machine learning mathematics

mohab metwally

11/2020

Contents

Introduction

0.1 notation

$(x, y) \in R^{n_x}, y \in 0, 1$ are the training dataset, where x is input, and y is corresponding output.

therefore $M = \{(x^1, y^1), (x^2, y^2), ..., (x^m, y^m)\}$ is the training set, and M_{test} is test example.

X is the input of training set and $\in R^{n_x \times m}$, $X = \begin{vmatrix} X^1 & X^2 & ... & X^m \end{vmatrix}$, $X^{(i)}$ is the ith column of length n, and can be written as x, or $x^{(i)}$.

Chapter 1

Logistic Regression as a neural network

1.1 definitions

we need an estimate function $\hat{y}$ for the input x, and weight prameters w$\in R^{n_x}, b \in R$.

logstic function is $\hat{y} = p(y = 1|x)$, and can be defined as follows: $\hat{y} = \sigma(w^T x + b)$, where the sigma function is defined by $\sigma(z) = \frac{1}{1+e^{-z}}$, and notice when z $\to \infty, \sigma = 1, z \to -\infty, \sigma = 0$.

1.2 cost function

starting with a estimation linear forward model $\hat{y}$, we calculate the difference between our estimate, and the real value y, and through optimization we try to minimize the difference, or loss/cost through gradient descent, then we update our model's parameters.

loss function is minimizing the difference between estimation $\hat{y}, y$, and can be defined as least squre $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)$, but least squares leads to non-convex loss function(with multiple local minimums).

there are different loss functions, but the most efficient is that which maximize the difference. we can define $P(y|x^{(i)}, \theta) = h(x^{(i)}, \theta)^{y^{(i)}}(1 -$

$h(x^{(i)},\theta)^{1-y^{(i)}}$, to increase the sensitivity to the training set we take the likelihood function, as the loss, $L(\theta) = \prod_{i=1}^{m} P(y|x^{(i)},\theta)$ see (**AppendixA**).

one final step in our model is that as m get larger L tend to go to zero, to solve this we define the average sum of log-likelihood, or loss function to be our Cost function.

we multiply by -1 since the sum of the log-likelihood function is negative.

the Cost function $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} log(h(x^{(i)},\theta)^{y^{(i)}}(1 - h(x^{(i)},\theta)^{1-y^{(i)}})$

loss function is defined as $L(\hat{y},y) = -[ylog(\hat{y}) - (1-y)log(1-\hat{y})]$, L$\in [0-1]$.

cost function is defined as the average of loss function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y^{(i)}},y)$

1.3 Gradient Descent

gradient descent is a way to tune the weighting parameters, the objective is the lean toward the fittest weights with respect to the least cost.

iterate through cost function **J** tuning with respect to weight parameters **w**, **b**.

iterate through: $w := w - \alpha\frac{\partial J}{\partial w}$, $b := b - \alpha\frac{\partial J}{\partial b}$, for tuning w, b for the least **J** possible, such that α is the learning rate of GD.

for simplicity $\partial J/\partial w$ replaced for ∂w, and similarly $\partial J/\partial b$ is replaced for ∂b.

forward propagation,

$$\partial w = \frac{\partial J}{\partial L}\frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w}$$

, similarly

$$\partial b = \frac{\partial J}{\partial L}\frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b}$$

.

$$\partial L/\partial \hat{y} = \frac{-y}{\hat{y}} + \frac{(1-y)}{1-\hat{y}}$$

,

$$\partial\hat{y}/\partial z = \frac{-e^{-z}}{1+e^{-z}} = \hat{y}(1-\hat{y}).$$

$\partial L/\partial z = \hat{y} - y.$

then we can deduce that the final iteration gradient descent step after calculating sigma, loss, and cost functions can be

$$w := w - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial b} = \frac{\alpha}{m}X^T(\hat{y}-y)$$

, and

$$b := b - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}-y)$$

.

