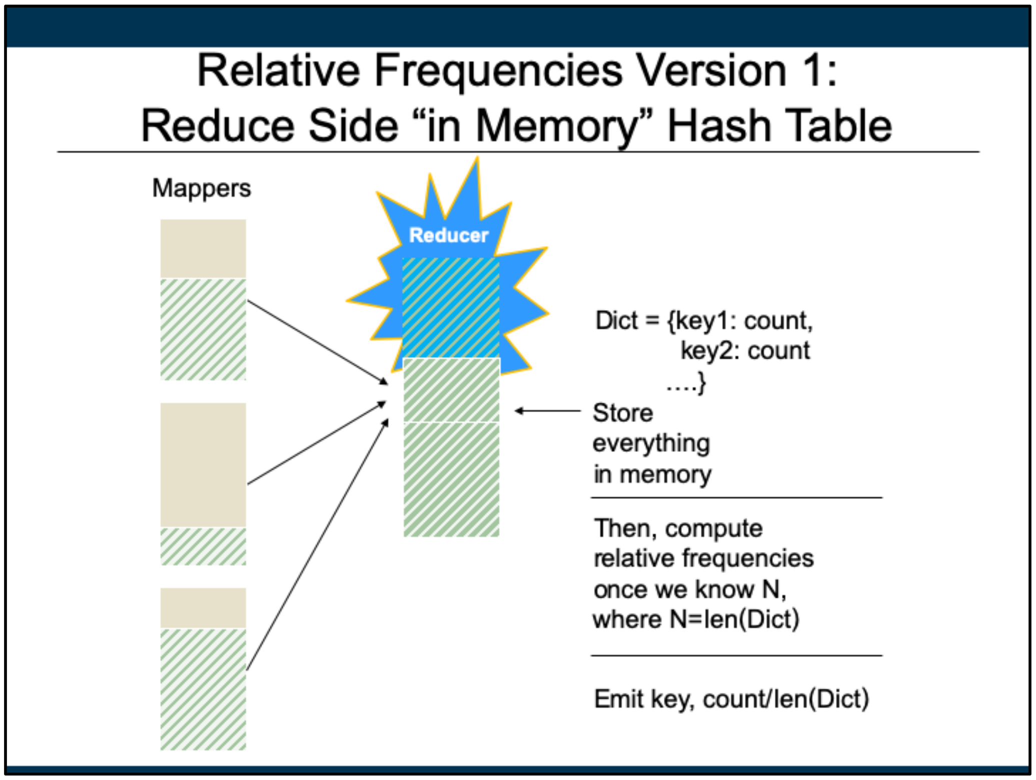
Relative Frequencies Version 1: Reduce Side "in Memory" Hash Table

Mappers
Reducer
Dict = {key1: count,
key2: count
….}
Store everything in memory
Then, compute relative frequencies once we know N, where N=len(Dict)
Emit key, count/len(Dict)

Let's start by building on what we learned when we were calculating means. We want to know how many times a key occurs out of the total number of observations in the dataset. Previously, we tallied the values and number of observations per key; now we will tally the number of observations per key, as well as the total number of observations in the dataset.

The problem is that we won't know the total number of observations until our reducer has seen all of the data! So… we'll need to store the key counts for all the keys in memory, as well as the total. Only at the very end can we divide each key count by the total.

Will this always work?



Relative Frequencies Version 1: Reduce Side "in Memory" Hash Table

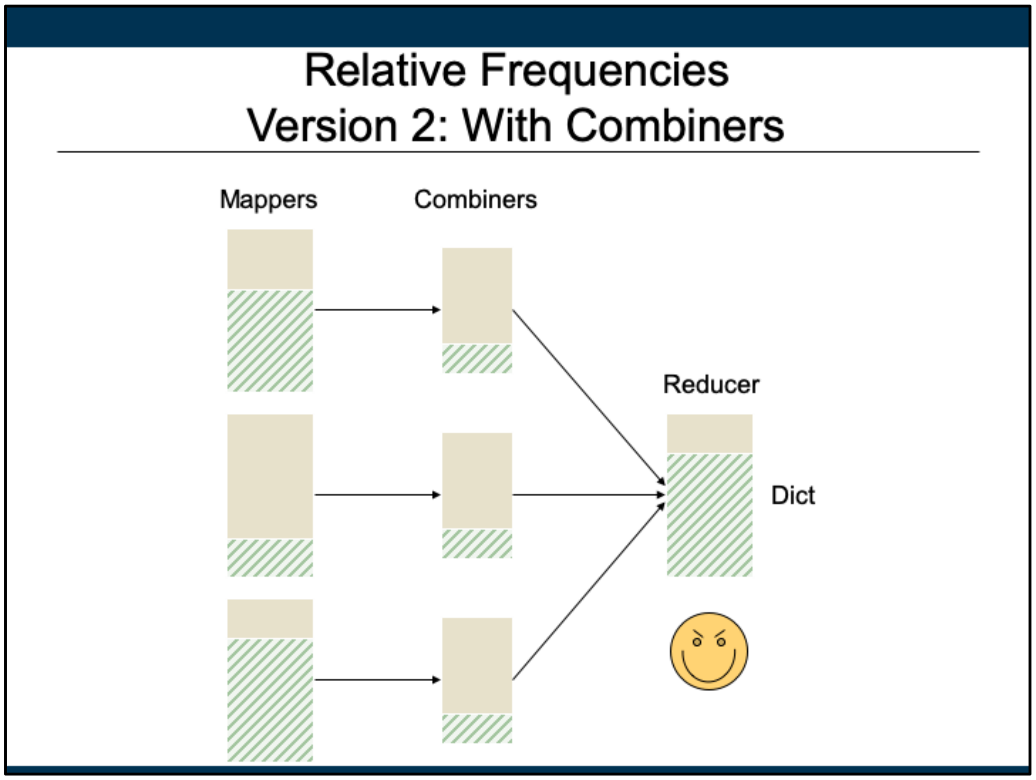
Dict = {key1: count,
key2: count
….}

Store everything in memory

Then, compute relative frequencies once we know N, where N=len(Dict)

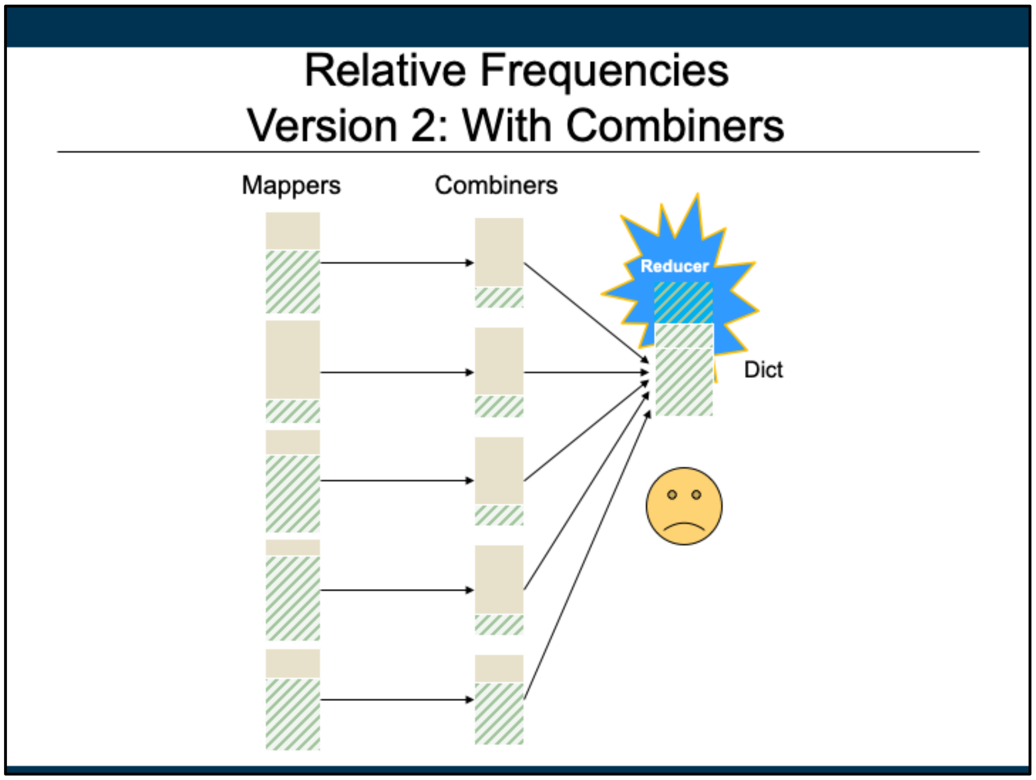
Emit key, count/len(Dict)

Oh no!! What if our dictionary won't fit in the reducer?
Can we do better?
YES!



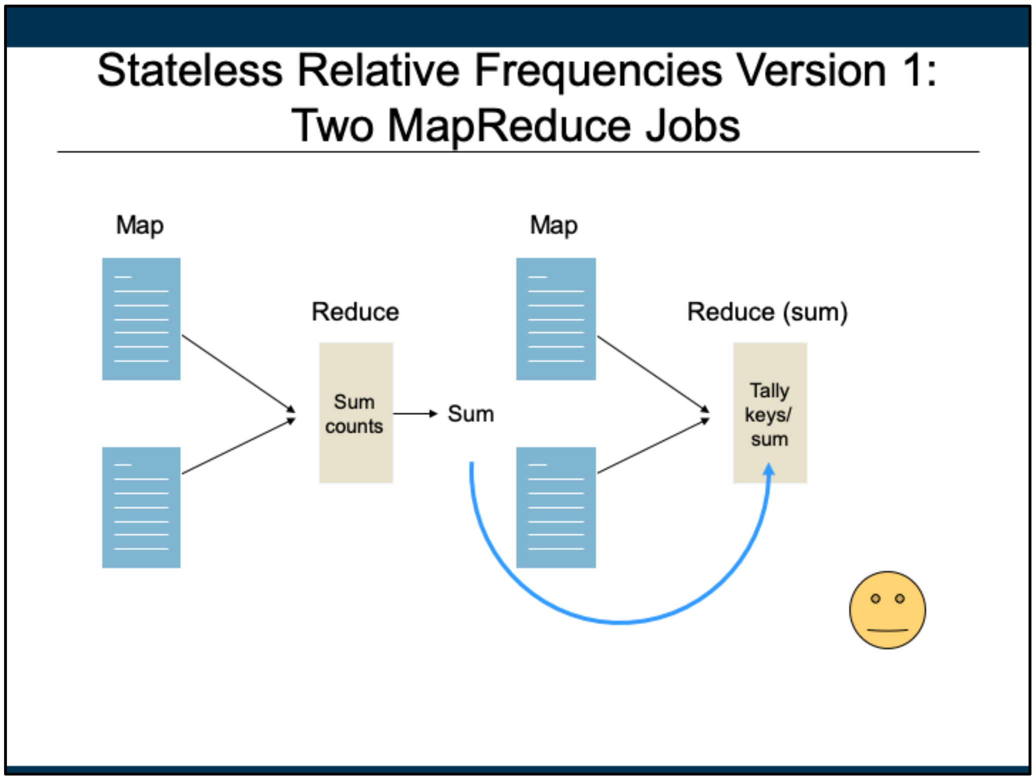
Relative Frequencies
Version 2: With Combiners

We can optimize our example by utilizing local aggregations and combiners
The question is: is this sufficient to scale to any sized dataset?

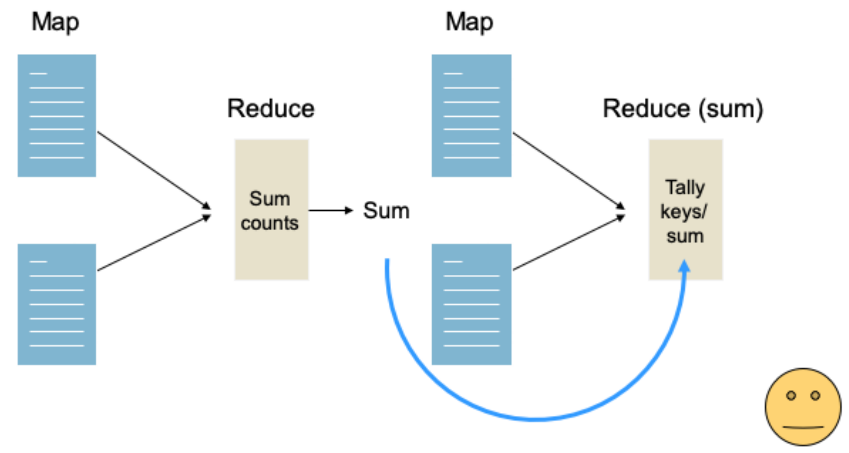


Relative Frequencies
Version 2: With Combiners

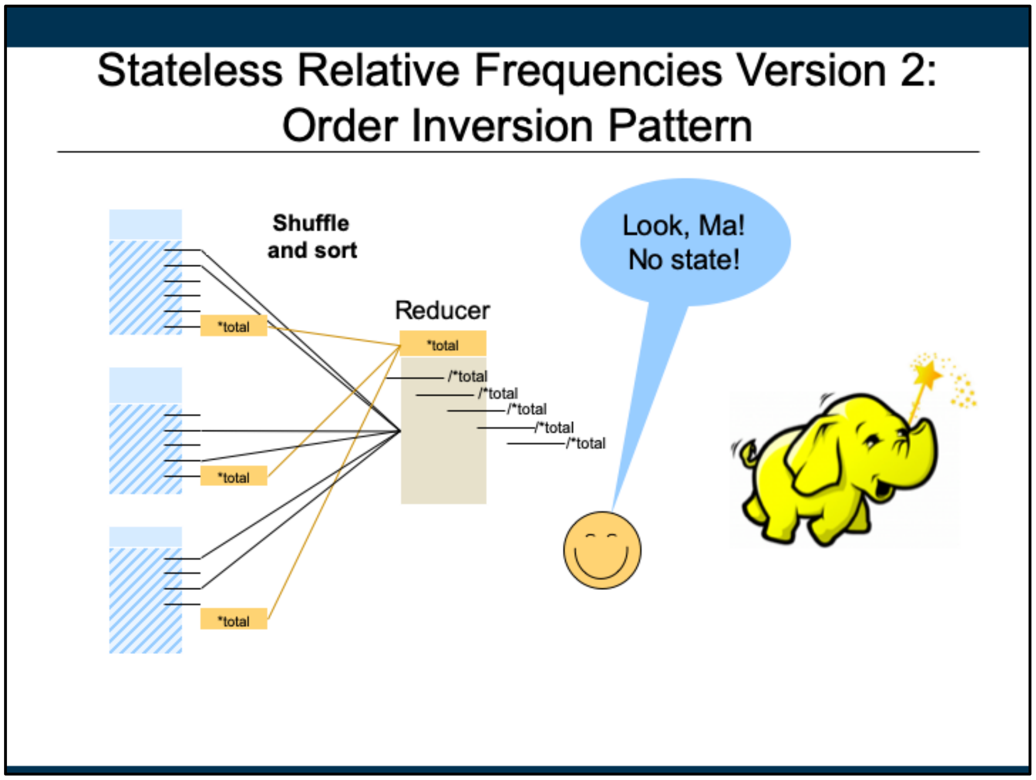
What if the number of keys was very large, ie., larger than could fit into the memory of a single reducer?



Stateless Relative Frequencies Version 1: Two MapReduce Jobs



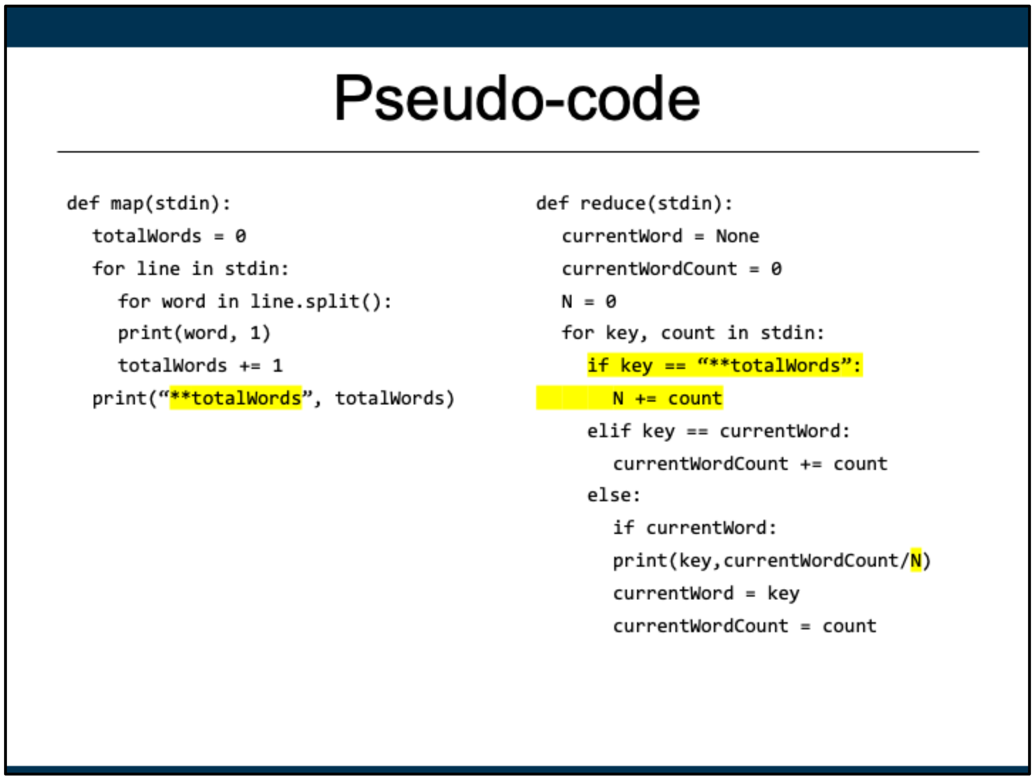
What if we knew the total number of observations ahead of time? We wouldn't need to keep anything (except the count of one key) in memory.
A naive approach might be to run 2 map reduce jobs, one that calculates the total count, and a second one that calculates the relative frequency for each key and emits the results as soon as all rows for a given key have been seen. This is a highly "scalable" approach in the sense that it doesn't matter how large the dataset is, it's guaranteed to work. However, if the dataset is that large, it's not very "efficient" to have to scan that dataset twice.



Stateless Relative Frequencies Version 2: Order Inversion Pattern

It turns out we can leverage the framework's sorting behavior to compute the total number of records ahead of time, with a single pass over the data.
Enter the Order inversion pattern!

