Elementary notes on NLP

Haosheng Zhou

August, 2022

This note mainly refers to the materials from DeepLearning.AI and is only used for educational purpose.

Sentiment Analysis

The task of sentiment analysis is to predict the sentiment of a certain sentence based on the information contained in a labelled training set. Let's see the standard steps below for conducting sentiment analysis.

Preprocessing

Remove indicating words from the sentences, like "@". Remove hyperlinks and hashtags by using regular expressions.

Tokenize the words in each sentence, remove blanks and tabs, convert to lower cases.

Remove stop words (I/am/at...) with no exact meanings and stem all the words so that happy/happier/happiest etc. are recognized as the same word.

The preprocessing step is very important and cannot be omitted, it's the fundemental step of all following methods.

Build Frequency Dictionary

Now assume we have got a list of cleaned words (preprocessed) for each sentence, we want to look through each sentence to find out the frequency each word has appeared. This provides direct connection with the sentiment of the sentence.

We set up a dictionary, with the key as a tuple $(word, label)$, here $word$ is just the cleaned single word and the label is in $\{0, 1\}$, indicating the appearance of this word happens in a sentence with negative/positive sentiment. The value correspondent to this key is just the overall times this word has appeared in the training set under a neg/pos sentiment.

Feature Extraction

After building up the dictionary, the first easy way we can think about to train the model is to make up a very large matrix as the training data. To illustrate this point, let's assume that there's N sentences in the training set and M unique words in the dictionary. We set up a $N \times M$ matrix such that the entry on row i and column j stands for the appearance of the j^{th} word in the i^{th} sentence, taking value 0 or 1. However, this matrix is so sparse and large so that model training takes very long time and the model won't be well trained base on this.

Another approach is to suppress this huge data matrix into features. We define 2 features for each sentence, the sum of the positive frequency of all unique words appearing in this sentence, i.e. $\sum_{w\in sentence} dict[(w, 1)]$, and the sum of the negative frequency of all unique words appearing in this sentence, i.e. $\sum_{w\in sentence} dict[(w, 0)]$. This is reasonable for the following reasons: (i): The larger the positive feature is, the more likely the sentence has positive sentiment, vice versa, so these features exhibits the bias of sentiment. (ii): Now instead of having a $N \times M$ matrix, we have a $N \times 2$ matrix, which is much smaller and is also dense. As a result, the time to train the model would be much less and the model would be expected to be trained well. The condensation of word counts into features is an important thought adopted.

NOTE: For such feature extraction, it's actually a loss of information because the model won't be able to dinstinguish "happy" from "not happy" by simply adding up counts. However, maybe we can actually make use of these information and its binary structure by applying biclustering algorithms like loop-counting? Maybe that will be interesting...

Logistic Regression

After these first steps, the only thing we have to do now is to apply different classifiers to the matrix of two features. The first method to try is just the logistic regression to do binary classifications.

Since the model of logistic regression is just $y = \sigma(w^T x)$ with $\sigma(y) = \frac{1}{1+e^{-y}}$ as the logistic function. This model can be trained using various methods like gradient descend, Newton's method, coordinate descent etc. However, we have to be careful that since there's an intercept term in the logistic regression, we have to put into an $N \times 3$ matrix, with the first column as constant 1. There's not much to talk about the training and prediction of logistic regression so I will just skip it.

Naive Bayes

Let's now build up the naive bayes classifier, and we will see what "naive" means in this context.

Actually, it's just a simple application of the Bayes Rule. Given a sentence, we wish to know whether its sentiment is positive or negative, that is to say, we want to know which one among $\mathbb{P}(pos|w_1, ..., w_n), \mathbb{P}(neg|w_1, ..., w_n)$ is larger, here $w_1, ..., w_n$ denotes all unique cleaned words appearing in this sentence (from the preprocessed step we have just done).

Apply Bayes Rule to find out:

$$\mathbb{P}(pos|w_1, ..., w_n) = \frac{\mathbb{P}(w_1, ..., w_n|pos)\,\mathbb{P}(pos)}{\mathbb{P}(w_1, ..., w_n)} \tag{1}$$

$$\mathbb{P}(neg|w_1, ..., w_n) = \frac{\mathbb{P}(w_1, ..., w_n|neg)\,\mathbb{P}(neg)}{\mathbb{P}(w_1, ..., w_n)} \tag{2}$$

Here $\mathbb{P}(pos), \mathbb{P}(neg)$ are just the frequency of the sentences with positive and negative sentiments in the training set. Notice that since these to expressions have the same denominator, we just have to calculate the numerators and compare!

For the calculation of $\mathbb{P}(w_1, ..., w_n|pos)$ (same work for the negative case), we now have to use the **Naive** condition, which means that given the sentiment of the sentence, all unique words are independent. This is a conditional independence statement, although this rarely holds in practice, e.g. given the sentiment of the sentence to be positive if "not" appears then another word with negative sentiment is more likely to appear, it's a reasonable assumption to make for the time being. Immediately we get:

$$\mathbb{P}\left(w_1, ..., w_n | pos\right) = \mathbb{P}\left(w_1 | pos\right) ... \mathbb{P}\left(w_n | pos\right) \tag{3}$$

Now to compute each of these probability, it's an easy job! Since we already have the frequency dictionary built, we just have to form it empirically:

$$\mathbb{P}\left(w | pos\right) = \frac{dict[(w,1)]}{\sum_{v \in dict} dict[(v,1)]} \tag{4}$$

It seems that we are at the end of this topic, but there are still some points we would like to be careful with. Firstly, we don't want to see any zero probability in the calculations because this is too extreme! For example, if a word "angry" appears in the sentence, and there is no negative appearance but there is positive appearance of "angry" in the training set, then we would say $\mathbb{P}\left(\text{"}angry\text{"} | neg\right) = 0$, thus $\mathbb{P}\left(neg | sentence\right) = 0$, which is obviously not so reasonable. The point here is that we would like to give tolerance to words that have never appeared in one of the positive or negative categories and we won't allow such word to make an extreme impact on the probability of the whole sentence being positive or negative. The strategy is simple, called **Laplacian Smoothing**. We modify the calculation of $\mathbb{P}\left(w | pos\right)$ such that now it is formed as:

$$\mathbb{P}\left(w | pos\right) = \frac{dict[(w,1)] + 1}{\sum_{v \in dict} dict[(v,1)] + \sum_{v \in dict} 1} \tag{5}$$

By adding 1 to numerator, this probability can never be 0 now By adding the number of all unique words in the dictionary to denominator, we still keep the probabilities adding up to 1.

Secondly, we would like to notice that since probabilities can never be greater than 1, the product of many probabilities would always be very close to 0, which may cause underflow for the float point calculation. The trick is just to use log probabilities and turn the product into sum to avoid such problems.

Originally, to judge the sentiment of a sentence, we only have to compute $\frac{\prod_w \mathbb{P}(w|pos)}{\prod_w \mathbb{P}(w|neg)} \frac{\mathbb{P}(pos)}{\mathbb{P}(neg)}$ and compare that with 1 to see whether we should predict it to be positive or negative. To take log of this, we just have to compute the λ value for each word w:

$$\lambda(w) = \log \frac{\mathbb{P}\left(w | pos\right)}{\mathbb{P}\left(w | neg\right)} \tag{6}$$

For any sentence, we just have to figure out all the words w_i that appear (unlike the case for the dictionary, w_i can have repeated words, which is natural) and compute:

$$\sum_{i=1}^{n} \lambda(w_i) + \log \frac{\mathbb{P}\left(pos\right)}{\mathbb{P}\left(neg\right)} \tag{7}$$

Here the first term is called log-likelihood and the second term is called log-prior. By comparing this sum with 0 (note that we have taken log, so it should not be 1), we can make predictions on the sentiment.