1.4 Model training

to train a logistic regression model given data set of **X,y** we divide it into 20% for testing, and 80% for training, such that the training set is used to train our model parameters, and testing set is a separate set to test our model's predictions.

we call $X = \{X_{training}, X_{testing}\}, y = \{y_{training}, y_{testing}\}$ the data set, and $\{X_{training}, y_{training}\}$ the training set,and $\{X_{testing}, y_{testing}\}$ the testing set.

using $X_{training}, y_{training}$ (for the rest of the chapter, and the book i will refer to them by X, y for simplicity) we start **Forward propagation** to estimate $\hat{y}$ calculate the difference between y, $\hat{y}$ through **Cost function** $J(y, \hat{y})$.

going backward to W, b we implement **Backward propagation** through $\frac{\partial J}{\partial \omega}$, and $\frac{\partial J}{\partial b}$.

finally we **update weight parameters** after sufficient iterations until we minimize our cost function completely.

1.5 Forward Propagation

we begin by initializing our weight, and bias parameters ω, b randomly.

1.5.1 Activation Functions

what is activation function?

sigmoid:

relu:

tanh:

Starting with input training set $\mathbf{X}$ in our model. we estimate

$$z = \omega^T X + b$$

. then

$$\hat{y} = \sigma(z)$$

.

calculate cost function

$$J(y, \hat{y})$$

.

1.6 Backward Propagation

after evaluating the cost function $J(y, \hat{y})$ we calculate it's derivative with respect to $\{\omega, b\}$.

$$\partial\omega = \frac{\partial L}{\partial \omega} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \omega}$$

$$\frac{\partial L}{\partial \hat{y}} = -(\frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})}) = \frac{\hat{y} - y}{\hat{y}(1-\hat{y})}$$

$$e^{-z} = \frac{1}{\hat{y}} - 1 = \frac{1-\hat{y}}{\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{e^{-z}}{1+e^{-z}} = (\hat{y})^2 e^{-z}$$

$$\partial\omega = X^T(\hat{y} - y)$$

$$\partial b = \hat{y} - y$$

1.7 Update parameters

we implement the following algorithm with a fixed number of iteration that is customized per application, and tuned by the Engineer, such that each application would require different tuning parameters from which is the iteration number.

we iterate the following: update the parameters ω, b, in the back propagation step using $\partial\omega$, ∂b.

$$\omega = \omega - \frac{\alpha}{m} X^T(y - \hat{y})$$

$$b = b - \frac{\alpha}{m}(y - \hat{y})$$

1.8 Summary

1.9 Logistic Regression in Python

1.10 References

Chapter 2

Neural Networks

2.1 Lingua franca

- RELU Activation Function: It turns out that using the Tanh function in hidden layers is far more better. (Because of the zero mean of the function). Tanh activation function range is [-1,1] (Shifted version of sigmoid function). Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem. RELU stands for rectified linear unit, it's a rectifier Activation function and can be defined as $f(x) = x^+ = max(0, x)$ or $\begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$ relu shows to be better replacement to sigmoid function σ for the reason that it help in vanquishing gradient problem.

- Neuron: is a linear regression algorithm denoted by z, or a, $z = W^T X + b$, such that W is the the weight vector of the network.

- Shallow Layers: also known as Hidden Layers, is a set of neurons, for example of the network of composed of input X, and output Y, with at least a single layer $L1$, and at most 2 layers, then the forward propagation will be as follows: we calculate the logistic function for the first layer (1), $z_i^1 = w^T X_i + b_i$, $\hat{Y}^{(1)} = \sigma(z_i^1)$, then we proceed to calculate the final logistic evaluation for the output layer with $\hat{Y}^{(1)}$ as an input instead of X, and so on we proceed replacing $\hat{Y}^{(i)}$ instead of X as new input.

- Layer: layer $L_{(i)}$ is $\hat{Y}^{(i)} = [\hat{y}_1^{(i)}, \hat{y}_2^{(i)}, ..., \hat{y}_n^{(i)}]$ such that n is the length of the layer $L_{(i)}$. each $\hat{y}_j^{(i)}$ is weighted with unique weight vector with previous layer $L_{(i-1)}$.
- Neural Network(NN): is a set of interconnected layers, $< X, L_1, L_2, ..., L_m, Y >$
- Deepness: shallow layer as defined to consist of 1-2 hidden layers, but on the other hand Deep Network is consisting of more than 2 inner, or hidden layers.

we discussed in previous chapter that $\frac{\partial \hat{y}}{\partial z}$, is actually for the logistic activation function σ only, we need to calculate the same derivation for tanh, and RELU.

for $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ is $\frac{\partial \hat{y}}{\partial z} = 1 - tanh(z)^2$

for relu activation function $\frac{\partial \hat{y}}{\partial z} = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$

in NN there are plenty of parameters to worry about, for example the weights need to be initialized randomly, with small values, and b can be initialized as zero.

