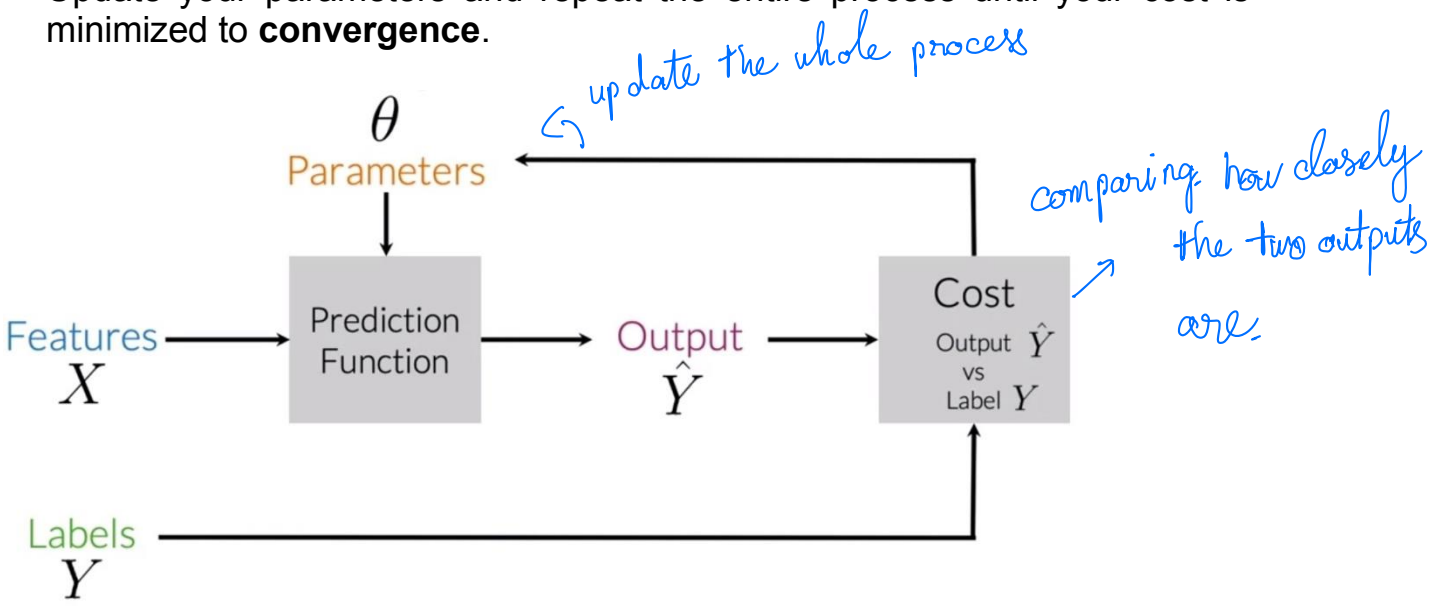
Week 1: Sentiment Analysis Using Logistic Regression

Overview

- Natural Language Processing (NLP) has changed a lot over the last several decades. The field has started off using primarily **rule-based systems** where someone might a rule like when you see the word *good*, assume this is a positive customer review. Next, came using probabilistic systems that perform much better, but still require a lot of hand engineering. Today, NLP relies much more on machine learning and deep learning.
- More recently with the rise of powerful computers, we can now train end-to-end systems that would have been impossible to train a few years ago. We can now capture more **complicated patterns** and we can use these models in **question answering, sentiment analysis, language translation, text summarization, chatbots**, etc.
- Many of these applications are built with attention models, which you are going to learn in this specialization. A few years ago, these models would take weeks or even months to train. But with attention, you can train these models in just a few hours.
- In this specialization, you learn to build NLP technologies including the same technologies as what's deployed in many large commercial systems.
- In course 1 of the Natural Language Processing Specialization offered by deeplearning.ai, you will:
  - Perform (positive/negative) **sentiment analysis** of tweets using:
    - Logistic regression
    - Naive Bayes.
  - Learn to represent words, queries and documents including other pieces of text as **numbers in vectors**. Use vector space models to discover relationships between words and use PCA to reduce the dimensionality of the vector space and visualize those relationships.
  - Write a simple English to French neural machine translation algorithm using **pre-computed word embeddings**. Learn about **locality sensitive hashing** to relate words via approximate k-nearest neighbor search, to make the process of searching more efficient.
- Pre-requisites: Python programming, basic knowledge of machine learning, matrix multiplications, and conditional probability.

Supervised Machine Learning

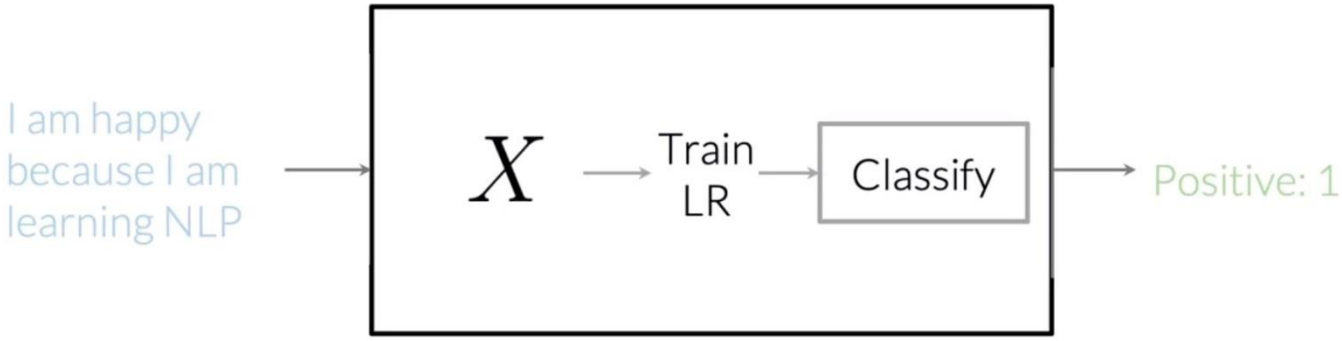
- In supervised machine learning, you have input features XX and a set of labels YY.
- To make sure you're getting the most accurate predictions based on your data, the goal is to minimize your error rates or cost as much as possible.
- To do this, run the prediction function which takes in parameters to map your features to output labels Y^Y^.
- The best mapping from features to labels is achieved when the **difference** between the expected values YY and the predicted values YY is **minimal**.
- The cost function does this by comparing how closely the **output** Y^Y^ is to the **label** YY.
- Update your parameters and repeat the entire process until your cost is minimized to **convergence**.



Sentiment Analysis

- In sentiment analysis, the objective is to predict whether a body of text, say a tweet, has a positive or a negative sentiment.
  - For e.g., let's say you have 1,000 product reviews. Sentiment analysis lets you build a system to automatically go through all of these product reviews to figure out what fraction of them are positive reviews vs. negative reviews.

- Let's model sentiment analysis as a supervised machine learning classification task using logistic regression.
- Building a logisitic regression classifier that performs sentiment analysis on tweets can be done in 3 steps:
  - **Extract features:** process the raw tweets in the training set and extract useful features.
    - Tweets with a positive sentiment have a label of 1, while those with a negative sentiment have a label of 0.
  - **Train:** Train your logistic regression classifier while minimizing the cost.
  - **Predict:** Come up with predictions using your learned model.



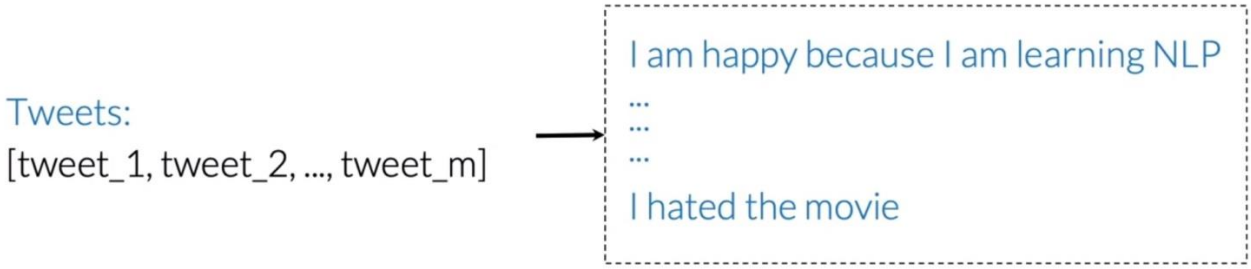
- For implementation purposes, Natural Language Toolkit (NLTK) is a popular open-source Python library for natural language processing. It offers modules for collecting, handling, and processing Twitter data and much more.

Feature Extraction

Sparse Representations

- To represent text as a vector, we need a **vocabulary** V that allows us to encode any body of text as a vector, more broadly, as an array of numbers.
- The vocabulary V would be the list of **unique words** from your list of tweets.
  - Building such a list requires going through all the words from all your tweets and saving every new word that appears in your search.
- In the example below, you would have the word *I*, then the word, *am*, then, *happy*, and so forth.

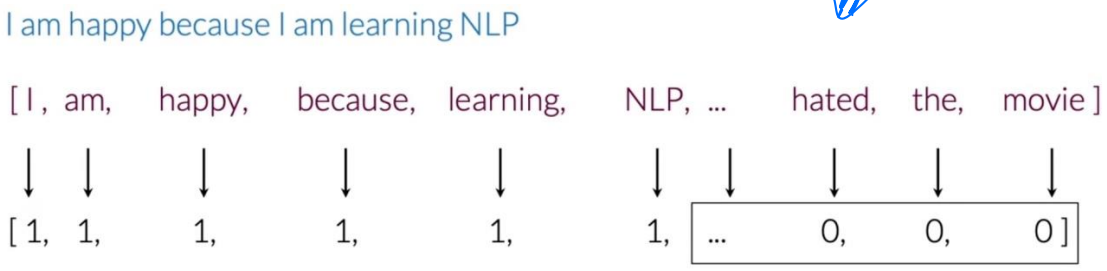
- Note that in this case, the word *I* and the word *am* would **not be repeated** in the vocabulary (since a vocabulary only holds unique words).



$V =$

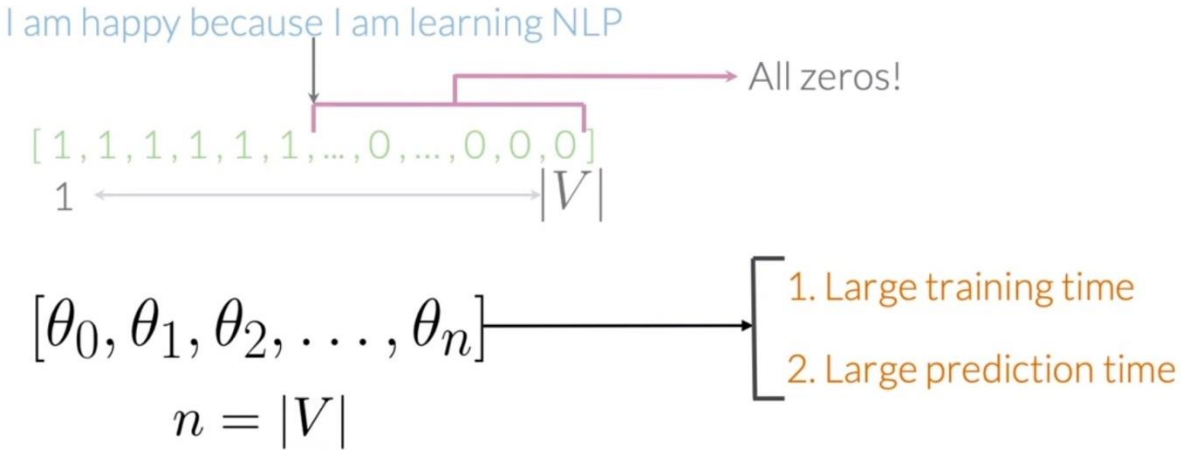
[I, am, happy, because, learning, NLP, ... hated, the, movie]

- To extract features using your vocabulary V with sparse representation, you would have to check if every word from your tweet appears in the vocabulary.
- If it does, like in the case of the word *I*, you would assign a value of 1 to that feature. If it doesn't appear in the vocabulary, you would assign a value of 0.
- In the below example, the representation of our sample tweet would have 6 ones and many zeros, which correspond to every unique word from your vocabulary that **isn't** in the tweet.
- This type of representation with a relatively small number of **non-zero values** is called a **sparse representation**, as shown below.



- **(-) Sparse representation leads to inefficiencies during training and testing:** The number of features in a sparse representation is equal to the size of your entire vocabulary V. This implies that a lot of features would be equal to 0 for every tweet. With the sparse representation, a logistic regression model would have to learn $n+1$ parameters, where n would be equal to the size of your vocabulary V. For large vocabulary sizes, this

would make your model inefficient in terms of training and inference/prediction/test time.

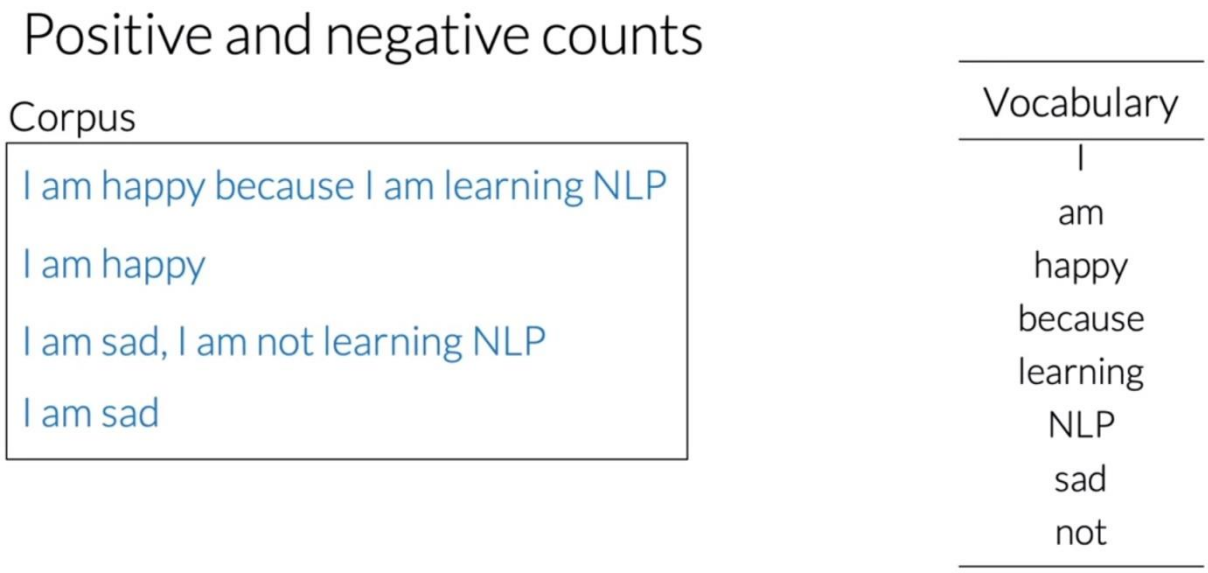


- For e.g., the number of parameters that a logistic regression classifier would have to learn using spare representations for feature extraction for the below inputs is 14.
  - `["I am happy because I am learning NLP", "I hated that movie", "I love working at DL"]`
- Summary:
  - Given some text, you can represent this text as a vector of dimension V by extracting features. Specifically, you did this for a tweet and you were able to build a vocabulary of dimension V.
  - As V gets larger, your training and inference/prediction/test time scales accordingly.

Negative and Positive Frequencies

- Frequencies (or counts) can serve as **features** into your logistic regression classifier. Specifically, given a word, you want to keep track of the number of times it shows up in a particular class.
- Let's build two classes:
  - A class associated with positive sentiment and,
  - Another class associated with negative sentiment.
- To get the **positive frequency** for a word in your vocabulary, count the **number of times** it appears in the **positive** tweets. Correspondingly, obtain the **negative frequency** for a word by counting the number of times it appears in the **negative** tweets.
- Remember to sum the frequencies of **unique words**, and **not duplicates**.

- Using both those counts, extract features and feed them into your logistic regression classifier.
- As an e.g., consider a corpus consisting of four tweets as shown below. Associated with that corpus, you would have a set of unique words, i.e., your vocabulary V. In the example below, your vocabulary V has **eight unique words**, as shown below.



- For your corpus above, you have a set of **two tweets** that belong to the **positive** class, and a sets of **two tweets** that belong to the **negative** class. To get the **positive frequency** in any word in your vocabulary, you will have to count the times as it appears in the **positive tweets**. For instance, the word *happy* appears one time in the first positive tweet, and another time in the second positive tweet, making it's positive frequency 2. Shown below is the positive frequency table (i.e., the positive count table).

Positive and negative counts

Positive tweets

I am happy because I am learning NLP
I am happy

Vocabulary	PosFreq (1)
I	3
am	3
happy	2
because	1
learning	1
NLP	1
sad	0
not	0

- Similarly, to obtain negative frequencies, consider the word *am* which appears 2 times in the first tweet and another time in the second one. So it's negative frequency is 3. Shown below is the overall negative frequency table.

Positive and negative counts

Vocabulary	NegFreq (0)
I	3
am	3
happy	0
because	0
learning	1
NLP	1
sad	2
not	1

Negative tweets

I am sad, I am not learning NLP
I am sad

- The table below encapsulates the positive and negative frequencies for your sample corpus. In practice, when coding, this table is a dictionary mapping from a **(word, class) pair** to its **frequency**. This represents a **frequency dictionary/hash-table**, which maps a word and a class to the number of times that word showed up in the corresponding class.

Vocabulary	PosFreq (1)	NegFreq (0)
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1

freqs: dictionary mapping from (word, class) to frequency

- Thus, a frequency dictionary basically maps the following relationship:
  $(\text{word}, \text{class}) \rightarrow \text{frequencies}$ $(\text{word}, \text{class}) \rightarrow \text{frequencies}$

Feature Extraction with Frequencies

- Recall that…
  - The frequency of a word in a class is simply the number of times that the word appears on the set of tweets belonging to that class.
  - The table summarizing this information is basically a **dictionary mapping** from (word, class) pairs to frequencies. Put simply, it tells us how many times each word showed up in its corresponding class.
- Using sparse representations, you learned to encode a tweet as a vector of dimension V.
- Using negative and positive frequencies, you can encode a tweet by representing as a vector of dimension 3. This makes your logistic regression classifier faster, because instead of learning V features, you only have to learn 3 features.
- As an e.g., lets consider an arbitrary tweet t, shown below. Below are the list of features for this sample t:
  - **First:** bias unit equal to 1.
  - **Second:** sum of the positive frequencies for every unique word in t.
  - **Third:** sum of negative frequencies for every unique word in t.
- Thus, to extract features for this representation, you would only have to **sum frequencies of words**, which can be easily accomplished using the **frequency dictionary** concept we introduced in the prior section.

freqs: dictionary mapping from (word, class) to frequency

$$X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$$

Features of tweet m | Bias | Sum Pos. Frequencies | Sum Neg. Frequencies

- For instance, looking at the frequencies for the positive class (which is the **second feature** being sought above) for the tweet shown below, the only words from the vocabulary that don't appear on these tweets are *happy* and *because*. Summing up the frequencies for all the other words (underlined in the figure below), you'll get a value equal to 8.

Vocabulary	PosFreq (1)
I	3
am	3
happy	2
because	1
learning	1
NLP	1
sad	0
not	0

I am sad, I am not learning NLP

$$X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$$

8

- Next, looking at the frequencies for the negative class (which is the **third feature** being sought above) for the same tweet, you need to sum the negative frequencies of the words from the vocabulary that appear on the tweet. For this example, you should get 11 after summing up the underlined frequencies.

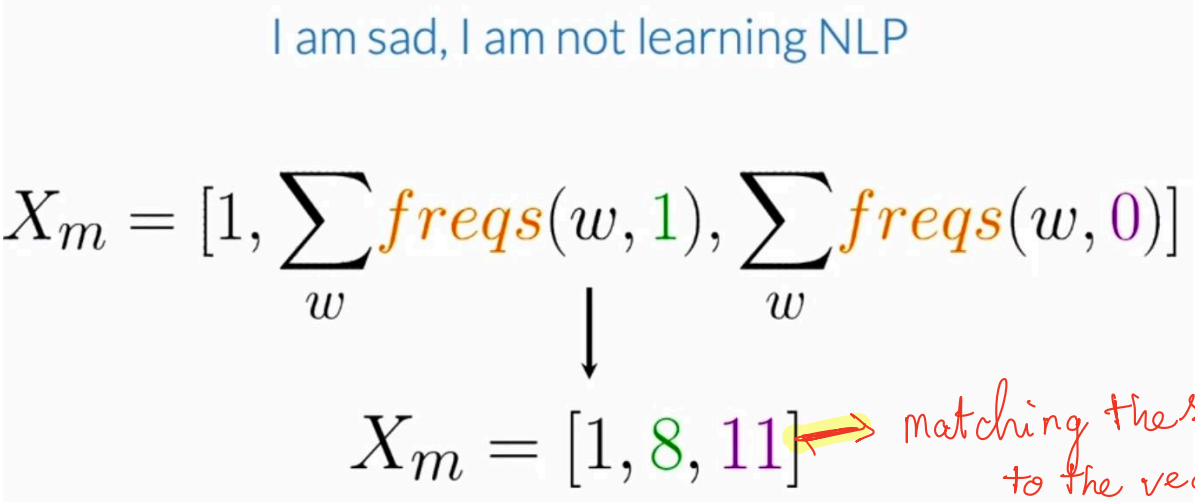
Vocabulary	NegFreq (0)
I	3
am	3
happy	0
because	0
learning	1
NLP	1
sad	2
not	1

I am sad, I am not learning NLP

$$X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$$

11

- So, for our given example above, this representation would be equal to the vector $[1,8,11]$[1,8,11], which is a vector of dimension 3, as shown below.



- In summary, you're using the frequency dictionary to extract features for the positive/negative class for sentiment analysis, to represent a tweet using a vector of dimension 3.
  - `[bias = 1, sum of the positive frequencies for every unique word on tweet, sum of the negative frequencies for every unique word on tweet]`
- Next, let's learn to pre-process your tweets and use these preprocessed words in place of words in your vocabulary.

polarity score

Preprocessing

- Preprocessing is the idea of distilling a dataset so that it only contains **essential** information for the downstream task. A byproduct of preprocessing is that it helps in **reducing the size of our vocabulary**, which in turns saves training and test time as discussed in the section on sparse representations.
- There are four major preprocessing concepts in NLP:
  - Tokenization
  - Removing stop words and punctuation
  - Stemming
  - Lower-casing

Tokenization

- To tokenize a string means to split the strings into individual words without blanks or tabs.

Stop Words and Punctuation Marks

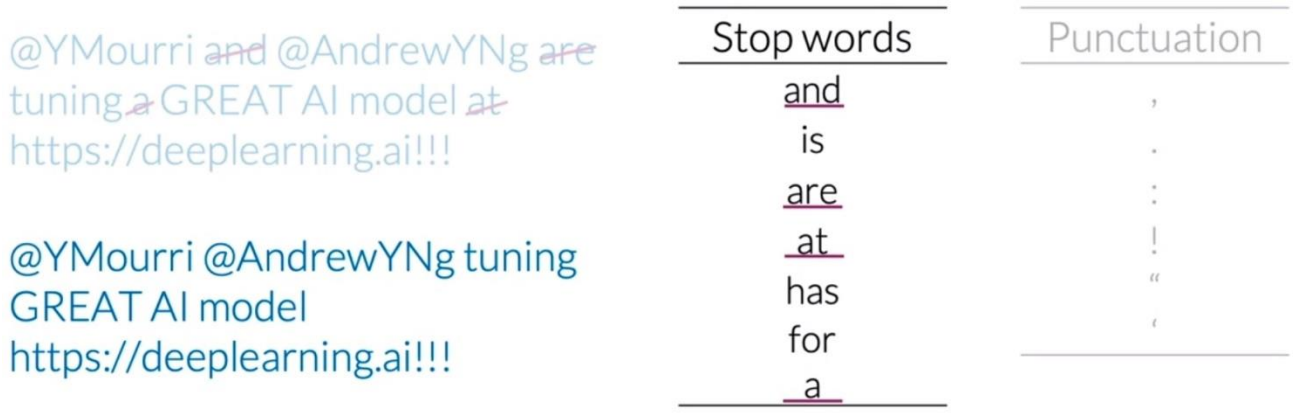
- Consider the below tweet. First, remove all the words that don't add significant meaning to the tweets, i.e., stop words and punctuation marks. In practice, you would compare your tweet against two lists, one with the stop words and another with punctuation.
  - Note that these lists are usually much larger, but for the purpose of this example, we've kept them short and simple.

@YMourri and @AndrewYNg are tuning a GREAT AI model at https://deeplearning.ai!!!

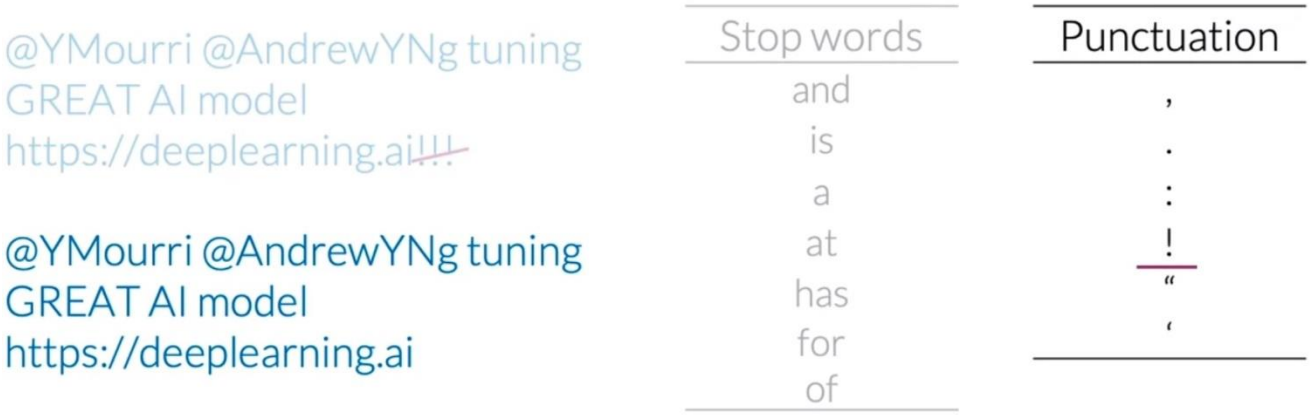
Stop words
and
is
are
at
has
for
a

Punctuation
,
.
:
!
“
‘

- Every word from the tweet that appears on the list of stop words should be eliminated. So you would have to eliminate the word *and*, the word *are*, the word *a*, and the word *at*.
- Show below in the figure is the tweet with the **stop words trimmed out**. Note that the overall meaning of the sentence can **still be inferred** without much effort.



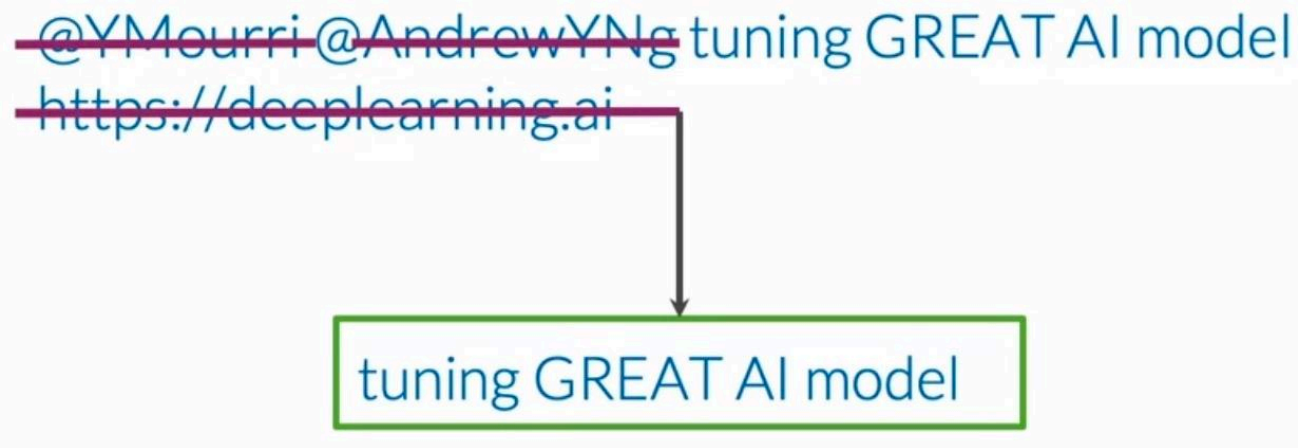
- Now, let's eliminate punctuation marks, which in the above e.g., are only exclamation marks.
- The preprocessed tweet without stop words and punctuation is shown below. However, note that in some contexts eliminating punctuation is **not desired** and in fact, adds important information to your specific task. You might thus need to customize the list of stop words depending on your task at hand.



- Note that certain groupings like `:)` and `...` should be retained when dealing with tweets because they are used to express emotions.

Task-dependent Stop Words

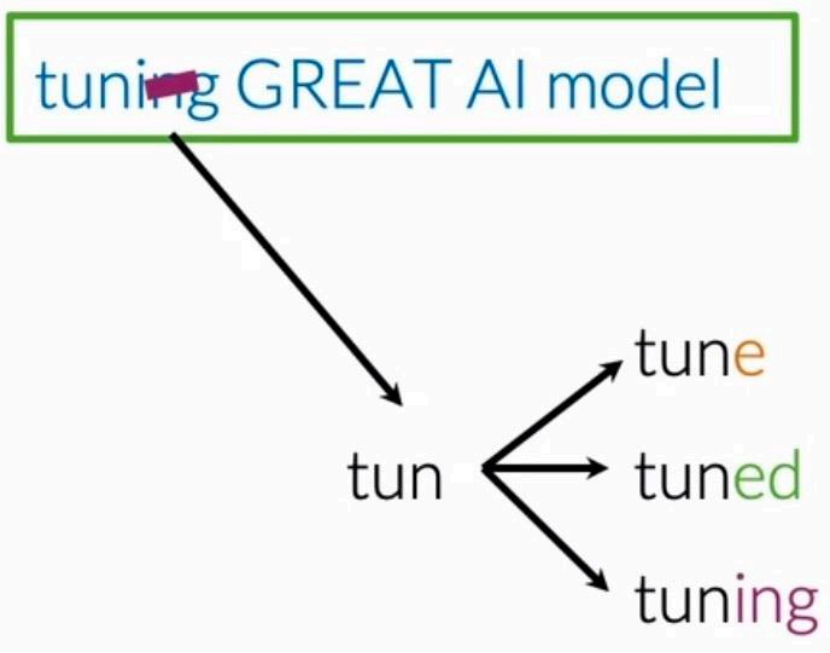
- Tweets often have handles, URLs, hashtags, stock market tickers and retweet marks, which generally **don't add any value** to our particular task of sentiment analysis. These can also be treated as another set of "stop words" for the purposes of preprocessing for the specific task at hand.
- Upon eliminating the two handles and URL in the tweet, the resulting tweets only contains important information related to its sentiment. *"Tuning GREAT AI model"* is clearly a positive tweet and a sufficiently good model should be able to classify it.



