Contents

1 Sentiment Analysis with Logistic Regression

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.

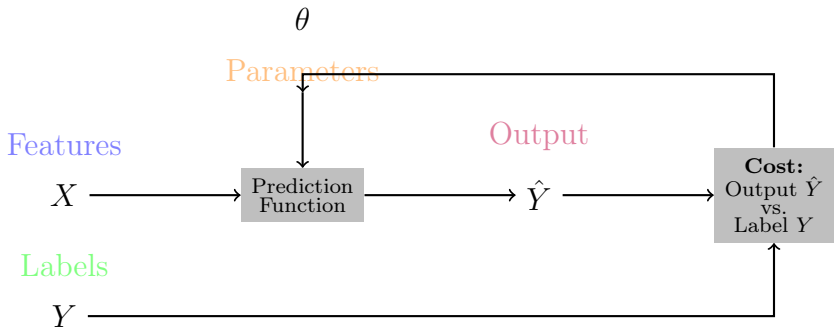


Figure 1: Overview of supervised machine learning.

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:

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

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

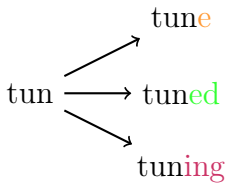
```
@YMourri and @AndrewYNg are tuning a GREAT AI Model at https://deeplearning.ai!!!
```

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



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 $\overset{\text{Preprocessing}}{\longrightarrow}$ `[happy, learn, nlp]` $\overset{\text{Feature Extraction}}{\longrightarrow}$ $[1, 4, 2]$

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

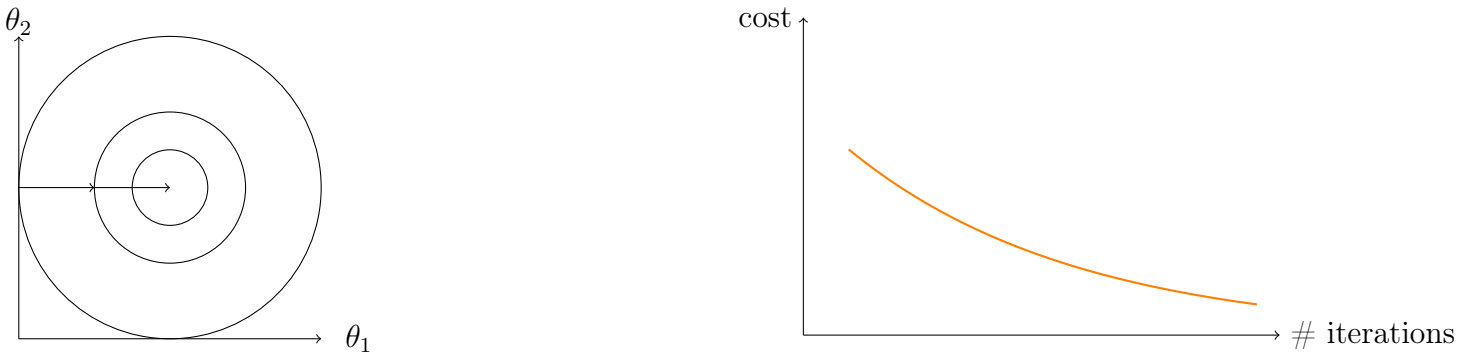
Previously, we've learned how to extract features, which we will now use to predict whether a tweet has a positive or negative sentiment. Logistic regression makes use of a sigmoid (or standard logistic) function which outputs a probability between zero and one. What's the recap from supervised machine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

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

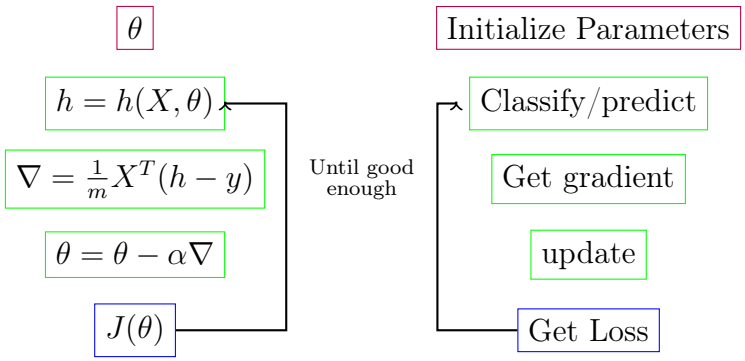
where i denotes the observation number. Note that as $\theta^T x^{(i)}$ gets closer and closer to $-\infty$, the denominator of the sigmoid expression blows up and as a result the output values gets closer to zero. Conversely, as the inner product $\theta^T x^{(i)}$ gets closer to ∞, the denominator 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:

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

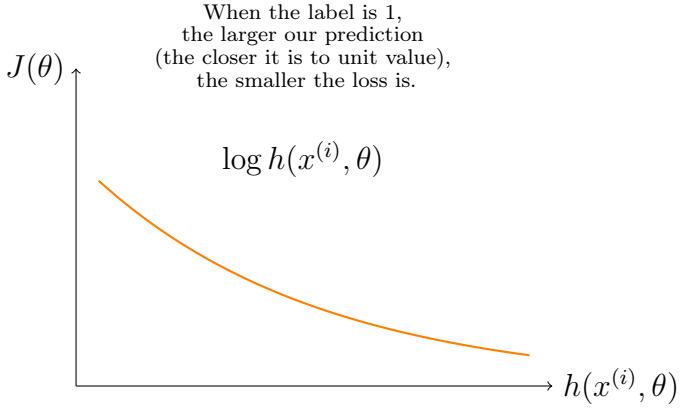
We can then rescale by $\frac{1}{m}$ to get *average* cost. Recall we are maximizing over $h(x^{(i)},\theta)$ in the equation above, and maximizing an equation is the same as minimizing its negative. Therefore,

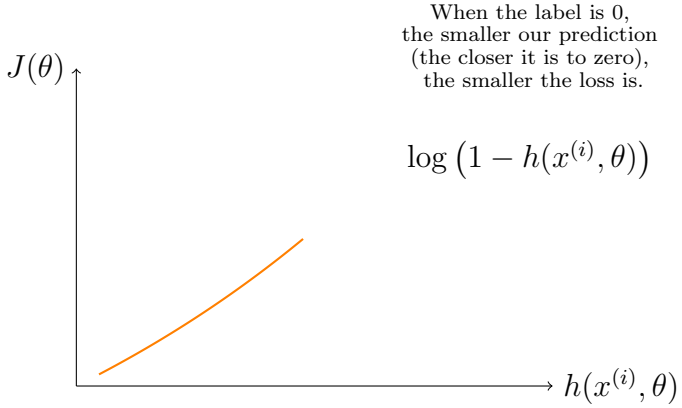
$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log h(x^{(i)},\theta) + (1 - y^{(i)}) \log \left(1 - h(x^{(i)},\theta)\right) \right].$$

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.