2.2 Model training

Training a deep neural network is analogous to training a logistic regression network, as discussed in previous chapter, a logistic regression model can be considered a neural network with zero **hidden layers**.

We follow the same algorithm, parameter initialization, but in this case we initialize the parameters for each layer, assume a network composed of 2 hidden layers $X \rightarrow L1 \rightarrow L2 \rightarrow Y$, layer $(L^{(i)}, L^{(i-1)})$ are interconnected with weight,bias parameters $\omega^{(1)}, b^{(1)}$, such that $\omega^{(i)}$ is a matrix of shape $(length(L^{(i)}), length(L^{(i-1)})$, and b is a vector of shape $(length(L^{(i)}, 1))$, so each node in the layer $L^{(i)}$ is connected with each node in previous layer $L^{(i-1)}$.

W, b are ought to be randomly initialized, in the logistic regression discussed in previous chapter, W, b should be initialized with zero values, but in

the case of the neural network zero initialization leads to $\hat{y} = 0$, and all the nodes share the same weight which violates the purpose of the nodes in the first place which is to capture features from the data set as much as possible, so random initialization is necessary, and not to overshoot, initialization better in range $]0 - 1[$ weighted by small value around 0.01 to reduce the sensitivity, in order for the gradient descent to not take for ever.

forward and back propagation are executed in a similar way to logistic regression with minor differences.

calculation of forward propagation using inputs from the previous layer instead of input data set, activation function can be **sigmoid**, **relu**, or **tanh**, it has been tested that **relu** activation function in the first layers shows better results, and **sigmoid** activation in the last layer to fit perfectly with the output classification y-vector in the range $[0 - 1]$

2.3 Parameter initialization

Iterating through each layer, such that L^i layer of length l_i nodes, and previous layer L^{i-1} of l_{i-1} nodes, the weight parameter at layer L is inter-connected with all nodes in the previous layer, meaning that nodes at layer L^i ought to have weight matrix of shape (l_i, l_{i-1}), and bias of shape $(l_i, 1)$.

Unlike the logistic regression, in Neural network initialization step is crucial step, in logistic regression there is only a single activation node extracting a single feature such as price of a house for example, but we intend to employ neural networks to capture as many features as possible inside each node, and therefor initialization ought to be with random values, otherwise we end up with similar copies of each node forward, and backward propagation.

and since we choose random W^i we can choose b^i to be zero.

2.4 Forward Propagation

similar to logistic regression forward propagation, done for each layer, with little difference that instead of using sigmoid function σ we replace it with **relu** function, which is shows better results, and activate the last layer with sigmoid function to distribute the output toward the extremes.

the forward propagation step is done on the input A^{i-1} from previous layer, with current layer L^i parameters ω^i, b^i.

we iterate through layers L^i such that $i \in [0, L]$

$$z^i = (\omega^i)^T A^{(i-1)} + b^i$$

.

$$A^i = \begin{cases} \text{relu}(z^i) \leftarrow \text{if i} < \text{L} \\ \sigma(z^i) \leftarrow \text{if i} = \text{L} \end{cases}$$

2.5 Backward Propagation

$$\frac{\partial L^i}{\partial \omega^i} = \frac{\partial L^i}{\partial A^{(i-1)}} \frac{\partial A^{(i-1)}}{\partial z^i} \frac{\partial z^i}{\omega^i}$$

$$\partial A^{(i-1)} = \partial z^i (\frac{\partial A^{(i-1)}}{\partial z^i})^{-1} = \partial z^i \frac{\partial z^i}{\partial A^{(i-1)}}$$

since the last activation function is sigmoid, then

$$\partial z^L = y^L - \hat{y^L}$$

the back propagation algorithm start with the following initialization step:.