However, what shall we do if we meet with words in the test set that has never appeared in the training set (of course, such word can't appear in the dictionary)? We can just ignore it since we have no knowledge whether it's a positive or negative word, which means that the associated λ value of this word is just taken as 0, no contribution to both sides.

Vector Space Model

Embedding

The first problem we would have to face is that how we should transform words into their vector representations (embeddings). There are two main ways to do this, called word-by-word transformation and word-by-document transformation.

For word-by-word style, we always define the concept of proximity, for example, if a word appears within distance $k = 2$ of the given word, we would call it a proximal word. Then, the word-by-word vector representation is just the count of all proximal words that we care about. For example, we have got a sentence *"I love to learn math because learning is interesting."*, let's set $k = 2$ and consider the vector representation of the word *math* with respect to words *love, learn, because, interesting*. It's obvious to see that the vector representation should be $[0, 2, 1, 0]$ since *learning* after stemming is the same as *learn*.

For word-by-doc style, we always have multiple corpus of different categories instead of a single corpus. And the construction is also a little bit different. For example, we have 3 styles of corpus A, B, C and we want to get the vector representation of the word *math* in these 3 corpus. We just count the appearances of this word in the corpus with each of the three categories, if it appears in A corpus for 100 times, B corpus for 1000 times and C corpus for 50 times, then the vector representation is just $[100, 1000, 50]$.

To sum up, when we want to discover the difference of appearance of words in a same context, we use word-by-word construction. If we want to discover the connection between the type of the corpus and the appearance of words, we use word-by-doc construction.

Concept Formation

Assume that we have already known the vector representations of all words, how can we make use of these data to fulfill NLP tasks?

One important thing to do is called concept formation, where we truly "define" a concept to play with. Definition can take place in various forms, and the simplest form to do the concept formation is to compute the **difference of vectors**.

To illustrate this point, we assume that we have got a corpus whose content is about cities and countries, and we believe that within such corpus the appearance of a country and its capital city has some kind of correspondence. **NOTE:** this is a crucial hidden assumption to make! If there's no underlying relationship between country and capital within the corpus, it would be absurd to predict the country-capital correspondence based on this corpus! As a result, we would likely conclude that there is a concept called "is capital", and it's formed as

$$vec(country) - vec(capital) \tag{8}$$

The specific meaning is that: we expect each country-capital tuple to have the similar difference between their vector representations. If we could accept such logic, then equipped with very little **prior knowledge**, e.g. Beijing is the capital of China, we can generalize it to all countries appearing in the corpus, i.e. we can fulfill the task to predict the country whose capital city is given.

Our simple strategy would be: first calculate

$$vec(country_1) - vec(city_1) + vec(city_2) \tag{9}$$

We would expect such expression to be the vector representation of $country_2$, but reality is rarely so idealistic. So instead of finding a word in our dictionary that has exactly such vector representation, we naturally find a word whose vector representation has the most similarity to the one given. Thus, as soon as we decide what similarity measure to choose, we are done with this prediction task.

Similarity Measures

To measure the similarity between two vectors, the most common strategy would be to measure the distance or angle formed by two vectors.

By calculating the Euclidean distance between the endpoints of two vectors and adding a negative sign, a similarity measure appears. However, such similarity measure is not general since it has not taken the difference in the modulus of two vectors into account. For example, *math* has vector representation $[100, 200, 300]$ in corpus A, B, C respectively, *difficult* has $[50, 100, 150]$ and *exercise* has $[200, 400, 600]$. We see that, on average, each appearance of *math* brings with 2 appearances of *exercise* and 0.5 appearance of *difficult*. By computing the Euclidean distance, however, we would conclude that *math* would be more similar to *difficult* than *exercise*, which is not the expected conclusion since we would intuitively think that any two of those three words have equally close relationship.

The solution is simple. Instead of using distance, why not use the acute angle formed by those two vectors? That gives birth to the cosine similarity defined as:

$$cos\theta = \frac{\alpha \cdot \beta}{||\alpha||_2 \cdot ||\beta||_2} \tag{10}$$

The larger such cosine score is, the smaller the angle, the more similar two vectors are. Such similarity score does not take the difference of vector modulus into consideration, thus being more robust.

PCA

To visualize the word clusters (words that share common characteristics in their vector representations), we would need to make use of dimension-reduction skills, since word vectors always have a large size and we are not able to plot anything over 3 dimensions.

Now that we are applying vector models, it's natural to think of the possible tools we are able to use in linear algebra. Principal Component Analysis consists of following stages: (i): Demean the matrix (ii): Compute covariance/correlation matrix (iii): Take orthogonal eigen-decomposition of cov/corr matrix (iv): Only keep the largest n eigenvalues and correspondent eigenvectors (v): Compute the reduced version of the original data using these eigenvectors.

To emphasize, we have to **demean** the matrix, i.e. extract the mean of each column from itself (here each row is a word vector). Lacking in this step would cause huge problems, since what PCA is actually doing is to find some new coordinate axes that well fits the original data. Imagine all data points are uniformly distributed inside an ellipse centered at origin, we would immediately recognize that the new coordinate axes should just be the direction of the long and short axes of the ellipse. However, if such an ellipse is centered at a point in the first quadrant that's very far away from the origin (case without demean), we would conclude in mistake that the new coordinate axes should always have direction towards the first quadrant.

Other details of PCA are omitted since it's a standard technique in statistics. We just want to state the value and importance of PCA in visualizing data and the structures contained.

Word Translation

Suppose we are know given a task to translate words from language A to language B, and we are given a training set where we know some of the corresponding words in two languages.

To use the vector model to solve this problem, notice that each word in its language has an embedding (vec rep), thus by organizing all the words appearing in the training set, we would have got two matrices, X and Y. Here X is the embedding matrix in language A, Y for language B, each row of X, Y stores the embedding of a word having the same meaning in two languages respectively. Naturally, translation between languages means connecting the embeddings of a word with the same meaning in different languages, i.e. we would like to find a matrix R such that $XR = Y$. Note that this expression has the meaning that the product of the i-th row of X and R is just the i-th row of Y, so R just has the concept of transition matrix between two bases of a finite-dimensional vector space.

Unfortunately, such R rarely exists since the column space of X is not necessarily a subset of the column space of Y, since we only know that X, Y has the same size $m \times n$ where m is the number of words that have embedding in the training set and n is the dimension of the embedding of a single word. As a result, we would apply relaxation, to find instead R such that it minimizes the loss:

$$l(R) = \frac{1}{m}||XR - Y||_F^2 \tag{11}$$

Here we take the square of Frobenius norm since it has a simple form of matrix derivative and $\frac{1}{m}$ is introduced as an average among all words. Let's compute the gradient of the loss:

$$\nabla l(R) = \frac{2}{m}X^T(XR - Y) \tag{12}$$

By applying gradient descent, we can update such R iteratively to get the optimal value. In brief, the training process of the word translation task is just equivalent to finding the optimal R using the word correspondence in two languages from the training set.