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:

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

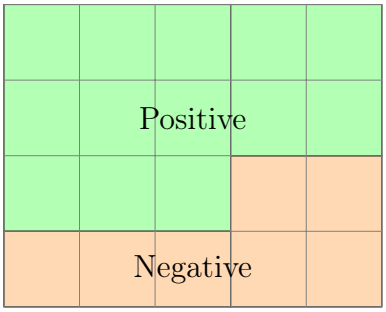
The vectorized version is simply given by

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

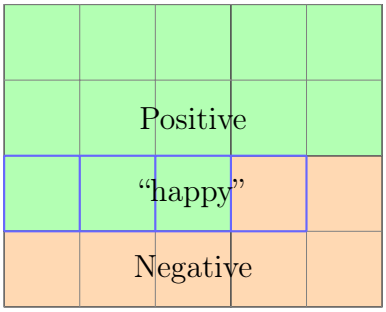
2 Sentiment Analysis with Naive Bayes

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”

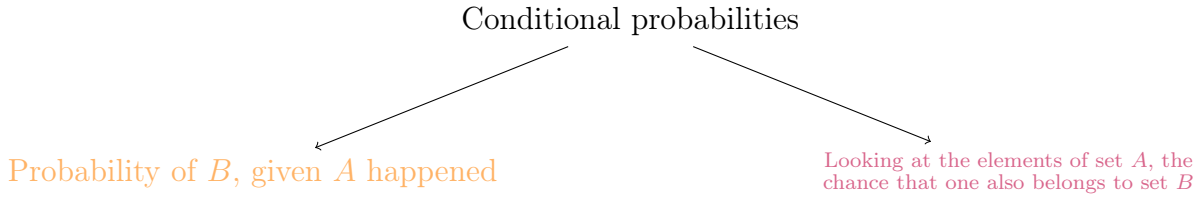
Within this corpus, observe that the word “happy” is sometimes associated with a positive sentiment, but also sometimes with a negative sentiment! How can this happen? Let’s explore the situation using probabilities.

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:



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. `process_tweet`(tweet) $\rightsquigarrow [w_1, w_2, \dots,]$)
   - 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 $\log \text{ prior } = \log \frac{D_{\text{Pos}}}{D_{\text{Neg}}}$.[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 $\Pr(\text{document}_k|\text{query}) > \text{threshold}$, 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.

3 Vector Space Models

3.1 Motivating Vector Space Models

Let's talk about vector spaces, and what types of information vectors can encode. We'll talk about vector space models, their advantages, and some applications.

Suppose you have two questions:

```
Where are you heading?
Where are you from?
```

The sentences are identical but for the last word. However, the meaning is entirely different. On the other hand, the two questions:

```
What is your age?
How old are you?
```

contain no overlapping words but have identical meaning. Vector space models can identify similarity even when sentences don't share common words; they can also be used for answering paraphrasing, and summarization. One advantage of vector space models is they allow us to capture dependencies between words. E.g.

```
You eat cereal from a bowl.
You buy something and someone else sells it.
```

In the first sentence, the words *cereal* and *bowl* are related (we're eating one thing out of the other). In the second sentence, the latter half of it depends on the first half. Vector space models can capture this and many other types of relationships among different sets of words. When using a vector space models, the way that representations are made are by identifying the *context* around each word in the text, and this captures the relative meaning. "You shall know a word by the company it keeps" (John Firth).

3.2 Word by Word and Word by Doc

Let's discuss how to construct vectors based off of a co-occurrence matrix. Depending on the task you're trying to solve, you can have several possible designs.

Word by word To get a vector space model using a word by word design, you'll make a co-occurrence matrix and extract vector representations for the words in your corpus. We'll also discuss how to find relationships between words and vetors, also known as their similarity. We define the **co-occurrence** of two different words as the number of times they occur together within a certain distance k. E.g. suppose our corpus has the following two sentences:

```
I like simple data.
I prefer simple raw data.
```

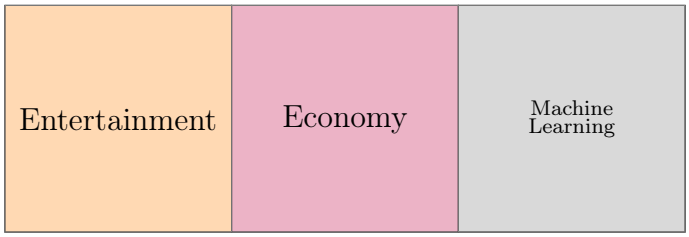
Then, the row of the co-occurrence matrix corresponding to the word `data` with a value $k = 2$ would be populated with the following values:

	simple	raw	like	I
data	2	1	1	0

Table 1: The word "data" appears within distance one of the word "simple" in the first sentence, and within distance two of the word simple in the second example. Both of these are within our tolerance of $k = 2$, so the word by word co-occurrence entry for "data" and "simple" in this example is the value 2. We can proceed to compare the word "data" with other words appearing in our vocabulary, populating a representation that is based on word by word co-occurrence counts within a certain window size.

With a word by word design, you get a representation with $|V|$ entries, where V is our vocabulary.

Word by document With a word by document design, you'll count the number of times that words from your vocabulary appear in documents that belong to specific categories.