$$\partial A^{(L)} = \frac{(\hat{y} - y)}{\hat{y}(1 - \hat{y})}$$

$$\partial A^{(L-1)} = (\omega^{(L)})^T (y^{(L)} - \hat{y^{(L)}})$$

$$\frac{\partial z^i}{\partial A^{(i-1)}} = \omega^T$$

$$\partial A^{(i-1)} = \omega^T \partial z^i$$

2.6 Summary

2.7 Deep neural networks in Python

Chapter 3

Natural language processing

3.1 pre-processing

the problem here is how to extract features **X** from the a sentence.

for example how to classify a sentence being positive, or negative, assigning 0 for negative, and 1 for positive, starting for a preprocessed sentence how to turn it into a feature set X.

but there are unnecessary punctuation, conjugation, and stops that need to be get rid of, so first we need to pre-process our dataset as follows:

1. iliminate handles and URLs
2. tokenize the string into words
3. remove stop words such as "and, is, a, on, etc" n
4. covert every word into it's stem form
5. lower case transformation

3.2 Example: positive, negative classifier

given sentence s="i love NLP, therefore i study it" how to classify s=['i', 'love', 'NLP', ',', 'therefore', 'i', 'study', 'it'] as positive or negative?.

for a set of strings $S = \{s_1, s_2, .., s_n\}$, matching each string against a vocabulary of all words to end up with two vectors of word frequency,

we create positive frequency vector freqs(w,1), and negative frequency vector freqs(w,0) such that w stand for sentence word, and such that $X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$

given a set of sentences, each is labeled for training as either positive, or negative. we mark each word in a positive-labeled sentence as positive, even if the word is negative, and conversely the opposite with negative-labeled sentences, we mark every word as negative.

first of all we create a set vocabulary V that includes all sets, or sentences S, $V = \{words(s_i)|s_i \in S\}$.

for example in training sets s_1, s_2, s_1="i love NLP, therefore i study it" labeled as positive, and s_2="society no longer in need for black magic, or superstition" labeled negative, we can extract pos-neg features against vocabulary V=['i', 'love', 'NLP', ',', 'therefore', 'study', 'it', 'society', 'no', 'longer', 'in', 'need', 'for', 'black', 'magic', 'or', 'superstition'], pos-freqs = [2, 1, 1, 1, 1, 1, 1, 0, 0, 0, ..., 0], neg-freqs = [0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]. X_m=[1, 8, 11]. for m training sets, $X_m = \begin{vmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} \\ ... & & \\ 1 & x_1^{(m)} & x_2^{(m)} \end{vmatrix}$

3.3 Logistic regression classifier

let's see how this fits inside the Gradient Descent algorithm, for $\sigma(X^i, \theta^i) = \frac{1}{1+e^{-z}}$, we add the bias b to the X^i itself, and therefore $z = \theta^T X^i$.

therefore for a positive z we get $\sigma > 0.5$, and inversely for negative z we have $\sigma < 0.5$.

now we ready for Gradient Descent, given initial parameters θ, we predict, or evaluate the logistic:

$\sigma(X^i, \theta_j)$, then loss function $L(\hat{y}, y) = -[\hat{y}log(\hat{y}) + (1-y)log(1-\hat{y})]$, gradient $\nabla = \partial J/\partial\theta_j = \frac{X^T}{m}(\hat{y} - y)$, updating $\theta_j := \theta_j - \alpha\nabla$. iterate through gradient descent k times.

3.4 Naive Bayes classifier

The prior probability represents the underlying probability in the target population that a tweet is positive versus negative. In other words, if we had no specific information and blindly picked a tweet out of the population set, what is the probability that it will be positive versus that it will be negative? That is the "prior".

The prior is the ratio of the probabilities $\frac{P(D_{pos})}{P(D_{neg})}$.

We can take the log of the prior to rescale it, and we'll call this the logprior

$$\text{logprior} = log\left(\frac{P(D_{pos})}{P(D_{neg})}\right) = log\left(\frac{D_{pos}}{D_{neg}}\right)$$

.