Stemming

- In NLP, stemming is simply transforming any word to its most general form or **base stem**, which can be defined as a set of characters that are used to construct the word and its derivatives.
- Let's take the first word *tuning* as an example (shown below). It's stem is *tun*, because:
  - Adding the letter *e*, makes it forms the word *tune*.
  - Adding the suffix *ed*, forms the word *tuned*.
  - Adding the suffix *ing*, it forms the word *tuning*.
- After you perform stemming on your corpus, the word *tune*, *tuned*, and *tuning* will **all** be reduced to the stem *tun*. This leads to your vocabulary being significantly reduced when you perform stemming for every word in the corpus, since you are effectively **eliminating all variations of a particular word, but one**.
- Some more examples:
  - *learn, learning, learned* and *learnt* stem from their common root: **learn**.

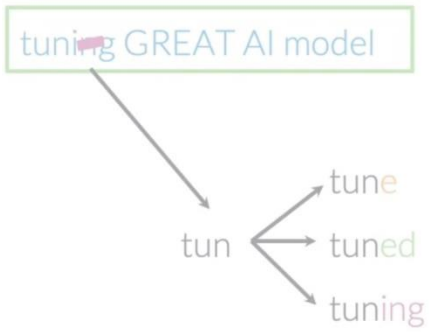
    - *motivation*, *motivated*, and *motivate* originate from their common stem: **motiv**.
- However, in some cases, stemming process produces words that are not **correct spellings** of the root word. For e.g., we can look at the set of words that comprises the different forms of happy: *happy, happiness* and *happier*. We can see that the prefix **happi** the most common stem throughout the entire set of related words. Note that we cannot choose **happ** because it is the stem of unrelated words like happen.

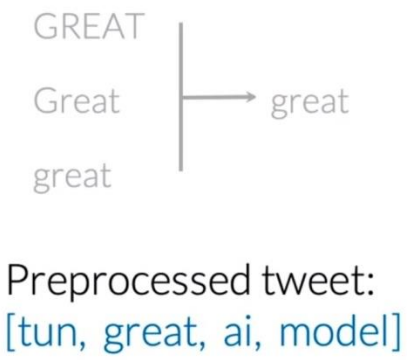


- The Porter Stemming Algorithm is a common algorithm for carrying out stemming in popular NLP libraries like NLTK.

Lower-casing

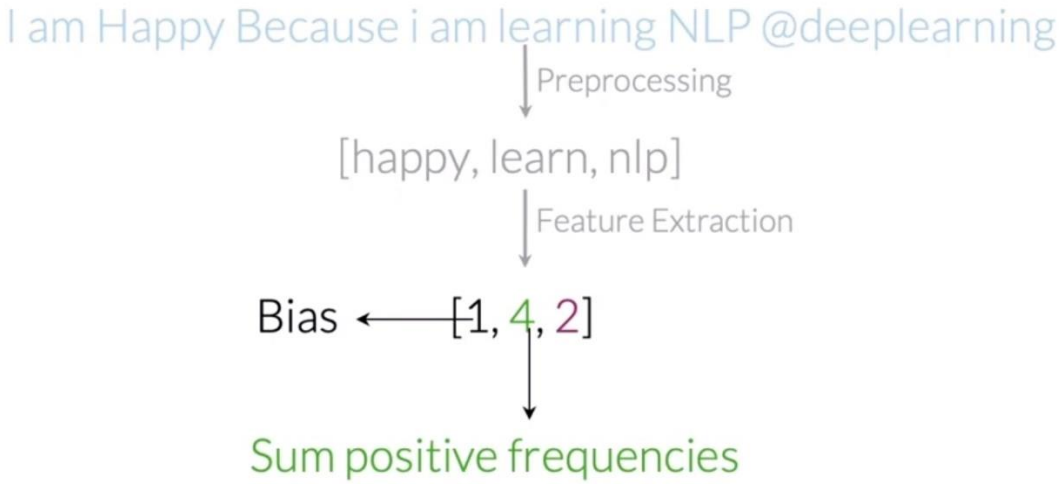
- To reduce your vocabulary even further without losing valuable information, you can **lowercase all words** in your tweet and then **sift out the unique ones**.
- So, in this case, the words *GREAT, Great* and *great* all would be treated as the same exact word. Thus, the final preprocessed-tweet as a list of words, becomes:



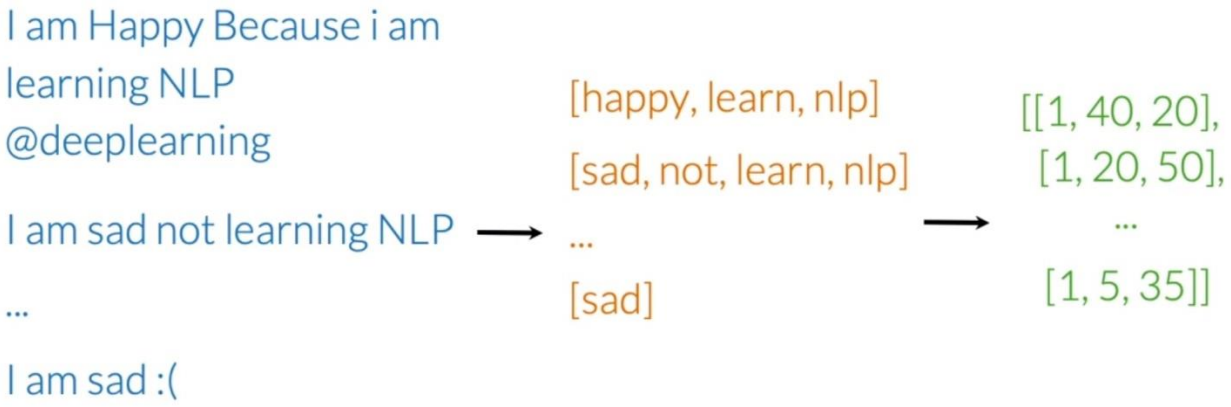


Generating the Features Matrix

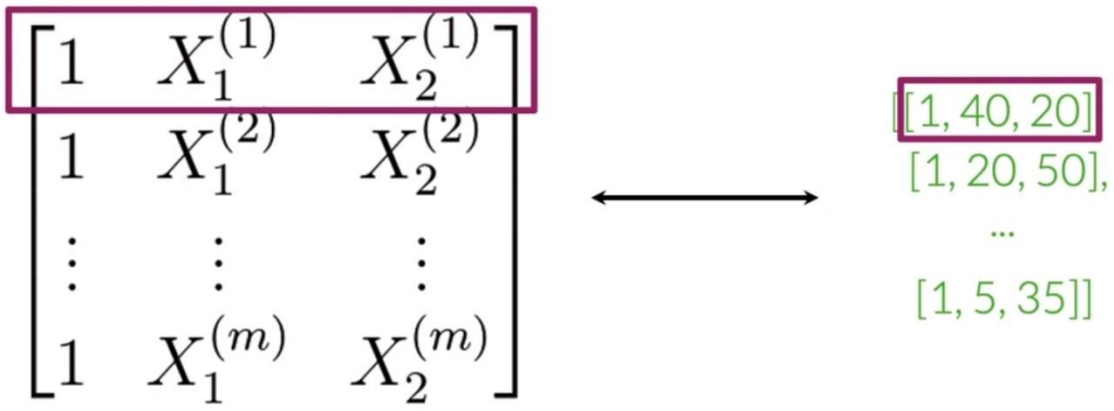
- The next step is to create a matrix X that houses the features of your training examples m. To generate X:
  - **Step 1:** Preprocess your tweet to get a list of words that contain that convey **relevant information** for the **downstream task** (in this case, sentiment classification), using the concepts discussed in the prior section on preprocessing.
  - **Step 2:** Based on the output from step 1, you can obtain the **positive/negative frequencies** of each unique word in the sample using the frequency dictionary.
  - **Step 3:** Obtain a vector with a **bias unit** and two additional features that store the summation of the **positive and negative frequencies of each unique word** in the sample.
- The figure below illustrates the above process of generating X from your input $(\text{Input} \rightarrow X)$:



- Practically, you would have to perform this process on a set of m raw tweets, where m is your dataset size. You would preprocess them one by one to get a list of essential words, one list per tweet. Next, you would be able to extract positive/negative features using a frequencies dictionary mapping.



- This yields a matrix X with m rows and 3 columns, where every row contains the features for a corresponding tweet sample (note that the first row of X is highlighted in the below figure, which corresponds to the first sample, denoted by $X_{(1)}$).



- The concept we've illustrated here involves packing features from multiple training examples together and operating on them simultaneously, rather than one-by-one. This is the concept of "vectorization" (more on this on the section on vectorization).

Putting It All Together in Code

- In code, the overall process of (i) preprocessing, (ii) generating individual features for each tweet sample and, (iii) building matrix X, involves the following steps:
  - **Step 1:** `builds_freqs()` builds the positive/negative frequencies dictionary.
  - **Step 2:** `np.zeros()` initializes feature matrix X with zeros. Set the number of **rows** as **number of tweets** and the number of **columns** as 3 to accommodate 3 features per sample.
  - **Step 3:** `process_tweet()` performs preprocessing on each tweet by (i) deleting stop words, punctuation, task-specific stop words (URLs, twitter handles etc.) and (ii) performing stemming and lower-casing.
  - **Step 4:** `extract_features()` extracts features by summing up the positive and negative frequencies of each unique word within each tweet. The feature set for each tweet sample, comprising of `[1, sum of the positive frequencies for every unique word on tweet, sum of the negative frequencies for every unique word on tweet]` is assigned to the ith row (corresponding to the ith tweet sample) within X.

```
freqs = build_freqs(tweets,labels) #Build frequencies dictionary
X = np.zeros((m,3)) #Initialize matrix X
for i in range(m): #For every tweet
        p_tweet = process_tweet(tweets[i]) #Process tweet
    X[i,:] = extract_features(p_tweet,freqs) #Extract Features
```

m is # of the tweets in the

Logistic Regression As a Classifier

- The extracted feature matrix X generated in the above step is fed into a binary classifier since our task involves predicting a 1 or 0 label corresponding to positive and negative sentiment respectively. In this particular instance, we're using the logistic regression classifier for **binary classification**.

Background

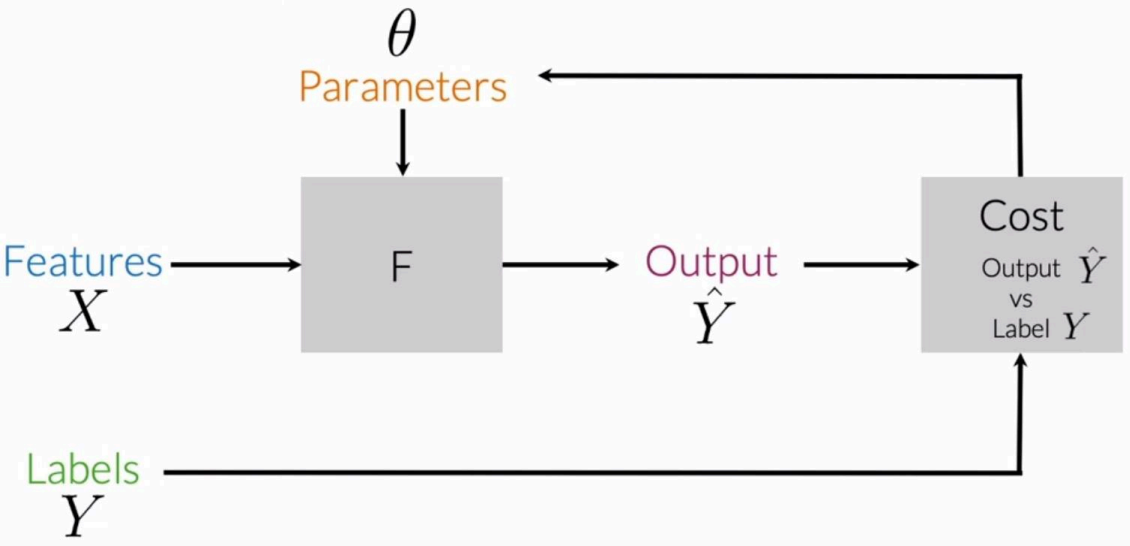
- Logistic regression is a supervised learning algorithm, which implies that the expected outputs, i.e., ground-truth labels Y corresponding to each training sample in X are available during training.
- Logistic regression takes a regular **linear regression** z and applies a **hypothesis function** on top of it (more on the hypothesis function in the section on sigmoid function).
- Formally:

$$h(z)=11+e-zwhere,\ z=\theta_0x_0+\theta_1x_1+\theta_2x_2+\ldots+\theta_Nx_N=\theta_Tx_{(i)}h(z)=11+e-zwhere,\ z=\theta0x0+\theta1x1+\theta2x2+\ldots+\thetaNxN=\thetaTx(i)$$

- Note that in some literature, z is referred to as the '**logit**', which essentially serves as the **input to the hypothesis** function.
- Note that the parameter vector θ is also referred to simply, as "weights". In some contexts, weights are referred to using a w vector or W matrix.

Sigmoid Function

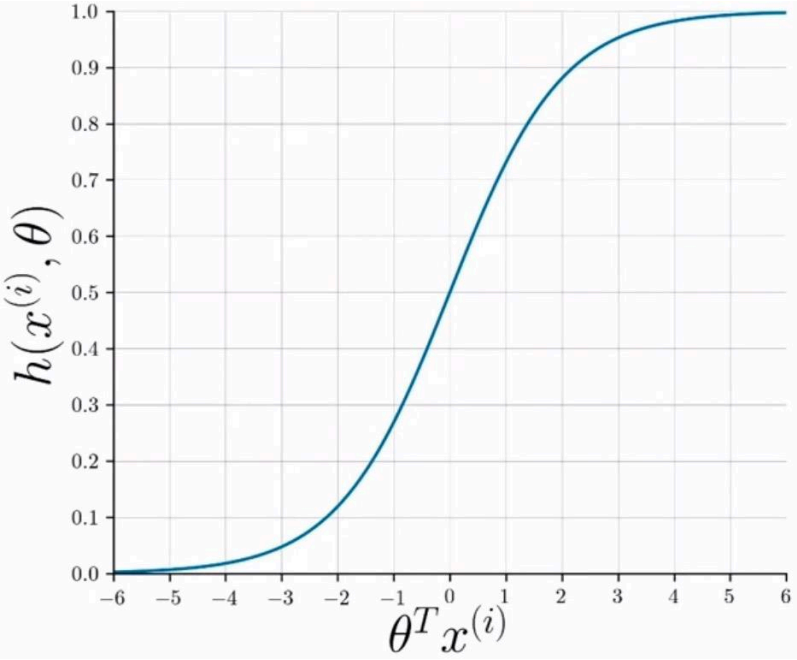
- In the figure below from the section on supervised machine learning, the transformation function $F(\cdot)$ that operates on parameters θ and the feature vector $X_{(i)}$ to yield the output $Y^$, is also known as the **hypothesis** function.



- Specifically, $h(x_{(i)};\theta)$h(x(i);θ) denotes the **hypothesis** function, where ii is used to denote the i_{th}ith observation or data point. In the context of tweets, it's the i_{th}ith tweet in the dataset.
- In logistic regression, the sigmoid function serves as the **hypothesis** function. Logistic regression makes use of the **sigmoid** function which squeezes/collapses/transforms the input into an output range of $[0,1]$[0,1], and can **interpreted as a probability measure**. Note that outside of the context of logistic regression, specifically in neural networks, the sigmoid function serves as a non-linearity and is denoted by $\sigma(\cdot)$σ(·).
- Formally:

$$h(x_{(i)};\theta)=\sigma(x_{(i)};\theta)=11+e_{-\theta Tx_{(i)}}h(x(i);\theta)=\sigma(x(i);\theta)=11+e-\theta Tx(i)$$

- Visually, the sigmoid function is shown below – it approaches 0 as the dot product of $\theta Tx_{(i)}$θTx(i) approaches $-\infty$−∞ and 1 as the dot product approaches ∞∞.

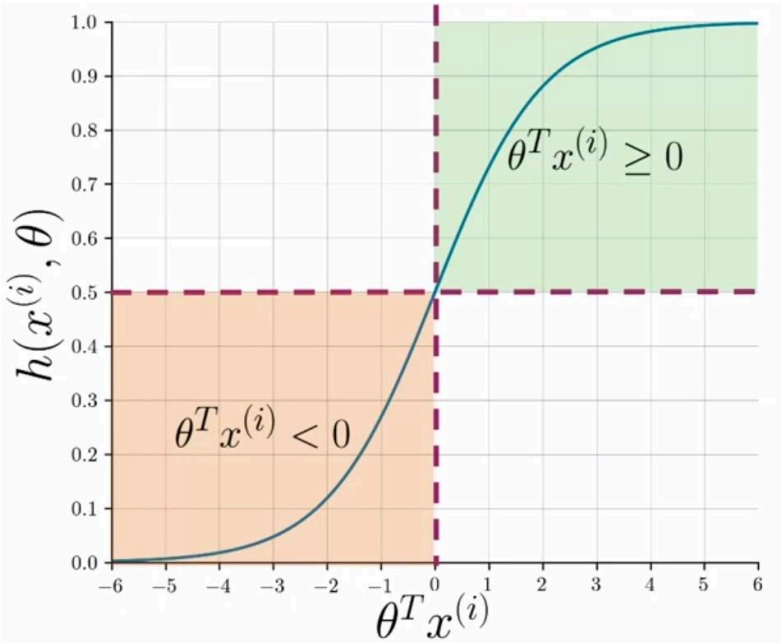


Classification Using Logistic Regression

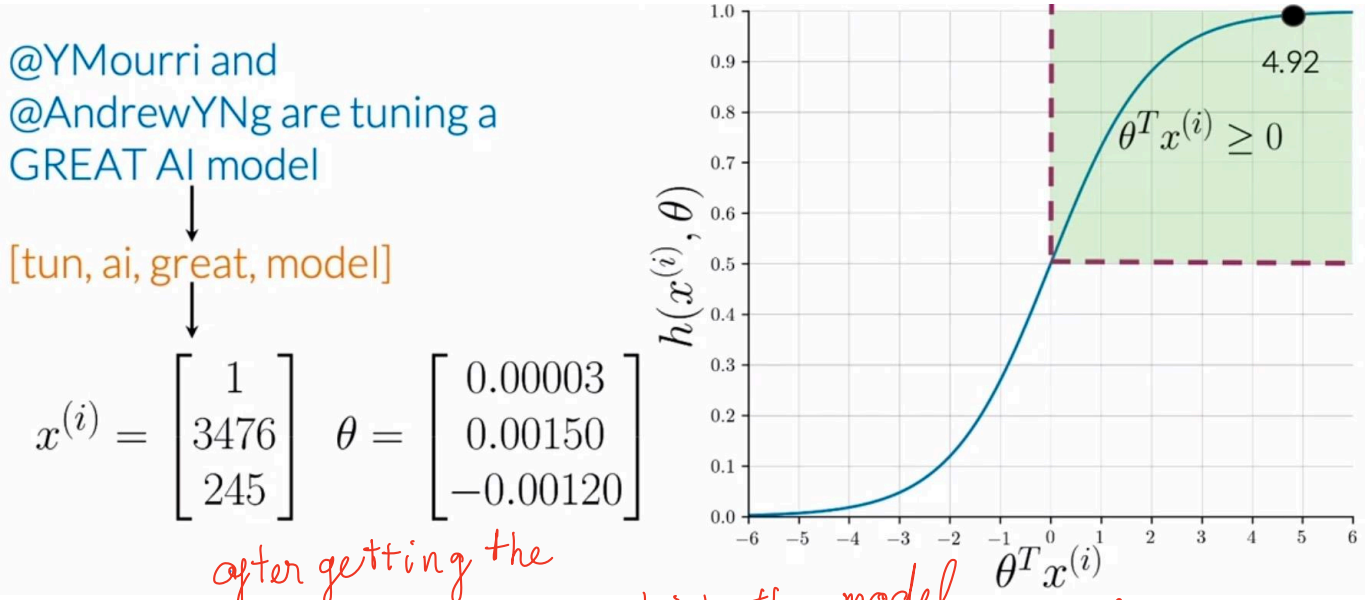
- For classification, a threshold is needed. Usually, it is set to be $0.50.5$ and this value corresponds to a dot product $\theta Tx_{(i)}$θTx(i) equal to 0.

- As shown below, whenever the dot product is ≥≥ 0, the prediction is **positive**, and whenever the dot product is << 0, the prediction is **negative**.
- Formally,

$$\text{classification prediction} = \begin{cases} 1 \text{ if } \theta^T x^{(i)} \geq 0 \\ 0 \text{ if } \theta^T x^{(i)} < 0 \end{cases} \text{classification prediction} = \{0 \text{if} \theta T x(i) < 0 1 \text{if} \theta T x(i) \geq 0$$



- Consider the example shown below in the figure.
  - After preprocessing the tweet and fetching a list of **essential words** (shown in orange), you can **extract features** given a frequencies dictionary and arrive at a vector $x^{(i)}$x(i). This feature vector for the i^{th}ith sample has a bias unit as the first feature, and two additional features that are the sum of positive and negative frequencies of the unique words in your processed tweets.
  - Assuming that you already have an **optimum set of parameters** θθ, your sigmoid function outputs $4.92$4.92 in this case, which leads to the prediction of a **positive sentiment**.



Computing Gradients

- Since computing gradients are the biggest component of the overall training process, we detail the process here.
- To update your weight vector θθ, you will apply gradient descent to iteratively improve your model's predictions.
- The gradient of the cost function J(θ)J(θ) with respect to **one** of the weights θjθj is denoted by ∂J(θ)∂θj∂J(θ)∂θj (or equivalently, ∇θjJ(θ)∇θjJ(θ)) as below. "Partial Gradient of the Cost Function for Logistic Regression" delves into the derivation of this equation.

  $$\partial J(\theta)\partial\theta_j = \nabla_{\theta_j} J(\theta) = 1m\sum_{i=1m}(y\hat{}_{(i)} - y_{(i)})x_j$$ ∂J(θ)∂θj=∇θjJ(θ)=1m∑i=1m(y^(i)−y(i))xj∂J(θ)∂θj=∇θjJ(θ)=1m∑i=1m(y^(i)−y(i))xj

  - where,
    - ii is the index across all mm training examples.
    - jj is the index of the weight θjθj, so xjxj is the feature associated with weight θjθj.
    - y^(i)y^(i) (or equivalently, h(X(i);θ))h(X(i);θ)) is the model's prediction for the iithith sample.

Update Rule

- To update the weight θjθj, we adjust it by subtracting a fraction of the gradient determined by αα:

$$\theta_j := \theta_j - \alpha \times \nabla_{\theta_j} J(\theta) \theta j := \theta j - \alpha \times \nabla \theta j J(\theta)$$

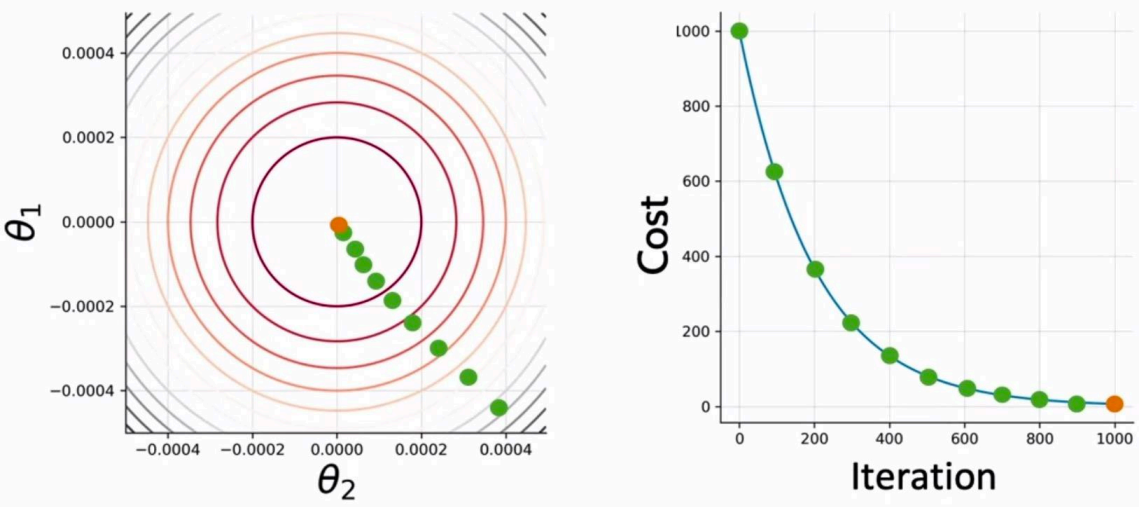
  - where :=:= denotes a paramater/variable update.
- Note that the above weight-update equation is called as the **update rule** in some literature.
- The **learning rate** $\alpha\alpha$, or the **step size**, is a value that helps us choose to control how big of a step a single update during our training process would involve.

Training Logistic Regression

- In the section on classification using logistic regression, we made an assumption that an **optimum set of parameters** $\theta\theta$ were available when inferring our prediction. In this section, we do away with this assumption and generate the optimum set of parameters $\theta\theta$ by putting together the following ideas we talked about in the prior sections:
  - Computing Gradients.
  - Update Rule.
  - Dimension Property of Matrices.
  - Dimension Balancing.
- To train your logistic regression classifier, i.e., learn $\theta\theta$ from scratch, iterate until you find the set of parameters $\theta\theta$ that minimize your cost function until convergence.

*Convergence is defined as the point during training where the overlap between your **prediction** and **ground-truth distributions** is at its peak.*

- To ease the process of visualization, assume that the loss only depends on the parameters $\theta_1\theta 1$ and $\theta_2\theta 2$, the cost function can be represented as a **2D contour plot**, as shown on the left side of the figure. On the right, you can see the **evolution of the cost function** as we iterate.

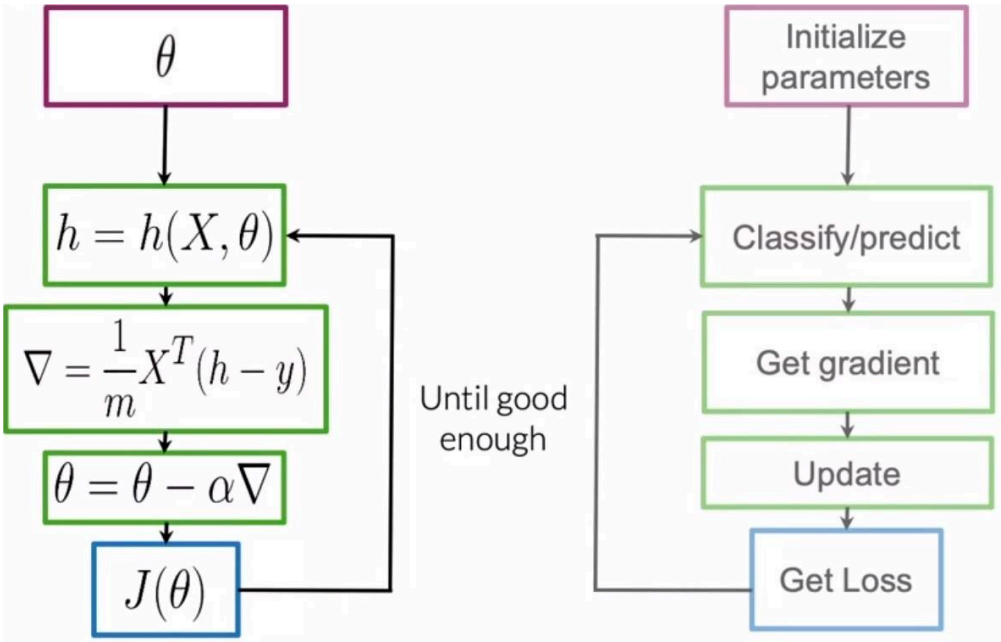


- The following steps outline the process of training:
  - **Initialize parameters**: Initialize the parameter vector $\theta\theta$ randomly or with zeros.
  - **Calculate logits**: The logits zz are calculated using the dot product of the feature vector xx with the weight vector $\theta\theta$, i.e., $z=\theta_T x z=\theta T x$.
  - **Apply a hypothesis function**: Use the sigmoid function $h(\cdot)h(\cdot)$ element-wise to transform each feature of each sample.
  - **Generate the final prediction**: Generate the final prediction $y\hat{}y\hat{}$ by comparing each element obtained at the output of the prior step with a static threshold, of say $0.50.5$ and inferring a 1 if the element $\geq\geq 0$, else 0.
  - **Compute the cost function**: Compute your cost function $J(\theta)J(\theta)$ to determine how far we are from our expected output (note that $J(\theta)J(\theta)$ can be viewed as a measure of our **unhappiness quotient** towards the model's accuracy).
    - The cost function basically tells us if we need more iterations before we converge to a minimized loss (or can be bounded by a "maximum" cap on the number of iterations). Typically, after several iterations, you reach your minima where your cost is at its minimum and you would end your training here.
  - **Calculate gradients**: Calculate the gradients of your cost function w.r.t. each parameter in the parameter vector $\theta\theta$.
  - **Apply update rule**: Update your parameter vector $\theta\theta$ in the direction of the gradient of your cost function using your update rule.
- After a set of iterations, you **traverse along the loss curve** (shown by the green dots in the figure above) and get closer to your **minima** where you

would end your training here when you converge (shown by the orange dot).