Here, we'd count the number of times that our words appear in the document that belong to each of the three categories. In this example, suppose that the word data appears 500 times in documents from your corpus related to entertainment, 6,620 times in economy documents, and 9,320 in documents related to machine learning. The word film might appear in each document's category 7,000, 4,000, and 1,000 times respectively.

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

Once you've constructed representations for multiple sets of documents or words, you'll get your vector space. For example, we could take a representation for the *words* "data" and "film" by looking at the rows

of the table (or matrix). However, we can also get a representation for each *category* of document type by looking at the columns! In our toy example, the vector space will have two dimensions: the number of times that the words "data" and "film" appear on the type of document.

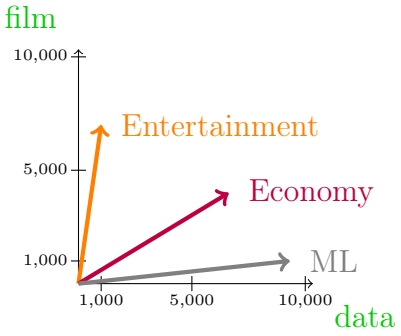


Figure 2: In this vector representation, the document categories Economy and Machine Learning are more similar to each other than they are to entertainment. Measures of similarity include angle (cosine similarity) and distance (Euclidean, for example).

3.3 Distance and Similarity Metrics

Euclidean distance Euclidean distance is a similarity metric: it identifies how far apart two points (or vectors) are from each other.

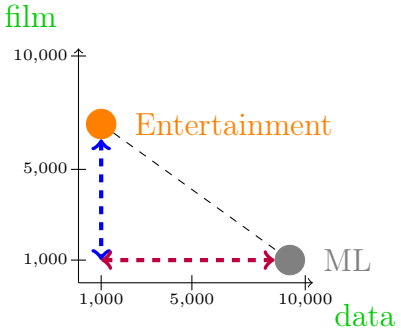


Figure 3: Here we visualize the distance between two points, i.e. the length of the line segment connecting them. In two dimensions, this is given by $d(B, A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$; the first term is the horizontal distance squared, and the second term is the vertical distance squared.

	data	boba	ice-cream
AI	6	0	1
drinks	0	4	6
food	0	6	8

Table 2: Suppose boba is word $\vec{w}$ and ice-cream word $\vec{v}$, and we wish to calculate the Euclidean distance between these two word-vectors. This is given by $d(\vec{v}, \vec{w}) = \sqrt{(1-0)^2 + (6-4)^2 + (8-6)^2} = \sqrt{1+4+4} = \sqrt{9} = 3$.

In higher dimensions, i.e. an n dimensional vector, the process is similar.

$$d(\vec{v}, \vec{w}) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2} = \|\vec{v} - \vec{w}\|_2.$$

Cosine similarity Another type of similarity function is cosine similarity. Euclidean distance can be problematic: in particular it can be biased by the size difference between the representations; to see why, consider the following example.

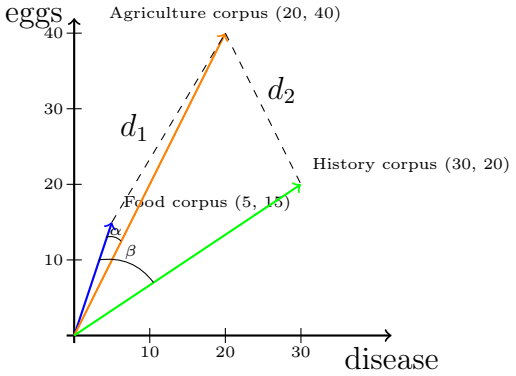


Figure 4: Realize that $d_2 < d_1$ due to the fact that, as measured by Euclidean distance, they are more similar in *magnitude*. In this case, the Agriculture and History corpora are much larger than the Food corpus; this causes their frequency counts along the dimensions of "disease" and "eggs" to be quite high, and their Euclidean distance is resultingly smaller than the distance between Food and Agriculture, which isn't intuitive. What about considering the cosine of the angle between the vectors as a measure of similarity? In this case $\alpha < \beta$, and it happens to be that cosine similarity is a better metric than Euclidean distance under this representation.

An alternative measure to Euclidean distance is Cosine similarity: i.e. the cosine of the inner angle between two vectors. If the angle is small, the cosine would be close to unit value, whereas if the angle is close to 90 degrees, the cosine approaches zero. Recall that a vector norm is defined as $\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$ and an inner-product is defined by $\langle u, v \rangle = u \cdot v = \sum_{i=1}^{n} u_i \cdot v_i$. Then, we can define cosine similarity as

$$\texttt{Cosine-similarity}(u, v) = \frac{\langle u, v \rangle}{\|u\|\|v\|}$$

If the input arguments (vectors) are perpendicular, the cosine of their inner angle is zero (since $\cos(90°) = 0$), whereas if they point in the same direction their cosine angle is unit valued (since $\cos(0°) = 1$, even if they have differing magnitudes).

Manipulating words in vector spaces How can we manipulate vectors using vector arithmetic to make predictions? I.e. we will use manipulate vector representations to infer unknown relations among words. Suppose we have a vector space with countries and their capital cities. Perhaps we know that the capital of the U.S. is Washington D.C., but that we don't know the capital of Russia, and we further we'd like to use our knowledge of Washington D.C. with the U.S. to figure it out: it's as easy as applying linear algebra.