- The order inversion pattern is so named because through proper coordination, we can access the result of a computation in the reducer (for example, an aggregate statistic) before processing the data needed for that computation
- The key insight is to convert sequencing of computations into a sorting problem



Pseudo-code

```
def map(stdin):
  totalWords = 0
  for line in stdin:
    for word in line.split():
    print(word, 1)
    totalWords += 1
  print("**totalWords", totalWords)
```

```
def reduce(stdin):
  currentWord = None
  currentWordCount = 0
  N = 0
  for key, count in stdin:
    if key == "**totalWords":
      N += count
    elif key == currentWord:
      currentWordCount += count
    else:
      if currentWord:
      print(key,currentWordCount/N)
      currentWord = key
      currentWordCount = count
```

Specifically, we'll make an adjustment to the mapper function, that emits the total count of words seen by this mapper. We'll assign a special key to this count by prefixing it with a * character, which will guarantee that when sorting is performed on the keys, this key will bubble up to the top of the list, and as a consequence, will appear before any other keys in the reduce phase. We'll now be able to aggregate all the total counts from each mapper in the reducer before we see any other keys. And voila! We have the total word count ahead of time, with a single pass over the data!

From Relative Frequencies to Naive Bayes

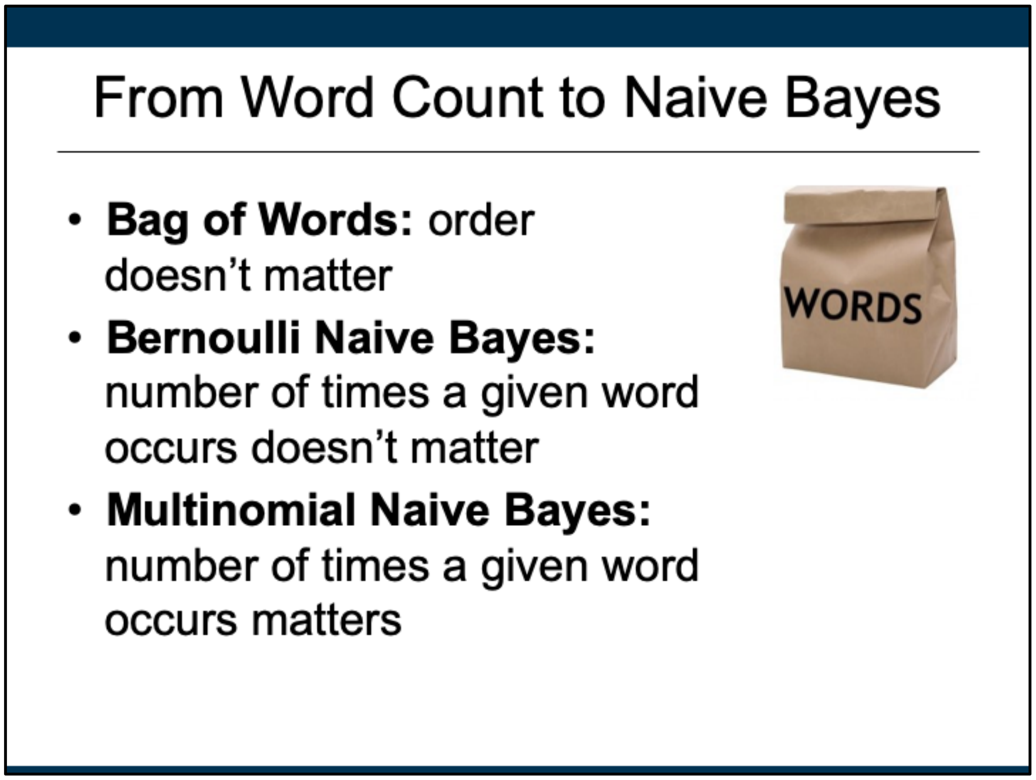
Next, we'll see how we can leverage the **order inversion pattern** to implement a naive Bayes text classification model in parallel.

See you in the next section!

Computing Relative Frequencies

The End

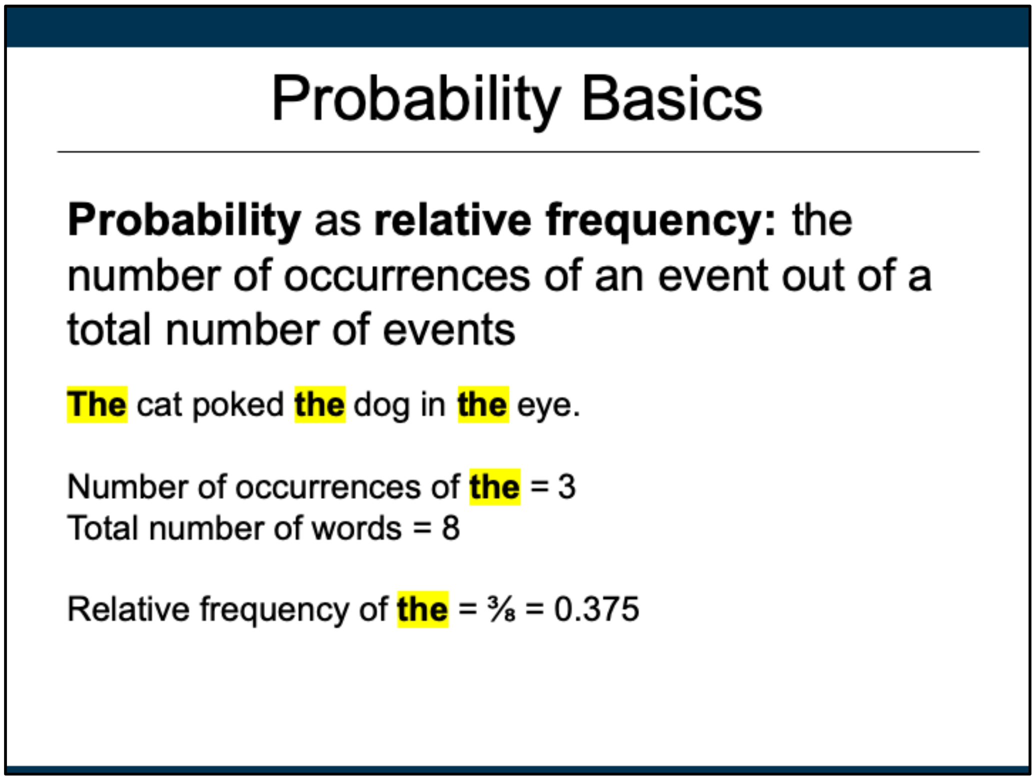
Naive Bayes Theory



From Word Count to Naive Bayes

- **Bag of Words:** order doesn’t matter
- **Bernoulli Naive Bayes:** number of times a given word occurs doesn’t matter
- **Multinomial Naive Bayes:** number of times a given word occurs matters

We’ll now take our word count example, and extend it to a simple classification algorithm, namely Naive Bayes.
You’ve probably heard the term “bag of words” model. Imagine cutting out all of the words from a document and throwing them into a bag in no particular order. In a nutshell, the model is not concerned what order the words are in, only that they occur in the document. Furthermore, the Bernoulli NB model doesn’t even care about how many times a given word appears. In contrast, the Multinomial model takes into consideration the number of times a given word occurs - more on that later.



Probability Basics

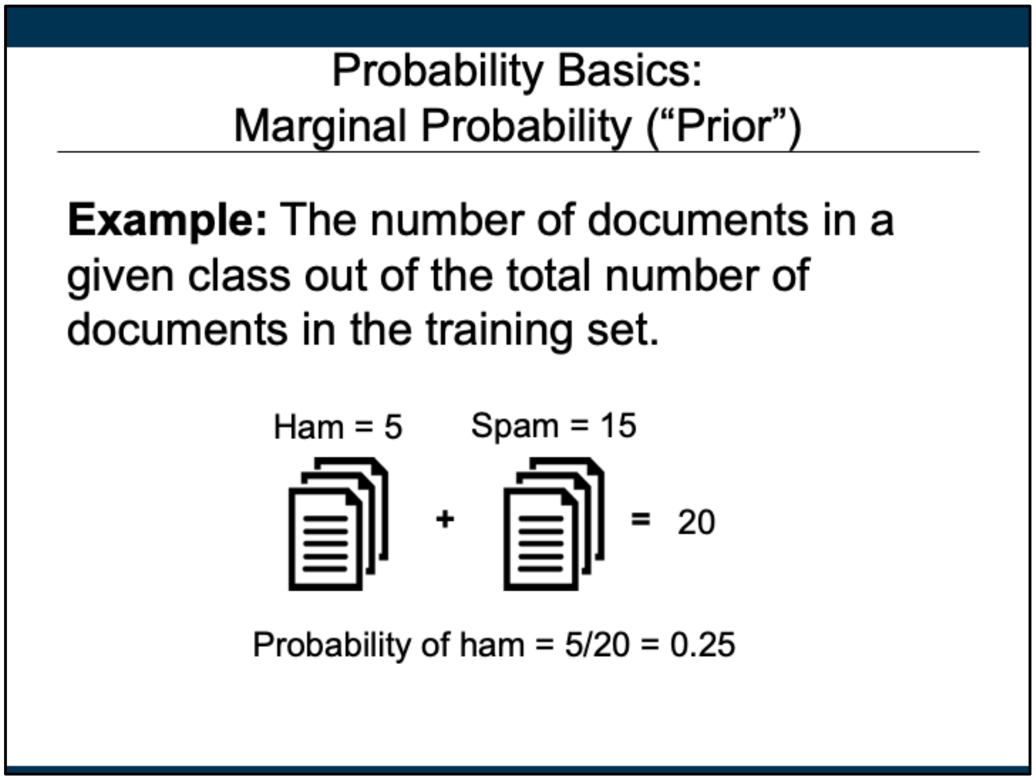
Probability as **relative frequency:** the number of occurrences of an event out of a total number of events

The cat poked **the** dog in **the** eye.

Number of occurrences of **the** = 3
Total number of words = 8

Relative frequency of **the** = ⅜ = 0.375

Before we dive into Naive Bayes, let’s review a little probability theory. We can think of probabilities as nothing more than the relative frequency with which an event occurs. In the case of NB text classification, an event is a word. If the word “the” occurs 3 times in a document of 8 words, then we can say that the relative frequency, or the probability of the word “the” is 3 out of 8, or 0.375



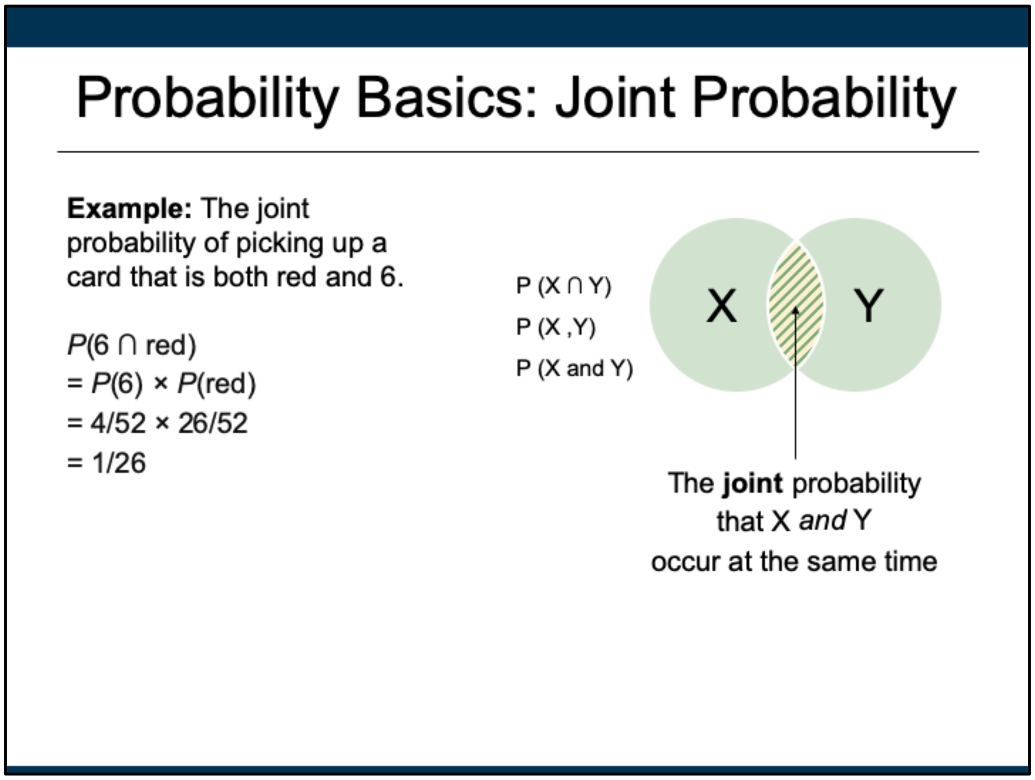
Probability Basics: Marginal Probability (“Prior”)

Example: The number of documents in a given class out of the total number of documents in the training set.

Ham = 5
Spam = 15
+
= 20

Probability of ham = 5/20 = 0.25

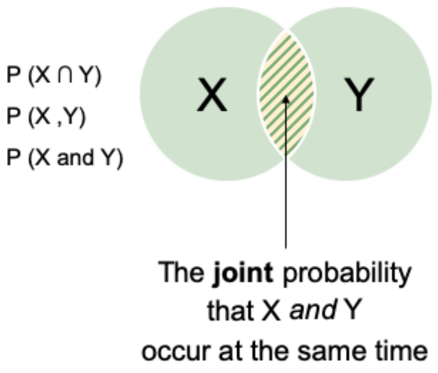
If we extend this to binary document classification, then we can say that the probability of a document belonging to class Ham is just the number of documents in our training set with the label Ham out of a total number of documents. This is called the marginal probability, or in terms of Bayes rule, the “prior”.



Probability Basics: Joint Probability

Example: The joint probability of picking up a card that is both red and 6.

$P(6 \cap \text{red})$
$= P(6) \times P(\text{red})$
$= 4/52 \times 26/52$
$= 1/26$

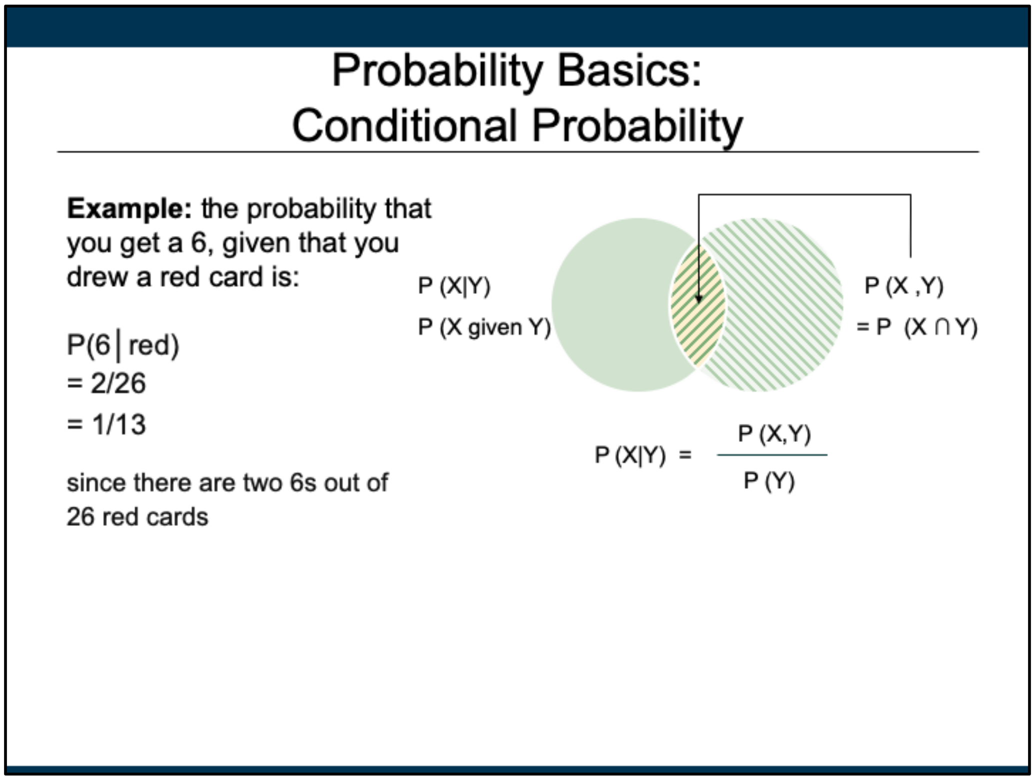

P (X ∩ Y)
P (X ,Y)
P (X and Y)
X
Y
The **joint** probability that X *and* Y occur at the same time

Joint probability is a statistical measure that calculates the likelihood of two events occurring together and at the same point in time. Joint probability is the probability of event Y occurring at the same time that event X occurs.

For example, the probability of picking a card that is both a six and red out of a deck of cards, can be expressed as the joint probability of 6 intersection red, or the probability of 6 times the probability of red. There are 4 sixes in a deck of cards, and 26 reds. Hence 4 divided by 52 times 26 divided by 52, equals 1 out of 26.

There are a couple of different ways to express this in notation, as can be seen in the diagram on the right. These are all equivalent.

Joint probability should not be confused with conditional probability, which is the probability that one event will happen *given that* another event happens.

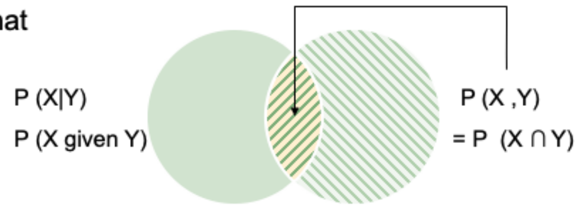


Probability Basics: Conditional Probability

Example: the probability that you get a 6, given that you drew a red card is:

P(6 | red)
= 2/26
= 1/13

since there are two 6s out of 26 red cards

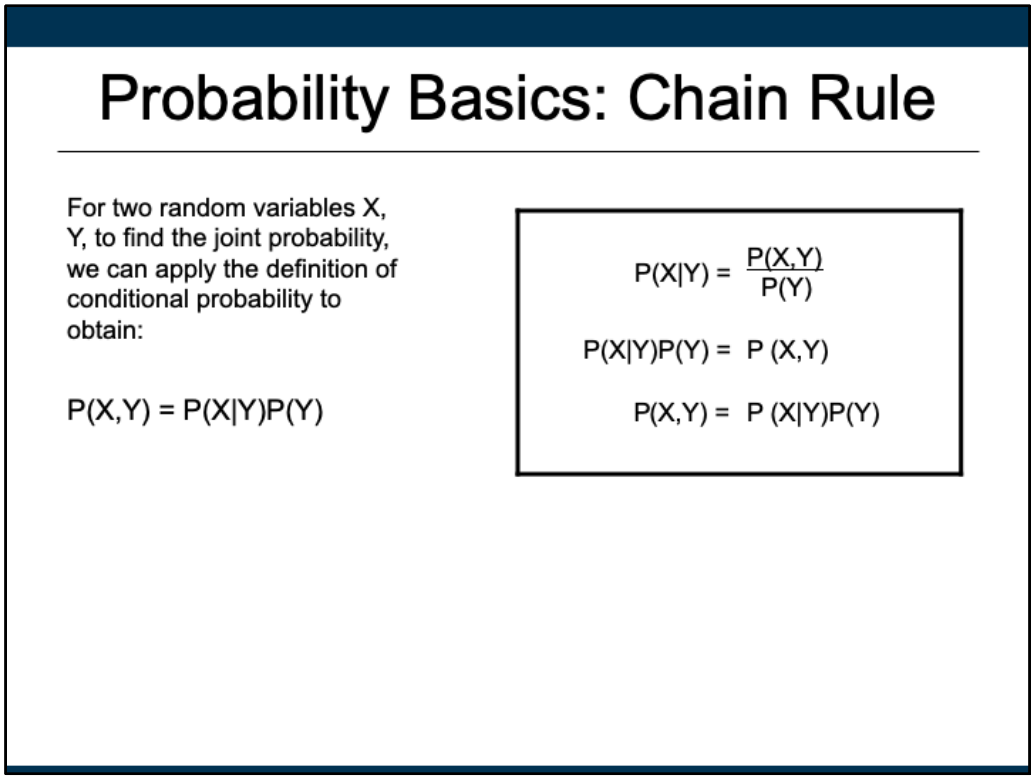


$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

...which is the probability that one event will happen *given that* another event happens.
We can use Joint probability to calculate Conditional probability.