*This algorithm is known as **gradient descent**, where the name is derived from the fact that we're using **gradients** to **route over the channels in the parameter-space** that lead to the **loss being minimized (until convergence)**.*

- The figure below illustrates the above process of training via gradient descent:



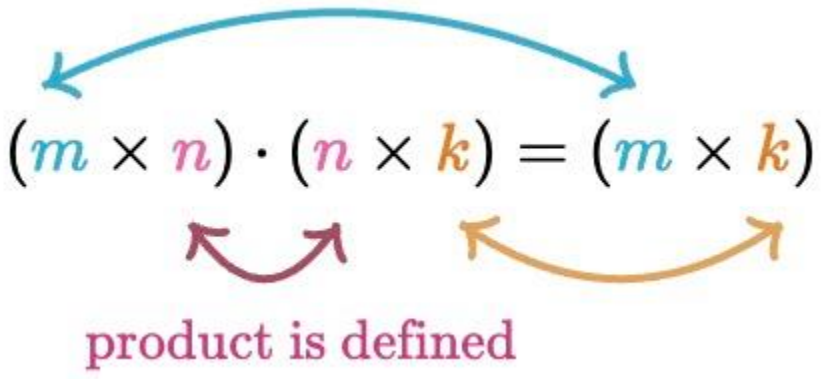
- Models are typically trained for several **epochs**, where each epoch is a **loop over the entire training set**. In other words, an epoch has elapsed when your model has seen **every** example in the training dataset **once**.

Vectorization

- Next, let's explore how we can optimize the process of gradient descent and make it efficient by dealing with all of our training examples in one shot, rather than attending to each sample one-by-one. To this end, a discussion on the dimension property of matrices and the concept of dimension balancing is worthwhile.

Dimension Property of Matrices

- During matrix multiplication, checking for the appropriate matrix dimensions is important to ensure your matrices satisfy the **dimension property** (illustrated in the figure below), which states that:
  - The product of two matrices is defined if and only if the number of **columns in the first matrix** is equal to the **number of rows in the second matrix**.
  - If the product is defined, the resulting matrix will have the **same number of rows as the first matrix** and the **same number of columns as the second matrix**.



Dimension Balancing

- Dimension balancing can be best explained using some practical examples.
- Consider the case of multiplying two vectors of the same type, i.e., row vectors or column vectors. Let's assume that we have two column vectors a and b of size $(m,1)$, we need to **transpose** the vector to the **left** to be able to obtain the **inner-dot product of the vectors** (which is a **scalar**). The case of two row vectors can be handled similarly. Formally:

$$(m,1)^T \cdot (m,1) = (1,m) \cdot (m,1) = (1,1)$$

- **Key takeaways**:
  - In the above example, the center dimension m needs to overlap for both vectors for multiplication to be defined, which follows from the dimension property of matrices that we reviewed earlier. In other words, the vectors need to be **similar-sized** for their dot product to be defined.
  - Note that if the vectors are of different types, i.e., say, one is a row vector while the other is a column vector, the **transposing**

operation (which is part of dimension balancing) is **not needed**. This case can be decomposed into two sub-cases:

- Dot-product of a row vector and a column vector: $(1,m)\cdot(m,1)=(1,1)$(1,m)·(m,1)=(1,1)
  - In this case, again, the center dimension needs to overlap (i.e., the vectors need to be **similar-sized**) for the dot-product to be defined.
- Multiplication of a column vector and a row vector: $(m,1)\cdot(1,n)=(m,n)$(m,1)·(1,n)=(m,n)
  - In this case, given the two entities undergoing the multiplication operation are vectors, the center dimension (which is 1) automatically overlaps. In this case, the vectors **do not** need to be similar-sized. Note that this yields a matrix and not a scalar, and thus represents the **multiplication** of two vectors and **not** their dot-product.

- As another example, let's consider the case of multiplying a vector and a matrix. Assume that we have a matrix AA of size (m,n)(m,n) and vector bb of size $(m,1)$(m,1). In this case, we need to **transpose** XX and place it on the **left** in order to satisfy the dimension property above, and yield a **vector** (of the same "row/column"-vector type as the input vector) at the output. Formally:

$$(m,n)^T\cdot(m,1)=(n,m)\cdot(m,1)=(n,1)$$(m,n)T·(m,1)=(n,m)·(m,1)=(n,1)

Implementing Vectorization

- We saw a preview of vectorization in the section on generating the features matrix. In this section, we offer a deeper treatment on the topic.
- Vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values (vectors/arrays) simultaneously.
  - Under the hood, modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD).
  - For e.g., a CPU with a $512$512 bit register could hold 16 $32$32-bit single precision doubles and do a single calculation $16x$16x faster than executing a single instruction at a time. Combining this with threading and multi-core CPUs leads to orders of magnitude performance gains.
- Overall, the process we laid out in the section on training logistic regression remains similar, with small changes at the relevant steps to accommodate operations on multiple training examples (in our particular case, the entire training dataset of mm examples), rather than attending to individual examples one-by-one.

- To vectorize gradient descent, for each iteration of your training loop, you'll calculate the cost function $J(\theta)$J(θ) using all mm training examples simultaneously (assuming we're using "batch gradient descent" here, other alternatives exist that let you operate on a mini-batch of examples at a time).
- To do so, we'll need to:
  - **Initialize parameters:** Initialize the parameter vector θθ randomly or with zeros.
  - **Vectorize ground-truth labels:** the ground-truth labels yy by stacking them into a column vector.
  - **Matricize training dataset:** Assemble all the samples in our training dataset as a matrix XX, with the i^{th}ith sample at the i^{th}ith row of the matrix.
  - Formally:

$$\theta=\begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \vdots\\ \theta_n\end{bmatrix};y=\begin{bmatrix}y_0\\ y_1\\ y_2\\ \vdots\\ y_n\end{bmatrix}X=\begin{bmatrix}1 & 1 & \dots & 1\\ x_{(1)1} & x_{(2)1} & \dots & x_{(m)1}\\ x_{(1)2} & x_{(2)2} & \dots & x_{(m)2}\\ \vdots & \vdots & \dots & \vdots\\ x_{(1)n} & x_{(2)n} & \dots & x_{(m)n}\end{bmatrix}$$

θ=[θ0θ1θ2⋮θn];y=[y0y1y2⋮yn]X=[11⋯1x(1)1x(2)1⋯x(m)1x(1)2x(2)2⋯x(m)2⋮⋮⋯⋮x(1)nx(2)n⋯x(m)n]θ=[θ0θ1θ2⋮θn];y=[y0y1y2⋮yn]X=[1x1(1)x2(1)1x1(2)x2(2)⋮⋮⋮1x1(m)x2(m)]

- **Calculate logits:** The logits zz are calculated using the dot product of the feature matrix XX with the weight vector θθ, i.e., $z=X\theta$z=Xθ
  - XX has dimensions $(m,n+1)$(m,n+1)
  - θθ has dimensions $(n+1,1)$(n+1,1), where nn is the number of features, and the additional element is for the bias term θ_0θ0 (note that the corresponding feature value x_0x0 is 1).
  - zz has dimensions $(m,1)$(m,1) because $(m,n+1)\cdot(n+1,1)=(m,1)$(m,n+1)·(n+1,1)=(m,1) owing to the dimension property of matrices.
- **Apply a hypothesis function:** Next, we apply the sigmoid/hypothesis function $h(\cdot)$h(·) to each logit zz, i.e., $h(z)=\text{sigmoid}(z)$h(z)=sigmoid(z).
  - $h(X;\theta)$h(X;θ) has the same dimensions as zz, i.e., $(m,1)$(m,1), because the sigmoid operation $h(\cdot)$h(·) is applied element-wise.
- **Generate the final prediction:** Lastly, the output prediction $\hat{y}$y^ is obtained by comparing each element of the resultant vector $h(X;\theta)$h(X;θ), with a static threshold, of say $0.5$0.5 and inferring a 1 if the element $\geq$≥ 0, else 0.
  - $\hat{y}$y^ has the same dimensions as $h(X;\theta)$h(X;θ) (or zz) because the thresholding operation is applied element-wise.
- The output prediction $\hat{y}$y^ can be formally visualized below:

$$\hat{y}=\begin{bmatrix}\hat{y}_0\\ \hat{y}_1\\ \hat{y}_2\\ \vdots\\ \hat{y}_n\end{bmatrix}$$

y^=[y0^y1^y2^⋮yn^]y^=[y0^y1^y2^⋮yn^]

- **Compute the cost function:** The cost function $J(\theta)$J(θ) is calculated by taking the following vector dot products: (i) yy and $\log(\hat{y})$log(y^) and, (ii) $1-y$1−y and $\log(1-\hat{y})$log(1−y^), followed by summing them together and averaging this result over mm training examples.
- Since both yy and $\hat{y}$y^ are column vectors of dimensions $(m,1)$(m,1), we need to **dimension balance** and transpose the vector to the left, so that the multiplication of a row vector with column vector yields the desired dot product, as shown in the equation below:

$$J(\theta)=-\frac{1}{m}\times(y^T\cdot\log(\hat{y})+(1-y)^T\cdot\log(1-\hat{y}))$$J(θ)=−1m×(yT·log(y^)+(1−y)T·log(1−y^))

- Instead of updating a single weight θ_iθi at a time, we can also vectorize the process of updating parameters by updating all the weights in the column vector θθ. We fetch the update rule from the section on update rule, and vectorize it by:
  - Ignoring the jj index corresponding to the j^{th}jth weight in the parameter vector θθ, signifying that we're using the entire parameter vector θθ for computation rather than an individual element in the vector.

$$\theta:=\theta-\alpha\times\nabla_\theta J(\theta)$$θ:=θ−α×∇θJ(θ)

- **Calculate gradients:** To obtain the vectorized gradient of the loss function partial derivative, we use the derivation in "Partial Gradient of the Cost Function for Logistic Regression", and vectorize it by:
  - Ignoring the ii and jj indices corresponding to the i^{th}ith training example and j^{th}jth weight respectively (and thus letting go of the summation in the process).
  - Considering the features matrix XX instead of the feature vector xx.
  - Because the dimensions of XX are (m,n)(m,n) and both $\hat{y}$y^ and yy are $(m,1)$(m,1), we need to transpose XX and place it on the left in order to perform matrix multiplication (again, we're carrying out **dimension balancing**).

$$\frac{\partial J(\theta)}{\partial\theta}=\nabla_\theta J(\theta)=\frac{1}{m}\times(X^T\cdot(\hat{y}-y))$$∂θ∂J(θ)=∇θJ(θ)=1m×(XT·(y^−y))

- **Apply update rule:** Substituting the above equation in the vectorized update rule yields the $(n+1,1)$(n+1,1)-dimensioned θθ we're seeking:

$$\theta:=\theta-\alpha\times(\frac{1}{m}\times(X^T\cdot(\hat{y}-y)))$$θ:=θ−α×(1m×(XT·(y^−y)))

- Rather than attending to samples one by one in a loop, vectorization is the preferred implementation route since it **drastically speeds up the training/inference process** by several orders of magnitude, especially on

modern CPUs and GPUs that offer **enhanced parallelism**. Read more on SIMD/AVX instructions here.

- Vectorization as a concept can extend to any kind of dataset. Similar to how we vectorized the entire set of training samples, we can carry out vectorization of the **testing dataset** as well, rather than feeding individual examples to the model at test time.

Testing Logistic Regression

- To evaluate the **generalization ability** of your classifier and thus, test your model, generate predictions on **unseen data** (data that the model has not come across during **training**) and check how well it fares.
- **Pre-requisites**:
  - X_{val}Xval and Y_{val}Yval: samples and their corresponding labels that were set aside during training (i.e., the validation set).
  - θθ: the set of optimum parameters that lead to a minimized loss, obtained as part of training.
- Armed with $(X_{val}:Y_{val})$(Xval:Yval) and θθ, we carry out the below steps to compute the accuracy of our model:
- **Apply the sigmoid function**:
  - Compute the sigmoid function $h(\cdot)$h(·) for X_{val}Xval with parameter vector θθ.
  - This is formally denoted as $h(X_{val};\theta)$h(Xval;θ), and referred to as hh being a function of X_{val}Xval **parameterized by** θθ.
- **Inferring your predictions using a static threshold**:
  - Evaluate if each value of $h(X_{val};\theta)$h(Xval;θ) is $\geq$≥ to a threshold value, often set to $0.5$0.5.
  - For e.g., let's assume $h(X_{val};\theta)$h(Xval;θ) is $[0.3,0.8,0.5]$[0.3,0.8,0.5], where the i_{th}ith value corresponds to the i_{th}ith sample from your validation set.
    - Next, you're going to assert if each of the components of $h(X_{val};\theta)$h(Xval;θ) is $\geq$≥ to $0.5$0.5.
    - Is $0.3$0.3 $\geq$≥ $0.5$0.5?? No. So our first prediction is equal to 0 (shown in the figure below with the orange underline).
    - Is $0.8$0.8 $\geq$≥ $0.5$0.5?? Yes. So our prediction for the second example is 1.
    - Is $0.5$0.5 $\geq$≥ $0.5$0.5?? Yes. So our third prediction is equal to 1, and so on.
  - At the end, you will have a vector populated with zeros and ones indicating predicted negative and positive examples, respectively.

$$X_{val}\ Y_{val}\ \theta$$

$$h(X_{val}, \theta)$$

$$pred = h(X_{val}, \theta) \geq 0.5$$

$$\begin{bmatrix} 0.3 \\ 0.8 \\ 0.5 \\ \vdots \\ h_m \end{bmatrix} \geq 0.5 = \begin{bmatrix} \underline{0.3 \geq 0.5} \\ 0.8 \geq 0.5 \\ 0.5 \geq 0.5 \\ \vdots \\ pred_m \geq 0.5 \end{bmatrix} = \begin{bmatrix} \underline{0} \\ 1 \\ 1 \\ \vdots \\ pred_m \end{bmatrix}$$

- **Compute the overlap between predictions and the ground-truth:**
  - Compare the predictions with the ground-truth from your validation data. Generate booleans of 1 and 0 corresponding to whether your predictions are match your ground-truth or not, respectively.
  - For e.g., if your prediction was correct, say in this case where both your prediction and your label are equal to 0, your vector will have a value equal to 1 in the first position (shown on the right-hand-side in the below figure with a green underline).
  - Conversely, if your second prediction wasn't correct, because your prediction and label disagree, your vector will have a value of 0 in the second position and so forth.

$$X_{val}\ Y_{val}\ \theta$$

$$h(X_{val}, \theta)$$

$$pred = h(X_{val}, \theta) \geq 0.5$$

$$\sum_{i=1}^{m} \frac{(pred^{(i)} == y_{val}^{(i)})}{m}$$

$$\begin{bmatrix} \underline{0} \\ 1 \\ 1 \\ \vdots \\ pred_m \end{bmatrix} == \begin{bmatrix} \underline{0} \\ 0 \\ 1 \\ \vdots \\ Y_{val_m} \end{bmatrix}$$

$$\begin{bmatrix} \underline{1} \\ 0 \\ 1 \\ \vdots \\ pred_m == Y_{val_m} \end{bmatrix}$$

- **Compute the model's accuracy on the validation set:**
  - To obtain the accuracy of your model:
    - Calculate the number of times that your predictions were correct by summing up the vector of the comparisons (since each correct prediction is essentially represented as a 1 in this vector).
    - Divide this number by the total number of samples/observations in your validation set, say mm, as shown on the left-hand-side in the above figure.

- This metric gives an estimate of how your logistic regression model will fare on unseen data.
  - Thus, if your accuracy is equal to $0.50.5$, it means that $50\%50\%$ of the time, your model is expected to do the right thing.
- For e.g., assume your $Y_{val}Yval$ and prediction vectors for 5 observations are as in the figure below.
  - Upon comparing each of their values, you'll have the following vector with a single 0 in the third position where the prediction and the label disagree.
  - Next, you sum the number of times that your predictions were right and divide that number by the total number of observations in your validation set. This yields an accuracy of $80\%80\%$.

$$Y_{val} = \begin{bmatrix} 0 \\ 1 \\ \underline{1} \\ 0 \\ 1 \end{bmatrix} \quad pred = \begin{bmatrix} 0 \\ 1 \\ \underline{0} \\ 0 \\ 1 \end{bmatrix} \quad (Y_{val} == pred) = \begin{bmatrix} 1 \\ 1 \\ \underline{0} \\ 1 \\ 1 \end{bmatrix}$$

$$\text{accuracy} = \frac{4}{5} = 0.8$$

Cost Function Intuition

- Let's dissect the cost function for logistic regression to build the intuition behind it.
- The cost function used for logistic regression is the **average of the log loss** across all training examples. Formally:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

- where,
  - m is the number of training examples.
  - $y(i)y(i)$ is the actual/ground-truth label of the $ithith$ training example.

- $h(z(\theta)^{(i)})$h(z(θ)(i)) (or equivalently, $h(h(x^{(i)});\theta)$h(h(x(i));θ), or simply, $\hat{y}^{(i)}$y^(i)) is the model's prediction for the i^{th}ith training example.

- Let's break it down into its components. Look at the **left-hand side** of the equation where you find a sum over the variable mm, which is the number of samples in your training set. This indicates that you're going to **sum over the cost** of each training example. Also, out front, the $-\frac{1}{m}$−1m, indicates that when combined with the sum, this would lead to an **averaging operation** across all training examples.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

- Inside the square brackets, the equation has two terms that are added together, one on the **left-hand side** and the other on the **right-hand side**. To consider how each of these terms contribute to the cost function for each training example, let's dive into each of them separately.

Left-hand Side Term

- The term on the left is the product of $y^{(i)}$y(i), which is the **label for each training example** and the **log of the prediction** $\log(h(x^{(i)});\theta)$log(h(x(i));θ) (or equivalently, $\log(\hat{y})$log(y^)), which is the logistic regression function being applied to each training example.
- Consider the case when your label is 0. In this case, the function $h(\cdot)$h(·) can return any value, and the entire term will be 0 because 0 times anything is just 0.
- Next, consider the case when your label is 1, and…
  - If your prediction is close to (or "tends" to) 1, then the log of your prediction will tend to 0, because $\log 1$log1 = 0, and thus the product will also tend to 0.
  - If your prediction tends to 0, then this term blows up and approaches $-\infty$−∞, which is then multiplied by the overall factor of -1−1 to convert it to $+\infty$+∞.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

$y^{(i)}$	$h(x^{(i)},\theta)$	
0	any	0
1	0.99	~0
1	~0	-inf

Right-hand Side Term

- Now, consider the term on the right-hand side of the cost function equation. In this case, if your label is 1, then the $1-y_{(i)}$1−y(i) term goes to 0. And so any value returned by the logistic regression function will result in 0 for the entire term, because again, 0 times anything is just 0.
- If your label is 0, and…
  - If your prediction tends to 0, then the products in this term will again tend to 0.
  - If your prediction tends to 1, then the log term will blow up and the overall term approaches $-\infty$−∞, which is then multiplied by the overall factor of -1−1 to convert it to $+\infty$+∞.
    - The closer the model prediction gets to 1, the larger the loss. If the model's prediction is $0.9999$0.9999, the loss becomes $-1\times(1-0)\times\log(1-0.9999)\approx 9.2$−1×(1−0)×log(1−0.9999)≈9.2

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

$y^{(i)}$	$h(x^{(i)},\theta)$	
1	any	0
0	0.01	~0
0	~1	-Inf

- **Key takeaways**:

- There is **only one term** in the cost function that is **relevant** when your label is 0 which is $-\log(\hat{y}$−log(y^), and another that is **relevant** when the label is 1 which is $-\log(1-\hat{y}$−log(1−y^).
- Intuitively, when your prediction is **close** to the label value, the loss is **small**, and conversely when your label and prediction **disagree**, the overall cost **goes up**.

Overall Minus Sign

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))\right]$$

- In each of the left/right-hand side terms, you're taking the log of a value between 0 and 1, which will always return a **negative** number. So, the minus sign out-front ensures that the overall cost will always be a **positive** number.

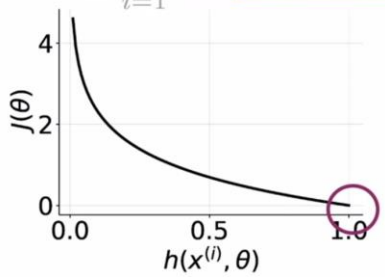
Geometric Interpretation

- Let's geometrically interpret what the cost function looks like for each of the labels, i.e., overall possible prediction values, 0 and 1.
- First, we're going to look at the loss when the label is 1.
  - In the plot shown below, you have your prediction value on the horizontal axis and the cost associated with a single training example on the vertical axis.
  - For $y=1$y=1, $J(\theta)$J(θ) simplifies to just $-\log(\hat{y})$−log(y^).
  - When the prediction tends to 1, the loss is close to 0, because your prediction **agrees** well with the label.
  - When the prediction tends to 0, the loss approaches ∞∞, because your prediction and the label **disagree strongly**.
  - Formally,

$$\text{if } y = 1 \Longrightarrow J(\theta) = L(\hat{y},1) = -\log(\hat{y}) = \begin{cases} 0 & \text{if } \hat{y} = 1 \\ \infty & \text{if } \hat{y} = 0 \end{cases}$$

if y = 1⟹J(θ)=L(y^,1)=−log(y^)={0 if y^ = 1∞ if y^ = 0if y =
1⟹J(θ)=L(y^,1)=−log(y^)={∞ if y^ = 00 if y^ = 1

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$



- Second, let's explore the loss when the label is 0.
  - In this case, the opposite is true when the label is 0, compared to the case above when the label is 1.
  - For $y=0$y=0, $J(\theta)$J(θ) reduces to just $-\log(1-\hat{y})$−log⁡(1−y^).
  - When your prediction tends to 0, the loss is also close to 0.
  - When your prediction tends to 1, the loss approaches ∞∞.
  - Formally,

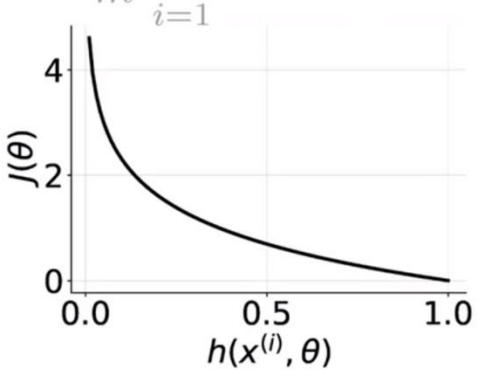
$$\text{if } y = 0 \Longrightarrow J(\theta) = L(\hat{y}, 0) = -\log(1-\hat{y}) = \begin{cases} 0 & \text{if } \hat{y} = 0 \\ \infty & \text{if } \hat{y} = 1 \end{cases}$$

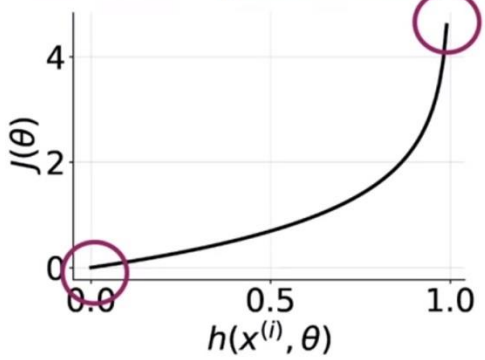
if y = 0⟹J(θ)=L(y^,0)=−log⁡(1−y^)={0 if y^ = 0∞ if y^ = 1

$$\text{if } y = 0 \Longrightarrow J(\theta) = L(\hat{y}, 0) = -\log(1-\hat{y}) = \begin{cases} \infty & \text{if } \hat{y} = 1 \\ 0 & \text{if } \hat{y} = 0 \end{cases}$$

if y = 0⟹J(θ)=L(y^,0)=−log⁡(1−y^)={∞ if y^ = 10 if y^ = 0

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$





Mathematical Interpretation

- Now, let's present a formal mathematical interpretation of what the cost function looks like for each of the labels.
- Consider the case when the label is 1. During optimization, since we're trying to minimize the loss $J(\theta)$J(θ), we want $\hat{y}$y^ to be as large as possible, which in turn pushes $\hat{y}$y^ towards 1 (which is the **correct** prediction).

- Next, consider the case when the label is 0. Again, during optimization, since we're trying to minimize the loss $J(\theta)$J(θ), we want $1-\hat{y}$1−y^ to be as large as possible, which in turn pushes $\hat{y}$y^ to be as small as possible, towards 0 (again, which is the **correct** prediction).

Further Reading

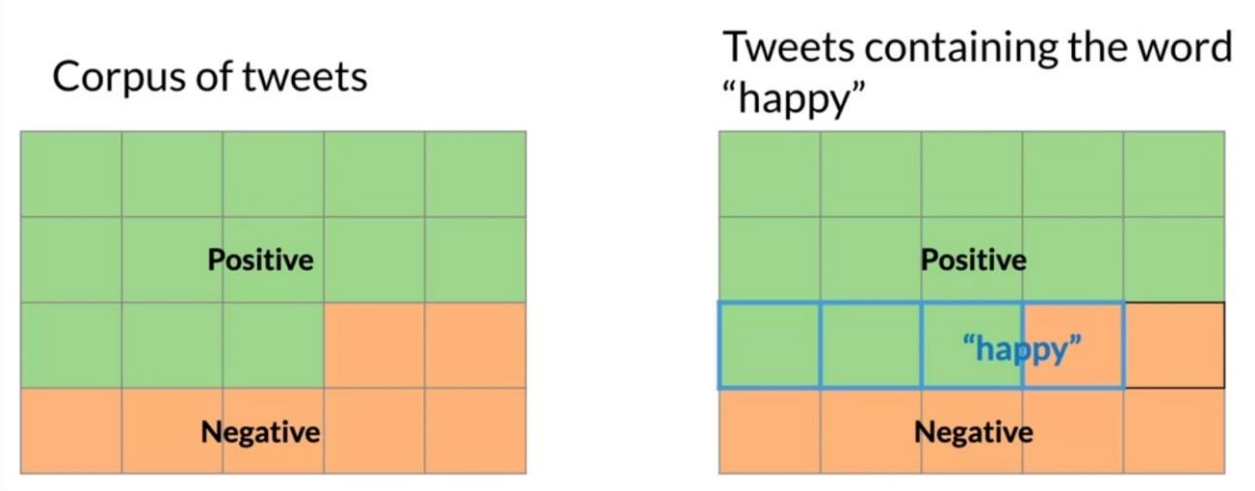
Here are some (optional) links you may find interesting for further reading:

- Partial Gradient of the Cost Function for Logistic Regression delves into the derivation of the partial derivative of the cost function for logistic regression.
- Dimension property to read more on matrix multiplication properties.
- Vectorization to understand the why and what of vectorization.
- Gradient Descent to explore the different types of gradient descent and their pros/cons.

Week 2: Sentiment Analysis Using Naive Bayes

Probability: Basics

- Imagine you have a corpus of 2020 tweets that can be categorized as having either **positive** or **negative** sentiment, but **not both**.
- Within that corpus, the word *happy* is sometimes being labeled positive and in some cases, negative.
  - This implies that there exist negative tweets that contain the word *happy*.
  - Conversely, (and obviously) there exist other words apart from *happy* in tweets labeled positive.
  - Shown below is a graphical representation of this "overlap". Let's explore how we may handle this case using probability, with a Venn diagram-based graphical interpretation.



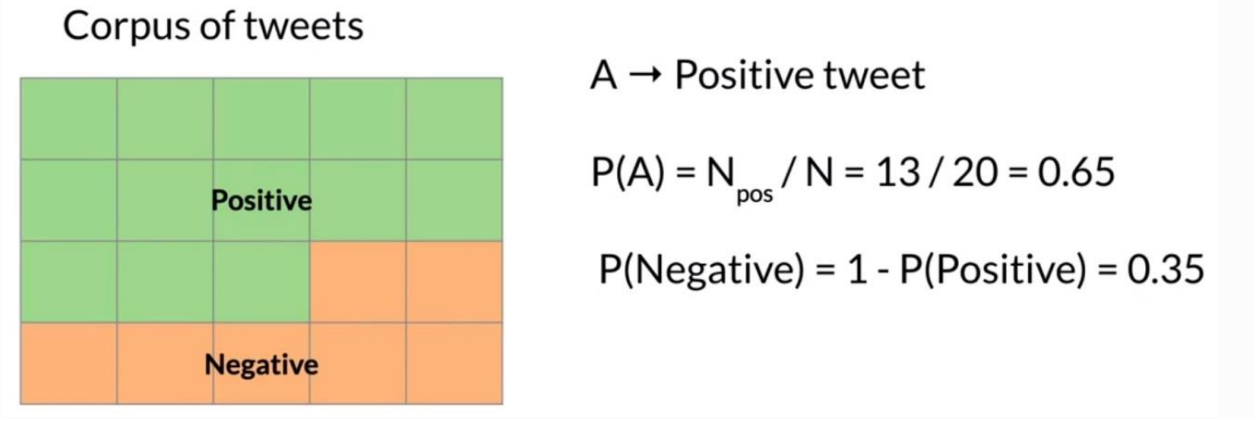
- One way to think about probabilities is by counting how frequently events occur. Let's define event AA as a tweet being labeled positive. The probability of event AA, denoted by P(A)P(A), is calculated as:

P(A)=number of positive tweets in the corpustotal number of tweets in the corpusP(A)=number of positive tweets in the corpustotal number of tweets in the corpus

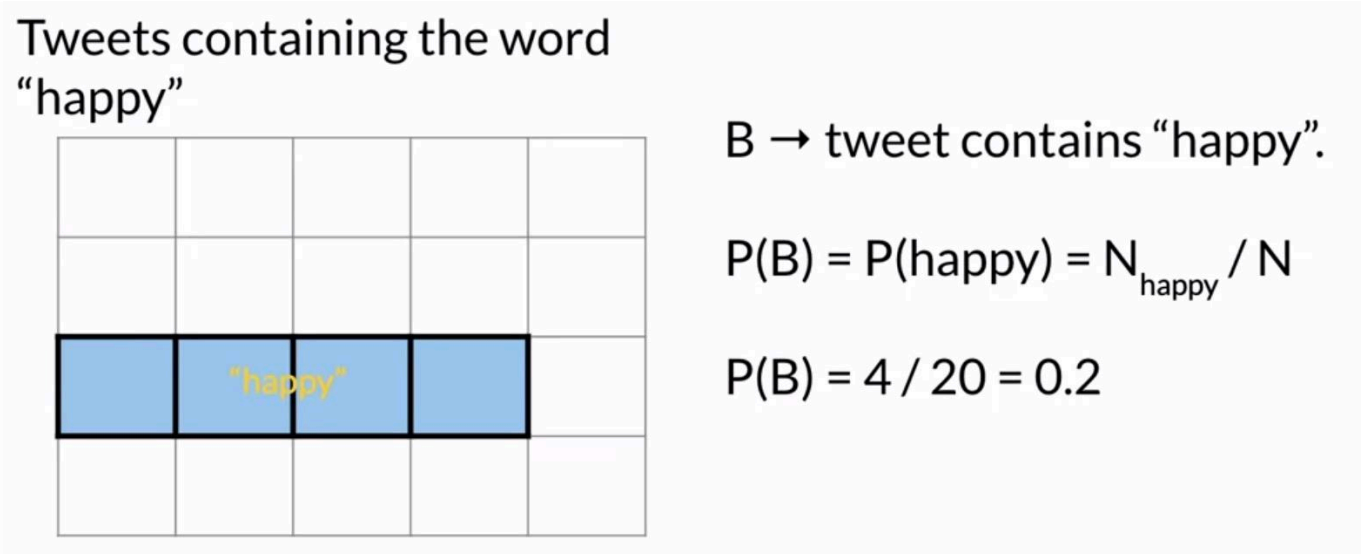
- In the example shown in the figure below, P(A)=1320=0.65P(A)=1320=0.65, or 65%65%. It's worth noting the complementary probability here, which is:

P(tweet expressing a negative sentiment)=1−P(tweet expressing a positive sentiment)P(tweet expressing a negative sentiment)=1−P(tweet expressing a positive sentiment)

- Note that for this to be true, all tweets must be categorized as either positive or negative but **not both**.



