CS584: Natural Language Processing Notes

Steven DeFalco

Spring 2024

Contents

1 Regular Expressions, Text Normalization, Edti Distance

Regular expressions can be used to specify strings we might want to extract from a document. ***Normalizing*** text means converting it to a more convenient, standard form. For example, most of what we are going to do with language relies on first separating out or ***tokenizing*** words from running text. Another part of text normalization is ***lemmatization***, the task of determining that two words have the same root, despite their surface differences. For example, the words *sang*, *sung*, and *sings* are forms of the verb *sing.* The word *sing* is the common lemma of these words, and a ***lemmatizer*** maps from all of these to *sing.* ***Stemming*** refers to a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word. Text normalization also includes ***sentence segmentation***: breaking up a text into individual setnences, using cues like periods or exclamation points. Finally, we'll need to compare words and other strings. We'll introduce a metrix called ***edit distance*** that measures how similar two strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other.

1.1 Regular Expressions

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Regular expressions are particularly useful for searching in texts, when we have a pattern to search for a ***corpus*** of texts to search through. A regular expression search function will search through the corpus, returning all texts that match the pattern.

1.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters; putting characters in sequence is called ***concatenation***. Regular expressions are **case sensitive**. We can solve this with the use of square braces [and]. The string of characters inside the braces specifies **disjunction** of characters to match. For example, the pattern /[wW]/ matches patterns containing either w or W.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ˆ. If the caret is the first symbol after the open square brace, the resulting pattern is negated. For example, the pattern /[ˆa]/ matches any single character (including special characters) except a. This is only true when the caret is the first symbol after the open square brace.

We use the question mark /?/, which means *the preceding character or nothing.* For example, /woodchucks?/ will match with either *woodchuck* or *woodchucks.* The ***Kleene star*** means *zero or more occurrences of the immediately previous* *character or regular expression.* So $/a^*/$ means *any string of zero or more as./ Kleene+* means *one or more occurrences of the immediaately preceding character or regular expression.* The period is used as a ***wildcard*** expression that mat ches any single character.

1.1.2 Disjunction, Grouping, and Precedence

The ***disjunction*** operator, also called the **pipe** symbol | mathes either the preceding or following strings. Typically, regular expressions always match the *largest* string they can: we say that patterns are **greedy**, expanding to cover as much of a string as they can. There are, however, ways to enforce **non-greedy** matching, using another meaning of the ? qualifiers. The operator $*?$ is a Kleene star that matches as little text as possible. The operator $+?$ is a Kleene plus that matches as little text as possible.

1.1.3 More Operators

Below are some aliases for common ranges, which can be used mainly to save typing.

- / d: any digit
- / D: any non-digit
- / w: any alphanumeric/underscore
- / W: a non-alphanumeric
- / s: whitespace (space, tab)
- / S: non-whitespace

A range of numbers can also be specified. So $/\{n, m\}/$ specifies from n to m occurrences of the previous char or expression. REs for counting are summarized below.

- $*$: zero or more occurrences of the previous char or expression
- $+$: one or more occurrences of the previous char or expression
- ?: zero or one occurrences of the previous char or expression
- $\{n\}$: exactly n occurrences of the previous char or expression
- $\{n, m\}$: from n to m occurrrences of the previous char or expression
- $\{n, \}$: at least n occurrences of the previous char or expression
- $\{, m\}$: up to m occurrences of the previous char or expression

Some certain special characters are referred to by special notation based on the backslash. The most common of these are the **newline** character / n and the **tab** character /.͡

- / $*$: an asterisk
- / .: a period
- / ?: a question mark
- / n: a newline
- / t: a tab

1.2 Words

A ***lemma*** is a set of lexical forms having the same stem, the same major part-of-speech, adn the same word sense. The ***word form*** is the full inflected or derived form of the word. ***Types*** are the number of distinct words in a corpus; if the set of words in the vocabular is V, the number of types is the vocabulary size $|V|$. ***Tokens*** are the total number N of running words.

The larger the corpora we look at, the more word types we find, and in fact this relationship bettween the number of types $|V|$ and number of tokens N is called ***Herdan's Law*** or ***Heaps' Law***. It is shown below where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^{\beta}$$

The value of β depends on the corpus size and the genre. Roughly then we can say that the vocabulary size for a text goes up significantly faster than the square root of its length in words.