To use our example from the previous slide, the probability of getting a six, given that you drew a red card, is as follows: We know that there are 26 red cards in the deck, and that 2 of those cards are sixes. Hence the probability of a six given red, is 2 divided by 26, which equals 1/13th.

We can see in the diagram on the right, that the conditional probability is just the joint probability divided by the probability of the event we are conditioning on.



Probability Basics: Chain Rule

For two random variables X, Y, to find the joint probability, we can apply the definition of conditional probability to obtain:

$P(X,Y) = P(X|Y)P(Y)$

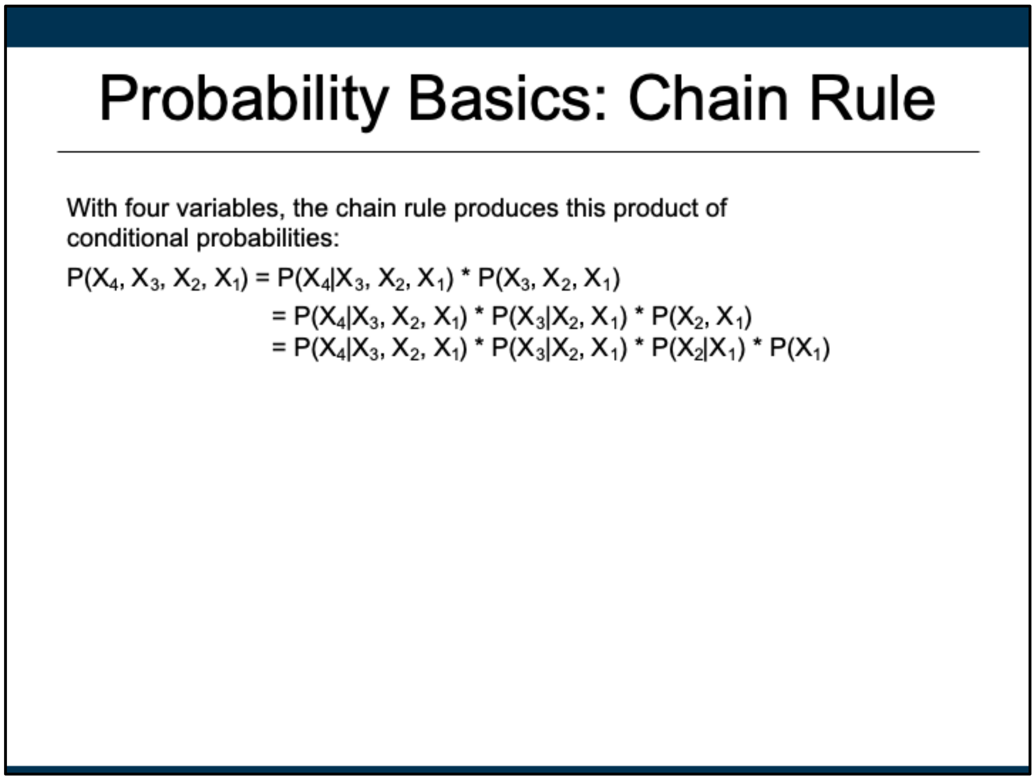
$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

$$P(X|Y)P(Y) = P(X,Y)$$

$$P(X,Y) = P(X|Y)P(Y)$$

In the previous slide, we used Joint probability to calculate conditional probability. Conversely, we can perform some basic algebra to find the joint probability distribution. In the case of two random variables, this gives us the probability of X and Y equals the probability of X given Y times the probability of Y

This brings us to the chain rule.

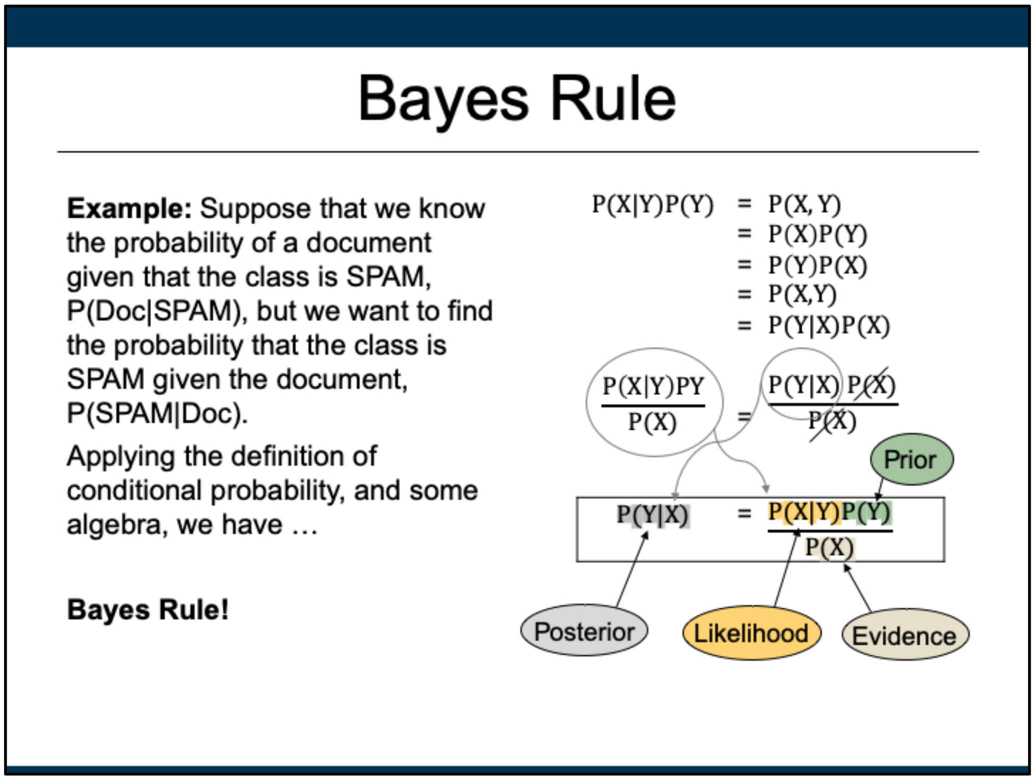


Probability Basics: Chain Rule

With four variables, the chain rule produces this product of conditional probabilities:

$$\begin{aligned}P(X_4, X_3, X_2, X_1) &= P(X_4|X_3, X_2, X_1) * P(X_3, X_2, X_1)\\ &= P(X_4|X_3, X_2, X_1) * P(X_3|X_2, X_1) * P(X_2, X_1)\\ &= P(X_4|X_3, X_2, X_1) * P(X_3|X_2, X_1) * P(X_2|X_1) * P(X_1)\end{aligned}$$

In probability theory, the chain rule (also called the general product rule[1]) permits the calculation of any member of the joint distribution of a set of random variables using only conditional probabilities.



Bayes Rule

Example: Suppose that we know the probability of a document given that the class is SPAM, P(Doc|SPAM), but we want to find the probability that the class is SPAM given the document, P(SPAM|Doc).

Applying the definition of conditional probability, and some algebra, we have …

Bayes Rule!

$$
\begin{aligned}
P(X|Y)P(Y) &= P(X,Y) \\
&= P(X)P(Y) \\
&= P(Y)P(X) \\
&= P(X,Y) \\
&= P(Y|X)P(X)
\end{aligned}
$$

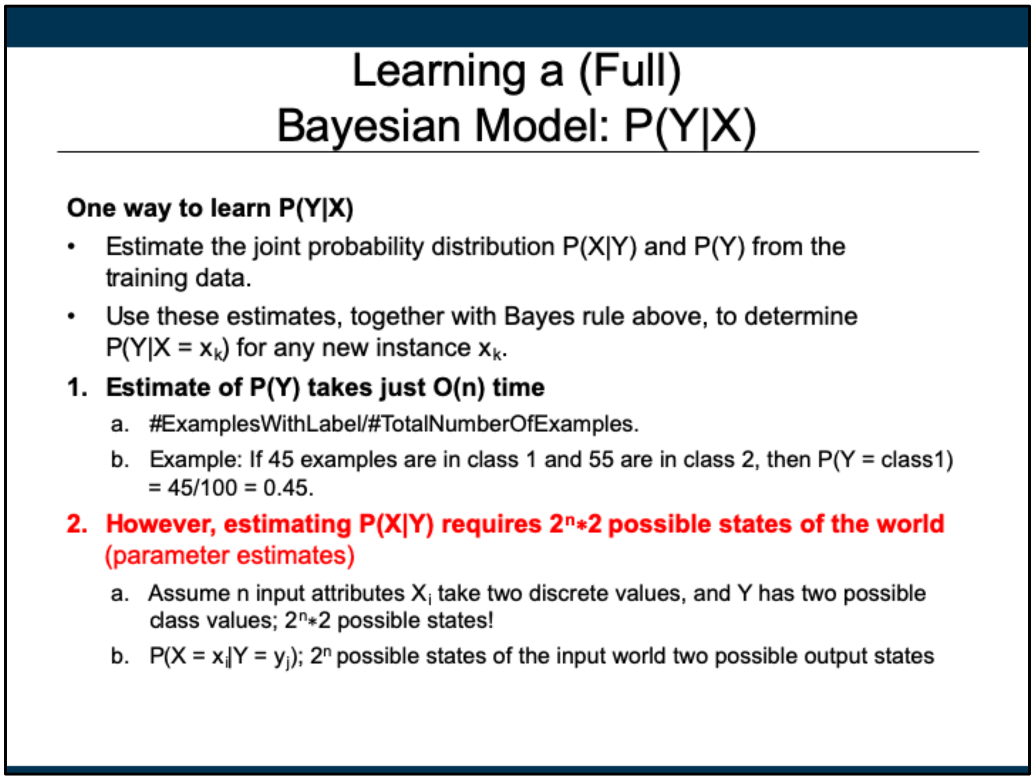
$$\frac{P(X|Y)PY}{P(X)} = \frac{P(Y|X)\,\cancel{P(X)}}{\cancel{P(X)}}$$

$$\boxed{P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}}$$

Now we are ready to state one of the most useful results in conditional probability: Bayes' rule. Suppose that we know P(X|Y), but we are interested in the probability P(Y|X).

We can apply the previously stated rules and, again, some basic algebra, to arrive at the formula for Bayes Rule.

Chain rule
Joint Probability
Associative property
Joint probability
Chain rule
Divide both sides by P(x)
Flip sides for the familiar representation of the Bayes rule



Learning a (Full) Bayesian Model: P(Y|X)

One way to learn P(Y|X)

- Estimate the joint probability distribution P(X|Y) and P(Y) from the training data.
- Use these estimates, together with Bayes rule above, to determine $P(Y|X = x_k)$ for any new instance x_k.

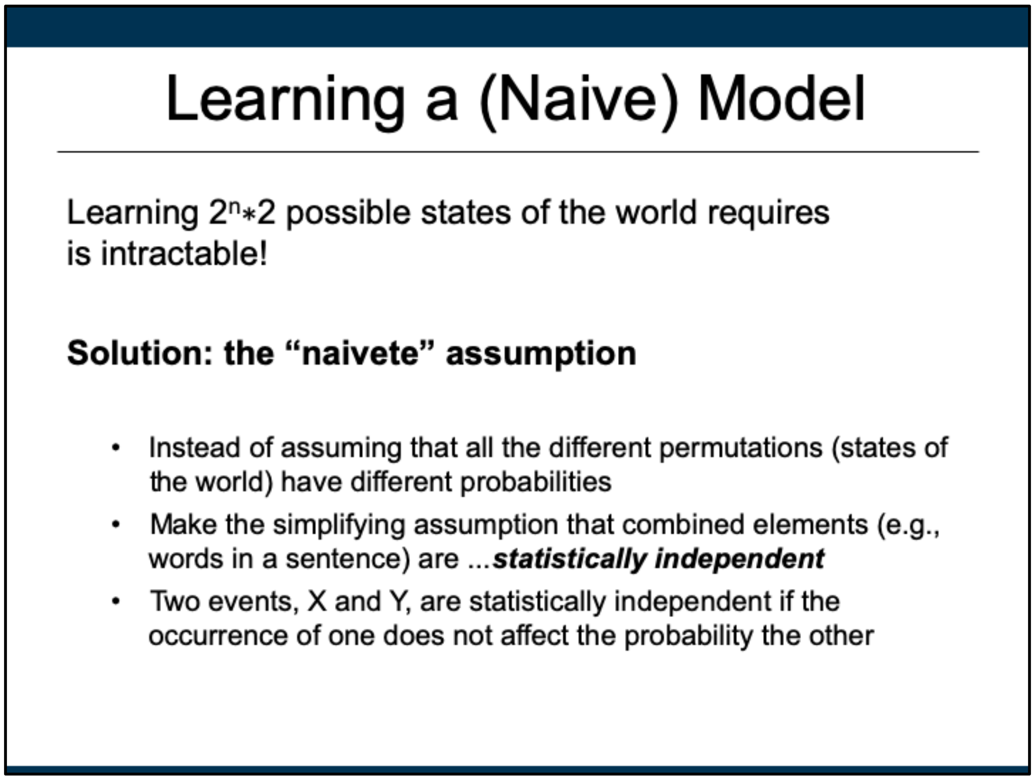
1. **Estimate of P(Y) takes just O(n) time**
   a. #ExamplesWithLabel/#TotalNumberOfExamples.
   b. Example: If 45 examples are in class 1 and 55 are in class 2, then P(Y = class1) = 45/100 = 0.45.
2. **However, estimating P(X|Y) requires 2^n*2 possible states of the world** (parameter estimates)
   a. Assume n input attributes X_i take two discrete values, and Y has two possible class values; 2^n*2 possible states!
   b. $P(X = x_i|Y = y_j)$; 2^n possible states of the input world two possible output states

But we're not out of the woods yet!

One way to learn the posterior probability of Y|X, is to estimate the joint probability distribution of X given Y as well as the probability of Y from the training data.

P(Y) is easy and takes just O(n) time. It is simply the relative frequency of Y in the training data.

However, estimating P(X|Y) requires calculating an exponential number of parameter estimates. In other words, given a corpus with vocabulary size n=100, and labels Y in the set of Ham or Spam, then we have 2^100 * 2 possible states of the world. It's not difficult to imagine a vocabulary size orders of magnitude larger than 100.
P265 IIR



Learning a (Naive) Model

Learning 2^n*2 possible states of the world requires is intractable!

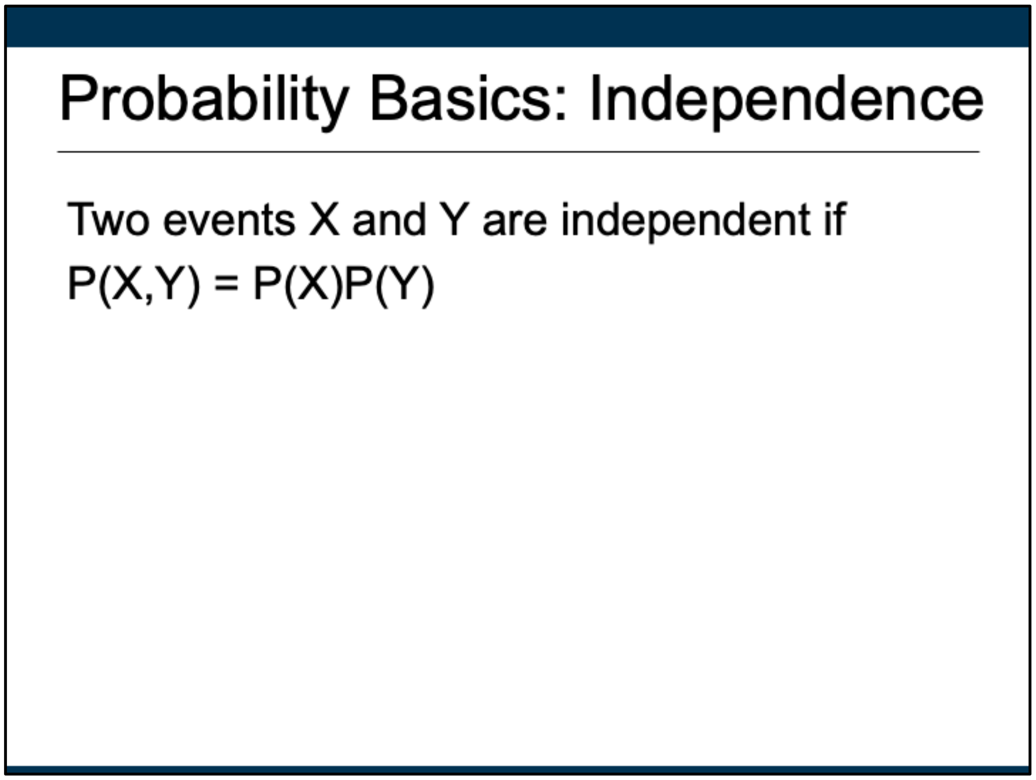
Solution: the "naivete" assumption

- Instead of assuming that all the different permutations (states of the world) have different probabilities
- Make the simplifying assumption that combined elements (e.g., words in a sentence) are ...***statistically independent***
- Two events, X and Y, are statistically independent if the occurrence of one does not affect the probability the other

It's not difficult to imagine a vocabulary size orders of magnitude larger than 100. Clearly, learning 2^n*2 possible states of the world is intractable!

So… instead of assuming that all of the different permutations (states of the world) have different probabilities, we'll make the simplifying assumption that combined elements (e.g. words in a sentence) are... STATISTICALLY INDEPENDENT.