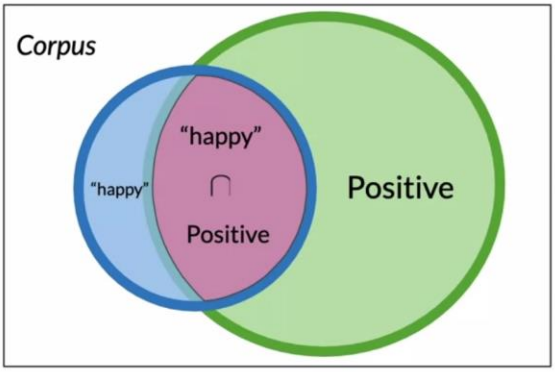
- Next, let's define event BB in a similar way by counting tweets containing the word *happy*.
- In the example shown in the figure below, the total number of tweets containing the word *happy* N_{happy}Nhappy is 4. The probability of event $P(B)$P(B) is thus $\frac{4}{20}=0.2$420=0.2, or 20%20%.



- From the diagram on the left-hand-side of the figure below, we can infer that there are 3 tweets that are labeled positive **and** contain the word *happy* (corresponding to the 3 overlapping cells that are colored pink).

- Another way of looking at this is shown in the Venn diagram in the right-hand-side of the figure below. In particular, notice the **intersection** of (i) the set of tweets which are labeled positive and, (ii) the set of tweets which contain the word *happy*.

$$P(A \cap B) = P(A, B) = \frac{3}{20} = 0.15$$

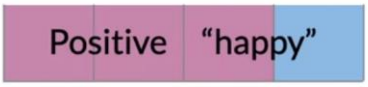


- Recall that the corpus contains 2020 tweets overall. In the context of the Venn diagram above, the associated probability is:

P(tweet is labeled positive and contains "happy")=P(intersection)P(entirecorpus)=P("happy" ∩ positive)P("happy" ∪ positive)=320=0.15 or 15%.P(tweet is labeled positive and contains "happy")=P(intersection)P(entirecorpus)=P("happy" ∩ positive)P("happy" ∪ positive)=320=0.15 or 15%.

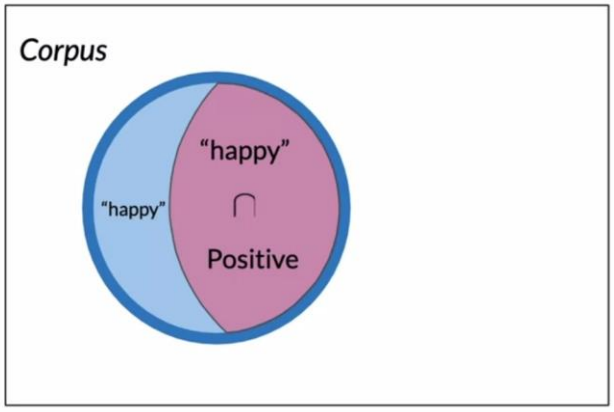
Conditional Probability

- The goal of this section is to derive Bayes' rule from the concept of conditional probabilities.
- Consider the tweets that contain the word *happy*, instead of the entire corpus. In other words, from our Venn formulation in the right-hand-side of the diagram below, you would be only considering the tweets inside the **blue circle** where many of the other positive tweets (that don't contain the word *happy*) are now excluded. Note that the purple area in the figure below denotes the probability that a positive tweet contains the word *happy*.



P(A | B) = P(Positive | “happy”)

P(A | B) = 3 / 4 = 0.75

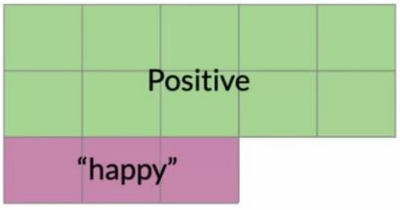


- In this scenario, the probability that a tweet is positive given that it contains the word *happy*, denoted by P(positive | “happy”)P(positive | “happy”), simply becomes:

P(positive | "happy")=number of positive tweets and contain "happy"number of tweets that contain "happy"P(positive | "happy")=number of positive tweets and contain "happy"number of tweets that contain "happy"

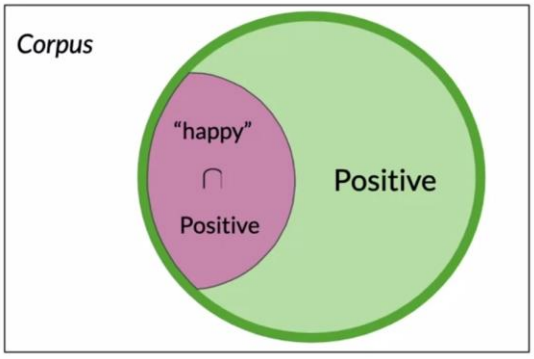
=P(positive ∩ happy)P("happy")=P(positive ∩ happy)P("happy")

- For the particular example in the above figure, P(positive | “happy”)=34=0.75 or 75%P(positive | “happy”)=34=0.75 or 75%.
  - Thus, a tweet has a 75% likelihood of being positive if it’s contains the word happy.
- Now, keeping the **overall** Venn diagram from our previous section in mind, think about the inverted case where you’re trying to fiure out the probability of whether a tweet contains the word *happy* given the tweet is labelled positive. Again, the purple area in the figure below denotes the probability that a **positive tweet contains the word *happy***.



P(B | A) = P(“happy”| Positive)

P(B | A) = 3 / 13 = 0.231

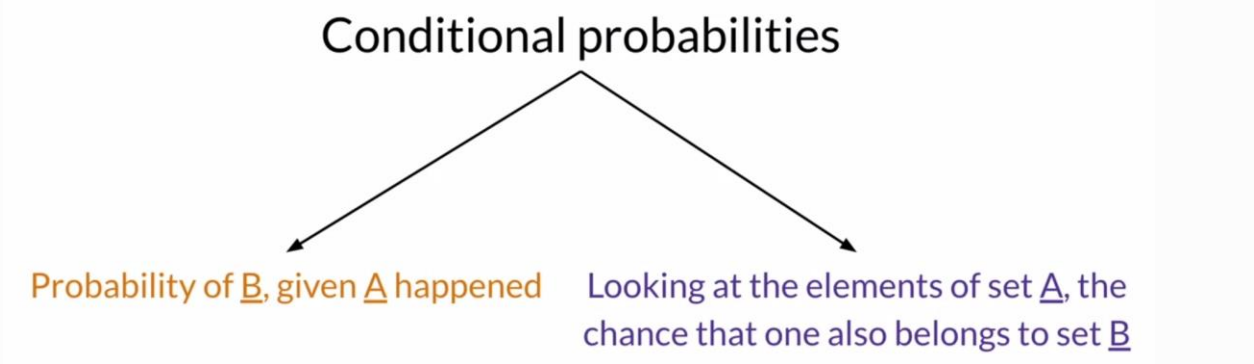


- Formally, the probability P(“happy” | positive)P(“happy” | positive) is:

P("happy" | positive)=number of positive tweets that contain "happy"number of tweets that are positiveP("happy" | positive)=number of positive tweets that contain "happy"number of tweets that are positive

=P("happy" ∩ positive)P(positive)=P("happy" ∩ positive)P(positive)

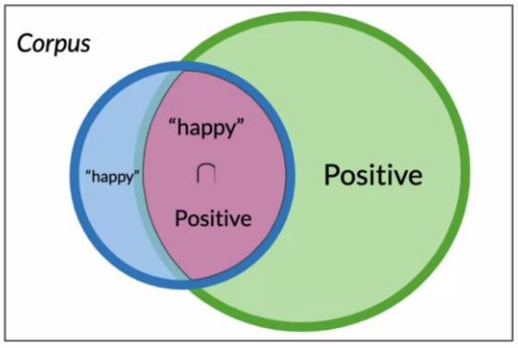
- For the particular example in the above figure, P(“happy” | positive)=313=0.23 or 23%P(“happy” | positive)=313=0.23 or 23%.
- With all of this discussion of the probability of meeting certain conditions, we are essentially talking about **conditional probabilities**. Conditional probabilities can be interpreted as the **probability of an outcome** BB **knowing that event** AA **has already happened**. In other words, given that we’re looking at an element from set AA, the probability that it also belongs to set BB.



Bayes’ Rule

- A graphical way to interpret this is using the the Venn diagram shown below. Using the previous example as reference, the probability of a tweet being positive given that it has the word *happy* is:

P(positive | "happy")=P(positive ∩ "happy")P("happy")P(positive | "happy")=P(positive ∩ "happy")P("happy")



$$P(\text{Positive}|\text{“happy”}) = \frac{P(\text{Positive} \cap \text{“happy”})}{P(\text{“happy”})}$$

Naives Bayes rule:

$$P(\text{Positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{Positive}) \times \frac{P(\text{Positive})}{P(\text{"happy"})}$$

- Let's take a closer look at the equation from the previous slide. You could write a similar equation by simply **swapping** the position of the two conditions. This gives you the conditional probability of a tweet containing the word *happy* given that it is a positive tweet as:

$$P(\text{"happy"} \mid \text{positive}) = P(\text{"happy"} \cap \text{positive}) P(\text{positive}) P(\text{"happy"} \mid \text{positive}) = P(\text{"happy"} \cap \text{positive}) P(\text{positive})$$

- Armed with both of these equations, you're now ready to derive Bayes' rule. To combine these equations, note that the intersections "happy" ∩ positive"happy" ∩ positive and positive ∩ "happy"positive ∩ "happy" represent the same quantity, no matter which way it's written.
- Think about the probability of a tweet being positive given that it contains the word *happy* P(positive | "happy")P(positive | "happy") in terms of the probability of a tweet containing the word *happy* given that it is positive P("happy" | positive)P("happy" | positive), with a little algebraic manipulation:

$$P(\text{positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{positive}) \times P(\text{positive}) P(\text{"happy"}) P(\text{positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{positive}) \times P(\text{positive}) P(\text{"happy"})$$

- This is now an **expression of Bayes' rule** in the context of our sentiment analysis problem.
- More generally, Bayes' rule states that:

$$P(X|Y) = P(Y|X) \times P(X) P(Y) P(X|Y) = P(Y|X) \times P(X) P(Y)$$

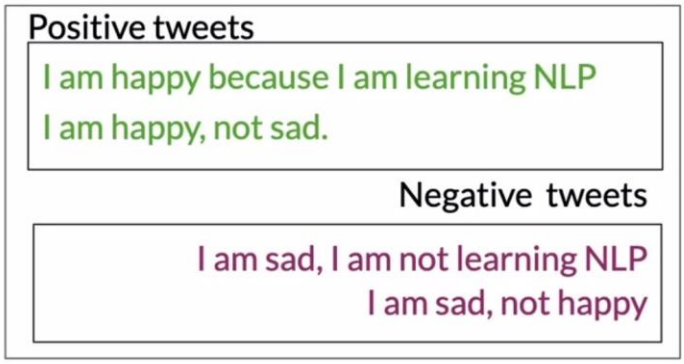
- This is the basic formulation of Bayes' rule using expressions of conditional probability. With Bayes' rule, you can calculate the probability of XX given YY P(X|Y)P(X|Y) if you already know the probability of YY given XX P(Y|X)P(Y|X), and the ratio of the probabilities of XX and YY.
- As an example, suppose that in your dataset, 25%25% of the positive tweets contain the word 'happy', 13%13% of the tweets in your dataset contain the word *happy*, and 40%40% of the total number of tweets are positive. You observe the tweet: `'happy to learn NLP'`. The probability that this tweet is positive is:

$$P(\text{positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{positive}) \times P(\text{positive}) P(\text{"happy"}) = \frac{25}{100} \times \frac{40}{13} = 0.77 \text{ or } 77\%. P(\text{positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{positive}) \times P(\text{positive}) P(\text{"happy"}) = \frac{25}{100} \times \frac{40}{13} = 0.77 \text{ or } 77\%.$$

Computing the Conditional Probabilities Table

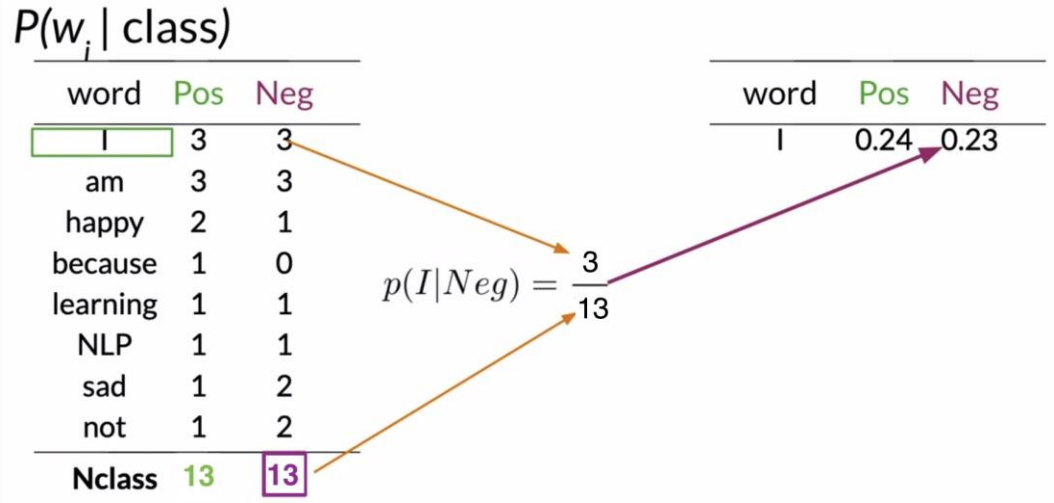
- Armed with two corpora, one with **positive tweets** and one with **negative tweets**, let's look into the steps needed to generate our conditional probabilities table.
- **Generate positive/negative frequency features:** Extract the vocabulary (i.e., the **unique words**) that appear in **both** your positive and negative corpora along with their counts. In other words, we're seeking to obtain the word-counts for each

occurrence of a word in both the positive corpus and negative corpus, just like we did before with logistic regression.
    - Note that the steps so far are common with what we've seen earlier with logistic regression.
- **Fetch total counts:** Get a total count of all the words in your positive corpus and negative corpus. That is, sum over the rows of the table below on the right-hand-side of the below figure. In this particular instance, for both positive and negative tweets, our total counts are 13 words.



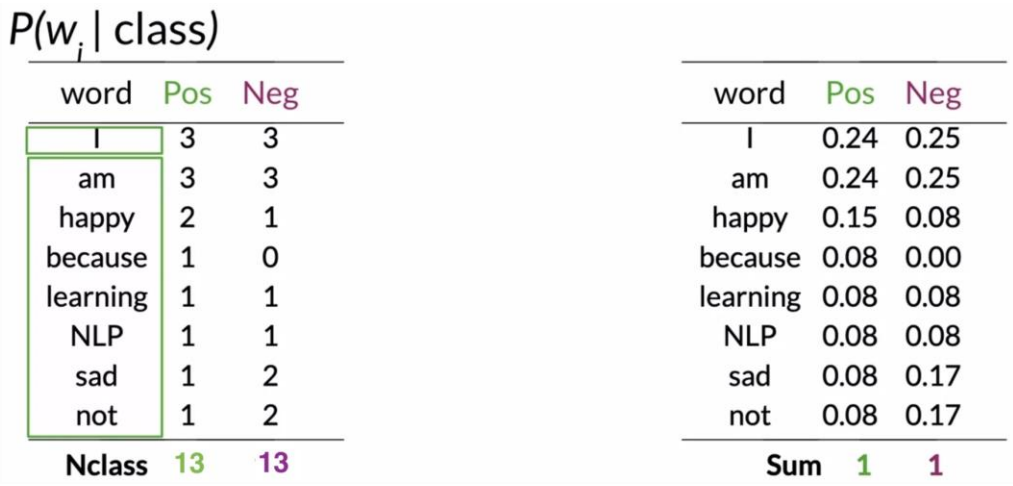
word	Pos	Neg
I	3	3
am	3	3
happy	2	1
because	1	0
learning	1	1
NLP	1	1
sad	1	2
not	1	2
N_{class}	13	13

- **Compute conditional probabilities given a class:** Next, as the first **new** step for Naive Bayes, and it's very important because it allows you to compute the conditional probabilities of each word given a class.
    - This is obtained by dividing the frequency of each word in a class by its corresponding sum of words in the class.
    - So for the word *I*, the conditional probability for the positive class would be $313=0.24313=0.24$. Store this value in a new table that contains the **conditional probabilities of each word in your tweet**, as shown in the figure below.
    - Similarly, for the word *I*, in the negative class, you get a conditional probability of $313=0.23313=0.23$. Store that in your conditional probabilities table as well.
    - Apply the same procedure for each word in your vocabulary to complete the conditional probabilities table.



These conditional probabilities basically represent how often the (unique) words in a tweet occur in positive and negative tweet samples.

- **Sanity check your conditional probabilities:** A key property of the conditional probabilities table is that if you sum over all the probabilities for each class, you'll get 1, since $\sum_i P_i = 1$ $\sum_i P_i = 1$.



- Let's investigate this table further to see what these numbers mean.

word	Pos	Neg
I	0.24	0.25
am	0.24	0.25
happy	0.15	0.08
because	0.08	0
learning	0.08	0.08
NLP	0.08	0.08
sad	0.08	0.17
not	0.08	0.17

- words that are equally probable don't add anything to the sentiment (I, am, because, learning)
- words with significant diff between probabilities sentiment

- First, note the words have a nearly identical conditional probability, viz., *I*, *am*, *learning*, and *NLP*. The interesting thing here is words that are **equally probable don't add much to the sentiment**.
- In contrast to these neutral words, look at some of these other words like *happy, sad,* and *not*. They have a significant difference between probabilities. These are your **power words** tending to express one sentiment or the other. These words carry a lot of weight in determining your tweet sentiments.
- Now let's take a look at *because*. Since, it only appears in the positive corpus, its conditional probability for the negative class is 0. When either of the probabilities (corresponding to the positive/negative classes) turn out to be 0, you have **no way of comparing between the two corpora**.
  - To avoid this, you'll need to smooth out your probability function, as described in the next section on Laplacian smoothing.

Laplacian Smoothing

→ just a way of not getting a 0

- For a particular word, the probability of a class being 0 leads to an issue for our **downstream calculations**, as follows:
  - Consider the case where the negative-class probability for a particular word is 0. Because the negative-class probability resides in the denominator in our Naive Bayes inference equation, this leads to a divide-by-0.
  - Similarly, if the positive-class probability for a particular word is 0, this will lead to the overall Naive Bayes inference expression zeroing out because the positive-class probability resides in the numerator.
- Let's dive into **Laplacian Smoothing**, a technique you can use to **avoid your probabilities being 0**.
- The expression used to calculate the conditional probability of a word $w_i$wi, given a class, is the frequency of the word in the corpus $P(w_i|class)$P(wi|class) is given by $freq(w_i,class)N_{class}$freq(wi,class)Nclass. Smoothing the probability function means that

you will use a slightly different formula from the original, as shown in the figure below.

$$P(w_i|class) = \frac{freq(w_i, class)}{N_{class}} \qquad class \in \{Positive, Negative\}$$

$$P(w_i|class) = \frac{freq(w_i, class) + 1}{N_{class} + V}$$

N_{class} = frequency of all words in class

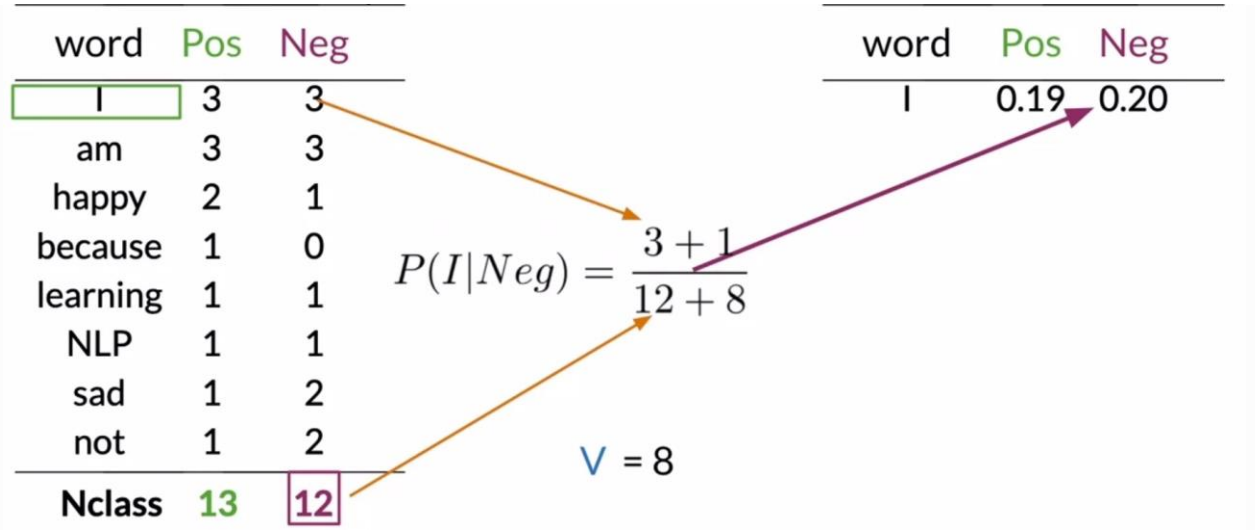
V = number of unique words in vocabulary

- Thus, to compute the positive and negative probability for a specific word w_i in the vocabulary, we have:

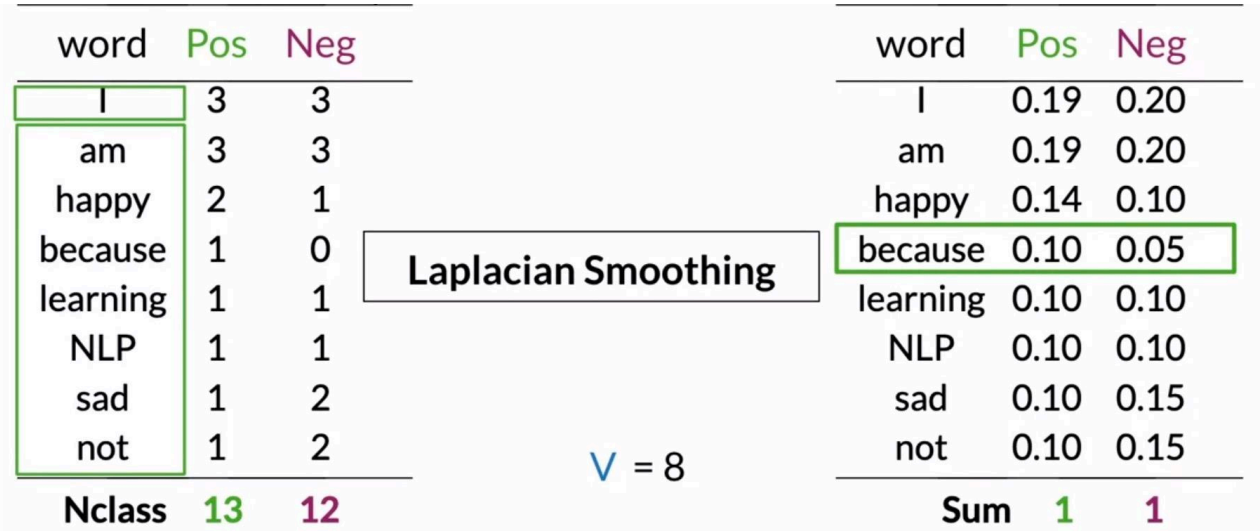
$$P(W_{pos}) = \frac{freq(w_i, pos) + 1}{N_{pos} + V} \quad P(W_{neg}) = \frac{freq(w_i, neg) + 1}{N_{neg} + V}$$

P(Wpos)=freq(wi,pos)+1Npos+VP(Wneg)=freq(wi,neg)+1Nneg+VP(Wpos)=freq(wi,pos)+1Npos+VP(Wneg)=freq(wi,neg)+1Nneg+V

- where:
  - $freq(w_i, pos)$ and $freq(w_i, neg)$ are the frequencies of word w_i in the positive and negative class respectively. In other words, the positive frequency of a word is the number of times the word is counted with the label of 1.
  - N_{pos} and N_{neg} are the total number of positive and negative words for all documents/tweets respectively.
  - V is the number of unique words in the entire dataset that encompasses **all classes**, positive and negative.
- Note that we've **added a 1** to the numerator for additive smoothing, which is another name for Laplacian smoothing. This little transformation avoids the probability being 0. However, it adds a new term to all the frequencies that is not correctly normalized by N_{class}. To account for this, we'll need to add a new term V to the denominator In other words, adding V in the denominator **helps account for the extra** $+1$ added to the numerator.
- Recall that your vocabulary V is the set of **unique words in your corpora consisting of both the positive and negative tweets** (i.e., the entire vocabulary and not just the unique words in a single class). So now all the probabilities in each column will sum to 1. This process is called Laplacian smoothing.
- Let's consider an example below. Note how the probabilities are computed for both the positive and negative columns. Since the size of our vocabulary V is 8, we use $V=8$ for **both the positive and negative cases** in the denominator.



- **Step 1:** Calculate is the number of unique words in your vocabulary V. In this particular e.g., you have 8 unique words.
- **Step 2:** Calculate the probability for each word in the positive and negative class.
  - Consider the word *I*:
    - For the positive class, you get $(3+1)/(13+8)=0.19$(3+1)/(13+8)=0.19.
    - For the negative class, you have $(3+1)/(12+8)=0.2$(3+1)/(12+8)=0.2.
    - … and so on for the rest of the table.
  - The numbers shown here have been rounded, but using this method the sum of probabilities in your table will still be 1.
  - Most importantly, note that the word *because* no longer has a probability of 0.
- Shown below is the tabulation of the conditional probabilities for each word in V for the positive and negative class:



- **Key takeaway**
  - Use Laplacian smoothing so your probabilities don't end up being 0.

Naive Bayes

- In the previous module, we learned how to classify tweets using logistic regression. In this week, we'll solve the same problem using a method called the Naive Bayes. It's a reasonably **good** and **easy-to-implement** baseline for many text classification tasks, including sentiment analysis.
- Naive Bayes is an algorithm that falls under the domain of supervised machine learning, and relies on word frequency counts just like logistic regression.

*The "naive" in Naive Bayes comes from the fact that this method makes the assumption that the features you're using for classification are all **independent**, which in reality is **rarely the case**.*

- Formally, the Naive Bayes inference condition rule for binary classification can be expressed as follows:

$$\prod_{i=1}^{m} P(w_i|pos)P(w_i|neg)\prod_{i=1}^{m} P(w_i|pos)P(w_i|neg)$$

- Dissecting this expression, you can see that we're going to take the product (across all of the unique words in your tweets, i.e., your vocabulary) of the probability for each word in the positive class divided by its probability in the negative class.
- Let's apply the Naive Bayes inference condition rule for binary classification to an example, using the conditional probabilities table we derived in the prior section on computing the conditional probabilities table. Say your tweet says, `"I'm happy today, I'm learning"`. Using the table of conditional probabilities (after applying Laplacian smoothing), we can predict the sentiment of the whole tweet as shown below:

Tweet: I am happy today; I am learning.

$$\prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)}$$

word	Pos	Neg
I	0.20	0.20
am	0.20	0.20
happy	0.14	0.10
because	0.10	0.05
learning	0.10	0.10
NLP	0.10	0.10
sad	0.10	0.15
not	0.10	0.15

- Now, let's calculate the Naive Bayes product for our tweet.
- For each word, select its probabilities from the table. So for *I*, you get a positive probability of $0.20.2$0.20.2 and a negative probability of $0.20.2$0.20.2. So the ratio that goes into the product is $0.20.2$0.20.2. Similarly, the ratios for the next words are as in the table below.

Word	P(wi|pos)P(wi|neg)P(wi|pos)P(wi|neg)
I	0.20.20.20.2
am	0.20.20.20.2
happy	0.140.100.140.10
today	−−
I	0.20.20.20.2
am	0.20.20.20.2
learning	0.100.100.100.10

- Note:
  - Since there is no entry for *today* in our conditional probabilities table, this implies that this **word is not in your vocabulary**. So we'll **ignore** its contribution to the overall score.
  - All the neutral words in the tweet such as *I* and *am* **cancel out** in the expression, as shown in the figure below.

