Contents

1 Classification and Vector Spaces

1.1 Intro to Supervised ML and Sentiment Analysis

In supervised ML, we have input features X and a set of labels Y. To get the most accurate predictions, we try to minimize our *error rates* or *cost function* as much as possible: to do this, we'll run our prediction function which takes in parameters θ to map you input features to output labels $\hat{Y}$. The best mapping is achieved when the difference between the expected values Y and the predicted values $\hat{Y}$ is minimized, which the cost function does by comparing how closely your output $\hat{Y}$ is to your label Y. You can then update your parameters and repeat the whole process until your cost is minimized.

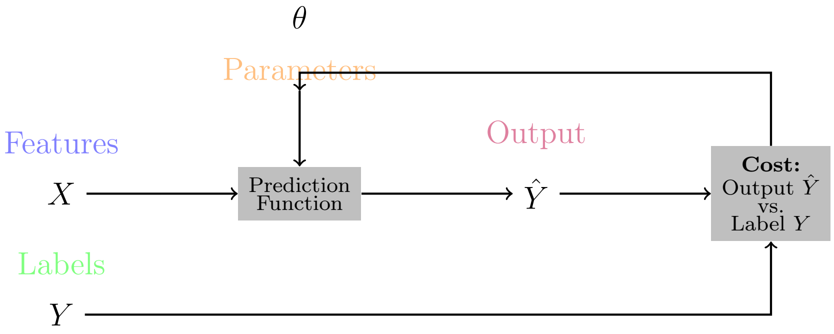


Figure 1: Overview of supervised machine learning.

How about the supervised ML classification task of sentiment analysis? Suppose we're given a tweet that says, "I'm happy because I'm learning NLP": and the objective in the task is to predict whether a tweet has a positive or negative sentiment. We'll do this by starting with a training set where tweets with a positive label have a label of unit value, and tweets with a negative sentiment have a label of zero. To get started building a logistic regression classifier that's capable of predicting sentiments of an arbitrary tweet, we first need to process the raw tweets in our training data set and extract useful features. Then, we will train our logistic regression classifier while minimimizing the cost. Finally, we'll be able to make predictions.

1.1.1 Representation of Text

How to represent text as a vector In order to represent text as a vector, we need to first build a vocabulary. We define the vocabulary V as the *set* of unique words from your input data (e.g. your listing of tweets). To get this listing, we quite literally need to comb through all words from all input data and save every new word that appears in our search. To represent a tweet as a vector, we can use a one-hot encoding with our vocabulary: i.e. each tweet will be represented with a length $|V|$ vector where elements are binary-valued - a one indicates the word is in the tweet and a zero indicates the absence of a word in a tweet. We call this a *sparse* representation because the number of non-zero entries is relatively small when compared with the number of zero entries. Realize that if we are running a logistic regression, we would require learning $|V| + 1$ parameters which can be problematic for large vocabularies. If not prohibitive, it would make training models take excessive time and making predictions would be expensive.

Negative and positive frequencies Let's discuss how to generate counts which can be used as features in our logistic regression classifier. Specifically, given a word, we wish to keep track of the number of times that it shows up as the positive class. Given another word, we wish to track how many times that word shows up in the negative class. Using both these counts, we can then extract features and use those features in our logistic regression classifier. Suppose we have the following corpus of tweets:

```
I am happy because I am learning NLP
I am happy
I am sad, I am not learning NLP
I am sad
```

Then our vocabulary is given by

Vocabulary
I
am
happy
because
learning
NLP
sad
not

For this particular example of sentiment analysis, we only have two sentiments (i.e. two classes): one class is associated with a positive sentiment and the other with a negative sentiment. So, taking your corpus, you'd have a set of two tweets that belong to the positive class, and two tweets which belong to the negative class. Let's calculate the positive frequencies by examining the first two tweets:

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

The same logic applies applies to getting negative frequencies. We can calculate these by examining our last two training examples.

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

So, we can now have an entire table for our corpus, where for each entry in V we associate with it a scalar value `PosFreq(1)` and another scalar value `NegFreq(0)`. In practice, we use a Python dictionary `freqs` mapping from `(word, class)` $\rightsquigarrow$ frequency.

1.1.2 Feature Extraction with Frequencies

Whereas we previously learned to encode a tweet as a vector of length $|V|$, we will now use our frequency counts to represent each tweet as a vector of length equal to one plus the number of classes in our set of labels. This gives us a much faster speed for our logistic regression classifier. How can we do this, exactly? We represent each tweet as follows:

$$\underbrace{X_m}_{\substack{\text{Features of}\\ \text{tweet } m}} = \left[\underbrace{1}_{\text{Bias}}, \underbrace{\sum_w \texttt{freqs}(w,1)}_{\substack{\text{Sum Pos.}\\ \text{Frequences}}}, \underbrace{\sum_w \texttt{freqs}(w,0)}_{\substack{\text{Sum Neg.}\\ \text{Frequencies}}}\right]$$

I.e. the first feature is a bias unit equal to unit value, the second is the sum of positive frequencies for every unique word on tweet m, and the third is the sum of negative frequencies for every unique word on the tweet. So, to extract the features for this *representation*, we only have to sum frequencies of words, which is straightforward. Let’s look at an example: “I am sad, I am not learning NLP”. The only words in our vocabulary that don’t appear in this sentence are “happy” and “because”: if we sum up the `PosFreq(1)` associated with the remaining words in our vocabulary, i.e. the words that appear in this tweet, we get a scalar value of eight. We do the same for the negative frequencies, and we get a scalar value of eleven. So, we represent “I am sad, I am not learning NLP” $\rightsquigarrow [1, 8, 11]$.