Two events X and Y are statistically independent, if the occurrence of one does not affect the probability the other



Probability Basics: Independence

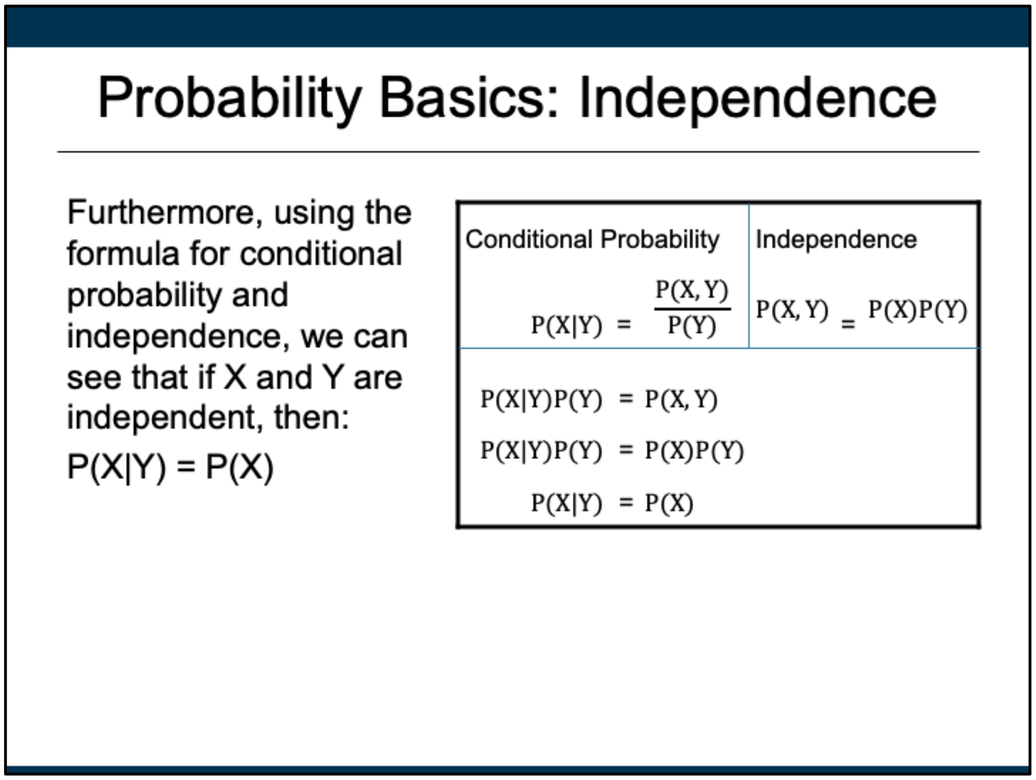
Two events X and Y are independent if
$P(X,Y) = P(X)P(Y)$

In other words, two events are independent if one does not convey any information about the other.

Let us now provide a formal definition of independence.

We can say "independence means we can multiply the probabilities of events to obtain the probability of their intersection", or equivalently, "independence means that conditional probability of one event given another is the same as the original (prior) probability". Let's take a look at the math (next slide).

https://www.probabilitycourse.com/chapter1/1_4_1_independence.php



Probability Basics: Independence

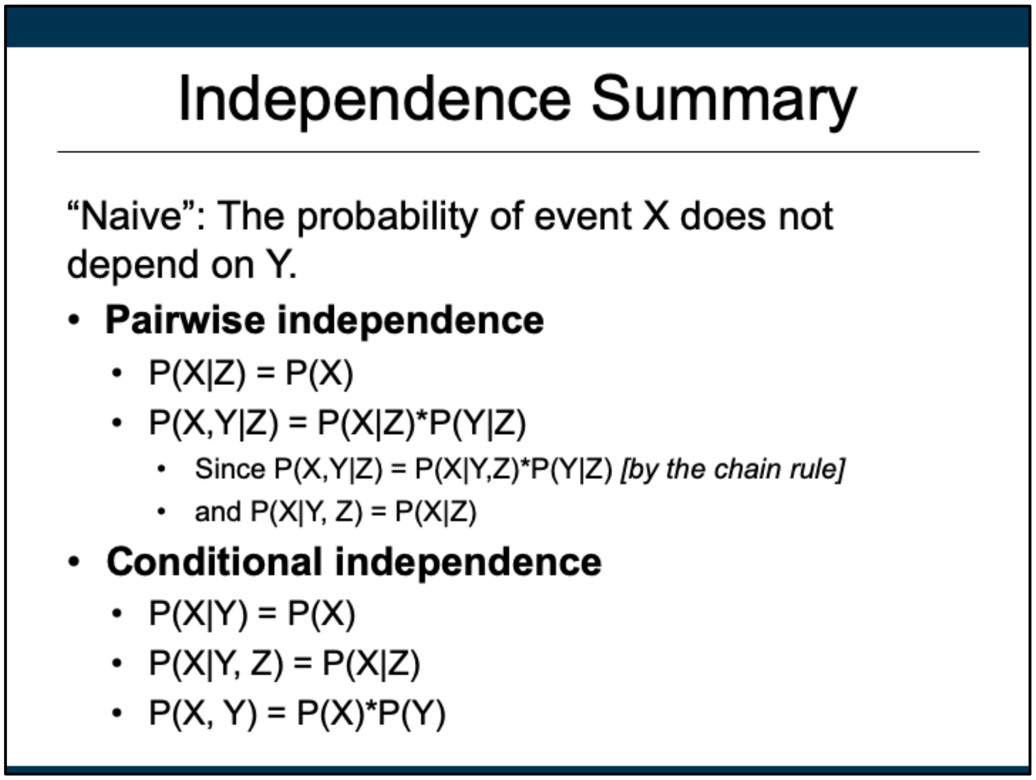
Furthermore, using the formula for conditional probability and independence, we can see that if X and Y are independent, then:

P(X|Y) = P(X)

Conditional Probability	Independence
$P(X\|Y) = \frac{P(X,Y)}{P(Y)}$	$P(X,Y) = P(X)P(Y)$

$$P(X|Y)P(Y) = P(X,Y)$$
$$P(X|Y)P(Y) = P(X)P(Y)$$
$$P(X|Y) = P(X)$$

Using the formula for conditional probability and independence, we can see that if X and Y are independent then:
P(X|Y) is simply P(X)

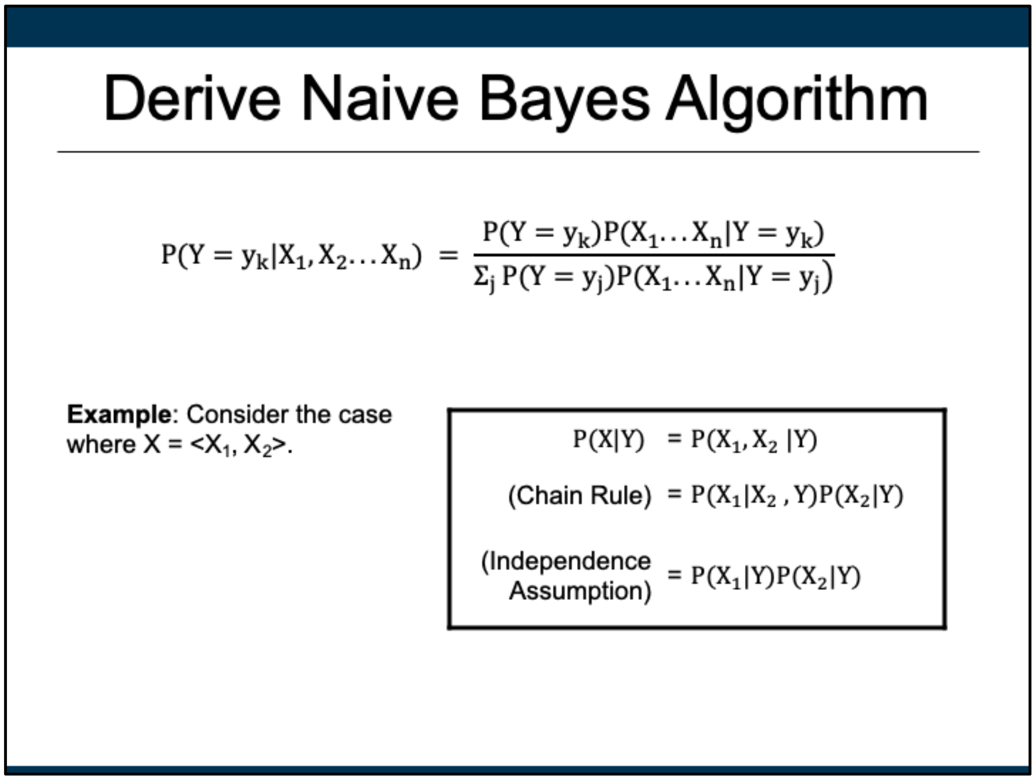


Independence Summary

"Naive": The probability of event X does not depend on Y.

- **Pairwise independence**
  - P(X|Z) = P(X)
  - P(X,Y|Z) = P(X|Z)*P(Y|Z)
    - Since P(X,Y|Z) = P(X|Y,Z)*P(Y|Z) *[by the chain rule]*
    - and P(X|Y, Z) = P(X|Z)
- **Conditional independence**
  - P(X|Y) = P(X)
  - P(X|Y, Z) = P(X|Z)
  - P(X, Y) = P(X)*P(Y)

In summary, we have the following definitions for pairwise independence, as well as conditional independence, which can be further broken down in multiple variables using the chain rule.



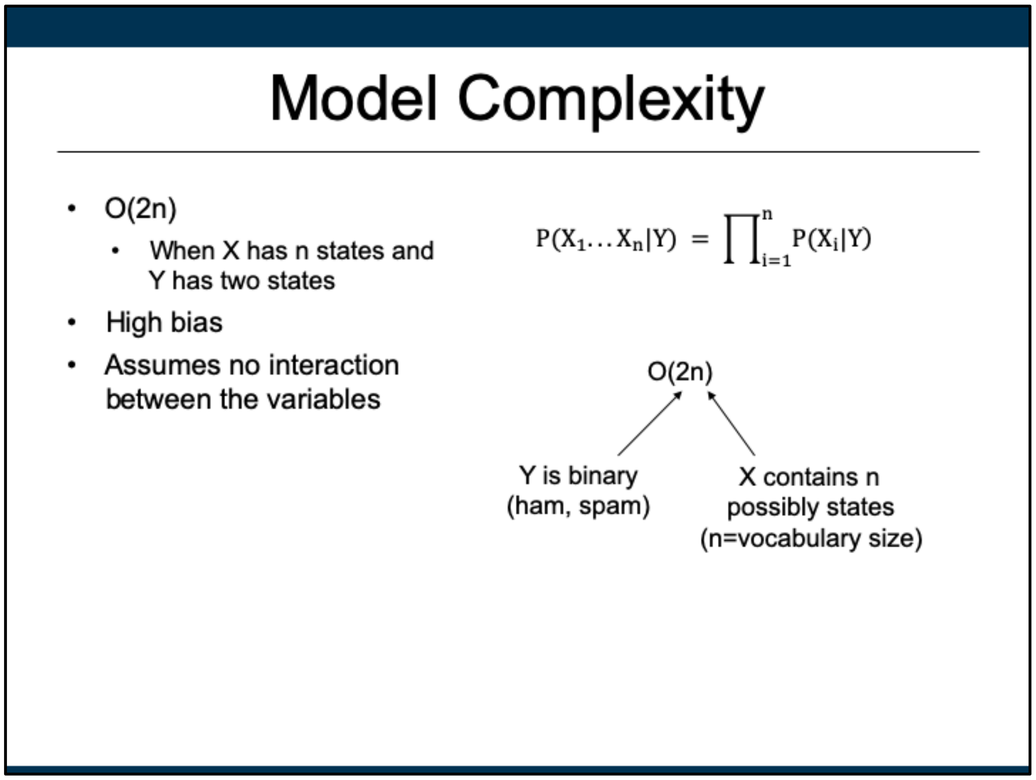
Derive Naive Bayes Algorithm

$$P(Y = y_k|X_1, X_2 \ldots X_n) = \frac{P(Y = y_k)P(X_1 \ldots X_n|Y = y_k)}{\Sigma_j P(Y = y_j)P(X_1 \ldots X_n|Y = y_j)}$$

Example: Consider the case where X = <X_1, X_2>.

$P(X\|Y)$	$= P(X_1, X_2 \|Y)$
(Chain Rule)	$= P(X_1\|X_2, Y)P(X_2\|Y)$
(Independence Assumption)	$= P(X_1\|Y)P(X_2\|Y)$

The Naive Bayes algorithm is a classification algorithm based on Bayes rule, that assumes the attributes X1...Xn are all conditionally independent of one another, given Y. The value of this assumption is that it dramatically simplifies the representation of P(X|Y), and the problem of estimating it from the the training data.



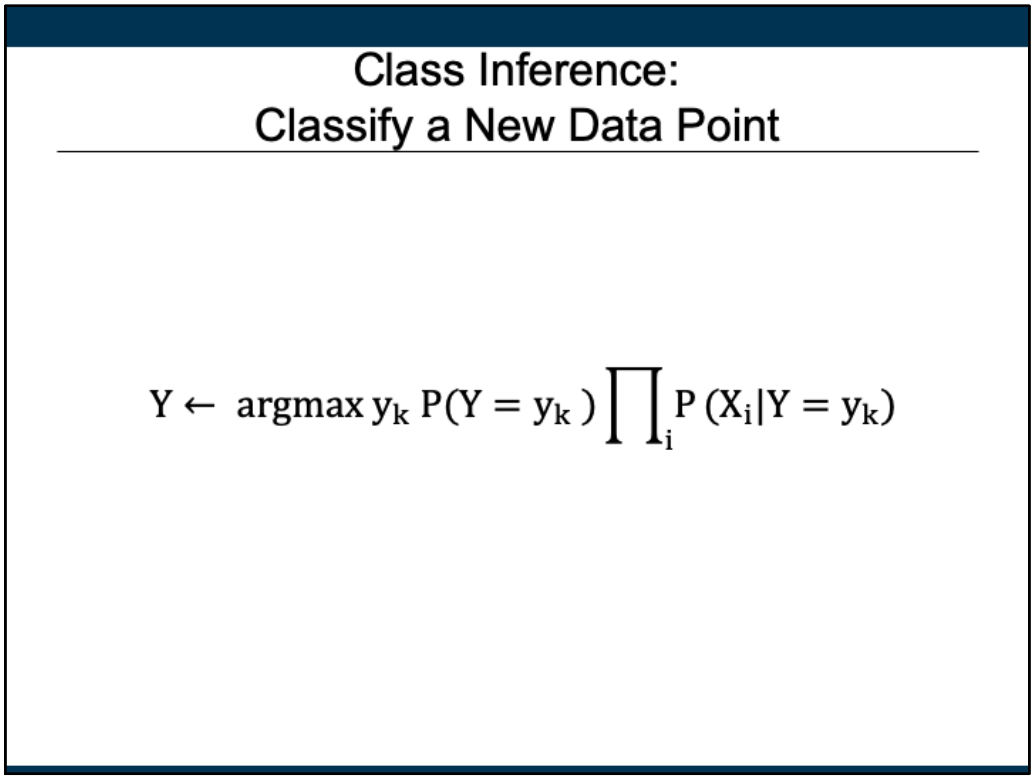
Model Complexity

- O(2n)
  - When X has n states and Y has two states
- High bias
- Assumes no interaction between the variables

$$P(X_1 \ldots X_n|Y) = \prod_{i=1}^{n} P(X_i|Y)$$

$$\begin{tikzcd} & O(2n) & \\ \text{Y is binary (ham, spam)} \arrow[ur] & & \text{X contains n possibly states (n=vocabulary size)} \arrow[ul] \end{tikzcd}$$

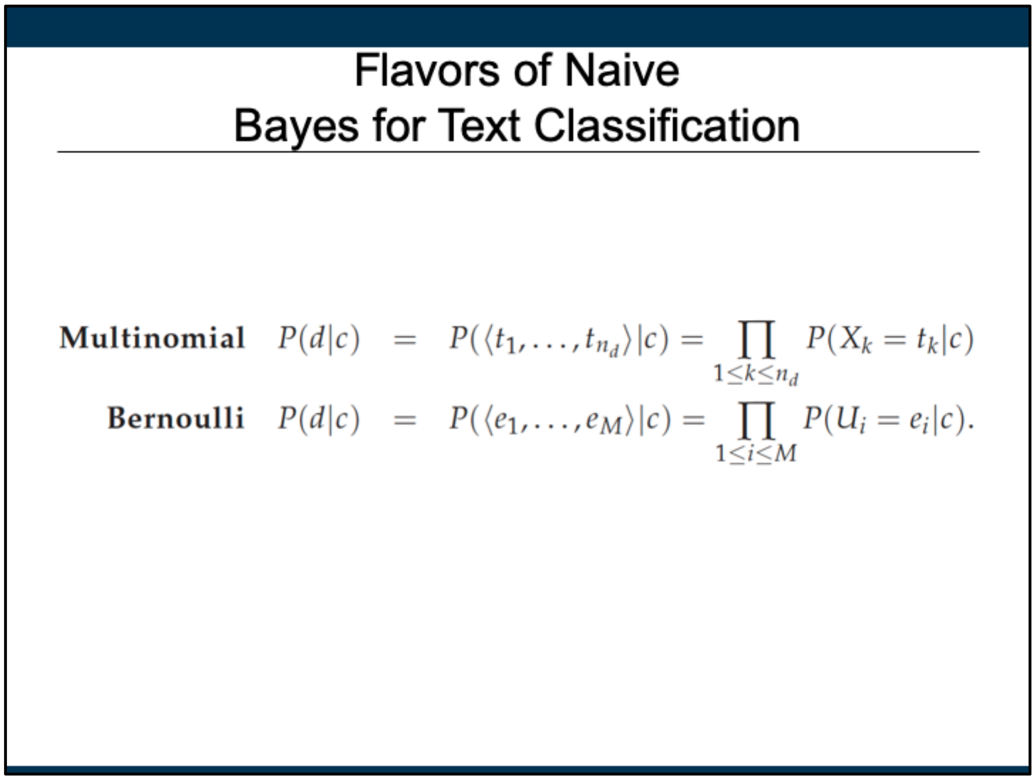
Conditional independence dramatically reduces model complexity from 2*2^n parameters to just 2n. How can NB be a good text classifier when its model of natural language is so oversimplified? The answer is that even though the probability estimates of NB are of low quality, its classification decisions are surprisingly good. It does not matter how accurate the estimates are. Despite the bad estimates,NB estimates a higher probability for the correct class, and therefore the classification is correct. Correct estimation implies accurate prediction, but accurate prediction does not imply correct estimation. NB classifiers estimate badly, but often classify well.



Class Inference: Classify a New Data Point

$$Y \leftarrow \text{argmax}\, y_k\ P(Y = y_k) \prod_i P(X_i | Y = y_k)$$

To classify a new data point, we simply calculate the probability of the class P(Y=yk) for all classes, and select the argmax, in other words, the class class with the highest probability.



