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.

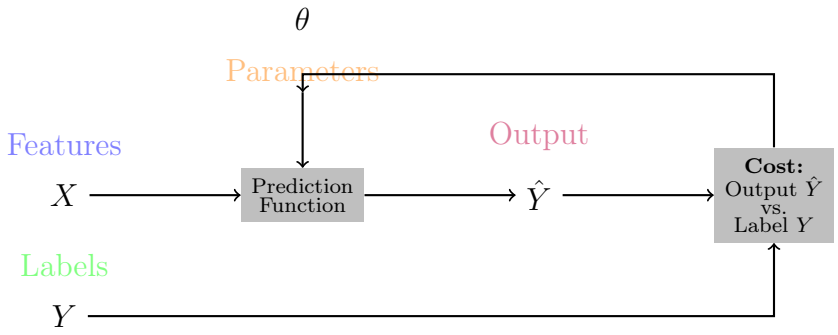


Figure 1: Overview of supervised machine learning.

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

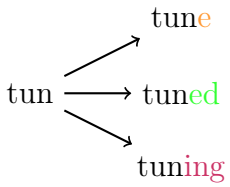
```
@YMourri and @AndrewYNg are tuning a GREAT AI Model at https://deeplearning.ai!!!
```

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

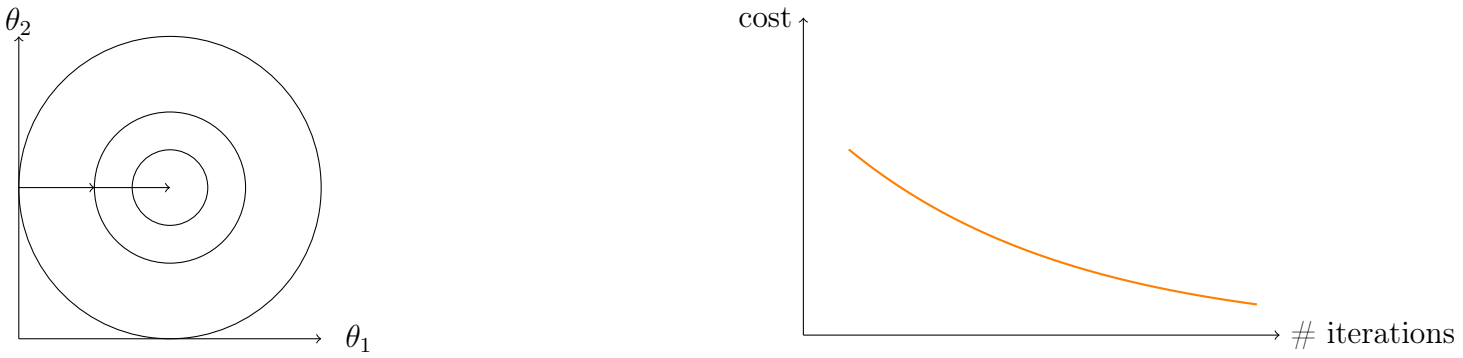
Previously, we've learned how to extract features, which we will now use to predict whether a tweet has a positive or negative sentiment. Logistic regression makes use of a sigmoid (or standard logistic) function which outputs a probability between zero and one. What's the recap from supervised machine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

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

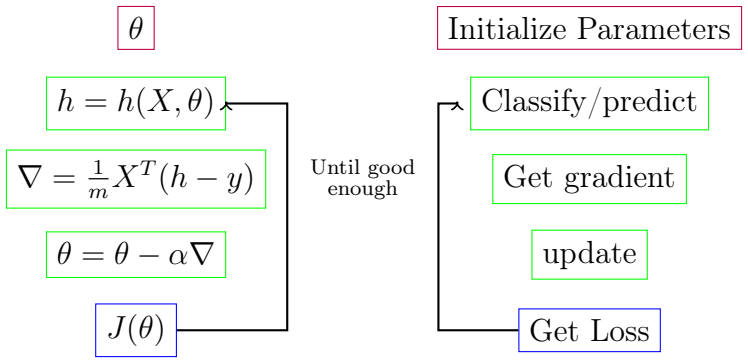
where i denotes the observation number. Note that as $\theta^T x^{(i)}$ gets closer and closer to $-\infty$, the denominator of the sigmoid expression blows up and as a result the output values gets closer to zero. Conversely, as the inner product $\theta^T x^{(i)}$ gets closer to ∞, the denominator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

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

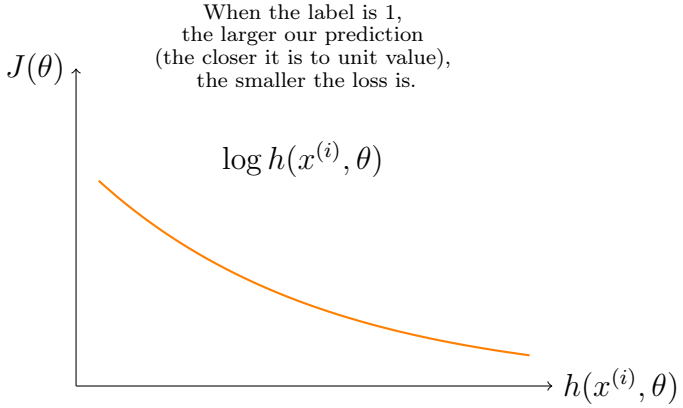
We can then rescale by $\frac{1}{m}$ to get *average* cost. Recall we are maximizing over $h(x^{(i)},\theta)$ in the equation above, and maximizing an equation is the same as minimizing its negative. Therefore,

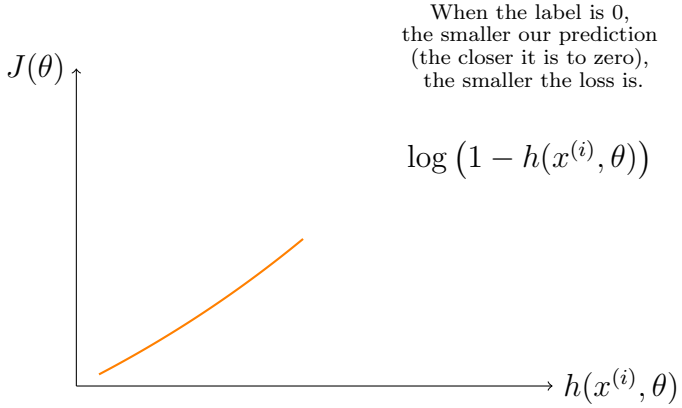
$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log h(x^{(i)},\theta) + (1 - y^{(i)}) \log \left(1 - h(x^{(i)},\theta)\right) \right].$$

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:

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

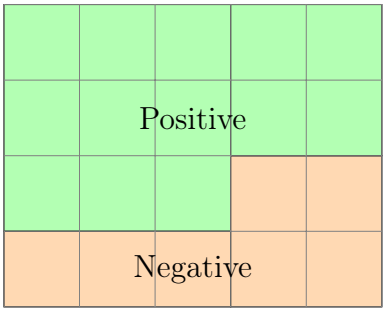
The vectorized version is simply given by

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

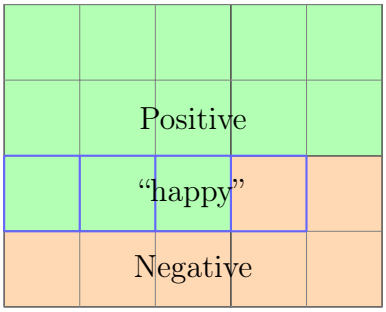
2 Sentiment Analysis with Naive Bayes

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"

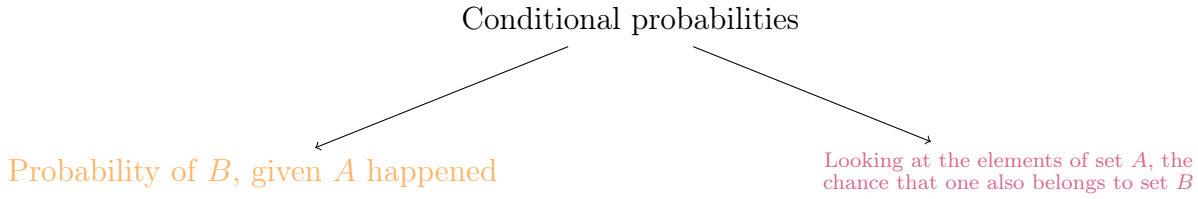
Within this corpus, observe that the word "happy" is sometimes associated with a positive sentiment, but also sometimes with a negative sentiment! How can this happen? Let's explore the situation using probabilities.

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.
```

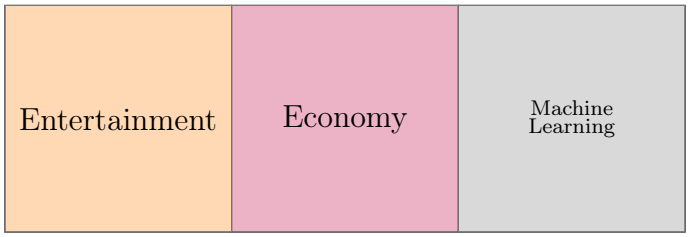
Then, the row of the co-occurrence matrix corresponding to the word `data` with a value $k = 2$ would be populated with the following values:

	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.

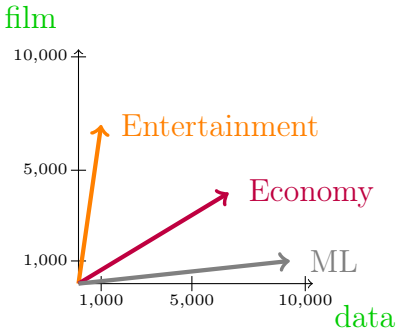


Figure 2: In this vector representation, the document categories Economy and Machine Learning are more similar to each other than they are to entertainment. Measures of similarity include angle (cosine similarity) and distance (Euclidean, for example).

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.

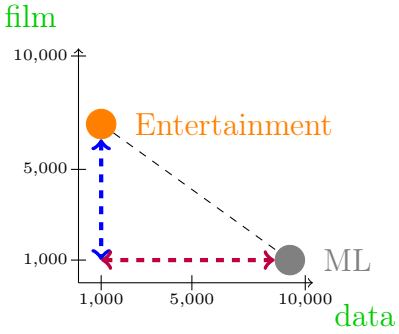


Figure 3: Here we visualize the distance between two points, i.e. the length of the line segment connecting them. In two dimensions, this is given by $d(B, A) = \sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$; the first term is the horizontal distance squared, and the second term is the vertical distance squared.

	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.

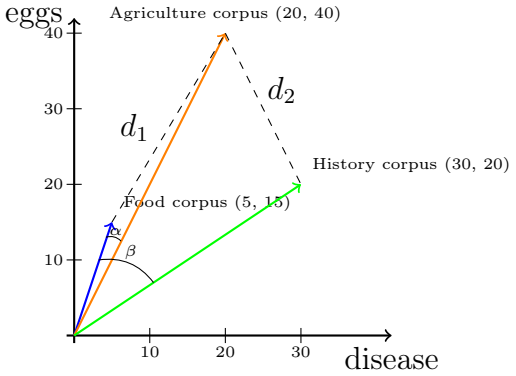


Figure 4: Realize that $d_2 < d_1$ due to the fact that, as measured by Euclidean distance, they are more similar in *magnitude*. In this case, the Agriculture and History corpora are much larger than the Food corpus; this causes their frequency counts along the dimensions of "disease" and "eggs" to 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.

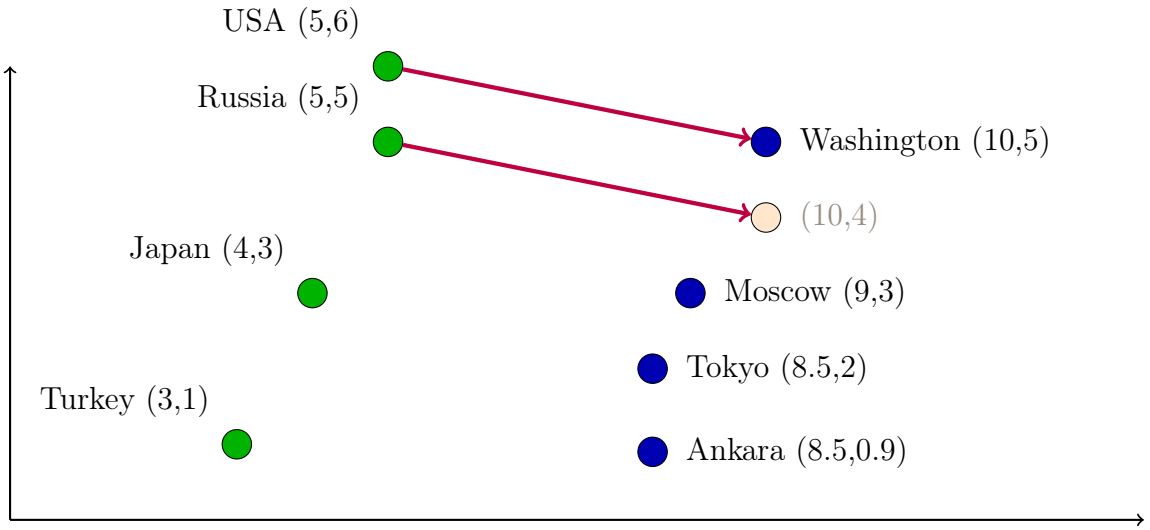
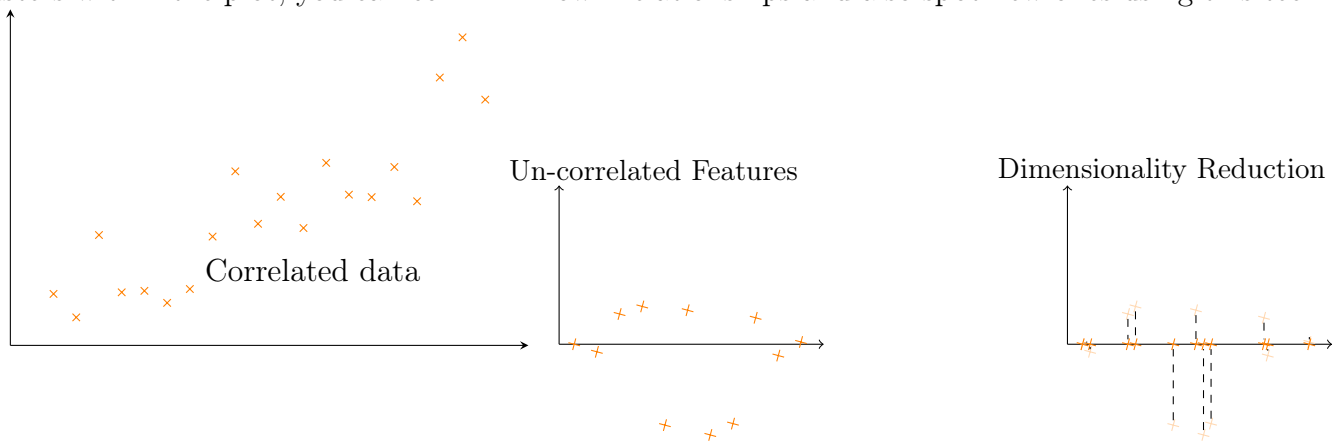


Figure 5: From Mikolov et. al. "Distributed Representations of Words and Phrases and their Compositionality": in this representation, the vector between capitals Washington and USA is given by $[\texttt{Washington} - \texttt{USA}] = [5, -1]$. The easiest way to predict the capital of Russia is to simply 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.

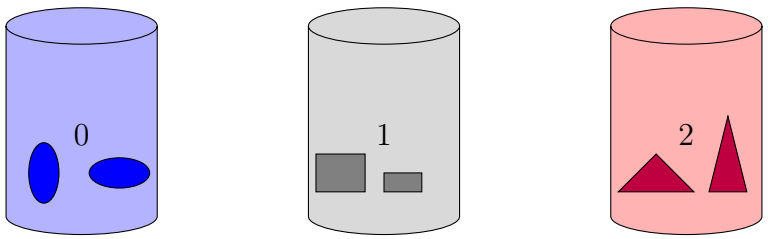


Figure 6: Here's one pictorial example of hash buckets. One bucket might end up storing blue ovals, another might store gray rectangles, and a third may store magenta triangles.

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.

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

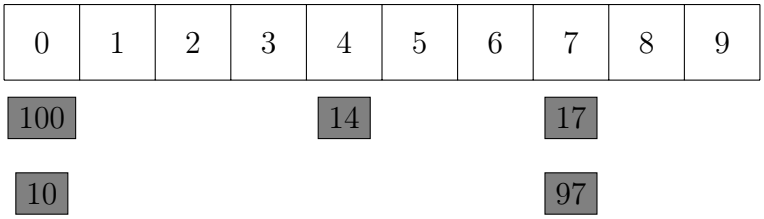


Figure 7: One example could be, in our context of a 1-D embedding, to take the value modulo ten and assign it to one of ten buckets $\{0, 1, \ldots, 9\}$.

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.

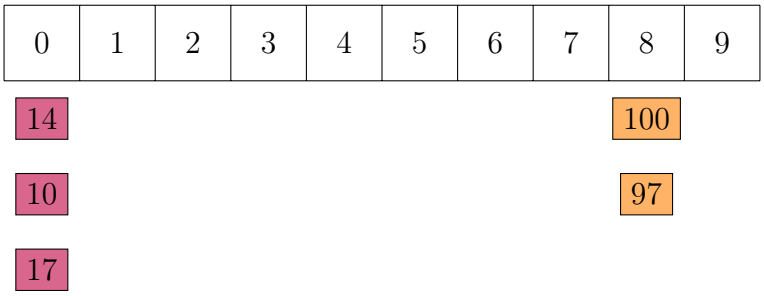


Figure 8: A good hash-function places similar word vectors in the same bucket.

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.

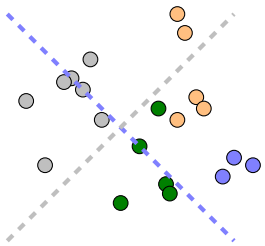


Figure 9: Let's say you want to find a way to know that these blue dots are somehow close to each other, and that these gray dots are also related to each other. First, divide the space using dashed lines, called *planes*. Notice that the blue plane slices up the space into vectors that are above it 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.

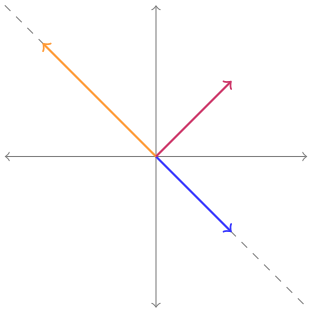


Figure 10: A plane can be visualized in 2-D space by a dashed gray line. It actually represents all possible vectors that would be sitting on that plane, i.e. the orange and blue vectors that are parallel to the plane. You can define a plane with a single vector. The magenta vector is perpendicular to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

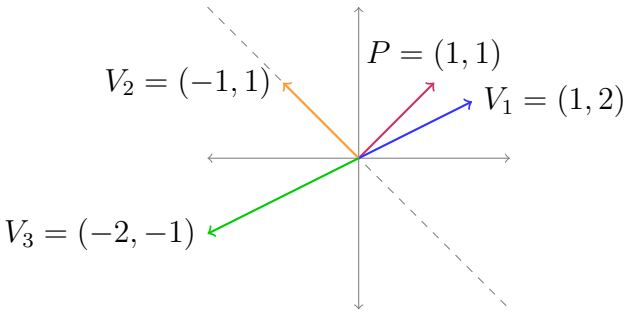


Figure 11: Consider three sample vectors in blue, orange, and green. The normal vector to the plane is labeled P. Let's focus on vector V_1: observe that the inner product $\langle P, V_1 \rangle = 3$. Similarly, $\langle P, V_2 \rangle = 0$, and $\langle P, V_3 \rangle = -3$. Realize that when the inner 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.

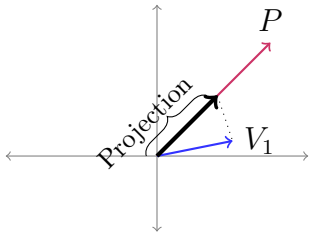


Figure 12: To visualize the dot product, imagine one of the vectors such as P, as if it's the surface of the Earth. Gravity pulls all objects *straight* down towards the surface of the Earth. Next, pretend you're standing at the end of the vector V_1. You tie a string to a rock and let gravity pull the 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.

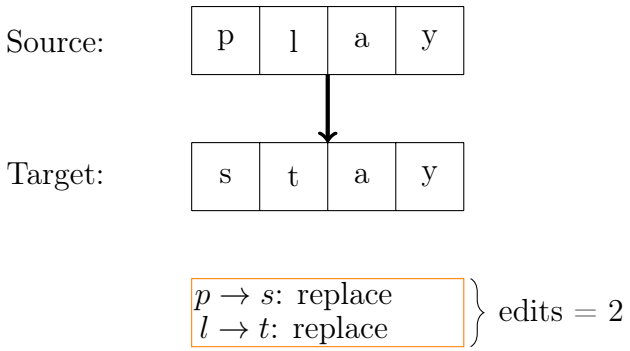


Figure 13: In order to turn "play" into "stay", what is the minimum number of edits required? To turn "p" into "s", we need a replacement $p \rightsquigarrow s$, and to turn "l" into "t" we similarly need a replacement $l \rightsquigarrow t$. Both remaining characters are the same so we do nothing, and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

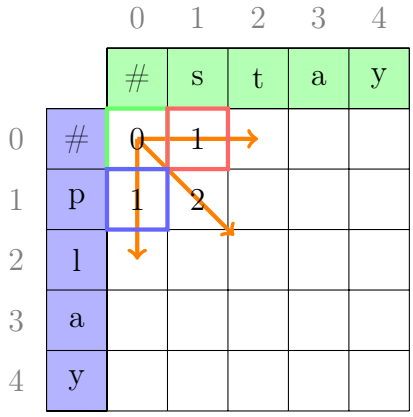


Figure 14: The # represents an empty string, and we've also set up row and column indices. If our table above is called D, we want the entries $D[i, j] = \text{source}[:i] \rightsquigarrow \text{target}[:j]$.

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.

6 Part of Speech Tagging

6.1 What is part of speech tagging?

Part of speech tagging refers to the category of words or the lexical terms in the language. E.g. in English, lexical terms include `noun, verb, adjective, adverb, pronoun, preposition`, and many others.

Why	not	learn	something	?
adverb	adverb	verb	noun	punctuation mark

We're going to use a short representation called *tags* to represent these categories. The process of assigning these tags to the words of a sentence is referred to as parts of speech tagging.

lexical term	**tag**	**example**
noun	NN	something, nothing
verb	VB	learn, study
determiner	DT	the, a
w-adverb	WRB	why, where
...		

Because parts of speech tags describe the characteristic structure of lexical terms in a sentence or text, we can use them to make assumptions about semantics. We can use them to identify named entities, e.g.