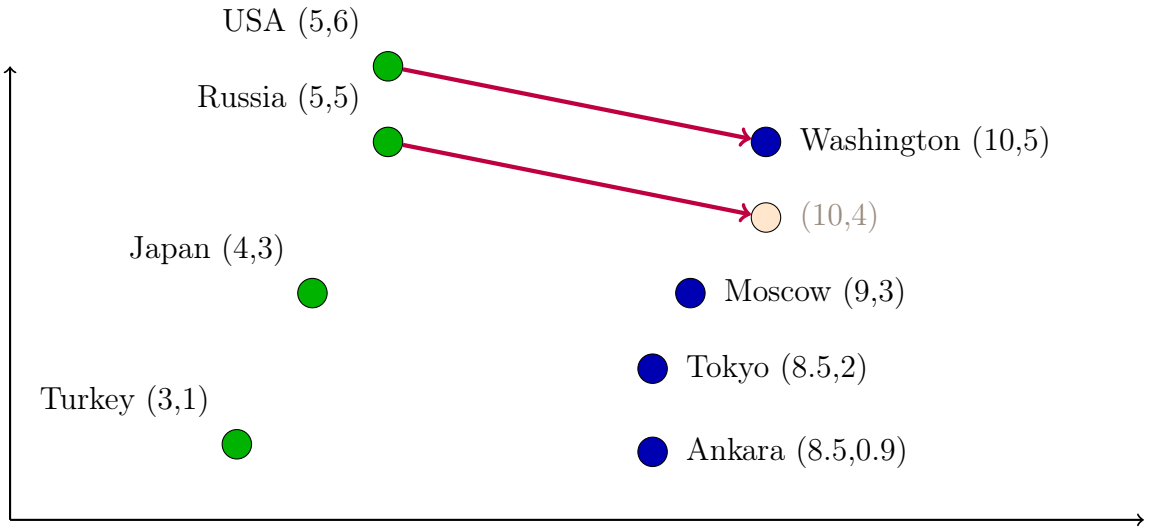
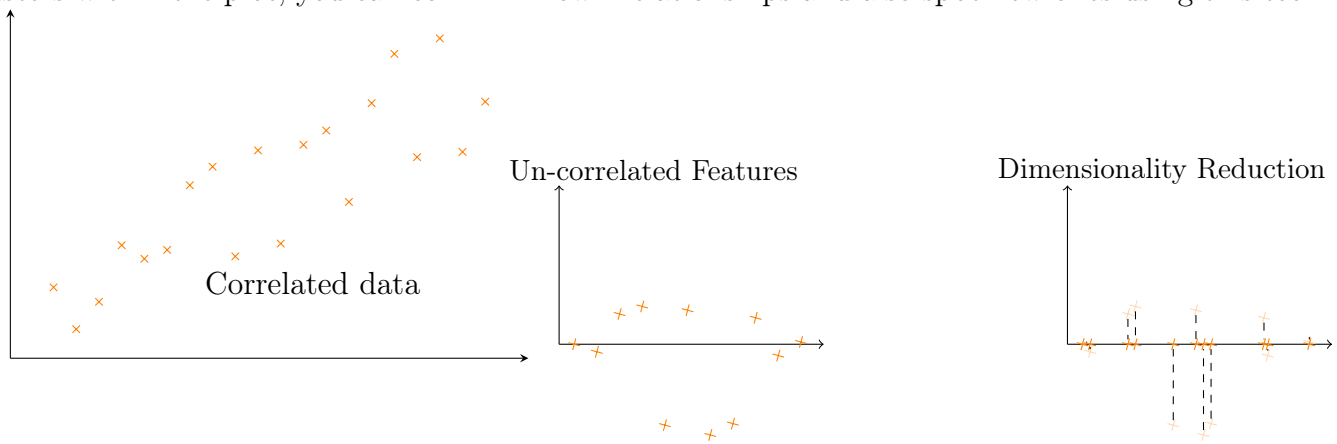


Figure 5: From Mikolov et. al. "Distributed Representations of Words and Phrases and their Compositionality": in this representation, the vector between capitals Washington and USA is given by $[\texttt{Washington} - \texttt{USA}] = [5, -1]$. The easiest way to predict the capital of Russia is to simply take $\texttt{Russia} + [5, -1] = [10, 4]$. Because there are no cities with that representation, we'll search for the capital city that is most similar by comparing each vector with the Euclidean distances (or Cosine similarities). The catch is that we need a vector space where the representations capture the relative meaning of words.

3.4 Principal Components Analysis

Visualization and PCA Often, we'll have vector representations in very high dimension, but we'll want to find a way to reduce the dimensions of the vectors to two dimensions so that we may plot it on a 2-D cartesian plane. What's the intuition? Principal Component Analysis is they key: it allows us to take a representation with $d > 2$ dimensions and project it down to a lower dimension, e.g. $d = 2$, which we can then plot. If you perform PCA on your data, you might find that your word representation captures relationship between pairs of words that you expect to be related, in so far as these related words appear in clusters within the plot; you can confirm known relationships and also spot new ones using this technique.



Using PCA, we first find a set of uncorrelated features. Then, we can project our data into a lower dimensional space, trying to retain as much information as possible.

PCA algorithm How do we get uncorrelated features? Realize that the eigenvectors of the covariance matrix of our data correspond to the uncorrelated features for our data, and the eigenvalue corresponds to

the amount of information retained by each feature. Note that it's essential that our eigenvectors are sorted according to their eigenvalues from largest to smallest (in magnitude): this means that if we use the first k eigenvectors to get a representation we are greedily selecting vectors that retain the most information from our data.

1. Mean-normalize data: $x_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}}$.
2. Obtain covariance matrix Σ.
3. Perform Singular Value Decomposition on Σ to get three matrices, where the first stores the eigenvectors U (columnwise), the second S stores the eigenvalues along the diagonal.
4. Dot produt to Project your data: $X' = XU[:, 0:k]$, where k is the number of dimensions we wish to reduce to.
5. Calculate Percentage of Retained Variance: $\frac{\sum_{i=0}^{k-1} S_{ii}}{\sum_{j=0}^{d-1} S_{jj}}$.

Eigenvectors give the direction of uncorrelated features, whereas the eigenvalues are the variance of the new features. The dot product gives the projection on uncorrelated features.

4 Machine Translation

4.1 Transforming Word Vectors

Word vectors can capture important properties of words. Let's make use of them to learn to align words into different languages, which will give us a basic translation program; locality sensitive hashing will speed it up.

Overview of translation In order to translate an English word to a French word, one naive way could be to generate an extensive list of English words and their associated French word. If you ask a human to do this, you would find someone who knows both languages to start making a list. If, however, you want a machine to do this, you need to first calculate word embeddings associated with English and word embeddings associated with French. Next, retrieve the English word embedding of a particular English word such as "cats", then find some way to transform the English word embedding into word embedding that has a meaning in the French word vector space; we'll see how to convert between vector spaces in a moment. The next step is to search for word vectors in the French word vector space that are most similar to the transformed vector representation: the most similar words are candidate words for our translation.