For the purpose of prediction, given a word embedding v in language A, compute vR as the word embedding in language B. Find the word in language B from the training set whose embedding has the most similarity to vR as the translation result. (e.g. cosine similarity)

Locality Sensitive Hashing (LSH)

Note that all steps above makes sense for word translation tasks, except the last point where we have to search through all the words for the one with the most similarity to the given embedding. This operation seems reasonable, but it's actually the most time-consuming part in practice. Let's first estimate the time complexity of this operation. Assume that we have m words in the training set with embedding, and each embedding is a vector with dimension n. For each word, we have to compute the cosine-similarity with the given vector v, which takes $O(n)$ time. So altogether we will have to spend $O(mn)$ time just to find the most similar single word. This time cost is not affordable under circumstances where m, n are both very large, i.e. a large corpus and a large embedding vector, which often happens.

That's why we have to think of a method to realize a trade-off between accuracy and efficiency to reduce the time cost of this step. Before that, let us consider a slightly more general question, which is to get the k nearest neighbors instead of the nearest neighbor (special case for $k = 1$). We would always assume that $k = O(1)$ so the time complexity of the strategy above won't change, but this just provides us with a more general approach that fits well in many situations. This problem is called kNN (find k nearest neighbors for a given vector under a certain similarity measure), note that **this is different from the k-means clustering problem**.

Now we put forward the locality sensitive hashing technique. The thought comes from a natural questioning on the concept of "neighborhood". If I have tolerance for error but want to reduce time complexity, why don't I first restrict myself to the embeddings that I know are not too far away from v and then only do calculations to search within those embeddings? To explain it in a more intuitive way, if I am now living in the U.S. and I want to visit some of my friends who are living near me, the first thing I would like to do is to look through all my friends living in the U.S. instead of looking through those who are living in China. By doing this, I don't have to calculate the distance from my location to the locations of my friends who are living in China any more, and it saves time. However, by restricting my options to only the friends living in the U.S., I might not get the truly optimal choice. For example, New York is much nearer Toronto than San Francisco, I might have friends in Canada living nearer than all my other friends, but by restricting myself to the U.S., I eliminate those better options. That illustrates the so-called accuracy-efficiency trade-off we are facing.

After getting this big idea, we can talk about how to implement this. The first problem is: how shall we know whether we are "not too far away from v". The answer is: **randomness**. Note that we want to judge whether we are "not too far away from v" without calculating any similarity (otherwise we return to the trivial implementation we talked about above). It might sound paradoxical, but we now have tolerance towards error, so "assuming" that certain embeddings are close to v could be reasonable.

The second problem is: how to make use of such randomness? We won't want to apply the randomness arbitrarily since that may lead to terrible performance. For example, if we randomly pick up ten friends and just assume that they are close to New York, we might get all of those living in Shanghai, a terrible choice! The strategy is to make use of **hyperplanes** in the vector space. We know that a hyperplane partitions the space into two disjoint parts. The part that includes v is naturally specified as the part near v, the part that does not include v is naturally specified as the part far away from v. Also, hyperplanes that go through the origin has very simple representation since they are uniquely determined by their normal vectors, so we only have to keep record of their normal vectors!

Combining the analysis we have conducted in the two paragraphs above, we should introduce random hyperplanes to divide the whole vector space into disjoint parts. In practice, we only need to generate and maintain random vectors as normal vectors for these hyperplanes. Then comes the third question: for each hyperplane, how do we know which of the two parts include v? This is simple because we just have to compute the inner product $v \cdot n$ between given embedding v and the normal vector of the hyperplane n. If the inner product is negative, then v is on the opposite side of the hyperplane from n, vice versa.

Now that we are able to tell which part of the vector space includes v, we finally introduce the data structure: **hash table**. Hash table has a number of buckets and stores objects with common characteristics in the same bucket. The "common characteristics" is defined by hash values, which is a function of the objects stored. When we want to hash a list of objects, we always compute a hash value for each object to decide which bucket it's going to, and then put the object into that bucket. In our application, it's clear that the buckets should be different parts of the vector space divided by the random hyperplanes and the objects should be word embeddings, but what are the hash values?

For example, we now have 2 random hyperplanes diving the vector space into 4 parts. Hash values should reflect an attribute of the difference that we care about! Here, we only care whether v is on the same or opposite side as the hyperplane's normal vector n w.r.t this hyperplane, which is characterized by the sign of the inner product $v \cdot n$. So a smart choice would be: assign a hyperplane with 1 if $v \cdot n \geq 0$, assign a hyperplane with 0 otherwise, as an indicator of v's location. Then, each hyperplane receives an indicator 0 or 1, and we can make use of this 0-1 string to construct hash values. Assume there are H hyperplanes, and the indicator of the i-th hyperplane is $H_i \in \{0, 1\}$. This binary structure immediately reminds us of the binary representation of integers, so it's natural to write down:

$$hash\ value = \sum_{i=0}^{H} 2^i \cdot H_i \tag{13}$$

Now, each disjoint part in the vector space gets its distinct hash value, and we can build a hash table and hash all the words in the training set so that different words would go into corresponding hash buckets. The only bucket we are interested in is just the bucket where v is located in (this bucket is the region where we "assume" that all embeddings within are close to v), so we just have to compute the hash value of v, take all words in this bucket and now we only have to compute similarities between v and the embeddings in this bucket to pick out the k nearest neighbors.

At last, we want to mention that this strategy is still not complete since only doing LSH for one time sufferes from large bias in results! To increase accuracy, our strategies are: (i): do not set up too many hyperplanes, since there will be 2^H hash buckets for H hyperplanes, large H reduces accuracy and increases space complexity (ii): do LSH repeatedly in different universe (different set of random hyperplanes), take the union of the words whose embeddings are located in the same hash bucket as v in each universe and do calculations for all of them at the very end (so there's no repeated calculations).

To conclude, we write out the whole process of LSH below:

- Create hash value function, the function computes the inner product between a given vector and the normal vectors of all hyperplanes, create a 0-1 indicator on the signs of the inner products, and take it as binary representation of an integer, which is the hash value of this given vector.
- For each universe (altogether U universes), randomly generate the normal vectors for all hyperplanes going through the origin (altogether H hyperplanes) and hash all word embeddings in the training set. Since we want to find the most similar embeddings to the given vector v, compute v's hash value and only keep all embeddings in that hash bucket, record those embeddings for further consideration.
- Take the union of all recorded embeddings across different universes. Calculate the similarity measure between v and each of those embeddings, take the k nearest neighbors.

LSH Analysis

How well does LSH work? Is it more efficient, is there a high possibility that LSH gives a very bad result? Let's do some analysis on this model.

First, compute time complexity. We have U universes. Fix a universe, computing a single hash value costs $O(mH)$, so hashing all the data in a single universe costs $O(mnH)$. Note that these hash tables can be used repeatedly for queries for different v, and we have to build them up once for permanent use. After hashing data for all universes, we get at most $X_1 + ... + X_U$ embeddings for calculations, with X_i *i.i.d.* stands for the number of vectors in the hash bucket where v is in. It's obvious to know that $\mathbb{E}X_i = \frac{n}{2^H}$ assuming the embeddings' hash values are uniformly distributed. Thus, a single query costs in average $O(\frac{U}{2^H}mn)$. Now for efficiency considerations, we always want to make sure that there are in average $O(1)$ embeddings in each hash bucket, so $H = O(\log n)$.

