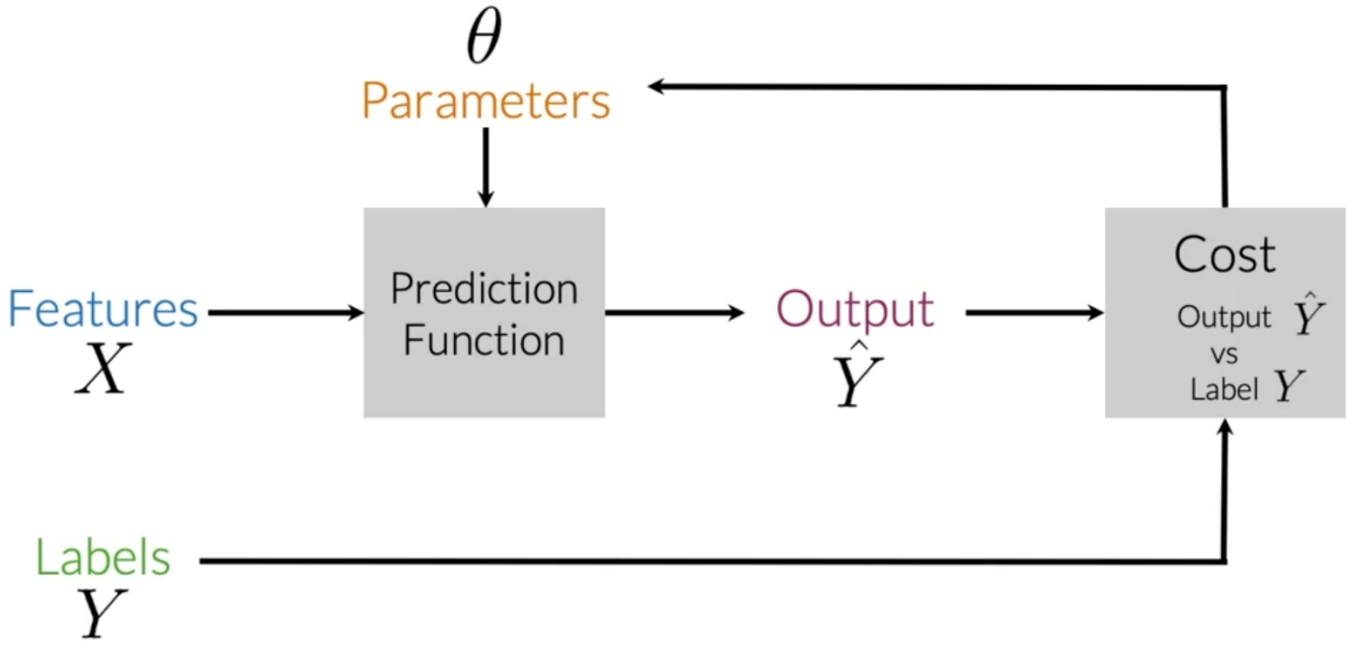
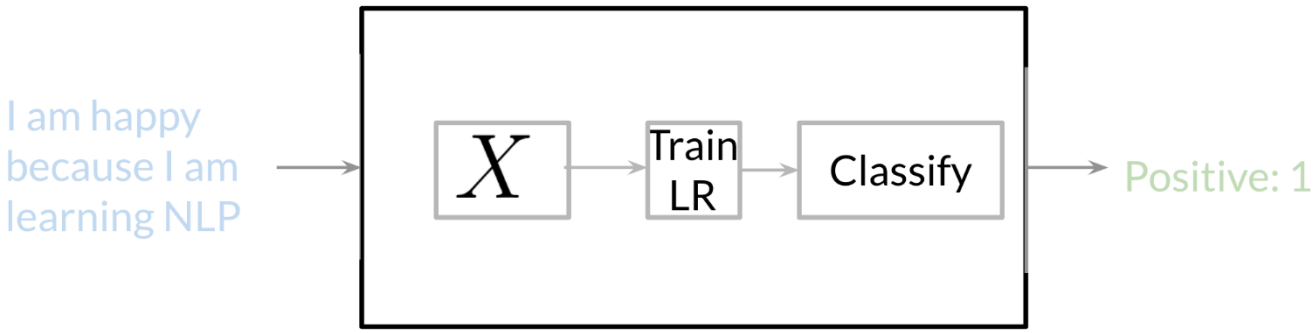
1. LOGISTIC REGRESSION

1.1 Supervised ML & Sentiment Analysis

In supervised machine learning, you usually have an input X, which goes into your prediction function to get your $\hat{Y}$. You can then compare your prediction with the true value Y. This gives you your cost which you use to update the parameters θ. The following image, summarizes the process.



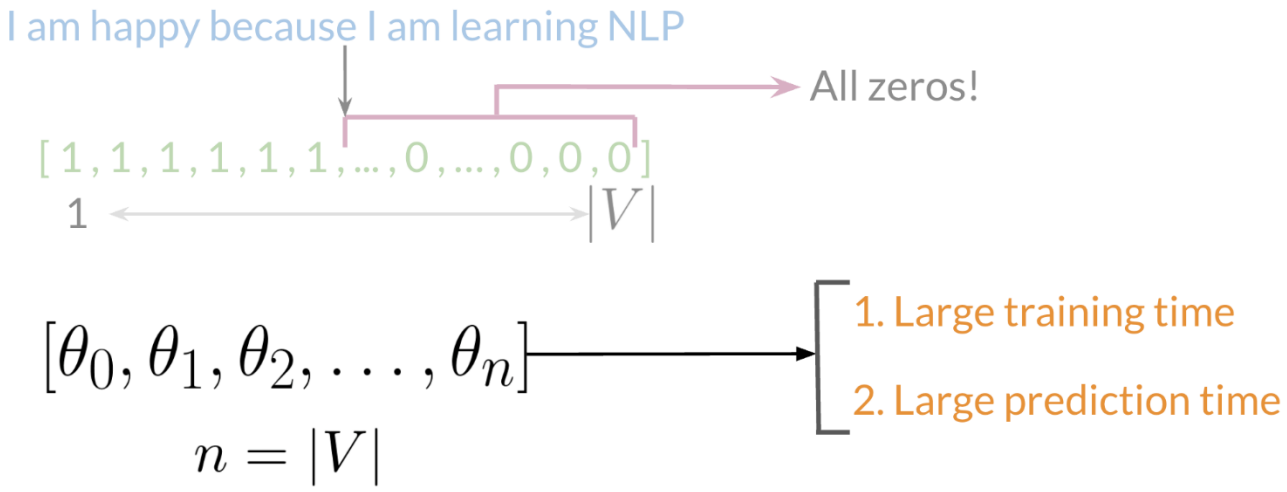
To perform sentiment analysis on a tweet, you first have to represent the text (i.e., "I am happy because I am learning NLP ") as features, you then train your logistic regression classifier, and then you can use it to classify the text.



Note that in this case, you either classify 1, for a positive sentiment, or 0, for a negative sentiment.

1.2 Vocabulary & Feature Extraction

Given a tweet, or some text, you can represent it as a vector of dimension V, where V corresponds to your vocabulary size. If you had the tweet "I am happy because I am learning NLP", then you would put a 1 in the corresponding index for any word in the tweet, and a 0 otherwise.



As you can see, as V gets larger, the vector becomes sparser. Furthermore, we end up having many more features and end up training θ V parameters. This could result in larger training time, and large prediction time.

1.3 Feature Extraction with Frequencies

Given a corpus with positive and negative tweets as follows:

Positive tweets	Negative tweets
I am happy because I am learning NLP	I am sad, I am not learning NLP
I am happy	I am sad

You have to encode each tweet as a vector. Previously, this vector was of dimension V. Now, as you will see in the upcoming videos, you will represent it with a vector of dimension 3. To do so, you have to create a dictionary to map the word, and the class it appeared in (positive or negative) to the number of times that word appeared in its 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

In the past two videos, we call this dictionary `freqs`. In the table above, you can see how words like happy and sad tend to take clear sides, while other words like "I, am" tend to be more neutral. Given this dictionary and the tweet, "I am sad, I am not learning NLP", you can create a vector corresponding to the feature as follows:

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

To encode the negative feature, you can do the same thing.

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

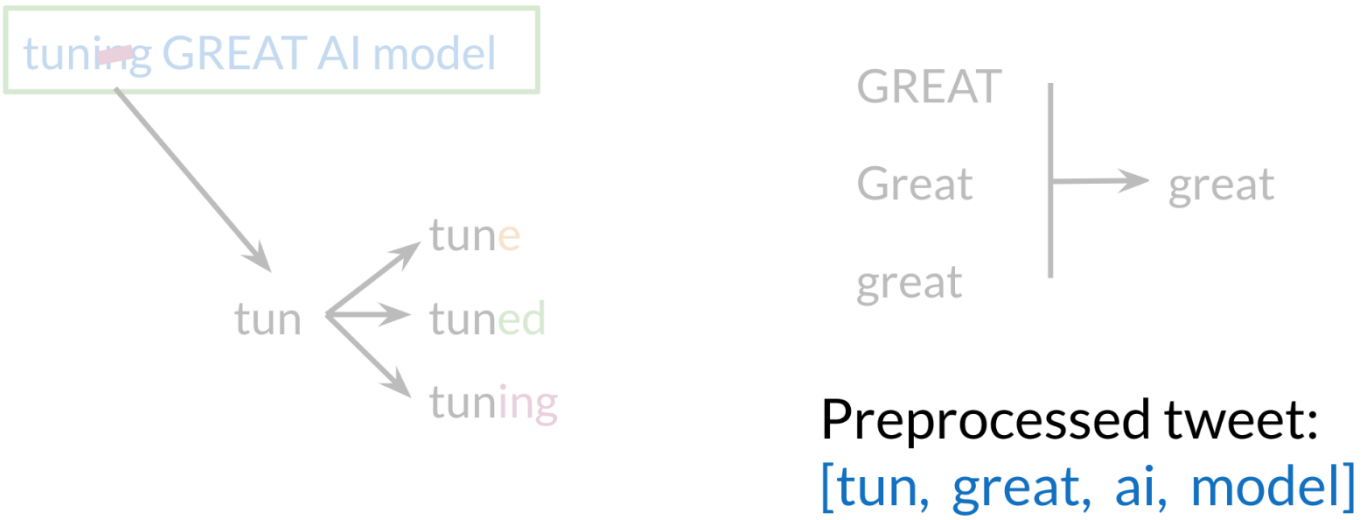
Hence you end up getting the following feature vector [1,8,11]. 1 corresponds to the bias, 8 the positive feature, and 11 the negative feature.

1.4 Pre-processing

When pre-processing, you have to perform the following:

1. Eliminate handles and URLs
2. Tokenize the string into words.
3. Remove stop words like "and, is, a, on, etc."
4. Stemming- or convert every word to its stem. Like dancer, dancing, danced, becomes 'danc'. You can use porter stemmer to take care of this.
5. Convert all your words to lower case.

