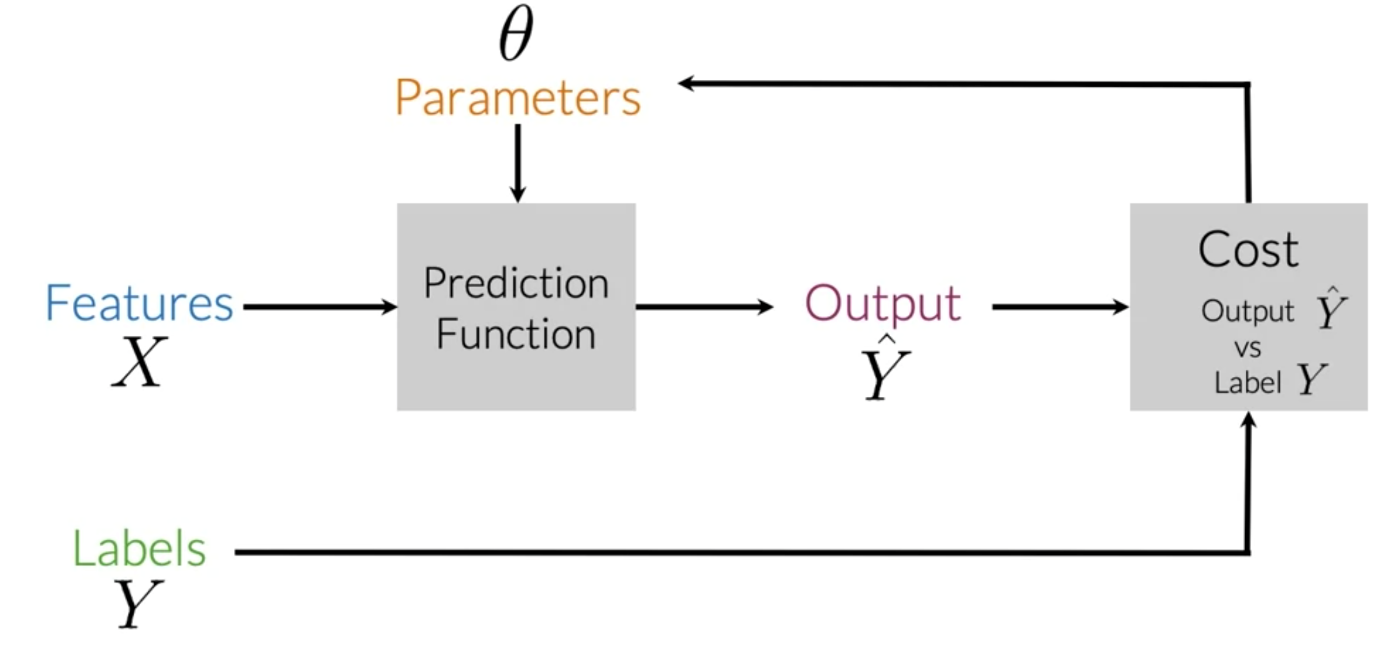
**Week 1 -**

**Supervised ML & Sentiment Analysis**

In supervised machine learning, you usually have an input X*X*, which goes into your prediction function to get your *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.