To conclude, we list the results as follows:

- The brutal kNN search costs $O(mn)$ for each query, no preprocessing is needed.
- LSH costs $O(Umn \log n)$ for preprocessing, but for each query it costs $O(Um)$.

Note that even if U is taken as $\log n$, the query speed of LSH is still much faster than that of the brutal kNN! Here, our assumption is way too strong, we even assume that the embeddings for consideration from different universes are disjoint! In practice, this rarely happens, so the time complexity of the LSH query can only go lower!!!

We then consider the case where LSH cannot give us the accurate result. We consider the case where $k = 1$. Denote p_{error} as the probability LSH cannot give us the accurate answer after running for all universes, denote p_e as the probability LSH cannot give us the accurate answer after running for a single universe. It's obvious that by independence of universe:

$$p_{error} = (p_e)^U \tag{14}$$

To get an upper bound on p_e, note that if such event happens, the nearest neighbor v_{nn} shall not be in the same hash bucket as v, which means that they have different hash values. So there exists at least one hyperplane such that v_{nn}, v are on opposite sides of this hyperplane. To simplify the problem, we assume that cosine similarity measure is taken and the acute angles formed by v and all word embeddings in the training set are $\theta_1, \theta_2, ..., \theta_n$. There is no doubt that

$$\theta_{nn} = \min_{1\le i\le n} \theta_i \tag{15}$$

$$p_e \le 1 - (1 - \frac{\theta_{nn}}{\pi})^H \tag{16}$$

The explanation is that p_e is greater than the probability that at least one hyperplane separates v, v_{nn}, and difference hyperplanes are independently generated as normal random vectors, so their normal directions should be uniformly distributed in angle. As H is larger, the upper bound increases but the time cost for a single query drops, this shows the so-called accuracy-efficiency trade-off.

Now we make some natural assumptions to do some simple calculations on the random variable θ_{nn} to get its density:

$$\theta_1, ..., \theta_n \overset{i.i.d.}{\sim} U(0, \pi) \tag{17}$$

$$\mathbb{P}(\theta_{nn} \ge x) = (1 - \frac{x}{\pi})^n \tag{18}$$

So, we estimate a probabilistic bound on p_{error}, note that $H = O(\log n)$:

$$\mathbb{P}(p_{error} \ge \delta) \le \mathbb{P}\left(\theta_{nn} \ge \pi[1 - (1 - \delta^{\frac{1}{U}})^{\frac{1}{H}}]\right) \tag{19}$$

$$= (1 - \delta^{\frac{1}{U}})^{\frac{n}{H}} \tag{20}$$

$$= O((1 - \delta^{\frac{1}{U}})^{n^{1-\varepsilon}}) \quad \forall \varepsilon > 0 \tag{21}$$

As what can be seen, we have exponentially decaying for this probability in n, so we argue that LSH for multi-universe works well for large corpus when $n \to \infty$.

Document Search

The main thoughts remain the same for words and documents, and the only difference is the way to derive the embedding of a sentence or document from the embedding of a word.

One simple strategy is call **Bag-of-Words**, which is to just add the embeddings of all words appearing in the document (of course, preprocessing is needed). Such approach is easy to use but has its disadvantages, e.g. it ignores the difference of the order of the appearances of words which may result in totally different meanings. However, it suffices to implement document search for simple sentences. The process is barely the same as that for words, get the document embedding of the given sentence and search in the training set the sentence that has the most similarity score. As we have mentioned, LSH can be applied to make this searching process more efficient.

Probability Methods

Naive Auto Correction

The very first task to deal with is the auto correction problem, where for any input word we would like to determine whether it's a legal word and show some possible corrections if it's misspelled. Actually, this task uses some really trivial probability methods, but the familiarity with the practice of auto correction is still important.

The process is simple: given a corpus, we clean the corpus (**NOTE**: here we only want to lower-case all letters and delete special letters but we won't apply the same preprocessing we have done in sentiment analysis since here spelling is an important matter to consider!) to get a list of all words appearing in the corpus. For each given word, we first check whether it's in the corpus, if so, there's no need to correct it. Otherwise, we introduce some similarity metric and search through all words that (i): are similar to the given words, (ii): appear in the corpus. For those words, we simply calculate the empirical frequency they appear in the corpus and sort all possible choices by this frequency value.

For example, we might adopt edit distance as the similarity metric, so if the given word has not appeared in the corpus, we just construct a list of all words in the corpus that have edit distance 1 with the given word. If this list is also empty, we search for words in the corpus that have edit distance 2 with the given word etc.. As long as we find a non-empty list, we calculate the empirical frequency of appearance of all words in the corpus just by diving the time that particular word appears by the total count of all words in the corpus. Then correction recommendations are made from words with the highest frequency to words with lower frequency.

Edit Distance

Edit distance works as a similarity metric between words, unlike cosine similarity, it does not require vector representation of words. Intuitively, edit distance is the number of minimum operations (the set of operations differs) needed to change one word into another.

Let's first see two different versions of edit distance, difference of edit distance only depends on the set of operations allowed. For Levenshtein edit distance, only insertion, deletion and replacement of one letter are allowed while for Damerau–Levenshtein edit distance, one more operation, the switching between two adjacent letters, is also allowed.

Let's then focus on the **Levenshtein** edit distance to see how to compute the edit distance between two words, which is a classical DP problem. Assume we want to change word w into word v, and the length of these two words are l_w, l_v. Note that the insertion can happen at the beginning of a word, so we insert an empty character at the beginning of both words as indent. For example, if w is "cat", then $w[0] ='', w[1] =' c', w[2] =' a', w[3] =' t'$. We define a matrix $D_{l_w \times l_v}$, with the meaning that $D[i,j]$ is the edit distance between $w[0]w[1]...w[i]$ and $v[0]v[1]...v[i]$.

Then we conclude that $D[l_w, l_v]$ is just the edit distance we want to find and the value of $D[i,j]$ only depends on the values of $D[i-1,j], D[i,j-1], D[i-1,j-1]$ by the following:

$$D[0,0] = 0 \tag{22}$$

$$D[i,0] = i \tag{23}$$

$$D[0,j] = j \tag{24}$$

$$D[i,j] = \min\{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+c_{ij}\} \tag{25}$$

$$c_{ij} = \begin{cases} 0 & if \quad w[i] = v[j] \\ 1 & else \end{cases} \tag{26}$$

The explanation for the last two lines is that when $w[i] = v[j]$, we only have to edit the parts except for the last letter, otherwise we have to make a replacement (cost 1) so that the last letter agree. Except for such approach, we can also think about editting from $w[0]w[1]...w[i-1]$ to $v[0]v[1]...v[j]$ and then delete the last letter in $w[0]...w[i]$ to get $w[0]...w[i-1]$, that's why we add 1 as the cost of deletion here. The same reasoning words for the other term, with 1 as the cost of insertion.

To mention, we can modify the cost of insertion, deletion and replacement as any value we want, which is called a weighted edit distance and we just have to do small modification to the equations above, the structure is still the same.

Speech Tagging

Speech tagging is an important task in NLP since it helps deal with words with multiple meanings. For example, "book" can work as a noun and a verb with different meanings. As a result, how to predict the tag of each word in a given sentence becomes a problem.