1.1.3 Preprocessing

There are two major concepts here: stemming and “stop words”. We’ll learn how to apply these preprocessing steps to our data.

Stop words Stop words are defined as those which don’t add significant meaning to the tweets; we *might* also choose to remove punctuation (if we decide it doesn’t provide information in our context). In practice, this means comparing our tweet against two sets: one with stop words (in English) and another with punctuation.

Stop Words
and
is
are
at
has
for
a

Punctuation
,
.
:
!
“
’

In practice the list of stop words and punctuation marks are much larger, but for pedagogical purposes these will serve well. We might start out with a tweet like

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

We then preprocess by stripping stop words “and”, “are”, a”at”, and “a”. The only punctuation that appears in this tweet that’s also in our list is the exclamation point(s). We might further decide that tweets should have handles and URLs removed, because these don’t add value for the specific task of sentiment analysis. In the end, we end up with a data point that looks like

```
tuning GREAT AI model
```

It’s clearly a positive tweet, and a sufficiently good model should be able to classify it. Now that the tweet contains the minimum necessary information, we can perform *stemming* for every word.

Stemming Stemming in NLP is simply transforming any word to its base stem, which you could define as the set of characters that are used to construct the words and its derivatives. Let's look at the first word in the example: its stem is "tun", since

$$\begin{tikzcd}
 & \text{tune} \\
\text{tun} \arrow[ur] \arrow[r] \arrow[dr] & \text{tuned} \\
 & \text{tuning}
\end{tikzcd}$$

If we were to perform stemming on our entire corpus, the words "tune", "tuned", and "tuning" all get reduced to the stem "tun". So, your vocabulary would be significantly reduced in performing this process. You can further reduce the size of the vocabulary without losing valuable information by *lower-casing* every word, e.g. the words "GREAT", "Great", and "great" all get treated as the same word. Perhaps our final preprocessed tweet looks like

`[tun, great, ai, model]`

In summary, for our example of sentiment analysis on tweets, we might preprocess as follows:

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. E.g. dancer, dancing, danced, becomes "danc". You can use Porter Stemmer to take care of this.
5. Convert all words to lowercase.

As an applied example:

I am Happy Because I am learning NLP @deeplearning $\xrightarrow{\text{Preprocessing}}$ `[happy, learn, nlp]` $\xrightarrow{\text{Feature Extraction}}$ $[1, 4, 2]$

where 1 is our bias term, 4 is the sum of positive frequencies, and 2 is the sum of negative frequencies. In practice, we are given a set of m raw tweets, and so wehave to process them one-by-one to process them into an $m \times 3$ matrix, where each row describes the features for a given tweet.

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

The process is simple: (i) build the frequencies dictionary, (ii) initialize the matrix X to match the number of tweets, (iii) go through your sets of tweets and carefully preprocess by deleting stop words, stemming, deleting URLs/handles, and lowercasing, and finally (iv) extract the features by summing up the positive and negative frequencies of each of the tweets.

```
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.
```

1.2 Logistic Regression

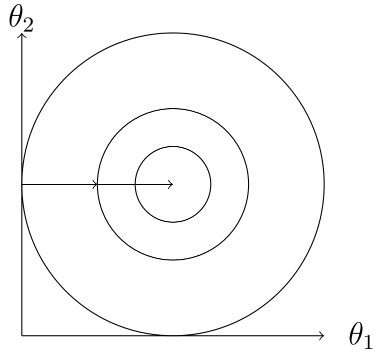
Previously, we've learned how to extract features, which we will now use to predict whether a tweet has a positive or negative sentiment. Logistic regression makes use of a sigmoid (or standard logistic) function which outputs a probability between zero and one. What's the recap from supervised machine learning? Recall figure 1.1: in the case of logistic regression our prediction function is going to be the standard logistic function:

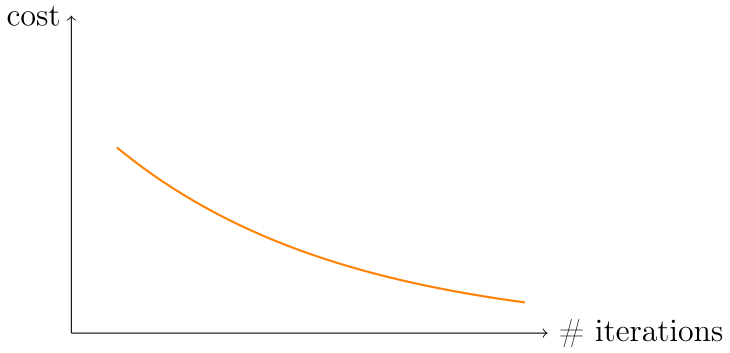
$$h(x^{(i)}, \theta) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}.$$

where i denotes the observation number. Note that as $\theta^T x^{(i)}$ gets closer and closer to $-\infty$, the denominator of the sigmoid expression blows up and as a result the output values gets closer to zero. Conversely, as the inner product $\theta^T x^{(i)}$ gets closer to ∞, the denominator of the sigmoid function approaches unit value and the resulting sigmoid expression evaluates to something near one. For classification, a threshold is needed, and it is natural to set it at $\frac{1}{2}$. For the logistic function, this threshold occurs when the inner product $\theta^T x^{(i)} = 0$. If the inner product is greater than (or equal to) zero, we classify as positive, else negative.