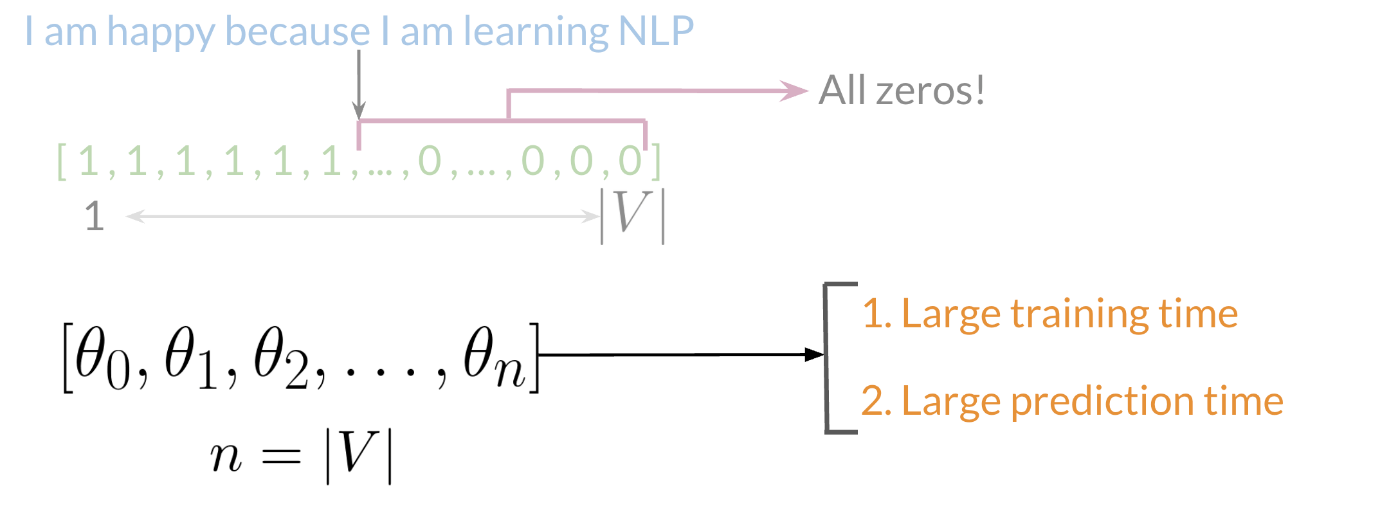
A diagram of a train class

Description automatically generated

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

**Vocabulary & Feature Extraction**

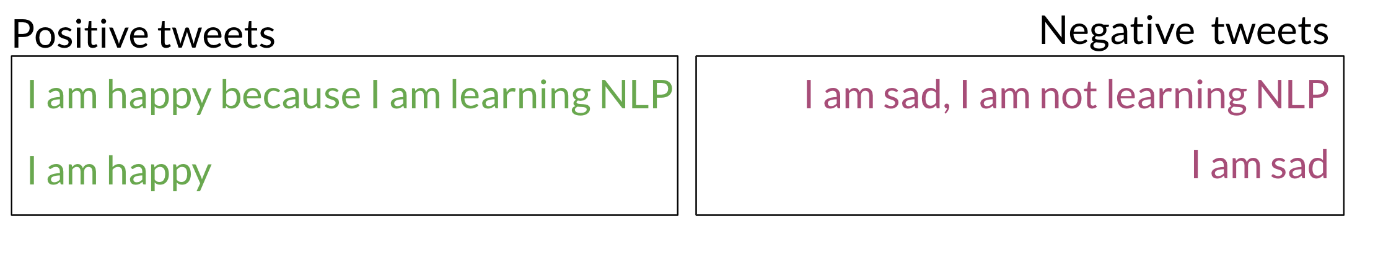
Given a tweet, or some text, you can represent it as a vector of dimension *V*, where *V* corresponds to your vocabulary size. If you had the tweet "I am happy because I am learning NLP", then you would put a 1 in the corresponding index for any word in the tweet, and a 0 otherwise.



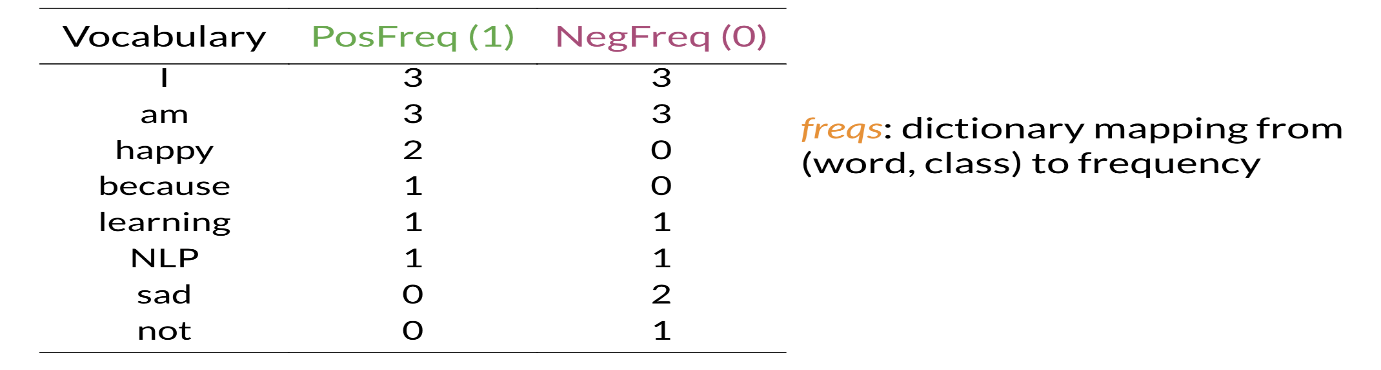
As you can see, as *V* gets larger, the vector becomes more sparse. Furthermore, we end up having many more features and end up training *θ* *V* parameters. This could result in larger training time, and large prediction time.