First, we should briefly talk about what kinds of tags there are. The tag of a word has many types, for example, "NN" stands for noun in the single form, "NNS" stands for noun in the plural form, "VB" stands for verb with base form, "WRB" stands for wh-adverb etc. To be concise, tags considers the part of speech, the special forms, the tense and other details (like comparatives, superlatives etc.).

Then, given a corpus, what model is suitable for this particular task? Actually, in English, it's often the case that the tag of neighboring words can provide hints for the tag of a particular word. For example, when we have a comparative, we would expect that there is a "than" next to it behind; when we have a superlative, we would expect that there is a "the" next to it in front. Here, we only assume that for any word, only the word next to it in front makes a difference. As a result, this directly fits into **Hidden Markov Model (HMM)**. This model is a match for the following reasons: (i): tags are hidden states, and states are connected with Markov property, i.e. the appearance of the next tag only depends on the current tag (ii): tags as states have emissions that are observable words (iii): the transition and emission probabilities can all be estimated empirically.

To say in other words, now we have an HMM that is completely known (transition & emission probabilities), and we observe a sequence of outputs (words) of this HMM, so we want to figure out the most likely sequence of hidden states (tags) the output belong to. This task is called **decoding** in HMM terms, and can be done with the **Viterbi** algorithm as we will introduce.

Details are exhibited in the next subsection and we will talk about how to build the HMM here. For the transition probabilities, they are probabilities of going to state B given that we are in state A. So it's natural to say that empirically this is just the count of all (A,B) tag pairs in the given sentence over the count of all (A,*) tag pairs, where * stands for arbitrary tag. For the emission probabilities, they are probabilities of exhibiting the word w given that this word is of tag A. So it's also natural to say that empirically this is just the count of word w that has tag A over the count of all words with tag A. These can be computed easily for the corpus given as the training set, that's why we say that this HMM is known.

NOTE: Just like what we've done in sentiment analysis, here we won't want to see any row of 0 in transition matrix or emission matrix, so Laplacian smoothing is adopted. To prevent underflow of float calculations, we only store log probabilities.

For prediction purpose and a given sentence, Viterbi algorithm directly tells us the correspondent tag of each word that is optimal in some sense, which is just the result.

We will see that by applying HMM, speech tagging enjoys more accuracy than the baseline model, which is just outputting the most frequent tag of a given word in the corpus. This is due to the fact that HMM exploits the connection between words! Actually, we are able to build more complicated HMM, for example, HMM that captures the information of two neighboring words (in this case, the states would be a pair of tags not a single tag), and such models are expected to do more accurate jobs.

Refer to the HMM notes for all details.

N-grams Model

Auto-complete task is considered in this section. To fulfill this task, we need to complete the sentence automatically after observing some given words such that the completed sentence is the most likely to appear among any other option.

The N-grams model is a simple model for the relationship between word appearances. Let's consider a sentence $w_1...w_n$ made up of n words $w_1, ..., w_n$, the probability for such sentence to appear is just

$$\mathbb{P}\left(w_1...w_n\right) = \mathbb{P}\left(w_1\right)\mathbb{P}\left(w_2|w_1\right)...\mathbb{P}\left(w_n|w_1...w_{n-1}\right) \tag{27}$$

By adopting the same strategy as we have done before, i.e. to approximate probabilities with empirical frequencies, we are able to compute all those probabilities on the right hand side.

$$\mathbb{P}\left(w_i|w_1...w_{i-1}\right) = \frac{C(w_1...w_i)}{\sum_w C(w_1...w_{i-1}w)} \tag{28}$$

where C stands for the empirical count of a certain sentence in the corpus.

However, we will be facing several problems for such a setting, i.e. we would rarely see any appearance for the sentence $w_1 ... w_{i-1}$ for a pretty large i, which results in the denominator being 0.

The first problem is solved by applying the **N-grams assumption**. The assumption asserts some kind of Markov property that N consecutive words are specified as the largest possible unit where words have certain connections, i.e. the appearance of a certain word only depends on the $N-1$ latest appearing words. By adopting such assumption, we immediately conclude that

$$\begin{aligned}\mathbb{P}\left(w_i|w_1...w_{i-1}\right) &= \mathbb{P}\left(w_i|w_{i-N+1}^{i-1}\right) && (29)\\ &= \frac{C(w_{i-N+1}...w_i)}{\sum_w C(w_{i-N+1}...w_{i-1}w)} && (30)\end{aligned}$$

For bi-gram model ($N=2$), it's obvious that we are assuming the appearance of a word only depend on the word appearing before and just next to the given word, with

$$\begin{aligned}\mathbb{P}\left(w_1...w_n\right) &= \mathbb{P}\left(w_1\right)\mathbb{P}\left(w_2|w_1\right)...\mathbb{P}\left(w_n|w_{n-1}\right) && (31)\\ &= \mathbb{P}\left(w_1\right)\prod_{i=2}^{n}\mathbb{P}\left(w_i|w_{i-1}\right) && (32)\end{aligned}$$

For tri-gram model ($N=3$), we are assuming the appearance of a word only depend on the latest 2 words appearing before the given word, with

$$\begin{aligned}\mathbb{P}\left(w_1...w_n\right) &= \mathbb{P}\left(w_1\right)\mathbb{P}\left(w_2|w_1\right)\mathbb{P}\left(w_3|w_1w_2\right)...\mathbb{P}\left(w_n|w_{n-1}w_{n-2}\right) && (33)\\ &= \mathbb{P}\left(w_1\right)\mathbb{P}\left(w_2|w_1\right)\prod_{i=3}^{n}\mathbb{P}\left(w_i|w_{i-2}w_{i-1}\right) && (34)\end{aligned}$$

It can be seen that there's another problem that eventually we get a product where exceptional terms appear. Is there a way for us to write $\mathbb{P}\left(w_1\right), \mathbb{P}\left(w_2|w_1\right)$ in the form of $\mathbb{P}\left(w_i|w_{i-2}w_{i-1}\right)$? The answer is yes, by inserting **start tokens** ahead of the sentence. It's easy to figure out that to get the consistent form of the product, we would need $N-1$ start tokens $\langle s\rangle$ ahead of the sentence in N-grams model.

For instance, in bi-gram model, insert a single start token ahead of the sentence, so that $\mathbb{P}\left(w_1\right)$ is now $\mathbb{P}\left(w_1|\langle s\rangle\right)$. In tri-gram model, insert 2 start tokens ahead so that the first 2 exceptional terms now become $\mathbb{P}\left(w_1|\langle s\rangle\langle s\rangle\right)$ and $\mathbb{P}\left(w_2|\langle s\rangle w_1\right)$.

Are we done with the model construction? Actually not yet! Note that we have $\sum_w C(w_{i-N+1}...w_{i-1}w)$ on the denominator of the conditional probability but this is not necessarily equal to $C(w_{i-N+1}...w_{i-1})$, depending on whether w_{i-1} is at the end of the sentence. To make these two quantities the same, why don't we insert an **end token** $\langle e\rangle$ at the end of each sentence. Now we get:

$$\mathbb{P}\left(w_i|w_1...w_{i-1}\right) = \frac{C(w_{i-N+1}...w_i)}{C(w_{i-N+1}...w_{i-1})} \quad (35)$$

where we assume in the following context that each sentence in the corpus has $N-1$ start tokens and 1 end token inserted. Note that these tokens are also considered words w_i.