```
The Eiffel Tower is located in Paris.
```

The terms "Eiffel Tower" and "Paris" are both named entities. Tags can also be used for co-reference resolution, i.e. if you have two sentences: "the Eiffel Tower is located in Paris. **It** is 324 meters high." then we can use parts of speech tagging to infer that "it" refers in this context to the Eiffel tower. A last example is in speech recognition: we can use parts of speech tags to check if a sequence of words has a high probability or not.

6.2 Markov Chains

Let's start with a small example with what we want to accomplish, and then let's see how Markov Chains can help accomplish our task. Suppose we are given a *sentence prefix*, "why not learn...", and we want to predict whether the following word is more likely to be a noun rather than another verb. If you examine the word "learn", note that it's a verb. Being familiar with the English language, we might guess that if you see a verb in a sentence, then the following word is more likely to be a noun rather than another verb. The core idea here is that the likelihood of the next word's parts of speech tag in a sentence tends to depend (only) on the parts of speech tag of the previous word. We can represent the likelihoods visually:

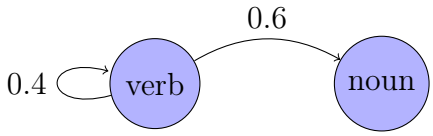


Figure 15: The nodes indicate states, and the arrows indicate transitions where the corresponding floating points values describe transition probabilities.

Markov chains defined A Markov Chain is a type of stochastic model that describes a sequence of possible events, with the property that to get the probability for each event, we need only the state of the previous event! The word stochastic means random. A stochastic model incorporates and models processes that have a random component to them. Markov chains can always be depicted graphically as a directed graph with nodes and edges. I.e. a Markov chain is a graph of states and transitions between these states. The key property of a Markov chain is the Markov property: the probability of the next event only depends on the current events.

Intuition for Markov property The Markov property is what allows us to keep our model simple by saying that all you need to determine the next state is the current state information; we don't need any other information from any of the previous states. Going back to an analogy: suppose we consider whether water is in a solid, liquid, or gaseous state. If we look at a cup of water that is sitting outside, the current state of the water is a liquid state. When modeling the probability that the water in the cup will transition into the gas state, we don't need to know the previous history of the water: whether it previously came from ice cubes or from rain clouds is irrelevant!

Applying Markov chains to parts of speech tagging Let's think about a sentence as a sequence of words with associated parts of speech tags. We can represent that sequence with a graph where the parts of speech tags are events that can occur depicted by the states of our model graph; the edges of the graph will have weights or transition probabilities associated with them which define the probability of going from one state to another.

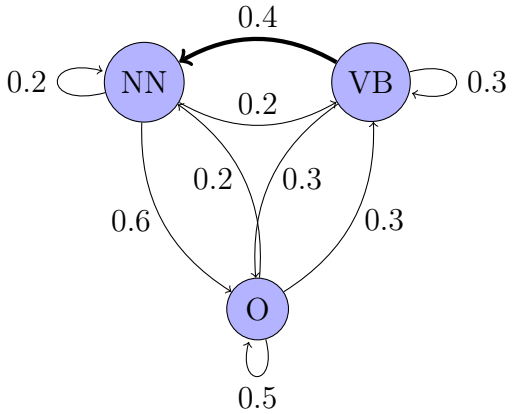


Figure 16: In this simple model, we have a state NN for nouns, VB for verbs, and all other states are captured by O. For the example sentence, "Why not learn ...", it happens that "learn" is a verb, categorized by state `VB` in our Markov chain depicted above. The probability that the next word is a noun is given by 0.4, denoted by the arrow (transition) from `VB` to `NN` (noun).

Transition probability matrix An equivalent way to represent a Markov chain is through a transition probability matrix P, or through what amounts to effectively a simple table. If there are n states in a model, then a transition matrix will be of dimension $n \times n$. Each *row* in the matrix represents transition probabilities from one state to all other states.

	NN	VB	O
NN (noun)	0.2	0.2	0.6
VB (verb)	0.4	0.3	0.3
O (other)	0.2	0.3	0.5

Table 5: Take a look at the first row: it denotes the transition probabilities *from* state `NN` i.e. when the current word is a noun. The columns in that row correspond to the future next states, and the entries in the table correspond to the likelihoods of transitioning to the next state given the current state. Notice that the row-sums are unit-valued sicne they are probability distributions.

Notice in the table above that it's of dimension $n \times n$. But this means we require a preceding word (and corresponding part of speech tag) in order to predict the next word. Unfortunately, with this set-up we cannot predict the first word in a document, since there is no preceding word! What we can do then is introduce an initial state π and include these probabilities in the table in the form of an $(n+1) \times n$ table.

	NN	VB	O
π (initial)	0.4	0.1	0.5
NN (noun)	0.2	0.2	0.6
VB (verb)	0.4	0.3	0.3
O (other)	0.2	0.3	0.5

Table 6: We encode an initial state such that we can predict the first parts of speech in a document.

Markov chain recap A Markov chain consists of a set of n states $Q = \begin{bmatrix} q_1 & q_2 & \dots & q_n \end{bmatrix}$, and we encode the transition probabilities in a transition matrix A

$$A = \begin{bmatrix} a_{1,1} & \dots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N+1,1} & \dots & a_{N+1,N} \end{bmatrix}$$

Let's now introduce the hidden Markov model, which are used to decode hidden states of a word; in our context that would be the parts of speech of that word.

6.3 Hidden Markov Model

The name implies that states are hidden or not directly observable. Going back to the Markov model that has the states for the parts of speech (such as a noun, verb, or other), we can now think of these states as *hidden* because these are not directly observable from our raw text data. This may seem confusing at first: to a human reading the sentence, we can infer what the parts of speech is, e.g. we can clearly tell that "jump" is a verb. From a machine's perspective, however, it only sees the text "jump" and doesn't know whether it is a verb or a noun. For a machine looking at the text data, it only observes the actual words: these words are observable because they can be seen directly by the machine. The Markov chain model and hidden Markov model have transition probabilities which can be represented by a matrix A of dimension $n + 1 \times n$, where n is the number of *hidden* states, e.g. table 6.2. A hidden Markov model additionally has *emission probabilities*: these describe the transition from the hidden states of our hidden Markov model.

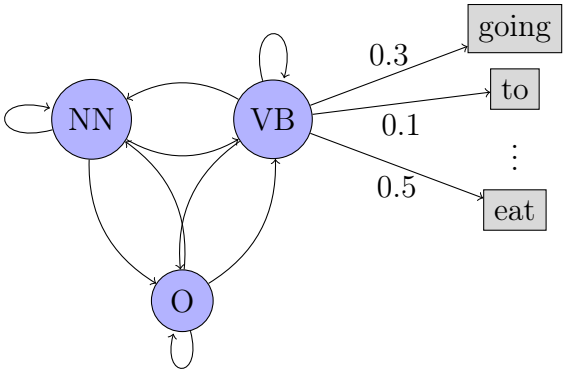


Figure 17: Here, we describe hidden states with a background shade of blue and observable states with gray.The interpretation here is that if the current parts of speech is a verb, then there is a 50% chance that the next (observable) word will be "eat".

We can of course equivalently represent emission probabilities by a matrix, call it B, with dimension $n \times |V|$ where n is the number of states and $|V|$ is the number of observable states.

	going	**to**	**eat**	...
NN noun	0.5	0.1	0.02	
VB verb	0.3	0.1	0.5	
O other	0.3	0.5	0.068	

Table 7: In this table, each row is designated for one of the hidden states. A column is designated for each of the observable states. Each row has the property that it's a probability distribution, i.e. $\sum_{j=1}^{|V|} B_{ij} = 1$. You may also at first be surprised that for a fixed observable state (word), it can have strictly positive probabilities for all hidden states (parts of speech); this is actually quite natural because the parts of speech can depend on the context of usage. E.g. in the sentences "hey lays on his back" and "I'll be back", the word "back" has a noun parts of speech in the former sentence and an adverb for the latter.

Calculating probabilities Let's learn this first conceptually and then mathematically.

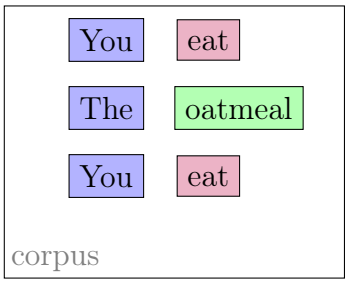


Figure 18: We've depicted a small training corpus; parts of speech tags are denoted by background colors.

To calculate transition probabilities, we'll actually *only* use the parts of speech tags from our training corpus. E.g. to calculate the probabilitiy of a blue speech tag transitioning to a purple one, we first have to count the occurrences of that tag combination in our corpus. In the example depicted above, there are exactly two instances of a blue followed by a purple parts of speech tag. Further, there are exactly three instances of blue parts of speech tags in our entire corpus. So, the conditional probability is $\frac{2}{3}$.

To formally calculate transition probabilities, we do so in several steps:

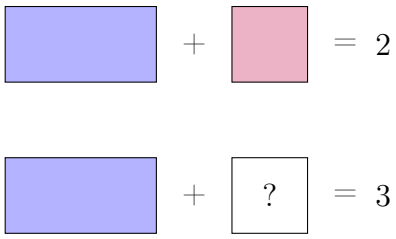


Figure 19: In our elementary example, we're looking for the conditional probability of transitioning from a blue parts of speech tag to a purple one. There are exactly two of these in the training corpus that appears in figure 6.3. There are a total of three instances where a blue parts of speech tag appears followed by something else, for our example corpus. Therefore, the transition probability is $\frac{2}{3}$.

1. Count occurrences of tag pairs in our training corpus: $C(t_{i-1}, t_i)$.
2. Calculate probabilities using the counts

$$\Pr(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{\sum_{j=1}^{n} C(t_{i-1}, t_j)},$$

   where n is the number of tags in our vocabulary.

As an example, suppose you want to train a model for a Haiku, which is a type of Japanese poetry: your training corpus is the following Haiku from Exra Pound, written in 1913:

```
In a station of the Metro
The apparition of these faces in the crowd:
Petals on a wet, black bough.
```

Consider each line of the corpus as a separate sentence. We'll first add a start token to each line or sentence in order to be able to calculate the initial probabilities using the previously defined formula. We then transform all words in the corpus to lowercase, so that the model is case insensitive.

Populating the transition matrix Now that we've processed our text corpus, it's time to populate the transition matrix, which will hold the probabilities of going from one state to another in our Markov model. We fill the first column of our matrix with the counts of the associated tags as in figure 6.3.

After having calculated a counts matrix A, we cal calculate the transition probabilities according to the formulae $\Pr(t_i, t_{i-1}) = \frac{C(t_{i-1}, t_i)}{\sum_{j=1}^{N} C(t_{i-1}, t_i)}$: realize that the entries in the table correspond to the numerator term, and the denominator corresponds to row-sums in our table A above.

	NN	VB	O	
π	1	0	2	3
NN	0	0	6	6
VB	0	0	0	0
O	6	0	8	14

Table 8: In this table, we've simply calculated $C(\cdot, \cdot)$ for each entry given our Haiku by Ezra Pound.

E.g. in our example

$$\Pr(\texttt{NN}|\pi) = \frac{C(\pi, \texttt{NN})}{\sum_{j=1}^{N} C(\pi, t_j)} = \frac{1}{3}$$

$A =$

	NN	VB	O
π	$C(\pi, \texttt{NN})$		
NN	$C(\texttt{NN}, \texttt{NN})$		
VB	$C(\texttt{VB}, \texttt{NN})$		
O	$C(\texttt{O}, \texttt{NN})$		

<s> in a station of the metro

<s> the apparition of these faces in the crowd:

<s> petals on a wet, black bough

Figure 20: Using the formula provided and our example corpus, we see that $C(\pi, \texttt{NN}) = 1$, and similarly for our first column $C(\texttt{NN}, \texttt{NN}) = 0$ $C(\texttt{VB}, \texttt{NN}) = 0$ $C(\texttt{O}, \texttt{NN}) = 6$. We might further realize that there are no parts of speech that are tagged `VB`, so this entire row is identically zero. Also, for the last entry in the table given by count $C(\texttt{O}, \texttt{O}) = 8$, realize that we need to count the comma character as its own word-token; we use a sliding window of length two counting bigrams of `other-other` parts of speech.

Or similarly, $\Pr(\texttt{NN}|\texttt{O}) = \frac{C(\texttt{O},\texttt{NN})}{\sum_{j=1}^{N} C(\texttt{O},t_j)} = \frac{6}{14}$. Problems arise: (i) we have an identically zero row which means its conditional probabilities can't be computed, and (ii) a lot of entries in the *counts* matrix A are zero which mean the corresponding transition probabilities are also zero. This won't work if we want our model to generalie well to other haikus, which may contain verbs! To handle this, we can add some small $\epsilon > 0$ to our formula, a technique known as *smoothing*:

$$\Pr(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i) + \epsilon}{\sum_{j=1}^{n} C(t_{i-1}, t_j) + N * \epsilon}.$$

The consequences of smoothing are (i) we no longer have zeros in our transition probability matrix, and (ii) cases where we had identically zero rows before now have a uniform prior over parts of speech which seems reasonable (e.g. in our example when smoothing is applied $\Pr(\texttt{VB}, \cdot) = \frac{1}{3}$). Note that we may not want to apply smoothing to the first row in our matrix which corresponds to starting from an empty token or start of string: this is because you'll effectively allow strictly positive probabilities for a sentence to start with any word in our vocabulary (including punctuation)!

Populating the emission matrix We can now calculate emission probabilities for our hidden Markov model; they are calculated similarly as our transition probabilities.

$B =$

	in	a	$\cdots$
NN	$C(\texttt{NN}, \texttt{in})$		
VB	$C(\texttt{VB}, \texttt{in})$		
O	$C(\texttt{O}, \texttt{in})$		

<s> in a station of the metro

<s> the apparition of these faces in the crowd:

<s> petals on a wet, black bough

Figure 21: In this matrix, instead of counting *pairs* of tags, we'll count how often a word is tagged with a specific tag such as `noun`, `verb`, or `other`. For example, in our corpus the `noun` tag is associated zero times with word "in", so $C(\texttt{NN}, \texttt{in}) = 0$. The `O` tag is associated with the word "in" twice, so $C(\texttt{O}, \texttt{in}) = 2$.

If we wish to apply smoothing:

$$\Pr(w_i|t_i) = \frac{C(t_i, w_i) + \epsilon}{\sum_{j=1}^{V} C(t_i, w_j) + N * \epsilon} = \frac{C(t_i, w_i) + \epsilon}{C(t_i) + N * \epsilon}$$

where here N is the number of *tags* and V is the size of our vocabulary.

6.4 The Viterbi algorithm

So, we've calculated our transition and emission probabilities for the Markov Chain and hidden Markov model. Given the parts of speech tag in these probabilities, you can easily select the most likely next parts of speech tag (or the most probable word). We do so simply looking up the correct entry in the respective row of the transition (or emission) probability matrix. But what if you are given an entire sentence, such as "why not learn something?" and you want to compute the most likely parts of speech tags given the sentence and your model. The sequence can be computed using the Viterbi algorithm, which is a graph algorithm. Let's consider a toy model and the example sentence, "I love to learn".

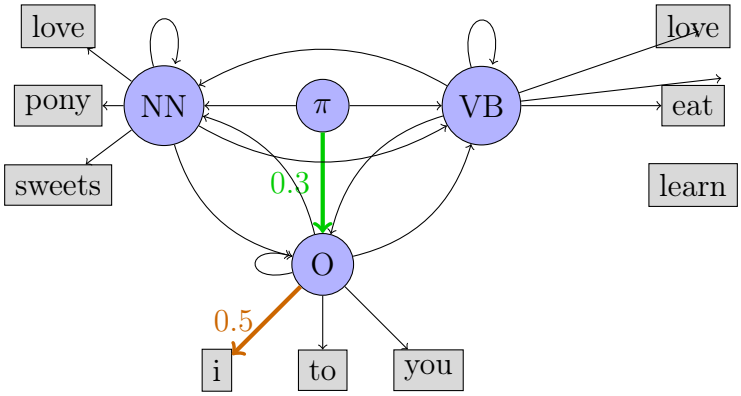


Figure 22: We're given the sequence `<s> I love to learn`, and we want to find the sequence of hidden states (i.e. parts of speech tags) that have the highest probability for this sequence.

- Realize that the word "love" can be emitted by both noun and verb states. We start from the initial states by selecting the next most probable hidden state which here is the `O` state as the word "I" can't be emitted from any other state in this toy-model. This involves the transition probability shown in green and the emission probability in orange: the joint-probability for observing word "I" and with a transition through the `O` state is 0.15, which we can get by multiplying the two transition and emission probabilities together.

- There are then two possibilities or ways to observe the word "love" in our model: either by traversing through the hidden states `NN` or the hidden states `VB`. Let's suppose that the transition probabilities are the same for going from hidden state `O` to either of these other two hidden states (e.g. probability is $\frac{1}{2}$); but, perhaps the emission probability for the word "love" is higher from hidden state `VB`, so we choose that path: if the emission probability is also $\frac{1}{2}$ then the joint probability of observing "love" with hidden state `VB` coming from `O` is $\frac{1}{4}$.

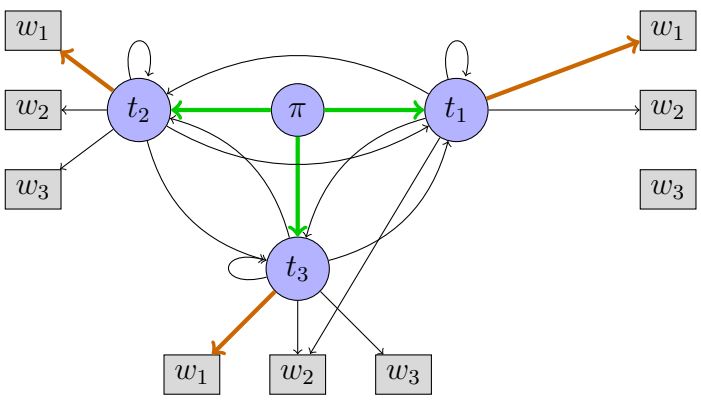
- Next, we transition back to hidden state `O` and emit "to" (we must transition back to this hidden state because it is the only one with the emission "to") with with combined probability $0.2 \times 0.4 = 0.08$, for example.

- Similarly, in the last step we return to the hidden state `VB` again as there is no other hidden state in the model that can emit observed state "learn".

The combined probability of the sentence is the product of individual probabilities that we computed in the steps above. The Viterbi algorithm actually computes several such paths at the same time in order to find the most likely sequence of hidden states. It uses the matrix representation of the hidden Markov model. The algorithm can be split into three main steps: the initialization step, the forward pass, and the backward pass. Given transition and emission probability matrices, we first populate and use auxiliary matrices C and D where C holds intermediate optimal probabilities and D holds indices of visited states. As you're traversing the model graph to find the most likely sequence of parts of speech tags for the given sequence of words $w_1, \ldots, w_k$. These two matrices C and D are both $n \times k$ where n is the number of parts of speech tags and k is the number of words in the given sequence.

6.4.1 Initialization

Let's see how to initialize a matrix that will tell you the parts of speech tag of every word. This matrix will tell you the probability that each word belongs to a certain parts of speech. In the initialization step, the first column of each of our matrices C and D are populated. The first column of C represents the probability of the transitions from the start states in the graph to the first tag t_i and word w_1.



$$C = \begin{array}{c|ccccc} & w_1 & w_2 & \cdots & w_k \\ t_1 & c_{1,1} & & & \\ \vdots & & & & \\ t_N & c_{N,1} & & & \end{array}$$

$$c_{i,1} = \pi_i \times b_{i,\texttt{cindex}(w_1)} = a_{1,i} \times b_{i,\texttt{cindex}(w_1)}$$

Figure 23: We're considering initialization of C's first column, which describes the transition probability from the start state in the graph to the first tag t_i and word w_1. So, it turns out that the first column entries $c_{i,1}$ are given by the product of the transition probabilities of the initial states with the respective emission probabilities. Since transition probabilities are contained in the first row of our matrix A, we can look them up here, and then multiply with the emission probability found in matrix B. The function $\texttt{cindex}(\cdot)$ simply returns the column index in the matrix B for the given word fed as input argument.

Above, we considered how to fill in the first column of matrix C. What about matrix D? Matrix D stores the labels that represents the different states we're traversing when finding the most likely sequence of parts of speech tags for the given sequence of words $w_1, \ldots, w_k$. In the first column, we can simply set all entries to zero, i.e. $d_{i,1} = 0$ for all i, as there are no preceeding parts of speech tags we have traversed.

6.4.2 Forward Pass

We now use the Viterbi algorithm to continue populating entries in the two matrices we just initialized: C and D. We'll see how we can assign parts of speech tags to certain words in the sentence. The remaining entries in C and D are populated column-by-column. For matrix C, we have the relation:

$$c_{i,j} = \max_k c_{k,j-1} \times a_{k,i} \times b_{i,\texttt{cindex}(w_j)}.$$

In each $d_{i,j}$, we simply save the corresponding k which maximizes the entry in C, i.e.

$$d_{i,j} = \arg\max_k c_{k,j-1} \times a_{k,i} \times b_{i,\texttt{cindex}(w_j)}.$$

6.4.3 Backward Pass

In this section we'll see how we can use the probability matrix, the one we created in the previous section, to create a path, such that we may assign a parts of speech tag to every word. This is the last step of the Viterbi algorithm: retrieving the most likely parts of speech tags for our given sequence of words. Since we've already populated matrices C and D, all we have to do is extract the path from the matrix D, which represents the sequence of hidden states that most likely generated our sequence of observable words $w_1, \ldots, w_K$. The first step is to calculate the index of the most likely sequence of hidden states generating the given sequence of words, given by

$$s = \arg\max_i c_{i,K}$$

where K is the last column in matrix C. We use this index to traverse backwards through D to reconstruct the sequence of parts of speech tags.

Example Suppose we have some input sentence.

```
<s> w1 w2 w3 w4 w5
```

We computed matrix C with the following entries.

$$C = \begin{bmatrix} & w_1 & w_2 & w_3 & w_4 & w_5 \\ t_1 & 0.25 & 0.125 & 0.025 & 0.0125 & 0.01 \\ t_2 & 0.1 & 0.025 & 0.05 & 0.01 & 0.003 \\ t_3 & 0.3 & 0.05 & 0.025 & 0.02 & 0.0000 \\ t_4 & 0.2 & 0.1 & 0.000 & 0.0025 & 0.0003 \end{bmatrix}$$

Then if we inspect the last column for the largest entry, clearly $s = \arg\max_i c_{i,K} = 1$, i.e. the highlighted entry in the first row. This index represents the last hidden state you traversed when you observe the word w_5: so the most likely hidden state or parts of speech associated with word w_5 is t_1.

$$D = \begin{bmatrix} & w_1 & w_2 & w_3 & w_4 & w_5 \\ t_1 & 0 & 1 & 3 & 2 & 3 \\ t_2 & 0 & 2 & 4 & 1 & 3 \\ t_3 & 0 & 2 & 4 & 1 & 4 \\ t_4 & 0 & 4 & 4 & 3 & 1 \end{bmatrix}$$

So, we add t_1 to the end of the seqence and look up the next index in D, which tells you where you came from.

```
<s> w1 w2 w3 w4 w5
                t1
```

When we examine entry D_{t_1,w_5}, we see its value is 3, which indicates that the most likely parts of speech tag for the preceding word w_4 is t_3.

```
<s> w1 w2 w3 w4 w5
            t3 t1
```

We continue on in this fashion:

```
<s> w1 w2 w3 w4 w5
 pi t2 t3 t1 t3 t1
```

Implementation note: numerical stability Multiplying several really small probabilities (or floats) together can result in underflow. To get around this, we use log probabilities, i.e. instead of

$$c_{i,j} = \max_k c_{k,j-1} \times a_{k,i} \times b_{i,\texttt{cindex}(w_j)}.$$

we use

$$\log(c_{i,j}) = \max_k \Big[\log(c_{k,j-1}) + \log(a_{k,i}) \Big] + \log(b_{i,\texttt{cindex}(w_j)})$$

where numbers are summed instead of multiplied.

7 Autocomplete and Language Models

7.1 N-grams

Overview Recall that a text corpus is typically a large database of text documents, e.g. all of the pages on Wikipedia, or all of the books from an author, or all tweets from one account. A *language model* is a tool that calculates the probabilities of sentences (which is just a sequence of words). Language models estimate the probability of a subsequent word based on past words. Given the following sentence prefix, how would you fill in the next word?