**Feature Extraction with Frequencies**

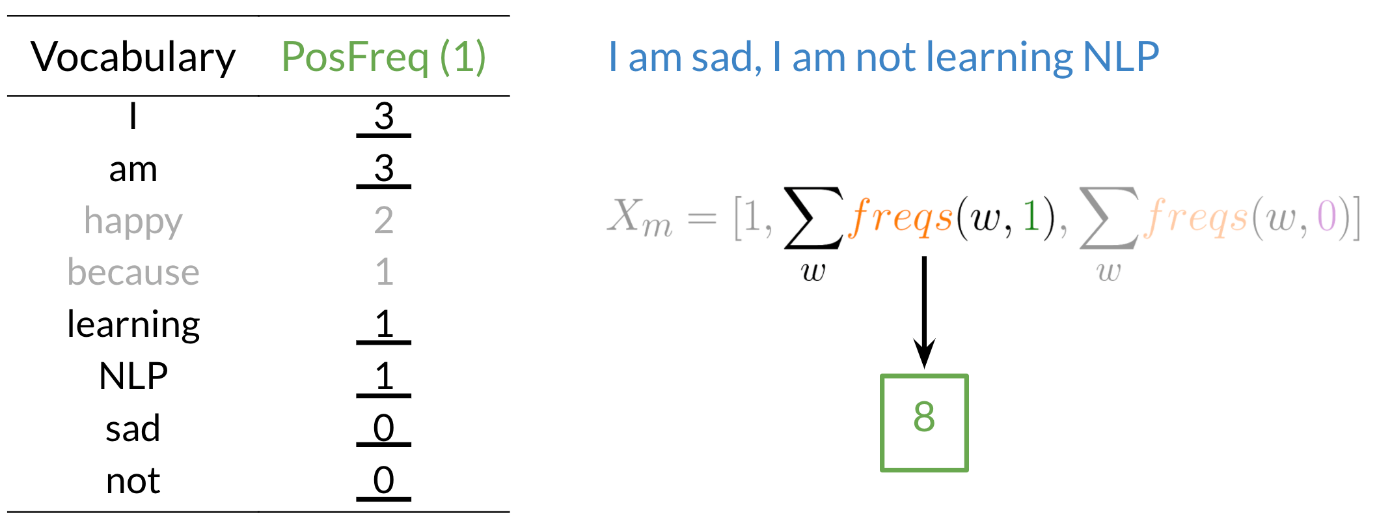
Given a corpus with positive and negative tweets as follows:



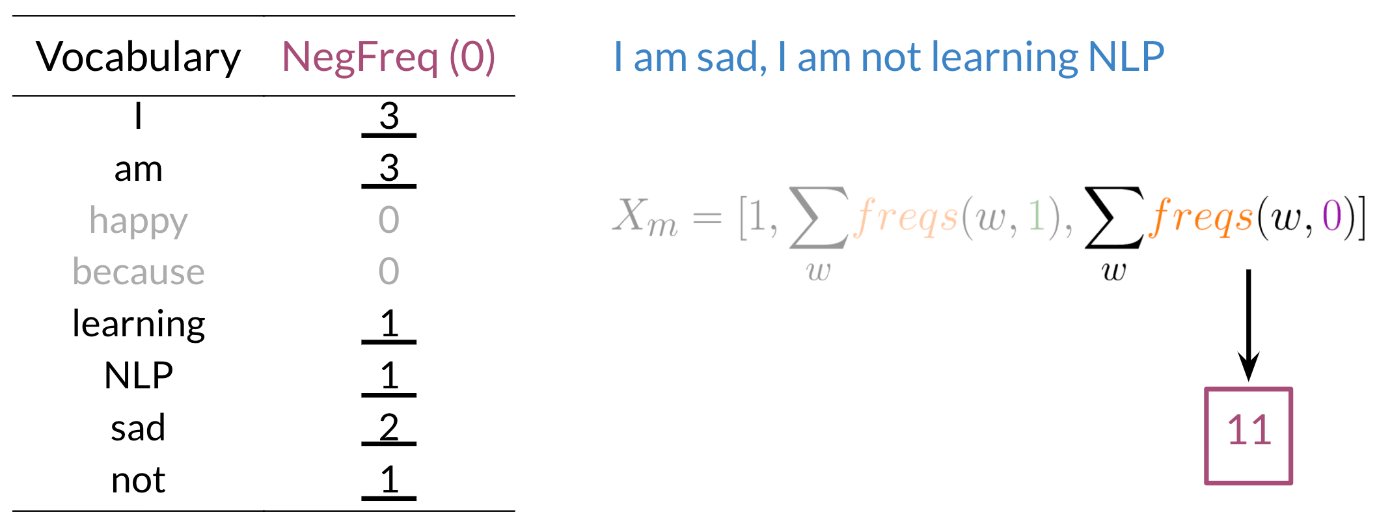
You have to encode each tweet as a vector. Previously, this vector was of dimension V*V*. Now, as you will see in the upcoming videos, you will represent it with a vector of dimension 33. To do so, you have to create a dictionary to map the word, and the class it appeared in (positive or negative) to the number of times that word appeared in its corresponding class.