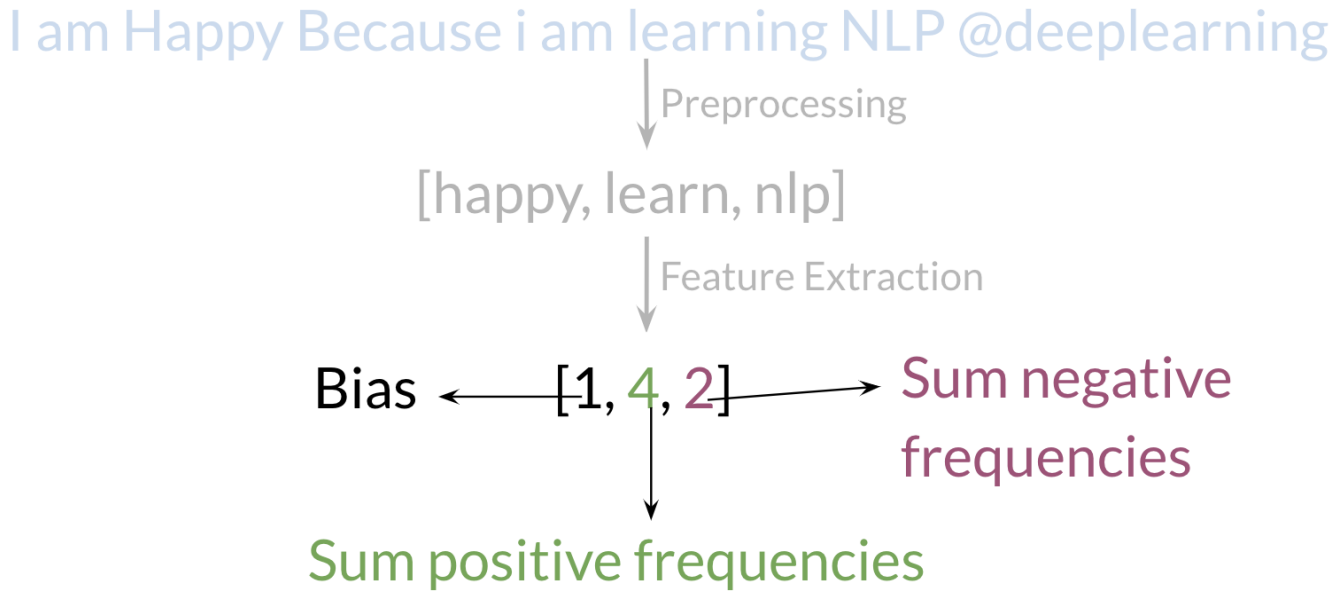
For example, the following tweet "@YMourri and @AndrewYNg are tuning a GREAT AI model at https://deeplearning.ai!!!" after pre-processing becomes



[*tun*,*great*,*ai*,*model*]. Hence you can see how we eliminated handles, tokenized it into words, removed stop words, performed stemming, and converted everything to lower case.

1.5 Putting it all together

Over all, you start with a given text, you perform pre-processing, then you do feature extraction to convert text into numerical representation as follows:



Your *X* becomes of dimension (*m*,3) as follows.

$$\boldsymbol{X} = \begin{bmatrix} 1 & X_1^{(1)} & X_2^{(1)} \\ 1 & X_1^{(2)} & X_2^{(2)} \\ \vdots & \vdots & \vdots \\ 1 & X_1^{(m)} & X_2^{(m)} \end{bmatrix}$$

When implementing it with code, it becomes as follows:

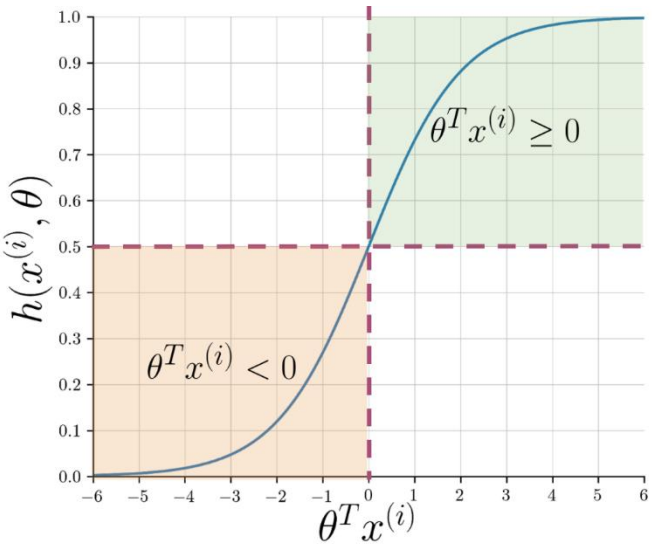
```
freqs = build_freqs(tweets,labels) #Build frequencies dictionary

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
```

You can see in the last step you are storing the extracted features as rows in your *X* matrix and you have *m* of these examples.

1.6 Logistic Regression Overview

Logistic regression makes use of the sigmoid function which outputs a probability between 0 and 1. The sigmoid function with some weight parameter θ and some input $x^{(i)}$ is defined as follows.

$$h(x^{(i)}, \theta) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$



Note that as $\theta^T x^{(i)}$ gets closer and closer to $-\infty$ the denominator of the sigmoid function gets larger and larger and as a result, the sigmoid gets closer to 0. On the other hand, as $\theta^T x^{(i)}$ gets closer and closer to ∞ the denominator of the sigmoid function gets closer to 1 and as a result the sigmoid also gets closer to 1.

Now given a tweet, you can transform it into a vector and run it through your sigmoid function to get a prediction as follows:

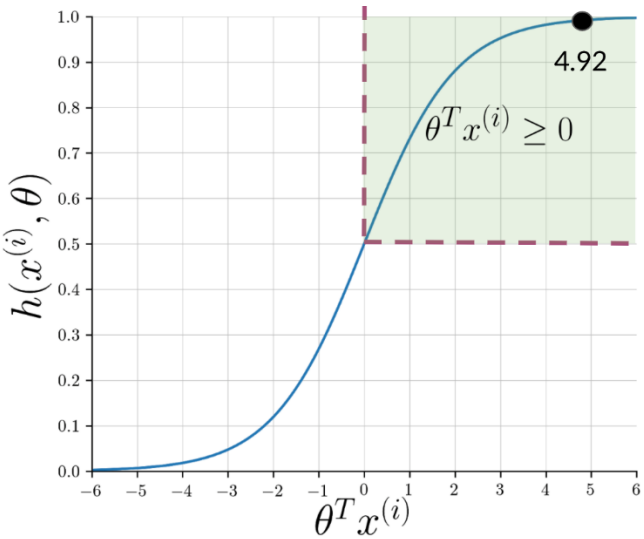
@YMourri and @AndrewYNg are tuning a GREAT AI model

↓

[tun, ai, great, model]

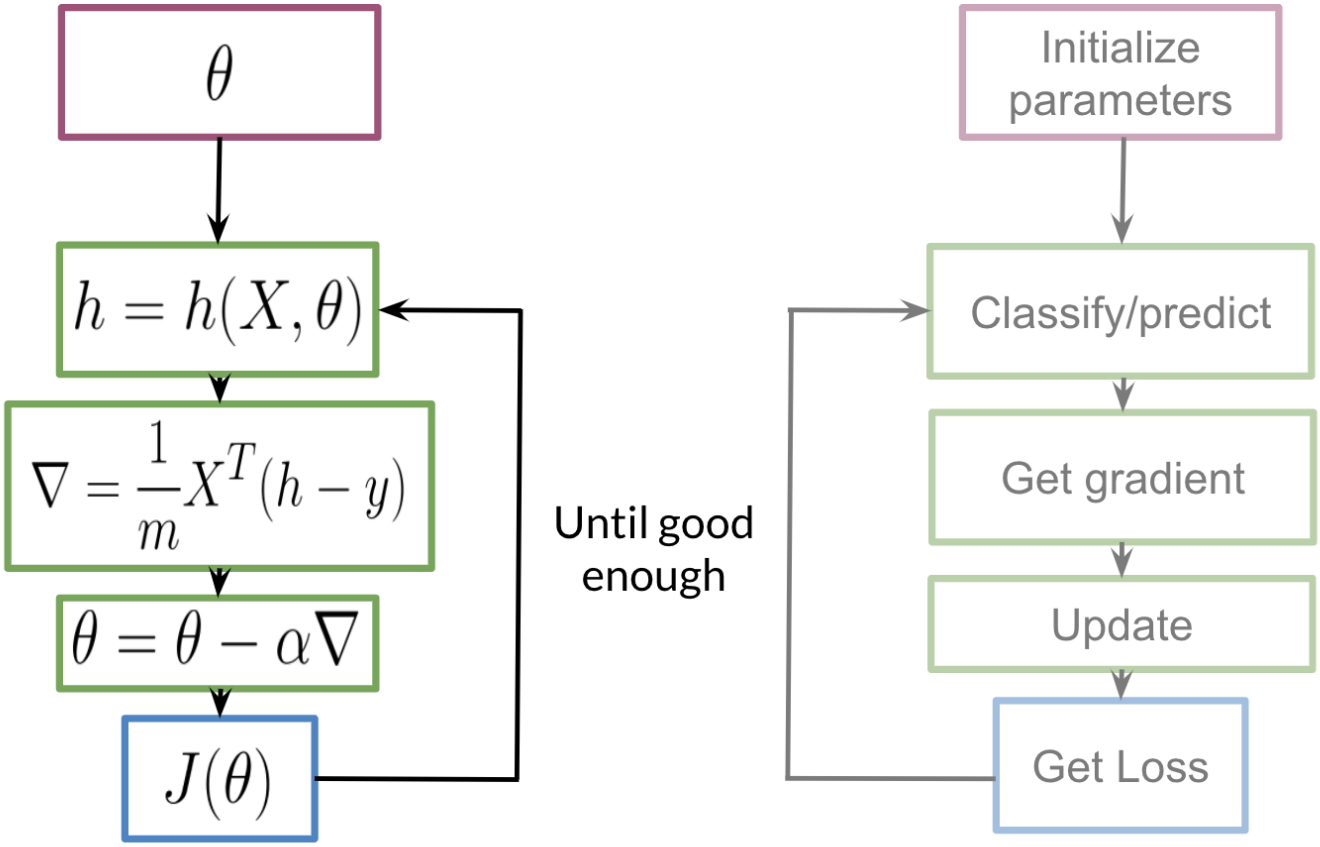
↓

$$x^{(i)} = \begin{bmatrix} 1 \\ 3476 \\ 245 \end{bmatrix} \quad \theta = \begin{bmatrix} 0.00003 \\ 0.00150 \\ -0.00120 \end{bmatrix}$$



1.7 Logistic Regression: Training

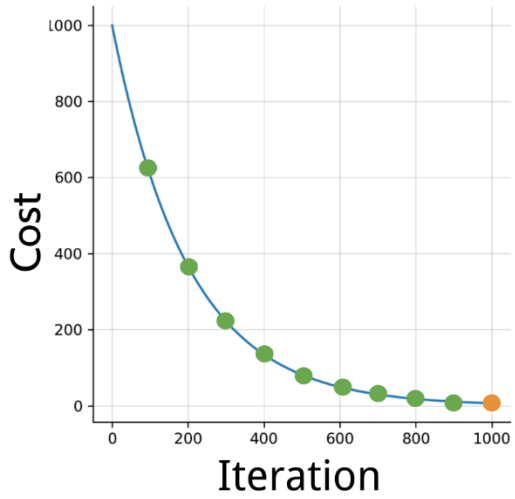
To train your logistic regression function, you will do the following:



You initialize your parameter θ, that you can use in your sigmoid, you then compute the gradient that you will use to update θ, and then calculate the cost. You keep doing so until good enough.

Note: If you do not know what a gradient is, don't worry about it. I will show you what it is at the end of this week in an optional reading. In a nutshell, the gradient allows you to learn what θ is so that you can predict your tweet sentiment accurately.

Usually, you keep training until the cost converges. If you were to plot the number of iterations versus the cost, you should see something like this:



1.8 Logistic Regression: Testing

To test your model, you would run a subset of your data, known as the validation set, on your model to get predictions. The predictions are the outputs of the sigmoid function. If the output is ≥=0.5, you would assign it to a positive class. Otherwise, you would assign it to a negative class.