Note that $log(\frac{A}{B})$ is the same as $log(A) - log(B)$. So the logprior can also be calculated as the difference between two logs:

$$\text{logprior} = \log(P(D_{pos})) - \log(P(D_{neg})) = \log(D_{pos}) - \log(D_{neg})$$

To compute the positive probability and the negative probability for a specific word in the vocabulary, we'll use the following inputs:

$freq_{pos}$ and $freq_{neg}$ are the frequencies of that specific word in the positive or negative class. In other words, the positive frequency of a word is the number of times the word is counted with the label of 1.

N_{pos} and N_{neg} are the total number of positive and negative words for all documents (for all tweets), respectively. - V is the number of unique words in the entire set of documents, for all classes, whether positive or negative.

We'll use these to compute the positive and negative probability for a specific word using this formula:

$$P(W_{pos}) = \frac{freq_{pos} + 1}{N_{pos} + V}$$

$$P(W_{neg}) = \frac{freq_{neg} + 1}{N_{neg} + V}$$

Notice that we add the "+1" in the numerator for additive smoothing.

To compute the loglikelihood of that very same word, we can implement the following equations:

$$\text{loglikelihood} = \log\left(\frac{P(W_{pos})}{P(W_{neg})}\right)$$

$$p = logprior + \sum_{i}^{N}(loglikelihood_i)$$

Some words have more positive counts than others, and can be considered "more positive". Likewise, some words can be considered more negative than others.

One way for us to define the level of positiveness or negativeness, without calculating the log likelihood, is to compare the positive to negative frequency of the word.

Note that we can also use the log likelihood calculations to compare relative positivity or negativity of words.

We can calculate the ratio of positive to negative frequencies of a word.

Once we're able to calculate these ratios, we can also filter a subset of words that have a minimum ratio of positivity / negativity or higher.

Similarly, we can also filter a subset of words that have a maximum ratio of positivity / negativity or lower (words that are at least as negative, or even more negative than a given threshold).

$$\text{ratio} = \frac{pos_{words} + 1}{neg_{words} + 1}$$

In prevision section a Logistic regression classified, but we can quick solve the same problem in much simpler algorithm through the evaluation of the likelihood of a sentence being positive matched against given Vocabulary table V.

Recall that conditional probability $p(w_i|pos) = \frac{p(w_i \cap pos)}{p(pos)}$, $p(pos) = freq(pos)/total$, $p(w_i \cap pos) = freq(w_i)/total$, then $p(w_i|pos) = freq(w_i)/freq(pos)$

Likelihood of a positive is defined as $\prod_{i=1}^{m} \frac{p(w_i|pos)}{p(w_i|neg)}$, if likelihood > 1 then sentence is positive, otherwise, it's negative.

to reduce the sensitivity of each word, and avoid getting 0 $p(w_i|class)$, $class \in \{pos, neg\}$ we modify the conditional probabilistic frequency using the so-called 'laplacian smoothing': $p(w_i|class) = \frac{freq(w_i|class)+1}{freq(class)+unique(V)}$, freq(class) is defined as N_{class}, and unique(V) is defined as N_V, for example $p(w_i|pos) = \frac{freq(w_i|pos)+1}{N_{pos}+N_V}$.

to keep the scale small as possible likelihood is replaced with log of likelihood coined with symbol $\lambda(w_i) = log(\frac{p(w_i|pos)}{p(w_i|neg)})$, and a prior=$log(\frac{p(pos)}{p(neg)})$, the classifier of a sentence W is equivalent to prior + $\sum_{i=1}^{m} \lambda w_i$, and if $\lambda > 0$ then it's positive, and negative otherwise.

3.5 costine similaritis

The cosine similarity function is:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

A and B represent the word vectors and A_i or B_i represent index i of that vector.

Note that if A and B are identical, you will get $cos(\theta) = 1$.

Otherwise, if they are the total opposite, meaning, $A = -B$, then you would get $cos(\theta) = -1$.

If you get $cos(\theta) = 0$, that means that they are orthogonal (or perpendicular).

Numbers between 0 and 1 indicate a similarity score.

Numbers between -1-0 indicate a dissimilarity score.