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:



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



Hence you end up getting the following feature vector [1,8,11] [1,8,11]. 11 corresponds to the bias, 88 the positive feature, and 1111 the negative feature.

**Preprocessing**

When preprocessing, 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 preprocessing becomes

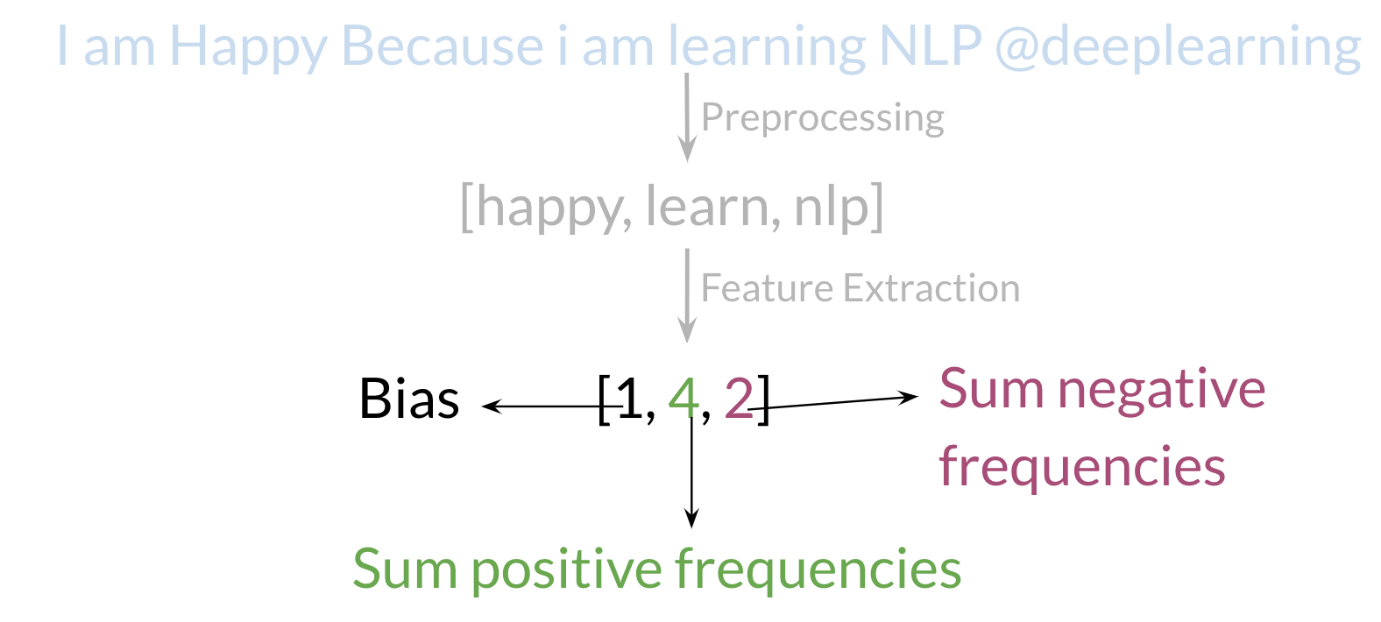
A screenshot of a computer

Description automatically generated

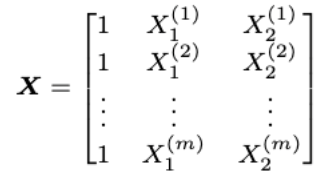
[tun,great,ai,model][*tun*,*great*,*ai*,*model*]. Hence you can see how we eliminated handles, tokenized it into words, removed stop words, performed stemming, and converted everything to lower case.