Tweet: I am happy today; I am learning.

$$\prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)} = \frac{0.14}{0.10} = 1.4 > 1$$

$$\frac{0.20}{0.20} * \frac{0.20}{0.20} * \frac{0.14}{0.10} * \frac{0.20}{0.20} * \frac{0.20}{0.20} * \frac{0.10}{0.10}$$

word	Pos	Neg
I	0.20	0.20
am	0.20	0.20
happy	0.14	0.10
because	0.10	0.05
learning	0.10	0.10
NLP	0.10	0.10
sad	0.10	0.15
not	0.10	0.15

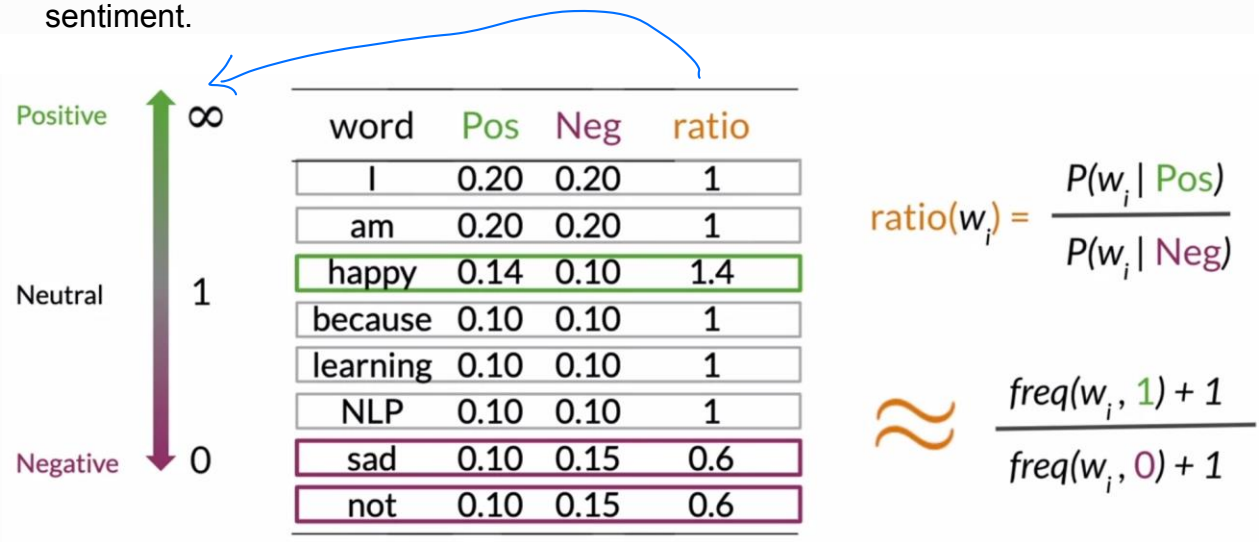
- Overall, what you end up with is 0.140.10=1.40.140.10=1.4.
- Since this value is higher than 1, we can infer that the words in the tweet collectively correspond to a **positive sentiment**, so you conclude that the tweet is **positive**.

Background

Log Likelihood

- Log likelihoods are just logarithms of the probabilities, like the (conditional) ones we calculated in the last section on computing the conditional probabilities table. Compared to raw probability numbers, they are much more convenient to work with and appear throughout deep-learning and NLP.

- Shown in the below figure is the the table we derived previously that contains the conditional probabilities of each word, for both classes, i.e., the positive or negative sentiment.



- Words can have many **shades** of emotional meaning. But for the purpose of sentiment classification, they're simplified into three categories: **neutral, positive, and negative**.
- Based on their conditional probabilities, we can identify the sentiment of a particular word wiwi. These classes/categories can be numerically estimated just by **dividing the corresponding conditional probabilities of this table**, as follows:

$$ratio(w_i)=P(w_i|pos)P(w_i|neg)ratio(wi)=P(wi|pos)P(wi|neg)$$

- Let's see how this ratio looks for the words in your vocabulary.
  - The ratio for the word *i* is $0.20.2=1.00.20.2=1.0$.
  - The ratio for the word *am*, is again, $0.20.2=1.00.20.2=1.0$.
  - The ratio for the word *happy* is $0.140.10=1.40.140.10=1.4$.
  - The ratio for the words *because*, *learning*, and *NLP*, the ratio is 1.
  - For *sad* and *not*, their ratio is $0.100.15=0.60.100.15=0.6$.
- Note that neutral words have a ratio near 1. Positive words have a ratio larger than 1.

*The larger the ratio, the **more positive** the word is going to be. On the other hand, negative words have a ratio **smaller than one**. The **smaller** the value, the **more negative** the word.*

- We can thus filter words depending on their **positivity** or **negativity**. These ratios are essential in Naive Bayes for binary classification.
- Let's illustrate this using an example. Recall that in Naive Bayes, you categorize a tweet as positive if the products of the corresponding ratios of every word appears in the tweet is larger than 1 and negative if it was less than 1. This is called the likelihood (boxed in blue in the figure below).

class $\in$ *{pos, neg}*
w -> *Set of m words in a tweet*

$$\frac{P(pos)}{P(neg)} \prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)} > 1$$

- A simple, fast, and powerful baseline
- A probabilistic model used for classification

Log Prior

- Let's talk about the term boxed in purple in the figure above.
- For each class, estimate the probability of occurrence. In our case, we have two classes: positive and negative. Next, let's assume:
  - $P(D_{pos})$P(Dpos) as the probability that the document/tweet is positive.
  - $P(D_{neg})$P(Dneg) as the probability that the document/tweet is negative.
  - Note that $P(pos)$P(pos) and $P(neg)$P(neg) represent $P(D_{pos})$P(Dpos) and $P(D_{neg})$P(Dneg) in the above figure.

    $P(D_{pos})=D_{pos}DP(D_{neg})=D_{neg}DP(Dpos)=DposDP(Dneg)=DnegD$

  - where,
    - DD is the total number of documents/tweets.
    - D_{pos}Dpos is the total number of positive tweets.
    - D_{neg}Dneg is the total number of negative tweets.
- The prior is the ratio of the probabilities $P(D_{pos})P(D_{neg})$P(Dpos)P(Dneg), which is effectively $D_{pos}D_{neg}$DposDneg. Thus, the ratio of the number of positive to negative tweets is called the **prior ratio**.
  - If you have **exactly the same number** of positive and negative tweets, the ratio is 1, which implies it needn't be considered.
  - However, when you have a **positive-negative class imbalance** (such datasets are called **imbalanced**), you need to accommodate the prior in your likelihood calculations.
- Intuitively, the prior probability represents the underlying probability in the target population that a tweet is positive versus negative. In other words, if we had no specific information and blindly picked a tweet out of the population set, the prior represents the probability that it will be positive versus negative.
- We can take the log of the prior to rescale it, and we'll call this the **log prior**:

$$\log \text{prior} = \log(P(D_{pos})P(D_{neg})) = \log(D_{pos}D_{neg})$$ log prior =log(P(Dpos)P(Dneg))=log(DposDneg)

- Using the quotient rule of logarithms which states that $\log(AB) = \log(A) - \log(B)$log(AB)=log(A)−log(B), the log prior can thus be calculated as the difference between the two log terms:

$$\log \text{prior} = \log(P(D_{pos})) - \log(P(D_{neg})) = \log(D_{pos}) - \log(D_{neg})$$ log prior =log(P(Dpos))−log(P(Dneg))=log(Dpos)−log(Dneg)

Preventing Numerical Underflow

- Another important consideration for your implementation of the Naive Bayes classifier is the possibility of a **numerical underflow**.
- Sentiment probability calculation requires multiplication of many numbers with values between $[0,1]$[0,1]. Carrying out such repeated multiplications runs the risk of numerical overflow since the magnitude of the result diminishes with every multiplication – in some cases, if the underlying hardware does not support the precision required to store such small magnitude numbers on the device.
- Luckily, there's a **mathematical trick** to solve this which involves using a **property of logarithms**.
  - Logarithms convert a product of numbers to an addition operation – this is referred to as the product rule in logarithms. This helps with the numerical underflow issues that we encounter when **multiplying multiple small numbers** in case of raw conditional probability ratios, because the multiplication operation is transformed to an addition. Formally, the product rule of logarithms states that:

$$\log(a \cdot b) = \log(a) + \log(b)$$log(a·b)=log(a)+log(b)

Binary Classification Formulation

- Recall that the formula you're using to calculate a score for Naive Bayes is the **prior multiplied by the likelihood**. The prior ratio coupled with the likelihood represent the full Naive Bayes formulation for binary classification, which is a **simple, fast, and powerful method** that you can use to establish a baseline quickly. Formally,

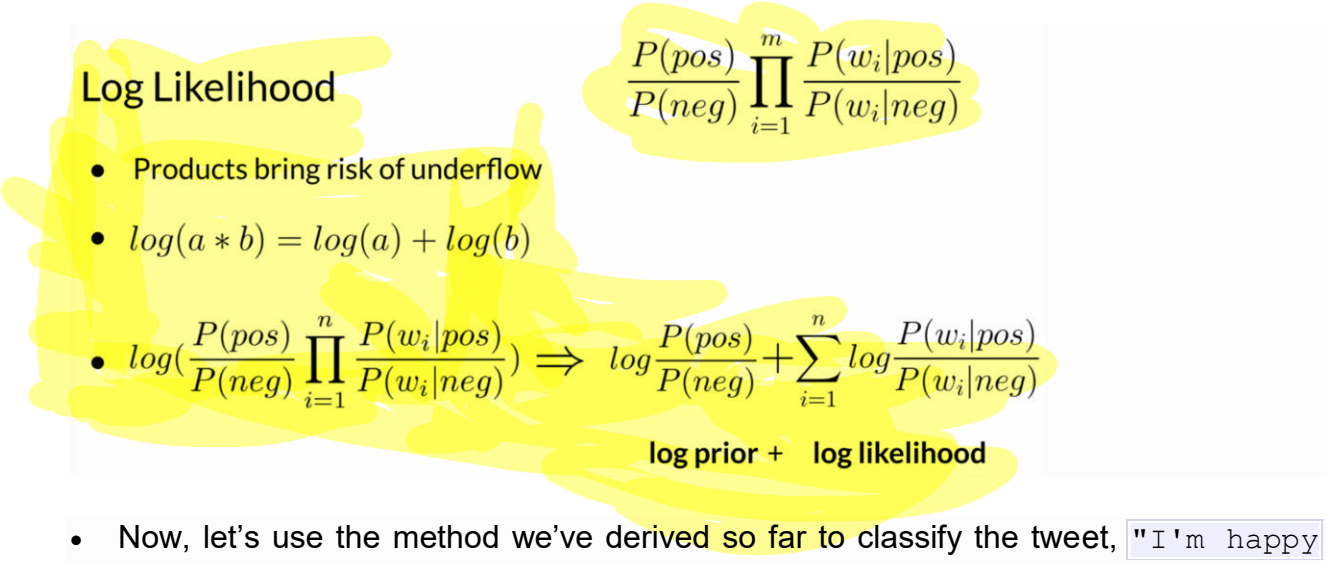
$$p = \text{prior} \times \text{loglikelihood} = P(pos)P(neg)\prod_{i=1}^{m} P(w_i|pos)P(w_i|neg)$$p=prior×loglikelihood =P(pos)P(neg)∏i=1mP(wi|pos)P(wi|neg)

- Based on our discussions of using logs in the above section on preventing numerical underflow, let's explore the trick of using the **log of the likelihood score** rather than the **raw likelihood score**. Recall that the likelihood score is essentially the ratio of the conditional probabilities corresponding to each of the possible classes.

- Owing to the product rule of logarithms, we may re-write the previous expression as the **sum** of the **log prior** and the **log likelihood**. Recall that the log likelihood itself is a sum of the logarithms of the conditional probability ratio of all unique words in your corpus. Formally,

$$p = \log \text{ prior} + \sum_{i=1}^{N} \log \text{ likelihood}(w_i)$$

  - where,
    - w_i represents a word in the tweet.
    - N is the number of words in the tweet.
- The figure below summarizes the above ideas:



- Now, let's use the method we've derived so far to classify the tweet, `"I'm happy because I'm learning"`.
- Note that the values in the table here are **unrelated** to the previous dataset. It uses a bigger corpus and shows just a subset of the vocabulary, which is why we have lower probabilities and the sums per column doesn't up to 1.
- Recall how we used the Naive Bayes inference condition earlier to get the sentiment score for your tweets. We're going to do something very similar to get the **log of the score**, also known as the **log likelihood**, denoted by λ. Formally,

$$\text{log likelihood} = \lambda(w) = \log(w|pos) \log(w|neg) \log$$

likelihood=λ(w)=log⁡(w|pos)log⁡(w|neg)

- Now let's calculate λ for every word in our vocabulary.
- For the word *I*, you get the logarithm of $0.05 0.05$ 0.050.05 or the logarithm of 1, which is equal to 0. Recall that the tweet will be labeled **positive** if the product is larger than 1.
  - By this logic, *I* would be classified as **neutral** at 0.

Just a different way to express ratio basically instead of having threshold of 1, we have threshold of 0 while things in the mid stands neutral.

tweet: I am happy because I am learning.

$$\lambda(w) = log\frac{P(w|pos)}{P(w|neg)}$$

this is important

$$\lambda(I) = log\frac{0.05}{0.05} = log(1) = 0$$

word	Pos	Neg	λ
I	0.05	0.05	0
am	0.04	0.04	
happy	0.09	0.01	
because	0.01	0.01	
learning	0.03	0.01	
NLP	0.02	0.02	
sad	0.01	0.09	
not	0.02	0.03	

- For *am*, you take the log of $0.040.040.040.04$, which again is equal to 0.

tweet: I am happy because I am learning.

$$\lambda(w) = log\frac{P(w|pos)}{P(w|neg)}$$

$$\lambda(\text{am}) = log\frac{0.04}{0.04} = log(1) = 0$$

word	Pos	Neg	λ
I	0.05	0.05	0
am	0.04	0.04	
happy	0.09	0.01	
because	0.01	0.01	
learning	0.03	0.01	
NLP	0.02	0.02	
sad	0.01	0.09	
not	0.02	0.03	

- For *happy*, you get a $\lambda\lambda$ of $2.22.2$, which is greater than 0, indicating a **positive sentiment**.

doc: I am happy because I am learning.

$$\lambda(w) = log\frac{P(w|pos)}{P(w|neg)}$$

$$\lambda(\text{happy}) = log\frac{0.09}{0.01} \approx 2.2$$

word	Pos	Neg	λ
I	0.05	0.05	0
am	0.04	0.04	0
happy	0.09	0.01	2.2
because	0.01	0.01	
learning	0.03	0.01	
NLP	0.02	0.02	
sad	0.01	0.09	
not	0.02	0.03	

- From here on out, you can calculate the log score of the entire corpus just by summing out the $\lambda\lambda$ terms.
- Thus, the log-likelihood of a tweet can be computed as the **sum of the $\lambda\lambda$ terms corresponding to each word in the tweet**.

- Considering our example:
  - For the word *I*, you add 0.
  - For *am*, you add 0.
  - For *happy*, you add $2.22.2$.
  - For *because*, *I* and *am*, you add 0.
  - For *learning*, you add $1.11.1$.
  - The log-likelihood sum is $3.33.3$. ⇒ this statement is positive

doc: I am happy because I am learning.

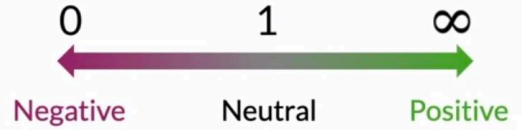
$$\sum_{i=1}^{m} log \frac{P(w_i|pos)}{P(w_i|neg)} = \sum_{i=1}^{m} \lambda(w_i)$$

log likelihood = 0 + 0 + 2.2 + 0 + 0 + 0 + 1.1 = **3.3**

word	Pos	Neg	λ
I	0.05	0.05	0
am	0.04	0.04	0
happy	0.09	0.01	2.2
because	0.01	0.01	0
learning	0.03	0.01	1.1
NLP	0.02	0.02	0
sad	0.01	0.09	-2.2
not	0.02	0.03	-0.4

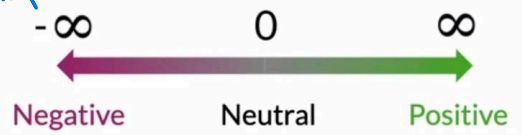
- Recall that the tweet was positive if the product was larger than 1, and negative otherwise. With the log of 1 equal to 0, the positive values indicate that the tweet is positive. A value less than 0 would indicate that the tweet is negative.
- Since the log-likelihood for this tweet is $3.33.3$, and this value is higher than 0, the tweet is marked **positive**.

$$\prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)} > 1$$



$$\sum_{i=1}^{m} log \frac{P(w_i|pos)}{P(w_i|neg)} > 0$$

⇒ prevent from getting too small so comp can't catch it

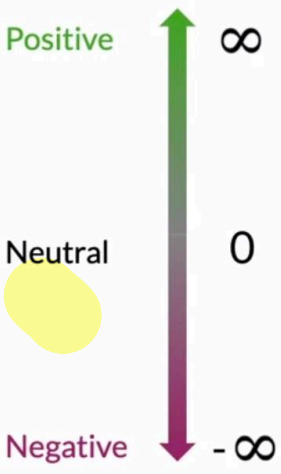


- Notice that this score is based entirely on the words *happy* and *learning*, both of which carry a positive sentiment. All the other words were **neutral** and didn't **contribute** to the score. This shows the kind of influence power words have.

- **Key takeaways**
  - Words are often emotionally ambiguous but you can simplify them into **three categories** (neutral, positive, and negative) and measure exactly where they fall within those categories for binary classification.
  - To do so, you divide the **conditional probabilities** of the words in each category/class.
  - This ratio can be expressed as a logarithm and is denoted by $\lambda\lambda$.
  - Computing the log of the likelihood helps reduce the risk of numerical underflow (due to multiplying many small numbers $\ll 1 \ll 1$).
  - Predict the sentiment of a tweet by summing all the $\lambda\lambda$ terms corresponding to each word that appears in the tweet. This score is called the **log-likelihood**.
  - For log-likelihood, the decision threshold is 0 instead of 1 as with raw scores. Positive tweets will have a log-likelihood above 0, and negative tweets will have a log-likelihood below 0.
    - Considering the **log prior** in case of **unbalanced datasets**, if $\log \text{likelihood} + \log \text{prior} > 0$log likelihood + log prior > 0, then the tweet has a positive sentiment, or negative otherwise.
  - Log-likelihoods make calculations **simpler** (since addition is simpler than multiplication, not only from a time-complexity perspective, but also a space-complexity perspective) and they also help with numerical stability.

Tweet sentiment:

$$log \prod_{i=1}^{m} ratio(w_i) = \sum_{i=1}^{m} \lambda(w_i) \quad > 0$$

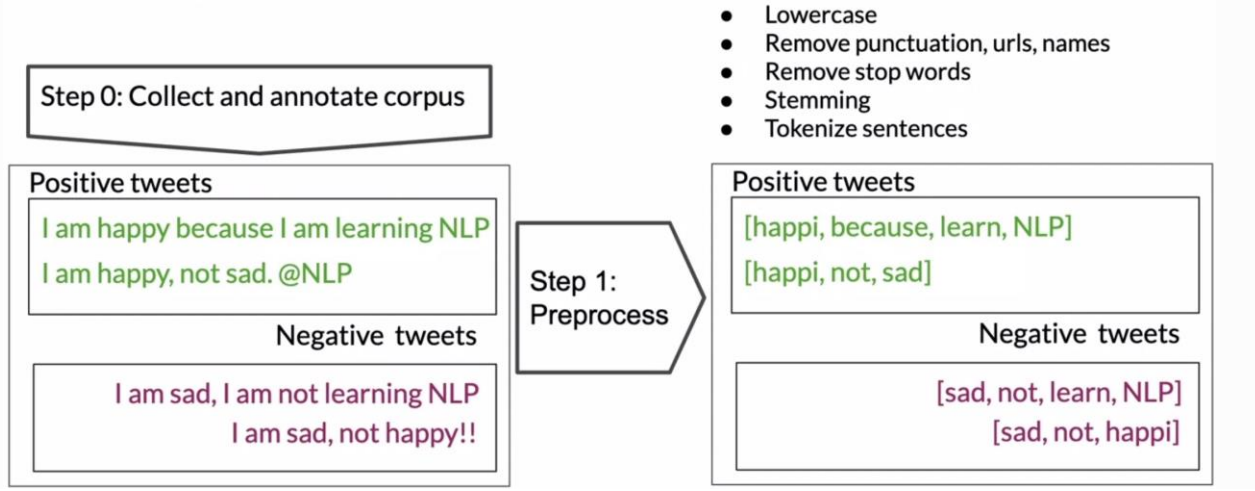


Training Naive Bayes

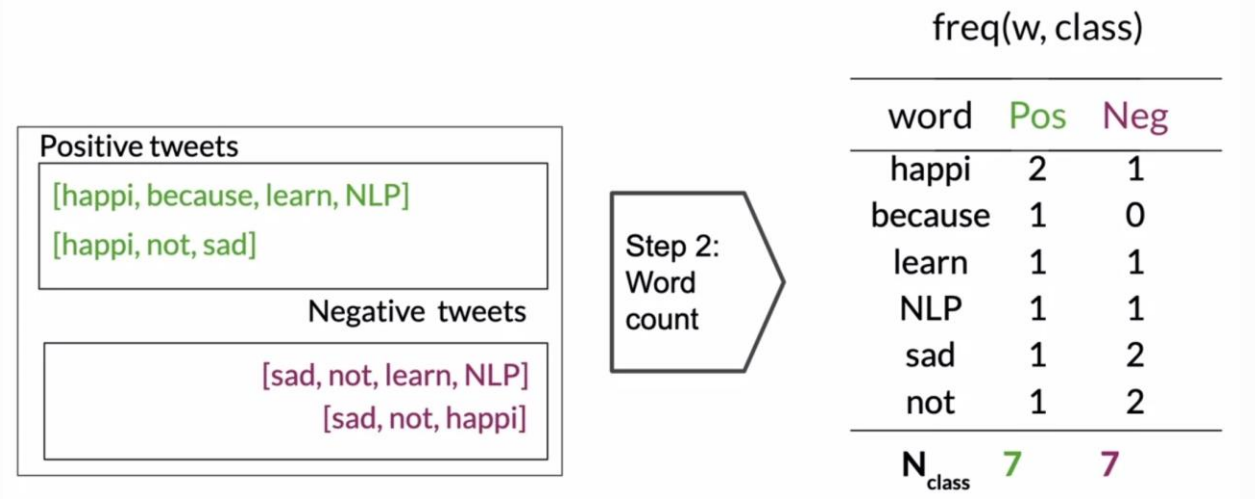
- Let's train the Naive Bayes classifier. Note that in this context, training has a slightly different meaning than in logistic regression or deep learning. There is **no gradient descent**, instead, we're just counting frequencies of words in a corpus.
- The steps involved in training a Naive Bayes model for sentiment analysis are as follows.
- **Gather data:**

- For any supervised machine learning project, step 0 is to gather and label the data required to train and test your model.

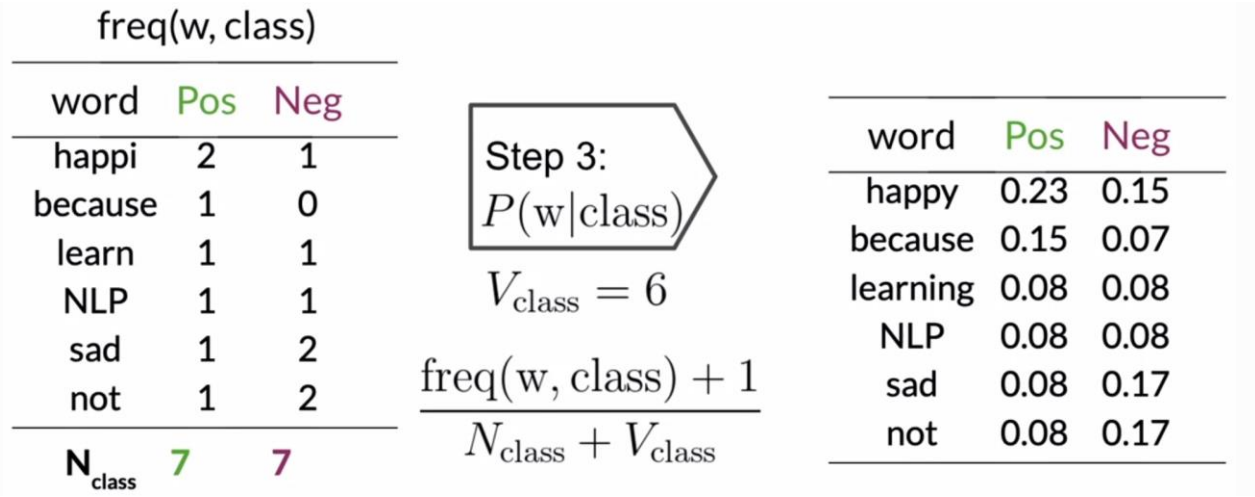
- **Annotate your dataset:**
  - For sentiment analysis of tweets, this step involves getting a corpus of tweets and annotating the dataset into two groups, positive and negative tweets.
- **Preprocessing:** The next step (shown in the figure below) is fundamental to your model's success. The preprocessing step as described in the previous module, eliminates noise from your data by removing words that don't offer the task at hand (in this case, sentiment analysis) relevant information about the content. These include all common words such as *I, you, are, is,* etc. that would not give us enough information on the sentiment. consists of five sub-steps:
  - Lowercase
  - Remove stop words and punctuation
  - Remove task-specific stop words: URLs, handles etc.
  - Stemming: reducing words to their common stem
  - Tokenize sentences: splitting the corpus into single words or tokens.
- At the output of this step, we obtain a corpus of clean, standardized tokens.
- Note that it is relatively straightforward to implement this processing pipeline with simple ideas. However, in the real world, gathering and processing of data typically takes up a big chunk of the project's timeline.
  - The famous 80/20 rule of data science aptly summarizes this observation: "80%80% of a data scientist's valuable time is spent simply finding, cleansing, and organizing data, leaving only 2020 to actually perform analysis."



- **Computing** freq(wi,class)freq(wi,class) **for each word in your vocabulary:** Once you have a clean corpus of tweets, obtain a vocabulary consisting of unique words in the corpora for each class. For each tuple of (word,class)(word,class), compute the frequencies similar to what we did with logistic regression. This will yield a table like the one shown in the figure here. Next, compute the sum of words and class in each corpus.



- **Get probability ratios** P(w|pos)P(w|neg)P(w|pos)P(w|neg) **using the Laplacian smoothing formula:**
  - From this table of frequencies, you get the conditional probability for a given class by using the Laplacian smoothing formula. As shown in the figure below, the number of unique words in V_{class}Vclass is 6.
  - You only account for the words in the table, not the total number of words in the original corpus corresponding to that class.
  - This produces a table of conditional probabilities for each word and each class, which only contains values greater than 0.



- **Compute** λ(w)λ(w)**:** Obtain the **lambda score for each word**, which is the log of the ratio of your conditional probabilities.

freq(w, class)

word	Pos	Neg
happi	2	1
because	1	0
learn	1	1
NLP	1	1
sad	1	2
not	1	2
N_{class}	7	7

Step 3: $P(\text{w}|\text{class})$

$V_{\text{class}} = 6$

$$\frac{\text{freq}(\text{w}, \text{class}) + 1}{N_{\text{class}} + V_{\text{class}}}$$

$$\lambda(w) = log\frac{P(\text{w}|\text{pos})}{P(\text{w}|\text{neg})}$$

Step 4: Get lambda

log of the ratio of the conditional probability

word	Pos	Neg	λ
happy	0.23	0.15	0.43
because	0.15	0.07	0.6
learning	0.08	0.08	0
NLP	0.08	0.08	0
sad	0.08	0.17	-0.75
not	0.08	0.17	-0.75

- **Estimate** log prior=log$P(pos)P(neg)$log prior=log P(pos)P(neg)**:** Calculate the log prior by counting the number of positive and negative tweets. The log prior is the log of the ratio of the number of positive tweets over the number of negative tweets. For balanced datasets, the log prior is 0. For unbalanced datasets, this term is important.

very important ★

Step 5: Get the log prior

D_{pos} = Number of positive tweets
D_{neg} = Number of negative tweets

$$\text{logprior} = log\frac{D_{pos}}{D_{neg}}$$

If dataset is balanced, $D_{pos} = D_{neg}$ and logprior = 0.

- **Key takeaways**
  - Annotate a dataset with positive and negative labels for tweets.
  - Pre-process the raw text to get a corpus of clean, standardized tokens.
  - Compute the frequencies corresponding to each word and class (positive/negative) $\text{freq}(w,\text{class})$freq(w,class).
  - Compute the conditional probabilities corresponding to each word and class $P(\text{w}|\text{pos});P(\text{w}|\text{neg})$P(w|pos);P(w|neg) using the frequencies computed in the prior step, and applying the Laplacian smoothing formula.
  - Obtain the conditional probability ratio $P(\text{w}|\text{pos})P(\text{w}|\text{neg})$P(w|pos)P(w|neg).
  - Compute $\lambda(\text{w})$λ(w) or the log-likelihood score for each word.

- Finally, estimate the log prior of the model or how **likely it is to see a positive tweet** in your corpus.

Testing Naive Bayes

- Let's work on applying the Naive Bayes classifier on validation examples to compute the model's accuracy. Note that we shall cover some special corner cases.
- The steps involved in testing a Naive Bayes model for sentiment analysis are as follows.
- **Use the validation set:** To evaluate/test the trained model, we take the conditional probabilities and use them to predict the sentiments of new unseen tweets, which in our case is the validation set of annotated tweets. The validation set includes data that was set aside during training and is composed of a set of raw tweets X_{val}Xval, and their corresponding sentiments, Y_{val}Yval.
- **Pre-processing:** Removing the punctuation, stemming the words, and tokenizing to produce a vector of words like this one.
- **Compute the λλ score for each unique word:** Using the λλ table (i.e., the log-likelihood table) for each unique word in our vocabulary, we compute the score for each word $\lambda(X_{val})$λ(Xval) in the input sample.
  - For words that have corresponding entries in the table, you sum over all the corresponding λλ terms.
  - Words that don't show up in the log-likelihood table are considered neutral and don't **contribute anything** to the overall score.
  - Note that - Words that are not seen in the training set are considered neutral, and so add **0** to the score. your model can only give a score for words **it's seen before**!
- **Obtain the overall score:** Summing up the scores of all the individual words, along with with our estimation of the log prior (important for an unbalanced dataset), we can predict the sentiment on a new tweet.
- **Obtain the final prediction:** The final prediction is $\text{score} > 0$score>0.
- Let's consider an example tweet, `"I passed the NLP interview"`, and use our trained model to predict if this is a positive or negative tweet:
- Look up each word from the vector in your log-likelihood table. Words such as *I*, *pass*, *the*, *NLP* have entries in the table, while the word *interview* does not (which implies that it needs to be ignored). Now, add the log prior to account for the imbalance of classes in the dataset. Thus, the overall score sums up to $0.480.48$, as shown in the figure below.
- Recall that if the overall score of the tweet is larger than 0, then this tweet has a positive sentiment, so the overall prediction is that this tweet has a **positive sentiment**. Even in real life, passing the NLP interview is a **very positive thing**.