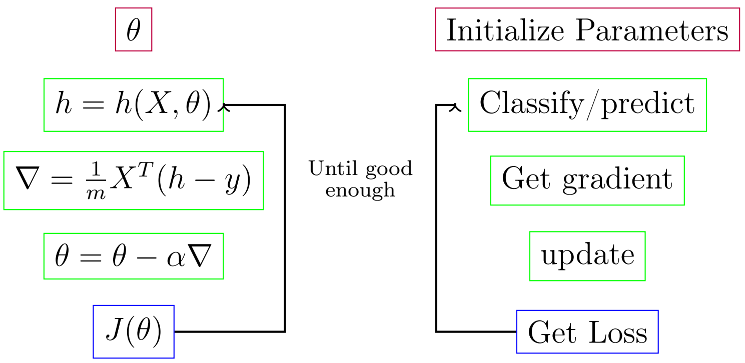
1.2.1 Learning Parameters

How to learn θ? To train a logistic regression classifier, we need to iterate until we find a set of parameters θ that minimizes our cost function. Suppose we have a loss that depends only on the parameters θ_1, θ_2: you might have a cost function that looks like follows, on the left, with the evaluation of the cost function plotted on the right as a function of the number of training iterations:





We might first initialize our parameters θ, then update our parameters in the direction of the *gradient of the cost function*. After a sufficient number of training steps, we will have updated θ to their optimal values where we are achieving near optimal cost. Let's quickly review this process of gradient descent for logistic regression:



1.2.2 Assessing model generalization

To analyze model fit, we need the following: $(X_{\text{val}}, Y_{\text{val}}, \theta)$, where we have *validation* data that was set aside during training, and a learned θ parameter vector. We will compute, for each example in X_{val}, the value of $h(\theta, x^{(i)})$ and compare it with our threshold value to make a prediction. In particular, our simple prediction function is given by

$$\hat{Y}_{\text{val}} = h(X_{\text{val}}, \theta) \geq \frac{1}{2}.$$

In particular, we will have a vector $h = \begin{bmatrix} h_1 & h_2 & \dots & h_m \end{bmatrix}$ where e.g. h_i could equal some float in $[0, 1]$, which we then convert into a binary label vector by applying our threshold. After building our predictions vector $\hat{Y}_{\text{val}}$, we can compare the predictions with the actual values and evaluate our test-set *accuracy*:

$$\texttt{Accuracy} = \sum_{i=1}^{m} \frac{\left(\texttt{pred}^{(i)} == \hat{Y}_{\text{val}}^{(i)}\right)}{m}.$$

This metric gives an estimate of the number of times of logistic regression model will work correctly on unseen data.

1.2.3 Deriving Gradient Descent for Logistic Regression

Motivating where cost function comes from Let's examine the equation for the cost function for logistic regression:

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

The deep learning notes derive this equation in detail in the introduction. Let us briefly recap.

$$\Pr(y | x^{(i)}, \theta) = h(x^{(i)}, \theta) y^{(i)} \left(1 - h(x^{(i)}, \theta)\right)^{(1 - y^{(i)})}.$$

We wish to maximize our function $h(\cdot, \theta)$ over the parameter space θ: when $y = 0$ we want $(1 - h(x^{(i)}, \theta))$ to be zero, and therefore $h(x^{(i)}, \theta)$ close to one. When $y = 1$, we want $h(x^{(i)}, \theta) = 1$. To model our entire dataset and not just one observation, we make an assumption of independence to arrive at a joint likelihood:

$$L(\theta) = \prod_{i=1}^{m} h(x^{(i)}, \theta)^{y^{(i)}} \left(1 - h(x^{(i)}, \theta)\right)^{(1 - y^{(i)})}.$$

Realize that if we "mess up" one prediction, we have the potential to "mess up" the entire cost function, which is what we want: we want a model that captures the entire dataset, where all datapoints are related. One issue: what happens when m grows? Then $L(\theta) \rightsquigarrow 0$, because the expressions $h(x^{(i)}, \theta)$ and correspondingly $(1 - h(x^{(i)}, \theta)$ are bounded between $(0, 1)$.

Optimization Using properties of logarithms (that they are monotone and maximizing a function under a monotone transformation doesn't change the optimum, and that they turn multiplication into addition), i.e.

$$\log(a * b * c) = \log a + \log b + \log c \quad \text{and} \quad \log a^b = b \log a.$$

We may now rewrite our optimization problem:

$$
\begin{aligned}
\max_{h(x^{(i)},\theta)} \log L(\theta) &= \log \prod_{i=1}^{m} h(x^{(i)},\theta)^{y^{(i)}} \left(1 - h(x^{(i)},\theta)\right)^{1-y^{(i)}} \\
&= \sum_{i=1}^{m} \log h(x^{(i)},\theta)^{y^{(i)}} \left(1 - h(x^{(i)},\theta)\right)^{1-y^{(i)}} \\
&= \sum_{i=1}^{m} \log h(x^{(i)},\theta)^{y^{(i)}} + \log \left(1 - h(x^{(i)},\theta)\right)^{1-y^{(i)}} \\
&= \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)},\theta) + (1-y^{(i)}) \log \left(1 - h(x^{(i)},\theta)\right)
\end{aligned}
$$