**Putting it all together**

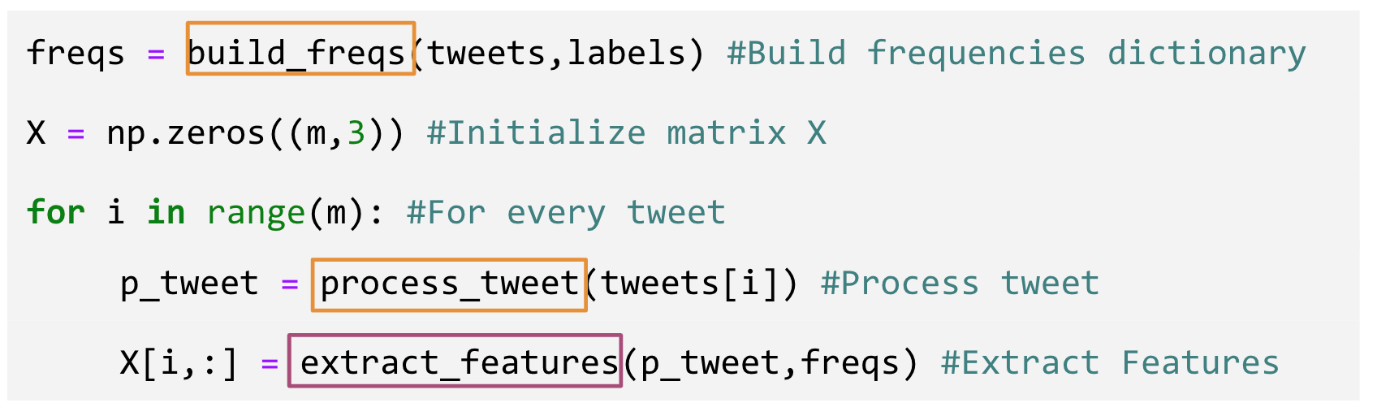
Overall, you start with a given text, you perform preprocessing, then you do feature extraction to convert text into numerical representation as follows:



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



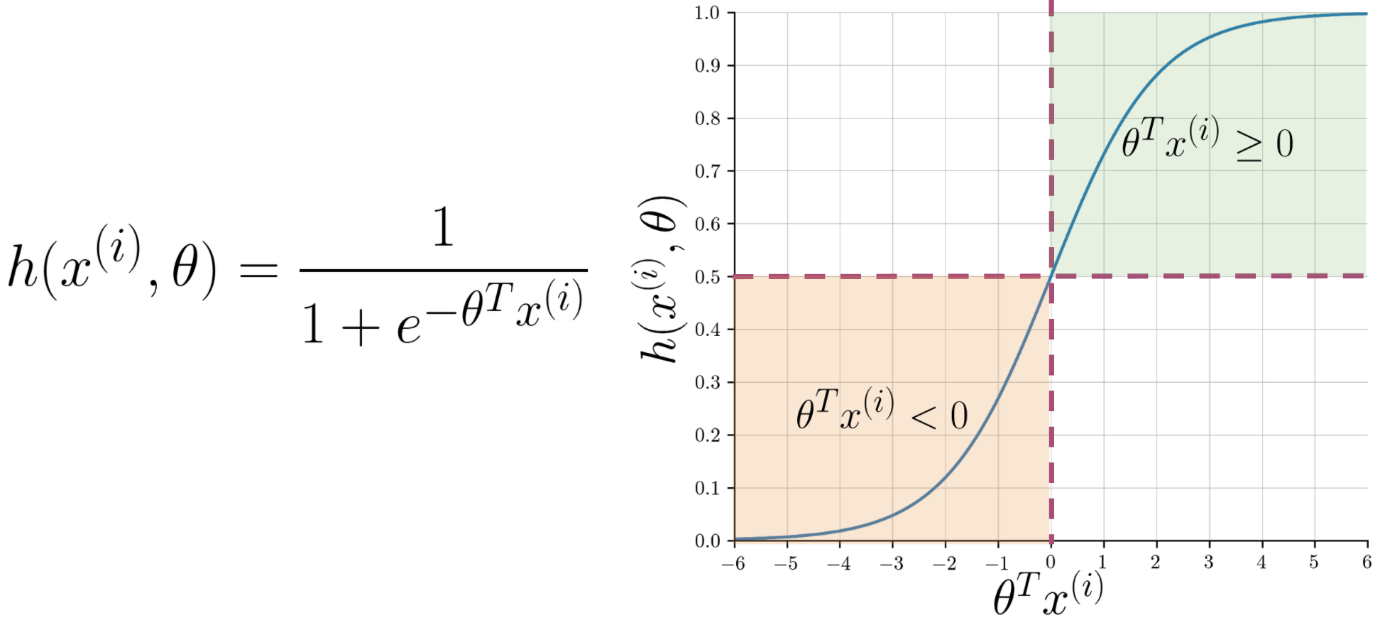
When implementing it with code, it becomes as follows:



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

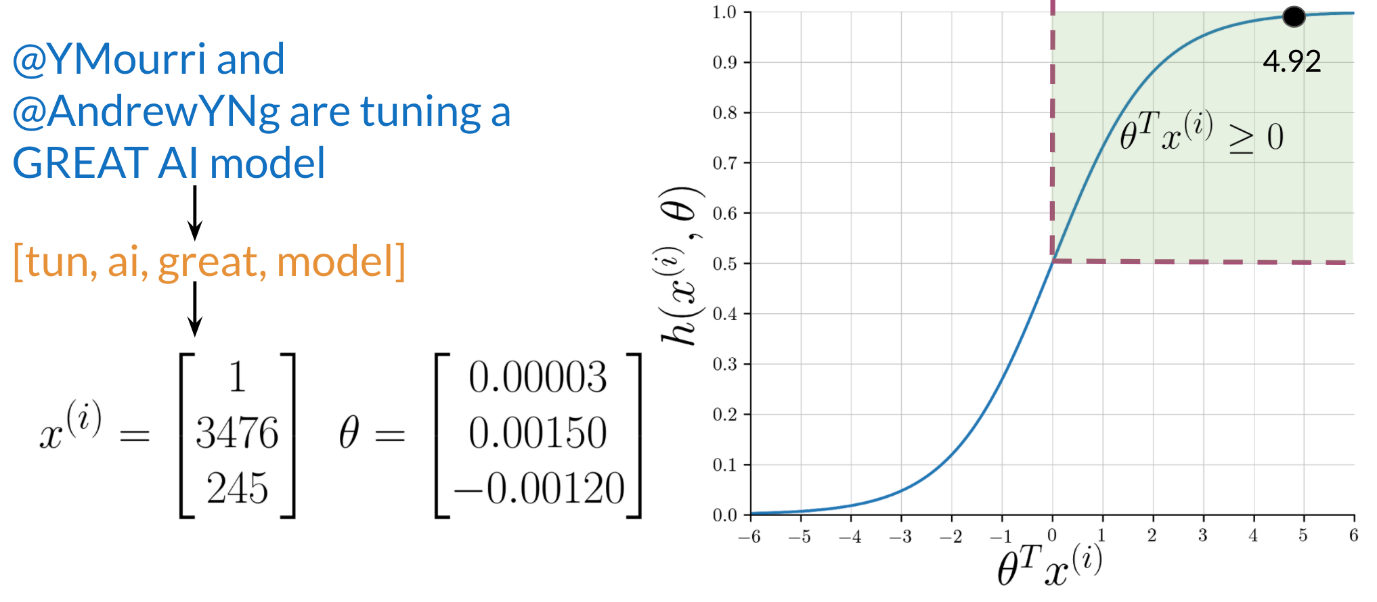
**Logistic Regression Overview**

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.