Inserting tokens makes it easier for us to deal with corner cases practically, and enables us to require the "count dictionaries" as the only information from the corpus. However, adding end token has another crucial advantage as stated below.

It's natural that sentences in the corpus will have different lengths. Our goal in auto-complete task is to figure out the sentence with the largest probability to appear. Of course, this means that the probability of the appearance of sentences with different lengths should be **comparable**! Let's use an example to illustrate why sentences without an end token would generate probabilities that are not comparable.

Assume we now have a corpus with three sentences consisting of only two words 'A' and 'B':

$$AB \tag{36}$$

$$AA \tag{37}$$

$$BB \tag{38}$$

Adopting the bi-gram model and adding start tokens but without end tokens, we conclude that the condition probabilities based on such corpus are

$$\mathbb{P}\left(A|\langle s\rangle\right) = \frac{2}{3} \tag{39}$$

$$\mathbb{P}\left(B|\langle s\rangle\right) = \frac{1}{3} \tag{40}$$

$$\mathbb{P}\left(A|A\right) = \frac{1}{2} \tag{41}$$

$$\mathbb{P}\left(B|A\right) = \frac{1}{2} \tag{42}$$

$$\mathbb{P}\left(A|B\right) = 0 \tag{43}$$

$$\mathbb{P}\left(B|B\right) = 1 \tag{44}$$

As a result, among all sentences with length 2 (start tokens not counted in length):

$$\mathbb{P}\left(\langle s\rangle AA\right) = \frac{1}{3} \tag{45}$$

$$\mathbb{P}\left(\langle s\rangle AB\right) = \frac{1}{3} \tag{46}$$

$$\mathbb{P}\left(\langle s\rangle BA\right) = 0 \tag{47}$$

$$\mathbb{P}\left(\langle s\rangle BB\right) = \frac{1}{3} \tag{48}$$

and they add up to 1. However, this is NOT what we want since the length is now fixed as 2. Such construction only ensures that for a fixed length, the probabilities of all sentences with such length add up to 1. What we actually hope to see is that the probabilities of all sentences with **all possible lengths** add up to 1, since such construction enables us to compare the probabilities of two sentences with different lengths in order to decide which one to take as the prediction.

Now adding the end token and repeat the steps above, we get the conditional probabilities based on the same corpus:

$$\mathbb{P}(A|\langle s\rangle) = \frac{2}{3} \tag{49}$$

$$\mathbb{P}(B|\langle s\rangle) = \frac{1}{3} \tag{50}$$

$$\mathbb{P}(A|A) = \frac{1}{2} \tag{51}$$

$$\mathbb{P}(B|A) = \frac{1}{2} \tag{52}$$

$$\mathbb{P}(A|B) = 0 \tag{53}$$

$$\mathbb{P}(B|B) = 1 \tag{54}$$

$$\mathbb{P}(\langle e\rangle|A) = \frac{1}{3} \tag{55}$$

$$\mathbb{P}(\langle e\rangle|B) = \frac{2}{3} \tag{56}$$

As a result, among all sentences with length 2 (start and end tokens not counted in length):

$$\mathbb{P}(\langle s\rangle AA\langle e\rangle) = \frac{1}{9} \tag{57}$$

$$\mathbb{P}(\langle s\rangle AB\langle e\rangle) = \frac{2}{9} \tag{58}$$

$$\mathbb{P}(\langle s\rangle BA\langle e\rangle) = 0 \tag{59}$$

$$\mathbb{P}(\langle s\rangle BB\langle e\rangle) = \frac{2}{9} \tag{60}$$

The sum of all probabilities of sentences with length 2 appearing is now $\frac{5}{9}$, leaving out $\frac{4}{9}$ probability for sentences with lengths other than 2 to appear, making the probabilities constructed comparable across sentences with different lengths!

Other Tips for N-grams

Some tips would be exhibited here for the N-grams model due to its importance in NLP. As stated before, the only information the corpus can provide us is just the counts of the appearances of the phrases consisting of $N-1$ or N consecutive words (start/end tokens are also considered words). However, we have still not completely solved the problem that the appearance count of a certain phrase can be 0, which causes problems if such count appears on the denominator. It's natural to think about **smoothing** once more to deal with such a problem.

The standard Laplacian smoothing can be adopted here, adding 1 to the count on the numerator, but what's the number we shall add to the count on the denominator? We define the **vocabulary** V as the set of all words for which we compute the conditional probabilities, so we shall actually add $|V|$ to the denominator given the vocabulary.

$$\mathbb{P}\left(w_i|w_1...w_{i-1}\right) = \frac{C(w_{i-N+1}...w_i)+1}{C(w_{i-N+1}...w_{i-1})+|V|} \tag{61}$$

Here we only describe the Laplacian smoothing, and other smoothing methods including backoff, interpolation can also be applied.

The problem is transformed into defining the vocabulary for the corpus now. By its function, we would expect the vocabulary to contain all the words that really matters, i.e. all the words that frequently appear in the main structure of the sentences. The reason for this is that including rarely-appearing words, e.g. people's names, in the vocabulary is meaningless since we would not expect such word to appear once more and the count of the appearance is also very small. If such words remain in the vocabulary and conditional probabilities are computed, these probabilities would be too small to work effectively.

For example, if the corpus contains many similar sentences: Alice likes apples / Bob likes apples / Cindy likes apples / ..., then the names "Alice", "Bob", "Cindy" actually makes no difference since the point is that in the corpus everybody likes apples. So when we meet with a name in the test set, it could be "Jack" that has not appeared in the corpus or it could be "Alice" that has appeared in the corpus once, we would both predict the next word to be "likes".

That's the reason why we do not want to include all words that have appeared in the corpus to be in the vocabulary. Instead, words that have appeared in the corpus frequently should be in the vocabulary. Hence, two criteria are provided as the principles constructing the vocabulary. One could either choose all words with frequency larger than the given **minimum word frequency** or choose the words with the largest frequency until the **size of the vocabulary** reaches a certain threshold. Here we adopt the former one, with the minimum frequency set as 2, i.e. any words appearing larger or equal to 2 times in the corpus will be put in the vocabulary.

After constructing the vocabulary, we would go back to the corpus and replace all out-of-vocabulary (OOV) words with the **unknown label** $\langle UNK\rangle$. Note that the unknown label is also seen as words and processed in exactly the same way we have stated before. The unknown label can actually be viewed as a set containing all words which are not of our interest.

Perplexity

Note that the N-grams is a **generative model**, i.e. it has the capability of generating things from nothing based on the knowledge learnt. The generation of N-grams model is particularly simple. Given some words of a sentence, N-grams just chooses the most likely word from the vocabulary based on the conditional probabilities until an end token is seen, indicating the end of a sentence.

That's where **perplexity** is introduced to measure the effect of the model. Its definition depends on a certain test set W:

$$PP(W) = \mathbb{P}\left(s_1,...,s_m\right)^{-\frac{1}{m}} \tag{62}$$

where s_i is the i-th sentence in the test set ending with the end token, and m is the overall number of words in W including the end token but not include the start token.

To be more specific, consider the bi-gram model, where $w_j^{(i)}$ stands for the j-th word in the i-th sentence and w_i stands for the i-th word in the test set concatenating all sentences.