1.3 Text Normalization

Before almost any natural language processing of a text, the text has to be normalized. At least three tasks are commonly applied as part of any normalization process:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

One commonly used tokenization standard is known as the ***Penn Treebank tokenization*** standard. This standard separates out clitics (*doesn't* becomes *does* plus *n't*), keeps hyphenated words together and separates out all punctuation.

Most tokenization schemes have two parts: a token learner and a token segmenter. The ***token learner*** takes a raw training corpus and introduces a vocabulary, a set of token. The ***token segmenter*** takes a raw test sentence and segments it into the tokens in the vocabulary. The most widely used algorithm is byte-pair encoding. The BPE token learner begins with a vocabulary that is just the set of all indivdual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent, adds a new merged symbol to the vocabulary, and replaces every adjacent in teh corpus with the new symbol. It continues to count and merge, creating new longer and longer character strings, until k merges haev been done creating k novel tokens; k is thus a parameter of the algorithm. The resuling vocabulary consists of the original set of characters plus k new symbols. The algorithm is usually run inside words, so the input corpus is first white-space-separated to give a set of strings, each corresponiding to the characters of a word, plus a special end-of-word symbols, and its counts.

2 Lecture 2: Machine Learning Basics

The ***training set*** is the sample of data used to fit the model. The ***validation set*** is the sample of data used to provide unbiased evluation of a model fit on the training dataset while tuning model hyperparameters. The ***test dataset*** is the sample of data used to provide an unbiased evaluation of a final model fit on the training dataset; this cannot be used for training.

Cross-validation allows us to estimate the generalization error based on training examples alone. In ***k-fold cross validation*** the original sample is randomly partitioned in k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k-1$ subsamples are used as training data.

In ***supervised learning***, we have a training dataset containing samples. In each samples, x_i are inputs and y_i are albels. We try to predict the labels using the inputs, and this is the basis of training. With ***unsupervised learning***, we have no labels and instead expect the model to learn some patterns from the dataset.
For **document classifications**, build a feature vector for each document

$$\{x_i, y_i\}_{i=1}^N$$

where x_i can be word counts, TF-IDF, topic distributions, etc...

A ***loss function*** (Loss(x, y, w)) qualifies how wrong you would be if you used w to make a prediction in x when the correct output is y. It is the object we want to minimize. Loss is a function of the parameters w and we can try to minimize it directly. We reduce the estimation problem to a **minimization** problem.

We have a cost function $J(\theta)$ we want to minimize. ***Gradient descent*** is an algorithm to minimize $J(\theta)$: for the current value θ, calculate the gradient of $J(\theta)$, then take small step in direction of negative gradient, and repeat. The ***gradient*** is the direction that increases the loss the most; we want to take the *negative* gradient in gradient descent.

Gradient descent is slow. An ***epoch*** is one pass through the data. Each iteration requires going over all training examples; this is expensive and slow when we have lots of data. In practice, we can use ***stochastic gradient descent***. At each epoch, we can randomly shuffle the data to ensure that each data point creates in *independent* change on the model, without being biased by the same points before them. In this implementation, it may never reach the local minima and could oscilate around it due to the fluctuations in each step. Instead of using the whole data for calculating gradient, in ***mini-batch gradient descent*** we use only a mini-batch of it instead of the whole dataset.

Learning rate affects the update of gradients; it is the amount that we update the weights. A too large learning rate may lead to diverging values. A too small learning rate my cost a lot of time to convert to the minimum points. ***Adaptive learning rate*** involves reducing the learning rate by some factor every few epochs. Divide the learning rate of each parameter by the root mean square of its previous derivatives.

2.1 Neural Networks

Logistic regression gives only linear decision boundaries which are limited and unhelpful when a problem is complex. Neural networks can learn much more complex functions and **nonlinear** decision boundaries. Some of the *pros* of neural networks are the following:

- used on a variety of domains
- predictions are very quick

Backpropagation provides an efficient procedure to compute derivatives. Without nonlinearities, deep neural networks can't do anything more than a linear transform. Extra layers could just be compiled into a single linear transform.

When initializing...

- initialize weights to small random values
- initialize hidden layers to 0 and output biases to optimal value if weights were 0

Regularization prevents overfitting when we have a lot of features. A loss function in practice includes regularization over all parameters. ***Early stopping*** is a popular regularization technique in which we monitor the performance for every epoch on the validation set during the training set and terminate the training as soon as the validation error reaches a minimum.

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. Each neuron has some probability of being dropped out during each iteration.