Transforming vectors using matrices Define a randomly selected matrix R and see how well it works to transform our English vector space representation X into our French vector space representation Y, i.e. compute $XR \approx Y$. In order for this to work, you'll need to first get a subset of English words and their French equivalence, get the respective word vectors, and stack the word vectors in their respective matrices X and Y. They key is to keep the rows lined up (or to align the row vectors): i.e. if the first row of the English matrix X contains the word representation for "cat", then so should the French matrix Y contain the word representation for the translated word. It's natural at this point to ask: if we already have the mappings from $X \rightsquigarrow Y$ (i.e. the translation pairs), then why not just store them in an associative data-structure like a dictionary...why train a model at all? The nice property of our transformation matrix R is that it will *generalize* to translate unseen example inputs. So, we only need to train on a subset of the English-French vocabulary, and *not* the entire vocabulary.

Finding a good transformation matrix We define our loss function as

$$\texttt{Loss} = \|XR - Y\|_F.$$

where $\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}$. If we start with a random matrix R, we can gradually improve upon it in an iterative fashion. We first compute the gradient by taking the gradient of the loss function with respect to matrix R, i.e. $g = \frac{\partial}{\partial R}\texttt{Loss}$, then update using gradient descent, i.e. $R \leftarrow R - \alpha g$. We can either pick a fixed number of times to go through the loop, or exit the loop when the loss falls below a certain threshold. In practice, it's easier to minimize the the square of the norm instead. What about the gradient of the loss function, which is defined as the derivative of the loss with respect to the matrix R. With our loss:

$$g = \frac{\partial}{\partial R}\texttt{Loss} = \frac{2}{m}\left(X^T(XR - Y)\right),$$

where m is the number of rows in our training matrix.

4.2 Approximate k-nearest neighbors and Locality Sensitive Hashing

Finding k-nearest neighbors of a vector In the context of Machine Translation, notice that the transformed word vector after the transformation of its embedding through a matrix R would be in the French vector space, but! it's not going to be identically equal to any of the word vectors in the French vector space. We need to search through the French word vectors to find a French word that is similar to the one we created from the transformation.

To solve this problem, let's look at a related problem: how do you find your friends who are living nearby? One way to do this is to go through your address book and for each friend, get their addres, and calculate how far away they are from our hometown of San Francisco. We then sort our friends according to their distance from San Francisco. Notice that if we have a lot of friends, this is a time intensive process. A more efficient way is to observe that, if it were somehow possible to *filter* our friends list to only those who live in the same continent, for example, then we could cut down our search space drastically. When you think about organizing subsets of a dataset efficiently, you might think about placing your data into buckets. If you think about buckets, then you'll definitely want to think about hash-tables.

Hash tables Let's say we have several data items and we want to group them into buckets by some kind of similarity. One bucket can hold more than one item and each item is always assigned to the same bucket.

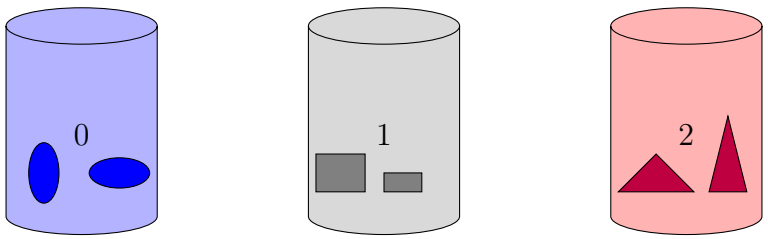


Figure 6: Here's one pictorial example of hash buckets. One bucket might end up storing blue ovals, another might store gray rectangles, and a third may store magenta triangles.

How can we do this with word vectors? Let's first assume that word vectors have 1-dimension instead of several hundred. So, each word is represented by a *single* scalar value. We need to figure out a mapping from values to hash buckets, i.e.

$$\texttt{HashFunction}(\text{vector}) \longrightarrow \text{Hash Value}.$$

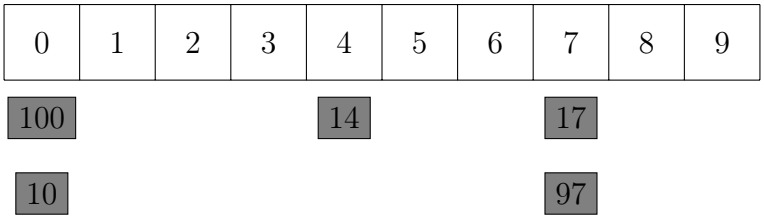


Figure 7: One example could be, in our context of a 1-D embedding, to take the value modulo ten and assign it to one of ten buckets $\{0, 1, \ldots, 9\}$.

A function that assigns a hash value is called a hash function.
To build this example out in code, we simply define

```
def basic_hash_table(value_l, n_buckets):
  def hash_function(value_l, n_buckets):
    return int(value) % 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
```

Notice that in our `hash_function()` definition, we are using a *dictionary*-comprehension, where we are associating integer keys with empty lists as value; this initializes our hash-table. Now, realize that our original goal was to place similar word vectors into the *same* bucket. In the example above, it doesn't look like numbers that are close to each other are in the same bucket, e.g. 10, 14, and 17 are all in different buckets. Idealy, we want a hash function that places similar word vectors in the same bucket; this brings us to locality sensitive hashing. Locality is another word for location, and sensitive is another word for caring. So locality sensitive hashing is a hashing method that cares very deeply about assigning items based on where they're located in vector space.

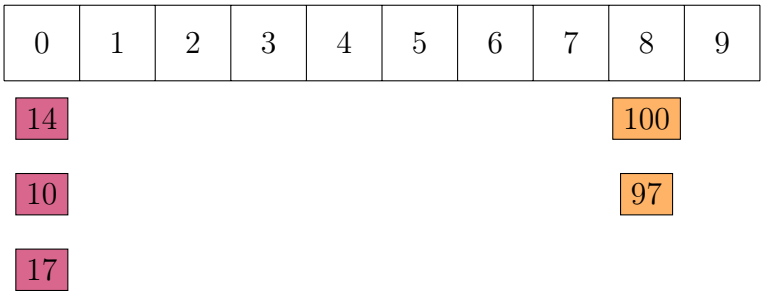


Figure 8: A good hash-function places similar word vectors in the same bucket.