```
Hellow, how are <...>
```

We'd probably guess "you"; building an n-gram language model can allow us to apply autocomplete to a sentence.

Language models have many applications: e.g. in *speech recognition* to convert the outputs to real words. As a particular example, an acoustic model may convert speech to text and it hears, "eyes awe of an", then the speech to text system can use a language model to know that it's more likely the sentence was "I saw a van". We can also use language models in *spelling correction*, to identify words that should be replaced based on the sentence they're in; e.g. Pr(entered the shop to buy) > Pr(entered the ship to buy). Lastly, they can be useful in *augmentative communication*, which are systems that take in a sequence of human hand gestures from a user and help them form words and sentences. E.g. people who are unable to physically talk or sign can instead use simple movements to select words from a menu; then, the system can speak for them, and word prediction using language models can suggest likely words from the menu.

In order to implement sentence autocompletion, we'll take several steps:

1. Transform your raw text corpus into a language model which returns the probability of the next word by using the previous words of a sentence.

2. Adapt your language model to deal with words the model hasn't seen before during training; these words are called out of vocabulary words. Here, we apply *smoothing* wherein we assume each out-of-vocabulary word or phrase has been seen at least once in the training corpus.

3. We then can choose the best language model using a perplexity metric.

What is an N-gram? An N-gram is simply a *sequence* of words. Note that the order matters, and so it's *not* as simple as a set of words. N-grams can also be characters or other elements, but for now let's focus on sequences of words. When we process a corpus, we'll treat punctuation like words, but all other special characters (such as codes), will be removed. Let's look at an example sentence:

`I am happy because I'm learning`

Then, we define *unigrams* for this corpus as the *set* of all unique single words that appear in the text.

`Unigrams` : {I, am, happy, because, learning}

Note that even though the word "I" appears in the corpus twice, it's once included once in the *set* of unigrams. *Bigrams* are defined as all sets of two words that appear contiguously (or side by side) in the corpus.

`Bigrams` : {I am , am happy, happy because, ...}

Notice again that the bigram "I am" can be found twice in the text but only is included once in the *set* of bigrams. We emphasize that the words must appear contiguously (or next to each other) in the corpus. As an example, the sequence of words "I happy" is not a bigram that appears in the corpus above, and so it has been omitted from the set of bigrams even though each individual word appears in the corpus. We can continue on in this fashion: *trigrams* represent unique *triplets* of words that appear in the sequence together in the corpus.

`Trigrams` : {I am happy, am happy because, ...}

Let's define some sequence notation. If we have a corpus of text that consists of 400 words, the sequence of words can be denoted by $w_1, w_2, w_3, \ldots, w_{500}$. The corpus length is denoted by the variable m. We will use the notation w_i^j to refer to the sequence of words $w_i, w_{i+1}, \ldots, w_{j-1}, w_j$.

Unigram probability In our example corpus "I am happy because I am learning", the size of the corpus is 7. So, for example, $\Pr(\text{I}) = \frac{2}{7}$ since the word "I" appears twice in the corpus. Similarly, $\Pr(\text{happy}) = \frac{1}{7}$. In general,

$$\Pr(w) = \frac{C(w)}{m},$$

where $C(w)$ denotes the frequency of word w in the corpus.

Bigram probability In the same example, "I am happy because I am learning", what is $\Pr(\text{am}|\text{I})$? This is given by $\Pr(\text{am}|\text{I}) = \frac{C(\text{I am})}{C(\text{I})} = \frac{2}{2} = 1$; i.e. we take the count of the bigrams observed in the corpus divided by the count of the unigram we are conditioning on. In this example, the bigram "I am" appears twice, and so does the unigram "I"; whence we arrive at our calculation. Similarly, $\Pr(\text{learning}|\text{am}) = \frac{C(\text{am learning})}{C(\text{am})} = \frac{1}{2}$. The generalized expression is given by:

$$\Pr(y|x) = \frac{C(xy)}{\sum_w C(xw)} = \frac{C(xy)}{C(x)},$$

where we remark that the last equality only follows if x is always followed by some other word!

Trigram probability Let's consider the same example, "I am happy because I am learning". $\Pr(\text{happy}|\text{I am}) = \frac{C(\text{I am happy})}{C(\text{I am})} = \frac{1}{2}$. In general,

$$\Pr(w_3|w_1^2) = \frac{C(w_1^2 w_3)}{C(w_1^2)},$$

where we could re-write the numerator several different ways: $C(w_1^2 w_3) = C(w_1 w_2 w_3) = C(w_1^3)$.

N-gram probability Let's generalize further:

$$\Pr(w_N|w_1^{N-1}) = \frac{C(w_1^{N-1}w_N)}{C(w_1^{N-1})}$$

where of course $C(w_1^{N-1}w_N) = C(w_1^N)$.

Modeling the probability of an entire sentence using N-grams How can we calculate the probability of a sentence, e.g. Pr(the teacher drinks tea) =? Let's recall the definition of conditional probability alongside the chain rule, i.e.

$$\Pr(B|A) = \frac{\Pr(A,B)}{\Pr(A)} \implies \Pr(A,B) = \Pr(A)\Pr(B|A).$$

Generalizing to longer sequences,

$$\Pr(A,B,C,D) = \Pr(A)\Pr(B|A)\Pr(C|A,B)\Pr(D|A,B,C).$$

The product of each successive word is the product of the successive probabilities of all words that came before it in the sequence. We can now apply the chain rule to answering our original question:

$$\Pr(\text{the teacher drinks tea}) = \Pr(\text{the})\Pr(\text{teacher}|\text{the})\Pr(\text{drinks}|\text{the teacher})\Pr(\text{tea}|\text{the teacher drinks}).$$

What if the sentence is not in the corpus? The corpus almost never contains the exact sentence we're interested in or even its longer subsequences! E.g.

$$\Pr(\text{tea}|\text{the teacher drinks}) = \underbrace{\frac{C(\text{the teacher drinks tea})}{C(\text{the teacher drinks})}}_{\text{both numerator and denominator terms likely zero}}.$$

The formula for the probability of the entire sentence can't give a probability estimate in this situation. As the sentence gets longer, the likelihood that more and more words will occur next to each other in this exact order becomes smaller and smaller. E.g. the likelihood that "the teacher drinks" appears in the corpus is smaller than the probability of the word "drinks".

Applying Markov assumption to model sentences using bigrams What if instead of looking for all the words before "tea", you just consider the previous word which is "drinks" in this case, i.e. we assume that

$$\Pr(\text{tea}|\text{the teacher drinks}) \approx \Pr(\text{tea}|\text{drinks}).$$

Then, our sentence probability calculate simplifies to

$$\Pr(\text{the teacher drinks tea}) \approx \Pr(\text{the})\Pr(\text{teacher}|\text{the})\Pr(\text{drinks}|\text{teacher})\Pr(\text{tea}|\text{drinks}).$$

In arguing that the approximation holds, we've effectively argued toward the Markov assumption where the probability of the subsequent word depends *only* on the previous word. Then, what we're saying is that the entire sentence can be modeled using a product of conditional probabilities

$$\Pr(w_1^n) \approx \prod_{i=1}^{n} \Pr(w_i|w_{i-1}).$$

We could also make this more flexible and instead assert that only the last N words matter, i.e. the Markov assumption here would be that the probability of a word depends only on its recent predecessors up through length N. For bigrams, we can approximate the probability

$$\Pr(w_n|w_1^{n-1}) \approx \Pr(w_n|w_{n-1})$$

whereas with N-grams we have

$$\Pr(w_n|w_1^{n-1}) \approx \Pr(w_n|w_{n-N+1}^{n-1}).$$

Starting and ending sentences In working with conditional probabilities that operate on a sliding window of two or more words, it's natural to wonder what happens at the beginning or end of a sentence? We'll use two new symbols to denote start and end markers for sentences, and we'll modify our input sentences using these symbols when processing bigrams or n-grams.

- Start of sentence symbol `<s>`
- End of sentence symbol `</s>`

Let's revisit our previous sentence, "the teacher drinks tea". We've previously asserted that

$$\Pr(\text{the teacher drinks tea}) \approx \Pr(\text{the}) \Pr(\text{teacher}|\text{the}) \Pr(\text{drinks}|\text{teacher}) \Pr(\text{tea}|\text{drinks}).$$

For the first word "the", we don't have context of the previous word, so you can't calculate a bigram probability, which we'll need to make predictions. We'll add in a special term so that each sentence of our corpus can be split into bigrams that we may calculate probabilities for.

$$\Pr(\texttt{<s>}\text{ the teacher drinks tea}) \approx \Pr(\text{the}|\texttt{<s>}) \Pr(\text{teacher}|\text{the}) \Pr(\text{drinks}|\text{teacher}) \Pr(\text{tea}|\text{drinks}).$$

Similar principles apply to N-grams. Instead of trying to calculate the sentence probability using a unigram probability for the first word, a bigram probability for the second word, and trigram probabilities for remaining words, as in:

$$\Pr(\text{the teacher drinks tea}) \approx \Pr(\text{the}) \Pr(\text{teacher}|\text{the}) \Pr(\text{drinks}|\text{the teacher}) \Pr(\text{tea}|\text{teacher drinks}).$$

We can instead convert our sentence to be "`<s><s>` the teacher rinks tea". So now, the sentence probability becomes the product of trigram probabilities:

$$\Pr(w_1^n) \approx \Pr(w_1| < s >< s >) \Pr(w_2| < s > w_1) \ldots \Pr(w_n|w_{n-2}w_{n-1}).$$

For an N-gram model, we simply add $N-1$ start tokens `<s>`. What about end of sentences? Recall that

$$\Pr(y|x) = \frac{C(xy)}{\sum_w C(xw)} = \frac{C(xy)}{C(x)}$$

where in the last step we simplified the denominator in a way that hinges on x always being followed by another word; i.e. when word x is the last word in th sentence, this simplification step does not hold. To see this, suppose we have a Corpus

```
Lyn drinks chocolate
John drinks
```

Then we have a discrepancy in the simplification step above (i.e. it doesn't hold), since clearly

$$\sum_w C(\text{drinks}\, w) = 1 \neq C(\text{drinks}) = 2$$

To continue using the simplified formula, we need to add end of sentence tokens. But there's another issue at stake: what if we have a very small corpus, e.g.

```
<s> yes no
<s> yes yes
<s> no no
```

There are only two unique words: {yes, no}. There are clearly $2^2 = 4$ possible sentences of length two, i.e

```
<s> yes yes
<s> yes no
<s> no no
<s> no yes
```

If we wanted to calculate $\Pr(\texttt{<s>}\,\text{yes}\,\text{yes})$, we could apply the Markov assumption and work with bigrams to say that it's equal to

$$\Pr(\texttt{<s>}\,\text{yes}\,\text{yes}) = \Pr(\text{yes}|\texttt{<s>}) \times \Pr(\text{yes}|\text{yes}) = \frac{C(\texttt{<s>}\,\text{yes})}{\sum_w C(\texttt{<s>}\,w)} \times \frac{C(\text{yes yes})}{\sum_w C(\text{yes}\,w)} = \frac{2}{3} \times \frac{1}{2}.$$

This is the probability of "yes yes" estimated from our corpus. We can do the same for all of the possible four combinations of bigrams:

$$\Pr(\texttt{<s>}\text{yes}\,\text{yes}) = \frac{1}{3}, \quad \Pr(\texttt{<s>}\text{yes}\,\text{no}) = \frac{1}{3}, \quad \Pr(\texttt{<s>}\text{no}\,\text{no}) = \frac{1}{3}, \quad \Pr(\texttt{<s>}\text{no}\,\text{yes}) = 0$$

Notice that $\sum_{2\text{ word}} \Pr(\ldots) = 1$, i.e. we have a probability distribution. We could do the same calculations for all $2^3 = 8$ possible sentences of length three, using the two words in our vocabulary. It turns out that we yield another probability distribution, i.e.

$$\Pr(\texttt{<s>}\text{yes}\,\text{yes}\,\text{yes}) = \ldots = \Pr(\texttt{<s>}\text{yes}\,\text{yes}\,\text{no}) = \ldots = \Pr(\texttt{<s>}\text{no}\,\text{no}\,\text{no})$$

and we again will find that $\sum_{3\text{ word}} \Pr(\ldots) = 1$. But what we really want is for the sum of the probabilities of *all* sentence lengths to be equal to one: in this way we can compare probabilities for sentences of different lengths. In other words, we want

$$\sum_{2\text{ word}} \Pr(\ldots) + \sum_{3\text{ word}} \Pr(\ldots) + \ldots = 1$$

There's a surprisingly simple trick for this: we pre-process our training corpus to simply add in an end of sentence token which we will denote by $< /s >$, after each sentence. So sentences get pre-processed:

```
the teacher drinks tea => <s> the teacher drinks tea </s>
```

The probability of our sentence is now modeled by

$$\Pr(\text{the}|\texttt{<s>}) \Pr(\text{teacher}|\text{the}) \Pr(\text{drinks}|\text{teacher}) \Pr(\text{tea}|\text{drinks}) \Pr(\texttt{<s>}|\textit{tea}).$$

This new term captures the probability that our sentence will end after the word "tea". This also fixes the issue with probability of the sentences of certain length being equal to one. Does this also resolve our problem with the bigram probability formula?

```
<s> Lyn drinks chocolate </s>
<s> John drinks </s>
```

Now there are two bigrams starting with the word "drinks", i.e. $\sum_w C(\text{drinks}\,w) = 2$ (which in this example are "drinks chocolate" and "drinks </s>"). Also realize that now this is consistent with $C(\text{drinks}) = 2$, so we can continue using our simplified formula for the bigram probability calculation.

What about for N-grams in general? It turns out that adding simply one end-of-sentence marker per sentence is enough. To see why, suppose we are processing trigrams and we have the example sentence

```
the teacher drinks tea => <s> <s> the teacher drinks tea </s>
```

We've used two start-of-sentence tokens because we're going to be working with trigrams.

Let's segway and quickly look at a bigram example. Suppose our corpus looks like

<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>

Then from this we can easily calculate conditional probabilities for a sample of bigrams:

$$\Pr(\text{John}|\texttt{<s>}) = \frac{1}{3}, \quad \Pr(\texttt{</s>}|\text{tea}) = \frac{1}{1}, \quad \Pr(\text{chocolate}|\text{eats}) = \frac{1}{2}, \quad \Pr(\text{Lyn}|\texttt{<s>}) = \frac{2}{3},$$

Now, let's calculate the probability of the sentence "Lynn drinks chocolate", this is given by $\frac{2}{3}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{2}{2} = \frac{1}{6} < \frac{1}{3}$, i.e. the probability returned by the model is lower than $\frac{1}{3}$ which is the fraction of sentences which match in the training corpus. This also applies to other sentences in the corpus; this is how the model generalizes!

7.1.1 The N-gram language model

Here's the general outline:

- Process the corpus into a Counts matrix; this captures the number of occurrences of relative n-grams.
- Transform the count matrix into a probability matrix that contains information about the conditional probability of the n-grams.
- Relate the probability matrix to the language model.
- Generative language model: we can then use the n-gram language model to generate new sentences from scratch.

We'll also learn how to deal with technical numerical issues that arise from multiplying a lot of small numbers when calculate the sentence probability.

Counts matrix Starting with the counts matrix, let's transform into conditional probabilities. The formula for this is given by:

$$\Pr(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})}. \tag{7}$$

The numerator of this formula is captured by the Counts matrix for all n-grams appearing in the corpus. All unique corpus $(N-1)$-grams make up the rows, and all unique words of the corpus make up the columns.

	<s>	</s>	I	study	learn
<s>	0	0	1	0	0
</s>	0	0	0	0	0
I	0	0	0	1	1
study	0	0	1	0	0
learn	0	1	0	0	0

Table 9: Suppose the corpus is: "<s> I study I learn </s>". The rows of the table above represent the first word of the bigram and the columns represent the second word of the bigram. For the bigram "study I", it appears once in the corpus. We can calculate hte count matrix in a single pass through the corpus using a sliding window composed of $n = 2$ words to represent the bigram: for each bigram you find, increment the value in the count matrix by one.

Probability matrix Now that we've used the counts matrix to provide the numerator for the n-gram probability formula, it's time to get the denominator in equation 7: this is as simple as normalizing each row by its row-sum. I.e.

$$\texttt{sum}(\text{row}) = \sum_{w \in V} C(w_{n-N+1}^{n-1}, w) = C(w_{n-N+1}^{n-1}).$$

The row sum is equivalent to the counts of the $(n-1)$-gram prefixes from the formula denominator: this will always be true since the $(n-1)$-gram prefix is always followed by some word. If the prefix was at the end of the sentence, it is now followed by the end of sentence token. Let's create a probability matrix from an example; the corpus is again "<s> I study I learn </s>".

	<s>	</s>	I	study	learn	sum
<s>	0	0	1	0	0	1
</s>	0	0	0	0	0	0
I	0	0	0	1	1	2
study	0	0	1	0	0	1
learn	0	1	0	0	0	1

Rescale by row sums $\longrightarrow$

	<s>	</s>	I	study	learn
<s>	0	0	1	0	0
</s>	0	0	0	0	0
I	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$
study	0	0	1	0	0
learn	0	1	0	0	0

Language model The next step is to connect the probability matrix with our definition of the language model; the language model is now a simple script that uses the probability matrix to estimate the probability of a given sentence. It estimates the probability by splitting the sentence into n-grams and then finding their probability in the probability matrix. Alternatively, the language model can predict the next element of a sequence by extracting the last $(n-1)$-gram from the end of a sequence; after that, the language model finds the corresponding row in the probability matrix and returns the word with highest probability. E.g. let's find the probability of the sentence "I learn", using the calculated probabilities from the previous section:

$$\Pr(\texttt{<s>}\ \text{I learn}\ \texttt{</s>}) = \Pr(\text{I}|\texttt{<s>})\Pr(\text{learn}|\text{I})\Pr(\texttt{</s>}|\text{learn}) = 1 \times \frac{1}{2} \times 1 = \frac{1}{2}.$$

Log probability The sentence probability calculation requires multiplication of many small, positive numbers less than unit value:

$$\Pr(w_1^n) \approx \prod_{i=1}^{n} \Pr(w_i|w_{i-1}).$$

This puts us as risk of running into numerical underflow. We use the fact that $\log(ab) = \log a + \log b$ to avoid this problem and proceed by calculating a log probability instead.

Generative language model One interesting application of a language model is text generation from scratch or using just a small hint. Suppose our corpus is given by:

```
<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>
```

E.g. the algorithm might choose "<s> Lynn" to start. Next, it will select a bigram that starts with "Lynn", perhaps "Lynn drinks". Then it chooses a bigram starting with "drinks", say "drinks tea". Finally, perhaps the model predicts an end of sentence at that point. Notice that our model will always predict end of sentence after predicting "tea". How does such an algorithm work?

- Randomly select among all bigrams starting with the sentence symbol <s> based on bigram probability; i.e. bigrams with higher values in the probability matrix are more likely to be chosen.

- Choose a new bigram at random from the bigrams beginning with the previously chosen word, and add this bigram to your sentence.
- Continue until the end of sentence token `</s>` is chosen; this happens when we randomly choose a bigram that starts with the previously chosen word and ends with the token `</s>`.

7.2 Perplexity

How can you tell how well your language model is performing? Recall that a language model assigns a probability to each sentence, where the model was trained on the corpus. Realize that for sentences appearing in the corpus, the model may assign relatively higher probabilities. Therefore, we should first split the corpus to have some testing and validation data that are *not* used in training. This is similar to other ML workflows: we create training, validation, and test sets of data. The training set is used to train your model, the validation set is used for things like tuning hyper-parameters, and the test set is held out for the end to estimate how the model would perform on unseen data as measured by an accuracy score.

In NLP, there are two main methods for splitting: (i) you can split the corpus by choosing longer continuous segments, such as Wikipedia articles, or (ii) you can randomly choose short sequences of words such as those within the sentences.

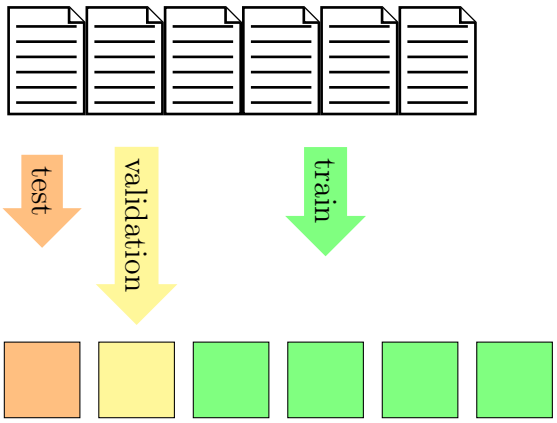


Figure 24: Splitting data based on longer continuous sequences such as entire (e.g. Wikipedia) documents: notice that we partition our train, validation, and test sets by document.

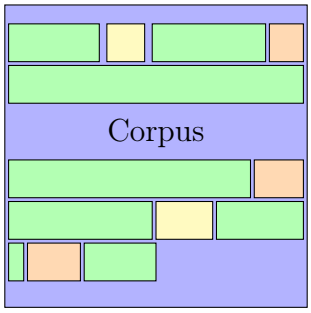


Figure 25: We can also randomly choose short sequences of words such as those that are within sentences to form training, validation, and test sets of data.

Once we have our training data split, we can evaluate our model on the test set using the perplexity metric.

$$\texttt{PP}(W) = \Pr(s_1, s_2, \ldots, s_m)^{\frac{-1}{m}}.$$

Where W is the test set containing m sentences s, s_i is the i-th sentence in the test set (each ending with `</s>`), m is the number of all words in the entire set including `</s>` but not including `<s>`. Perplexity is designed to measure whether a sequence of text looks like it was written by a human as opposed to being generated by a computer program. A text that is written by humans is more likely to have a lower perplexity score, whereas a text chosen at random is more likely to have a higher perplexity score. We start by calculating the probability of all sentences in your test set, and then raise the probability to the power $-1/m$. I.e., perplexity is essentially the inverse probability of the test set normalized by the number of words in the test set. So, the higher the language model estimates the probability of your test set, the lower the perplexity is going to be.[7]

Let's take a look at an example of two language models that are going to return different probabilities for your test set W. Suppose there are 100 words in your test set, so $m = 100$. The first model returns a 0.9 probability of the test set, which is very high and indicates the model is effective and is predicting the test set well. The perplexity is near 1: $\Pr(W) = 0.9 \implies \texttt{PP}(W) = 0.9^{\frac{-1}{100}} \approx 1$. The second model has a low probability for the test set, say 10^{-250}, then $\texttt{PP}(W) = (10^{-250})^{\frac{-1}{100}} \approx 316$ which is very high.

For context, good language models have perplexity scores between $[20, 60]$ and sometiems even lower for English. Perplexities for character level language models (where you track characters instead of words) will be lower.

Perplexity in a bigram language model In a bigram model, you calculate the products of bigram probabilities for all sentences, then take raise to the power $-1/m$. Realize that the bigger the test set is (i.e. the larger m is), the lower the final perplexity will be.

$$\texttt{PP}(W) = \sqrt[m]{\prod_{i=1}^{m} \prod_{j=1}^{|s_i|} \frac{1}{\Pr(w_j^{(i)} | w_{j-1}^{(i)})}}$$

where $w_j^{(i)}$ denotes the j-th word in the i-th sentence. If all sentences in the test sets are concatanated, the formula can be simplified to the products of probabilities of bigrams in the entire sets.

$$\texttt{PP}(W) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{\Pr(w_i | w_{i-1})}}$$

where in this context w_i denotes the i-th word in the test set.

Log perplexity Some papers use log-perplexity. This means that we apply the transform

$$\texttt{PP}(W) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{\Pr(w_i | w_{i-1})}} \longrightarrow \log \texttt{PP}(W) = \frac{-1}{m} \sum_{i=1}^{m} \log_2 \Pr(w_i | w_{i-1}).$$

In a good model with perplexity between $[20, 60]$, the log-perplexity would range from $[4.3, 5.9]$.

7.3 Unknown Words

Sometimes, we'll encounter words that we did not see during training, i.e. unknown or *out-of-vocabulary* words; we can add a special new word `UNK` to our probability calculations from our training corpus to accommodate this. We can also consider which words should be considered known by our vocabulary. Let's look at this in further detail.

[7] Perplexity is closely related to entropy, which measures uncertainty.

What is an out of vocabulary word? A vocabulary is a set of unique words supported by a language model. There are certain tasks such as question answering where you will encounter and generate only words from a fixed set of words: i.e. a chatbot can only answer limited sets of questions. The fixed set of words is what's formally called a *closed vocabulary*. But in other use cases, you have to deal with words you haven't seen before which results in what's called an *open vocabulary*. E.g. you might see a name of a new city in your training data. The unknown words are called out of vocabulary words (or OOV for short).

Adding a new special word `UNK` One way to deal with the unknown words is to replace them by a special word `<UNK>`. This literally means replacing each instance of an unknown word with `<UNK>` in your data.

1. Create a vocabulary V.
2. Replace any word that is in the training corpus but *not* in vocabulary by `<UNK>`.
3. Count the probabilities with `<UNK>` as with any other word.
4. Now, we may apply the n-gram language model probability calculations in the same way as before, just with the addition of `<UNK>`.

E.g. suppose we've been given the following training corpus:

```
<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>
```

We have decided to only include words in our vocabulary that appear at least twice in our training corpus. The post-processed corpus will then look like

```
<s> Lyn drinks chocolate </s>
<s> <UNK> drinks <UNK> </s>
<s> Lyn <UNK> chocolate </s>
```

Our vocabular includes words with frequency (in our training corpus) greater than or equal to two, i.e. $V = \{\text{Lyn, drinks, chocolate}\}$. If we were given a new sentence, say "Adam drinks chocolate", then we would also replace any out-of-vocabulary words with `<UNK>`, i.e.

```
<s> Adam drinks chocolate </s>  -->  <s> <UNK> drinks chocolate </s>
```

How to create a vocabulary V We can create a vocabulary from a training corpus based on different criteria.

- Minimum word frequency f (usually a small number): i.e. only include words in the vocabulary V that appear at least f times in our training corpus. Replace all words that appear fewer than f times with `<UNK>`. This ensures that the words that "we care about a lot", i.e. the ones which repeat a lot in our training corpus, are certainly present in our vocabulary.
- Alternatively, we could decide what the maximum size of our vocabulary should be, i.e. fix $|V|$, and then include words with highest frequency until we fill the vocabulary.

Effect of <UNK> on perplexity? One thing to be mindful of when using <UNK>s is the effect on measures of perplexity. It turns out that the more <UNK>s we use, the lower the perplexity tends to be; it'll look like our language model is getting better and better, but our model might be prone to spitting out a sequence of <UNK>s with high probability instead of a meaningful sentence. Due to this limitation, we recommend using <UNK>s sparingly. Lastly, and for the aforementioned reasons, it's important to only compare models using perplexity when they have the same vocabulary V.

7.4 Rare Words and *Smoothing*

When we train an n-gram model on a limited corpus, some of the probabilities for rare words may be skewed; we can remedy this using a technique known as *smoothing*. We'll first see an example of how an n-gram missing from the corpus can affect the estimation of the n-gram probability. Then, we'll cover popular smoothing techniques. And finally, we'll touch on other methods such backoff and interpolation.

Missing n-grams We have resolved the issue of completely unknown words. But there is still missing information from our language model: i.e. how can we manage the probability of an n-gram made up of words appearing in the corpus, but where the n-gram itself is not present? Recall our example corpus:

```
<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>
```

Notice that each of the words {John, eats} are present in the corpus, but that the bigram "John eats" is not. The count of the bigram "John eats" is zero, and correspondingly the probability returned by a naive model would be zero as well. I.e. the numerator and denominator terms on the right hand side of the following expression

$$\Pr(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})}$$

can be zero; everything that doesn't occur in the training corpus is considered impossible.

7.4.1 Techniques for Smoothing

This is a technique that can help with the aforementioned problem of missing n-grams when constructing n-gram language models. Recall that we've used smoothing in constructing the transition probabilities for parts of speech tagging. We'll use the same idea for n-gram probabilities.

Add one smoothing Let's focus on a technique known as "add-one smoothing" otherwise known as *Laplacian smoothing*, which mathematically changes our probability calculations to

$$\Pr(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{\sum_{w \in V} [C(w_{n-1}, w) + 1]} = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + |V|}.$$

This change can be interpreted as simply adding one occurrence to each bigram; this means that bigrams that were missing from the corpus will now have a strictly nonzero probability. In the denominator, we are adding one for each possible bigram, starting with the word w_{n-1}: this corresponds to adding one to each cell in the row indexed by the word w_{n-1} in the counts matrix. We repeat this for as many times as there are words in the vocabulary. Because the one in the sum doesn't depend on index w, we can take it out of the sum and replace with $|V|$. This technique really *only works* when the real counts are large enough to outweigh the "plus one" term however: otherwise, the probabilities of missing words would be too high. That said, add-one-smoothing helps quite a lot because now there are no bigrams with identically probability.

Add k smoothing If you have a larger corpus, you can instead add $k > 1$. This technique makes the probabilities *even smoother*. The formula is similar:

$$\Pr(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-1}, w_n) + k}{\sum_{w \in V} [C(w_{n-1}, w) + k]} = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + k \times |V|}.$$

Other more advanced smoothing methods include Kneser-Ney or Good-Turing.

Backoff There's another technique to dealing with n-grams that do not occur in the training corpus, and that is to use information about $(n-1)$-grams, $(n-2)$-grams, and so on. With the backoff approach, if n-gram information is missing, you use $(n-1)$-gram, but if that's also missing you resort to using $(n-2)$-gram, and so on until you find a non-zero probability. Using the lower level n-grams, i.e. $(n-1)$-gram, $(n-2)$-gram, all the way down to a unigram, distorts the probability distribution. Especially for smaller corpora, some probability needs to be discounted from the higher level n-gram in order to use it for lower-level n-grams. Katz backoff method uses this technique when counting. For large sized, web-scale corpora, a method known as "stupid-backoff" has also been shown to be effective: in this approach, no discounting is applied. If a higher order n-gram probability is missing, lower order n-grams are used, just multiplied by some constant (a constant of about $\frac{4}{10}$ was shown to work well empirically). Let's apply Backoff to an example