Flavors of Naive Bayes for Text Classification

$$\begin{aligned} \textbf{Multinomial} \quad P(d|c) &= P(\langle t_1,\ldots,t_{n_d}\rangle|c) = \prod_{1\leq k\leq n_d} P(X_k = t_k|c) \\ \textbf{Bernoulli} \quad P(d|c) &= P(\langle e_1,\ldots,e_M\rangle|c) = \prod_{1\leq i\leq M} P(U_i = e_i|c). \end{aligned}$$

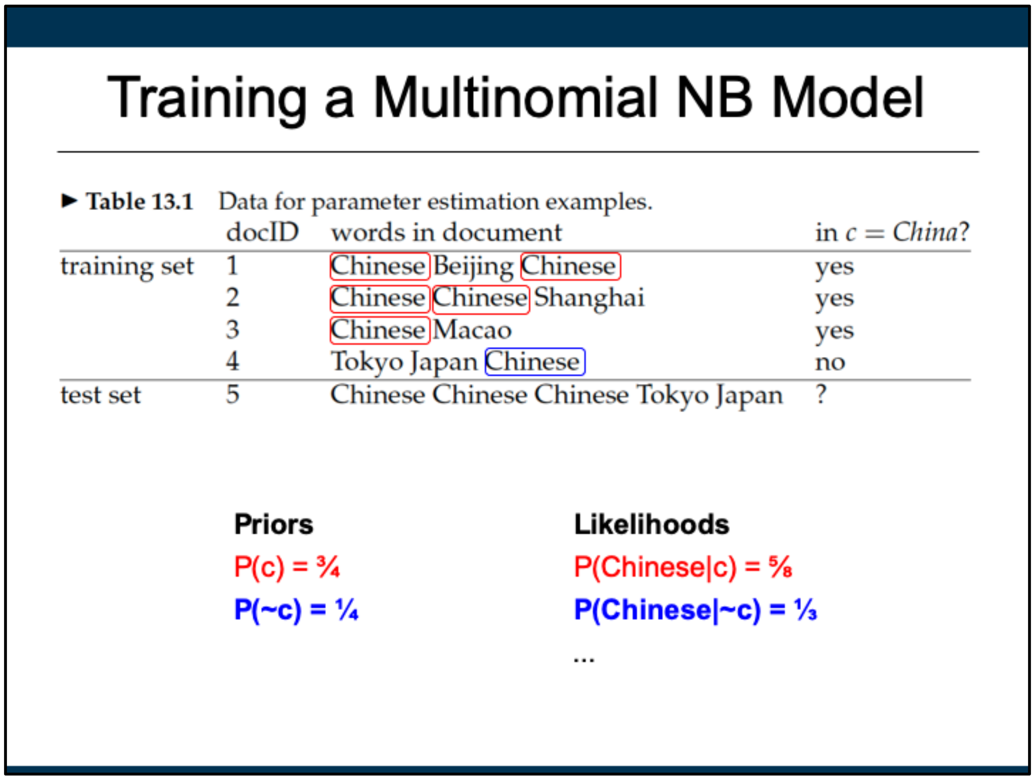
For the rest of this section we'll be focusing on the Multinomial Naive Bayes model. However, it's worth noting that it's not the only flavor out there. The three main flavors of NB are Multinomial, Bernoulli, and Gaussian

Both Multinomial and Bernoulli are well suited to text classification, where each feature corresponds to an observation for a particular word. On the other hand, when dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a normal (or Gaussian) distribution. In this setting a Gaussian NB model is the appropriate approach.

The main difference between Multinomial and Bernoulli NB, is the fact that MNB takes into considerations the number of times an event occurs, whereas Bernoulli uses binary term occurrence features rather than term frequencies.

For an in-depth discussion of the differences between these models read Chapter 13 of the Introduction to Information Retrieval by Manning, Raghavan, and Schütze which is also listed in your reading assignments for this week.

~~Talk about differences in interpretation as well as robustness.~~



Training a Multinomial NB Model

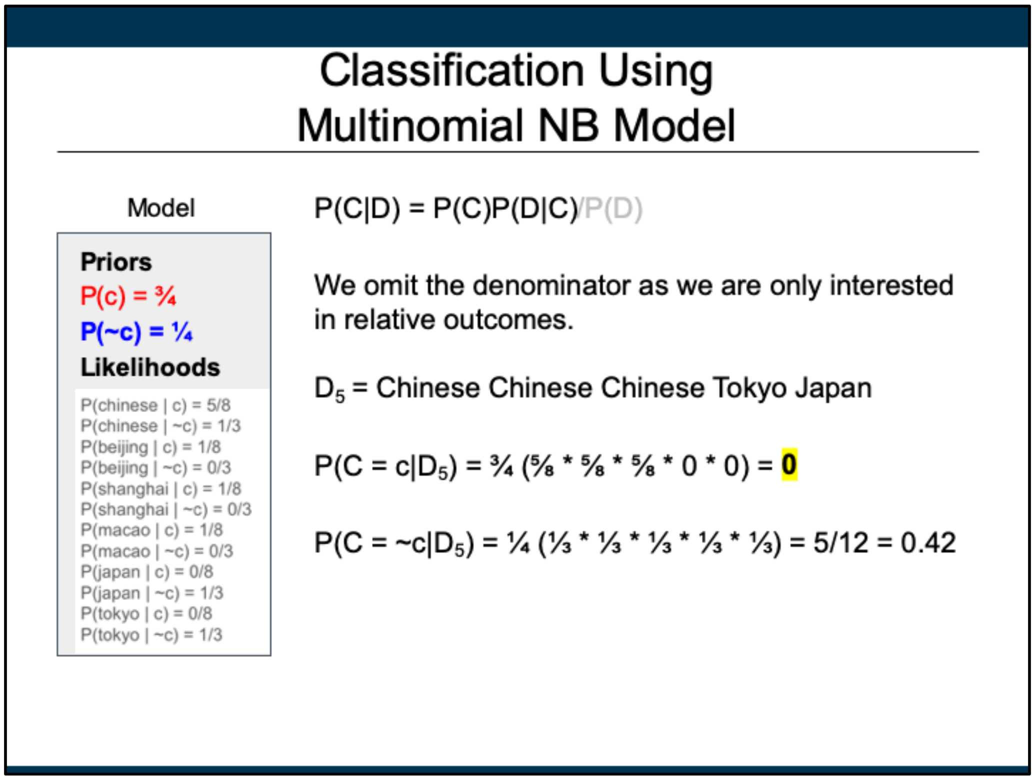
► **Table 13.1** Data for parameter estimation examples.

	docID	words in document	in $c = China$?
training set	1	Chinese Beijing Chinese	yes
	2	Chinese Chinese Shanghai	yes
	3	Chinese Macao	yes
	4	Tokyo Japan Chinese	no
test set	5	Chinese Chinese Chinese Tokyo Japan	?

Priors	**Likelihoods**
P(c) = ¾	P(Chinese|c) = ⅝
P(~c) = ¼	**P(Chinese|~c) = ⅓**
	…

Let’s look at an example:
To train a MNB model, we’ll first have to calculate the prior probabilities for each class. This is simply the ratio of the number of documents in each class to the total number of documents.
For each word in the corpus, we’ll also need to calculate the likelihood of the word for each class. The probability of the word “Chinese” given class ‘china’ is ⅝ - since the word “Chinese” occurs 5 times out of a total of 8 words in class china. We do this for all words and all classes in the training data. This constitutes our model.

(Will need stylus to walk through calculations)



Classification Using Multinomial NB Model

Model

Priors

$P(c) = 3/4$

$P(\sim c) = 1/4$

Likelihoods

P(chinese | c) = 5/8
P(chinese | ~c) = 1/3
P(beijing | c) = 1/8
P(beijing | ~c) = 0/3
P(shanghai | c) = 1/8
P(shanghai | ~c) = 0/3
P(macao | c) = 1/8
P(macao | ~c) = 0/3
P(japan | c) = 0/8
P(japan | ~c) = 1/3
P(tokyo | c) = 0/8
P(tokyo | ~c) = 1/3

$P(C|D) = P(C)P(D|C)/P(D)$

We omit the denominator as we are only interested in relative outcomes.

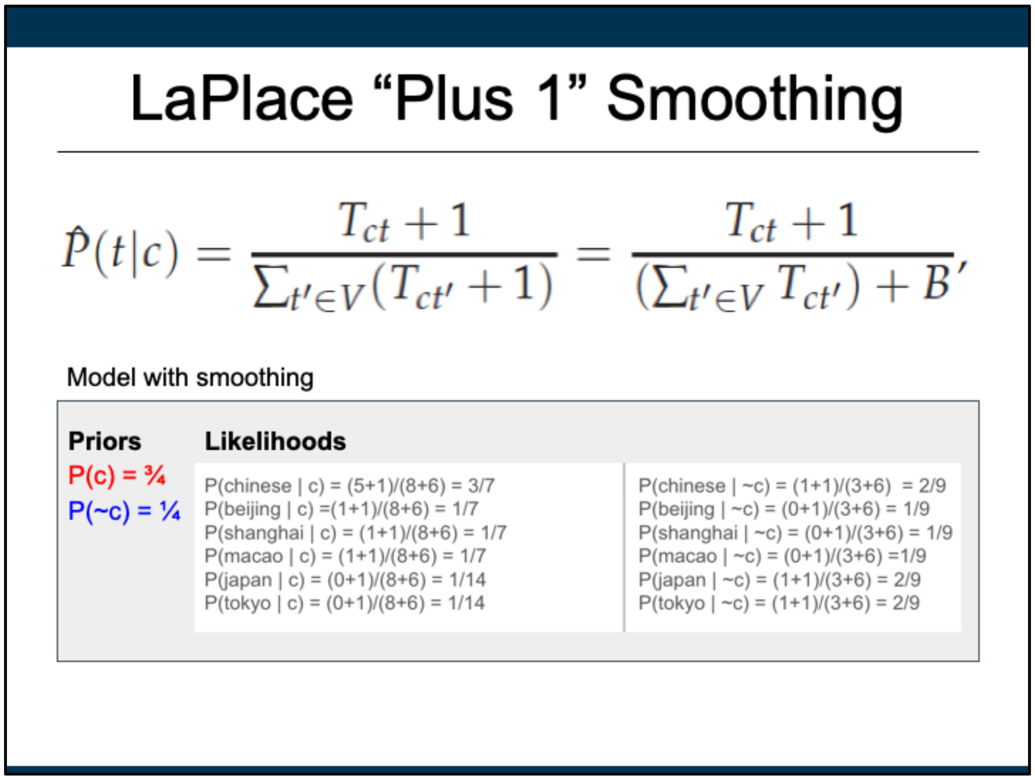
D_5 = Chinese Chinese Chinese Tokyo Japan

$P(C = c|D_5) = \frac{3}{4} (\frac{5}{8} * \frac{5}{8} * \frac{5}{8} * 0 * 0) = \mathbf{0}$

$P(C = \sim c|D_5) = \frac{1}{4} (\frac{1}{3} * \frac{1}{3} * \frac{1}{3} * \frac{1}{3} * \frac{1}{3}) = 5/12 = 0.42$

We'll now predict the class of document 5 using the model as seen on the left. It seems that our prediction is incorrect, given the intuition that the word "Chinese" occurs three times in the document, a more reasonable prediction would be china class.

So what happened here? Notice that the words Tokyo and Japan never occur in the china class in the training data. When the probability of a term = 0 for some class, then the whole class probability gets set to 0. This is problematic because the algorithm is not able to take into consideration any of the remaining term probabilities.



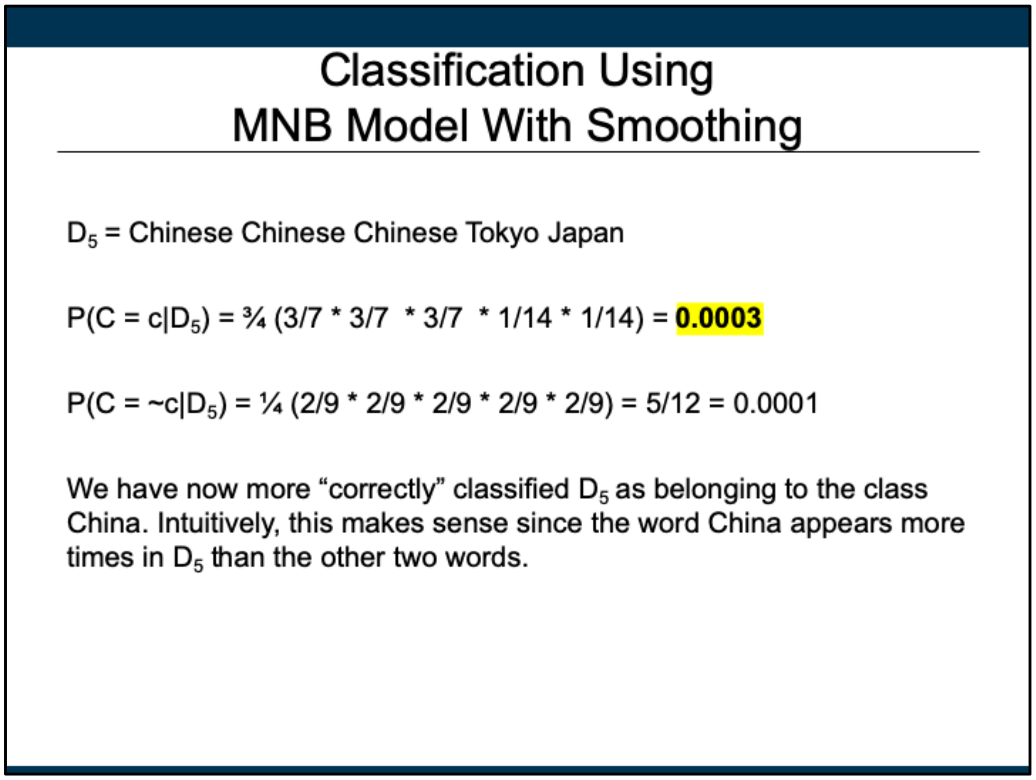
LaPlace “Plus 1” Smoothing

$$\hat{P}(t|c) = \frac{T_{ct}+1}{\sum_{t'\in V}(T_{ct'}+1)} = \frac{T_{ct}+1}{(\sum_{t'\in V}T_{ct'})+B'},$$

Model with smoothing

Priors	**Likelihoods**	
P(c) = ¾	P(chinese | c) = (5+1)/(8+6) = 3/7	P(chinese | ~c) = (1+1)/(3+6) = 2/9
P(~c) = ¼	P(beijing | c) =(1+1)/(8+6) = 1/7	P(beijing | ~c) = (0+1)/(3+6) = 1/9
	P(shanghai | c) = (1+1)/(8+6) = 1/7	P(shanghai | ~c) = (0+1)/(3+6) = 1/9
	P(macao | c) = (1+1)/(8+6) = 1/7	P(macao | ~c) = (0+1)/(3+6) =1/9
	P(japan | c) = (0+1)/(8+6) = 1/14	P(japan | ~c) = (1+1)/(3+6) = 2/9
	P(tokyo | c) = (0+1)/(8+6) = 1/14	P(tokyo | ~c) = (1+1)/(3+6) = 2/9

To address this problem, and eliminate zeros, we can use Laplace “Add-1” smoothing, which simply adds a 1 to each count (cf. Section 11.3.2):
We can think of smoothing as a type of regularization.



Classification Using MNB Model With Smoothing

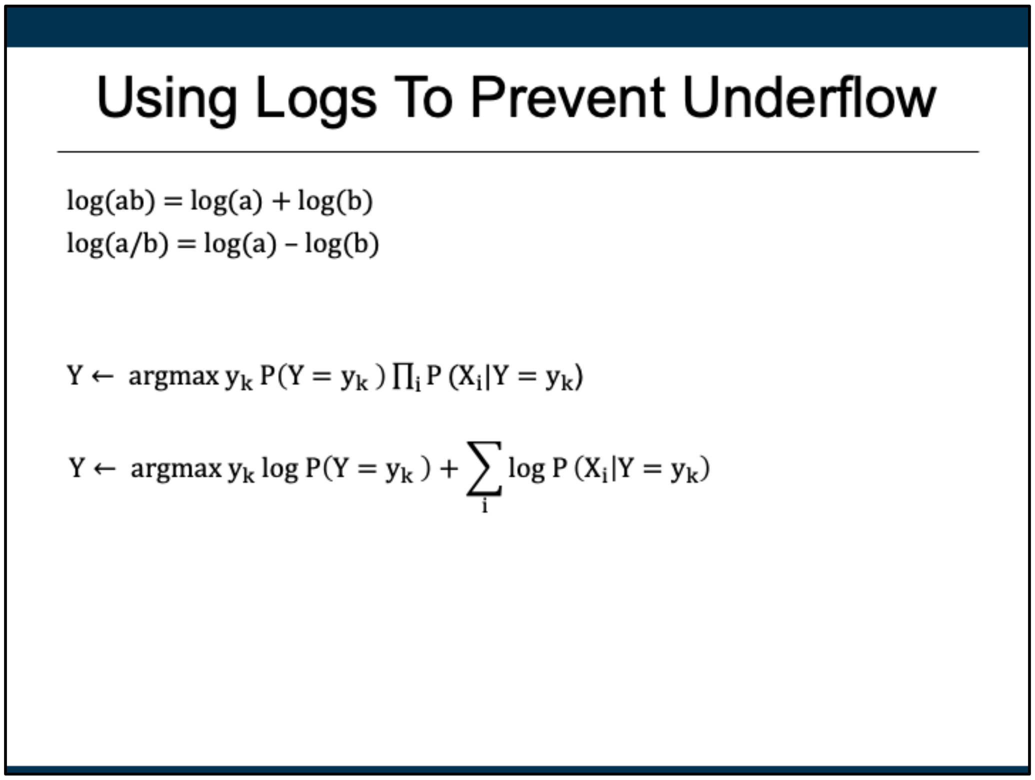
D_5 = Chinese Chinese Chinese Tokyo Japan

P(C = c|D_5) = ¾ (3/7 * 3/7 * 3/7 * 1/14 * 1/14) = **0.0003**

P(C = ~c|D_5) = ¼ (2/9 * 2/9 * 2/9 * 2/9 * 2/9) = 5/12 = 0.0001

We have now more “correctly” classified D_5 as belonging to the class China. Intuitively, this makes sense since the word China appears more times in D_5 than the other two words.

That looks better! The probability that document 5 belongs to class china is just that little bit higher than not-china.