- log-likelihood dictionary $\lambda(w) = log\frac{P(w|pos)}{P(w|neg)}$
- $logprior = log\frac{D_{pos}}{D_{neg}} = 0$
- Tweet: [I, pass, the, NLP, interview]

This word hasn't appeared in the corpus so it will be treated as neutral.

word	λ
I	-0.01
the	-0.01
happi	0.63
because	0.01
pass	0.5
NLP	0
sad	-0.75
not	-0.75

$$score = -0.01 + 0.5 - 0.01 + 0 + logprior = 0.48$$

$$pred = score > 0$$

- To test the performance of our classifier on unseen data, we'll need to implement an accuracy function to measure the performance of our trained model. To do so…
- **Obtain the predictions for each entry in X_{val}Xval:** Compute the score of each sample in X_{val}Xval, like you just did previously, then evaluates whether each score is greater than zero. This produces a vector populated with 0s and 1s indicating if the predicted sentiment is negative or positive for each tweet sample in the validation set, as shown below.

- $X_{val}\ Y_{val}\ \lambda\ logprior$

$$score = predict(X_{val}, \lambda, logprior)$$

$$pred = score > 0$$

$$\begin{bmatrix} 0.5 \\ -1 \\ 1.3 \\ \vdots \\ score_m \end{bmatrix} > 0 = \begin{bmatrix} 0.5 > 0 \\ -1 > 0 \\ 1.3 > 0 \\ \vdots \\ score_m > 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \\ pred_m \end{bmatrix}$$

- **Compute the prediction accuracy:**
  - With our new predictions vector, we can compute the accuracy of your model using the validation set. To do so, compare your predictions for each X_{val}Xval against its ground truth Y_{val}Yval. If the values are equal and your prediction is correct, you will get a value of 1, and 0 otherwise.
  - Once you have compared the values of every prediction with the true labels of your validation sets, you can compute the accuracy as the sum of this vector

divided by the number of examples in the validation sets, as shown below. This is similar to what we did with logistic regression.

- X_{val} Y_{val} λ $logprior$

$score = predict(X_{val}, \lambda, logprior)$

$pred = score > 0$

$$\frac{1}{m}\sum_{i=1}^{m}(pred_i == Y_{val_i})$$

$$\begin{bmatrix} 0 \\ \underline{1} \\ 1 \\ \vdots \\ pred_m \end{bmatrix} == \begin{bmatrix} 0 \\ \underline{0} \\ 1 \\ \vdots \\ Y_{val_m} \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ \underline{0} \\ 1 \\ \vdots \\ pred_m == Y_{val_m} \end{bmatrix}$$

- **Key takeaways**
  - To test the performance of your trained Naive Bayes model, use a validation set to predict the sentiment score for unseen tweets.
  - Compared your predictions with ground truth labels provided as part of the validation set to calculate the accuracy of the model by identifying the proportion of tweets that were correctly predicted by your model.
  - Words that don't appear in the λλ table are treated as neutral, and don't add anything to the class score.

Applications

- Beyond sentiment classification, Naive Bayes lends itself to a plethora of applications.
- When you use Naive Bayes to predict the sentiments of a tweet, what you're actually doing is **estimating the probability for each class** using the **joint probability of the words in the classes**. The Naive Bayes formula states that the **ratio between these two probabilities** is given by the **products of the priors and the likelihoods**, as shown in the figure below. You can use this ratio between conditional probabilities for much more than sentiment analysis.

$$P(pos|tweet) \approx P(pos)P(tweet|pos)$$
$$P(neg|tweet) \approx P(neg)P(tweet|neg)$$

$$\frac{P(pos|tweet)}{P(neg|tweet)} = \frac{P(pos)}{P(neg)} \prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)}$$

- **Author identification:** If you had two large corpora, each written by different authors, you could train the model to recognize whether a new document was written by one or the other. Suppose if you had some works by Shakespeare and some by Hemingway, you could calculate the λλ for each word to predict how likely a new word is to be used by Shakespeare or Hemingway. This method also allows us to determine author identity.

Author identification:

$$\frac{P(\text{  }|\text{book})}{P(\text{  }|\text{book})}$$

- **Spam filtering:** Using information taken from the sender, subject and content, we can decide whether an email is spam or not.

Spam filtering:

$$\frac{P(\text{spam}|\text{email})}{P(\text{nonspam}|\text{email})}$$

- **Information retrieval:** One of the earliest uses of Naive Bayes was filtering between **relevant** and **irrelevant** documents in a database. Given the sets of

keywords in a query, in this case, you only needed to calculate the likelihood of the documents given the query. However, the challenge here is that you can't know beforehand what a relevant or irrelevant document looks like. So you can **compute the likelihood for each document in your dataset** and then **store the documents based on its likelihoods**. You can choose to keep the first kk results or the ones that have a likelihood larger than a certain threshold. Formally:

$$P(\text{document}_\text{k}|\text{query}) \propto \prod_{i=0}^{|query|} P(\text{query}_\text{i}|\text{document}_\text{k})$$

Retrieve document if $P(\text{document}_\text{k}|\text{query}) > \text{threshold}$

- **Word disambiguation:** Word disambiguation involves breaking down words for **contextual clarity**. Consider that you have only **two possible interpretations** of a given word within a text. Let's say you don't know if the word *bank* in your reading is referring to the bank of a river or to a financial institution. To disambiguate your word, calculate the score of the document, given that it refers to each one of the possible meanings. In this case, if the text refers to the concept of river instead of the concept of money, then the corresponding score P(river|text)P(river|text) will be larger than 1.

$$\frac{P(\text{river}|\text{text})}{P(\text{money}|\text{text})}$$

Bank:

- **Key takeaway**
  - Bayes Rule and it's naive approximation has a wide range of applications. It's a popular method since it's relatively simple to train, use and interpret.

Assumptions

- Let's understand the assumptions that underlie the Naive Bayes method.
- The primary assumption is the independence of features.
- Naive Bayes is a very simple model because it doesn't require setting any **custom parameters**. This method is referred to as naive because of the assumptions it makes about the data. The first assumption is the independence between the predictors or features associated with each class and the second has to do with your validation set. Let's explore each of these assumptions and understand how they could affect your results.

Independence

- To illustrate the idea of independence between features, let's consider the following sentence: `It is sunny and hot in the Sahara Desert.`
- Naive Bayes assumes that the words in a sentence are **independent** of one another, but in reality, this typically **isn't the case**. The word *sunny* and *hot* often appear together as they do in this example. Taken together, they might also be related to the location they're describing, such as a *beach* or a *desert*. So the words in a sentence are not always necessarily independent of one another, but Naive Bayes assumes that they are. This could affect your estimation of the conditional probabilities of individual words.

- Let's consider the task of sentence completion, given a sentence: `"It's always cold and snowy in ___"`, Naive Bayes might assign equal probability to the words spring, summer, fall, and winter even though from the context you can see that **winter** should be the most likely candidate. In the next modules of this specialization, we will look into sophisticated methods that can handle these cases better.

“It’s always cold and snowy in ___.”

spring?? summer? fall?? winter??

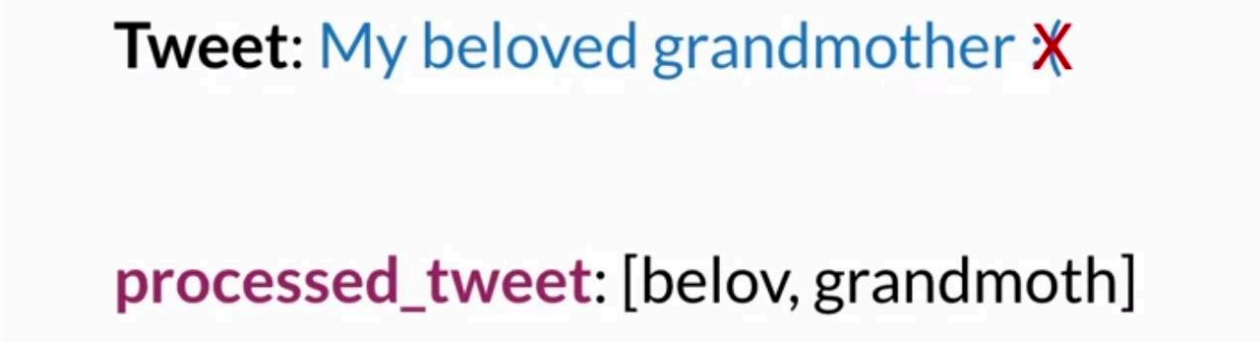
Relative Class Frequencies in Corpus

- Another issue with Naive Bayes is that it relies on the **distribution of the training dataset**, i.e., the proportion of the individual classes in the dataset.
- A good data set will contain the same proportion of positive and negative tweets as a random sample would. However, most available annotated corpora are **artificially balanced** for ease of analysis.
- In a real tweet stream, **positive tweets occur more often** than their negative counterparts.
    - One reason for this is that negative tweets might contain content that is banned by the platform or muted by the user such as inappropriate or offensive vocabulary. Assuming that reality behaves as your training corpus, this could result in a skewed optimistic or pessimistic model.
- Thus, the relative frequency of the classes can affect the model’s performance if they are not representative of the real-world distribution.

- **Key takeaways**
    - The assumption of independence in Naive Bayes is very difficult to guarantee, but despite that, the model works pretty well in certain situations.
    - The relative frequency of positive and negative tweets in your training data sets needs to be **balanced** in order to deliver accurate results.

Error Analysis

- Let's delve into analyzing errors associated with cases where the model's prediction didn't turn out right and lead to a misclassified sentence, in our particular case of sentiment classification.
- There are several steps in the model's pipeline where issues can lead to errors in the model's prediction. Examples below:
  - Incorrect preprocessing can lead to our model not being to able to clearly figure out the semantic meaning behind the input sentence.
  - Word order drastically affects the meaning of a sentence.
  - Some quirks of languages come naturally to humans, but confuse Naive Bayes models.
  - Let's delve into each of them one-by-one.
- **Preprocessing**
  - One of your main considerations when analyzing errors in NLP systems should be what the preprocessed version of the text actually looks like.
  - Consider the tweet: `My beloved grandmother :(`.
  - The sad face at the end of the tweet is very important to the sentiment of the tweet because it simply tells you the tweet carries a negative sentiment.
  - If you're removing punctuation, then the processed tweet will leave behind the stem-words associated with only `beloved grandmother`, which appears to be a very positive tweet.
  - Similarly, `My beloved grandmother!` would be a very different sentiment.

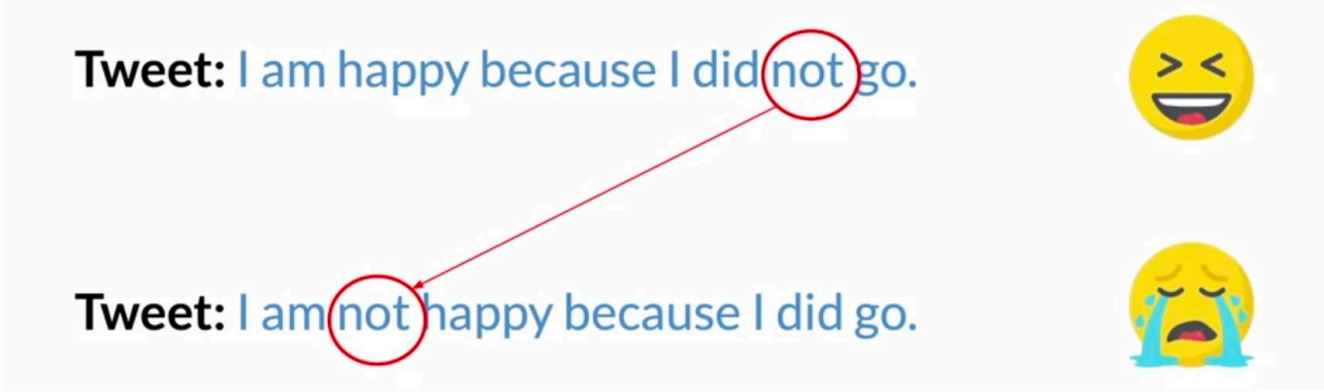


- Consider another example: `This is not good, because your attitude is not even close to being nice.`
  - If you remove the neutral words such as *not* and *this*, what you're left with is the following: *Good, attitude, close, nice*.
  - From this set of words, any classifier will infer that this is something very **positive**.

Tweet: This is not good, because your attitude is not even close to being nice.

processed_tweet: [good, attitude, close, nice]

- **Word order**
  - The input pipeline isn't the only potential source of trouble.
  - Consider the tweet: `I'm happy because I did not go` – this is a purely positive tweet. Now, consider another tweet: `I'm not happy because I did not go`, with a negative sentiment.
  - In this case, the *not* is important to the sentiment but gets missed by your Naive Bayes classifier.
  - Thus, word order can be as important to spelling.



- **Adversarial attacks**
  - A common pit-fall of Naive Bayes are adversarial attacks.
  - The term adversarial attack describes some common language phenomenon like sarcasm, irony, and euphemism. Humans pick these up quickly but machines are terrible at it.
  - Consider a tweet: `This is a ridiculously powerful movie. The plot was gripping and I cried right through until the ending`.
    - The tweet contains a somewhat positive movie review, but pre-processing might suggest otherwise. If you pre-process this tweet, you'll get a list of mostly negative words, but as you can see, they were actually used to describe a movie that the author enjoyed.

Tweet: This is a ridiculously powerful movie. The plot was gripping and I cried right through until the ending!

processed_tweet: [ridicul, power, movi, plot, grip, cry, end]

- **Key takeaways**
  - Errors in a Naive Bayes-powered system can happen at the following locations in a typical model's pipeline:
    - Preprocessing
      - Removing punctuation (example ':(')
      - Removing stop words
    - Word order
      - E.g.: `I am happy because I did not go` vs. `I am not happy because I did go`
    - Adversarial attacks
      - Easily detected by humans but algorithms are usually terrible at it!
      - Sarcasm, Irony, Euphemisms, etc.
      - E.g.: `This is a ridiculously powerful movie. The plot was gripping and I cried right through until the ending.`

Further Reading

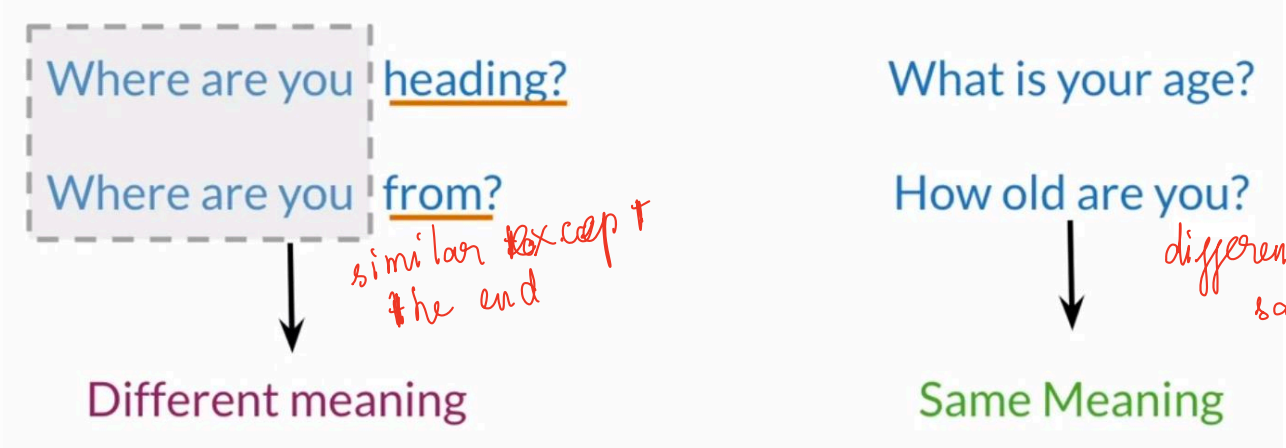
Here are some (optional) links you may find interesting for further reading:

- Properties of logarithms.
- 80/20 rule of data science.

Week 3: Sentiment Analysis Using Naive Bayes

Vector Space Models → for. question paraphrasing

- Suppose you have two questions: `"Where are you heading?"` and `"Where are you from?"`. Note that these sentences have identical words, except for the last ones. However, they both have a different meaning, as shown in the left-hand-side of the figure below.
- On the other hand, say you have two more questions whose words are completely different but both sentences mean the same thing, as shown in the right-hand-side of the figure below.



- Vector space models help you identify whether the first pair of questions or the second pair are **similar** in meaning **even if they do not share the same words**. Similarity identification has applications in **question answering, paraphrasing, and summarization**.
- Vector space models also allow you to capture **dependencies between words** by representing words as vectors, referred to as **word vectors/vector representations of words/word embeddings**.
  - Consider the sentence: `"You eat cereal from a bowl"`. Here, you can see that the word **cereal** and the word **bowl** are related.
  - Now let's look at this other sentence: `"You buy something and someone else sells it"`. What the sentence is saying is that someone sells something **because** someone else buys it. The second half of the sentence is dependent on the first half.
- With vector space models, you can capture **relationships** among different sets of words. Vector space models are used in information extraction to answer questions

in the style of *who, what, where, how,* etc. – which have applications in information extraction, machine translation and chatbots.

- You eat *cereal* from a *bowl*
- You *buy* something and someone else *sells* it

Information Extraction

Machine Translation


Chatbots

- John Firth, a famous English linguist, once said: `"You shall know a word by the company it keeps"`. This is one of NLP's **most fundamental concepts**. When using vector space models, the way that representations are made is by identifying the context around each word in the text, and this captures the relative meaning.

"You shall know a word by the company it keeps" Firth, 1957


(Firth, J. R. 1957:11)

- **Key takeaways**
  - Vector space models enable representing words in text documents like tweets, articles, queries, or more generally, an object that contains text, as a **vector**.
  - Since these word vectors capture **relative meaning** by identifying the **context** of the word, they lend themselves to applications where similarity identification is of

essence. This includes the fields of information retrieval, indexing, relevancy ranking, information filtering etc.
    - For e.g., when you look up a query online, the algorithm relies on the above concepts to give you back your results.
  - NLP fundamental concept: `"You shall know a word by the company it keeps"` - Firth, 1957.
  - Applications include:
    - Information Extraction
    - Machine Translation
    - Chatbots

Word by Word Design

- Let's delve into how you can construct word vectors from scratch using a **co-occurrence matrix**. Depending on the task at hand that you're trying to solve, you can have several possible designs.
- To get a vector space model using a word by word design, you'll compute a co-occurrence matrix and extract vector presentations for the words in your corpus.
- The co-occurrence of two different words is the number of times that they appear in your corpus together **within** a certain word distance k.
  - For instance, suppose that your corpus has the following two sentences: `"I like sample data"` and `"I prefer simple raw data"`.
  - The row of the co-occurrence matrix corresponding to the word *data* with $k=2$ would be populated as shown in the figure below.
  - For the column corresponding to the word *simple*, you would get a value equal to 2 because data and simple co-occur in the first sentence within a distance of 1 word, and in the second sentence within a distance of 2 words.
  - The row of the co-occurrence matrix corresponding to the word data would look like this if you consider the co-occurrence with the words *simple, raw, like,* and *I*. In this case, the vector representation of the word *data* would be equal to $2,1,1,0$.
  - With a word by word design, you can get a representation with n entries, with $n \in [1,\text{size of your entire vocabulary}]$.
- Let's take another example. Suppose you're using a word by word design, and you are building the co-occurrence matrix for the following text: `"In general, I love music. But I love pop music more than any other musical genre. To me, music is my greatest love."`
  - The right value for the co-occurrence of "love" and "music", if $k=2$ is 2.

Number of times they *occur together within a certain distance k*

k=2

I like simple data

I prefer simple raw data

	simple	raw	like	I
data	2	1	1	0

Word by Document Design

- For a word by document design, the process is quite similar. In this case, you'll count the times that a particular word from your vocabulary appears in documents that belong to specific categories.
  - For instance, you could have a corpus consisting of documents between different topics like *entertainment*, *economy*, and *machine learning*. Next, you would have to count the number of times the word appears in documents that belong to each of the three categories.
- In our particular e.g., suppose that the word *data* appears $500$500 times in documents from your corpus related to *entertainment*, $6620$6620 times in *economy* documents, and $9320$9320 in documents related to *machine learning*. Similarly, the word *film* appears in each document's category $7000,4000$7000,4000 and $1000$1000 times respectively.

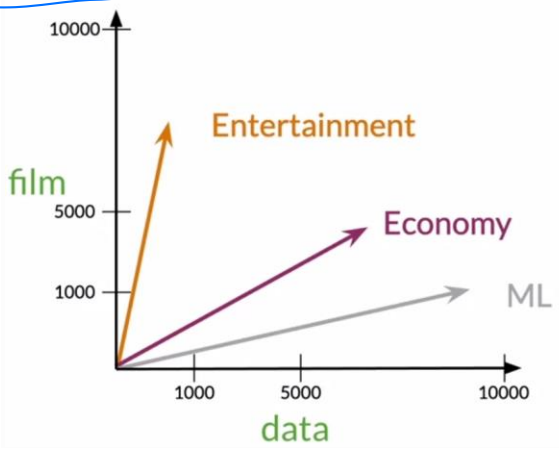
Number of times a word *occurs within a certain category*

	Entertainment	Economy	Machine Learning
data	500	6620	9320
film	7000	4000	1000

- Once you've constructed the representations for multiple sets of documents or words, the next step is to port these to **vector space**.

- Using the matrix we generated, we'll take the representation for every category of document by looking at the columns. So the vector space will have 2 dimensions – the number of times that the words *data* and *film* appear on the type of document.
- For the *entertainment*, *economy* and *machine learning* category, you would have the vector representation shown in the below figure. Note that in the vector space, it is easy to see that the *economy* and *machine learning* documents are much more similar than they are to the *entertainment* category.

Vector space



	Entertainment	Economy	ML
data	500	6620	9320
film	7000	4000	1000

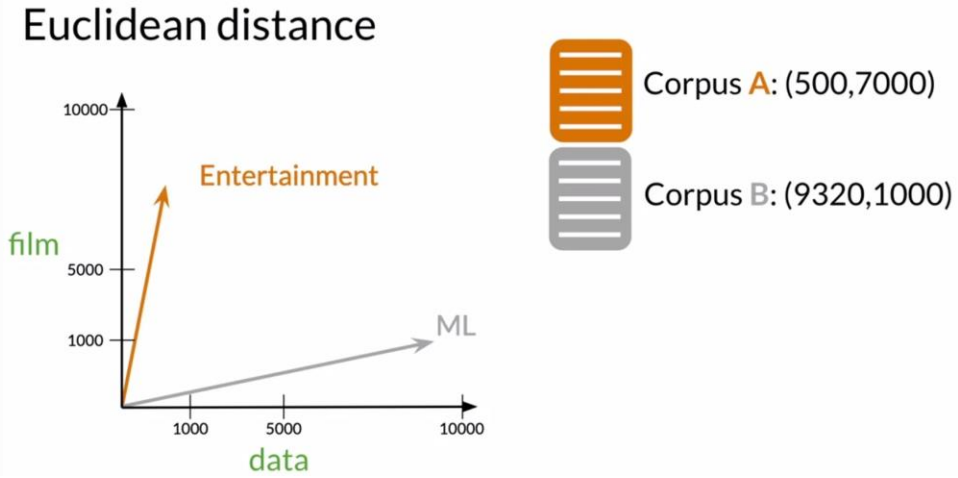
Measures of "similarity:"
Angle
Distance

- In the next section on Euclidean distance, we'll look into how we can perform comparisons between vector representations using **cosine similarity and the Euclidean distance** in order to get the angle and distance between them.
- **Key takeaways**
  - You can get vector spaces using two different designs: word by word and word by document.
    - **Word by Word:** count the co-occurrence of words, i.e., the number of times two words occur together within a certain distance kk.
    - **Word by Document:** count the number of times a word occurs within each category of documents.
  - In vector spaces, you can determine the "closeness" or relationships between different types of documents using similarity.

Euclidean Distance

- In vector space, you can find relationships between words and vectors, such as a measure of their **similarity**.
- Euclidean distance is a similarity metric that allows you to **compare two vectors**. This metric allows you to identify how far two points or two vectors are apart from each other.
- Let's use 2 of the corpora vectors we saw in the previous section on word by document design. Recall that in that example, there were two dimensions: the number of times that the word *data* and the word *film* appeared in the corpus. In the

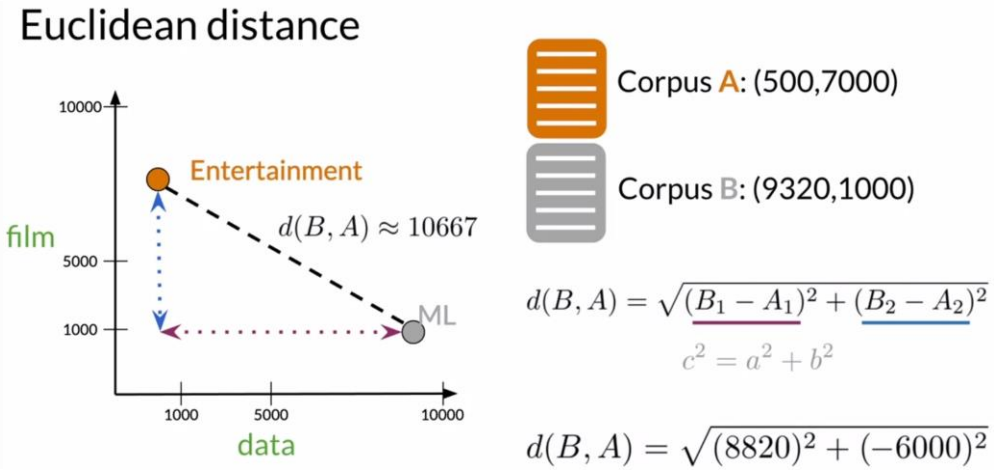
figure below, corpus A is be the entertainment corpus and corpus B is the machine learning corpus.



- Now let's represent those vectors as points in the vector space.
- The Euclidean distance is the **length of the straight line segment connecting the two points**. To obtain the Euclidean distance between two points P_1P1 and P_2P2 having co-ordinates (x_1,y_1)(x1,y1) and (x_2,y_2)(x2,y2), the formula is as follows:

$$d(P_1,P_2)=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$ d(P1,P2)=(x2−x1)2+(y2−y1)2−−−−−−−−−−−−−−−−−−−−−−√

- The first term is the **horizontal distance squared** and the second term is the **vertical distance squared**.
- Looking deeper, we can observe that this formula follows from the Pythagorean theorem, as shown in the figure below. Using the values in the figure below in the equation to find the distance between corpora AA and BB, you will arrive at the expression $\sqrt{(8820)^2+(-6000)^2}=10667$(8820)2+(−6000)2−−−−−−−−−−−−−−−−−−−−−−√=10667. 0667.



- Now, let's generalize the intuition we've developed to vector spaces in higher dimensions.
- The Euclidean distance **extends well to higher dimensions**. To get the Euclidean distance between two n-dimensional vectors:
  - Get the **difference** between **each of their dimensions**.
  - **Square** those differences.
  - **Sum** them up.
  - Get the **square root** of your results.
- This process is simply the generalization of the one we saw earlier with 2 dimensions.
- Formally, the Euclidean distance between vector representations is given by the following formula:

$$d(v,w)=d(w,v)=\sqrt{(v_1-w_1)^2+(v_2-w_1)^2+\ldots+(v_n-w_n)^2}=\sqrt{\sum_{i=1}^{n}(v_i-w_i)^2}$$

  - where,
    - v and w are the word vectors whose distance is being calculated.
    - n is the number of elements in the vector (referred to as an n-dimensional vector space).
- The more similar the words, the more likely the Euclidean distance will be close to 0.
- Recall that in linear algebra, this formula is known as the **norm of the difference** between the points/vectors that you are comparing.
- Let's walk through an example using the co-occurrence matrix shown in the figure below.

	data	boba ($\vec{w}$)	ice-cream ($\vec{v}$)
AI	6	0	1
drinks	0	4	6
food	0	6	8

$$= \sqrt{(1-0)^2 + (6-4)^2 + (8-6)^2}$$

$$= \sqrt{1+4+4} = \sqrt{9} = 3$$

Euclidean formula!

$$d(\vec{v},\vec{w}) = \sqrt{\sum_{i=1}^{n}(v_i - w_i)^2} \longrightarrow \text{Norm of } (\vec{v}-\vec{w})$$

- Assuming v and w to be the vector representations of the words *ice cream* and *boba*, the Euclidean distance between the two vectors is:

$$d(v,w)=\sqrt{(1-0)^2+(6-4)^2+(8-6)^2}=\sqrt{1+4+4}=\sqrt{9}=3$$

- Let's take a look at the implementation of the Euclidean distance in Python. We can use the **linalg module** from numpy `np.linalg.norm(a-b)` to get the norm of the difference between two vectors a and b. Note that the norm function works for **n-dimensional spaces**.

```
# Create numpy vectors v and w
v = np.array([1, 6, 8])
w = np.array([0, 4, 6])

# Calculate the Euclidean distance d
d = np.linalg.norm(v-w)
# Print the result
print("The Euclidean distance between v and w is: ", d)

The Euclidean distance between v and w is: 3
```

numpy 으로 calculation

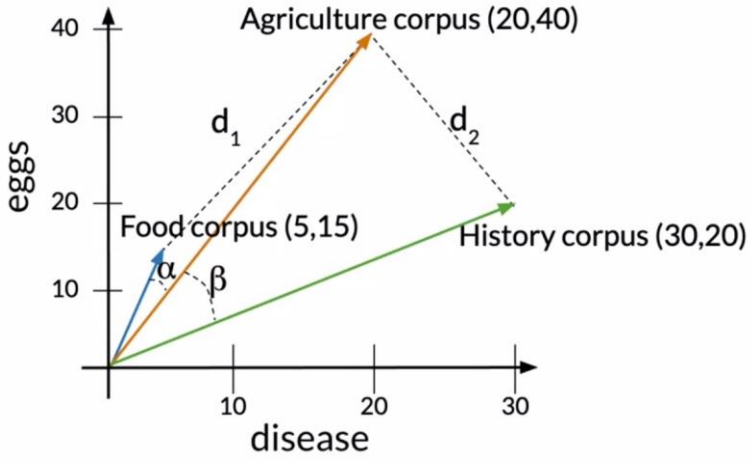
- **Key takeaways**
  - Euclidean distance is the length of the straight line that connects any two points, i.e., two vectors in vector space.
  - To get the Euclidean distance, calculate the **norm of the difference** between the vectors being compared. By using this metric, you can get a sense of how **similar two documents or words are**.
  - Euclidean distance can however be misleading when the corpora have **different sizes**.