3 Deep Feedforward Networks

Deep feedforward networks have the goal of approximating some function (target function). They are typically represented by composing together many different functions; there are multiple hidden layers and one output layer.

Nearly all deep learning algorithms can be described as particular instances of a farily simple recipe: combine

- a specification of a dataset
- a cost function
- an optimization procedure
- and a model

To optimize, for linear models, we could use normal equations, closed form optimization. For nonlinear models, it requires us to choose an iterative numerical optimization procedure. The largest difference between linear models and neural networks is the **nonlinearity** of a neural network causes mos tinteresting loss functions to become **nonconvex**. Thus, neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value.

Architecture refers to the overall structure of the netwrok: how many units it should have and how these units should be connected to each other. Most neural networks are organized into groups of units called layers.

A problem is convex if its objective is convex function, the inequality constraints f_j are convex, and the equality constraints h_j are affine.

Momentum is designed to accelerate learning. It accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. Adaptive learnng rates are a simple idea where the learning rate is reduced by some factor every few epochs.

4 Vector Semantics

Words can be represented in multiply ways:

- Knowledge-based representation
- One-hot representation
- Co-occurrence Matrix
- Low-dimensional dense word vector

We could represent words as one-hot vectors where the vector dimension is equal to the number of words in the vocabulary. However, this will lead to high dimensional representation and thus inefficiency in storage and computation.

TF-IDF stands for term frequency - inverse document frequency. Where **TF** is the frequency of word v appearing in the current document and **DF** is the number of documents where word v appears.

When representing with co-occurrence matrix there are two options:

- ***Window***: use window around each word. Captures boith syntactic and semantic information
- ***Word-document***: gives general topics

The limitations with simple co-occurrence vectors are the increase in size with vocabulary, very high dimensional (requires a lost of storage), and models are less robust. A solution is to use low dimensional dense vectors.

4.1 Word Vectors

Distributional semantics is the idea that a word's meaning is given by the words that frequently appear close-by. When a word w appears in a text, its ***context*** is the set of words that appear nearby. Use the many contexts of w to bukld up a representation of w.

We will build a dense vector for each word, chosen so that the vectors of words that appear in similar context will be similar.

Word2vec is a framework for learning word vectors. The idea is that we have a large corpus of text and every word in a fixed vocabulary is represented by a vector. Go through each position t in the text, which has a center word c and context (outside) words o. Use the similarity of the word vectors for c and o to calculate the probability of o given c (or vice versa). Keep adjusting the word vectors to maximize this probability.

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_t. The objective function J is the (average) negative log likelihood: *minimizing the objective function will result in maximizing the predictive accuracy.* To train a model, we adjust parameters to minimize a loss for a simple convex function over two parameters.

When calculating gradients, find the gradient for each center vector v in a window and also calculate the gradients for outside vectors u. Generally, in each window we will compute updates for all parameters that are being used in that window.

Skip-grams (SG) predict context (outside) words given the center word. Skip-gram model with negative sampling is too computationally expensive. Hence, in standard word2vec you implement the skip-gram model with negative sampling. The main idea is to train a binary logistic regressions for a true/positive pair (center word and word in its context window) versus several noise/negative pairs (the center word paired with a random word). We take k negative samples and maximize the probability that real outside word appears, minimize probability that random words appear around center word.

There are two ways to evaluate word vectors.

- ***Intrinsic evaluation*** is evaluation on a specific/intermediate subtask. This is fast to compute. Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy. Identify the word vector which maximizes the cosine similarity. Using intrinsic evaluation techniques should be handeled with care.
- ***Extrinsic evaluation*** is evaluation on a real task. This can take a long time to compute accurately.

Remark Most words have lots of meanings. One vector can not necessarily capture all these meanings.

Different senses of a word reside in a linear superposition (weighted sum) in standard word embeddings like word2vec. Because of ideas from sparse coding, you can actually separate out the senses (providing they are relatively common).

5 Language Modeling

Language modeling is the task of predicting what word comes next. Formally, given a sequence of words, compute the probability distribution of the next word. A system that does this is called a ***language model***. Language modeling is a subcomponent of many NLP tasks.

Pre-training first and then fine-tuning on a specific downstream task, allows to capture context-aware word representation. **BERT** is pretraining with specially designed tasks on large scale unlabaled data, which is very effective to serve as a general-purpose semantic features and improves the performance of NLP taks.