Locality sensitive hashing A key method for reducing the computational cost of finding k-nearest neighbors in high-dimensional spaces is locality-sensitive hashing. Let's start with examples of word vectors with two dimensions.

Visualizing a dot product Let's visualize the dot product further.

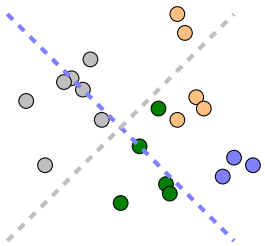


Figure 9: Let's say you want to find a way to know that these blue dots are somehow close to each other, and that these gray dots are also related to each other. First, divide the space using dashed lines, called *planes*. Notice that the blue plane slices up the space into vectors that are above it or below it, where the blue bectors all happens to be on the same side of the blue plane. Similarly, the gray vectors all lie above the gray plane. It looks like the planes can help us bucket the vectors into subsets based on their location, which is exactly what we want: a hashing function that is sensitive to the location of the items that it's assigning into buckets.

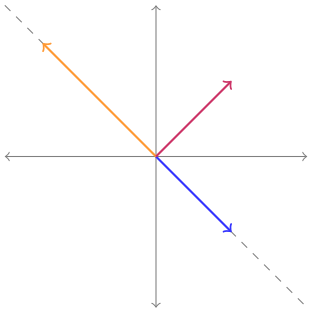


Figure 10: A plane can be visualized in 2-D space by a dashed gray line. It actually represents all possible vectors that would be sitting on that plane, i.e. the orange and blue vectors that are parallel to the plane. You can define a plane with a single vector. The magenta vector is perpendicular to the plane, and it's called the normal vector to that plane: it is perpendicular to any vectors that lie on the plane.

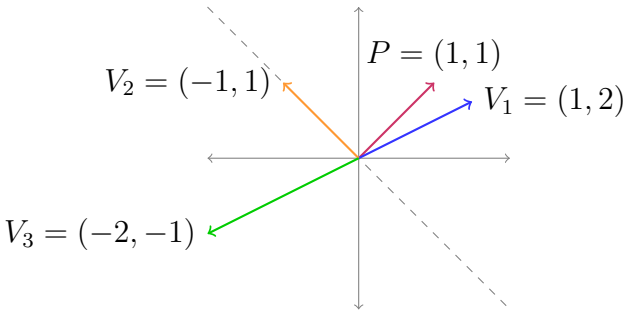


Figure 11: Consider three sample vectors in blue, orange, and green. The normal vector to the plane is labeled P. Let's focus on vector V_1: observe that the inner product $\langle P, V_1 \rangle = 3$. Similarly, $\langle P, V_2 \rangle = 0$, and $\langle P, V_3 \rangle = -3$. Realize that when the inner product is positive with the normal vector, the vector is on one side of the plane, and if the inner product is negative it's on the opposite side of the plane. If the dot product is zero, the vector is on the plane.

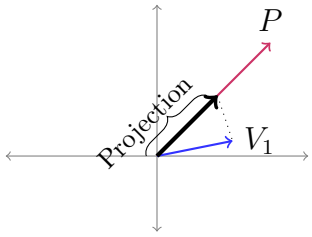


Figure 12: To visualize the dot product, imagine one of the vectors such as P, as if it's the surface of the Earth. Gravity pulls all objects *straight* down towards the surface of the Earth. Next, pretend you're standing at the end of the vector V_1. You tie a string to a rock and let gravity pull the rock to the surface of vector P: the string is prependicular to vector P. Now, if you draw a vector that's in the same direction of P but ends up at the rock, you'll have what's called the *projection* of vector V_1 onto vector P. The magnitude or length of that vector is equal to the dot product of V_1 and P, i.e. $\|PV_1^T\|$. The sign of the dot product indicates the direction with respect to the purple normal vector: a positve sign indicates a vector is pointed in the same direction, whereas a negative sign means they are pointed in opposite directions.

We can implement a simple Python function to determine whether a vector v is pointed in the same or oppositive direction of a normal vector P:

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

We first compute the inner product, then examine the sign, then convert the sign to a one, zero, or minus one depending on whether the inner product is positive, zero, or negative.

Multiple planes How can we combine information from multiple planes into a single hash value? Previously, we saw how we can examine the sign of the dot product between the normal vector of a plane and a vector representing our data, and we could get a notion of position relative to the plane. Here, we're going to examine how to use information from multiple planes in order to get a single hash value for your data in your vector space, since in order to divide your vector space into manageable regions, you'll want to use more than one plane.

- For each plane, find out whether a vector is on the positive or negative side of that plane: you'll get multiple signals, one for each plane and we need to find a way to combine them all into a single hash value. I.e. compute $PV_i^T \rightsquigarrow \text{sign}_i \rightsquigarrow h_i$ where $h_i \in \{0, 1\}$ according to $h_i = \mathbb{1}_{\text{sign}_i \geq 0}$, indicating the side of the plane that the vector lies on.
- Compute $\texttt{hash} = \sum_{j=1} h_j 2^{j-1}$.

```
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
```

Approximate k-nearest neighbors Let's devise an algorithm that computes k-nearest neighbors much faster than brute search. We've seen how a few planes can help to divide a vector space into regions. But how do you know if a set of planes is the *optimal* way to divide the vector space? You can't, so why not create multiple sets of random planes to divide up the vector space into multiple, independent sets of hash tables? This is akin to creating multiple copies of the universe, or a multiverse. You can make all of these different sets of random planes in order to help find a good set of k-nearest neighbors, as follows:

- Create multiple sets of planes, in parallel.
- For each *set* of planes, determine the neighbors i.e. the words that are in the same hash-bucket as the word in question.