Cosine Similarity

Intuition

- The cosine similarity is another similarity function. Using the cosine of the inner angle between the two vectors, it tells you whether two vectors are similar or not.
- When comparing vector representations of **corpora of different sizes**, Euclidean distance can **misrepresent** the measure of similarity between two vectors. The cosine similarity metric can help overcome this problem.

- To illustrate a scenario where the euclidean distance can prove to be problematic, let's consider the following example.
- Suppose that you are in a vector space where the corpora are represented by the occurrence of the words *disease* and *eggs*. Shown below in the figure is a representation of the *food*, *agriculture* and *history* corpus. Each one of these corpora have text related to that subject.
- However, the corpora differ in the sense that their **word totals** have a **significant mismatch**. Specifically, the *agriculture* and the *history* corpus have a similar number of words, while the *food* corpus has a relatively small number.



Euclidean distance: $d_2 < d_1$

Angles comparison: $\beta > \alpha$

The cosine of the angle between the vectors

- Let's define the euclidean distance between the *food* and the *agriculture* corpus as d_1d1 and that between the *agriculture* and the *history* corpus be d_2d2.
- As you can see, the distance d_2d2 is smaller than the distance d_1d1, which suggests that the *agriculture* and *history* corpora are more similar than the *agriculture* and *food* corpora, which **intuitively doesn't make sense**.
- Enter cosine similarly, which is a common method for determining the similarity between vectors by computing the **cosine of their inner angle**.
  - If the angle is small, the cosine tends to 1. As the angle approaches 90°90∘, the cosine tends to 0. This implies that larger the angular delta between the two vectors, lesser the similarity.
  - In our particular example shown in the figure above, the angle αα between *food* and *agriculture* is **smaller** than the angle ββ between *agriculture* and *history*, indicating that *food* and *agriculture* are **more similar** than *agriculture* and *history*, which gels well with intuition.
- Thus, in this case, the cosine of those angles is a **better proxy of similarity** between these vector representations than their euclidean distance.

Background

Norm of a Vector

- Recall from linear algebra that the **norm** (or magnitude) of an n-dimensional vector v is the **square root of the sum of its elements squared**:

$$norm(\vec{v}) = ||\vec{v}|| = \sqrt{\sum_{i=1}^{n} v_i^2} \quad \text{norm}(v\rightarrow) = ||v\rightarrow|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Dot-product of Two Vectors

- The **dot product** (or scalar product or inner product) between two n-dimensional vectors v and w is the **sum of the products between their elements in each dimension of the vector space**:

$$\vec{v} \cdot \vec{w} = \sum_{i=1}^{n} v_i \cdot w_i \quad v\rightarrow \cdot w\rightarrow = \sum_{i=1}^{n} v_i \cdot w_i$$

  - Note that the two vectors need to be of the **same size** for their dot product to be defined.
  - Also, note that the dot product of two vectors yields a scalar, i.e., **a single number**.
- Finally, note that the norm of a vector can be written as the **square root of the dot product of the vector with itself**, as shown below:

$$norm(\vec{v}) = ||\vec{v}|| = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{v \cdot v} \quad \text{norm}(v\rightarrow) = ||v\rightarrow|| = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{v \cdot v}$$

- For a detailed discussion on the geometric interpretation of the dot product and different ways of calculating it, refer "Engineering Math - Quick Reference: Inner Product/Dot Product".

Implementation

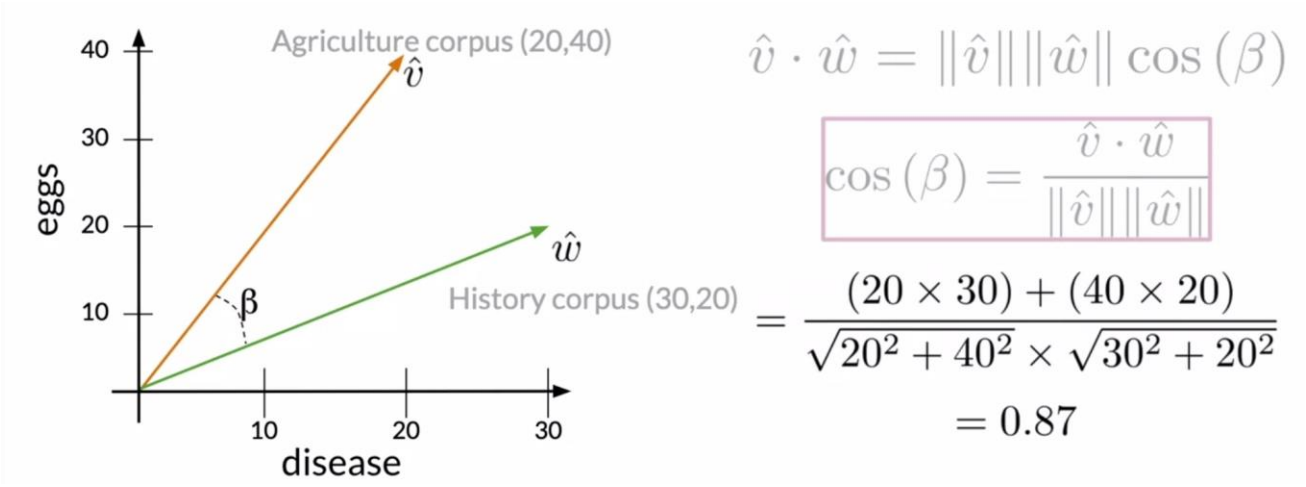
- Let's explore how we may calculate the cosine similarity metric using the cosine of the inner angle of two vectors.
- Using the example we discussed in the section on intuition, let's delve into the similarity of the *agriculture* and *history* corpora. Recall that in this example, you have a vector space where the representation of the corpora is given by the number of occurrences of the words *disease* and *eggs*.
- The angle between those vector representations is denoted by β. The agriculture corpus and the history corpus is represented by the vector v and w respectively. The dot products between vectors v and w is defined as follows:

$$v\hat{} \cdot w\hat{} = ||v\hat{}|| \cdot ||w\hat{}|| \cdot \cos(\beta) \Rightarrow \cos(\beta) = \frac{v\hat{} \cdot w\hat{}}{||v\hat{}|| \cdot ||w\hat{}||} \therefore \cos(\beta) = \frac{\sum_{i=1}^{n} v_i w_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \sqrt{\sum_{i=1}^{n} w_i^2}}$$

$$v\hat{} \cdot w\hat{} = ||v\hat{}|| \cdot ||w\hat{}|| \cdot \cos(\beta) \Rightarrow \cos(\beta) = \frac{v\hat{} \cdot w\hat{}}{||v\hat{}|| \cdot ||w\hat{}||} \therefore \cos(\beta) = \frac{\sum_{i=1}^{n} v_i w_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \sqrt{\sum_{i=1}^{n} w_i^2}}$$

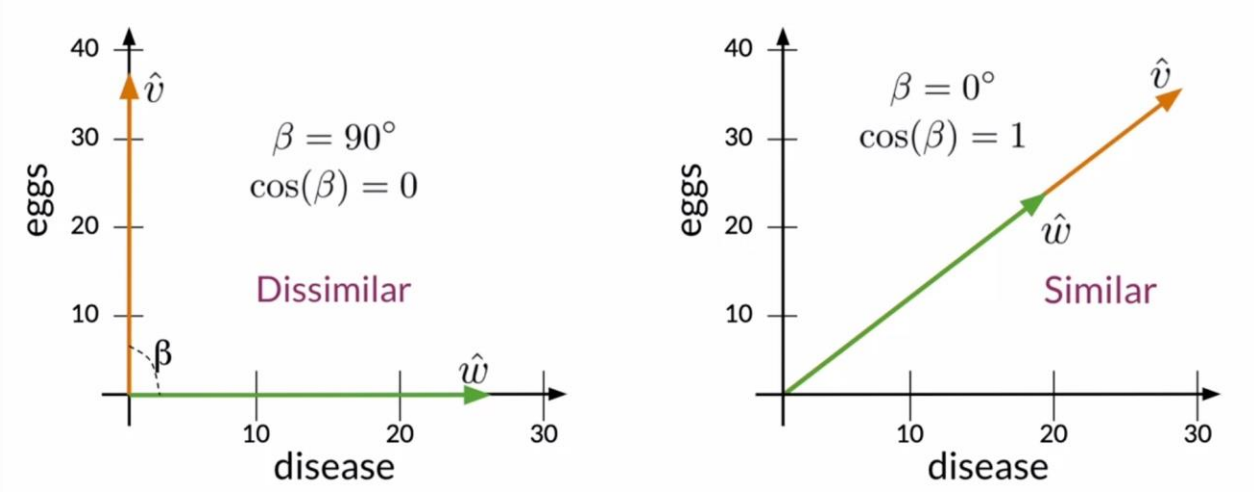
- where v and w represent the word vectors and v_i or w_i represent index i of the respective vector.

- Interpreting the output of cosine similarity depending on the **angle between vectors** v and w:
  - If the vectors are **identical**, you get $\cos(\theta)=\cos(0)=1$.
  - If the vectors are **opposites**, i.e., $A=-B$, you get $\cos(\theta)=\cos(180^\circ)=-1$.
  - If the vectors are **orthogonal (or perpendicular)**, you get $\cos(\theta)=\cos(90^\circ)=0$.
- Also, interpreting the output of cosine similarity depending on the **magnitude of the output**:
  - An output $\in[0,1]$ indicates a **similarity** score.
  - An output $\in[-1,0]$ indicates a **dissimilarity** score.
- Thus, the cosine of the inner angle β between the two vectors is given by the dot product between the vectors v and w divided by the product of the two norms.
- Replacing the actual values from the vector representations should give you this expression in the figure below.
  - In the numerator, you have the product between the occurrences of the words, *disease* and *eggs*.
  - In the denominator, you have the product between the norm of the vector representations of the *agriculture* and *history* corpora.
  - Ultimately, you obtain a cosine similarity of 0.87.



- Next, let's delve into how the value of the cosine similarity is related to the similarity of two vectors, as shown in the figure below.
  - Consider the case where two vectors are **orthogonal** in a vector space. Note that it is only possible to have positive values for any dimension, which implies that the maximum angle between pair of vectors is 90°. In this case, the cosine would

be equal to 0, and it would mean that the two vectors have **orthogonal directions** or that they are **maximally dissimilar**.
    - Now, let's look at the case where the vectors have the **same direction**. In this case, the angle between them is $0\circ 0\circ$ and the cosine is equal to 1, because the cosine of 0 is 1.
    - Thus, as the cosine of the angle between two vectors approaches 1, the closer their directions are.



- Cosine similarity outputs values $\in[0,1]$∈[0,1].
- To calculate the cosine similarity between vectors aa and bb using numpy:

```
import numpy as np

# Create numpy vectors v and w
a = np.array([1, 0, -1, 6, 8])
b = np.array([0, 11, 4, 7, 6])

cosine_similarity = np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

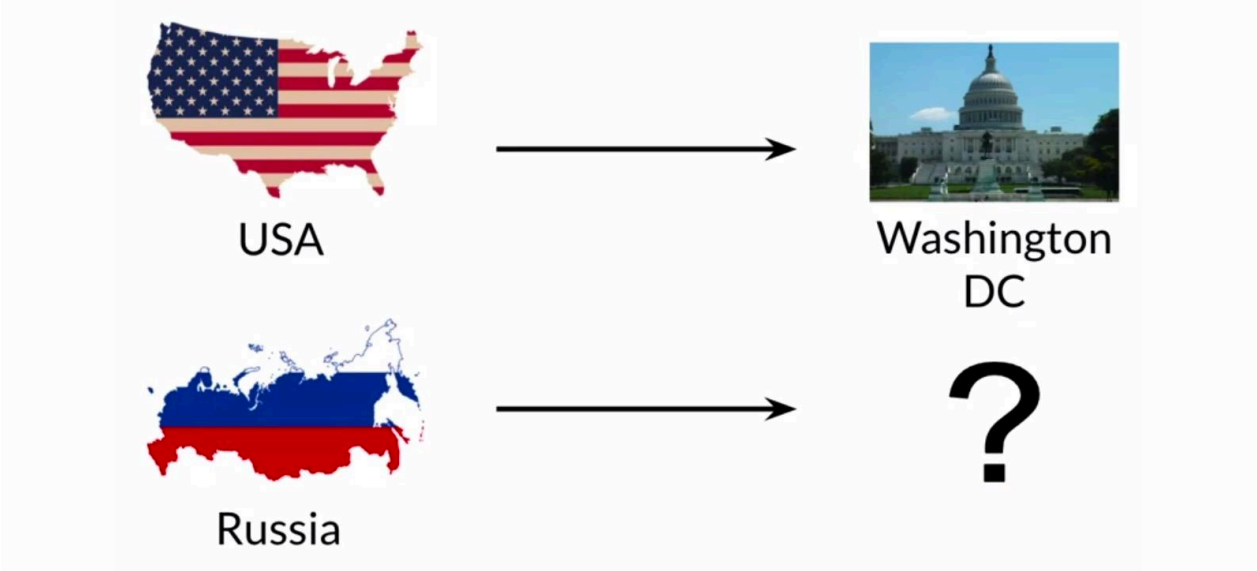
- **Key takeaways**
    - The cosine similarity between a pair of vectors is proportional to the similarity between the directions of the vectors that you are comparing.
    - **(+)** The intuition behind the use of the cosine similarity as a metric to compare the similarity between two vector representations is that it isn't **biased** by the **size difference** between the representations. This is the main advantage of the cosine similarity metric over euclidean distance.
    - For two vectors v,wv,w the cosine of their inner angle ββ is given by:

```
np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b)
```

- Cosine similarity outputs values $\in[0,1]$∈[0,1]. On the other hand, an output $\in[-1,0]$∈[−1,0] indicates the degree of **dissimilarity**.

Word Manipulation in Vector Spaces

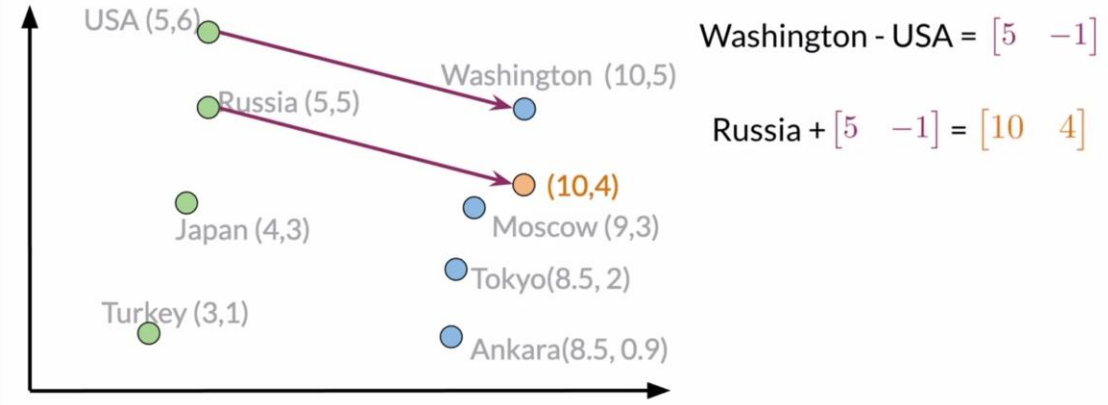
- Let's perform vector manipulation using simple vector arithmetic, i.e., by adding vectors and subtracting vectors. We'll apply the idea of manipulate vector representations to the task of inferring unknown relationships among words, such as predicting the countries of certain capitals.
- Suppose that you have a vector space with countries and their capital cities. While you know that the capital of the United States is *Washington DC*, you don't know the capital of *Russia*, but you would like to use the **known relationship** between *Washington DC* and the *USA* to figure it out. Vector manipulation involves using some simple vector algebra to carry out such tasks.



- For this particular example, let's assume that you are in a hypothetical two-dimensional vector space that has different representations for different countries and capitals cities.
- First, you will have to find the relationship between the *Washington DC* and *USA* vector representations. In other words, we need to figure out the vector that connects *Washington DC* and *USA*. To do so, get the **difference** between the

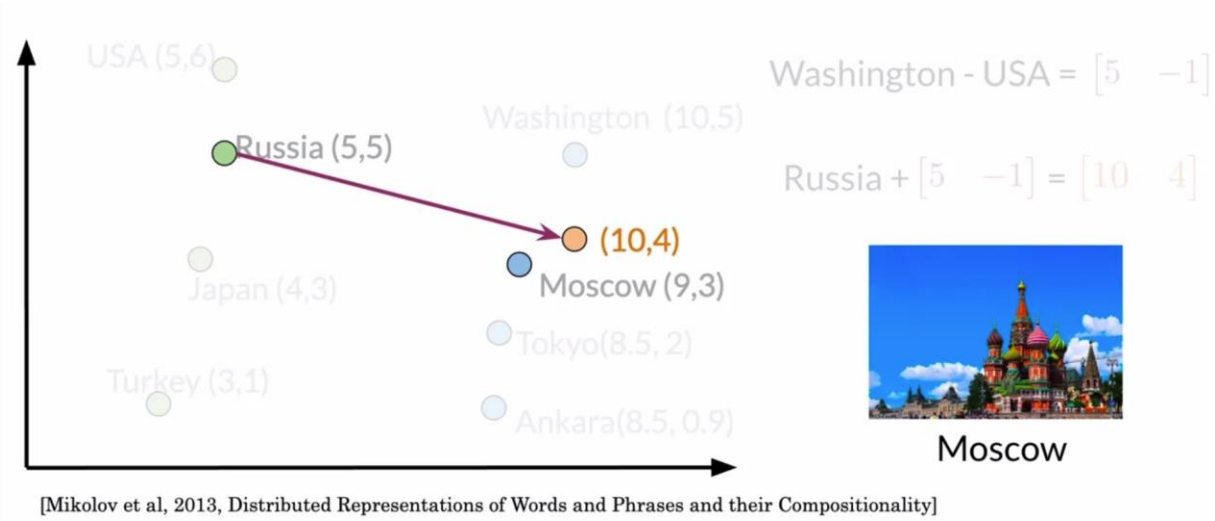
two **vectors**. This vector will tell you how many **units on each dimension** you'll need to move in order to find a country's capital in that vector space.

- To find the capital city of *Russia*, all you have to do is to sum it's vector presentation with the difference vector.
- You thus deduce that the capital of *Russia* has a vector representation of $(10,4)$(10,4).



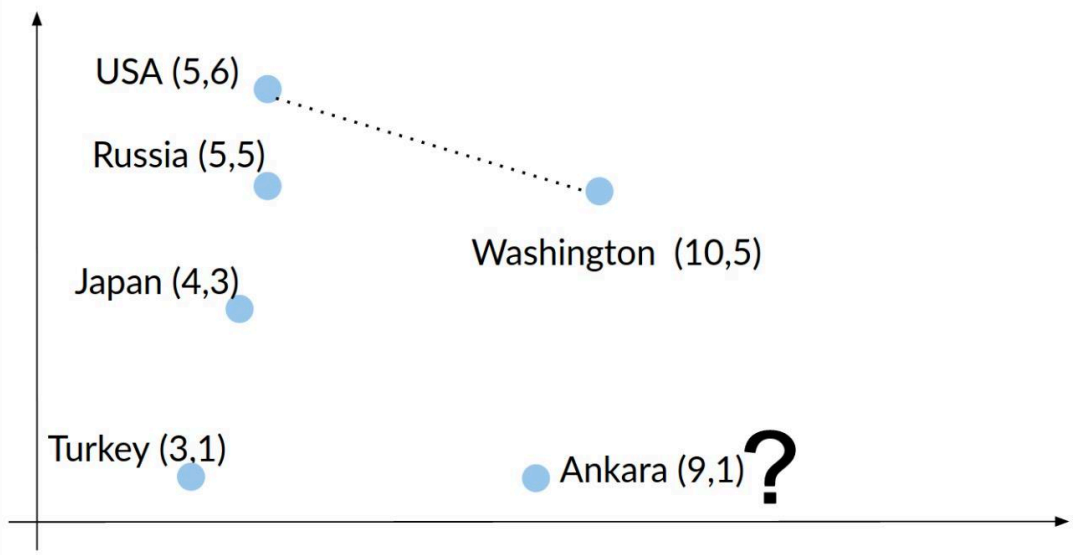
[Mikolov et al, 2013, Distributed Representations of Words and Phrases and their Compositionality]

- However, since here are no cities with that representation, you'll have to take the one that is the **closest** to it by comparing each vector with the Euclidean distances or cosine similarities. In this case, the vector representation that is closest to $(10,4)$(10,4) is the one for *Moscow*.
- Thus, using this simple process, we could predict the capital of *Russia* knowing the capital of the *USA*. The only **requirement** here is that you need a **vector space** where the representations capture the **relative meaning** of words.



[Mikolov et al, 2013, Distributed Representations of Words and Phrases and their Compositionality]

- Let's take another example. Using the same method, let's predict the country whose capital is *Ankara*. Turkey is the answer with an Euclidean distance d between your prediction vector and the correct vector as 1.41.



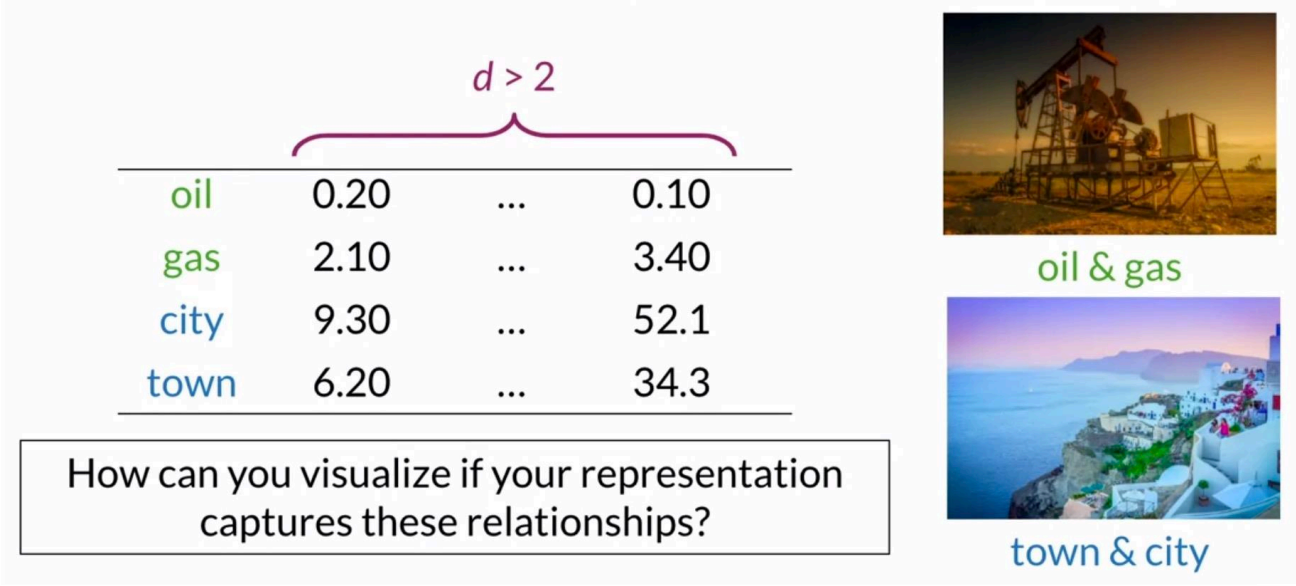
- **Key takeaways**
  - **(+)** Vector manipulations enable deriving **unknown relationships** between words using **known relationships**. This speaks to the importance of vectors spaces where the representations of words capture the **relative meaning** in natural language.
  - With the above vector representations, we've explored the clustering of all vectors when plotted in a 2D setting, where the vectors of the words have similar **contexts** are encoded **similarly**.
  - This type of encoding consistency lends itself to **identifying patterns**.
    - For e.g., to find the closest words to the word *doctor* by computing cosine similarity, you'll come across *nurse, cardiologist, surgeon,* etc.

PCA

Visualization of Word Vectors

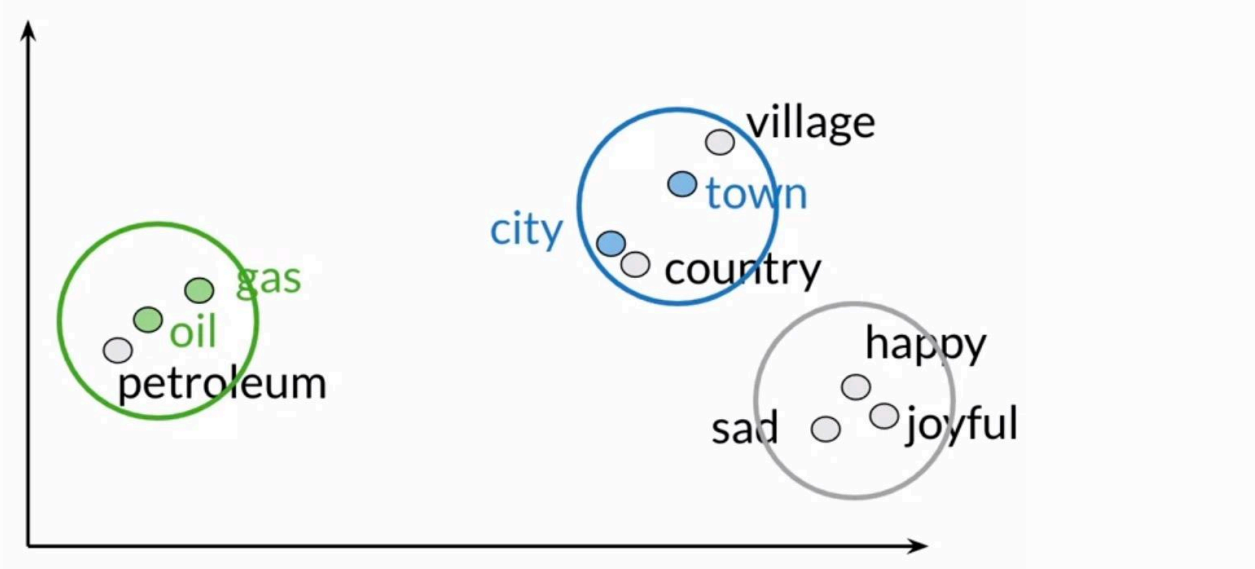
- Often, you'll end up having vectors in very high dimensions. While operating on such vectors in very high dimensional spaces might be computationally feasible, it is **impossible to visualize results** in such high dimensional spaces. To visualize word vectors on an XY plane in order to identify relationships between words, we'll need a way to reduce its dimension-space to 2 dimensions.

- Principal Component Analysis (PCA) allows you to do so. PCA enables visualization of vector representations with high dimensions by **projecting** them onto a **lower-dimensional space**.
- Let's begin by motivating the idea of visualizing vector presentation of words. Shown below is an example of vector representation for some words in a vector space.

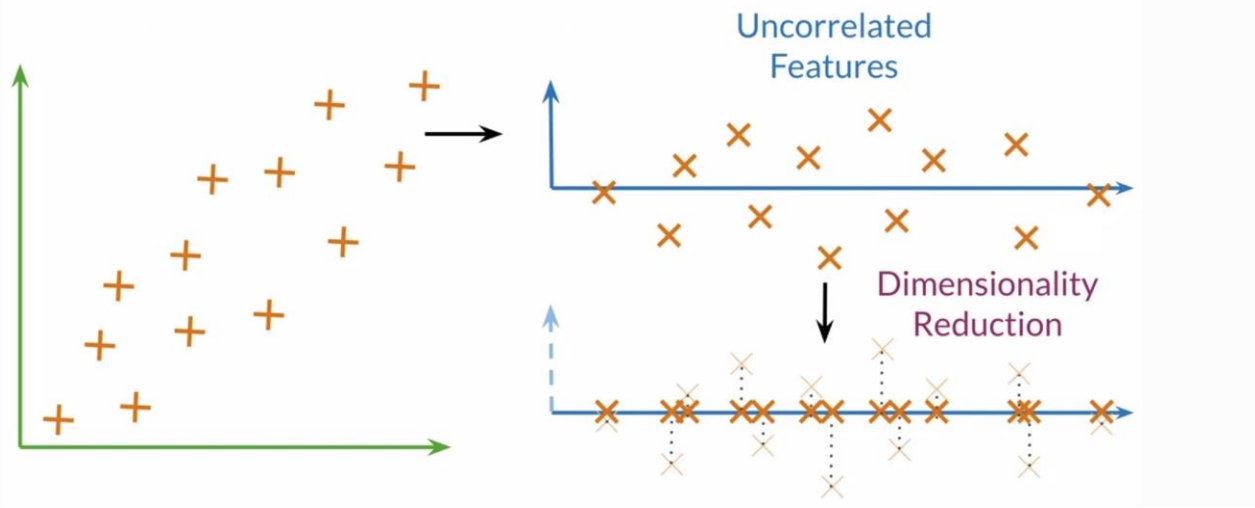


- To represent a set of words in the 2D space that currently exist in a high dimensional space, **dimensionality reduction** is of essence. PCA enables you to get a representation in a vector space with fewer dimensions. To ease the process of visualization of data, you can get a reduced representation with fewer features (say, 2 or 3).

- Applying PCA on your data can get you a 2D representation, which you can then use to plot an XY visual of your words. In our particular example, you might find that your initial representation captured the relationship between the words *oil*, *gas*, *city* and *town* since they appear to be **clustered** with related words in your two-dimensional space. You can even find other relationships among your words that you didn't expect, which is a fun and useful possibility!



- Let's look into a quick overview on how PCA works. We'll begin with a 2D vector space for the sake of simplicity. Say that you want your data to be represented by one feature instead.
  - Using PCA, you'll find a set of uncorrelated features.
  - These uncorrelated features are then projected to a one dimensional space, while trying to retain as much information as possible. In this case, by retaining maximum information we mean that the **Euclidean distance** between the **original vectors** and their **projected siblings** is minimal. Hence, vectors that were originally close in the embeddings dictionary will produce lower dimensional vectors that are **still close** to each other.



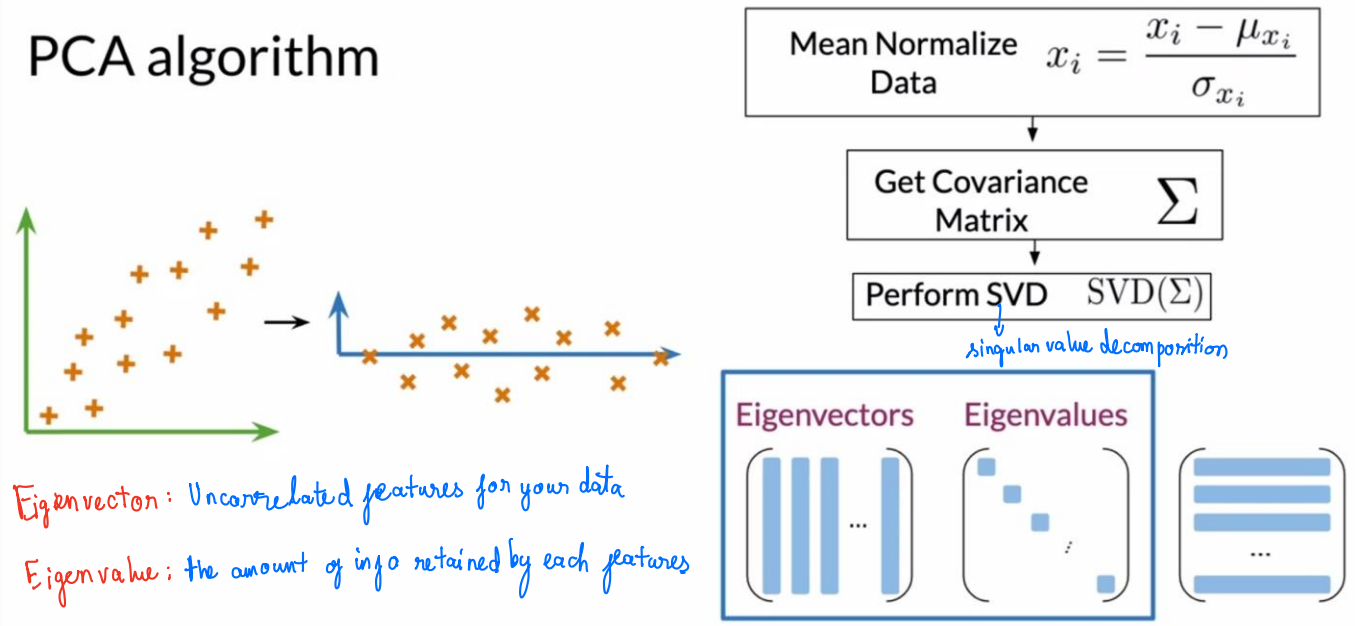
- **Key takeaways**
  - PCA is an algorithm used for dimensionality reduction that can find **uncorrelated features** in your data. It's helpful for visualizing your data to check if your representation is capturing **relationships** among words.

Original Space → Uncorrelated features → Dimension reduction

- After obtaining uncorrelated features, PCA projects your data representation to a lower dimension while **retaining as much information as possible**.

Implementation

- Let's learn about the underlying concepts behind PCA, namely **eigenvectors and eigenvalues from the covariance matrix of your data**, and how they help to carry out dimensionality reduction of your features.
- Recall the concepts of eigenvectors and eigenvalues of matrices from algebra. While the exact implementation details behind eigenvectors and eigenvalues are irrelevant for the purposes of our discussion, the point to note is that PCA utilizes these concepts under-the-hood. Once the covariance matrix is obtained from the original data, the **eigenvectors** yield directions of uncorrelated features, while the eigenvalues represent the **variance of your data** on each of those new features.
- Now let's delve into the implementation process.
- **Get uncorrelated features**
  - To get a set of uncorrelated features, you will need to normalize your data, get the covariance matrix, and finally, perform Singular Value Decomposition (SVD) on the covariance matrix to get a set of 3 matrices. Note that SVD is already implemented in popular numerical libraries such as NumPy so we can abstract away its implementation details for now.
  - The first of those matrices contains the eigenvectors stacked as column vectors. The second one has the eigenvalues on the diagonal, as shown graphically below:

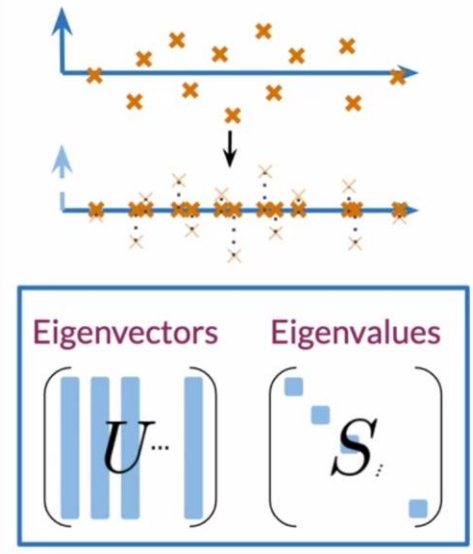


- **Project uncorrelated features to a lower-dimensional vector space**

- The next step is to project your data to a new set of features. This step involves using the eigenvectors and eigenvalues obtained in the previous step.
- Let's denote the eigenvectors with U and the eigenvalues with S.

3. **Dot-products between your data and eigenvectors**: perform dot products between the matrix containing your word embeddings and the first n columns of the U matrix, where n denotes the desired number of dimensions. For visualization, it is common practice to have 2 or 3 dimensions.
4. **Variance in the new vector space**: Using the eigenvalues, you'll get the percentage of variance retained in the new vector space.

- As an important side note, for the purposes of **dimensionality reduction**, the **principal components** contained in the rotation matrix, i.e., the eigenvectors, should be sorted in **descending order** of the explained variance, i.e., the eigenvalues. In other words, the features with the highest variance should appear first.
  - This implies that the first components retain most of the power to explain the **patterns that generalize the data**.
  - This also ensures that you retain as much information as possible from your original embedding.
  - Nevertheless, for other opposing applications such as **novelty detection**, we are usually interested in the patterns that explain much less variance, in which case the **principal components** are usually sorted in the **ascending order** of the explained variance.
  - Note that most numerical libraries sort these matrices for you.



Dot Product to Project Data $X' = XU[:, 0:2]$

Percentage of Retained Variance $\frac{\sum_{i=0}^{1} S_{ii}}{\sum_{j=0}^{d} S_{jj}}$

- t-SNE is another visualization method, that is particularly well suited for the visualization of high-dimensional datasets. It is extensively applied in image processing, NLP, genomic data and speech processing. Read more on it in [CS231n | Visualizing and Understanding].
- **Key takeaways**
  - **Goals**

        - **(+)** PCA enables high-dimensional data visualization by carrying out dimensionality reduction to represent data in a **reduced space of uncorrelated variables**, say on a 2D plane.
    - **Process**
        - At a top level, PCA involves the following steps:
            - **Get uncorrelated features:** Using the word representations in your original vector space, get **uncorrelated features**.
            - **Project uncorrelated features to a lower-dimensional vector space:** Reduce the dimensionality of your data by projecting your data to a vector space with the **desired number of features**, while trying to retain as much information as possible from your original embedding.
    - **Concepts**
        - PCA is based on the **Singular Value Decomposition (SVD)** of the **covariance matrix** of the original high dimensional dataset.
        - Eigenvectors from the "decomposed" covariance matrix contain the principal components, which are encoded as a rotation matrix that contains the directions of the uncorrelated features.
        - Eigenvalues associated with the aforementioned eigenvectors represent the "explained variance" of each principal component, i.e., variance of your data on the new features. In other words, they signal the amount of information retained by each new feature.
    - **Implementation**
        - **Get uncorrelated features**
            0. Mean normalize data by operating on **each feature** separately (across the entire dataset).
            1. Compute the covariance matrix.
            2. Perform SVD on the covariance matrix to get the eigenvectors and eigenvalues.
        - **Project uncorrelated features to a lower-dimensional vector space**
            0. The dot product of your word embeddings and the eigenvectors matrix will project your data onto a new vector space of the desired dimension.
            1. Compute the amount of retained variance (in %) using the eigenvalues.

Putting It Together with Code

- Let's string together our learnings into a program performs PCA on a dataset where each row corresponds to a word vector in a high dimensional space (say, $300$300). This implies that the dimensions of the dataset would be (m,n)(m,n), where mm are the number of observations, i.e., the words in the dataset and $n=300$n=300 are the number of features/dimensions per observation/word.
- The goal is to use PCA to transform $300 \rightarrow n_components$300→n_components dimensions.

- The shape of the "reduced" matrix should be $(m,n\_components)$(m,n_components).
- Here are the steps involved in performing PCA on our dataset:

1. Mean normalize (also referred to as de-meaning the data).
   - Use `np.mean(a,axis=None)`. Recall that each row is a word vector, and the number of columns are the number of dimensions in a word vector.
   - If you set `axis = 0`, you take the mean for each column. If you set `axis = 1`, you take the mean for each row.
2. Get the covariance matirx.
   - Use `np.cov(m, rowvar=True)`. This calculates the covariance matrix. By default, `rowvar` is True.
     - From the NumPy documentation: "If `rowvar` is `True` (default), then each row represents a variable, with observations in the columns."
   - Note that in our case, each row is indeed a vector observation for a particular word, and each column is a feature for that word, which is why `rowvar` is set to `True`.
3. Get the eigenvalues using `np.linalg.eigh()`. Use `eigh` rather than `eig` since R is symmetric. The performance gain when using `eigh` instead of `eig` is substantial.
   - Use `np.linalg.eigh(a, UPLO='L')`.
4. Sort the eigenvectors and eigenvalues by decreasing order of the eigenvalues.
   - Use `np.argsort()` to sort the values in an array from smallest to largest, then returns the indices from this sort.
   - In order to reverse the order of a list, you can use: `x[::-1]`.
5. Get a subset of the eigenvectors (choose how many principal components you want to use using `n_components`).
   - To apply the sorted indices to eigenvalues, you can use `x[indices_sorted]`.
   - When applying the sorted indices to eigenvectors, note that each column represents an eigenvector. In order to preserve the rows but sort on the columns, you can use `x[:, indices_sorted]`.
6. Return the transformed dataset by multiplying the eigenvectors with the original data.
   - To transform the data using a subset of the most relevant principal components, take the matrix multiplication of the eigenvectors with the original data.
   - The shape of the original data is (m,n)(m,n). The subset of eigenvectors are in a matrix of shape $(n,n\_components)$(n,n_components).
   - To multiply these together, take the transposes of both the eigenvectors $(n\_components,n)$(n_components,n) and the data (n,m)(n,m). The product of these two has dimensions $(n\_components,m)$(n_components,m).
   - Finally, take its transpose to obtain $(m,n\_components)$(m,n_components), which is the desired shape.