```
<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>
```

In this example, the probability of the trigram "John drinks chocolate" can't be directly estimated from the corpus. So, the probability of the bigram "drinks chocolate", multiplied by some constant (say $c = \frac{4}{10}$), would be used instead.

$$\Pr(\text{chocolate}|\text{John drinks}) \approx c \times \Pr(\text{chocolate}|\text{drinks}).$$

Interpolation Another alternative to backoff is to use a linear interpolation of all orders of n-gram. This means that you would always combine the weighted probability of the n-gram, $(n-1)$-gram, all the way down to unigrams. E.g. when calculating the probability of the trigram

$$\Pr(\text{chocolate}|\text{John drinks}) = \lambda_1 \Pr(\text{chocolate}|\text{John drinks}) + \lambda_2 \Pr(\text{chocolate}|\text{drinks}) + \lambda_3 \Pr(\text{chocolate}),$$

where $\lambda_1 > \lambda_2 > \lambda_3$ and $\sum_i \lambda_i = 1$. More generally,

$$\hat{\Pr}(w_n|w_{n-2}\, w_{n-1}) = \lambda_1 \Pr(w_n|w_{n-2}\, w_{n-1}) + \lambda_2 \Pr(w_n|w_{n-1}) + \lambda_3 \Pr(w_n) \qquad \text{where} \qquad \sum_i \lambda_i = 1.$$

The lambdas are *learned* from the validation parts of the corpus: you get them by maximizing the probability of sentences in the validation set. To do this, you use a fixed language model trained from the training parts of the corpus to calculate n-gram probabilities and then optimize the λs. The interpolation can be apply to general n-grams by applying more λs.

8 Word Embeddings with Neural Networks

Word vectors, also known as word embeddings, are the cornerstone of NLP. Recall that we've previously used word embeddings to find semantic analogies between words, and also to calculate distances between words and similarities between words. Separately, it's possible to perform sentiment analysis by combining word embeddings with a classifier, or to classify customer reviews (or comments) from user feedback surveys. Advanced uses of word embeddings include machine translation systems, information extraction, and question-answering. Our key learning objectives in this section are:

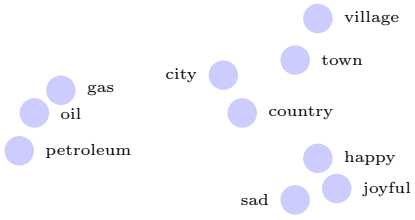


Figure 26: As an example, word embeddings help capture semantic relationships. When we project the word embeddings into a lower dimensional space, we can often times see clusters of words that represent related concepts.

- Identify the key concepts of word representations: we will represent words numerically such that we may use them in mathematical models.
- Generate word embeddings: we'll see how we can *learn* word embeddings from data with some commonly used methods.
- Prepare text for machine learning: transform a corpus of text into training sets for a machine learning model.
- Implement the continuous bag-of-words model, which is one of the many ways word embedding can be created.

8.1 Word Representations

Let's now learn how to create a matrix to represent all of the words in our vocabulary. Each vector in the matrix will correspond to one of the words.

Naively: represent words as integers The simplest way to represent words as numbers is, for a given vocabulary, to assign a unique integer ID to each word. E.g. based on a vocabulary of 1,000 basic English

Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000

words, we might assign the number one to word "a", the number two to "able", and so on until we assign something like one-thousand to "zebra". While this schema might seem simple, the problem is that the order of words, alphabetical in this example, doesn't make much sense from a semantic perspective. For example, there's really no good intuitive reason why "happy" should be assigned a greater number than "hand" for instance, or a smaller one than "zebra".

One-hot vectors Instead, let's try using a numerical column vector to represent a word of the vocabulary. If again we have a vocabulary of 1,000 words, each vector would contain 1,000 elements and each row would be labeled with one of the words. We can now use a one-hot encoding: place a one in the row that has the same label and a zero everywhere else. For example, the word "happy" would be represented as a column vector where there is a unit-value in the row corresponding to "happy" and zeros for all other rows in the vocabulary.

"happy"	
0	a
0	able
0	about
⋮	⋮
0	hand
⋮	⋮
1	happy
⋮	⋮
0	zebra

Table 10: The above column-vector is a one-hot representation of the word "happy". It has as many row-entries as there are elements in our vocabulary.

Pros and cons of one-hot vectors Note that one-hot vectors have the advantage over using integers beacuse one-hot vectors do not imply any relationship between two words (other than they are either equal or not), e.g. each vector simply says the word is either "happy" or it's not, or the word is either "zebra" or it's not. There are, however, two major limitations for using one-hot vectors in NLP: (i) barring the smallest of vocabularies, these word vectors are going to be huge, which means processing/working with them on a compute will take lots of space and time, and (ii) their representation again doesn't carry the words semantic meaning. For instance, *all* words that are not the same have the same distance between them, e.g. the word "happy" is just as similar to the word "paper" as it is to the word "excited" (but intuitively, we would think that "happy" and "excited" are closer in distance, of course).

Mapping between integer representation to one-hot vectors Realize that we can easily map back and forth between one-hot vectors and the integer representation for a word. We do this simply by mapping the words in the rows to their corresponding row number. In this way, e.g. "happy" may be word 621 would be represented as a column vector with a 1 in row 621, and it can also be represented as the integer-ID 621.

8.2 Word Embeddings

Although there are basic ways to represent words as numbers, these representations fail to carry meaning. Let's now move to embeddings, which are vectors that do indeed carry meaning.

Intuition for how a (scalar) vector representation can capture semantic meaning Let's first get an intuition for how word vectors can carry the meaning of words, starting in 1D.

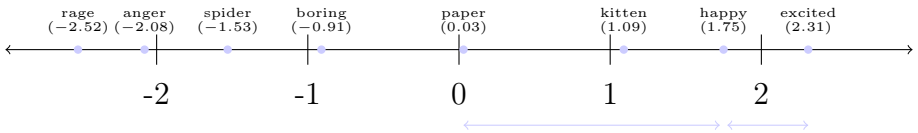


Figure 27: Consider a horizontal number line, where words to the left of the origin are considered negative in some way and words on the right are considered positive in some way. Words that are much more negative are further to the left, and words that are much more positive are further to the right. You could store their positions as scalars (or vectors of length 1). Notice that you can use any decimal value instead of binary values $\{0, 1\}$, which means that we can now say that "happy" and "excited" are more similar to each other compared with the word "paper": their distance is smaller.

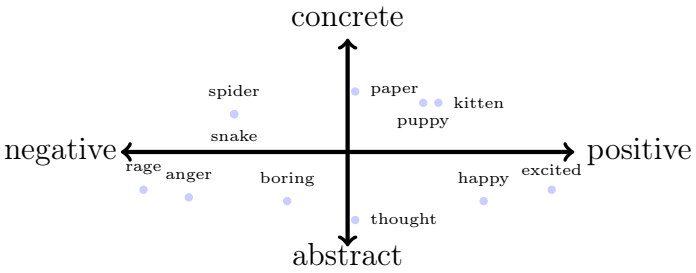


Figure 28: We have now extended our word representation into two dimensions: the vertical number line represents how concrete (or abstract) the object is; words that are higher on the number line are more concrete and words that are lower on the number line are more abstract. Notice that words that are both more concrete and positve like "puppy" and "kitten" are close together. We can store the two numbers that represent the position on the cartesian plane as a vector of length two. So, what we've done here is to represent the vocabulary of words with a small vector of length two. We've gained a little bit of meaning while also giving up some precision: the vector is less precise than one-hot vectors in that it's possible for two words to be located on the same point in this 2D plot, such as the words "snake" and "spider".

Extending our intution to more than 1 dimension Let's now consider how adding a second dimension can help capture more meaning of words.

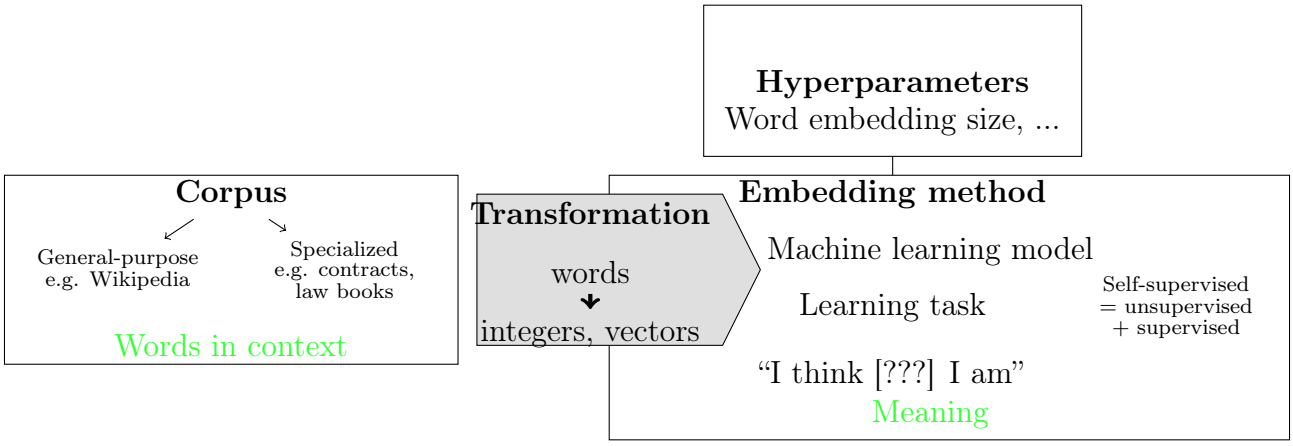
Word embeddings represent words in a vector form that both has relatively low dimension (e.g. hundreds to thousands), which makes it practical for calculations, and carries the semantic meaning between words, which makes it possible to determine how semantically close a pair of words really are. In a general purpose vocabulary, we might expect the vector representation for "forest" to be similar to "tree", but for both of those to be very different for the vector "tickets". Using word vectors, we can also fill in analogies: e.g. "Paris is to France as Rome is to?". Encoding the meaning of words is also the first step toward encoding the meaning of entire sentences, and this is the foundation for more complex NLP use cases (such as question-answering and translation).

8.2.1 How to Create Word Embeddings

To create word embeddings we need two things: a corpus of text and an embedding method.

Choice of corpus The corpus contains the words you want to embed organized in the same way they would be used in the context of interest. For example, if you want to generate embeddings based on Shakespeare, then your corpus would be the full and original text of Shakespeare and *not* study notes, slide presentations, or keywords from Shakespeare, for example. The context of a word refers towards other

words or a combination of other words said to occur around that particular word; the context is important as this is what will give meaning to each word embedding. I.e. a simple vocabulary list of Shakespeare's most common words would *not* be enough to create an embedding. A corpus could be either a general purpose set of documents such as Wikipedia articles, *or* it could be more specialized such as an industry or enterprise specific corpuse to capture the nuances of the context (e.g. for NLP use cases on legal topics, you could use contracts and law books as the corpus).



Embedding methods as a self-supervised task The embedding method creates the word embeddings from the corpus. There are many types of possible methods, but we will focus on modern methods based on machine learning models which are set to learn the word embeddings. In particular, the ML method performs a learning task and the main byproducts of this task are the word embeddings. For instance, the task could be to learn to predict a word based on the surrounding words in a sentence of the corpus, as in the case of the continuous bag-of-words approach. The specifics of the task will ultimately define the meaning of the individual words. We say that the task is *self-supervised*: it is both unspervised in the sense that the input data, the corpus, is unlabeled, but it's also supervised in the sense that the data itself provides the necessary context which would ordinarily make up the labels. So, the corpus is a self-contained data set that contains both the training data and the data that enables supervision of the task.

Dimension of embeddings Word embeddings can be tuned with hyperparameters, one of which is the *dimension* of the word embedding. In practice, the size of an embedding is a few hundred to a few thousand dimensions: using more dimensions captures more nuanced meanings but is more computationally expensive both at training time and later down the line when using the word embedding vectors, which leads to diminishing returns.

8.2.2 Word embedding methods

There are many ways to generate word embeddings, e.g. `word2vec` by Google, which initially popularized the use of ML to generate word embeddings. The algorithm uses a shallow neural netwok to learn embeddings: it proposes two model architectures

- Continuous bag of words: a simple but efficient approach. The objective is to learn to predict a missing word given surrounding words.
- Continuous skip-gram, also known as skip-gram with negative sampling: does the reverse of the continuous bag of words method by learning to predict the word surroundings given an input word.

There's also Global Vectors (or GloVe) by Stanford, which involves factorizing the logarithm of the corpus's word co-occurrence matrix, which is similar to the counter matrix we've used before. Then there's fastText by Facebook, which is based on the skip-gram model and takes into account the structure of words by representing words as an n-gram of characters: this enables the model to support previously unseen words, by inferring their embedding from the sequence of characters they are made of and the corresponding sequences that it was initially trained on. E.g. it can create similar embeddings for the words "kitty" and "kitten" even if it had never seen the word "kitty" before: this is because the two words are made up of a similar sequence of characters. Another benefit of fastText is that word embedding vectors can be averaged together to make vector representations of phrases and sentences.

Advanced, context specific, word embedding methods Deep learning methods can also be used to refine the representation of words according to their contexts. In the previous models, a given word always has the same embedding. IN these more advanced models, words have different embeddings depending on their surrounding context. This adds support for *polysemy* or words with similar meanings: e.g. the word "plant" can either refer to an organism (such as a flower) a factory, or which can even be an adverb. A few examples of more advanced models that generate word embeddings are BERT (or bidirectional encoder representation from transformers by Google), ELMO (embeddings from language models), or GPT-2 (generative pre-training 2 by OpenAI). If you want to use these advanced methods, you can find off the shel pre-trained models on the internet, and then fine tune these models using your own corpus to generate high-quality, domain specific word embeddings.

8.2.3 Continuous bag-of-words model

Recall that to create word embeddings, we need a corpus and a machine learning model that performs a learning task. One byproduct of the task is a set of word embeddings. You also need a way to transform the corpus into a representation that is suited to the machine learning model. In the case of the continuous bag-of-words model, the objective of the task is to predict a missing word based on the surrounding words. The rationale is that if two unique words are both frequently surrounded by similar sets of words when used in various sentences, then those two words tend to be related in their meaning; put differently, they are related semantically.

```
The little <???> is barking
```

In the above sentence, something is barking. With a large enough corpus, the model will learn to predict that the missing word is related to dogs, such as the word dog itself or puppy, hound, terrier, and so on. So, the model will end up learning the meaning of the word based on the context.

Using the corpus to create training data for the prediction task Let's say that the corpus is the sentence

```
I am happy because I am learning
```

We will ignore punctuation for now. For a given word in the corpus, e.g. "happy", which we will call the *center* word, we define the *context words* as the four surrounding words (the two words before, and the two words after). We will denote $C = 2$ as the half-size of the context, and it is a hyperparameter of the model.

To train our model, we'll need a set of examples. Each example will be made of context words and the center word to be predicted. In the first example, the window is "I am happy because I", and the model will take the context words "I am because I", and should predict the center word "happy". We now slide the window by one word: the next training example that you have is "am happy beause I am", where the input to the model is the context words "am happy I am" and we want to predict the target center word

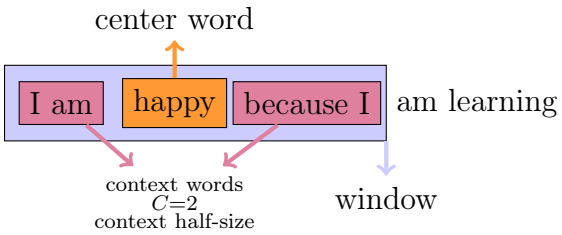
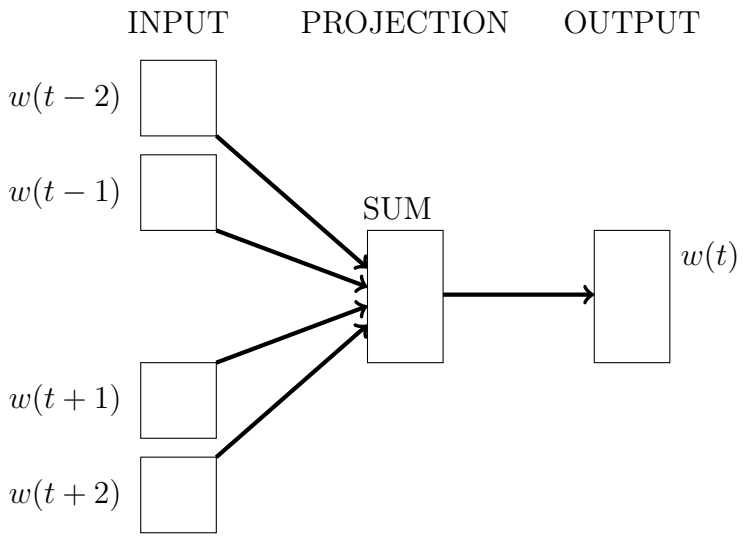


Figure 29: In this example, the context words for the center word "happy" are "I am" and "because I". We define the *window* as the count of the center word plus the context words; so, in this example the size of the window is equal to five.

"because". We continue to slide the window by one word again, and the model will take the window "happy because I am learning", and we should predict "I". This is described in detail in Efficient estimation of Word Representations in Vector Space. The model architecture is given by a shallow neural network:



8.3 Cleaning and Tokenization

Let's go into more details on a fundamental technique for cleaning text data, in particular we'll discuss practical advice on how to clean a corpus and split it into words, or more accurately, tokens, through a process known as *tokenization*.

1. Consider the words in our corpus as case insensitive, i.e. treat "The" identically no matter its casing/capitalization. This can be achieved by converting the entire corpus to either all lowercase or all uppercase.

2. Handle punctuation, which can be done in a variety of ways and which is suitable depends on the context. For example, we might choose to represent all interrupting punctuation marks (such as full stops, commas, and question marks) as a single special word of the vocabulary. You could choose to ignore non-interrupting punctuation marks such as quotation marks. You could also collapse multi-sign marks, such as triple question marks, into a single mark and so on.

3. You may want to handle numbers. E.g. if numbers don't carry an important meaning in your use case, you could drop all of the numbers. However, numbers may have significant meaning that is

relevant to your use case. Examples include 3.14 (an approximation to π), 90210 (the name of a television show and the area code for Beverly Hills, CA). Therefore, one could argue towards keeping these numbers in the corpus as is. If your corpus has many unique numbers, e.g. many area codes, you may find that it makes more sense to replace all the numbers with a special token, such as <NUMBER>. This allows the model to know that the important thing is that it's a number, instead of trying to distinguish between numbers themselves.

4. Handle special characters, e.g. mathematical symbols, currency symbols, section and paragraph signs, and line markup signs etc. It's sometimes safe to drop these entirely.

5. Lastly, especially if working on a modern corpus of user inputs (e.g. tweets or user reviews), you should handle special words such as emojis and hashtags. Depending on if and how you want your model to confer meaning to them, you could, for example, consider that each emoji or hashtag be treated as an individual word.

Consider as an example "Who ♡ "word embeddings" in 2020? I do!!!". This short tweet contains an emoji, punctuation, and a number.

```
# pip install nltk
# pip install emoji
import nltk
from nltk.tokenize import word_tokenize
import emoji

nltk.download('punkt') # Download pre-trained Punkt tokenizer for English.

# Use a regex to collapse all interrupting punctuation signs and replace them with a full-stop.
# Question mark and exclamation points replaced with single full-stop.
data = re.sub(r'[,!?;-]+', '.', corpus)
data = nltk.word_tokenize(data) # tokenize strings to words.
data = [ ch.lower() for ch in data
         if ch.isalpha()
         or ch == '.'
         or emoji.get_emoji_regexp().search(ch)
       ]
```

Sliding window of words in Python Suppose we've cleaned and tokenized the corpus and we now have an array of tokens or words. Let's now consider how to extract the center words and their context words, which will serve as examples to train the continuous bag of words model.

```
def get_windows(words, C = 2):
  """ Words is a list of tokens."""
  i = C
  while i < len(words) - C:
    center_word = words[i];
    context_words = words[(i-C):i] + words[(i+1):(i+C+1)]
    yield context_words, center_word
    i += 1
```

Notice this is a *generator* function; with a *yield* we can pause execution of the function until the next time it is invoked.

Transforming words into vectors At this point, we've cleaned and tokenized our corpus and extracted the context words and center word from a window that slides across our prepared corpus. We're now ready to transform the center words and context words into a mathematical form that's going to be consumed by the continuous bag of words model. Let's start with the center word: for the sample corpus