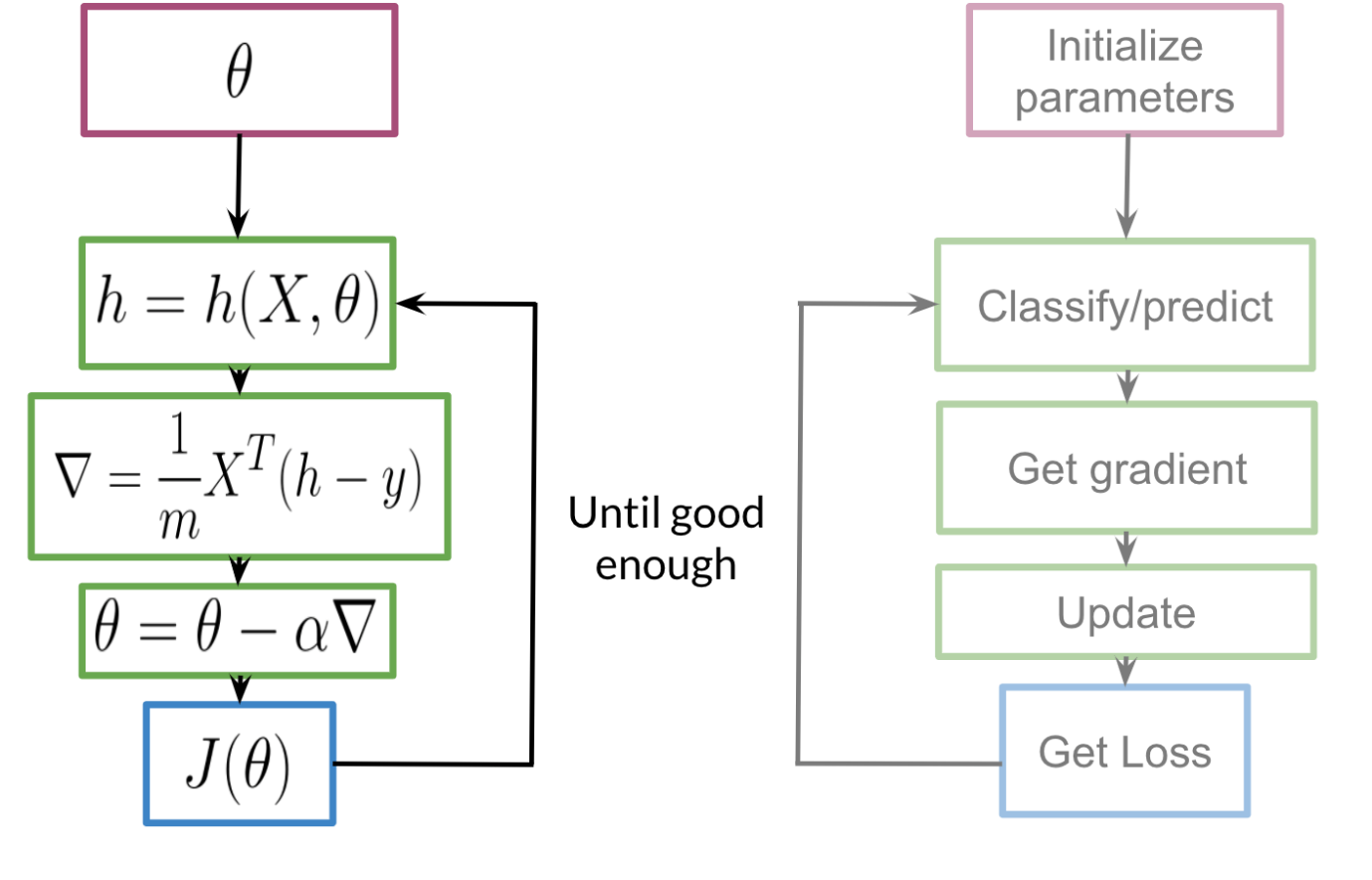
Note that as *θTx*(*i*) gets closer and closer to −∞ the denominator of the sigmoid function gets larger and larger and as a result, the sigmoid gets closer to 0. On the other hand, as *θTx*(*i*) gets closer and closer to ∞ the denominator of the sigmoid function gets closer to 1 and as a result the 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:



**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:** In a nutshell, the gradient allows you to learn what *θ* is so that you can predict your tweet sentiment accurately.

