CS263 SP22: Language Models, Embeddings, and HMMs

Nurrachman Liu
rachliu@cs.ucla.edu

Abstract

We examine basic understanding of language models, sparse-embeddings, dense-embeddings, and part-of-speech-tagging. For language models, we compare custom n-gram model with those of NLTK. For embeddings, we compare custom sparse-embeddings and skip-gram-with-negative-sampling with those of NLTK or Gensim. For POS-tagging, we built some Bayesian network tools for running MPE for doing POS-tag decoding, but ran out of time; instead, we show POS-tag example results from NLTK. For classification, we examine NaiveBayes model and logistic regression, and compare with results from implementations using ScikitLearn.

1 Introduction

We explore some custom implementations of basic concepts for language-modeling and classification. We use the following datasets: **Language-modeling:** Gutenberg corpus (Austen's 'Emma'); Penn Treebank sample (3000 sentences, 40k words). **POS-tagging:** Penn Treebank sample (3000 sentences, 40k words). **Classification:** *disaster-tweets* dataset (7613 documents with 2 classes); the *imdb movie review* (25k reviews for training, 25k testing).

1.1 Motivation and Goals

With no prior NLP experience, I wanted to build my intuition in NLP by working on some basic concepts from the ground up. For **Language-modeling**, I tried custom implementations of N-gram, sparse embeddings (Term-Document, TFIDF), and dense embeddings (SGNS). For **POS-tagging**, I wanted to try building my own HMM from the ground-up, including building the foundational code for the graphical models, training, and inference. For this part, I completed my own implementation of Factor multiplication and marginalization; wrote most of the MPE (unfinished) but did not time to write the training code for the HMM. I therefore show some results from NLTK's trained taggers as an example of an end result for POS-tagging. For **Classification**, I tried Logistic Regression, Naive Bayes, and Neural models.

Where possible, I compare the result to known implementations (ScikitLearn, Gensim, NLTK, etc).

2 Language Models

2.1 N-gram language model

Language models are "models that assign probabilities to sequences of words" [1]. The simplest type of such model is the n-gram: a sequence of n words. Fundamentally, the n-gram sequence model applies the Markov assumption: that the current word depends only on the previous (n-1) words. The 1-gram model (unigram) thus is like the Bag-of-Words (BoW) model, since both can model a document as a bunch of words without retaining the sequencing between them. As we increase the length of n, the sequence information we store increases: the 2-gram (bigram) thus treats a sequence of words like a 1st-order Markov-chain, where each word is only depend on the previous word.

2.1.1 Implementation Notes

Tokenization has many things to consider that all impact the effectiveness of the vocabulary. Specifically, we pad every sentence with (n-1) start-tokens; replace all periods with an end-token; strip all punctuation; lowercase all symbols.

Sampling: instead of using *backoff* to address sparsity, I randomly sample out-of-band words when the random probability p exceeds the total probably mass of words at each key. Equivalently, this means that if a key (word) is sparse (does not have many co-occurrences with other words), then there is low probability mass and most of the time I will sample randomly from the vocabulary. Thus, if total sparsity is high, then the bigram with random sampling becomes like the unigram instead.

2.1.2 Visualization

A common way to visualize a language model is to generate some sentences from it by sampling from the probability distribution. Here are some sentences sampled from bigram distributions for Austen's Emma.

My bigram implementation:

- <b> I malt insisted summons propose already Warmth restoration precedent grant
- she Will held fatigued workmen influence counteract flew reappearance _sensation_ first
- she union faint been dropt favourite deem cheek wheres Madness frighten

NLTK unigram implementation:

- cannot thing Elton other burn I this change imparted understand
- " without presume Stokes wish be to himself ." of
- probably , going all seem as as ; Harriet also

NLTK bigram implementation:

- **Unavailable** because some (word, prev_word) combinations do not appear.

NLTK everygram(2) implementation:

- must go the sleek , she belonged to each other
- to bring the only those of the smallest apparent sense
- is as for Emma has occurred but still I can

2.1.3 Observations

Robustness of ngrams due to sparsity issue: I handled sparsity issue by automatically sampling uniformly from the out-of-key words; this seems to definitely perform worse than supplementing a bigram with unigram for robustnes (i.e., everygram(2)), comparing the visualizations.