3.5.1 Euclidean distance

You will now implement a function that computes the similarity between two vectors using the Euclidean distance. Euclidean distance is defined as:

$$d(\mathbf{A}, \mathbf{B}) = d(\mathbf{B}, \mathbf{A}) = \sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2 + \cdots + (A_n - B_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

n is the number of elements in the vector

A and B are the corresponding word vectors.

The more similar the words, the more likely the Euclidean distance will be close to 0.

3.6 Principle Component Analysis (PCA)

PCA is a method that projects our vectors in a space of reduced dimension, while keeping the maximum information about the original vectors in their reduced counterparts. In this case, by *maximum infomation* we mean that the Euclidean distance between the original vectors and their projected siblings is minimal. Hence vectors that were originally close in the embeddings dictionary, will produce lower dimensional vectors that are still close to each other.

such that similar words will be clustered next to each other. For example, the words 'sad', 'happy', 'joyful' all describe emotion and are supposed to be near each other when plotted. The words: 'oil', 'gas', and 'petroleum' all describe natural resources. Words like 'city', 'village', 'town' could be seen as synonyms and describe a

Before plotting the words, you need to first be able to reduce each word vector with PCA into 2 dimensions and then plot it. The steps to compute PCA are as follows:

- Mean normalize the data
- Compute the covariance matrix of the data (Σ).
- Compute the eigenvectors and the eigenvalues of your covariance matrix
- Multiply the first K eigenvectors by the normalized data.

3.7 Machine Translation

Given dictionaries of English and French word embeddings you will create a transformation matrix 'R'

Given an English word embedding, **e**, you can multiply **eR** to get a new word embedding **f**.

Both **e** and **f** are row vectors.

we can then compute the nearest neighbors to 'f' in the french embeddings and recommend the word that is most similar to the transformed word embedding.

Find a matrix 'R' that minimizes the following equation.

$$\arg\min_{\mathbf{R}} \|\mathbf{XR} - \mathbf{Y}\|_F$$

The Frobenius norm of a matrix A (assuming it is of dimension m, n) is defined as the square root of the sum of the absolute squares of its elements:

$$\|\mathbf{A}\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

In the real world applications, the Frobenius norm loss:

$$\|\mathbf{XR} - \mathbf{Y}\|_F$$

is often replaced by it's squared value divided by m:

$$\frac{1}{m}\|\mathbf{XR} - \mathbf{Y}\|_F^2$$

where m is the number of examples (rows in $\mathbf{X}$).

The same R is found when using this loss function versus the original Frobenius norm.

The reason for taking the square is that it's easier to compute the gradient of the squared Frobenius.

The reason for dividing by m is that we're more interested in the average loss per embedding than the loss for the entire training set.

The loss for all training set increases with more words (training examples),

so taking the average helps us to track the average loss regardless of the size of the training set.

3.7.1 Loss function L

The loss function will be squared Frobenoius norm of the difference between

matrix and its approximation, divided by the number of training examples m.

Its formula is:

$$L(X, Y, R) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij})^2$$

where a_{ij} is value in ith row and jth column of the matrix $\mathbf{XR} - \mathbf{Y}$.

The norm is always nonnegative (we're summing up absolute values), and so is the square.

When we take the square of all non-negative (positive or zero) numbers, the order of the data is preserved.

For example, if $3 > 2, 3^2 > 2^2$

Using the norm or squared norm in gradient descent results in the same location of the minimum.

Squaring cancels the square root in the Frobenius norm formula. Because of the chain rule, we would have to do more calculations if we had a square root in our expression for summation.

Dividing the function value by the positive number doesn't change the optimum of the function, for the same reason as described above.

We're interested in transforming English embedding into the French. Thus, it is more important to measure average loss per embedding than the loss for the entire dictionary (which increases as the number of words in the dictionary increases).

3.7.2 gradient descent

Calculate the gradient of the loss with respect to transform matrix 'R'.

The gradient is a matrix that encodes how much a small change in 'R' affect the change in the loss function.

The gradient gives us the direction in which we should decrease 'R'

to minimize the loss.

m is the number of training examples (number of rows in X).