Using Logs To Prevent Underflow

$\log(ab) = \log(a) + \log(b)$

$\log(a/b) = \log(a) - \log(b)$

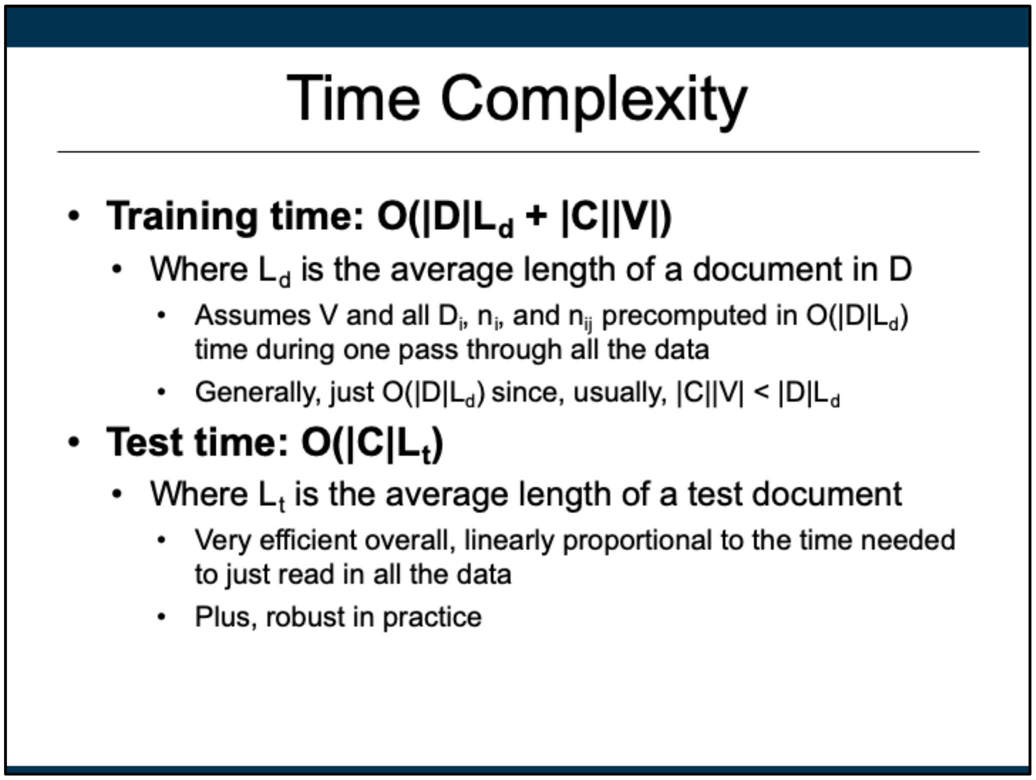
$$Y \leftarrow \operatorname{argmax}\, y_k\, P(Y = y_k)\, \prod_i P(X_i | Y = y_k)$$

$$Y \leftarrow \operatorname{argmax}\, y_k \log P(Y = y_k) + \sum_i \log P(X_i | Y = y_k)$$

In addition to the theoretical considerations such as plus one smoothing, we also need to concern ourselves with computational limitations.

Multiplying lots of probabilities, which are between 0 and 1, can result in floating-point underflow. In other words, the probabilities would become so small that they would be impossible to represent, and as such, and even despite our best efforts, we could end up with 0 probabilities.

Since log(xy) = log(x) + log(y), it is better to perform all computations by summing logs of probabilities rather than multiplying probabilities.
The class with highest final un-normalized log probability score is still the most probable.
Note that model is now just max of sum of weights…


Time Complexity

- **Training time: $O(|D|L_d + |C||V|)$**
  - Where L_d is the average length of a document in D
    - Assumes V and all D_i, n_i, and n_{ij} precomputed in $O(|D|L_d)$ time during one pass through all the data
    - Generally, just $O(|D|L_d)$ since, usually, $|C||V| < |D|L_d$
- **Test time: $O(|C|L_t)$**
  - Where L_t is the average length of a test document
    - Very efficient overall, linearly proportional to the time needed to just read in all the data
    - Plus, robust in practice

P261 IIR ch13.

The complexity of computing the parameters is O(|C||V|) because the set of parameters consists of |C||V| conditional probabilities and |C| priors. Where |V| is the size of the vocabulary.

The preprocessing necessary for computing the parameters (extracting the vocabulary, counting terms, etc.) can be done in one pass through the training data. The time complexity of this
component is therefore O(|D|L_ave), where |D| is the number of documents and L_ave is the average length of a document

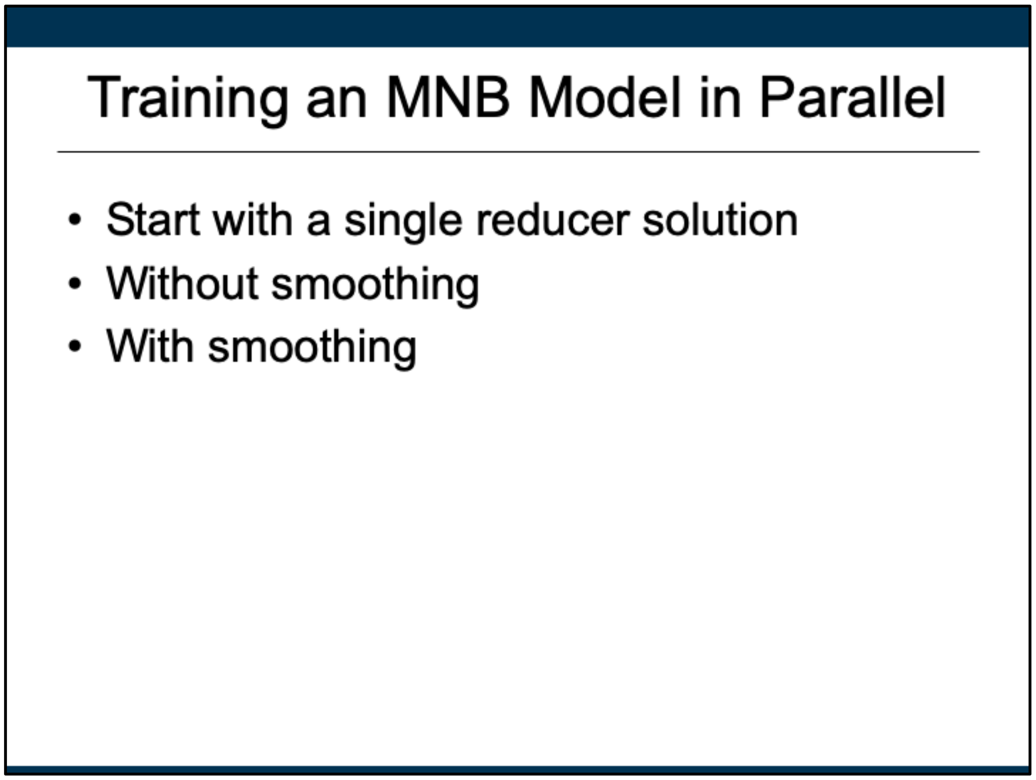
More Examples

Naive Bayes examples

Naive Bayes Theory

The End

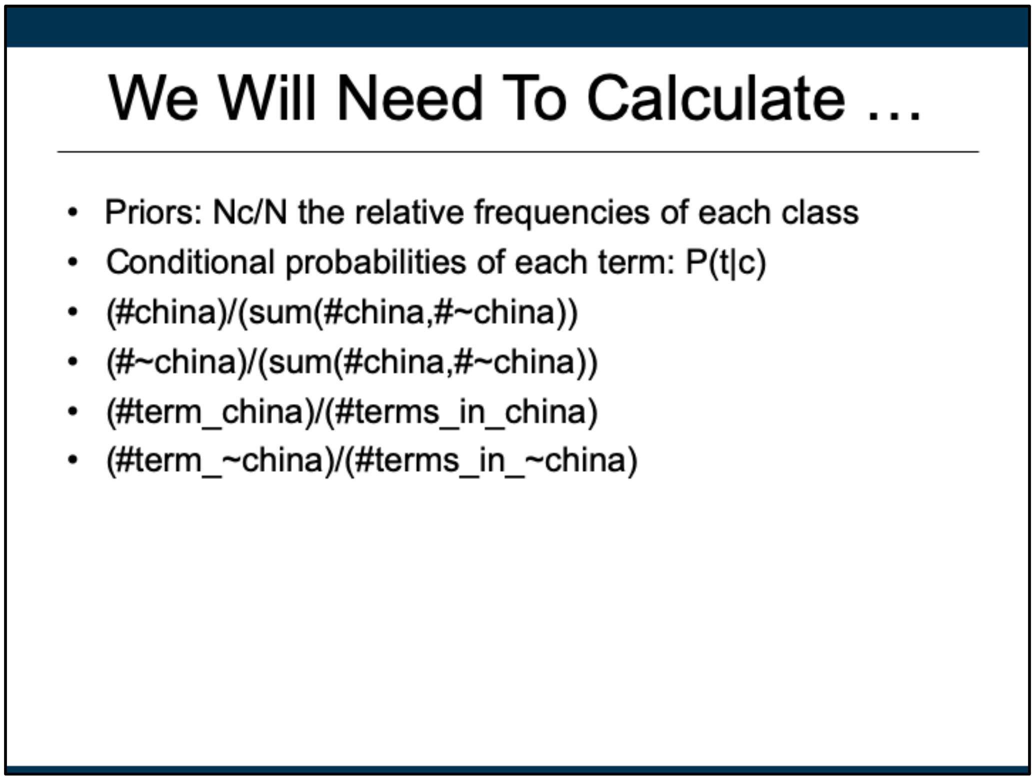
Naive Bayes at Scale



Training an MNB Model in Parallel

- Start with a single reducer solution
- Without smoothing
- With smoothing

Now that we know how to compute relative frequencies in Map Reduce, we will leverage that knowledge to train a NB model in parallel, as well as classify new examples.
We'll need to build a model file from the data, so we can use it to classify new examples.
In this section we'll again focus on the Multinomial model using a single reducer. First we'll build an unsmoothed model and then we'll add LaPlace "add 1" smoothing.

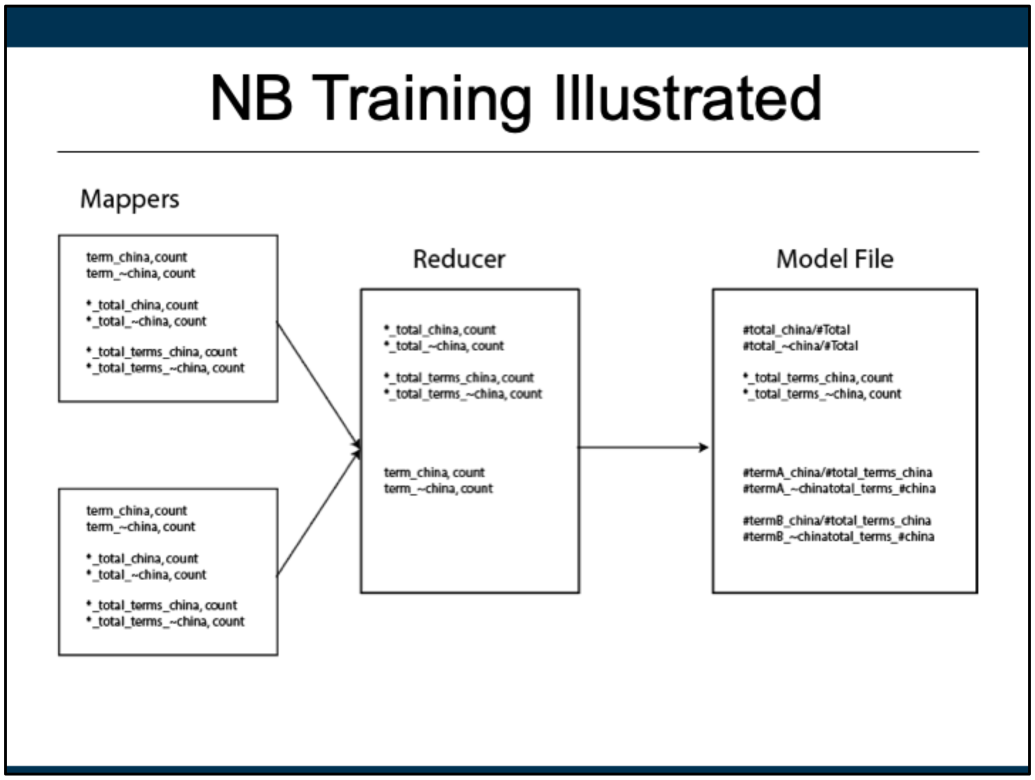


We Will Need To Calculate …

- Priors: Nc/N the relative frequencies of each class
- Conditional probabilities of each term: P(t|c)
- (#china)/(sum(#china,#~china))
- (#~china)/(sum(#china,#~china))
- (#term_china)/(#terms_in_china)
- (#term_~china)/(#terms_in_~china)

Sometimes it helps to think about the algorithm design by working our way backwards. We'll first establish what information our reducers are going to need, and then we'll make sure that our mappers are emitting this information.

We'll use the order inversion pattern to make sure we have our class tallies available for each division

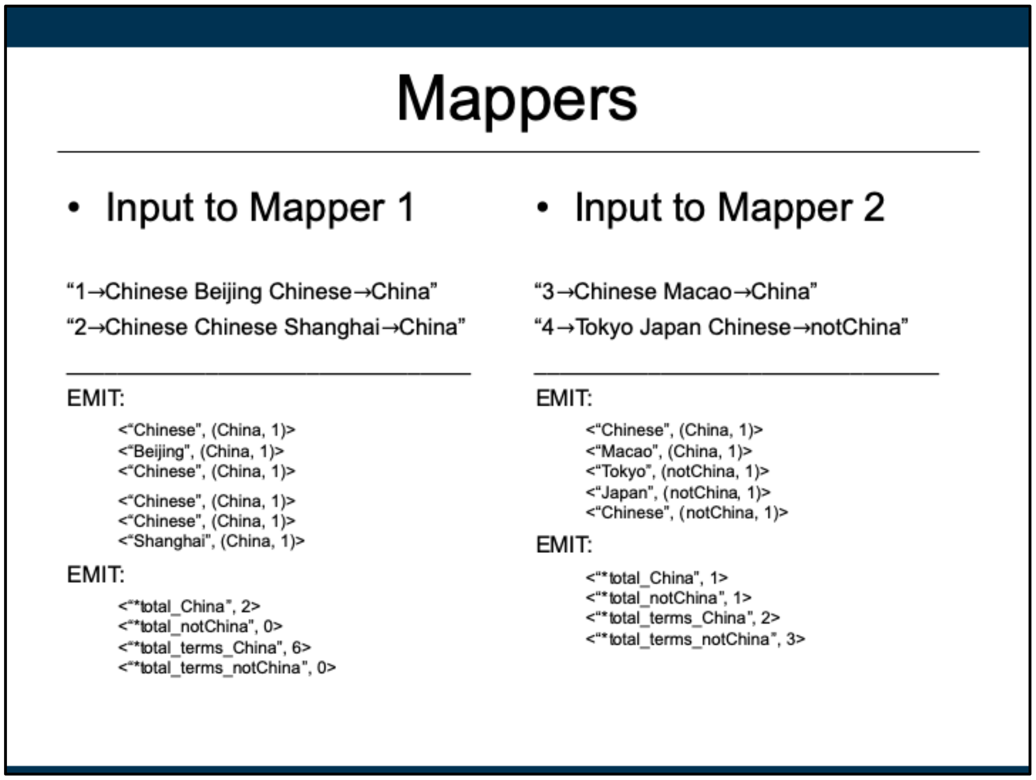


NB Training Illustrated

Mappers
term_china,count
term_~china, count
*_total_china,count
*_total_~china, count
*_total_terms_china, count
*_total_terms_~china, count
term_china,count
term_~china, count
*_total_china,count
*_total_~china, count
*_total_terms_china, count
*_total_terms_~china, count
Reducer
*_total_china,count
*_total_~china, count
*_total_terms_china,count
*_total_terms_~china, count
term_china, count
term_~china, count
Model File
#total_china/#Total
#total_~china/#Total
*_total_terms_china, count
*_total_terms_~china, count
#termA_china/#total_terms_china
#termA_~chinatotal_terms_#china
#termB_china/#total_terms_china
#termB_~chinatotal_terms_#china

To calculate likelihoods, the reducer will need access to the total number of terms in each class.
For the priors, the reducer will need access to the total number of documents in each class, as well as the total number of documents in the corpus.

What (key,value) pairs need to be emitted?



Mappers

- Input to Mapper 1

“1→Chinese Beijing Chinese→China”
“2→Chinese Chinese Shanghai→China”

EMIT:

```
<“Chinese”, (China, 1)>
<“Beijing”, (China, 1)>
<“Chinese”, (China, 1)>

<“Chinese”, (China, 1)>
<“Chinese”, (China, 1)>
<“Shanghai”, (China, 1)>
```

EMIT:

```
<“*total_China”, 2>
<“*total_notChina”, 0>
<“*total_terms_China”, 6>
<“*total_terms_notChina”, 0>
```

- Input to Mapper 2

“3→Chinese Macao→China”
“4→Tokyo Japan Chinese→notChina”

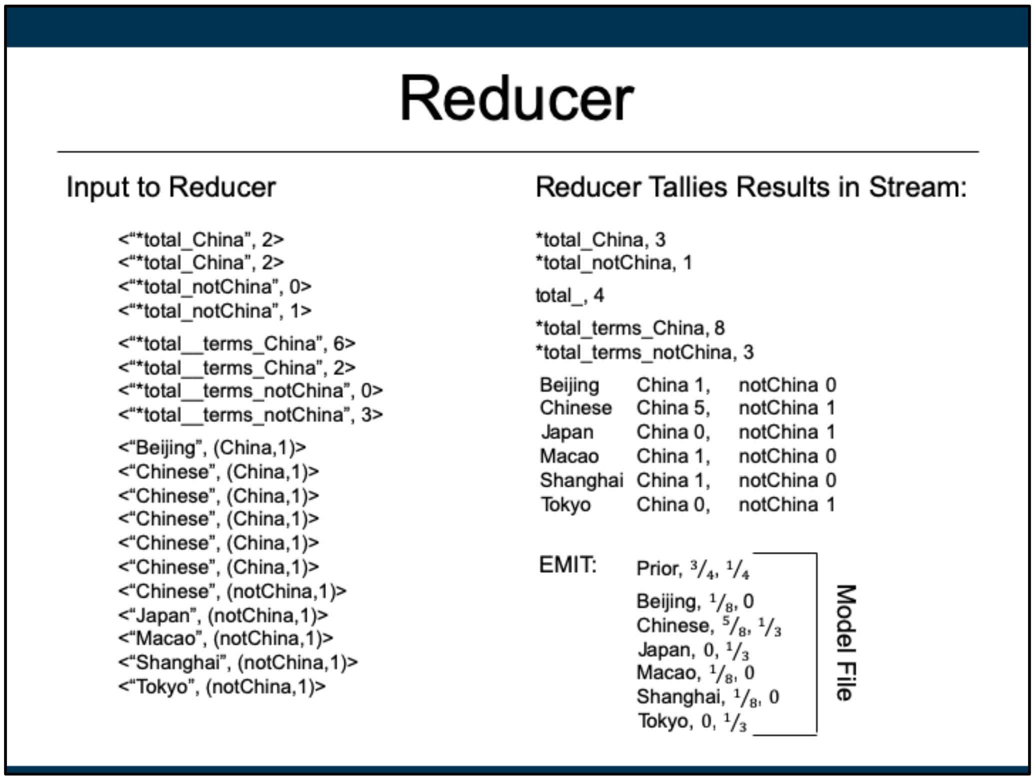
EMIT:

```
<“Chinese”, (China, 1)>
<“Macao”, (China, 1)>
<“Tokyo”, (notChina, 1)>
<“Japan”, (notChina, 1)>
<“Chinese”, (notChina, 1)>
```

EMIT:

```
<“*total_China”, 1>
<“*total_notChina”, 1>
<“*total_terms_China”, 2>
<“*total_terms_notChina”, 3>
```

We can use the order inversion pattern we talked about earlier, and emit all of our totals with a special key that starts with an Asterix character, guaranteeing that it will be seen in the reducer before any other words.