Sampling randomly from out-of-band to address sparsity becomes like unigram: The high sparsity means that my sampling method to address sparsity will sample randomly from out-of-key words most of the time, explaining why my bigram is more like a unigram.

End-token generation: difficulty generating the end-token. There are many words and the probability of generating an end-token was very low. Thus I limited the length artificially.

2.1.4 Conclusions

Increasing robustness (i.e., to sparsity) using unigram, etc supplementation (i.e., everygrams) seems to be very effective. We see that it can already make a simple everygram(2) model look superficially acceptable.

I also believe that my random sampling from the vocabulary for sparsity may be slightly wrong: I equally choose from the remaining vocabulary with $1/|V|$, instead of weighting the frequency of the out-of-band words. I believe that the way supplementation with unigram works, is that it will weight the words such that the choices are not uniform as $1/|V|$ but weighted with their unigram frequencies; I think I would need to look at the NLTK ngram model code to see exactly how they sample their ngram tables to make use of the increased information from the everygrams.

3 Vector Semantics and Embeddings

Given a corpus (collection of documents), we can represent a word in some vector space. By the distributional hypothesis, which postulates that words with similar meanings are close to each other since they will appear *near* each other, word vectors with closer meanings will thus be closer in the vector space.

A *term-document matrix* represents the occurrence-counts of a word in each document as a row-vector. All word-vectors together as rows then comprise the matrix. The columns form document-vectors, representing a document by the occurrence counts of every word in the vocabulary. The transpose of this matrix is the *document-term matrix*, which has the same meaning but just transposed.

A *term-term matrix* is a co-occurrence matrix between all words in a corpus within some window. If the window is the entire-document, then the co-occurrence matrix's cells represent the number of times two words appear in the same document. The window can also be smaller, such as ± 4 words. In the latter case, then the term-term matrix is more appropriately called the *term-context matrix*.

Taking the dot-product between any two word-vectors or document-vectors thus measures similarity. For example, if two word-vectors appear in mostly the same documents, then their dot-product will be quite large. If two document-vectors are composed of mostly the same words, then their dot-product will be quite large. Thus, if X is the *term-document matrix*, then $X^T X$ is the *term-term matrix* representing word-similarities, and XX^T is the *document-document matrix* representing document-similarities.

The above *word-document matrix* or *word-word (co-occurrence) matrix* thus are occurrence counts of words (wrt documents or other words). However, these counts are biased to words that appear commonly, but may not add substantial meaning (e.g., 'the'), and penalizes words that may be very informative (e.g., 'Hiroshima') but appear seldomly. We can normalize these occurrence counts (called $tf(w, d)$) by the number of documents in which a word appears: $N/support(w)$ where $support(w)$ is the number of documents a word appears in (more commonly called $df(w)$). Thus, the normalization of occurrence counts of a word in a document is: $td(w, d) \times \frac{N}{df(w)}$. This is called the *tf-idf score (weight).*

The pointwise mutual information (PMI), encodes the ratio of the probability of two words occurring in the same context versus their independent probability of happening: $PMI = log_2(\frac{P(w,c)}{P(w)*P(c)})$. The $P(w, c)$ are just the co-occurrence counts of the words w and c. $P(w)$ is the probability of the word occurring over all contexts; it is the marginal over contexts. $P(c)$ is the probability of the context word occurring over all target words; it is the marginal over words. If the set of target words (i.e., 'w') and set of context words (i.e., 'c') are equal (i.e., the word-word matrix is square-symmetric), then $P(w)$ and $P(c)$ are the same.

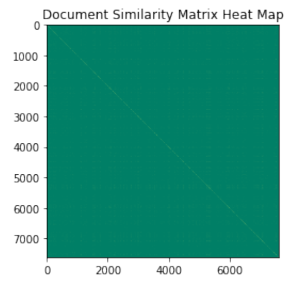
3.1 Implementation

Conversion to Numpy sparse arrays: One issue was that the term-document matrix is quite large, but very sparse. Computing its dot-product thus took a long time; I converted its internal representation to use numpy sparse arrays which made the computations tractable.