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m} \prod_{j=1}^{|s_i|} \frac{1}{\mathbb{P}\left(w_j^{(i)} | w_{j-1}^{(i)}\right)}} \tag{63}$$

$$= \sqrt[m]{\prod_{i=1}^{m} \frac{1}{\mathbb{P}\left(w_i | w_{i-1}\right)}} \tag{64}$$

where different sentences are considered to be independent. So actually perplexity is some kind of geometric mean of probabilities. If the result is the same as the test set, then perplexity is just 1, and better model has lower perplexity. Since perplexity can sometimes be very small, log-perplexity is also always used.

Note that perplexity can be affected with many factors in a model. Take the N-gram model as example, the more unknown labels there are, the lower perplexity the model will typically have. It's thus important to fix these hyperparameters when comparing the perplexity of different models.

One might ask: why are we defining perplexity to evaluate a model? Actually, perplexity works especially well for **probabilistic models**. In such models, a probability is often computed for difference options, where the one with the highest probability is taken. However, if two models provide exactly the same predictions, they might not be working at the same accuracy. For example, the first model tells us word A has probability 0.6 of appearing here and word B has probability 0.4 while the second model tells us word A has probability 0.9 of appearing here and word B has probability 0.1. Although they both predict correctly that word A should appear here, the second model is doing a far better job since the probability it computes for word A is much higher! That's why perplexity is introduced as some kind of average of the predicted probability. Perplexity does not care about the final prediction result, but cares about the predicted probability instead, thus being a more reliable measure for probabilistic models than accuracy.

N-gram Procedure

At this point, the whole procedure of the N-grams model is presented, but maybe a little bit messy. Here we provide the whole procedure in a clear way:

- Build up the vocabulary as a subset of the words appearing in the corpus (min frequency etc.)
- Scan the corpus to label OOV words and add start & end tokens to each sentence
- Scan the corpus again to build count dictionaries for N-grams and $(N-1)$-grams
- Compute conditional probabilities
- Predict on the test set and compute perplexity

Train Word Embedding

In the previous section of the vector model, we have talked about word embeddings, i.e. vector representation, and their applications. It's quite obvious that after word embeddings are derived, various techniques in linear algebra and vector calculus can then be applied. Instead of the simple word-by-word and word-by-document constructions of word embeddings based on counts of appearances we have mentioned in the previous section, we want to explore more effective ways of building up word embeddings.

Continuous Bag of Words (CBOW)

The CBOW is a way to build word embeddings based on a machine learning approach. The main assumption adopted is that the appearance of a certain word depends on the closest words next to it. As a result, after cleaning the corpus and turning sentences into word tokens, the concept of **context half-size** is introduced. For each word called **center word**, the **context words** are defined to be all words that have distances to the center word no greater than the context half-size. The center word and the context words are called a **window**, i.e. a neighborhood of the center word.

For example, for a sentence "A B C D E", if the context half-size is fixed as 1, then there are 3 different windows: B as center word with A, C as context words, C as center word with B, D as context words, D as center word with C, E as context words. CBOW accepts the assumption that the appearance of the center word is only affected by its context words. As a result, we would expect the context words to make a prediction on the center word, thus providing us with a prediction model, and the word embedding of the center word is just a by-product of such a model.

To be specific, let's start with one-hot vectors (standard basis vectors with one entry as 1 and other entries as 0) as word embeddings. There's no doubt that given a corpus, we collect all word tokens that have appeared in the corpus to form a vocabulary and we only obtain word embeddings for those words in the vocabulary. Then why don't we use one-hot vectors as word embeddings directly? The problem is that doing so would result in enormous word embedding vectors and the sparsity of one-hot vectors causes many problems in the following applications. For instance, if there are 100000 different words appearing in the corpus, then each word embedding would have to be a vector with length 100000, too enormous to be practically useful. However, although one-hot vectors are not good embeddings themselves, they do provide important understandings since the entries in a one-hot vector can be understood as probabilities. It's more than natural to say that a word A has probability 1 to be itself, and has probability 0 to be any other word in the vocabulary. So how shall we make use of such explanation to derive word embeddings that are dense and contains information about relationship between words?

The answer is: forget about the word embeddings temporarily and turn to building a machine learning model for the prediction of center words given the context words. We will see in a later context how this is going to generate the word embeddings as by-products.

The Prediction Model

Now if we want to build such a model to predict the center word based on the context words, we know that the center word can be represented as a one-hot vector, but how to characterize the information contained in the context words? Recall the probabilistic explanation of one-hot vector, we can adopt a similar strategy to write out the correspondent one-hot vectors for all context words and take the average of those vectors. The average vector is still a non-negative vector with all entries add up to 1, so it still has probabilistic meaning! (another hidden assumption here is that the order of context words has no impact on the center word) If one draws only one word from all the context words, the average vector actually works as the probability distribution of the taken word being any one of the context words.

Now we have successfully constructed the training set of this model: for each window, there is a one-hot vector for the center word and an average vector (which is a probability distribution vector) for context words. The prediction model should be able to predict the center word on seeing the average vector. Multiple models can be introduced here to complete this task and the model we choose here is **a neural network (NN) with one hidden layer**.

Let's dive into the construction of such NN. The input layer has size V, where V is the size of the vocabulary and the output layer also has size V, since both the one-hot vector for the center word and the average vector have length V. What about the hidden layer? It should have size N, a **hyperparameter** standing for the size of a word embedding vector we wish to have. The best value of such N can be determined through the standard cross-validation technique, but here we will just view N as a fixed constant. Assume $W_1.b_1$ to be the weight matrix and the bias vector between the input layer and the hidden layer, W_2, b_2 to be the weight matrix and the bias vector between the hidden layer and the output layer, now we only have to choose two activation functions to construct this NN. Since the input and output vectors have probabilistic explanations as probability distributions, we have to make sure that the output layer generates a vector whose entries are non-negative and sum up to 1. As a result, the activation function for the hidden layer should be selected as the softmax function:

$$S : \mathbb{R}^V \to \mathbb{R}^V \tag{65}$$

$$x \quad \to \begin{bmatrix} \frac{e^{x_1}}{\sum_{i=1}^V e^{x_i}} \\ \frac{e^{x_2}}{\sum_{i=1}^V e^{x_i}} \\ \cdots \\ \frac{e^{x_V}}{\sum_{i=1}^V e^{x_i}} \end{bmatrix} \tag{66}$$

The activation function of the input layer is naturally selected as the ReLU function which imitates the activation of neurons:

$$R : \mathbb{R}^N \to \mathbb{R}^N \tag{67}$$

$$x \to \begin{bmatrix} \max\{x_1, 0\} \\ \max\{x_2, 0\} \\ ... \\ \max\{x_N, 0\} \end{bmatrix} \tag{68}$$

Now that the forward propagation structure of the NN is fixed, we only need a loss function so that back propagation can be used to update the weight matrices and biases along their gradient directions. Since the input and output of such NN are both probability distribution vectors, cross entropy is naturally introduced as the loss function, representing the difference between the true probability distribution of the center word (which should be a one-hot vector) and the predicted probability distribution of the center word.