- X_{val} Y_{val} θ

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

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

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

In the video, I briefly mentioned *X* validation. In reality, given your *X* data you would usually split it into three components. X_{train}, X_{val}, X_{test}. The distribution usually varies depending on the size of your data set. However, an 80,10,10 split usually works fine.

To compute accuracy, you solve the following equation:

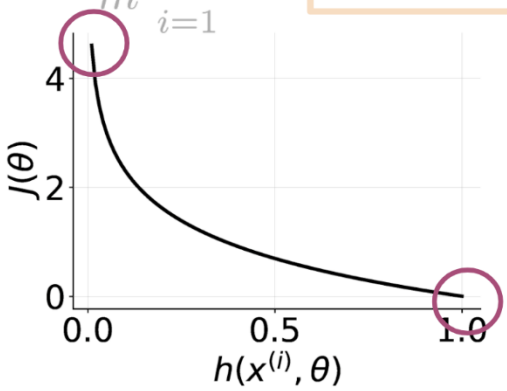
$$\text{Accuracy} \longrightarrow \sum_{i=1}^{m} \frac{(pred^{(i)} == y_{val}^{(i)})}{m}$$

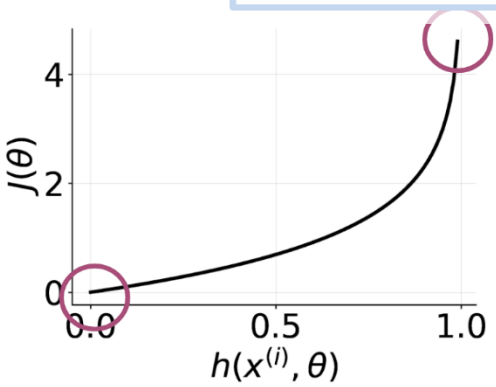
In other words, you go over all your training examples, m*m* of them, and then for every prediction, if it was wright, you add a one. You then divide by *m*.

1.9 Optional Logistic Regression: Cost Function

This is an advanced optional reading where we delve into the details. If you do not get the math, do not worry about it - you will be just fine by moving onto the next component. In this part, I will tell you about the intuition behind why the cost function is designed the way it is. I will then show you how to take the derivative of the logistic regression cost function to get the gradients.

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





[Read more about derivation]

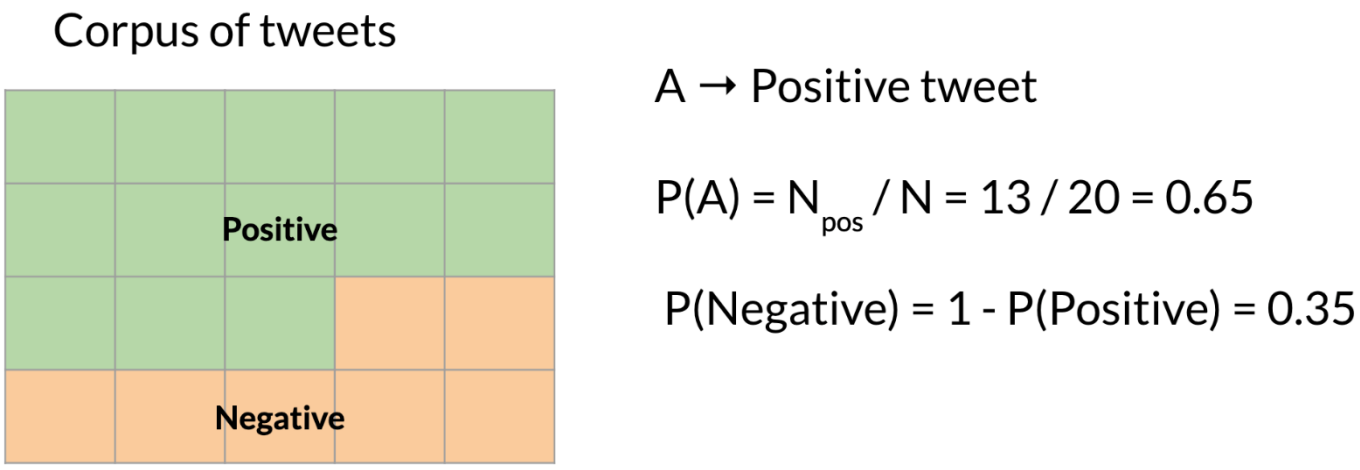
1.10 Optional Logistic Regression: Gradient

Read more ….

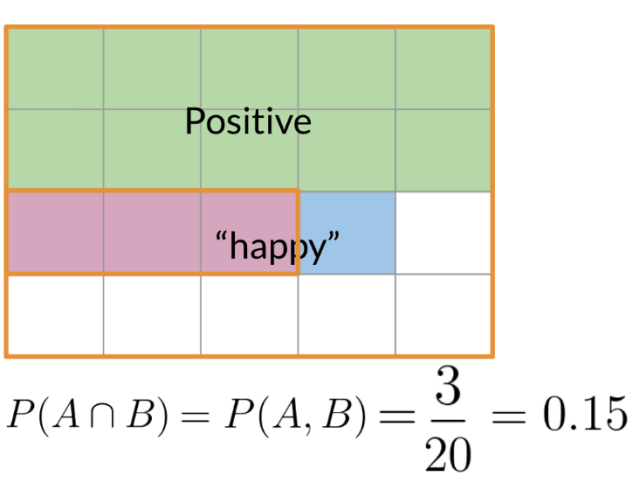
2. NAÏVE BAYES

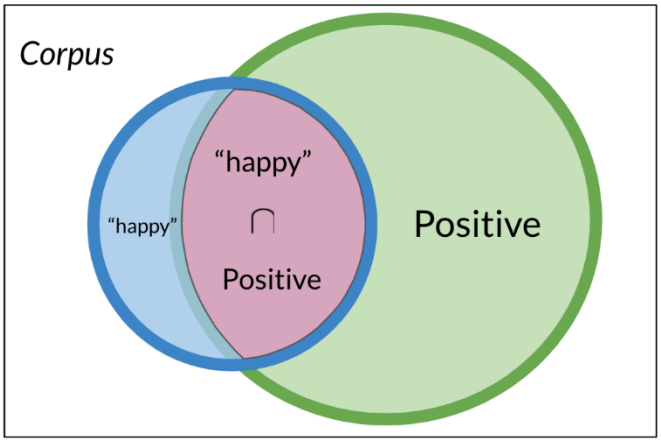
2.1 Probability and Bayes' Rule

You learned about probabilities and Bayes' rule.



To calculate a probability of a certain event happening, you take the count of that specific event and you divide by the sum of all events. Furthermore, the sum of all probabilities has to equal 1.

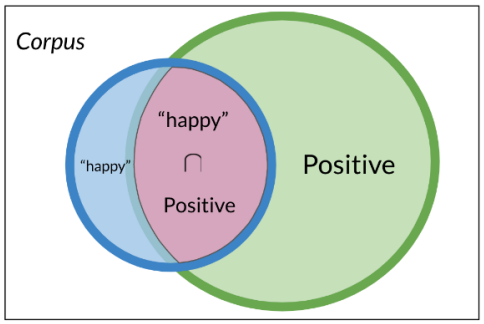




To compute the probability of 2 events happening, like "happy" and "positive" in the picture above, you would be looking at the intersection, or overlap of events. In this case red and blue boxes overlap in 3 boxes. So the answer is $\frac{3}{20}$.

2.2 Bayes' Rule

Conditional probabilities help us reduce the sample search space. For example given a specific event already happened, i.e. we know the word is happy:



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

Then you would only search in the blue circle above. The numerator will be the red part and the denominator will be the blue part. This leads us to conclude the following:

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

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

Substituting the numerator in the right hand side of the first equation, you get the following:

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

Note that we multiplied by P(positive) to make sure we don't change anything. That concludes Bayes Rule which is defined as

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

2.3 Naive Bayes Introduction

To build a classifier, we will first start by creating conditional probabilities given the following table:

Positive tweets

I am happy because I am learning NLP

I am happy, not sad.

Negative tweets

I am sad, I am not learning NLP

I am sad, not happy

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	12

This allows us compute the following table of probabilities:

$$P(I|Pos) = \frac{3}{13}$$

And so on …

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

Once you have the probabilities, you can compute the likelihood score as follows

Tweet: I am happy today; I am learning.

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

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

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

A score greater than 1 indicates that the class is positive, otherwise it is negative.

2.4 Laplacian Smoothing

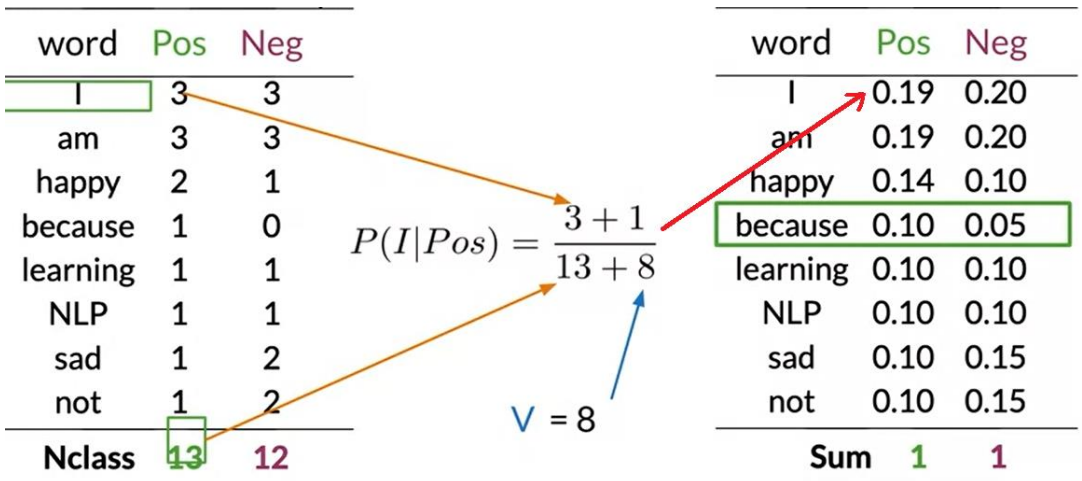
If a word (say 'because') does not appear in the training, then it automatically gets a probability of 0, to fix this we add smoothing as follows

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

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

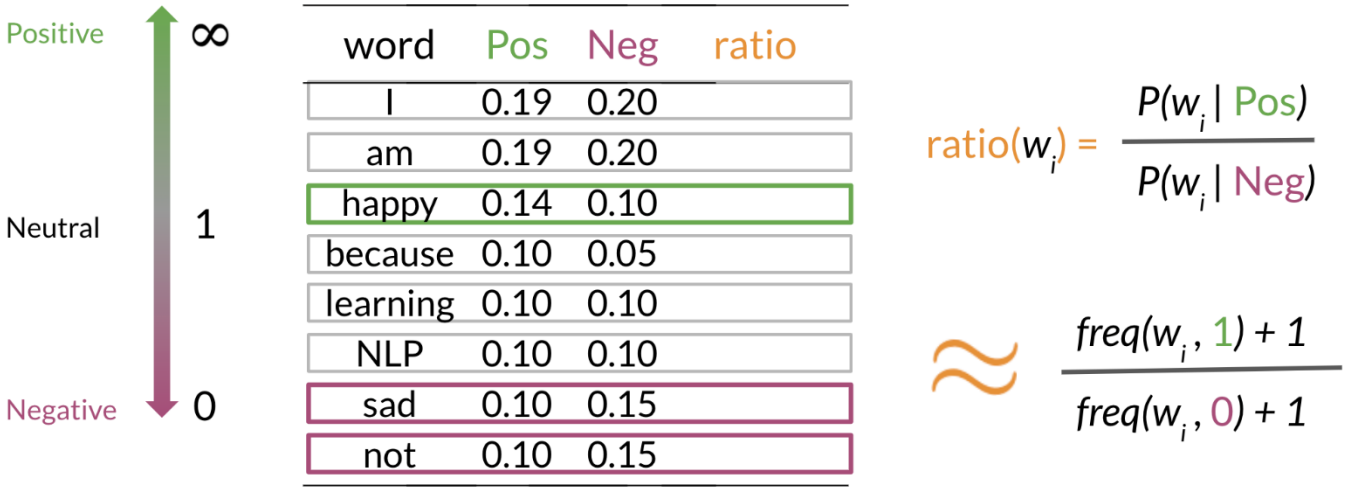
N_{class} = frequency of all words in class

V_{class} = number of unique words in class



2.5 Log Likelihood, Part 1

To compute the log likelihood, we need to get the ratios and use them to compute a score that will allow us to decide whether a tweet is positive or negative. The higher the ratio, the more positive the word is:



To do inference, you can compute the following:

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

As *m* gets larger, we can get numerical flow issues, so we introduce the log, which gives you the following equation:

- $log(a * b) = log(a) + log(b)$
- $log(\frac{P(pos)}{P(neg)} \prod_{i=1}^{n} \frac{P(w_i|pos)}{P(w_i|neg)}) \Rightarrow log\frac{P(pos)}{P(neg)} + \sum_{i=1}^{n} log\frac{P(w_i|pos)}{P(w_i|neg)}$

log prior + log likelihood

We further introduce λ as follows:

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	
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	

Having the λ dictionary will help a lot when doing inference.