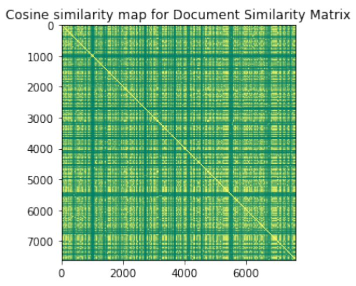
3.2 Visualizations

Figure 1 shows the similarity matrices. The left (a) shows the raw (unnormalized) dot-products. The subsequent two are normalized (cosines). After normalizing, we can see some definite structure in both the document and word similarities, implying there is some information we can exploit for (binary) classification.

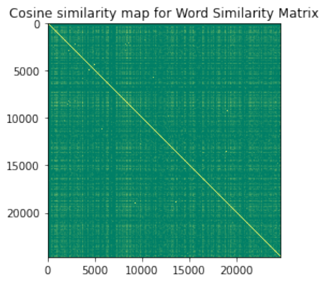
Figure 2 shows that after applying the TFIDF weighting, the document similarities in one class (i.e., yellow) actually decrease, which makes sense given the purpose of TFIDF which will use the inverse document frequency to boost or penalize some words.



(a) Unnormalized Doc-Doc

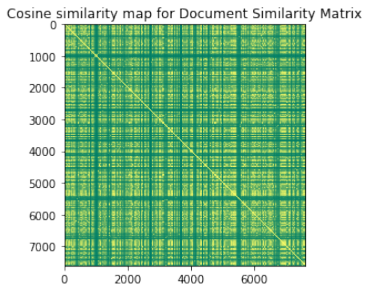


(b) Doc-Doc

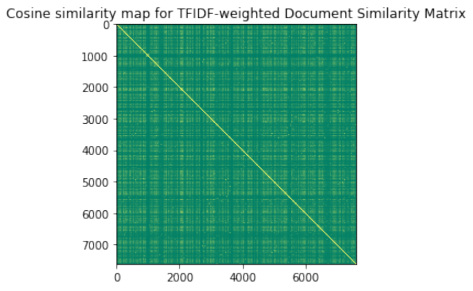


(c) Term-Term

Figure 1: Visualization of Term-Document Similarity matrices. Vocab size: 24k. Doc size: 7k.



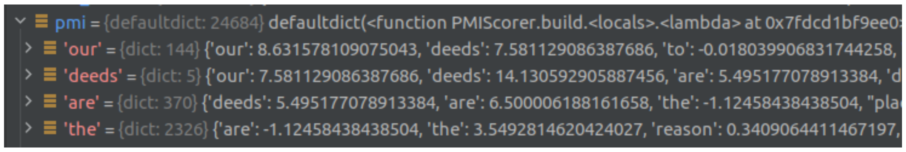
(a) Regular



(b) TFIDF

Figure 2: Comparison visualizations of the Document-Document Similarity matrices for Regular versus TFIDF.

Figure 3 shows a snapshot of the PMIs for a 5-word context window. Notice that they make intuitive sense: the PMI for target-word 'our' and context-word 'to' is negative, meaning their actual co-occurrence is lower than their individual occurrences. Conversely, target-word 'deeds' and context-word 'our' has a very high PMI (≈7). Figure 4 shows the PMI heatmaps for the same 5-word context window. The dataset does not seem to offer enough data to give an informative word-word PMI scores, as seen.