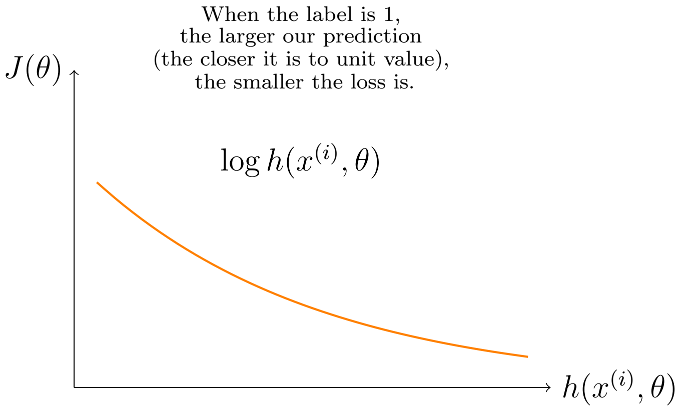
We can then rescale by $\frac{1}{m}$ to get *average* cost. Recall we are maximizing over $h(x^{(i)},\theta)$ in the equation above, and maximizing an equation is the same as minimizing its negative. Therefore,

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

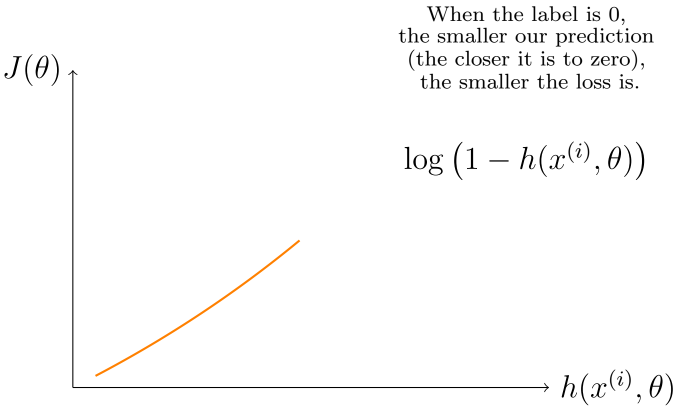
A vectorized implementation is given by

$$
\begin{aligned}
h &= g(X\theta) \\
J(\theta) &= \frac{1}{m} \cdot \left(-y^T \log(h) - (1-y)^T \log(1-h) \right)
\end{aligned}
$$

Intuition for loss function of logistic regression Now, let's just go over some intuition here. Consider the term on the left-hand side of the parenthesized expression: this is the relevant term in your cost function when your label is 1. The term on the right is relevant when the label is zero. In general, this loss function simply says: the closer the prediction is to the observed label, the smaller the loss incurred. We can plot the cost as a function of our the prediction value for a single training example.



When the label is 1,
the larger our prediction
(the closer it is to unit value),
the smaller the loss is.



When the label is 0,
the smaller our prediction
(the closer it is to zero),
the smaller the loss is.

Deriving logistic regression gradient The general form of logistic regression is given by

```
Algorithm 1: General form of gradient descent
while not converged, and for all $j$ do
    $\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
end
```

We can work out the derivative using partial calculus to fill in the expression further:

```
Algorithm 2: Gradient descent for logistic regression
while not converged, and for all $j$ do
    $\theta_j \leftarrow \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left( h(x^{(i)}, \theta) - y^{(i)} \right) x_j^{(i)}$
end
```

A vectorized implementation is given by

$$\theta := \theta - \frac{\alpha}{m} X^T \left(H(X, \theta) - Y \right).$$

Partial derivative of $J(\theta)$ It'll be helpful to first calculate the derivative of the sigmoid function.

$$\begin{aligned} h(x)' &= \left(\frac{1}{1+e^{-x}} \right)' = \frac{-(1+e^{-x})'}{(1+e^{-x})^2} = \frac{-1' - (e^{-x})'}{(1+e^{-x})^2} = \frac{0 - (-x)'(e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \left(\frac{1}{1+e^{-x}} \right) \left(\frac{e^{-x}}{1+e^{-x}} \right) = h(x) \left(\frac{+1-1+e^{-x}}{1+e^{-x}} \right) = h(x) \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}} \right) = h(x)(1-h(x)). \end{aligned}$$

The above was all for a computation of the derivative of the sigmoid function. But what about the derivative of $h(x^{(i)}, \theta) = \frac{1}{1+e^{-\theta^T x^{(i)}}}$ with respect to θ_j? Using the chain rule, because of the inner product $\theta^T x^{(i)}$, and applying toward θ_j, we see that the derivative would be

$$h(x^{(i)}, \theta) \left(1 - h(x^{(i)}, \theta) \right) x_j^{(i)}.$$

Now, we can compute the partial derivative of our loss function with respect to θ_j:

$$\begin{aligned}
\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{-1}{m}\sum_{i=1}^{m}\left[y^{(i)}log(h(x^{(i)},\theta)) + (1-y^{(i)})log(1-h(x^{(i)},\theta))\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\frac{\partial}{\partial\theta_j}log(h(x^{(i)},\theta)) + (1-y^{(i)})\frac{\partial}{\partial\theta_j}log(1-h(x^{(i)},\theta))\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{y^{(i)}\frac{\partial}{\partial\theta_j}h(x^{(i)},\theta)}{h(x^{(i)},\theta)} + \frac{(1-y^{(i)})\frac{\partial}{\partial\theta_j}(1-h(x^{(i)},\theta))}{1-h(x^{(i)},\theta)}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{y^{(i)}\frac{\partial}{\partial\theta_j}h(x^{(i)},\theta)}{h(x^{(i)},\theta)} + \frac{(1-y^{(i)})\frac{\partial}{\partial\theta_j}(1-h(x^{(i)},\theta))}{1-h(x^{(i)},\theta)}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{y^{(i)}h(x^{(i)},\theta)(1-h(x^{(i)},\theta))\frac{\partial}{\partial\theta_j}\theta^T x^{(i)}}{h(x^{(i)},\theta)} + \frac{-(1-y^{(i)})h(x^{(i)},\theta)(1-h(x^{(i)},\theta))\frac{\partial}{\partial\theta_j}\theta^T x^{(i)}}{1-h(x^{(i)},\theta)}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[\frac{y^{(i)}h(x^{(i)},\theta)(1-h(x^{(i)},\theta))\frac{\partial}{\partial\theta_j}\theta^T x^{(i)}}{h(x^{(i)},\theta)} - \frac{(1-y^{(i)})h(x^{(i)},\theta)(1-h(x^{(i)},\theta))\frac{\partial}{\partial\theta_j}\theta^T x^{(i)}}{1-h(x^{(i)},\theta))}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}(1-h(x^{(i)},\theta))x_j^{(i)} - (1-y^{(i)})h(x^{(i)},\theta)x_j^{(i)}\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}(1-h(x^{(i)},\theta)) - (1-y^{(i)})h(x^{(i)},\theta)\right]x_j^{(i)} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} - y^{(i)}h(x^{(i)},\theta) - h(x^{(i)},\theta) + y^{(i)}h(x^{(i)},\theta)\right]x_j^{(i)} \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} - h(x^{(i)},\theta)\right]x_j^{(i)} \\
&= \frac{1}{m}\sum_{i=1}^{m}\left[h(x^{(i)},\theta) - y^{(i)}\right]x_j^{(i)}
\end{aligned}$$

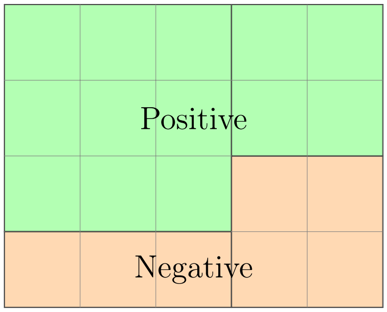
The vectorized version is simply given by

$$\nabla J(\theta) = \frac{1}{m}\cdot X^T\left(H(X,\theta) - Y\right).$$

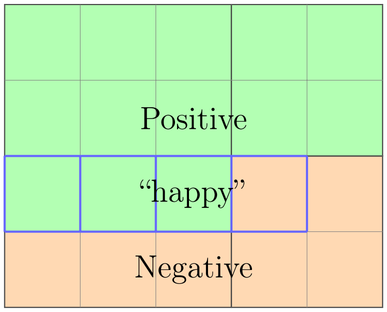
2 Naive Bayes

2.1 Probability and Bayes Rule

Imagine you have an extensive corpus of tweets that can be categorized as either positive or negative, but not both.



Corpus of tweets



Tweets containing the word “happy”

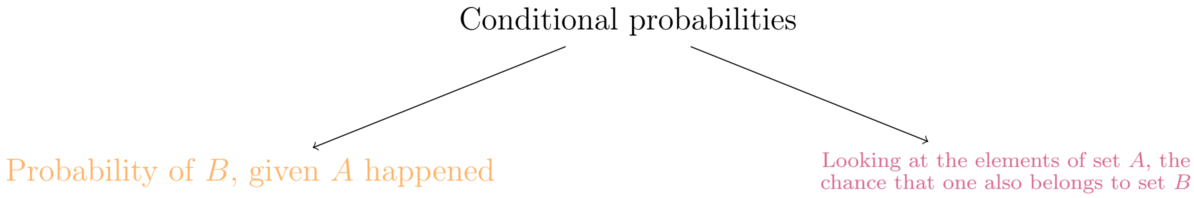
Within this corpus, observe that the word “happy” is sometimes associated with a positive sentiment, but also sometimes with a negative sentiment! How can this happen? Let’s explore the situation using probabilities.

Intro probability Suppose we define an event A as a positive tweet. Then, $\Pr(A) = \Pr(\text{Positive}) = \frac{N_{\text{pos}}}{N}$, i.e. the ratio between the number of positive-sentiment tweets relative to the total number of tweets observed in our data. In our example above, it comes out $\frac{13}{20}$.[1] Let’s define an event B as a tweet containing the word “happy”. In our examples above, this happens to be four, i.e. $\Pr(B) = \Pr(\text{“happy”}) = \frac{N_{\text{happy}}}{N}$ is $\frac{4}{20}$. Building on this, $\Pr(A \wedge B) = \Pr(A, B)$ which happens to be $\frac{3}{20}$ in our pictorial example above.