  (Note that each set of planes will yield a different (possibly overlapping) set of neighbors)

By using multiple sets of random planes for locality-sensitive hashing, you have a more robust way of searching the vector space for a set of vectors that are possible candidates to be nearest neighbors. This technique is known as *approximate* nearest neighbors because you're not searching the entire vector space, but just a subset of it. So the output is not the absolute k-nearest neighbors, but it's approximately the kk-nearest neighbors. We sacrifice precision in order to gain efficiency in our search.

How can we do this in code?

```
num_dimensions = 2
num_planes = 3
random_planes_matrix = np.random.normal(size = (num_planes, num_dimensions))
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_plan_matrix(random_planes_matrix, v)
```

Searching documents How can we use approximate k-nearest neighbors to search for pieces of text related to a query in a large collection of documents? We simply create vectors for both and find the nearest neighbors. In order to get ready to perform document search, we need to think through how to represent documents as vectors, instead of just words as vectors. Let's say you have a document composed of three words: "I love learning". How can we represent the entire document as a vector? We could find the word vectors for each individual word, and then simply add them together; in this case the resulting document vector has the same dimension as the word vector representation. Let's code this up:

```
word_embedding = {'I': np.array([1, 1, 1]), 'love': np.array([-1, 0, 1]), 'learni
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)
```

Reference: "Speech and Language Processing" by Jurafsky, Martin, Norvig, and Russell.

5 Autocorrect and Minimum Edit Distance

5.1 Autocorrect

What is autocorrect exactly? It can mean slightly different things depending on the context. A key concept in performing autocorrect is quantifying how far apart two strings are, and calculating the minimum number of characters needed to be changed to go from one string to another.

What is autocorrect? It changes misspelled into the correct ones. E.g.

`Happy birthday deah friend!` $\rightsquigarrow$ `Happy birthday dear frend!`

But what if we typed "deer" instead of "deah" or "dear": the word might be spelled correctly, but its context is incorrect. This is a more sophisticated problem that we will revisit later; for now, let's focus on *misspelled* words.

1. Identify a misspelled word.
2. Find strings n edit distance away: a string that is 1-edit distance away might be more likely to be a correct replacement than a string that is 2-edit distance away.
3. Filter candidates - only retain real words that are spelled correctly.
4. Calculate word probabilities - this informs us how likely each word is to appear in the context; we choose the most likely candidate as our replacement.

Identify a misspelled word How do we know that a word is misspelled? Well, if it's spelled correctly, we could look it up in a dictionary; otherwise, it's probably a misspelled word.

```
if word not in vocab:
  misspelled = True
```

If a word is not in a dictionary, we flag it for correction. Recall that we've limited our focus to just spelling errors, not contextual errors. There are more sophisticated techniques for identifying words that are probably incorrect by looking at neighboring words, but for now, identifying a word as incorrect by its apparent misspelling will yield a powerful model that works well.[4]

Find strings n-edit distance away An edit is a type of operation performed on a string to change it into another string. Edit distance counts the number of these operations, so that the n-edit distance metric tells us how far away one string is from another.

- Consider an *insert* operation, which adds a letter to a string at any position. E.g. if we take "to" and insert a "p" at the end we get "top", or if we insert a "w" in the middle we get "two".
- We can also apply *delete* operations, which remove a letter from a string. E.g. we can start with "hat" and delete single characters to yield "ha", "at", "ht".
- There's also a *swap* operation, which allows us to swap two adjacent letters. E.g. we can start with "eta" and swap to get "eat" or "tea". This does *not* include switching two non-adjacent letters, i.e. a swap does not let us create "ate" by switching the "e" and the non-adjacent "a" in "eta".
- We can also *replace* one letter with another. E.g. "jaw" can be made into "jar" or "paw".

By combining these edits, we can finda list of all possible strings that are n-edits away from a given string. For auto-correct, n is usually 1-3 edits.

[4] We conceded this means a word like "deer" in our introductory example would not be flagged by this autocorrect program.

Filter candidates Notice how many of the strings generated don't look like actual words: to filter strings and keep real words we'll want to only consider correctly spelled words from our candidate list. We can again use a known dictionary to make comparisons against a known vocabulary.

Calculate probabilities The final step is to calculate word probabilities and find the most likely word from the candidates. Consider an example sentence, "I am happy because I am learning".

Word	Count
I	2
am	2
happy	1
because	1
learning	1

Table 3: To calculate the probability of a word in a sentence, we need to calculate the word frequencies, and in addition the total number of words in the body of texts or corpus. The total number of (non-unique) words in the body of text or corpus is 7. Normally, a corpus would be much larger, e.g. all of the Harry Potter books. But for this pedagogical example, we consider our entire corpus to be this single sentence.

For any word, we define the probability of its appearance $P(w)$ as

$$P(w) = \frac{C(w)}{|V|},$$

where $C(w)$ is the number of times the word appears, and V is the total size of the corpus. For autocorrect, we simply find the word candidate with the highest probability and choose that word as the replacement.

5.2 Minimum edit distance

Let's consider a slightly different problem: you're given two strings (possibly entire documents), and you want to evaluate how similar they are. Given one string, the minimum edit distance is the lowest number of operations needed to transform one string into the other.[5] For calculating minimum edit distance, we'll use three edit operations: insert (add a letter), delete (remove a letter), and replace (change 1 letter to another).

Up until this point, we've treated all edit operations as having the same (unit-valued) cost. But now we'll consider different costs for different types of operations:

Edit Cost:	
Insert	1
Delete	1
Replace	2

Table 4: The above costs are intuitive if we think about a replace as a delete followed by an insert.

But what about much longer strings and large corpora of texts or even DNA strings? You can try and solve these problems by brute force: adding one added distance at a time and enumerating all posibilities until one string changes to another; recognize that this requires exponential work with respect to the input string length. A much faster way is to use dynamic programming.

[5]Applications in NLP include spelling correction, document similarity, and machine translation.