```
I am happy because I'm learning
```

we first create a vocabulary, which is the set of unique words in the corpus $V = \{\text{am, because, happy, I, learning}\}$. We can now encode each word as a column one-hot vector, just like in section 8.1. In this way, the vector for "am" will have a unit value in the row corresponding to "am" and a zero everywhere else, for example. For the context words, we'll create a *single* vector that represents the average of individual one-hot vectors. E.g.

$$\frac{1}{4} \times \begin{bmatrix} & \text{I} & & \text{am} & & \text{because} & & \text{I} \\ \text{am} & 0 & & 1 & & 0 & & 0 \\ \text{because} & 0 & & 0 & & 1 & & 0 \\ \text{happy} & 0 & + & 0 & + & 0 & + & 0 \\ \text{I} & 1 & & 0 & & 0 & & 1 \\ \text{learning} & 0 & & 0 & & 0 & & 0 \end{bmatrix} = \begin{bmatrix} \text{I am because I} \\ 0.25 \\ 0.25 \\ 0 \\ 0.5 \\ 0 \end{bmatrix}$$

So, if we repeat this process for our vocabular, we come up with the following inputs for our ML algorithm.

Context words	Context words vector	Center word	Center word vector
I am because I	$[0.25; 0.25; 0; 0.5; 0]$	happy	$[0; 0; 1; 0; 0]$
am happy I am	$[0.5; 0; 0.25; 0.25; 0]$	because	$[0; 1; 0; 0; 0]$
happy because am learning	$[0.25; 0.25; 0.25; 0, 0.25]$	I	$[0; 0; 0; 1; 0]$

Architecture of the CBOW model The continuous bag of words model is based on a shallow, dense neural network with an input layer, a single hidden layer, and an output layer. The input to the model is a vector of context words, which we'll call X, and the output is the vector of the predicted center word $\hat{y}$.

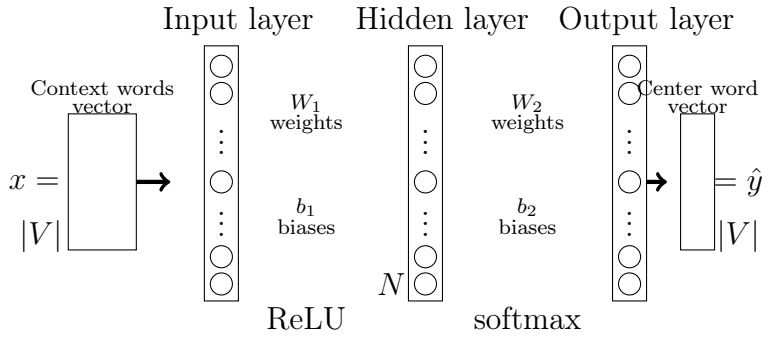


Figure 30: In this neural network, the size of the input and output vectors is the size of the vocabulary, $|V|$. If you recall from the previous section on word embedding process, the size or dimension of the word embeddings is a hyperparameter we select. This hidden layer is related to our embedding, and its size is a hyperparameter N; in practice N is in the range of hundreds to thousands. We're going to use a regular feed-forward neural network, i.e. a dense neural network with fully connected layers. Our algorithm will learn weights matrices W_i and bias terms b_i as we train it. For the input-to-hidden layer, we'll use a Rectified Linear Unit activation function, whereas for the output layer we'll use a softmax.

Dimensions of the CBOW model architecture Let's be clear on the dimensions of our vectors and matrices in our neural network diagram.

- Our input layer is of dimension $|V| \times 1$.
- To get the values in the hidden layer, we first calculate $z_1 = W_1 x + b_1$. Then, we compute the activation $h = \texttt{ReLU}(z_1)$. In terms of dimensions, W_1 is $N \times |V|$, where N is the size of the word embeddings. b_1 is of dimension $N \times 1$. The hidden layer is therefore of dimension $N \times 1$, since z_1 is an $N \times 1$ column vector and `ReLU` is applied element-wise.
- To get the values of the output layer, we need to calculate (i) the weighted sum from the hidden layer plus the bias term $z_2 = W_2 h + b_2$, and then apply the activation function $\hat{y} = \texttt{softmax}(z_2)$. Notice that W_2 is of dimension $|V| \times N$, and b_2 is of length $|V|$. When we compute z_2, it's a vector of length $|V|$, and when we pass it through the softmax we get back a vector of length $|V|$.

Vectorized implementation What if instead of passing a single training example through our neural network, we wanted to pass through several examples at a time. This makes the learning process quicker and is known as *batch processing*. Say you want to feed m input context word vectors into the neural network during each iteration; m is called the batch size and is a hyperparameter for the model that we define at training time. We can prepare a data matrix X where the m column vectors of length $|V|$ represent words: we get a data matrix of dimension $|V| \times m$.

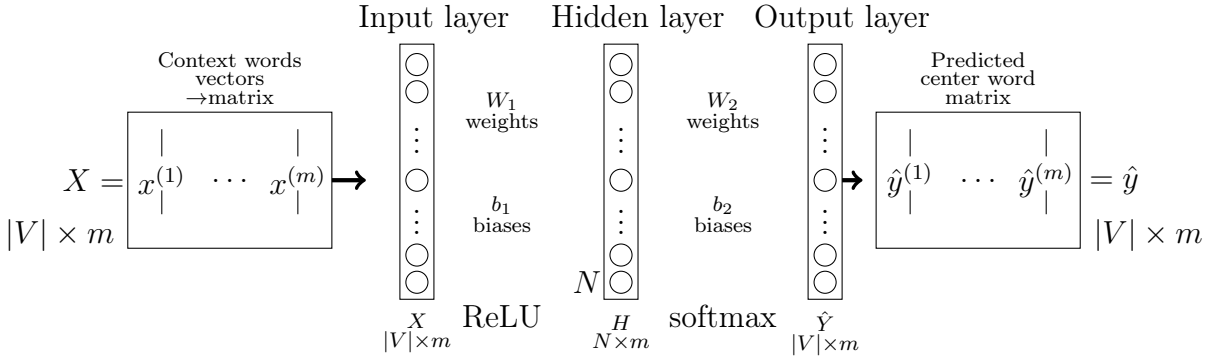


Figure 31: When batch processing data, our input becomes a matrix X where the columns are the representations for context words; the input matrix is of dimension $|V| \times m$, since we have m examples and we're using a one-hot encoding for our initial word representation. We then calculate $Z_1 = W_1 X + B_1$ where now B_1 is an $N \times m$ matrix of m *identical* b_1 bias vectors, each of length N; note that we can perform *broadcasting* in Python to add a vector to a matrix and the desired result is achieved. We continue with calculating elementwise $H = \texttt{ReLU}(Z_1)$. Next, we compute $\hat{Y}$, via $Z_2 = W_2 H + B$ followed by $\hat{Y} = \texttt{softmax}(Z_2)$. We finally see that $\hat{Y}$ is of dimension $|V| \times m$: i.e. we have m column vectors, where each one corresponds to a predicted center word. Essentially, the first column of the input matrix is transformed into the first column of the output matrix, and similarly for the remaining $m - 1$ input vectors.

Activation functions Let's first discuss `ReLU`, or rectified linear unit, then we'll get into `softmax`. Excepting input neurons, which are used "as-is", each layer first calculates the weighted sum of its inputs, adds a bias, and then passes the result into an activation function. I.e. for the first hidden layer we calculate $z_1 = W_1 x + b_1$, and to get h, we feed z_1 through the activation function $h = \texttt{ReLU}(z_1)$. ReLU is a popular general purpose activation function that only activates the neuron if the sum of its weighted input is positive. Mathematically, we have that $\texttt{ReLU}(x) = \max(0, x)$. We can plot the function:

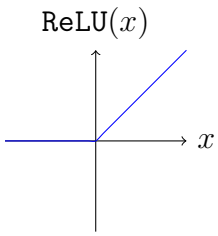


Figure 32: A plot of the ReLU function.

E.g., $\texttt{ReLU}(\begin{bmatrix}5.1 & -0.3 & \dots & -4.6 & 0.2\end{bmatrix} = \begin{bmatrix}5.1 & 0 & \dots & 0 & 0.2\end{bmatrix}$. Now, for the output layer we'll calculate the weighted sum of the values from the hidden layer and add the bias vector, i.e. $z = W_2 h + b_2$, and then we pass this through the activation function $\texttt{softmax}(z) = \hat{y}$. The *softmax* function takes in a vector $z \in \mathbb{R}^k$ and outputs a vector $[0, 1]^k$ where the result is a probabilitiy distribution, i.e. $\sum = 1$. When we apply softmax to z, the result is that we get a probability distribution over the words in the corpus.

z		$\hat{y}$	Probabilities of being center word
z_1		$\hat{y}_1$	a
$\vdots$		$\vdots$	$\vdots$
z_i	$\xrightarrow{\texttt{softmax}}$	$\hat{y}_i$	happy
$\vdots$		$\vdots$	$\vdots$
$z_{\lvert V\rvert}$		$\hat{y}_{\lvert V\rvert}$	zebra

Table 11: The softmax function takes in a vector of real numbers $z \in \mathbb{R}^k$, as opposed to `ReLU` which operates on scalars. The output of softmax is a vector of real numbers in the interval $[0, 1]$ which sum to 1 and can be interpreted as probabilities of exclusive events. In the case of a continuous bag of words model, when you apply softmax to z, you obtain an output vector $\hat{y}$ with $|V|$ rows, where each row corresponds to a word of the vocabulary of the corpus. The values of the vector can be interpreted as the probability that the center word which is what the model is trying to predict is the word that is assigned to each of the rows.

Mathematically,

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

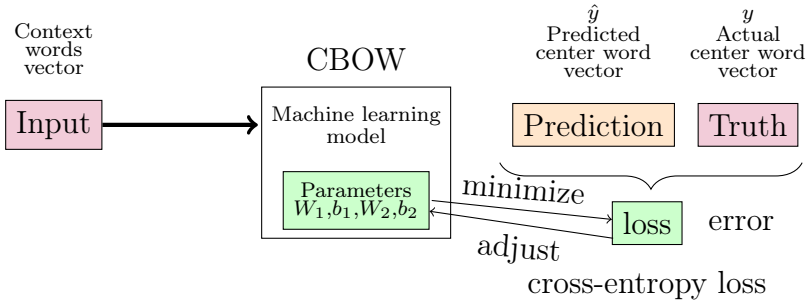
The exponential transforms all inputs to positive real numbers, and the denominator normalizes the value to lie in the unit interval.

Cost function for softmax In order to predict one of the $|V|$ possible words, we need to minimize a certain cost function. The formula for cross-entropy loss is given by

$$J = -\sum_{k=1}^{|V|} y_k \log \hat{y}_k.$$

Let's look at a quick example, the corpus "I am happy because I am learning" with context "I am because I" and target word "happy" (window is "I am happy because I").

More generally, $J = -\log \hat{y}_{\texttt{actual word}}$.



y		$\hat{y}$	$\log \hat{y}$	$y \odot \log \hat{y}$
0	am	0.083	-2.49	0
0	because	0.03	-3.49	0
1	**happy**	**0.611**	-0.49	-0.49
0	I	0.225	-1.49	0
0	learning	0.05	-2.49	0

Table 12: In this case, we predicted the correct target word and so the loss was $J = 0.49$.

8.3.1 Training a CBOW model

We've learned to use the cross-entropy loss function to measure the error made by a neural network of a continuous bag of words model when making a single prediction.

- We will first propagate the examples forward into the network.
- We'll then calculate the cost, which is an extension of the loss which extends over a *batch* (or multiple) examples.
- Finally, we'll use back-propagation and gradient-descent to optimize the parameters of the network and improve its predictive performance.

Forward propagation With forward propagation, we take our input values and pass them forward through the neural network from input to output through each successive layer, an dwe calculate the values of the layers as we do so. Starting with a batch of examples represented by X, we first propagate X forward to get $\hat{Y}$, where $X, \hat{Y}$ are $|V| \times m$. What are the formulae? *Loss* for a *single example* is given by $J = -\sum_{k=1}^{|V|} y_k \log \hat{y}_k$. When working with *batches of examples*, we'll calculate the *cost*, i.e. the mean of cross-entropy losses for each of the m examples. Formally, we can index over i from 1 to m, where i denotes the observation number, and j will range from 1 to $|V|$ over our vocabulary:

$$J_{\texttt{batch}} = \frac{-1}{m} \sum_{i=1}^{m} \sum_{j=1}^{|V|} y_j^{(i)} \log \hat{y}_j^{(i)} = \frac{-1}{m} \sum_{i=1}^{m} J^{(i)},$$

where $J^{(i)}$ is the loss for observation i.

Backward propagation and gradient descent How do we find the weights for our linear layer and our word embeddings? We have to minimize our cost, but how do we do this? We can use back propagation, which involves calculating the partial derivatives of the cost function with respect to the weights and biases, i.e.

$$\frac{\partial J_{\texttt{batch}}}{\partial W_1}, \quad \frac{\partial J_{\texttt{batch}}}{\partial W_2}, \quad \frac{\partial J_{\texttt{batch}}}{\partial b_1}, \quad \frac{\partial J_{\texttt{batch}}}{\partial b_2}.$$

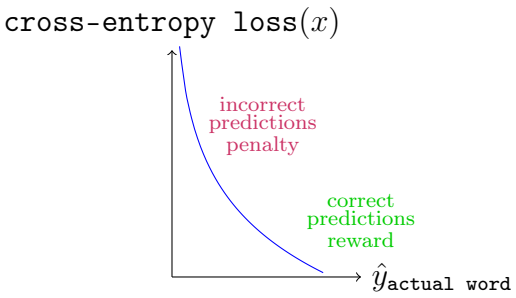


Figure 33: As you can see, in our simplified cost function, the model is penalizing incorrect predictions by incurring a high cost, and giving correct predictions a reward by incurring a lower cost.

Recall that $J_{\texttt{batch}}$ is a function of the weights and biases, i.e. $J_{\texttt{batch}} = f(W_1, W_2, b_1, b_2)$. The *back* in back-propagation comes from the way the formulae for the derivatives are obtained, using the chain rule for derivatives. At a rough level of detail, applying the chain rule involves starting from the output layer and working your way through the layers using the derivatives that you have calculated previously.[8]

$$\begin{aligned}
\frac{\partial J_{\texttt{batch}}}{\partial W_1} &= \frac{1}{m}\texttt{ReLU}\left(W_2^T\left(\hat{Y} - Y\right)\right)X^T \\
\frac{\partial J_{\texttt{batch}}}{\partial W_2} &= \frac{1}{m}\left(\hat{Y} - Y\right)H^T \\
\frac{\partial J_{\texttt{batch}}}{\partial b_1} &= \frac{1}{m}\texttt{ReLU}\left(W_2^T\left(\hat{Y} - Y\right)\right)\mathbb{1}_m^T \\
\frac{\partial J_{\texttt{batch}}}{\partial b_2} &= \frac{1}{m}\left(\hat{Y} - Y\right)\mathbb{1}_m^T,
\end{aligned}$$

where here $\mathbb{1}_m^T = \begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}$ of length m. Realize that for a matrix A, $A\mathbb{1}_m^T$ denotes the row-sums of A. In Python, we do this with something like

```
np.sum(A, axis = 1, keepdims = True)
```

The second technique we need is gradient descent, which adjusts the weights and biases of the neural network using the gradient to minimize the cost. We have a hyperparameter α which is the learning rate, some small strictly positive value usually less than one: a smaller α leads to more gradual updates to our weights matrices, whereas a larger number allows for a faster update of the weights.

$$W_1 \leftarrow W_1 - \alpha\frac{\partial J_{\texttt{batch}}}{\partial W_1}, \qquad W_2 \leftarrow 2_1 - \alpha\frac{\partial J_{\texttt{batch}}}{\partial W_2}, \qquad b_1 \leftarrow b_1 - \alpha\frac{\partial J_{\texttt{batch}}}{\partial b_1}, \qquad b_2 \leftarrow b_2 - \alpha\frac{\partial J_{\texttt{batch}}}{\partial b_2}$$

Extracting word embedding vectors After training our continuous bag of words model, we have to extract the word embeddings. Remember: these are byproducts of our learning algorithm not direct outputs. Also recall that word vectors carry the meaning of the vocabulary based on the context words in the corpus. There are three alternative word embeddings that we can extract from our model, actually.

1. Consider each column of W_1 as the column embedding vector of a word in the vocabulary. Recall that W_1 has $|V|$ number of columns, i.e. one for each word in the vocabulary. This means that we have an embedding for each word in our vocabulary. The mapping between columns of W_1 and words uses the same order as the input rows; in other words, if our corpus is "I am happy because I am

[8] Backpropagation is a great example of dynamic programming.

learning" if the five rows of the input vector correspond to $\begin{bmatrix}\text{am} & \text{because} & \text{happy} & \text{I} & \text{learning}\end{bmatrix}$, then in W_1 the first column vector will be the embedding for "am", and the second column vector will be the embedding for "because", etc.

2. We could also use each row of W_2 as the word embedding row vector for the corresponding word. Matrix W_2 has $|V|$ rows, one for each word in the vocabulary. Again, the order is the same as the input vector or matrix. E.g. with our example corpus and inputs, the first row would be the word embedding row vector for "am", the second for "because", and so on.

3. The third and final option is to take the average of the two previous representations: so, if you want the word embedding column vectors, you would average the corresponding embeddings from W_1 and the transpose of W_2 to obtain a new matrix W_3 of dimension $N \times |V|$. You can then extract the word embeddings from each column of W_3 just like we did previously.

8.4 Evaluating Word Embeddings

There are two evaluation metrics: intrinsic and extrinsic evaluation. Which one to use depends on the task you're optimizing for.

8.4.1 Intrinsic Evaluation

This method assesses how well the word embeddings inherently capture the semantic or syntactic relationships between the words. Semantics often refers to the meaning of words, whereas syntax refers to the grammar. You could do this by testing the word embeddings on analogies! You could test semantic analogies, as in: find the missing word in "France is to Paris as Italy is to __". You could also evaluate the embeddings on syntactic analogies, such as plurals, tenses, and comparatives. An example is: "seen is to saw as been is to?". One consideration you should be aware of for this approach, is that there could be several correct possible answers. As in, "wolf is to pack as bee is to __", where the collective noun and missing word could, for instance, be swarm *or* colony. As an example from the original Word2Vec research publication, when a continuous skip-gram model was used, the analogies obtained were not perfect: e.g. the embeddings completely failed to capture relationships between chemical elements and their symbols. The symbol for copper is `Cu`, the symbol for zinc is `Zn`, the symbol for gold is `Au`, and the symbol for Uranium is `U`, but the Word2Vec embedding outputs the word plutonium instead of `U` in the last case (getting the others correct).

We can also perform intrinsic evaluation by using clustering to group similar word embedding vectors, and determining if the clusters capture related words. This could be automated using a human made reference, such as a thesarus. Simply plotting or visualizing the word embedding vectors qualifies as a basic intrinsic evaluation of the vectors that relies on human judgement to assess the quality of the embeddings.

8.4.2 Extrinsic Evaluation

To evaluate word embeddings with extrinsic evaluation, we use the word embeddings to perform an external task: this is typically a real world task that you initially needed the embeddings for. Then, we use the performance metric of this task as a proxy for the quality of the embeddings. Examples of useful word level tasks included named entity recognition or parts of speech tagging. A named entity is something that can be referred to by a proper name. E.g. in the sence "Andrew works at deeplearning.ai", the word Andrew is a named entity and is categorized as a person. The word deeplearning.ai is another named entity, and is categorized as an organization. A useful task may be to train a model that can help you identify and categorize named entities in a sentence. You could then evaluate this classifier on the test set with some selected evaluation metric, such as accuracy or F1 score. The classifier's performance on the

evaluation metric represents the combined performance of both the embeddings and the classification task. Extrinsic evaluation methods are the ultimate test to ensure that the word embeddings are actually useful. However, their main drawbacks are that (i) the evaluation will naturally be more time-consuming than an intrinsic evaluation method and (ii) if the performance is poor, then the performance metrics provides no infromation as to *which* parts of the end-to-end process is resposnible (i.e. is it the word embeddings themselves or the external task used to test them?).

9 Neural Networks for Sentiment Analysis

This section is about creating neural networks using multiple layers, which is going to simplify the task of sentiment analysis considerably. A neural network is a computational structure that can approximate any function and learn to recognize patterns.

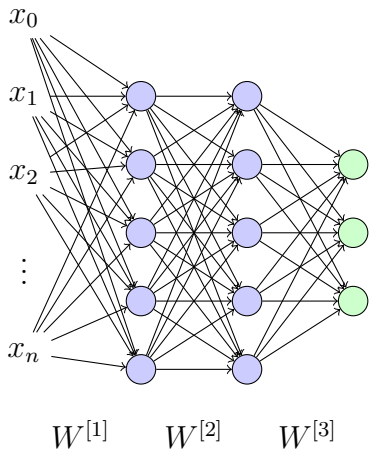


Figure 34: Four-layer, fully connected neural network with a three-dimensional output.

How does a neural network operate mathematically? We denote the i-th activation layer by $a^{[i]}$. First define $a^{[0]} = X$, i.e. define the input data to be the "first" activation layer. To get the values for each layer's activation $a^{[i]}$, you must first compute

$$z^{[i]} = W^{[i]} a^{[i-1]}$$

which depends on both the weights matrix from that layer and the activations from the previous layer. Finally, we obtain the activations for the i-th layer by applying an activation function g, to the values of z:

$$a^{[i]} = g^{[i]}(z^{[i]}).$$

Inherently, this computation moves "forward" through the computational graph, which is why the process is known as forward propagation. One particular neural network we could implement could revisit our problem of sentiment analysis, it will look like:

Initial representation We'll start by listing all of the unique words from the corpus: this is our vocabulary. Then, we'll assign an integer index to each word.

Then, for each word in your tweet, add the index from your vocabulary to construct a vector by replacing each word with its corresponding integer index. After you have all of the vector representations for your tweets, you'll need to find the one with maximal size, and fill (or pad) every smaller vector with zeros to match that size. I.e. we ensure that all our tweet representations have the same size even if our tweets don't.

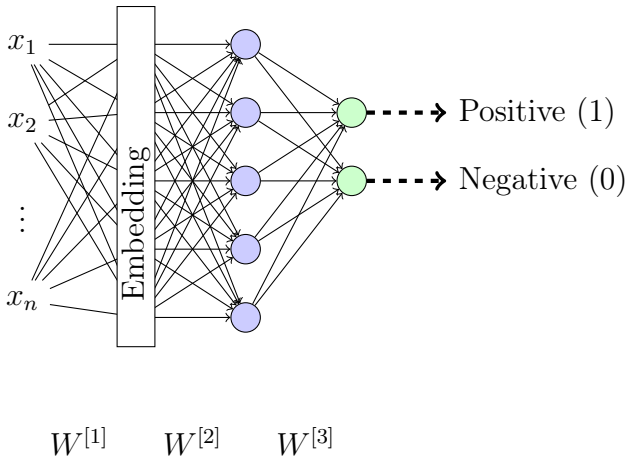


Figure 35: In this neural network, we use it to predict positive vs. negative sentiment for an input text of length n. There will be an embedding layer that will transform your representation into one that is optimal for the task at hand. Finally, it will have a hidden layer with a `ReLU` activation function and then an output layer with the softmax activation function that yields probabilities for whether a tweet has positive or negative sentiment. Such a neural network can allow us to predict the sentiment well even if the tweet is complicated, e.g. "This movie was almost good"; we would not have been able to classify this correctly with a more simple method like Naive Bayes because they miss important information in the context.

Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000

9.1 Neural Networks in Trax

Trax is built on top of Tensor Flow, which allows us to create a deep neural network in a sequential fashion. Suppose we have a network similar to what's depicted in figure 9, where this time the (two) intermediate hidden layers use sigmoid activation functions and the output layer will use a softmax activation. In Trax, you'll need to specify the model architecture; this is easily done:

```
from trax import layers as tl
Model = tl.Serial(
  tl.Dense(5),
  tl.Sigmoid(),
  tl.Dense(5),
  tl.Sigmoid(),
  tl.Dense(3),
  tl.Softmax()
)
```

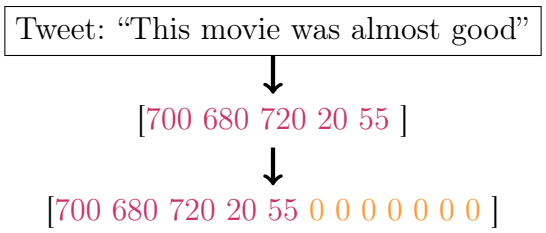


Figure 36: We first take a raw tweet and replace each word with its corresponding index-ID from the vocabulary to get an initial representation. We then pad the vector with zeros according to our longest vector representation.

Observe that specifying the architecture in this way mimicks the order in which the computations are carried out in the neural network. When using Trax, it's possible to take advantage of parallel (or GPU/TPU) compute resources. The framework also records algebraic computations for gradient evaluation, which can be useful in diagnosing models. Trax is one of the latest frameworks available.

- Trax documentation.
- Trax source code.

Trax layers A *serial* model is cmoposed of a series of layers, which is the basic class in Trax. We now forray into how classes work and their methods. In Python, a class is a way to define common properties and methods for similar objects; i.e. classes define common variables and behavior for the object associated with them. E.g., let's say that we have different colors, so we create a `color` class. Perhaps our class has common data attributes such as `r, g, b` for every item defined the class, and some methods associated with it.

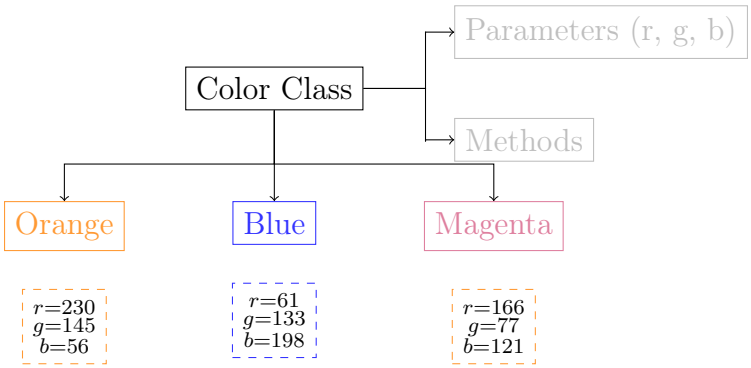


Figure 37: Each of the instances from your class in this case are the colors orange, blue, and magenta. They each possess a specific set of RGB values.

How do we define a class in Python?

```
class MyClass(Object):
  def __init__(self, y):
    self.y = y
  def my_method(self, x):
    return x + self.y
  def __cal__(self, x):
    return self.my_method(x)
```

The name of our class is `MyClass` and it inherits from a base Python `Object`. The `__init__()` method is called when you initialize any instance of the class and assign values to its data attributes. We can also define custom methods to do whatever we'd like, e.g. we define `my_method` to add whatever value is stored in the y attribute to the input argument x. Lastly, the `__call__()` method is used when you call an already initialized instance.

Subclasses We can define subclasses, or classes who derive/inherit from a parent class. E.g.

```
class SubClass(MyClass):
  def my_method(self, x):
    return x + self.y**2
```

Any method that is defined in a subclass will override corresponding methods in the parent class. So, if you redfine `my_method`, the `__init()__` and `__call__` methods are inherited from `MyClass` but the `my_method` sub-routine is replaced.

Dense and `ReLU` layers Basically,

$$\texttt{Dense Layer} \longrightarrow z^{[i]} = W^{[i]}a^{[i-1]}$$
$$\texttt{ReLU Layer} \longrightarrow g(z^{[i]}) = \max(0, z^{[i]}),$$

where $W^{[i]}$ denotes trainable parameters.

Serial layers Dense and ReLU layers perform single steps as part of forward computation. We can define a serial neural network as a composition of layers that operates in a sequence. Imagine a basic neural network as in figure 9: there are some dense and some activation layers, and the sequential arrangement of these calculations can be done in Trax. The new serial layer is composed of the dense and activation layers, and we can think of our entire neural network as being held in this one layer.

Other layers We've previously worked with a few different ways to represent text data. WHen using neural networs for NLP tasks, *embedding layers* are often included in the model. To this end, we also are then required to introduce a *mean layer*. In NLP, we typically have a set of unique words called the vocabulary V. An embedding layer takes an index assigned to each word in the vocabulary and maps it to a representation of the word with a prespecified dimension.

Vocabulary	Index		
I	1	0.020	0.006
am	2	-0.003	0.010
happy	3	0.009	0.010
because	4	-0.011	-0.018
learning	5	-0.040	-0.047
NLP	6	-0.009	0.050
sad	7	-0.044	0.001
not	8	0.011	-0.022

Table 13: In this example, the embedding size is two. The embeddings are actually trainable weights in a model, i.e. we want to learn a matrix of dimension $|V| \times$ `embedding_size`.

There's also a *mean layer*, which simply means taking elementwise mean across a set of embeddings. This mean-layer does not have any trainable parameters, because it simply computes a mean.

Training Deep learning frameworks facilitate the automatic computation of gradients; this is much simpler than back-propagating by hand. Suppose

$$f(x) = 3x^2 + x \implies \frac{\partial f(x)}{\partial x} = 6x + 1.$$

The gradient is just a derivative when we're dealing with a scalar valued function. In Trax:

```
def f(x):
  return 3*x**2 + x

grad_f = trax.math.grad(f)
```

The function `grad` returns a *function* that evaluates the gradient of f at a point. Using the `grad()` function to train a model is painless:

```
y = model(x)
grads = grad(y.forward)(y.weights, x)
```

That is, we first apply the gradient function with the forward method of our model as a single parameter. Then, we evaluate the gradient with our weights and inputs. After obtaining the gradients for a model, we simply iterate until convergence is reached.

```
# While convergence not yet reached:
weights -= alpha*grads
```

This means that we're basically performing forward and backward propagation in a single line.

10 Recurrent Neural Networks for Language Modeling

Although we've previously learned how to use N-gram language models to compute the probability of a sequence of words, we'll learn now how the aforementioned method requires a lot of computational storage requirements, and how to perform the task more efficiently.

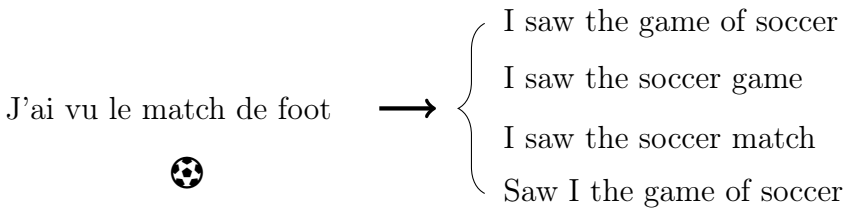


Figure 38: Suppose we have a French-English translation task, where we've procured several candidate translations. If we wanted an accurate translation, we could compute the probabilities of each sentence using a model like an N-gram, and select the sentence with highest probability. In this case that would be "I saw the soccer game".

Recall that in order to build an N-gram language model, we have to compute conditional probabilities, which for bigrams are of the form $\Pr(w_2|w_1) = \frac{C(w_1 w_2)}{C(w_1)}$, and at the end, to get the probability of a sentence, we multiply out the conditional probabilities of each word in the sentence using its previous $(N-1)$ words; e.g. $\Pr(w_1, w_2, w_3) = \Pr(w_1) \times \Pr(w_2|w_1) \times \Pr(w_3|w_2)$. For an N-gram model we use conditional probabilities based on the previous $N-1$ words, and the notion generalizes. There are downsides to this approach:

- Realize that in order to capture dependencies between distant words, we require a large N in our N-gram model in order to model the conditional probabilities of very long sentences.
- Even in the case of a large corpora, your model would need a lot of space and RAM to store all possible combinations in a transition probabilities table.

Recurrent neural networks and gated recurrent units are two models that are much more efficient than N-grams for NLP tasks like machine translation.

10.1 Recurrent Neural Networks

Recurrent Neural Networks (or RNNs) allow us to capture some dependencies that we could not have otherwise captured with traditional N-gram language models. Let's look at an example:

`Nour was supposed to study with me. I called her, but she did not _____.`

If we used a traditional language model, such as a trigram, to try and complete this sentence, you would compare different word probabilities and select the word with the highest chance of following the words "did not". For a typical corpus, a trigram model might select "have" as the successor word, but in this context that choice doesn't make any sense![9] As a better alternative, RNNs aren't limited to looking at just the previous N words; instead, they *propagate information* from the beginning of a sentence to the end. I.e. RNNs implicitly look at *every* previous word rather than the last $N-1$, and so we might end up corrrectly predicting "answer" in our example above. If we wanted an N-gram model capable of capturing this relationship, we'd have to have account for seven word long sentences which becomes impractical.

How does an RNN work? To see how an RNN works, let's look at just the second sentence from our previous example, "I called her but she did not ___".

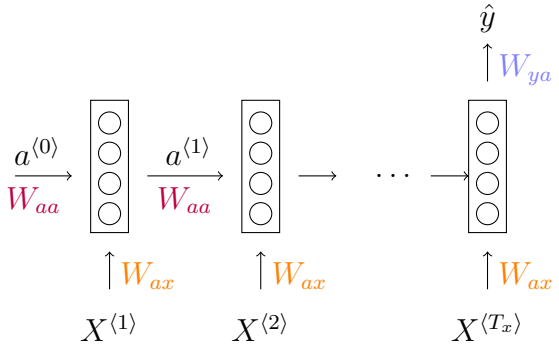


Figure 39: When a recurrent neural network makes a prediction for the second element in the input sequence $X^{\langle 2 \rangle}$, it also gets to input some information from what was computed from time step one. In general, the activation from the previous step is carried forward. To kick things off, we'll also have some made up activation at time step zero, e.g. we might choose to initialize $a^{\langle 0 \rangle}$ with a vector of all zeros. For each input element in the sequence, we use shared weight parameters W_{ax}, and to govern how we pass forward activations we used shared weight parameters W_{aa}. Lastly, there is a set of weights W_{ya} that govern how to map from hidden neurons to an output prediction, which in our above example would be the last word of the sentence.

Part of the "magic" of RNNs comes from the fact that every word in the sequence is multiplied by the wame weights matrix W_{ax}, and information is propagated from beginning to end via a different set of shared weights matrices W_{aa}; these weights are learnable parameters, alongside weights W_{ya}. Realize that the number of parameters for an RNN remains fixed regardless of the input sequence length.

[9] Other candidate words for a reasonable trigram model would still not make sense and include words from a set like, {want, respond, choose, have, ask, attempt, answer, know}.

Applications of RNNs There are many types of architectures for RNNs which can be grouped according to their inputs and outputs.

- **One to One:** input is a set of low or non-correlated features x and output is a single value y. This isn't much different from a conventional neural network.
- **One to Many:** e.g. the input is a single image and you'd like the output to be a caption describing that image. We take in a single image and we output multiple words.
- **Many to One:** e.g. sentiment analysis. You may have an input sequence for which you'd like a single output.
- **Many to Many:** involve multiple inputs and multiple outputs, e.g. machine translation. The architecture is known as encoder-decoder. The first part of the neural network is called an encoder because it simple encodes a sequence of words into a single representation that captures the overall meaning of the sentence; it doesn't return any output $\hat{y}$ until the decoder stage.

Math in simple RNNs Let's look at a vanilla RNN and basic recurrent units, with a many-to-many architecture.

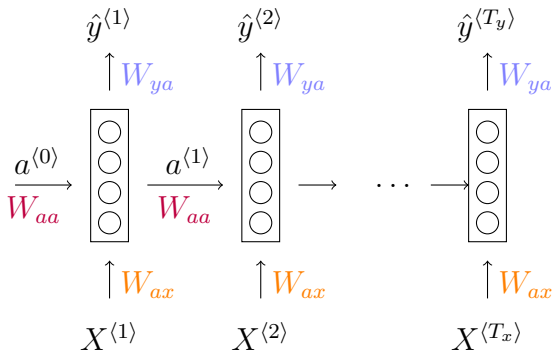


Figure 40: At each time-step, we take in an input X^{T_i}, a hidden state h, and we make a prediction $\hat{y}^{T_i}$. Additionally, we propagate a new hidden state to the next time step. The hidden state at each time step is computed with an activation function g with arguments equalt to the product between a parameter matrix W_{aa} and the previous hidden state $a^{\langle T_{i-1} \rangle}$.

The complete equation for our many-to-many architecture looks like:

$$a^{\langle t \rangle} = g\left(W_{aa}a^{\langle t-1 \rangle} \oplus W_{ax}x^{\langle t \rangle} + b_a\right)$$

where $\oplus$ denotes element-wise addition. After computing the hidden state at time t, we can feed it through an activation function to compute the output

$$\hat{y}^{\langle t \rangle} = g\left(W_{ya}a^{\langle t \rangle} + b_y\right).$$

11 LSTMs and Named Entity Recognition

12 Siamese Networks