Deriving Bayes rule What if instead of the entire corpus, we only consider tweets that contain the word “happy”? $\Pr(A|B) = \Pr(\text{Positive}|\text{“happy”}) = \frac{3}{4}$ in our example. But we can also do the same thing for positive tweets, i.e.

$$\Pr(B|A) = \Pr(\text{“happy”}|\text{Positive}) = \frac{3}{13}.$$

There are two equivalent ways of thinking about conditional probabilities:


Conditional probabilities
Probability of B, given A happened
Looking at the elements of set A, the chance that one also belongs to set B

So, in our context:

$$\Pr(\text{Positive}|\text{“happy”}) = \frac{\Pr(\text{Positive} \wedge \text{“happy”})}{\Pr(\text{“happy”})}. \tag{2}$$

And by symmetry:

$$\Pr(\text{“happy”}|\text{Positive}) = \frac{\Pr(\text{“happy”} \wedge \text{Positive})}{\Pr(\text{Positive})}. \tag{3}$$

Realize that the intersection operation between two events is symmetric, i.e. that the numerator in equations 2 and 3 are identical. Therefore,

$$\Pr(\text{Positive}|\text{“happy”}) = \Pr(\text{“happy”}|\text{Positive}) \times \frac{\Pr(\text{Positive})}{\Pr(\text{“happy”})}. \tag{4}$$

This is an expression of Bayes rule in the context of the sentiment analysis problem. More generally: Bayes rule specifies that $\Pr(X|Y) = \Pr(Y|X) \times \frac{\Pr(X)}{\Pr(Y)}$.

[1] Of course, because tweets can only be positive or negative but not both, the probability of a negative tweet is simply the complement, i.e. $\Pr(\text{Negative}) = 1 - \Pr(\text{Positive})$ which in our example happens to be $\frac{7}{20}$.

2.2 Naive Bayes

Naive Bayes is often a "very good, quick, and dirty baseline" for many text classification tasks; it's an example of supervised machine learning and as such shares many similarities with logistic regression. It's called Naive because it makes the assumption that the features you're using for classification are all independent, which in reality is *rarely* the case. As per usual, we start with two corpora: one for the positive tweets and one for the negative tweets:

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	13

The above word frequencies table is the backbone input to our naive bayes algorithm: it allows us to compute conditional probabilities. E.g. $\Pr(\text{I}|\text{Pos}) = \frac{3}{13}$. We can do this for each word in our vocabulary, i.e. compute the conditional probability of it appearing in each class. Notice that if you sum over the probabilities for a particular class, you get 1.

word	Pos	Neg
I	$\frac{3}{13}$	$\frac{3}{13}$
am	$\frac{3}{13}$	$\frac{3}{13}$
happy	$\frac{2}{13}$	$\frac{1}{13}$
because	$\frac{1}{13}$	0
learning	$\frac{1}{13}$	$\frac{1}{13}$
NLP	$\frac{1}{13}$	$\frac{1}{13}$
sad	$\frac{1}{13}$	$\frac{2}{13}$
not	$\frac{1}{13}$	$\frac{2}{13}$
Sum	1	1

Let's inspect some of the entries: notice that for a few words in the vocabulary, their conditional probabilities of appearing in either class are (nearly) identical: words that are equally probable don't add anything to the sentiment. On the other hand, words like happy, or sad, not are "power" words which tend to express one sentiment or another. These words carry a lot of weight in determining your tweet sentiments. As a separate note, examine the word `because`: it only appears in the positive corpus, and so its conditional probability for the negative class is zero: when this happens we have no way of comparing between the two corpora which will become a problem for subsequent calculations. We'll see how we can "smooth" our probability function.

Suppose we get a new tweet, "I am happy today; I am learning." and we want to classify its sentiment. We use the following expression:

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

So, for our tweet example, we have (word by word, and skipping "today" because it doesn't appear in our vocabulary):

$$\frac{\frac{3}{13}}{\frac{3}{13}} \times \frac{\frac{3}{13}}{\frac{3}{13}} \times \frac{\frac{2}{13}}{\frac{1}{13}} \times \frac{\frac{3}{13}}{\frac{3}{13}} \times \frac{\frac{3}{13}}{\frac{3}{13}} \times \frac{\frac{1}{13}}{\frac{1}{13}} = \frac{2}{13} > 1.$$

Because the ratio is greater than unit value, we conclude that overall the sentiment of the tweet is positive.

Laplacian smoothing This is a technique we use to avoid probabilities being identically zero. Typically, the expression used to calculate the conditional probability of a word, given the class, is

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

where $N_{\text{class}} =$ frequency of all words in class. Laplacian smoothing does the following; supposing $|V|$ is the number of unique words in the vocabulary

$$\Pr(w_i|\text{class}) = \frac{\texttt{freq}(w_i, \text{class}) + 1}{N_{\text{class}} + |V|}. \tag{5}$$

By adding a one to our numerator, we ensure the expression is non-zero. However, this is not correctly normalized by N_{class}, and so we add a new term to the denominator $|V|$; this ensures the probabilities all sum to one. E.g. in our example table in section 2.2 describing positive and negative word frequencies in our corpora of tweets, we can use this to compute

$$\Pr(\text{I}|\text{Pos}) = \frac{3+1}{13+8}.$$

We can apply Laplacian smoothing to every entry in our table and end up with a new table of conditional probabilities where the column-sums are unit valued. Notice that if we apply this technique to the word "because" in our example, and specificially for the negative class, that $\Pr(\text{because}|\text{Negative}) = \frac{0+1}{13+8} > 0$ which solved our original problem of getting a divide by zero in the formula for Naive Bayes $\prod_{i=1}^{m} \frac{\Pr(w_i|\text{Pos})}{\Pr(w_i|\text{Neg})}$.