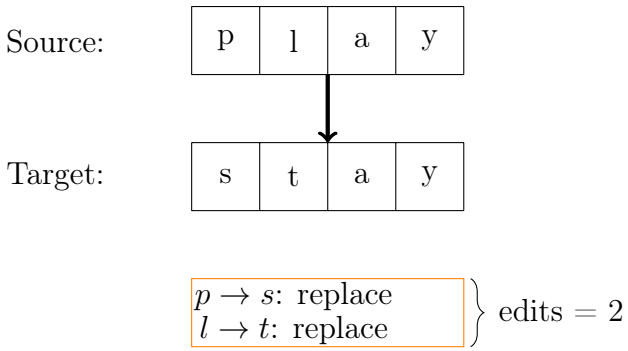


Figure 13: In order to turn "play" into "stay", what is the minimum number of edits required? To turn "p" into "s", we need a replacement $p \rightsquigarrow s$, and to turn "l" into "t" we similarly need a replacement $l \rightsquigarrow t$. Both remaining characters are the same so we do nothing, and the total number of edits is now two.

5.2.1 Minimum edit distance algorithm

We'll start by laying out the problem as follows.

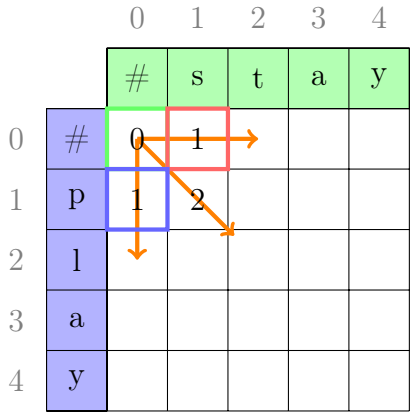


Figure 14: The # represents an empty string, and we've also set up row and column indices. If our table above is called D, we want the entries $D[i, j] = \text{source}[:i] \rightsquigarrow \text{target}[:j]$.

We've defined $D_{i,j}$ to represent the minimum edit distance between source[: i] and target[: j]. If we have a source of length m and a target of length n then $D_{m,n}$ represents the minimum edit distance to transform one entire word to another. We'll compute this from the shortest substring to the full string. The intuition is that we can build upon each sub-problem by combining solutions, e.g. finding the minimum edit distance between two letters is easy. Then, we increase the problem size one letter at a time, building on what you already know.

- The first step is to transform the source empty string into the target empty string: this requires zero operations and so the edit distance is simply zero.
- Now, we can move on to transforming our source "p" into an empty string, which we can do with a single delete: edit distance is unit value.
- We can also move on to transforming our source empty string to a target "s", which we can do with a single insertion: edit distance is unit value.
- Now, let's look at how to go from source "p" to target "s": there is more than one possible way to make this transformation. Each possible sequence of edits is known as a *path*:

- starting with "p", you can insert "s" on the end to get "ps", then delete "p" from the beginning to get "s". This has a cost of one insert and one delete; notice that we've already calculated the cost of inserting a letter "s" as its given by the table entry above (in red) that we've already filled in. So, we actually just calculate the cost of deleting the "p" and add it onto the cost that you've stored in the red box above, which is $1 + 1 = 2$.
- starting with "p", we can delete "p" to get hashtag (empty string) and then insert "s" to get "s". Notice again that we've already calculated the cost of deleting "p" (in blue) to the left: the blue box is the cost of going from "p" to hashtag (empty string) by deleting "p", so we can calculate the cost of inserting "s" and add it to the cost that you've stored in the blue box on the left. This is $1 + 1 = 2$.
- The third way is to replace "p" with "s" with a replacement, which has cost 2. We can think of this as going from the green cell to the target cell.

- Now, we take the minimum of all three of these paths which yields 2 in this case.

Realize that in order to fill in a cell, we need to know the values of the cells above, to the left, and adjacent upper-left. In doing so, we can benefit from calculations already performed. We can We can generalize this to a formula such that we can fill in the entire table; but first, we must fill in the first row and first column in order to get the sufficient inputs to calculate the remaining entries.

Filling in first column: this corresponds to transforming "play" into the empty string. We can use the following reccurrence relation:

$$D_{i,j} = D_{i-1,j} + \texttt{del\_cost}.$$

I.e. we simply look at the cell above and add an extra delete edit. I.e. to make the string "p" into the empty string, we need one delete operation. To make "pl" into the empty string, we need to delete "p" and "l", which are two delete operations, and so on. Now, at $D_{4,0}$ we have the minimum edit distance for "play" to be transformed into the empty string, which is of course simply given by four deletions with a cost of four.

Filling in first row: this corresponds to transforming the empty string into "stay". We can use the following, slightly different reccurrence relation:

$$D_{i,j} = D_{i,j-1} + \texttt{ins\_cost}.$$

I.e. we simply look at the cell to the left and add an extra insertion edit.

Generalized dynamic programming formula for minimum edit distance There is also a general formula for filling in each entry of our table

$$D_{i,j} = \min \begin{cases} D_{i-1,j} + \texttt{del\_cost} \\ D_{i,j-1} + \texttt{ins\_cost} \\ D_{i-1,j-1} + \mathbb{1}_{\texttt{src[i]} \neq \texttt{tar[j]}} \times \texttt{rep\_cost} \end{cases}$$

The first argument to the `min` operator corresponds to "coming from the cell above", whereas the second argument corresponds to "coming from the cell to the left", and finally if you come from the cell to the upper-left then you do one of two things: either (i) add the replacement cost if the two corresponding letters between source and target don't match, or (ii) add nothing if they do (because there is no edit to be done for letters that are already the same).[6]

[6]It can be interesting to color the table according to the magnitude of the edit distances in the cells. This can reveal some interesting patterns. In our example table, fours will lie along the diagonal beyond "the middle" of the table, because this corresponds to where the prefixes have been made the same and no more edits are to be made.

Levenshtein distance and backtrace We've been using Levenshtein distance metric. Finding the minimum edit distance doesn't always solve the whole problem. Sometimes, we need to know how we got there as well: we can do this by storing a backtrace which is simply a pointer in each cell letting you know where you came from to get there. So, you know the path taken across the table from the top-left corner to the bottom right corner (i.e. how we transformed our string); this tells us the edits that were made and is particularly useful in problems dealing with string alignment. We lastly mention that our algorithm laid out in the last section is an example of dynamic programming: we solve the smallest subproblem first and then reuse that result to solve the next bigger subproblem, saving that result, and continuing on. This is a well known technique in computer science.