The formula for the gradient of the loss function $L(X, Y, R)$ is:

$$\frac{d}{dR}L(X, Y, R) = \frac{d}{dR}\Big(\frac{1}{m}\|XR - Y\|_F^2\Big) = \frac{2}{m}X^T(XR - Y)$$

3.7.3 fixed number of iterations

You cannot rely on training loss getting low – what you really want is the validation loss to go down, or validation accuracy to go up. And indeed - in some cases people train until validation accuracy reaches a threshold, or – commonly known as "early stopping" – until the validation accuracy starts to go down, which is a sign of over-fitting.

Why not always do "early stopping"? Well, mostly because well-regularized models on larger data-sets never stop improving. Especially in NLP, you can often continue training for months and the model will continue getting slightly and slightly better. This is also the reason why it's hard to just stop at a threshold – unless there's an external customer setting the threshold, why stop, where do you put the threshold?

Stopping after a certain number of steps has the advantage that you know how long your training will take - so you can keep some sanity and not train for months. You can then try to get the best performance within this time budget. Another advantage is that you can fix your learning rate schedule – e.g., lower the learning rate at 10

Pseudocode:

1. Calculate gradient g of the loss with respect to the matrix R.

2. Update R with the formula:

$$R_{\text{new}} = R_{\text{old}} - \alpha g$$

Where α is the learning rate, which is a scalar.

3.7.4 k-Nearest neighbors algorithm

k-NN is a method which takes a vector as input and finds the other vectors in the dataset that are closest to it.

The 'k' is the number of "nearest neighbors" to find (e.g. k=2 finds the closest two neighbors).

3.7.5 Searching for the translation embedding

Since we're approximating the translation function from English to French embeddings by a linear transformation matrix R , most of the time we won't get the exact embedding of a French word when we transform embedding e

of some particular English word into the French embedding space.

This is where k-NN becomes really useful! By using 1-NN with eR as input, we can search for an embedding f (as a row) in the matrix Y which is the closest to the transformed vector eR.

Note: Distance and similarity are pretty much opposite things.

We can obtain distance metric from cosine similarity, but the cosine similarity can't be used directly as the distance metric.

When the cosine similarity increases (towards 1), the "distance" between the two vectors decreases (towards 0).

We can define the cosine distance between u and v as

$$d_{\cos}(u, v) = 1 - \cos(u, v)$$

3.7.6 LSH and document search

In this part of the assignment, you will implement a more efficient version of k-nearest neighbors using locality sensitive hashing. You will then apply this to document search.

Process the tweets and represent each tweet as a vector (represent a document with a vector embedding).

Use locality sensitive hashing and k nearest neighbors to find tweets that are similar to a given tweet.

we will now implement locality sensitive hashing (LSH) to identify the most similar tweet.

Instead of looking at all 10,000 vectors, you can just search a subset to find its nearest neighbors

we can divide the vector space into regions and search within one region for nearest neighbors of a given vector.

3.7.7 Bag-of-words (BOW) document models

Text documents are sequences of words.

The ordering of words makes a difference. For example, sentences "Apple pie is

better than pepperoni pizza." and "Pepperoni pizza is better than apple pie"

have opposite meanings due to the word ordering.

However, for some applications, ignoring the order of words can allow

us to train an efficient and still effective model.

This approach is called Bag-of-words document model.

Document embedding is created by summing up the embeddings of all words in the document.

If we don't know the embedding of some word, we can ignore that word.

3.7.8 Choosing the number of planes

Each plane divides the space to 2 parts.

So n planes divide the space into 2^n hash buckets.

We want to organize 10,000 document vectors into buckets so that every bucket has about 16 vectors.

For that we need $\frac{10000}{16} = 625$ buckets.

We're interested in n, number of planes, so that $2^n = 625$. Now, we can calculate $n = \log_2 625 = 9.29 \approx 10$.

In 3-dimensional vector space, the hyperplane is a regular plane. In 2 dimensional vector space, the hyperplane is a line.

Generally, the hyperplane is subspace which has dimension 1 lower than the original vector space has.

A hyperplane is uniquely defined by its normal vector.

Normal vector n of the plane π is the vector to which all vectors in the plane π are orthogonal (perpendicular in 3 dimensional case).

3.7.9 Getting the hash number for a vector

For each vector, we need to get a unique number associated to that vector in order to assign it to a "hash bucket".