Log likelihoods Words can have many shades of emotional meaning, but for the purpose of sentiment classification they can be simplified into three categories: neutral, positive, and negative. A word can be taxonomized according to its conditional probabilities. We simply calculate for each word

$$\text{ratio}(w_i) = \frac{\Pr(w_i|\text{Pos})}{\Pr(w_i|\text{Neg})} \approx \frac{\texttt{freq}(w_i, 1) + 1}{\texttt{freq}(w_i, 0) + 1}.$$

If this ratio is identically unit valued, the word is neutral. Words that are more positive tend to have higher ratios (larger than one), and words that are more negative tend to have lower ratios (less than one). Observe that the ratio can lie in $[0, \infty)$.

It turns out that in our previous formulation of Naive Bayes, we assumed balanced class sizes. The correct formula for the likelihood includes the prior ratio, which becomes important for unbalanced datasets (where e.g. the number of positive and negative tweets is not equal):

$$\frac{\Pr(\text{Pos})}{\Pr(\text{Neg})} \prod_{i=1}^{m} \frac{\Pr(w_i|\text{pos})}{\Pr(w_i|\text{neg})} \tag{6}$$

Recognize that this computation involves the product of many probabilities that lie in $(0, 1]$, and we run the risk of numerical underfluw if the number returned "is so small it can't be stored on your device". There is a nice mathematical trick that avoids this pitfall, and that's to use properties of logarithms: $\log(a \times b) = \log(a) + \log(b)$.

$$\log \frac{\Pr(\text{Pos})}{\Pr(\text{Neg})} \prod_{i=1}^{m} \frac{\Pr(w_i|\text{pos})}{\Pr(w_i|\text{neg})} \rightsquigarrow \underbrace{\log \frac{\Pr(\text{Pos})}{\Pr(\text{Neg})}}_{\text{log prior}} + \underbrace{\sum_{i=1}^{m} \log \frac{\Pr(w_i|\text{Pos})}{\Pr(w_i|\text{Neg})}}_{\text{log likelihood}}.$$

Let $\lambda(w) = \log \frac{\Pr(w|\text{Pos})}{\Pr(w|\text{Neg})}$; we calculate this for each word in our vocabulary. Realize that neutral words (i.e. ones where $\Pr(w|\text{Pos}) = \Pr(w|\text{Neg})$) have $\lambda(w) = \log(1) = 0$. A positive sentiment is indicated by $\lambda(w) > 0$, and correspondingly $\lambda < 0$ indicates a negative sentiment. By using logarithms, we can reduce the risk of numerical underflow. Realize that our log-likelihood term can be expressed as $\sum_{i=1}^{m} \log \frac{\Pr(w_i|\text{Pos})}{\Pr(w_i|\text{Neg})} = \sum_{i=1}^{m} \lambda(w_i) \in (-\infty, \infty)$; we emphasize that our decision boundary is zero with our log-likelihood formula.

Training Naive Bayes In the context of Naive Bayes, "train" means something different than in logistic regression or deep learning: there's no gradient descent; we're just counting word frequencies in a corpus. There are five steps for training a Naive Bayes model:

1. Collect and annotate corpus (e.g. with positive and negative tweets)
2. Preprocessing (e.g. $\texttt{process\_tweet}(\text{tweet}) \rightsquigarrow [w_1, w_2, \ldots,])$
   - Lowercase
   - Remove punctuation, urls, names, etc.
   - Remove stop words
   - Stemming
   - Tokenize sentences
3. Compute word counts, i.e. $\texttt{freq}(w, \text{class})$ and N_{class}.
4. Apply Laplacian smoothing to compute $\Pr(w|\text{class}) = \frac{\texttt{freq}(w,\text{class})+1}{N_{\text{class}}+|V_{\text{class}}|}$.
5. Calculate $\lambda(w) = \log \frac{\Pr(w|\text{Pos})}{\Pr(w|\text{Neg})}$.
6. Get the log-prior, which involves first counting D_{Pos} = number of positive tweets and D_{Neg} = number of negative tweets, whereby log prior $= \log \frac{D_{\text{Pos}}}{D_{\text{Neg}}}$.[2]

Testing Naive Bayes Once you've trained your model, you test it by taking the conditional probabilities derived and using them to predict the sentiments of new unseen tweets. We can evaluate model performance using test set accuracy. In particular, suppose we are given a tweet "I passed the NLP interview!", and then after preprocessing we end up with [I, pass, the, NLP, interview]. We then look up our $\lambda(w)$'s that were calculated when we "trained" our model and compute the score, i.e. the log-prior plus log-likelihood for the unseen test case and compare it to our threshold (of zero). The values of the words that aren't in our vocabulary are treated as neutral (i.e. zeros) and do not contribute to the final score (likelihood) of the unseen word: $\texttt{pred} = \mathbb{1}_{\texttt{score}>0}$. If we are given a bunch of unseen words, i.e. data set aside during training $(X_{\text{val}}, Y_{\text{val}})$, we (i) compute $\texttt{score} = \texttt{predict}(X_{\text{val}}, \lambda, \texttt{log-prior})$ and then (ii) predict $\texttt{pred} = \mathbb{1}_{\texttt{score}>0}$, and then (iii) compute test accuracy given by $\frac{1}{m}\sum_{i=1}^{m}(\texttt{pred}_i == Y_{\text{val}_i})$.

