Note for Reinforcement Learning 2nd Edition

August 15, 2018

1 Text Classification

Text can be treated as a sequence of characters, words, phrases, named entities, sentences or paragraphs etc.

Tokenization is a process that splits input into useful unit(token) for the task at hand. (Can be word, sentence, paragraph). Example tokenizers are WhiteSpaceTokenizer, WordPunctTokenizer, TreebankWordTokenizer.

Token normalization operates on individual token. There are two main types of normalizations: stemming and lemmatization. **Stemming** uses simple rules and heuristics to remove/replace suffixes (e.g. Porter's Stemmer). **Lemmatization** use more advnace tehcnique such as vocabulary/morphological analysis (e.g. WordNet lemmatizer). Further normalization includes normalizing capital letters and acronyms.

Classical approach

Bag of Words(BOW) is a feature representation of text. For a set of text samples $\{s_1, \ldots, s_n\}$, we can extract a set of distinct tokens ("today", "a", "nice") or token pairs/triplets ("nice weather") $\{t_1, \ldots, t_m\}$. We define the bag of word representation to be an $n \times m$ frequency matrix B where

$$B_{ij} = \text{ \# of times } t_j \text{ appears in } s_i$$

n-grams consecutive tokens extracted from text. An example of bigram representation of a sentence, "Today is sunny" would be ("today is", "is sunny"). The problem is this would cause B to have too many columns. We should remove the high frequency n-grams (e.g. stop words such as "a", "the") which are not useful, and low frequency n-grams which are consisted of typos, rare n-grams (prevent overfitting). We should keep the medium frequency n-grams, they are the most representative of the sample set. To filter the n-grams with high and low frequency, we use two measures: **Term frequency(TF)** and **Inverse document frequency(IDF)**.

$$\text{TF: } td(t, d) = \text{Frequency of n-gram } t \text{ in document } d$$

We can use the following ways to calculate TF: binary (0, 1), raw count $f_{t,d}$, term frequency $f_{t,d} / \left(\sum_{t' \in d} f_{t',d}\right)$ and log normalization $1 + log(f_{t,d})$.

$$\text{IDF: } idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

Where D is the set of all documents in corpus. $|\{d \in D : t \in d\}|$ is the set of document where the term t appears.

$$\text{TF-IDF : } tdidf(t, d, D) = td(t, d) idf(t, D)$$

is a useful quantity to rank the terms (n-grams). Large td-idf value gives terms that are abundunt in a small number of documents.

We can get a better BOW by using TD-IDF for B_{ij} and normalize each row with L_2-norm.

Logistic regression can be used to do sentiment classification

$$p(y = 1 \mid x) = \sigma(w^T x)$$

Where x is rows of BOW matrix.

In the case of a large dataset distribute across machines, we need to map n-gram to column index of the BOW matrix. It is conveninent to use hash mapping to column indices.

$$ngram \to hash(ngram) \mod 2^{20}$$

Hash function can be defined as

$$hash(s) = \sum_{i=0}^{n} s[i] p^i$$

where s is a string, p a given prime and $s[i]$ is the ith charCode.

In the example of spam filtering, we might want to customize for each user. So the term t might be a spam word for user A but not other users. To do this, we change the hash function to

$$hash_u(s) = hash(u + \text{"\_"} + s) \mod 2^b$$

Where u is the user id string and + is string concatenation. In this way, "userA_spamword" and "userB_spamword" are basically different words customized for A and B.

Deep Learning approach

BOW matrix is very sparse and high dimensional, instead we use Word2Vec embeddedings which are dense vectors in a much lower dimension. Embeddings are generated by a projection from high dimension one-hot space to a submanifold such that words with similar mearning/functional role are near one another on the submanifold.

Analogous to n-gram, we use 1d convolution to achieve the same thing. Given a sequence of tokens $s_0, \ldots, s_m$ and embeddings dimension of k, we compute the $m \times k$ embedding matrix for the sequence where the ith row is the embedding vector for s_i. Next we perform convolution for each ith row up to $i + n$th row with a $n \times k$ filter matrix F where the choice of n is analogous to choosing n-gram. For example (n = 2) :

$$\begin{pmatrix} a \\ fine \\ whether \\ today \end{pmatrix} \overset{embedding}{\Longrightarrow} \begin{pmatrix} 0.3 & 0.4 \\ 0.1 & 0.5 \\ 0.6 & 0.7 \\ 0.9 & 0.2 \end{pmatrix}$$

$$\begin{pmatrix} 0.3 & 0.4 \\ 0.1 & 0.5 \\ 0.6 & 0.7 \\ 0.9 & 0.2 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 4 \times 0.3 + 3 \times 0.4 + 2 \times 0.1 + 1 \times 0.5 \\ 4 \times 0.1 + 3 \times 0.5 + 2 \times 0.6 + 1 \times 0.7 \\ 4 \times 0.6 + 3 \times 0.7 + 2 \times 0.9 + 1 \times 0.2 \end{pmatrix}$$

This is 1D-convolution in that the filter only slides downward. Note that we can perserve the output row count by zero padding.

Convolution is essentially a dot product between an n-gram vector and filter vector (if you flatten them into vectors). Similar n-grams give similar value since they are close in cosine distance. Also filter can act as a detector for a certain type of n-gram(mearning) if the n-gram is consine aligns with the filter. To detector mutiple meanings, we use a bank of filters.

Finally, we need to fix the output dimension pooling the output of each filter(usually max pooling). If we have a bank of b filters, the pooling output would be a b dimensional vector.

Text as characters

Instead of text as a sequence of words, we can use text as a sequence of characters. For each character we can represent it as a one hot vector. For a document string of length m, and an alphabet of letter of size k, we can present the document as a $m \times k$ matrix where each row is a one hot vector of the m-th letter in the document.

Once we have the matrix, we can apply filter-pooling layers as many layer deep as needed since each layer's output is a smaller/same size matrix.

(TODO: Include tri-letter representation of text)

2 Language Modeling

Given a sequence of words $w = ()w_1, w_2, \dots, w_k)$, we can predict it's probability by

$$p(w) = p(w_1, w_2, \dots, w_k) = p(w_1)p(w_2 \mid w_1)p(w_k \mid sw_1 \dots w_{k-1}) \dots$$

We can simply this using Markov assumption cut off the dependency after $n-2$ terms.

$$p(w_i \mid w_1 \dots w_{i-1}) = p(w_i \mid w_{i-n+1} \dots w_{i-1})$$

When $n = 2$, $p(w_i \mid w_1 \dots w_{i-1}) = p(w_i \mid w_{i-1})$. This is the bigram model.

Note that $p(w) = p(w_1)p(w_2 \mid w_1) \dots p(w_k \mid w_{k-1})$ now has two problems. Suppose the sentences are "A tree is here", "Two houses are there", firstly $p(a) = count(a)/count(all - words) = 1/8$ which is far too small since it is divided over all words while the two sentences only start with either "a" or "two". We solve this by adding a "start" token at the beginning of the sentence so $p(a)$ becomes $p(a \mid start) = 1/2$.

Secondarily, w can be any length, this distribution is not normalized across all w (For example: $p(a) + p(two) = p(a \mid start) + p(two \mid start) = 1/2 + 1/2 = 1$). To solve this, we add "end" token at the end of a sentence.

Combining both (use # for both start and end), we have

$$p(w) = p(w_1 \mid \#)p(w_2 \mid w_1) \dots p(w_k \mid w_{k-1})p(\# \mid w_k)$$