Using Hyperplanes to split the vector space:

We can use a hyperplane to split the vector space into 2 parts.

All vectors whose dot product with a plane's normal vector is positive are on one side of the plane.

All vectors whose dot product with the plane's normal vector is negative are on the other side of the plane.

Encoding hash buckets:

For a vector, we can take its dot product with all the planes, then encode this information to assign the vector to a single hash bucket.

When the vector is pointing to the opposite side of the hyperplane than normal, encode it by 0.

Otherwise, if the vector is on the same side as the normal vector, encode it by 1.

If you calculate the dot product with each plane in the same order for every vector, you've encoded each vector's unique hash ID as a binary number, like [0, 1, 1, ... 0].

hash algorithm:

We've initialized hash table 'hashes' for you. It is list of $N_{universe}$ matrices, each describes its own hash table. Each matrix has N_{dims} rows and N_{planes} columns. Every column of that matrix is a N_{dims} dimensional normal vector for each of N_{planes} hyperplanes which are used for creating buckets of the particular hash table.

First multiply your vector 'v', with a corresponding plane. This will give you a vector of dimension N_{planes}.

You will then convert every element in that vector to 0 or 1.

You create a hash vector by doing the following: if the element is negative, it becomes a 0, otherwise you change it to a 1.

You then compute the unique number for the vector by iterating over N_{planes}

Then you multiply 2^i times the corresponding bit (0 or 1).

You will then store that sum in the variable $hash_{value}$.

$$hash = \sum_{i=0}^{N-1} \left(2^i \times h_i\right)$$

3.8 Probabilistic model of pronounciation and spelling

3.8.1 auto-correction

the misspelling is quite common in writing, and to transduct a word from the misspelled form to dictionary closest word, most relevant to the context for spelling, and pronunciation we utilize **Bayes Rule**, and **the noisy channel model**, and this problem can be divided into two categories:

1. word error detection: in which the algorithm is run on the word in isolation.

2. context error detection: where correction take place in a specific context.

80% of the misspelled words are caused by single-eror misspellings: can be divided into four categories:

- insertion: mistyping the as ther
- deletion: mistyping the as th
- substitution: mistyping the as thw
- transposition mistyping the as hte

3.8.2 Bayesian inference model

given a noisy word through noisy channel, **O** as our observation, we need to match it to the nearest word in the dictionary.

we build a vocabulary **V**, and our model ought to map noisy **O** to $\hat{w}$.

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w|O)$$

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} \frac{P(O|w)P(w)}{P(O)}$$

since we iterate through the whole word set in the vocabulary **V** P(O) is fixed, and we can ignore it, then:

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(O|w)P(w)$$

where P(w) is the **prior**, P(O—w) is the **likelihood** function.

p(w) is the word frequency: $\frac{frequency(w)}{sizeofthecorpus}$, to avoid getting zero frequency we use Laplacian smoothing:

$$p(w) = \frac{freq(w) + 1}{N + V}$$

such that V is the size of vocabulary in this context, and N is the size of the corpus.

there different algorithms for error correction, and processing, among those are **minimum edit distance**, **Viterbi forward**, **CYK**, **Earley**.

3.8.3 Minimum edit distance

It's a metric value between different noise channels for the same word, or weight for insertion, deletion, and substitution, weighting each by 1, but substitution by 2 (insertion+substitution), known as **Levenshtein** distance.

given two words target, and source, word distance can be calculated through Dynamic programming, laying out the target of length$\rightarrow n$ in the first column, and source of length $\rightarrow m$ in the first row, and creating matrix **distance** of size $\rightarrow n$ (n+1,m+1).

looping through each column i from $0 \rightarrow n$, and each row j from $0 \rightarrow m$:

$$dstance[i,j] \leftarrow Min \begin{cases} \text{(distance[i-1,j] + inseration-cost}(target_j) \\ \text{distance[i-1, j-1] + substraction-cost}(source_j, target_i) \\ \text{distance[i,j-1] + inseration-cost}(source_j)) \end{cases}$$

Appendices

Appendix A

Introduction to probabilities

A.1 Naive Bayes

Appendix B

Covariance

Appendix C

Single Value Decomposition