Given two probability measures $\mathbb{P}, \mathbb{Q}$, the difference of these two measures can be measured by the **KL-divergence**

$$D_{KL}(\mathbb{P}||\mathbb{Q}) = \mathbb{E}_{\mathbb{Q}}\left(\frac{d\mathbb{P}}{d\mathbb{Q}}\log\frac{d\mathbb{P}}{d\mathbb{Q}}\right) \tag{69}$$

$$= \mathbb{E}_{\mathbb{P}}\left(\log\frac{d\mathbb{P}}{d\mathbb{Q}}\right) \tag{70}$$

If you are not familiar with the notation $\frac{d\mathbb{P}}{d\mathbb{Q}}$ as the Radon-Nikodym Derivative here, just take it as the quotient of density in the continuous case or the quotient of probability mass in the discrete case. The main point here is that we want to make the difference between the true probability distribution of the center word y and the predicted probability distribution of the center word $\hat{y}$ as small as possible. Therefore, it's natural to minimize the KL-divergence. Since $y, \hat{y}$ are both discrete probability distributions with V components, we may write out the specific form of the KL-divergence as

$$D_{KL}(y||\hat{y}) = \sum_{i=1}^{V} y_i \log\frac{y_i}{\hat{y}_i} \tag{71}$$

with the convention that $0 \cdot \log 0 = 0$.

Since our objective is to find the W_1, W_2, b_1, b_2 minimizing the KL-divergence, and only $\hat{y}$ is relevant to these parameters, slightly modify the equation above to get:

$$-\sum_{i=1}^{V} y_i \log \hat{y}_i = D_{KL}(y||\hat{y}) - \sum_{i=1}^{V} y_i \log y_i \tag{72}$$

Note that $-\sum_{i=1}^{V} y_i \log y_i$ is the entropy of the distribution y, having nothing to do with all the parameters, minimizing $L(y, \hat{y}) = -\sum_{i=1}^{V} y_i \log \hat{y}_i$ is just enough. Such L is just called the **cross entropy** between the true and the predicted distribution. The cross entropy has its name since its form is very similar to that of entropy and minimizing cross entropy is equivalent to minimizing KL-divergence.

After explaining why we take the cross entropy as the loss function of this NN, we can figure out the back propagation processes for training the parameters. According to what we have described,

$$H = R(W_1 X + b_1) \tag{73}$$

$$\hat{y} = S(W_2 H + b_2) \tag{74}$$

where X is the input average vector for context words and H is the hidden layer vector, the ReLU and softmax are applied for each column of the matrix.

Before deriving formulas for the purpose of training such a model, let's consider a version with batches of samples. Up till now, the NN should accept a single vector $X \in \mathbb{R}^V$ at a time, do forward propagation to calculate the predicted $\hat{y} \in \mathbb{R}^V$, and do back propagation to update the parameters based on the cross entropy loss. For each context word vector in the corpus, such operation needs to be done, resulting in a very long training time.

Here comes the trade-off between efficiency and accuracy, a **batch** of samples is introduced to send multiple samples into the NN for training purpose at one time. For example, if we adopt a batched version with batch size m, $X \in \mathbb{R}^{V\times m}$ should now be a matrix where each column is a $V \times 1$ column vector for a particular sample in the training set. As a result, we would expect the output layer to provide us with a matrix $\hat{Y} \in \mathbb{R}^{V\times m}$ where the i-th column of $\hat{Y}$ is the predicted probability distribution of the center word given the context words correspondent to the i-th column of X. Batched NN enables us to train much faster than the original one (where batch size is actually 1), although some accuracy is lost, it's perfectly acceptable when the batch size is not so big. Now modify all the parameters and vectors into the batched version, we get:

$$X \in \mathbb{R}^{V\times m}, W_1 \in \mathbb{R}^{N\times V}, B_1 \in \mathbb{R}^{N\times m}, H \in \mathbb{R}^{N\times m} \tag{75}$$

$$W_2 \in \mathbb{R}^{V\times N}, B_2 \in \mathbb{R}^{V\times m}, \hat{Y} \in \mathbb{R}^{V\times m} \tag{76}$$

$$H = R(W_1 X + B_1) \tag{77}$$

$$\hat{Y} = S(W_2 H + B_2) \tag{78}$$

where B_1 has each column a copy of b_1 and B_2 has each column a copy of b_2. The only problem left is: how shall we modify the loss function such that it fits well with the batched version of NN. The answer is not too difficult since we can calculate m cross entropies within correspondent samples, we shall take the average of these cross entropies, i.e.

$$Y, \hat{Y} \in \mathbb{R}^{V \times m} \tag{79}$$

$$Y = \left[y_1, ..., y_m\right], \hat{Y} = \left[\hat{y}_1, ..., \hat{y}_m\right] \tag{80}$$

$$L_{batch}(Y, \hat{Y}) = \frac{1}{m}\sum_{k=1}^{m} L(y_k, \hat{y}_k) \tag{81}$$

$$= -\frac{1}{m}\sum_{k=1}^{m}\sum_{i=1}^{V} Y_{ik}\log \hat{Y}_{ik} \tag{82}$$

We are all done with the details of this prediction task for the center word given the context words. Eventually, use gradient descent to train all parameters based on the cross entropy loss (standard back propagation). The training can be done for several epochs (if all samples are used for training for exactly one time, then one **epoch** is done).

Get Word Embeddings

After training such a prediction model, we can immediately get the word embedding information from the trained parameters W_1, W_2. This may seem very weird at first glance since we rarely look into the trained weight matrices of a neural network.

There's multiple ways to extract the information, the most common ways are: recall the size of W_1, W_2 that $W_1 \in \mathbb{R}^{N \times V}, W_2 \in \mathbb{R}^{V \times N}$, it's thus natural to take **the i-th column of** W_1 as the word embedding of the i-th word in the vocabulary with embedding size N. Similarly, **the i-th row of** W_2 can also be taken as the word embedding of the i-th word in the vocabulary with embedding size N. Certainly, the **average** of the i-th column of W_1 and the i-th row of W_2 also works as the word embedding.

As a result, we might ask: why is it reasonable to do this? W_1X gives a matrix whose columns are linear combinations of the columns of W_1 with the rows of X working as coefficients. As a result, the columns of H are actually the ReLU-activated versions of the linear combinations of the columns of W_1, with a bias b_1 added. If two words have close meanings, the prediction tasks with these two words as center words in the same batch should have similar hidden layer values (since hidden layer actually works as extracted features), so their respective columns in W_1 shall not be too far apart. The similar reasoning works for rows of W_2 and the readers are welcome to write it down.

Evaluation of Word Embeddings

At last, we briefly talk about how to evaluate the word embeddings, i.e. does it work better compared to another embedding. There are two types of evaluation, intrinsic and extrinsic.

Intrinsic evaluation focuses on the meaning of the words. For example, use the concept formation we have talked about in previous sections to build a model in order to test whether there is a fixed relationship between word embeddings when the words have special connections in concepts. Besides, clustering shows us the words in the same cluster based on their word embeddings, and checks can be made to evaluate whether the clusters are reasonable. It's always a good idea to apply dimension-reduction techniques such as PCA and visualize the word embeddings to capture the differences between words.

Extrinsic evaluation focuses on the usefulness of word embeddings when fulfilling other NLP tasks. For example, build a speech tagging model to accept the word embeddings as given data and create measures as evaluation on whether this set of word embeddings provide reasonable results. By doing this, particular word embedding methods may be selected for particular NLP tasks.