Reducer

Input to Reducer

<“*total_China”, 2>
<“*total_China”, 2>
<“*total_notChina”, 0>
<“*total_notChina”, 1>

<“*total__terms_China”, 6>
<“*total__terms_China”, 2>
<“*total__terms_notChina”, 0>
<“*total__terms_notChina”, 3>

<“Beijing”, (China,1)>
<“Chinese”, (China,1)>
<“Chinese”, (China,1)>
<“Chinese”, (China,1)>
<“Chinese”, (China,1)>
<“Chinese”, (China,1)>
<“Chinese”, (notChina,1)>
<“Japan”, (notChina,1)>
<“Macao”, (notChina,1)>
<“Shanghai”, (notChina,1)>
<“Tokyo”, (notChina,1)>

Reducer Tallies Results in Stream:

*total_China, 3
*total_notChina, 1

total_, 4

*total_terms_China, 8
*total_terms_notChina, 3

Beijing	China 1,	notChina 0
Chinese	China 5,	notChina 1
Japan	China 0,	notChina 1
Macao	China 1,	notChina 0
Shanghai	China 1,	notChina 0
Tokyo	China 0,	notChina 1

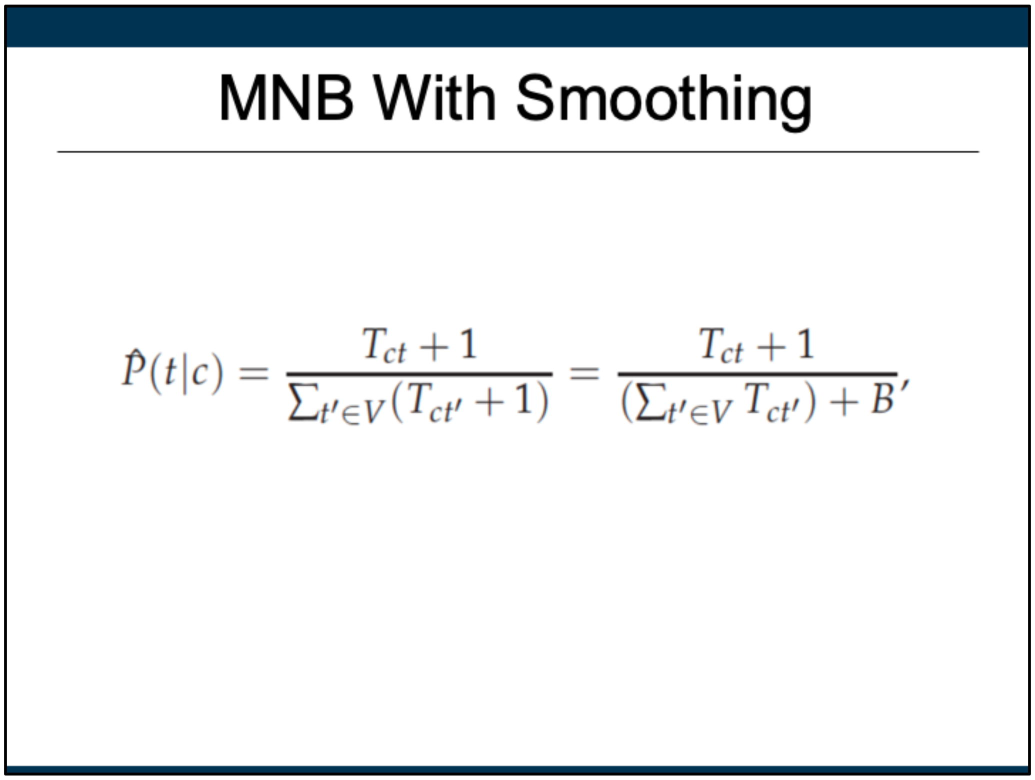
EMIT:

Prior, $^{3}/_{4}$, $^{1}/_{4}$

Beijing, $^{1}/_{8}$, 0
Chinese, $^{5}/_{8}$, $^{1}/_{3}$
Japan, 0, $^{1}/_{3}$
Macao, $^{1}/_{8}$, 0
Shanghai, $^{1}/_{8}$, 0
Tokyo, 0, $^{1}/_{3}$

Model File

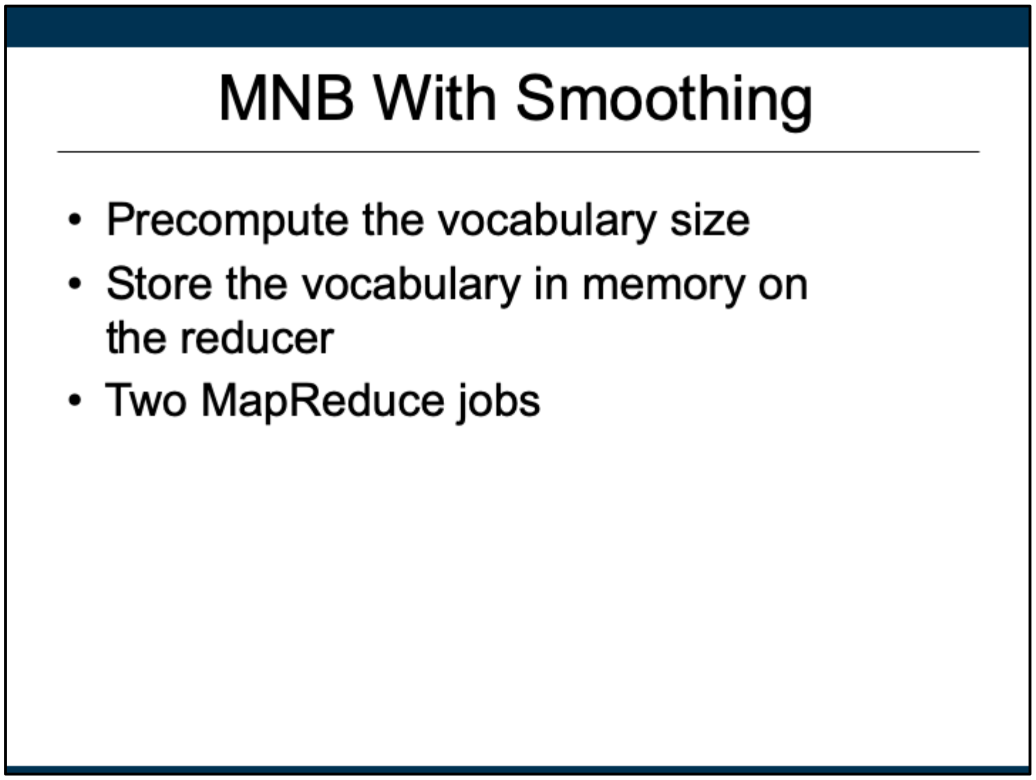
Now we can calculate all the marginals before we use them in the denominator to calculate the conditional probabilities of our training set.



MNB With Smoothing

$$\hat{P}(t|c) = \frac{T_{ct} + 1}{\sum_{t' \in V}(T_{ct'} + 1)} = \frac{T_{ct} + 1}{(\sum_{t' \in V} T_{ct'}) + B'},$$

So that was OK, but… Unfortunately it doesn’t get us very far. In order to account for unseen class examples in the training set, we’ll need to add LaPlace smoothing - otherwise known as “add 1” smoothing as we saw in the previous section.
The biggest difference in terms of algorithm design is the fact that we now need access to the vocabulary size in the denominator.

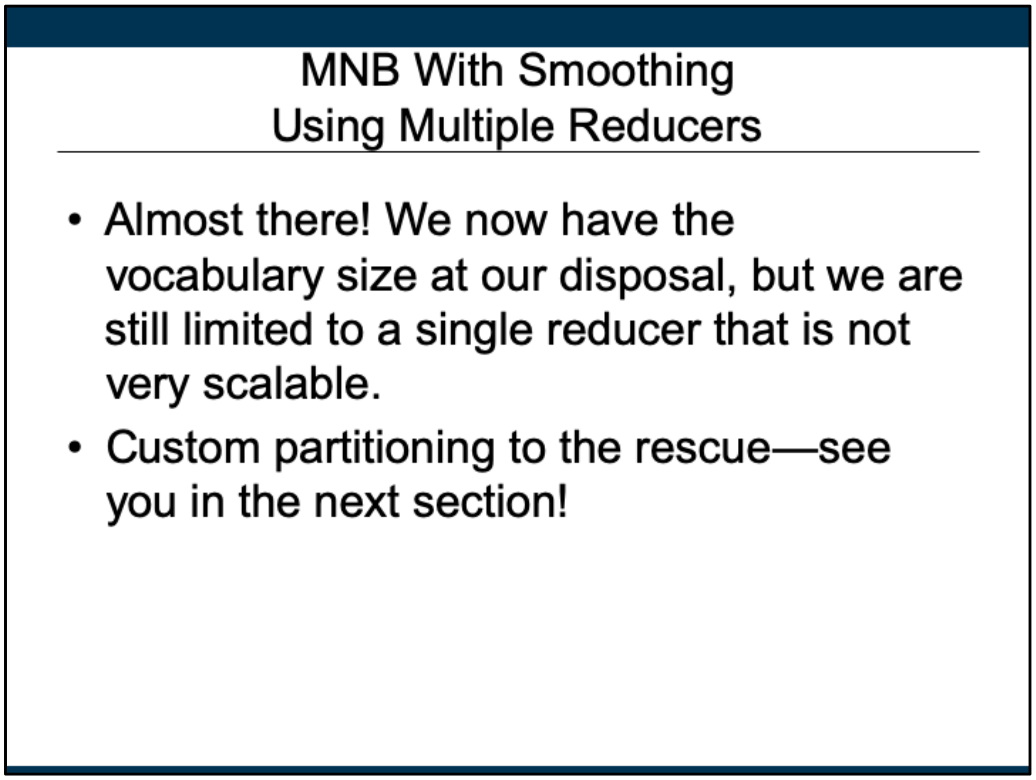


MNB With Smoothing

- Precompute the vocabulary size
- Store the vocabulary in memory on the reducer
- Two MapReduce jobs

How can we obtain the vocabulary size? Wouldn't it be nice if we already knew the size of the vocabulary and we could just pass it in as a parameter to our mapreduce job?

The only change we need to make is on the reduce side where we calculate the conditional probabilities.



MNB With Smoothing Using Multiple Reducers

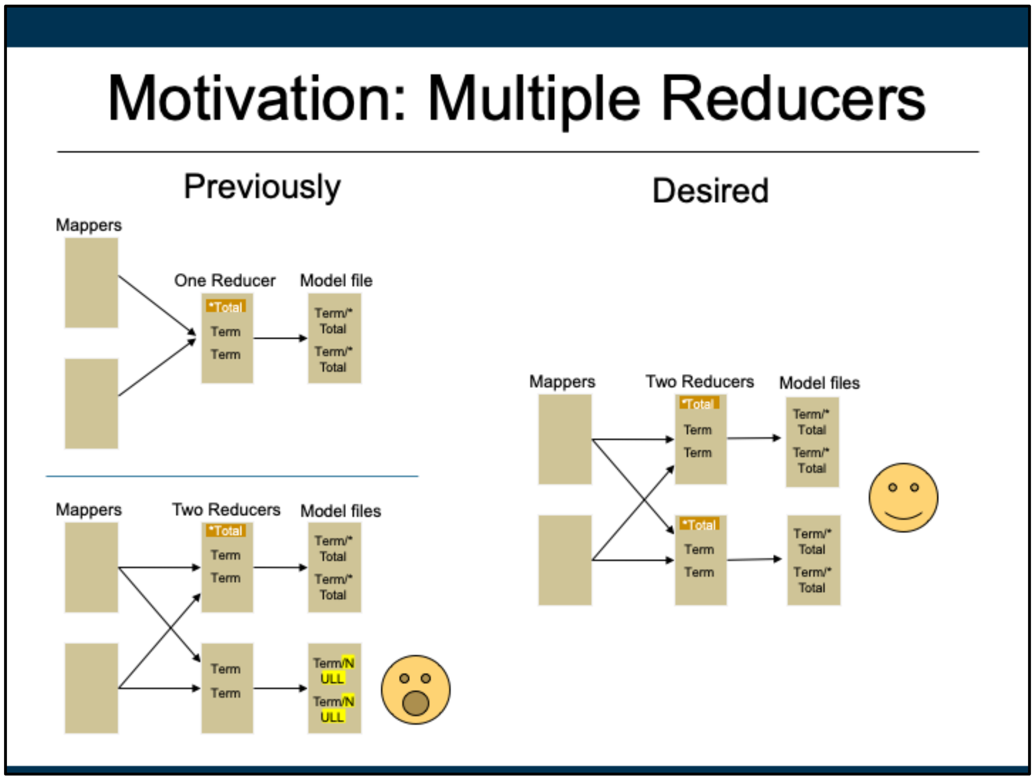
- Almost there! We now have the vocabulary size at our disposal, but we are still limited to a single reducer that is not very scalable.
- Custom partitioning to the rescue—see you in the next section!

You may have noticed that this entire section we've only used a single reducer because we needed to know the total counts for both the terms and the documents. Even with the order inversion pattern, that doesn't get us very far if the data is too big to fit in memory on a single machine.

Naive Bayes at Scale

The End

Custom Partitioning

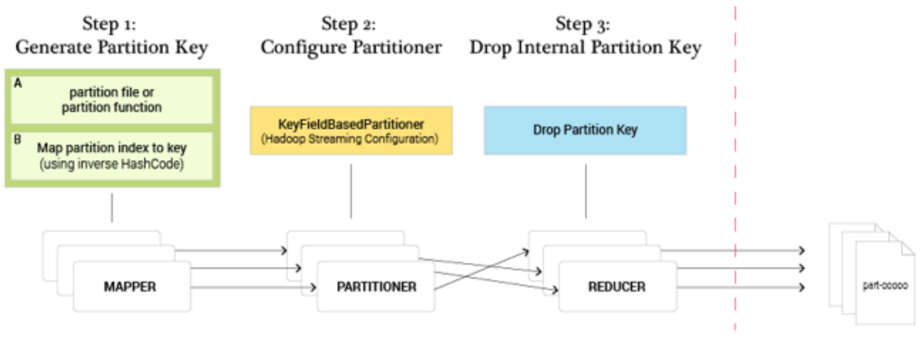


Motivation: Multiple Reducers

Our order inversion pattern works great with a single reducer but isn't enough to solve our problem at scale. Remember that the values for a given key are guaranteed to end up in the same reducer. This goes for our special *total key as well. Unfortunately this also means that none of our other reducers will receive any totals and the application will fail, as can be seen in the lower left diagram.

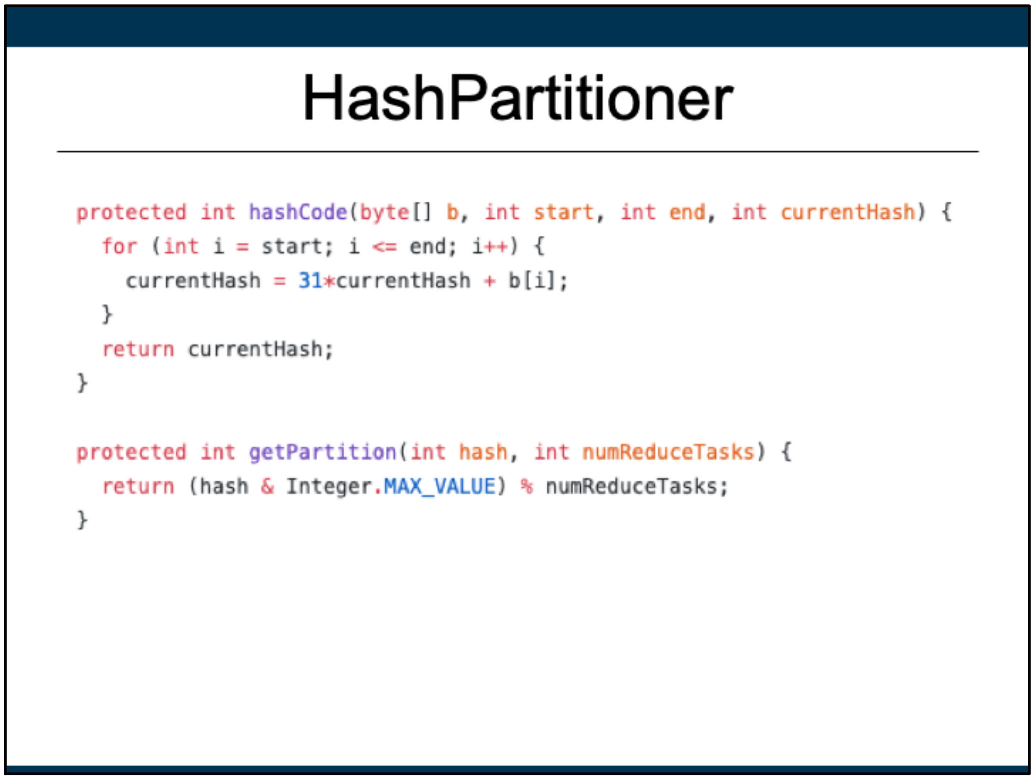
To obtain the desired result as in the diagram on the right, we'll need more control over which keys go to which reducers.

HashPartitioner


Step 1:
Generate Partition Key
Step 2:
Configure Partitioner
Step 3:
Drop Internal Partition Key
A
partition file or partition function
B
Map partition index to key (using inverse HashCode)
KeyFieldBasedPartitioner
(Hadoop Streaming Configuration)
Drop Partition Key
MAPPER
PARTITIONER
REDUCER
part-00000

In this section we introduce one of the most important design considerations for any distributed computing framework: Partitioning.
We'll also see how we can leverage custom partitioning to implement MNB with Smoothing using multiple reducers.
With custom partitioning, we can control data locality and reduce the amount of I/O and traffic over the network.
How we partition our data lies at the heart of writing efficient MapReduce applications whether we're using Hadoop, Spark, or any other distributed framework.

What we're looking at here is a high level schematic describing the steps to implement custom partitioning in Hadoop.