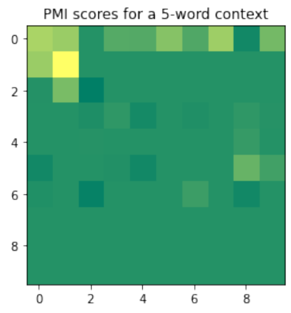


```
pmi = {defaultdict: 24684} defaultdict(<function PMIScorer.build.<locals>.<lambda> at 0x7fdcd1bf9ee0
 'our' = {dict: 144} {'our': 8.631578109075043, 'deeds': 7.581129086387686, 'to': -0.018039906831744258,
 'deeds' = {dict: 5} {'our': 7.581129086387686, 'deeds': 14.130592905887456, 'are': 5.495177078913384, 'd
 'are' = {dict: 370} {'deeds': 5.495177078913384, 'are': 6.500006188161658, 'the': -1.12458438438504, "pla
 'the' = {dict: 2326} {'are': -1.12458438438504, 'the': 3.5492814620424027, 'reason': 0.3409064411467197,
```

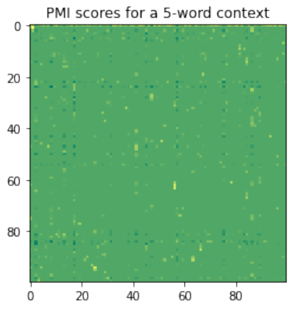
(a) PMI

Figure 3: PMI scores for tweets.

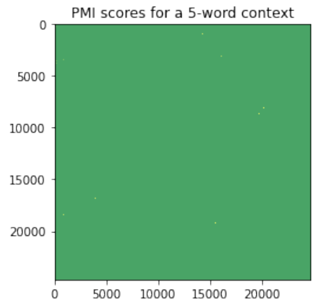
Figure 5 shows the zoomed-in PMIs as context-window is swept from 5-words to 15-words and to 61 words. The PMI information increases, but still not enough to give discernible patterns without zooming-in. An interesting effect is observed for the first 0-20 words, which seem to have a high correlation amongst themselves. This effect occurs because as we increase the context-window, all the first words in the first tweet will be completely cross-correlated with just themselves. Intuitively, we expect this to happen to the second tweet, and so on for the first few tweets, and so expect to see some small block matrices down the diagonal; we don't see this, however, perhaps because even after just one tweet, tweets start becoming cross-correlated in common words.



(a) 10×10

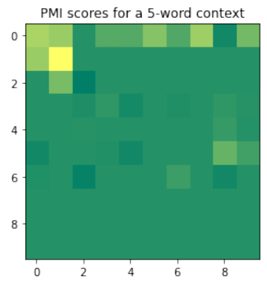


(b) 100×100

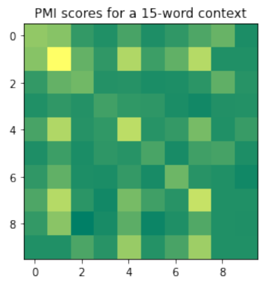


(c) 24k×24k

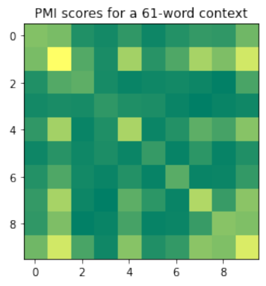
Figure 4: PMI scores for tweets for a 5-word context window.



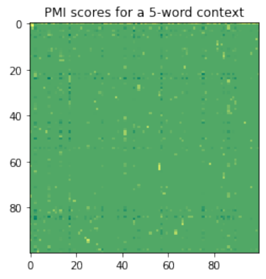
(a) 5-wd ctx,10×10



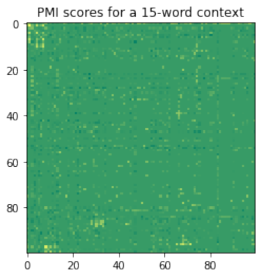
(b) 15-wd ctx,10×10



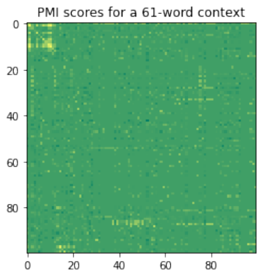
(c) 61-wd ctx,10×10



(d) 5-wd ctx,100×100



(e) 15-wd ctx,100×100



(f) 61-wd ctx,100×100

Figure 5: Zoomed-in PMI scores for tweets for a various word context windows.