Applications of Naive Bayes There's more we can do than just sentiment analysis. For example, we could do author identification: if you had two large corpora each written by different authors, you could train the model to recognize whether a document was written by one author or the other. Or, if you had some works by Shakespeare and some works by Hemmingway, you could calculate the λ for each word to predict how likely a new word is to be used by Shakespeare or alternatively Hemmingway. Another common use is spam filtering: $\frac{\Pr(\text{spam}|\text{email})}{\Pr(\text{non-spam}|\text{email})}$. One of the earliest applications of Naive Bayes was to filter between relevant and irrelevant documents in a database. I.e. given a set of keywords in a query, in this case, you can calculate the likelihood of the documents given the query:

$$\Pr(\text{document}_k|\text{query}) \propto \prod_{i=0}^{|\text{query}|} \Pr(\text{query}_i|\text{document}_k).$$

Then, we have a decision rule that suggests retrieval if $\Pr(\text{document}_k|\text{query}) >$ threshold, and then *sort* the documents based on their likelihoods; perhaps we choose to keep the first m results or ones with a

[2] If the dataset is balanced, the log-prior is zero since $D_{\text{Pos}} = D_{\text{Neg}}$ and $\log(1) = 0$.

likelihood above a certain threshold. Lastly, we can also use Naive Bayes for word disambiguation, i.e. breaking words down for contextual clarity. Consider that you have two possible interpretations of a given word within a text: let's say you don't know if the word "bank" in a text is referring to the bank of a river or a financial institution. To disambiguate your word, calculate the score of the document: $\frac{\Pr(\text{river}|\text{text})}{\Pr(\text{money}|\text{text})}$.

2.2.1 Assumptions of Naive Bayes

Naive Bayes is a very simple model: it doesn't involve setting any parameters. The method is called "naive" because of the assumptions it makes about the data. The first assumption is independence between predictors within each class, and the second has to do class (im)-balance with your validation sets. Let's explore each in detail and how they can affect our results.

Independence To illustrate what independence between features looks like, lets consider the following example:

```
It is sunny and hot in the Sahara Desert.
```

Naive Bayes assumes the words in a piece of text are independent of one another, but as you can see this is not always the case: the words "sunny" and "hot" often appear together as they do in this example. When taken together, they might also be related to the thing they're describing like a beach or a desert; so the words in a sentence aren't really independent of one another. But, Naive Bayes assumes that they are. The implication is that we could end up under *or* over estimating the conditional probabilities of individual words. E.g. if your task was to complete the sentence: "It's always cold and snowy in {blank}", then Naive Bayes might assign equal probability to the words spring, summer, fall, and winter even though from the context winter is the most likely candidate.[3]

Distribution of training data A good data set will contain the same proportion of (e.g. positive and negative) classes as a random sample would. However, most available annotated corpora are artificially balanced. E.g. in a real tweet stream a positive tweet is more likely to be sent than a negative tweet. Part of this has to do with platform decisions to perhaps ban content that is inappropriate or contains offensive vocabulary. Assuming that reality behaves as your training corpus could result in a very optimistic *or* pessimistic model.

2.2.2 Error Analysis

No matter what NLP method you use, you'll one day find yourself faced with an error, e.g. a missclassified sentence. How can we analyze such errors? Let's consider some possible errors in the model prediction that can be caused by:

- Removing punctuation and stop words – semantic meaning can be lost in the preprocessing step.
- Word order – can affect the meaning of a sentence.
- Adversarial attacks – language quirks can confuse Naive Bayes classifiers.

Removing punctuation Let's consider an example tweet: "My beloved grandmother :(". The sad face punctuation in this case is *very* important to the sentiment of the tweet because it tells you what's happening; but, if we remove punctuation then the processed tweet will leave behind a different (positive) sentiment. After processing, we may end up with [belov, grandmoth] which appear positive in nature.

[3] More sophisticated methods can deal with this issue.

Removing (stop) words It's not just about punctuation either, consider as an example "this is not good, because your attitude is not even close to being nice". If we remove stop words, we're left with [good, attitude, close, nice]. From this set of words, any classifier would infer that the sentiment is positive. There are techniques we will learn about later to handle "nots" and word-order. For now, the takeaway is to look at the processed data to make sure your model can get an accurate read.

Word order The input pipeline isn't the only source of trouble. E.g. consider the following two tweets:

```
I am happy because I did not go.
I am not happy because I did go.
```

The first is purely positive, the latter is negative. In this case, the "not" is important to the sentiment but gets missed by the Naive Bayes classifier: word order can be as important as spelling.

Adversarial attacks Lastly, let's discuss adversarial attacks which essentially describe a language phenomenon like sarcasm, irony, and euphemism. Humans pick these up quickly but machines are terrible at it. The tweet, "This is a ridiculously powerful movie. The plot was gripping and I cried right through until the ending" contains a somewhat positive movie review, but pre-processing might suggest otherwise. I.e. if we pre-process, you'll get a list of mostly negative words, but these words were in fact used to describe a movie that the author enjoyed. Applying Naive Bayes to this list of words would yield a negative score, unfortunately.