An ***n-gram*** is a large chunk of n consecutive words. The idea of n-gram models is to collect statistics about how frequent different n-grams are, and use those to predict the next word. In general, this is an insufficient model of language because langauge has long-distance dependencies.
The ***Maximum Likelihood Estimate*** (MLE) of some parameter of a model M from a training set T maximizes the likelihood of the training set T given the model M.

Perplexity is the inverse probability of the test set, normalized by the number of words. Minimizing perplexity is the same as maximizing probability. N-grams only work well for word prediction if the test corpus looks like the training corpus. In real life, it often does and thus we must train robust models that generalize. With ***laplace smoothing***, just pretend that we saw each word one more time than we did (just add one to all the counts). ***Backoff*** simply uses less context. ***Interpolation*** mixes the probability estimates from all classes.

In ***Good Turing estimation***, reallocate the probability mass of n-grams that occur c+1 times in the training data to the n-grams that occur c times. In particular, reallocate the probability mass of n-grams that were seen once to the n-grams that were never seen.

5.1 Neural Language Modeling

The idea of ***Recurrent Neural Network*** (RNN) is to apply the same weights W repeatedly. RNN can process any length input and model size doesn't increase for longer input. Computation for step t can use information from many steps back and the weights are shared across time stamps. However, recurrent computation is very slow. To **train a RNN language model**...

1. Get a big corpus of text which is a sequence of words.

2. Feed into RNN-LM; compute output distribution for every step t.

3. Loss function on step t is usual cross-entropy between out predicted probability and the true next word.

4. Average this to get the overall loss for entire training set.

Computing loss and gradients across entire corpus is *too expensive*! Stochastic gradient descent allows us to compute loss and gradient for small set of data and update. In practice, consider the inputs as a sentnece and compute loss $J(\theta)$ for a sentence, update weights, and repeat. Must backpropagate over time steps $i = t, \ldots, 0$, summing gradients as you go ("***backpropagation through time***"). Just like a n-gram LM, we can use a RNN LM to generate text by repeated sampling. Sampled output is the next step's input.

The standard evaluation metric for language models is **perplexity**. Perplexity is the inverse probability of corpus, according to LM normalized by the number of words. This is equal to the exponential of the cross entropy loss $J(\theta)$.

6 More on RNNs

Vanishing gradient is a problem because gradient signals from far away are lost because it's much smaller than gradient signal from close-by. If the gradient becomes vanishingly small over long distances, then we can't tell whether there's no dependency between step t and $t + n$ in the data and we have wrong parameters to capture the true dependency between t and $t + n$. Due to vanshign gradient, RNN-LMs are better at learning from **sequential recency** than syntactic recency.

If the gradient becomes too big (***exploding gradient problem***), then the SGD update step becomes too big. This can cause bad updates: we take too large a step and reach a bad parameter configuration. A solution is ***gradient clipping***: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update (take a step in the same direction, but a smaller step).

The main problem with the vanishing gradient problem is that it's too difficult for the RNN to learn to preserve information over many timesteps. In ***Long Short-Term Memory (LSTM)***, on step t, there is a hidden state $h^{(t)}$ and a cell state $c^{(t)}$. Both are vectors of length n. The cell stores long-term information. The LSTM can erase, write, and read information from the cell. The selection of which information is erased/written/read is controlled by three corresponding gates. The gates are also vectors of length n. On each timestamp, each element of the gates can be open (1), closed (0), or somewhere in between.

Forget gate controls what is kept vs forgotten, from previous cell state. ***Input gate*** controls what parts of the new cell content are written to cell. ***Output gate*** controls what parts of cell are output to hidden states. ***New cell content*** is the new content to be written to the cell. ***Cell content*** erases some content from the last cell state, and writes from new cell content. ***Hidden state*** reads some content from the cell.
Gated Recurrent Units (GRUs) have two gates. The ***update gate*** controls what part of hidden states are updated vs preserved. The ***reset gate*** controls what parts of previous hidden states are used to compute new content. Like LSTM, GRU makes is easier to retain long term information.

The biggest difference between LSTM and GRU is that GRU is quicker to compute and has fewer parameters. LSTM is a good default choice, but switch to GRU if you want something more efficient.

Remark Bidirectional RNNs are only applicable if you have access to the entire input sentence.

RNNs are already *deep* in on dimension, but we can also make them deep in another dimension by applying multiple RNNs—this is a multi-layer RNN. This allows the network to compute more complex representations. High performing RNNs are often multi-layer (but not as deep as convolutional networks).