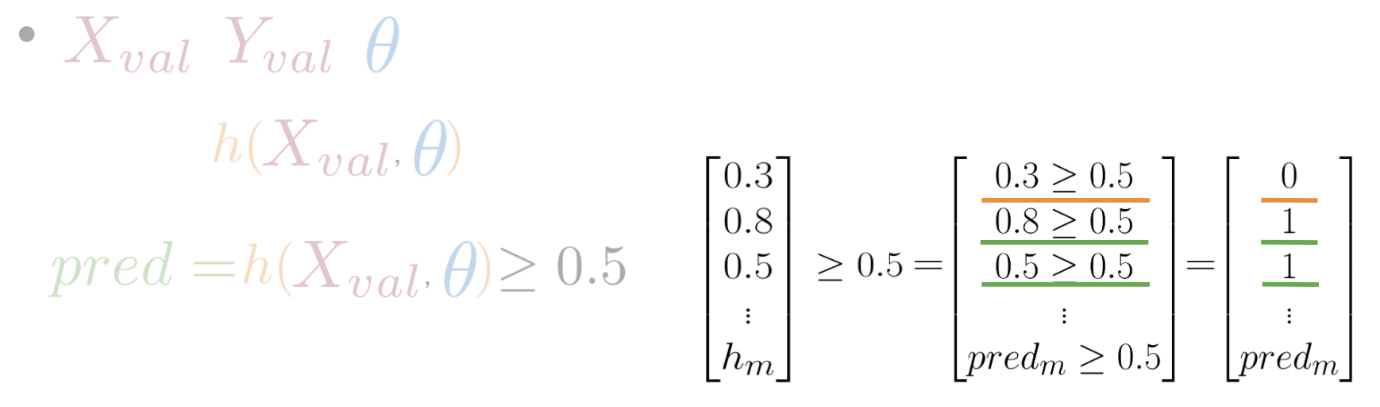
Usually, you keep training until the cost converges. If you were to plot the number of iterations versus the cost, you should see something like this:

A graph of cost and separation

Description automatically generated with medium confidence

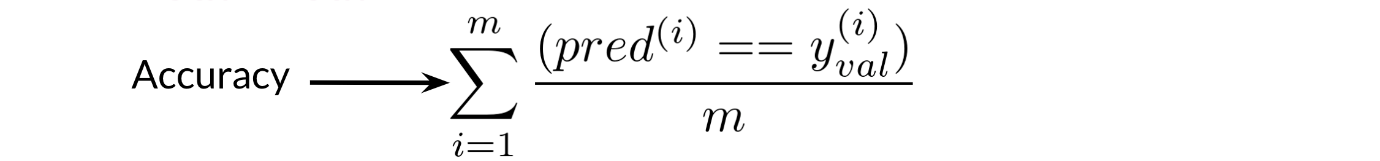
**Logistic Regression: Testing**

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.



In the video, I briefly mentioned *X* validation. In reality, given your *X* data you would usually split it into three components. *Xtrain*​, *Xval*​, *Xtest*​. 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:

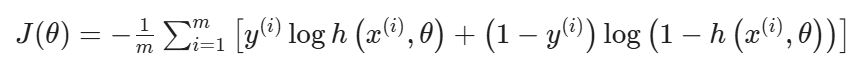


In other words, you go over all your training examples, *m* of them, and then for every prediction, if it was right you add a one. You then divide by *m*

**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.

The logistic regression cost function is defined as



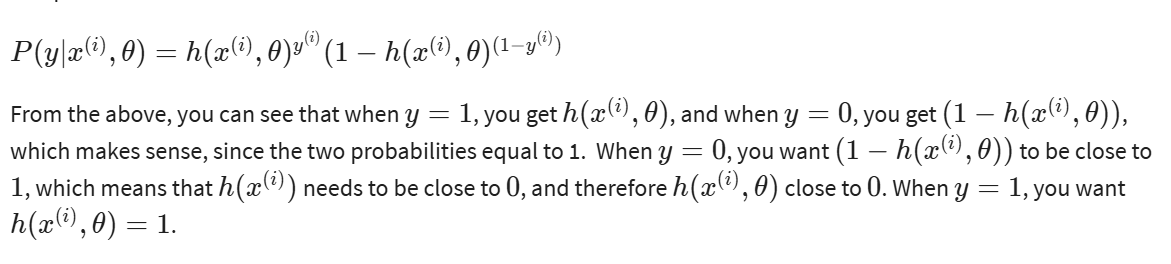
A graph of a mathematical equation

Description automatically generated with medium confidence

As you can see in the picture above, if *y*=1 and you predict something close to 0, you get a cost close to ∞. The same applies for then *y*=0 and you predict something close to 11. On the other hand, if you get a prediction equal to the label, you get a cost of 0. In either, case you are trying to minimize *J*(*θ*).

**Math Derivation**

To show you why the cost function is designed that way, let us take a step back and write up a function that compresses the two cases into one case.



A white background with black text

Description automatically generated

A white text with black text

Description automatically generated

A math equations on a white background

Description automatically generated

A screenshot of a math problem

Description automatically generated