2.6 Log Likelihood Part 2

Once you computed the λ dictionary, it becomes straightforward to do inference:

doc: I am happy because I am learning.

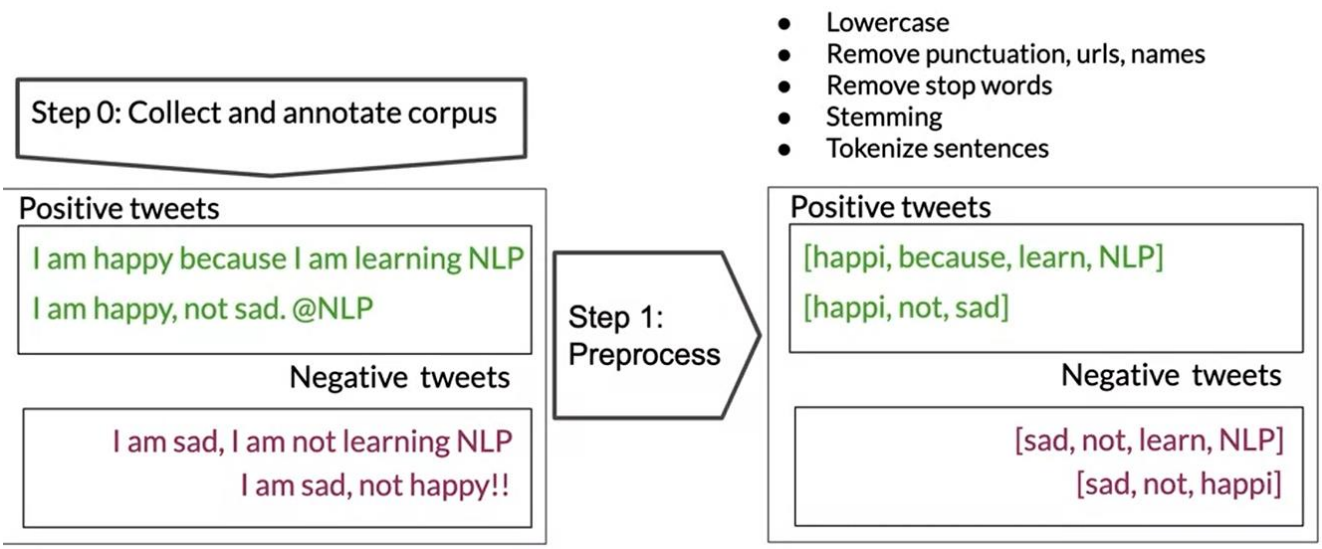
$$\sum_{i=1}^{m} log\frac{P(w_i|pos)}{P(w_i|neg)} = \sum_{i=1}^{m} \lambda(w_i)$$

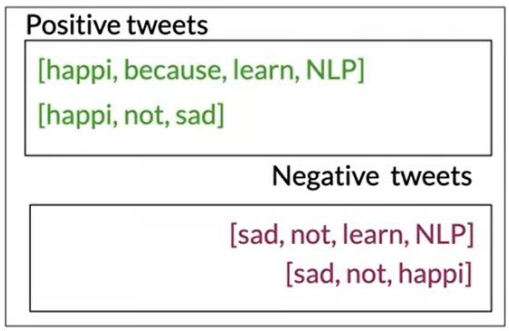
log likelihood = 0 + 0 + 2.2 + 0 + 0 + 0 + 1.1 = **3.3**

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

As you can see above, since 3.3 > 0, we will classify the document to be positive. If we got a negative number, we would have classified it to the negative class.

2.7 Training Naïve Bayes





Step 2: Word count

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

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: $P(\mathrm{w}|\mathrm{class})$

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

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

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

Step 4: Get lambda

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

Step 5: Get the log prior

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

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

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

2.8 Testing naïve Bayes

- 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]

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

$$pred = score > 0$$

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

The example above shows how you can make a prediction given your λ dictionary. In this example the *log prior* is 0 because we have the same amount of positive and negative documents (i.e., log1=0).

Words that are not present in training corpus are considered as neutral during testing.

After prediction, we follow exactly same approach as in logistic regression

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

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

$$pred = score > 0$$

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

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

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

$$pred = score > 0$$

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

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

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

2.9 Applications of Naive Bayes

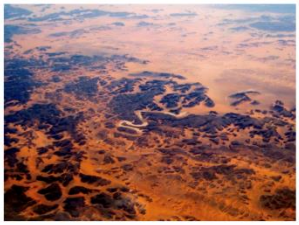
There are many applications of naive Bayes including:

- Author identification
- Spam filtering
- Information retrieval
- Word disambiguation

This method is usually used as a simple baseline. It also really fast.

2.10 Naïve Bayes Assumptions

Naïve Bayes makes the independence assumption and is affected by the word frequencies in the corpus. For example, if you had the following

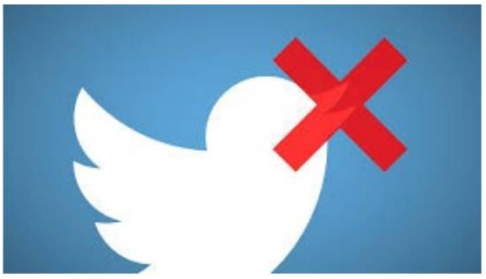


“It is sunny and hot in the Sahara desert.”

“It’s always cold and snowy in ___.”

In the first image, you can see the word sunny and hot tend to depend on each other and are correlated to a certain extent with the word "desert". Naive Bayes assumes independence throughout. Furthermore, if you were to fill in the sentence on the right, this naive model will assign equal weight to the words "spring, summer, fall, winter".

Relative frequencies in corpus



On Twitter, there are usually more positive tweets than negative ones. However, some "clean" datasets you may find are artificially balanced to have to the same amount of positive and negative tweets. Just keep in mind, that in the real world, the data could be much noisier.

2.11 Error Analysis

There are several mistakes that could cause you to misclassify an example or a tweet. For example,

- Removing punctuation
- Removing words

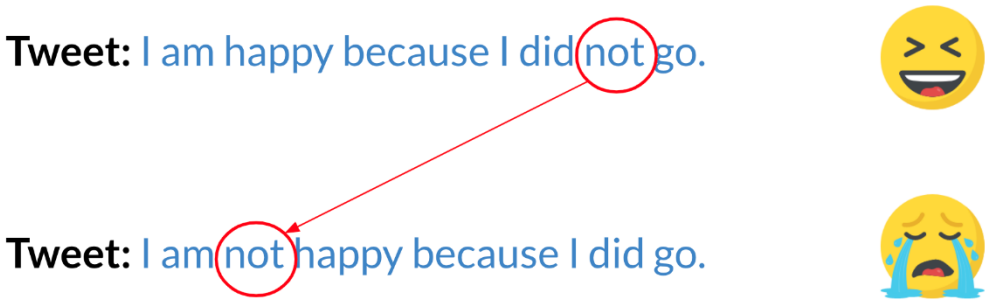
Tweet: This is not good, because your attitude is not even close to being nice.

processed_tweet: [good, attitude, close, nice]

Tweet: My beloved grandmother :(

processed_tweet: [belov, grandmoth]

- Word order



- Adversarial attacks

These include sarcasm, irony, euphemisms.

3. VECTOR SPACE MODELS

3.1 Vector Space Models

Vector spaces are fundamental in many applications in NLP. If you were to represent a word, document, tweet, or any form of text, you will probably be encoding it as a vector. These vectors are important in tasks like information extraction, machine translation, and chatbots. Vector spaces could also be used to help you identify relationships between words as follows:

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

Information Extraction

Machine Translation


Chatbots

The famous quote by Firth says, **"You shall know a word by the company it keeps".** When learning these vectors, you usually make use of the neighbouring words to extract meaning and information about the centre word. If you were to cluster these vectors together, as you will see later in this specialization, you will see that adjective, nouns, verbs, etc. tend to be near one another. Another cool fact, is that synonyms and antonyms are also very close to one another. This is because you can easily interchange them in a sentence and they tend to have similar neighbouring words!

3.2 Word by Word and Word by Doc.

Word by Word Design

We will start by exploring the word by word design. Assume that you are trying to come up with a vector that will represent a certain word. One possible design would be to create a matrix where each row and column corresponds to a word in your vocabulary. Then you can iterate over a document and see the number of times each word shows up next each other word. You can keep track of the number in the matrix. In the video I spoke about a parameter *K*. You can think of *K* as the bandwidth that decides whether two words are next to each other or not.

I like simple data

I prefer simple raw data

k=2

	simple	raw	like	I
data	2	1	1	0

n

In the example above, you can see how we are keeping track of the number of times words occur together within a certain distance *k*. At the end, you can represent the word data, as a vector *v*=[2,1,1,0].

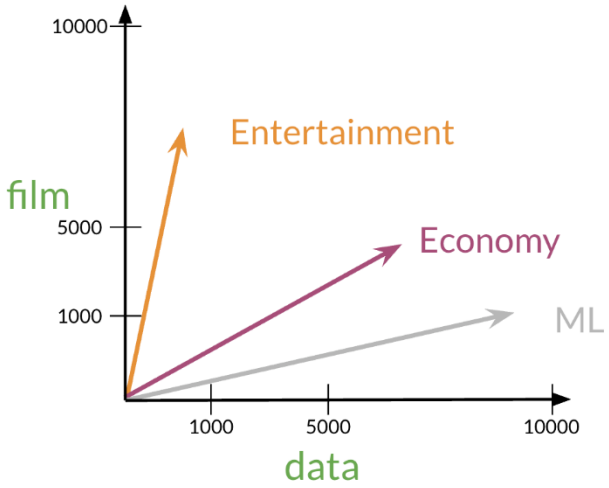
Word by Document Design

You can now apply the same concept and map words to documents. The rows could correspond to words and the columns to documents. The numbers in the matrix correspond to the number of times each word showed up in the document.

Corpus

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

You can represent the entertainment category, as a vector *v*=[500,7000]. You can then also compare categories as follows by doing a simple plot.