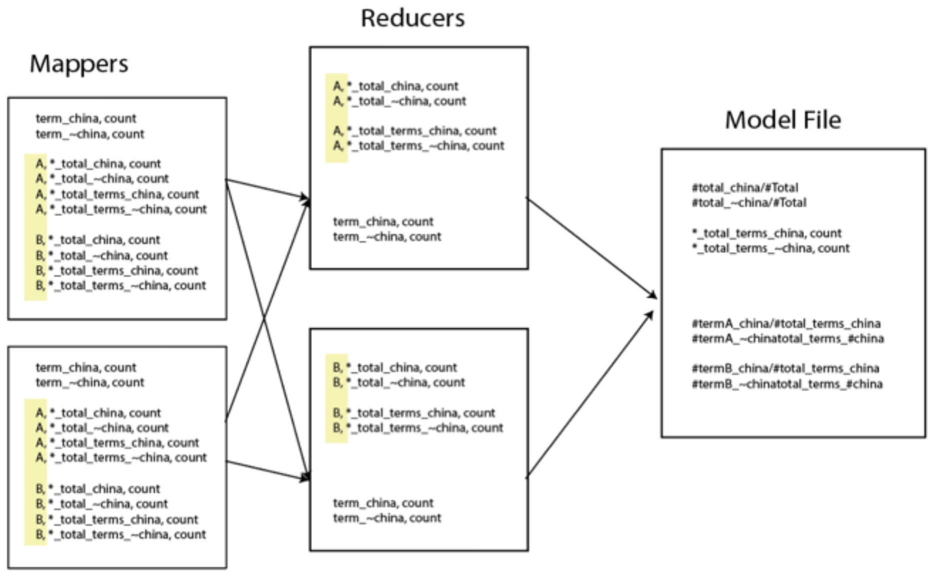
HashPartitioner

```
protected int hashCode(byte[] b, int start, int end, int currentHash) {
  for (int i = start; i <= end; i++) {
    currentHash = 31*currentHash + b[i];
  }
  return currentHash;
}

protected int getPartition(int hash, int numReduceTasks) {
  return (hash & Integer.MAX_VALUE) % numReduceTasks;
}
```

By default, Hadoop MapReduce uses a hash function, modulo the number of reducers, to partition data emitted from the mappers by key.
As we saw in previous weeks, the keys (and their associated values) are then sorted and sent to the reducers, such that the reducers are guaranteed to receive all of the values for a given key.
Up to now, we've let the framework do its business without any input from us, the programmer. This is both a convenience and a benefit that the framework provides.

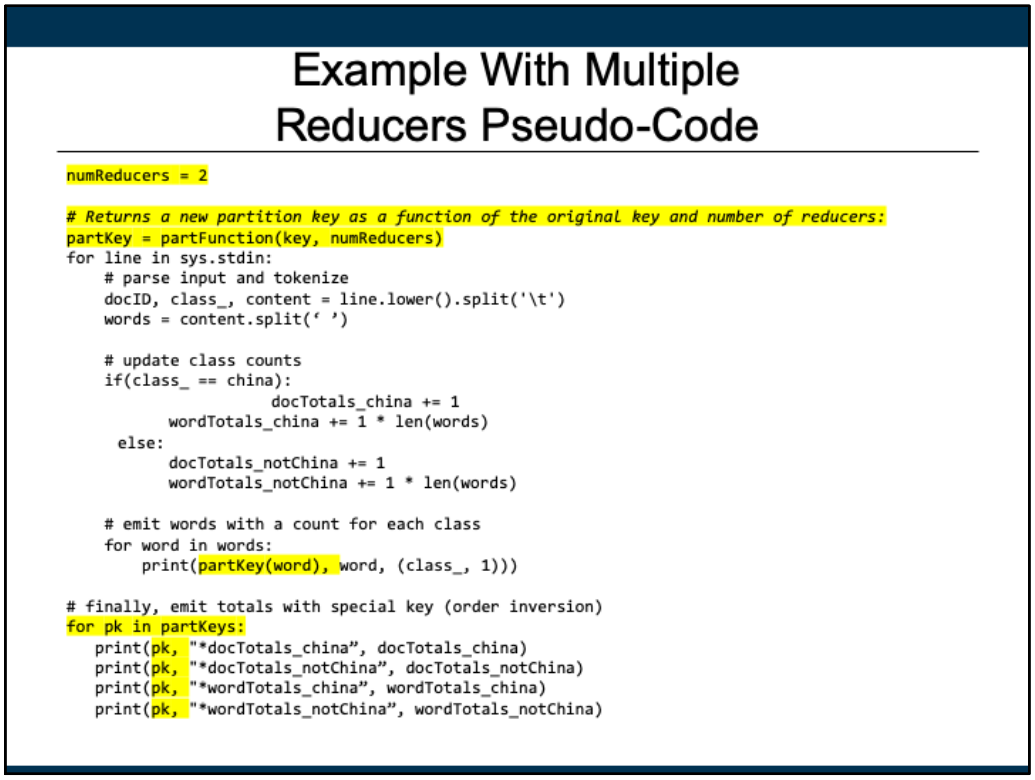
But what if we wanted to control which keys get sent to which reducers?

Example With Multiple Reducers Illustrated


Reducers
Mappers
Model File
term_china, count
term_~china, count
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
term_china, count
term_~china, count
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
term_china, count
term_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
term_china, count
term_~china, count
#total_china/#Total
#total_~china/#Total
*_total_terms_china, count
*_total_terms_~china, count
#termA_china/#total_terms_china
#termA_~chinatotal_terms_#china
#termB_china/#total_terms_china
#termB_~chinatotal_terms_#china

By emitting an additional key from the mappers, we can leverage the framework's default partitioning behavior to our advantage. Let's say I have two reducers. I want my mappers to emit the terms and counts just as before, but I also want the mappers to emit the totals such that they end up in both reducers. I can prepend a new partition key to my key value pairs, and then emit all of my totals twice - once to each reducer.

TODO: pre-partition keys for terms need to be illustrated: A, term_china, count



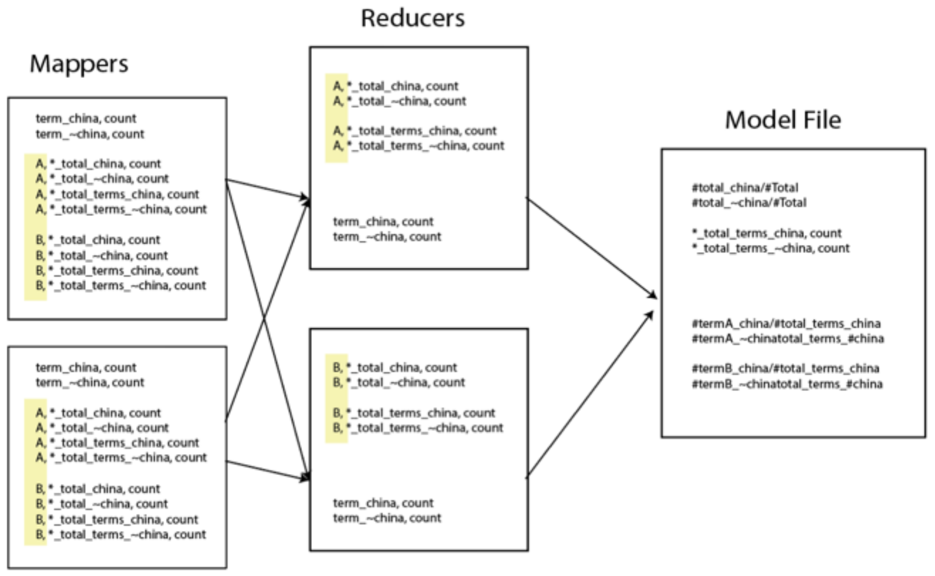
Example With Multiple Reducers Pseudo-Code

```
numReducers = 2

# Returns a new partition key as a function of the original key and number of reducers:
partKey = partFunction(key, numReducers)
for line in sys.stdin:
    # parse input and tokenize
    docID, class_, content = line.lower().split('\t')
    words = content.split(' ')

    # update class counts
    if(class_ == china):
                      docTotals_china += 1
           wordTotals_china += 1 * len(words)
      else:
           docTotals_notChina += 1
           wordTotals_notChina += 1 * len(words)

    # emit words with a count for each class
    for word in words:
        print(partKey(word), word, (class_, 1)))

# finally, emit totals with special key (order inversion)
for pk in partKeys:
   print(pk, "*docTotals_china", docTotals_china)
   print(pk, "*docTotals_notChina", docTotals_notChina)
   print(pk, "*wordTotals_china", wordTotals_china)
   print(pk, "*wordTotals_notChina", wordTotals_notChina)
```

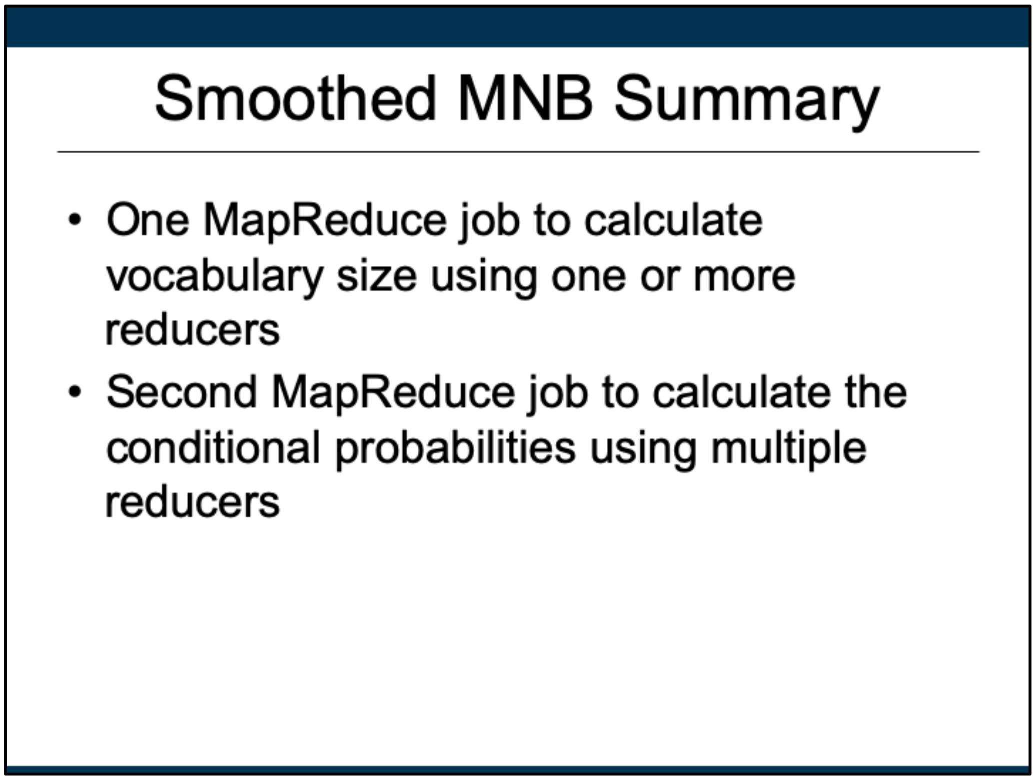
Using our previous pseudo-code example, we'll need to add a couple of things.

First of all, we need to know the number of reducers. This value can be hardcoded, or better yet, read from the Hadoop configuration.
Then we need to define a partition function which takes as input the original key, and the number of reducers. This function returns a new key which we will prepend to all of our key value pairs.
Finally, for each new partition key, we'll emit all of the totals.

Example With Multiple Reducers Illustrated


Mappers
term_china, count
term_~china, count
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
term_china, count
term_~china, count
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
Reducers
A, *_total_china, count
A, *_total_~china, count
A, *_total_terms_china, count
A, *_total_terms_~china, count
term_china, count
term_~china, count
B, *_total_china, count
B, *_total_~china, count
B, *_total_terms_china, count
B, *_total_terms_~china, count
term_china, count
term_~china, count
Model File
#total_china/#Total
#total_~china/#Total
*_total_terms_china, count
*_total_terms_~china, count
#termA_china/#total_terms_china
#termA_~chinatotal_terms_#china
#termB_china/#total_terms_china
#termB_~chinatotal_terms_#china

Here it is again, illustrated. It should now be clear that we've obtained the result we were looking for.



Smoothed MNB Summary

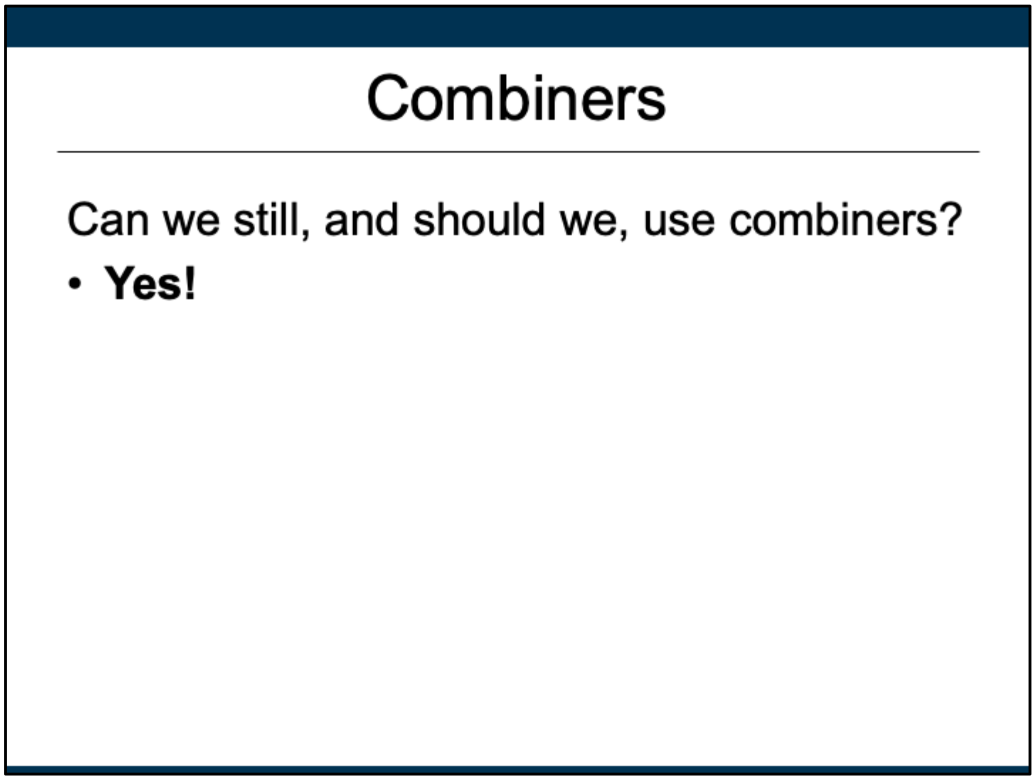
- One MapReduce job to calculate vocabulary size using one or more reducers
- Second MapReduce job to calculate the conditional probabilities using multiple reducers

It should be clear now that we can calculate the entire MNB model with smoothing in a single map reduce job if we limit ourselves to a single reducer.

With multiple reducers we can calculate the entire MNB model without smoothing in a single MapReduce job.

Unfortunately, it's not possible to calculate the vocabulary size in this setting, so a smoothed MNB model still requires two steps.

1. One to calculate the vocabulary
2. And two, to calculate all the conditional probabilities.



Combiners

Can we still, and should we, use combiners?

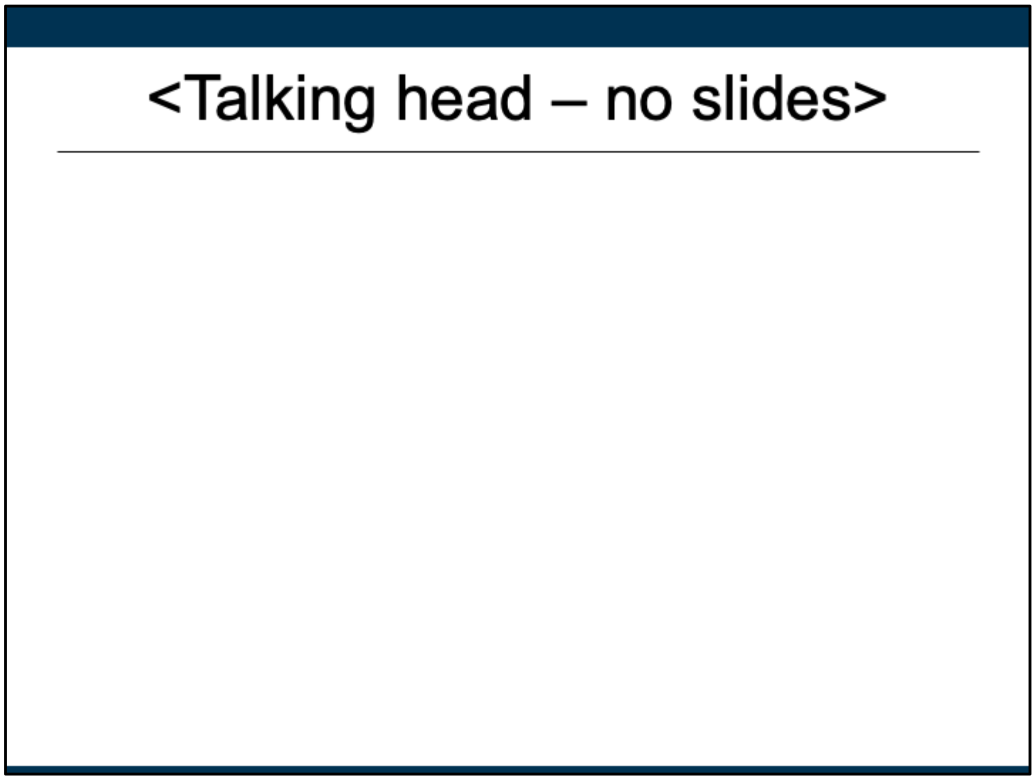
- **Yes!**

You will be implementing Naïve Bayes in the homework. Your job is to leverage the design patterns from this week including combiners. Have fun!

Custom Partitioning

The End

Week 3 Summary


<Talking head – no slides>

We are almost at the end of this lecture. It's hard to believe that we started this lectu
re a long time ago talking about how important it is to think about RAM, and the relat
ionship to disc
space and to bandwidth and how we leverage our powers of 10 back of envelope calc
ulations. We talked about priority queues. We talked about merge sorts. We saw how
these all get to play a very important role in Hadoop. In particular, they play a key rol
e in the Hadoop shuffle operation. And the shuffle operation includes all steps betwe
en the mapper output and the average user input. And man, is that a lot of work. The
re is a lot of work going on there in terms of buffering data, partitioning data, sorting
data, spilling data to disk, merging data, transferring data, merge sorting. It's a lot of
work. So the shuffle is the heart and soul of the MapReduce framework.
We also got to look at some design patterns that help us build mappers and reducers
and really solve problems at scale within the MapReduce framework. We looked at th
e use of combiners. We looked at in-memory mappers. We looked at order
inversion and how we could solve problems in different ways and basically incur differ
ent network costs. And we looked at computing relative frequencies. We introduced
other design patterns, as well, including custom partitioning.
What is key in MapReduce is synchronization and communication. We got to see seve
ral examples in this lecture, where we constructed complex keys and values that bring
together data necessary for computation. This is used in all of the above design patte
rns. We got to see how we control partitioning of intermediate data. This is used, in p
articular, for order inversion and for value

to key type conversions as well. So controlling the sort order of intermediate keys is al
so important, and so we really leveraged the shuffle part of the MapReduce framewo
rk.

This concludes our overview of MapReduce algorithm design. It should be clear by no
w that although the programming model forces us to express algorithms in terms of a
 small set of rigidly defined components, there are many tools at one's disposals to sh
ape the flow of computation. These patterns apply not just to Hadoop MapReduce, b
ut also to Spark, as we shall see in the remaining part of this class.