```
def compute_pca(X, n_components=2):
    """
    Input:
        X: of dimension (m, n) where each row corresponds to a word vector
        n_components: Number of components you want to keep.
    Output:
        X_reduced: original data transformed into  dims/columns + regenerated
    """

    # mean center the data
    X_demeaned = X - np.mean(X, axis=0)

    # calculate the covariance matrix
    covariance_matrix = np.cov(X_demeaned, rowvar=False)

    # calculate eigenvectors & eigenvalues of the covariance matrix
    eigen_vals,   eigen_vecs   =   np.linalg.eigh(covariance_matrix, UPLO='L')

    # sort eigenvalue in increasing order (get the indices from the sort)
    idx_sorted = np.argsort(eigen_vals)

    # reverse the order so that it's from highest to lowest.
    idx_sorted_decreasing = idx_sorted[::-1]

    # sort the eigenvalues by idx sorted decreasing
    eigen_vals_sorted = eigen_vals[idx_sorted_decreasing]

    # sort eigenvectors using the idx_sorted_decreasing indices
    eigen_vecs_sorted = eigen_vecs[:,idx_sorted_decreasing]

    # select the first n eigenvectors (n is desired dimension
    # of rescaled data array, or dims_rescaled_data)
    eigen_vecs_subset = eigen_vecs_sorted[:, 0:n_components]

    #  transform  the  data  by  multiplying  the  transpose  of  the eigenvectors
    # with the transpose of the de-meaned data
    # Then take the transpose of that product.
    X_reduced = np.dot(eigen_vecs_subset.T, X_demeaned.T).T
```

```
    return X_reduced
```

Document As a Vector

- To represent a document as a vector, simply add up the word vectors for its constituent words.
- Programmatically:

```
import numpy as np

word_embeddings = {
                    "I":        np.array([1,  0,  1]),
                    "love":     np.array([-1, 0,  1]),
                    "learning": np.array([1,  0,  1])
                  }
words_in_document = ['I', 'love', 'learning', 'not_a_word']

document_embedding = np.array([0, 0, 0])
for word in words_in_document:
    document_embedding += word_embeddings.get(word, 0)

print(document_embedding) # Prints [1 0 3]
```

Week 4: Building a Machine Translation System using Locality Sensitive Hashing

Overview

- In Vector Spaces, we learned about:
  - Word vectors and how they capture relative meanings of words.
  - Representing a document as a vector by adding up the vector representations for its constituent words.
- In this topic, we'll build a **simple machine translation system** that makes use of **locality sensitive hashing** to improve the performance of **nearest neighbor search**.

Background: Frobenius Norm

- The Frobenius norm measures the magnitude of a matrix. It is a generalization of the vector norm for matrices.
- Formally, the Frobenius norm of a matrix AA is given by,

$$\|A\|_F \equiv \sum_{i=1}^{m} \sum_{j=1}^{m} |a_{ij}|^2 \quad \|A\|F \equiv \sum i=1m \sum j=1m |aij|2$$

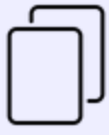
- Interpreting the above equation, the Frobenius norm can be obtained by taking the elements in the matrix, squaring them, adding them up and taking the square root of the result.
- Let's take an example. Consider a matrix A:

$$A = [2222] \quad A=[2222]$$

- To calculate its norm,

$$\|A\|_F = 2^2+2^2+2^2+2^2 \quad \sqrt{\|A\|_F = 4} \quad \|A\|F=22+22+22+22 \quad \|A\|F=4$$

- Programmatically, here's the Frobenius norm in code:

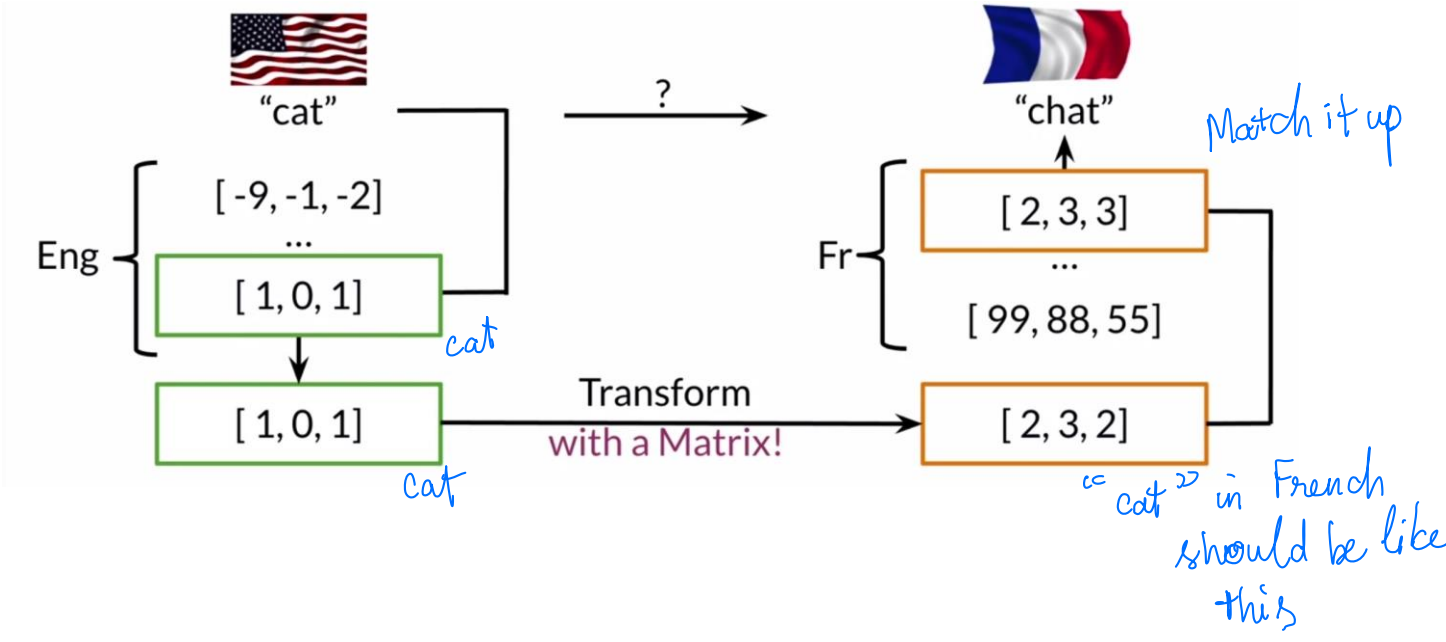


```
import numpy as np

# Start with a matrix A
A = np.array([[2, 2], [2, 2]])
# Use np.square() to square all the elements
A_squared = np.square(A)
```

```
print(A_squared)    # Prints [[4, 4],
                    #         [4 ,4]]

# Use np.sum(), then np.sqrt() to get 4.
A_Frobenius = np.sqrt(np.sum(A_squared))
print(A_Frobenius) # Prints 4.0
```

Machine Translation: Transforming Word Vectors

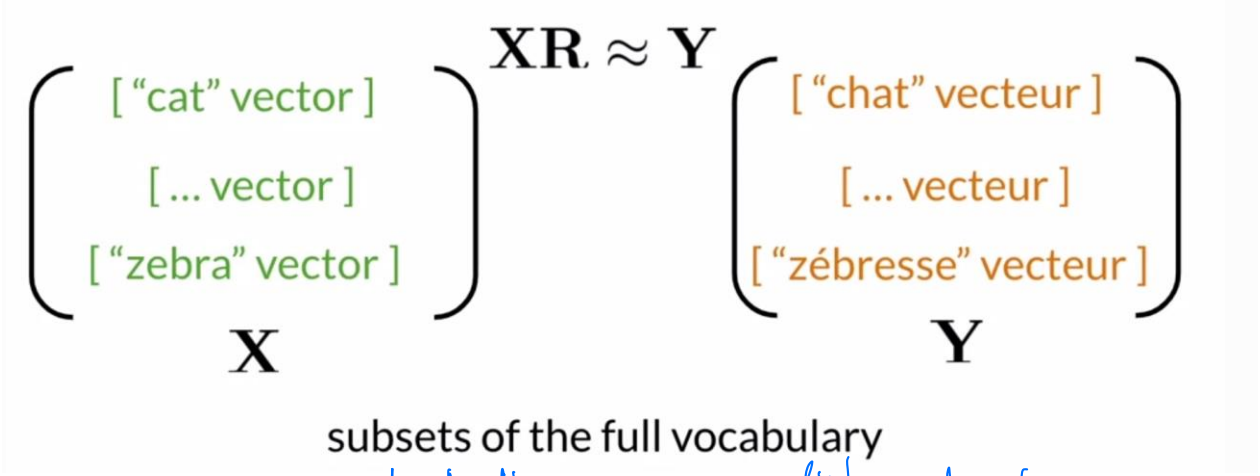
- To build our first basic translation program, let's use word vectors to to align words in two different languages. Later, we'll explore locality sensitive hashing to speed things up!
- To get an overview of machine translation, let's starting with an example of English →→ French translation. In order to translate an English word to a French word, a naive way would be to generate an **extensive list** of English words and their associated **French** word.
- For a **human** to accomplish this task, you would need to find someone who knows **both languages** to start making a list.
- For a **machine** to accomplish this task, you'll need to perform the following steps:
  - Calculate word embeddings for words in both your English and French vocabulary.
  - Retrieve the word embedding of a particular English word such as *cat*, and find a way to transform the English word embedding into a word embedding that has a meaning in the French word vector space (more on this below).
  - Take the transformed word vector and search for word vectors in the French word vector space that are similar to it. These similar words are **candidates** for your translation.
  - If your machine does a good job, it may find the word *chat*, which is the French word for *cats*.



- Putting these ideas together with code:

```
import numpy as np

R = np.array([[2, 0], [0, -2]]) # Define the matrix R
x = np.array([[1, 1]])          # Define the vector x
prod = np.dot(x, R)              # Multiply x by R using np.dot ((1 x 2) x (2 x 2)) to yield a two dimensional vector.
print(prod)                      # Prints [[2, -2]]
```

- Let's figure out how to find a transformation matrix R that can transform English word vectors into corresponding French word vectors. We'll start with a randomly initialized matrix R and evaluate it by using it translate English vectors, denoted by matrix X, and compare that to the actual French word vectors, denoted by matrix Y.
- In order for this to work,
  - Get a subset of English words and their French equivalents.
  - Get their respective word vectors.
  - Stack the word vectors in their respective matrices X and Y.
    - Note that the key here is to make sure the English and French word vectors are **aligned**. For e.g., if the first row of matrix X contains the word *cat*, the first row of the matrix Y should contain the French word for *cat*, which is *chat*.
- Keep in mind that the **broader goal** here is to use a subset of these words to find your transformation matrix R which can be used to translate words that are **not part of your original training set**. We thus only need to train on a **subset of the English-French vocabulary** and not the entire vocabulary.



X : corresponds to the matrix of english word vectors

Y : corresponds to the matrix of French word vectors

R : is the mapping

Steps required to learn R:

- Initialize R
- For loop

$$Loss = \|XR - Y\|_F$$

$$g = \frac{d}{dR} Loss$$

$$R = R - \alpha * g$$

Here is an example to show you how the frobenius norm works.

$$\|\mathbf{XR} - \mathbf{Y}\|_F$$
$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$
$$\|\mathbf{A}_F\| = \sqrt{2^2 + 2^2 + 2^2 + 2^2}$$
$$\|\mathbf{A}_F\| = 4$$
$$\|\mathbf{A}\|_F \equiv \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

In summary you are making use of the following:

- $\mathbf{XR} \approx \mathbf{Y}$
- minimize $\|\mathbf{XR} - \mathbf{Y}\|_F^2$

- Given that our objective is to make the transformati possible, i.e., $XR \approx YXR \approx Y$, we would need t between $XRXR$ and YY to get the optimal version of R
- Next, to find a good matrix RR we simply compare t actual French word embeddings YY. Comparison is es take $XRXR$ and subtract the YY matrix. Taking the Fr gives us the loss, interpreted as the measure of our transformation matrix RR.

$$Loss = ||XR - Y||_F$$

- As an example of calculating the Frobenius expression $XR - YXR - Y$ yields a matrix.
- Now, assume for the sake of this example that there are only two words in our vocabulary, which represents the number of rows in our matrix RR and the word embeddings have 2 dimensions, which represent the number of columns in the matrix. Thus, matrices XX, RR, YY and AA are matrices of dimensions $2 \times 22 \times 2$.
- Now, if A is:

$$A = [1111]A = [1111]$$

- The Frobenius norm of AA is $||A||_F = 1_2 + 1_2 + 1_2 + 1_2 - - - - - - - - - - - - - - - \sqrt{} = 2||A||F = 12 + 12 + 12 + 12 = 2.$
- Starting with a random matrix RR, we can iteratively improve this matrix RR using gradient descent:
  - Compute the gradient by taking the derivative of this loss function with respect to the matrix RR.
  - Update the matrix RR by subtracting the gradient weighted by the learning rate $\alpha\alpha$.
  - Either pick a fixed number of times to go through the loop or check the loss at each iteration and break out of the loop when the loss falls between a certain threshold.
- Algorithmically,
  1. Initialize RR randomly, or with zeros.
  2. In a loop:

$$Loss = ||XR - Y||_F gradient\ g = dd RLossupdate\ R = R - \alpha g Loss = ||XR - Y||F gradient\ g = dd RLossupdate\ R = R - \alpha g$$

- Note that in practice, it's easier to minimize the **square of the Frobenius norm**. In other words, we can cancel out the square root in the Frobenius norm formula by taking the square.
  - The reason behind using the square of the Frobenius norm is that its easier to take the derivative of the squared expression rather than dealing with the square root within the Frobenius norm (explained below).
- Re-visiting our example from earlier,

$$A=[2222]\|A\|_{2F}=(2_2+2_2+2_2+2_2-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\!\!-\sqrt{})_2=16\therefore Loss=\|XR-Y\|_{2F}=16A$$
$$=[2222]\|A\|F2=(22+22+22+22)2=16\therefore Loss=\|XR-Y\|F2=16$$

- Next, let's calculate the gradient of the loss function.
- As seen earlier, the loss is defined as the square of the Frobenius norm. Starting with our loss equation,

$$Loss=||XR-Y||_{2F}Loss=||XR-Y||F2$$

- The gradient is the derivative of the loss with respect to the matrix R:

$$g=ddRLoss=2m(X_T(XR-Y))g=ddRLoss=2m(XT(XR-Y))$$

  - where the scalar m is the number of rows or words in the subset that we're using for training.
- Based on the above gradient expression, it is easy to understand why the derivative of the squared Frobenius norm is relatively easy to calculate compared to that of the Frobenius norm.
- **Key takeaways**
  - With **one trainable matrix** you can learn to align word vectors from one language to another and thus carry out machine translation.

K-nearest Neighbors

- A key operation needed to find a matching word in the previous video was finding the K-nearest neighbors of a vector.
- We will focus on this operation in the next few videos as it's a basic building block for many NLP techniques.
- Notice that it transformed word vector after the transformation of its embedding through an R matrix would be in the French word vector space. But it is not going to be necessarily identical to any of the word vectors in the French word vector space. You need to search through the actual French word vectors to find a French word that is similar to the one that you created from the transformation. You may find words such as *salut* or *bonjour*, which you can return as the French translation of the word hello. So the question is, how do you find similar word vectors? To understand how to find similar word vectors, let's look at a related question. How do you find your friends who are living nearby? Let's pretend that you are visiting San Francisco in the United States and you're visiting your dear friend Andrew. You also want to visit your other friends over the weekend, preferably those who live nearby. One way to do this is to go through your address book and for each friend get their address, calculate how far they are from San Francisco. So one friend is in Shanghai, the other friend is in Bangalore, and another friend is in Los Angeles. You can sort your friends by their distances to San Francisco, then rank them by how close they are. Notice that if you have a lot of friends, which I'm sure you do, this is a very time intensive process. Is there a more efficient way to do this? Notice that two of these friends live in another continent, while the third friend lives in the United States.

Could you have just searched for a subset of friends who live in the United States? You might have realized that it may not have been necessary to go through all of your friends in your address in order to find the ones closest to you. You might have imagined if you somehow could filter on which friends were all in a general region, such as North America, then you could just search within that sub group of friends. If there is a way to slice up the geographic space into regions, you could search just within those regions. When you think about organizing subsets of a dataset efficiently, you may think about placing your data into buckets. If you think about buckets, then you'll definitely want to think about hash tables. Hash tables are useful tools for any kind of work involving data, and you'll learn about hash tables next. In this video, I showed you how using K-nearest neighbors you could translate a word even if it's transformation doesn't exactly match the word embedding in the desired language. And I introduced you to hash tables, a useful data structure that you will learn about in the next video.

- **Key takeaways**
  - To translate from XX to YY using the RR matrix, you may find that $XRXR$ doesn't correspond to any specific vector in YY.
  - KNN can search for the KK nearest neighbors from the computed vector $XRXR$.
  - Thus searching in the whole space can be slow, using a hash tables can minimize your search space.
  - Hash tables may omit some neighbors that are close to your computed vector.

Hash Tables and Hash Functions

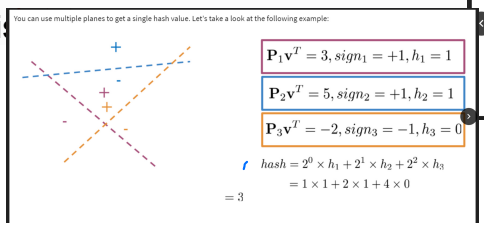
Hash value: : a key that tells which buckets it assigned to

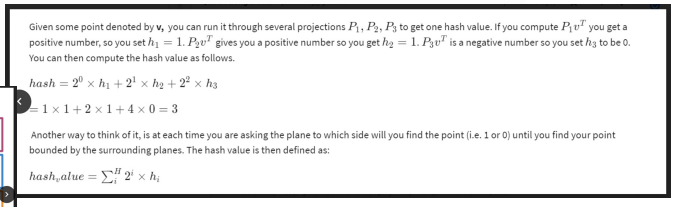
- A hush function maps an object to a bucket in a hash table
- A simple hash function : Hash Value = vector % number of buckets
- This function does not store similar objects in the same bucket ⇒ Locality sensitive hashing

Locality Sensitive Hashing

Locality sensitive hashin is a technique that allows you to hash similar inputs into the same buckets with high probability. Instead of the typical buckets, you can cluster the points by deciding whether they are above or below the line.

- Separate the space using hyperplanes
- For each vector v compute its scalar product **sign** with a normal vector on a hyperplane h_i.
- the product **sign** determines on which side the vector is.
- If the dot product is positive the h_i = 1, else h_i=0
- the hash value i





Approximate Nearest Neighbors

- Subdivide the space using multiple sets of random planes
- Each subdivision will give (probably) different vectors in the same bucket of the computed vector
- Those different vectors (from multiple subdivisions) are good candidates to be the k nearest neighbors
- This Approximate nearest neighbors is not perfect but it is much faster than naive search

Key Takeaways

- Learnings from this topic:
  - Implement machine translation to translate the English word *hello* to the French word *bonjour*.
  - Implement document search. For e.g., given a sentence, “Can I get a refund?”, you can search for similar sentences such as, “What’s your return policy?” or “May I get my money back?”
  - Both the tasks of machine translation and document search have common underlying concepts:
    - Transform a word to its vector representation.
    - k-nearest neighbors, which is a way of searching for similar items.
    - Hash tables which help you distribute your word vectors into subsets by dividing the vector space into regions.
    - Locality sensitive hashing, which helps you perform approximate k-nearest neighbors, an efficient way of searching for similar word vectors.
  - To implement machine translation and document search, the task of finding similar word vectors is of utmost essence.