	Entertainment	Economy	ML
data	500	6620	9320
film	7000	4000	1000

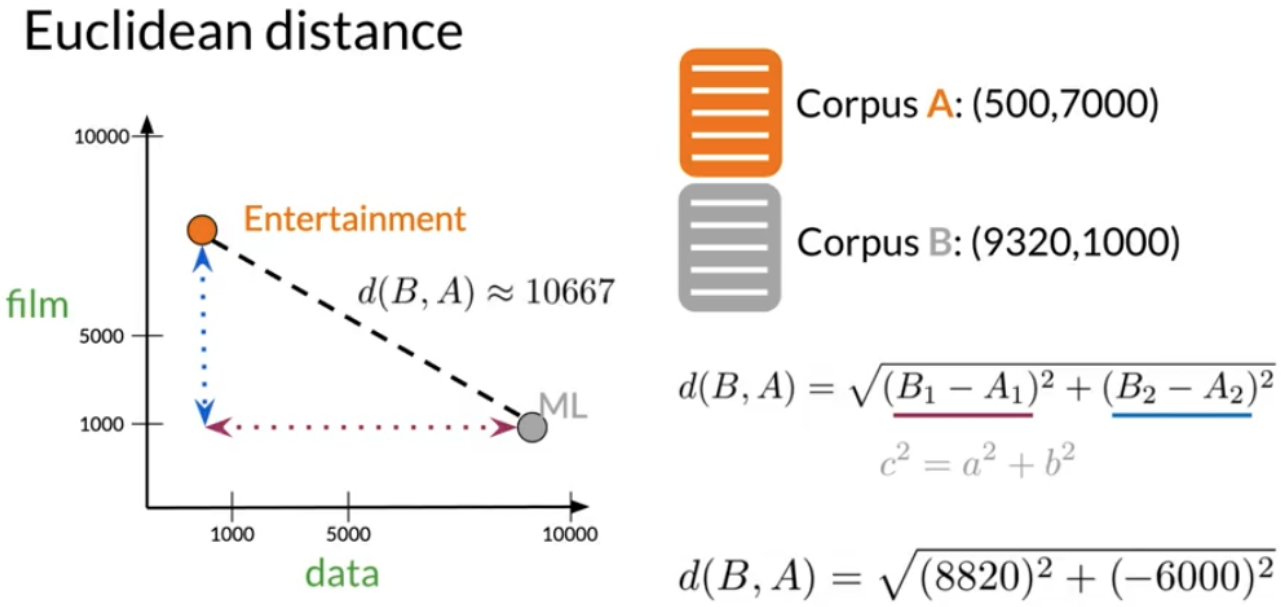
Measures of "similarity:"
Angle
Distance

Later this week, you will see how you can use the angle between two vectors to measure similarity.

3.3 Euclidian Distance

Let us assume that you want to compute the distance between two points: *A*, *B*. To do so, you can use the Euclidean distance defined as

$$d_{(B,A)} = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$$



You can generalize finding the distance between the two points (*A*,*B*) to the distance between an *n* dimensional vector as follows:

$$d(\bar{v}, \bar{w}) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}$$

Here is an example where I calculate the distance between 2 vectors (*n*=3).

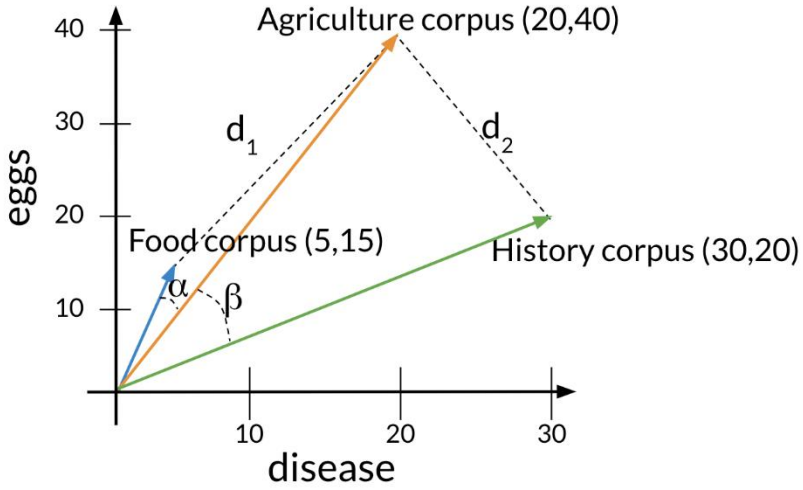
	data	$\vec{w}$ boba	$\vec{v}$ ice-cream
AI	6	0	1
drinks	0	4	6
food	0	6	8

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

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

3.4 Cosine Similarity: Intuition

One of the issues with Euclidean distance is that it is not always accurate and sometimes we are not looking for that type of similarity metric. For example, when comparing large documents to smaller ones with Euclidean distance one could get an inaccurate result. Look at the diagram below:



Euclidean distance: $d_2 < d_1$

Angles comparison: $\beta > \alpha$

The cosine of the angle between the vectors

Normally the **food** corpus and the **agriculture** corpus are more similar because they have the same proportion of words. However the food corpus is much smaller than the agriculture corpus. To further clarify, although the history corpus and the agriculture corpus are different, they have a smaller euclidean distance. Hence $d_2 < d_1$.

To solve this problem, we look at the cosine between the vectors. This allows us to compare β and α.

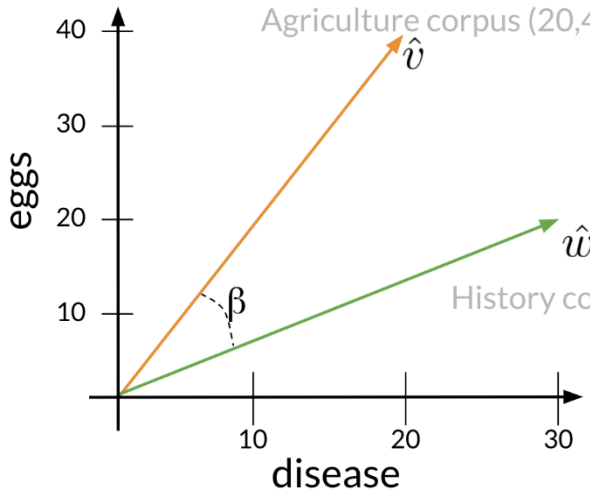
3.5 Cosine Similarity

Before getting into the cosine similarity function remember that the **norm** of a vector is defined as:

$$\|\bar{v}\| = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

The **dot product** is then defined as:

$$\bar{v} \cdot \bar{w} = \sum_{i=1}^{n} \bar{v}_i \cdot \bar{w}_i$$



$$\hat{v} \cdot \hat{w} = \|\hat{v}\| \|\hat{w}\| \cos(\beta)$$

$$\cos(\beta) = \frac{\hat{v} \cdot \hat{w}}{\|\hat{v}\| \|\hat{w}\|}$$

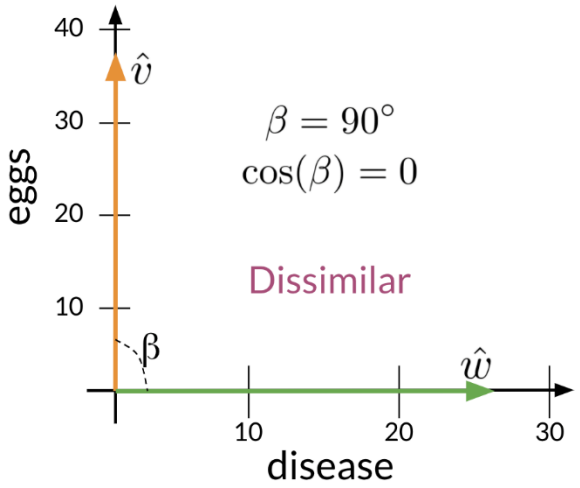
$$= \frac{(20 \times 30) + (40 \times 20)}{\sqrt{20^2 + 40^2} \times \sqrt{30^2 + 20^2}}$$

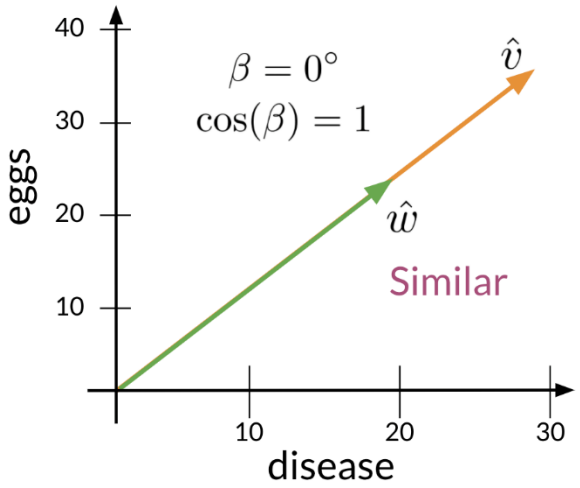
$$= 0.87$$

The following cosine similarity equation makes sense:

$$\cos(\beta) = \frac{\bar{v} \cdot \bar{w}}{\|\bar{v}\| \|\bar{w}\|}$$

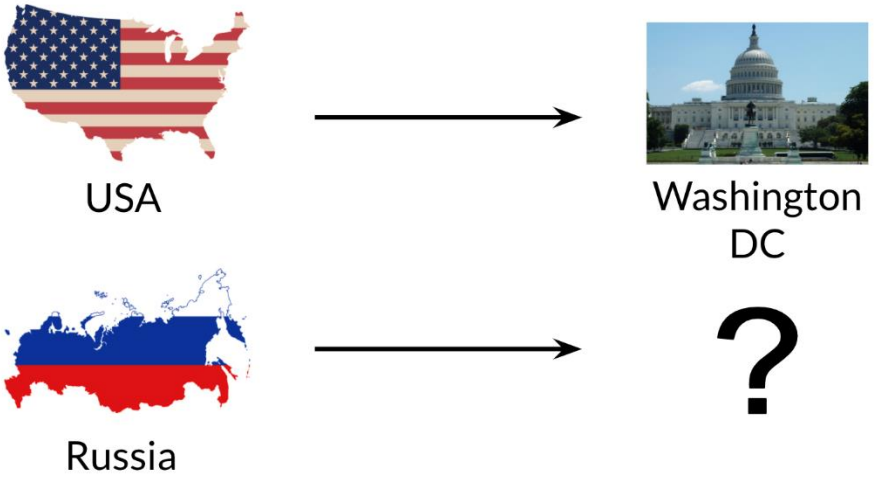
If $\bar{v}$ and $\bar{w}$ are the same then you get the numerator to be equal to the denominator. Hence β=0. On the other hand, the dot product of two orthogonal (perpendicular) vectors is 0. That takes place when β=90.



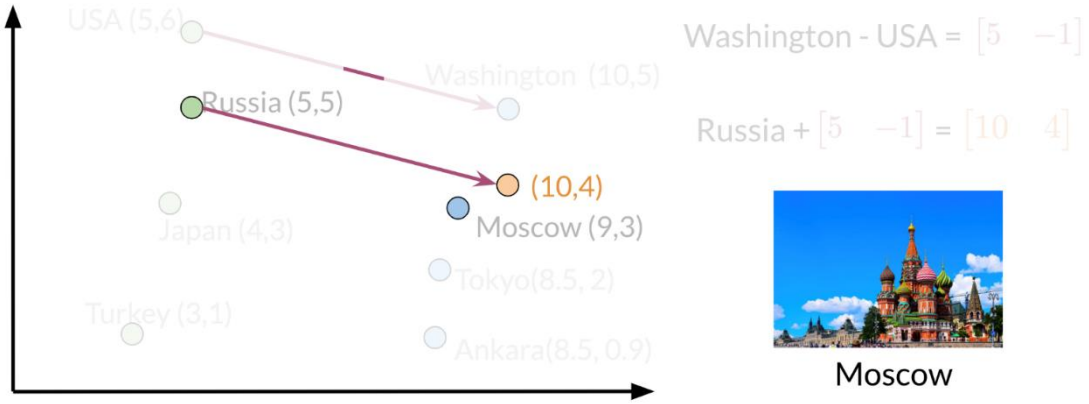


3.6 Manipulating Words in Vector Spaces

You can use word vectors to actually extract patterns and identify certain structures in your text. For example:



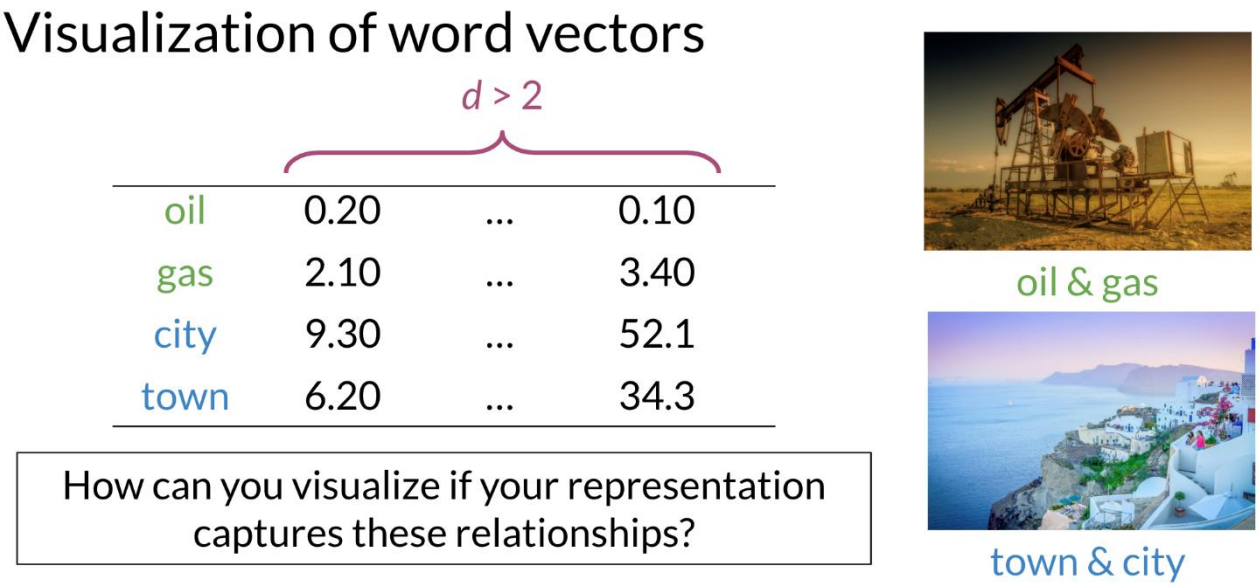
You can use the word vector for Russia, USA, and DC to actually compute a **vector** that would be very close to that of Moscow. You can then use cosine similarity of the **vector** with all the other word vectors you have and you can see that the vector of Moscow is the closest. Isn't that cool?



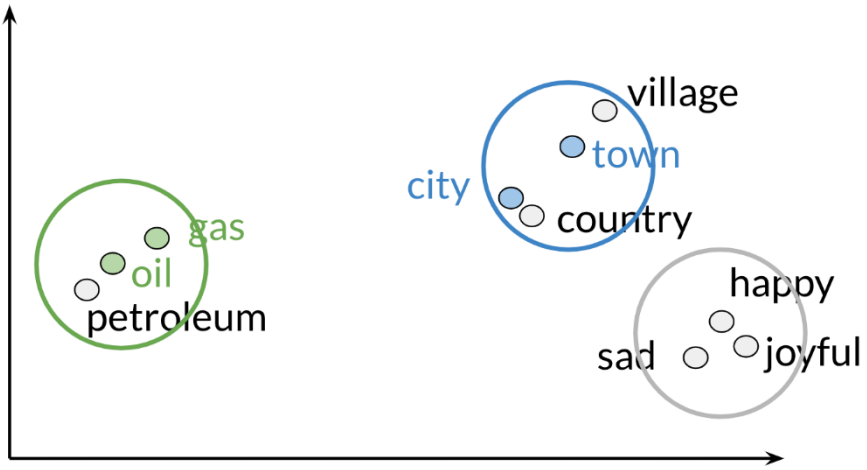
Note that the distance (and direction) between a country and its capital is relatively the same. Hence manipulating word vectors allows you identify patterns in the text.

3.7 Visualization and PCA

Principal component analysis is an unsupervised learning algorithm which can be used to reduce the dimension of your data. As a result, it allows you to visualize your data. It tries to combine variances across features. Here is a concrete example of PCA:



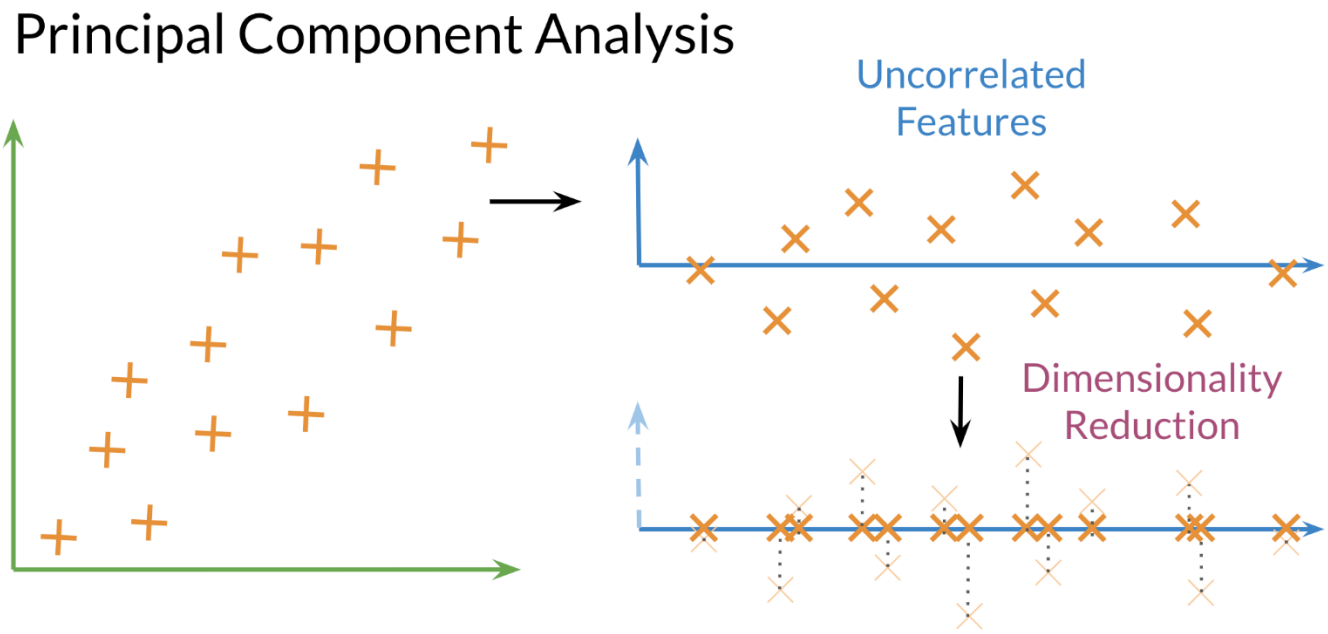
Note that when doing PCA on this data, you will see that oil & gas are close to one another and town & city are also close to one another. To plot the data you can use PCA to go from $d>2$ dimensions to $d=2$.



Those are the results of plotting a couple of vectors in two dimensions. Note that words with similar part of speech (POS) tags are next to one another. This is because many of the training algorithms learn words by identifying the neighbouring words. Thus, words with similar POS tags tend to be found in similar locations. An interesting insight is that synonyms and antonyms tend to be found next to each other in the plot. Why is that the case?

3.8 PCA algorithm

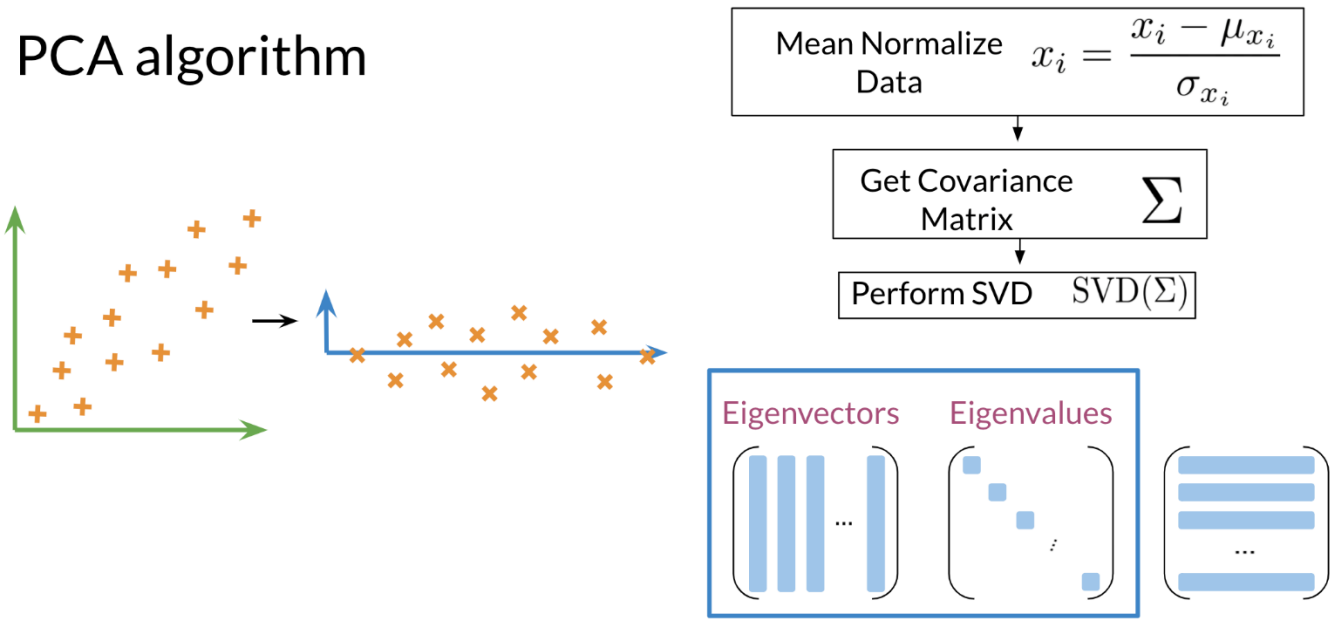
PCA is commonly used to reduce the dimension of your data. Intuitively the model collapses the data across principal components. You can think of the first principal component (in a 2D dataset) as the line where there is the most amount of variance. You can then collapse the data points on that line. Hence you went from 2D to 1D. You can generalize this intuition to several dimensions.



Eigenvector: the resulting vectors, also known as the uncorrelated features of your data

Eigenvalue: the amount of information retained by each new feature. You can think of it as the variance in the eigenvector.

Also each **eigenvalue** has a corresponding eigenvector. The eigenvalue tells you how much variance there is in the eigenvector. Here are the steps required to compute PCA:



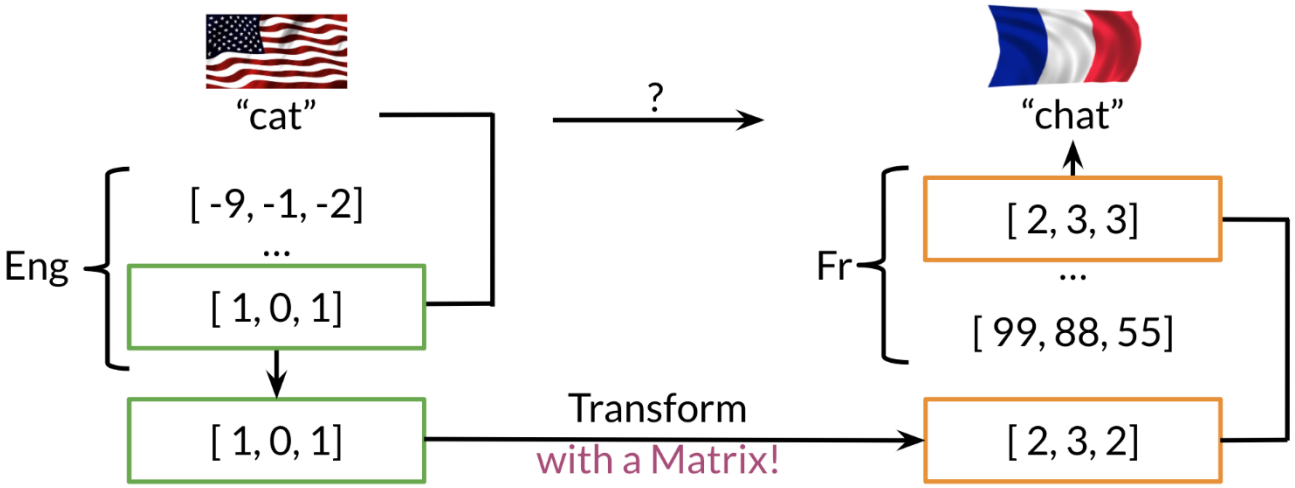
<u>Steps to Compute PCA:</u>

- Mean normalize your data
- Compute the covariance matrix
- Compute SVD on your covariance matrix. These returns [U S V] = *svd*(Σ). The three matrices U, S, V are drawn above. U is labelled with eigenvectors, and S is labelled with eigenvalues.
- You can then use the first n columns of vector *U*, to get your new data by multiplying *XU*[:,0:*n*].

4. MACHINE TRANSLATION

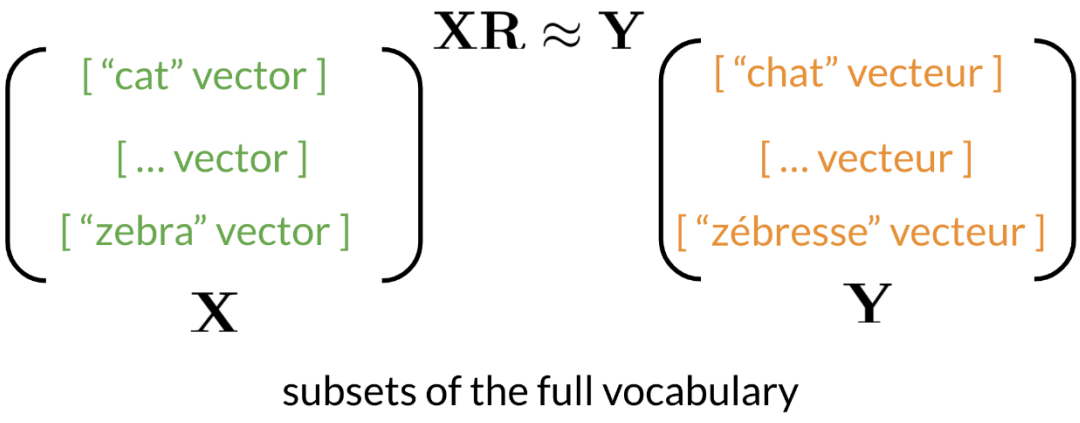
4.1 Transforming word vectors

In the previous week, I showed you how we can plot word vectors. Now, you will see how you can take a word vector and learn a mapping that will allow you to translate words by learning a "transformation matrix". Here is a visualization:



Note that the word "chat" in French means cat. You can learn that by taking the vector corresponding to "cat" in English, multiplying it by a matrix that you learn and then you can use cosine similarity between the output and all the French vectors. You should see that the closest result is the vector which corresponds to "chat".

Here is a visualization of that showing you the aligned vectors:



Note that X*X* corresponds to the matrix of English word vectors and Y*Y* corresponds to the matrix of French word vectors. R*R* is the mapping matrix.

Steps required to learn R*R*:

- Initialize R
- For loop
  - Loss = $=\| XR - Y \|_F$
  - $g = \frac{\partial}{\partial R} Loss$
  - R = $R - \alpha * g$

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

- Let $A_F = \| XR - Y \|_F = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$
- $A_F = \sqrt{2^2 + 2^2 + 2^2 + 2^2} = 4$

The general formula for frobenius norm is given as following:

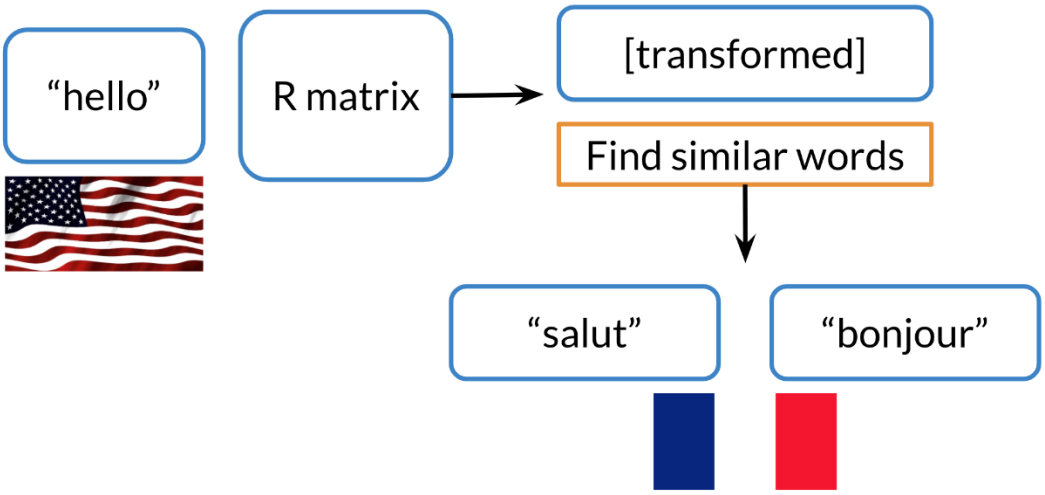
$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} (a_{ij})^2}$$

In summary you are making use of the following:

- XR ≈ Y i.e., the distances between XR and Y should be minimum.
- minimize $\|XR - Y\|_F^2$

4.2 K-nearest neighbours

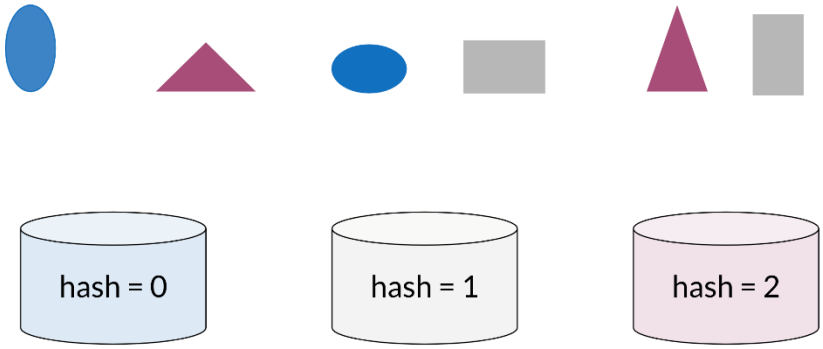
After you have computed the output of *XR* you get a vector. You then need to find the most similar vectors to your output. Here is a visual example:



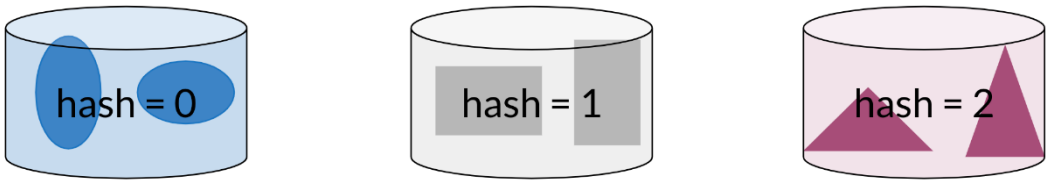
In the video, we mentioned if you were in San Francisco, and you had friends all over the world, you would want to find the nearest neighbours. To do that it might be expensive to go over all the countries one at a time. So, we will introduce hashing to show you how you can do a look up much faster.

4.3 Hash tables and hash functions

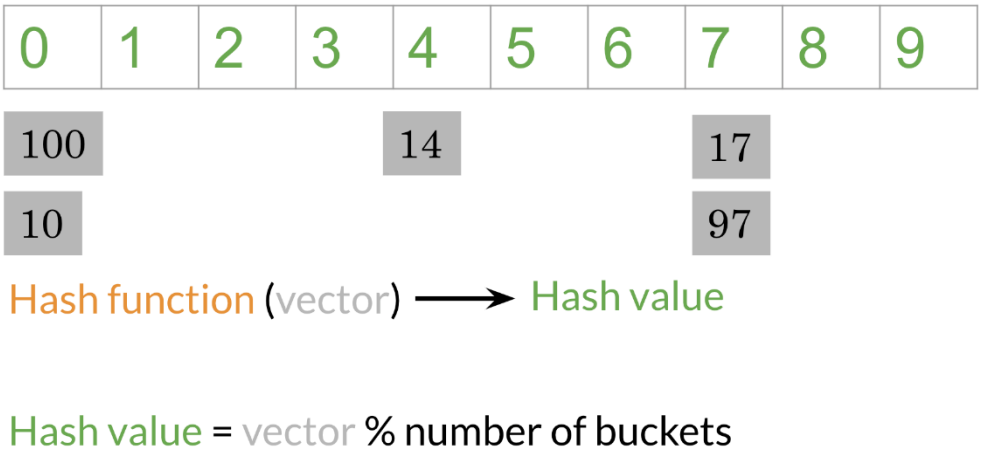
Imagine you had to cluster the following figures into different buckets:



Note that the figures blue, red, and Gray ones would each be clustered with each other



You can think of hash function as a function that takes data of arbitrary sizes and maps it to a fixed value. The values returned are known as *hash values* or even *hashes*.



The diagram above shows a concrete example of a hash function which takes a vector and returns a value. Then you can mod that value by the number of buckets and put that number in its corresponding bucket. For example, 14 is in the 4th bucker, 17 & 97 are in the 7th bucket. Let's take a look at how you can do it using some code.

```
def basic_hash_table(value_l,n_buckets):

    def hash_function(value_l,n_buckets):
        return int(value_l) % n_buckets

    hash_table = {i:[] for i in range(n_buckets)}
    for value in value_l:
        hash_value = hash_function(value,n_buckets)
        hash_table[hash_value].append(value)

    return hash_table
```

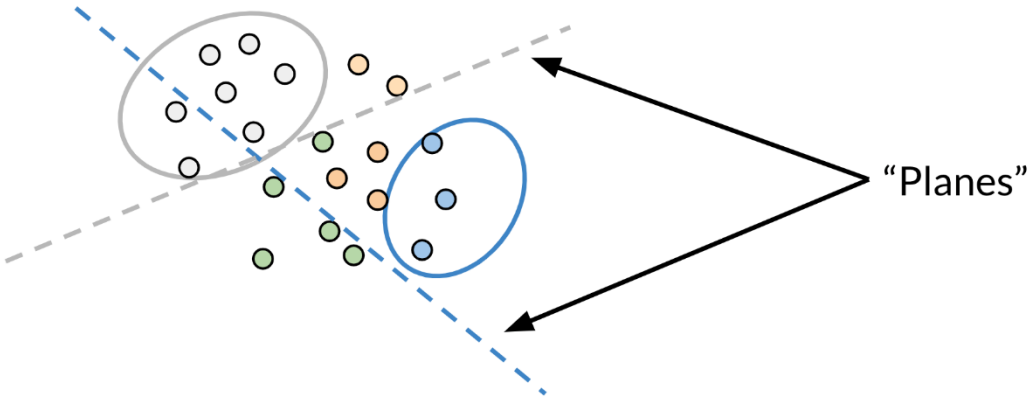
The code snippet above creates a basic hash table which consists of hashed values inside their buckets. **hash_function** takes in *value_l* (a list of values to be hashed) and *n_buckets* and mods the value by the buckets. Now to create the *hash_table*, you first initialize a list to be of dimension *n_buckets (*each value will go to a bucket). For each value in your list of values, you will feed it into your **hash_function,** get the *hash_value*, and append it to the list of values in the corresponding bucket.

Now given an input, you don't have to compare it to all the other examples, you can just compare it to all the values in the same *hash_bucket* that input has been hashed to.

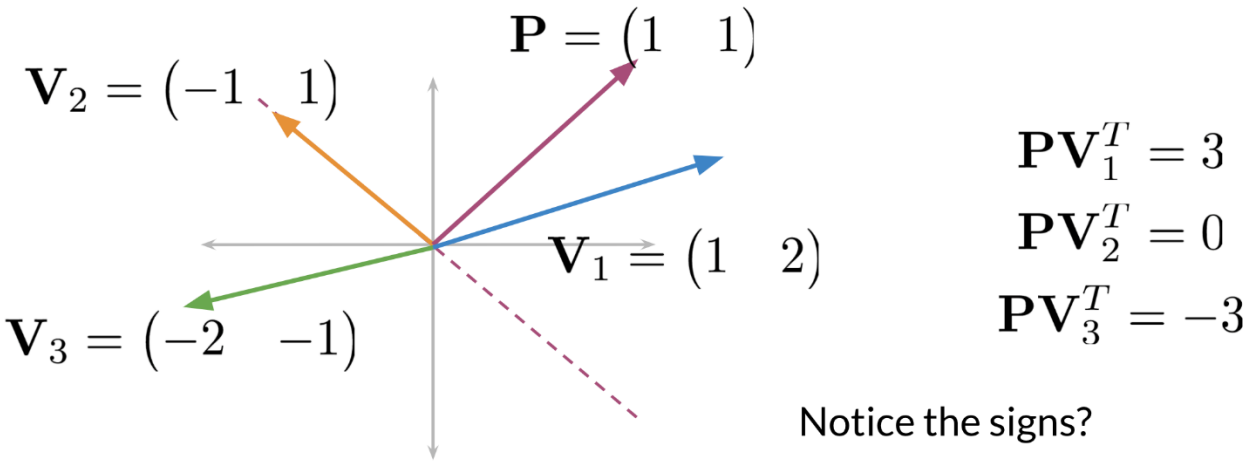
When hashing you sometimes want similar words or similar numbers to be hashed to the same bucket. To do this, you will use "locality sensitive hashing." Locality is another word for "location". So locality sensitive hashing is a hashing method that cares very deeply about assigning items based on where they're located in vector space

4.4 Locality sensitive hashing

Locality sensitive hashing is a technique that allows you to hash similar inputs into the same buckets with high probability.

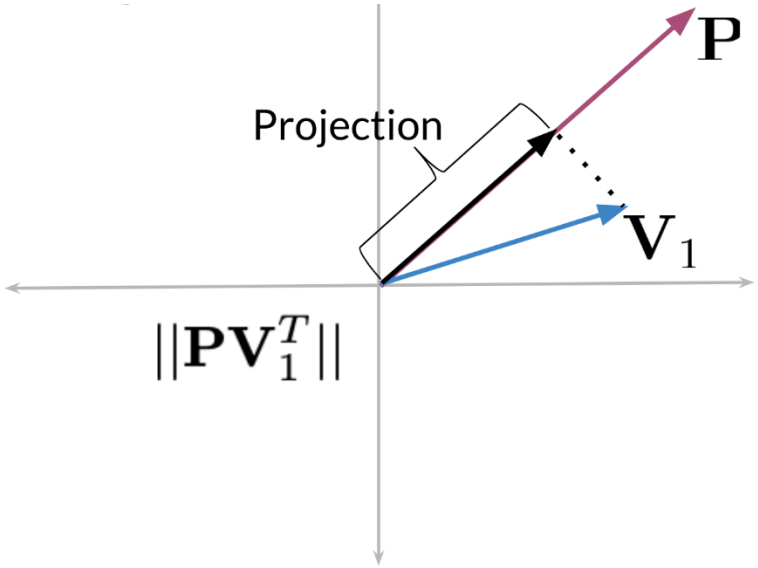


Instead of the typical buckets we have been using, you can think of clustering the points by deciding whether they are above or below the line. Now as we go to higher dimensions (say n-dimensional vectors), you would be using planes instead of lines. Let's look at a concrete example:



Given perpendicular vector located at (1,1) and three vectors V_1 = (1,2), V_2 = (-1,1), V_3 = (-2,-1) you will see what happens when we take the dot product. First note that the dashed line is our plane. The vector with point P = (1,1) is perpendicular to that line (plane). Now any vector above the dashed line that is multiplied by (1,1) would have a positive number. Any vector below the dashed line when dotted with (1,1) will have a negative number. Any vector on the dashed line multiplied by (1,1) will give you a dot product of 0.

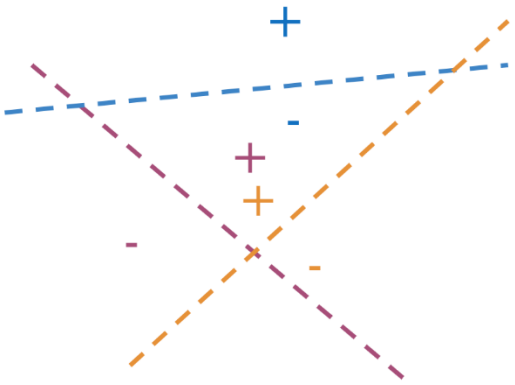
Here is how to visualize a projection (i.e. a dot product between two vectors):



When you take the dot product of a vector V_1 and a P, then you take the magnitude or length of that vector, you get the black line (labelled as Projection). The sign indicates on which side of the plane the projection vector lies.

4.5 Multiple Planes

You can use multiple planes to get a single hash value. Let's take a look at the following example:



$$\mathbf{P}_1\mathbf{v}^T = 3, sign_1 = +1, h_1 = 1$$

$$\mathbf{P}_2\mathbf{v}^T = 5, sign_2 = +1, h_2 = 1$$

$$\mathbf{P}_3\mathbf{v}^T = -2, sign_3 = -1, h_3 = 0$$

$$hash = 2^0 \times h_1 + 2^1 \times h_2 + 2^2 \times h_3$$
$$= 1 \times 1 + 2 \times 1 + 4 \times 0$$
$$= 3$$

Given some point denoted by **v,** you can run it through several projections P_1, P_2, P_3 to get one hash value. If you compute P_1v^T you get a positive number, so you set h_1 =1. P_2v^T gives you a positive number so you get h_2 = 1. P_3v^T is a negative number so you set h_3 to be 0. You can then compute the hash value as follows.

$$hash = 2^0 h_1 + 2^1 h_2 + 2^2 h_3 = 1 \times 1 + 2 \times 1 + 4 \times 0 = 3$$

Another way to think of it, is at each time you are asking the plane to which side will you find the point (i.e. 1 or 0) until you find your point bounded by the surrounding planes. The hash value is then defined as:

$$hashvalue = \sum_{i=0}^{H} 2^i h_{i+1}$$

Here is how you can code it up:

```
def hash_multiple_plane(P_l,v):
    hash_value = 0

    for i, P in enumerate(P_l):
        sign = side_of_plane(P,v)
        hash_i = 1 if sign >=0 else 0
        hash_value += 2**i * hash_i

    return hash_value
```

P_l is the list of planes. You initialize the value to 0, and then you iterate over all the planes (P), and you keep track of the index. You get the sign by finding the sign of the dot product between v and your plane P. If it is positive, you set it equal to 1, otherwise you set it equal to 0. You then add the score for the ith plane to the hash value by computing $2^i h_{i+1}$

4.6 Approximate nearest neighbours

Approximate nearest neighbours does not give you the full nearest neighbours but gives you an approximation of the nearest neighbours. It usually trades off accuracy for efficiency. Look at the following plot:

You are trying to find the nearest neighbour for the red vector (point). The first time, the plane gave you green points. You then ran it a second time, but this time you got the blue points. The third time you got the orange points to be the neighbours. So, you can see as you do it more times, you are likely to get all the neighbours. Here is the code for one set of random planes. Make sure you understand what is going on.

```
num_dimensions = 2 #300 in assignment
num_planes = 3 #10 in assignment

random_planes_matrix = np.random.normal(
                       size=(num_planes,
                             num_dimensions))
```

```
array([[ 1.76405235  0.40015721]
       [ 0.97873798  2.2408932 ]
       [ 1.86755799 -0.97727788]])
```

```
v = np.array([[2,2]])
```

```
def side_of_plane_matrix(P,v):
     dotproduct = np.dot(P,v.T)
     sign_of_dot_product = np.sign(dotproduct)
     return sign_of_dot_product

num_planes_matrix = side_of_plane_matrix(
                    random_planes_matrix,v)
```

```
array([[1.]
       [1.]
       [1.])
```

See notebook for calculating the hash value!

4.7 Searching documents

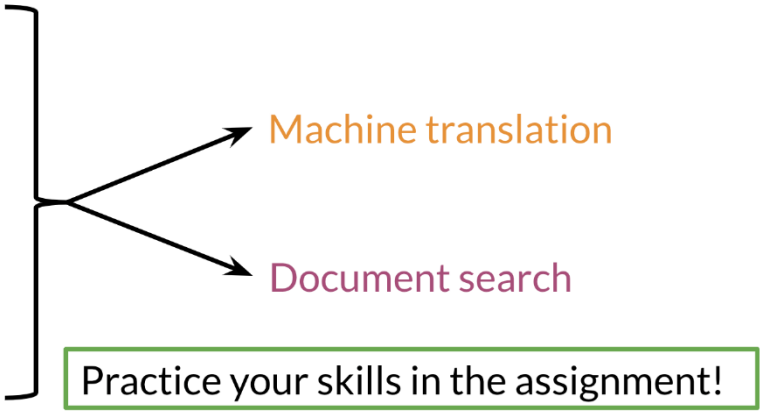
The previous video shows you a toy example of how you can actually represent a document as a vector.

```
word_embedding = {"I": np.array([1,0,1]),
                  "love": np.array([-1,0,1]),
                  "learning": np.array([1,0,1])}

words_in_document = ['I', 'love', 'learning']

document_embedding = np.array([0,0,0])

for word in words_in_document:
    document_embedding += word_embedding.get(word,0)

print(document_embedding)
```

```
array([1 0 3])
```

In this example, you just add the word vectors of a document to get the document vector. So, in summary you should now be familiar with the following concepts:

- Transform vector
- "K nearest neighbors"
- Hash tables
- Divide vector space into regions
- Locality sensitive hashing
- Approximated nearest neighbors



Good luck with the programming assignment!