3.3 Next steps

Ideally, we'd like to cluster the documents into the two groups. The ordering of the documents (and words) in the above matrices are arbitrary. We could try running clustering algorithms on the documents to partition them into the two classes. We could then assess the effectiveness of the similarity scores by looking at the similarity scores of the documents in the clusters.

4 Dense Embeddings

4.1 SGNS

Skip-Gram with Negative Sampling (SGNS), commonly referred to as word2vec, is a dense embedding. The intuition of the SGNS is to use the weights of a binary-classifier *as a dense embedding*. It turns out, empirically, that dense embeddings perform much better than sparse ones, possibly because they may capture synonymy better [1].

Intuitively, we can view the logistic regression binary classifier in SGNS as arising from changing the task of predicting the *neighboring-word* given an *input word* (e.g., *input-word* "thou", *neighbor-word* "shalt"), to a task that takes as input both the *input-word* and desired (correct) *neighbor-word* (e.g., input is now: ("thou", "shalt")), and then outputs a Probabilistic score to indicate whether they are indeed neighbors (1) or not (0).

Then, the negative sampling technique is applied so that for every input pair of (*input-word*, actual-*neighbor-word*), k negative samples (*input-word*, not-*neighbor-word*) are generated as well.

SGNS uses two weight matrices to represent two dense embeddings: one for *input-words*, $\mathbf{W}$, and one for *context-words*, $\mathbf{C}$. When a word is in the role of the *input-word*, its embedding from $\mathbf{W}$ is used; and when a word is in the role of a *neighbor-word* (context-word), its embedding from $\mathbf{C}$ is used. Thus, both $\mathbf{W}$ and $\mathbf{C}$ represent every word in the vocabulary, and so have $|V|$ rows. The *dense* aspect comes in the sizes of these vectors; they are now d where $d << |V|$. Typically, sizes of d are from 50 to 300.

For training, we take the embeddings (weights) of every pair (input-word, neighbor-word), and calculate the dot-product of the *input-word* embedding against each *neighbor-word* embedding; the sigmoid is applied to get the scores. We know the correct score (either 1 or 0 for positive or negative sample); and so the loss function is known. We take the gradient of this loss function with respect to the three parameters of the two weight matrices: w, c_{pos}, c_{neg}, representing the weights for the *input-word*, correct *neighbor-word*, and incorrect *neighbor-word*, respectively, to update the weights.

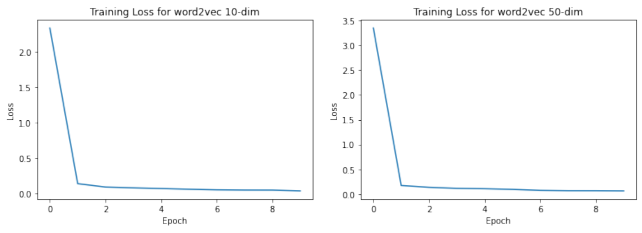
4.1.1 Implementation

We wrote a SGNS class that trains the word2vec embeddings, taking as input the desired embedding size d, half-window-size hws (i.e., number of context-words is $2 \times hws + 1$, k ratio between pos and neg samples, and the input vocabulary. We trained it using gradient descent. The code is in the appendix.

Some implementations use a one-hot encoding of the input center and context words; this is equivalent to a look-up of the word's dense representation (row) in either the $\mathbf{W}$ or $\mathbf{C}$ matrix. We don't do this as we have the index of every word since we created the vocabulary to index mapping.

4.1.2 Visualization

Figure 6 shows the training loss for 10-dim and 50-dim weights.



(a) 10-dim (b) 50-dim

Figure 6: Training loss for custom SGNS implementation on tweets dataset.

Figure 8 shows the word-similarity matrices generated from the trained weights for SGNS for 10 and 50 dimension dense representations. For comparison, the term-term matrix generated from the raw document co-occurrence counts is shown on the left. We can see some similarities, for example, words around $\approx$ 16000 and 22000 in all three matrices show a thick, band. The Gensim SGNS model's word-similarity matrix is on the right, which we take as the correct output. Our model weights has issues in that weights appear to be computed incorrectly; however, due to lack of time, we are not sure why. The code is attached, and we plan to debug it in the future.

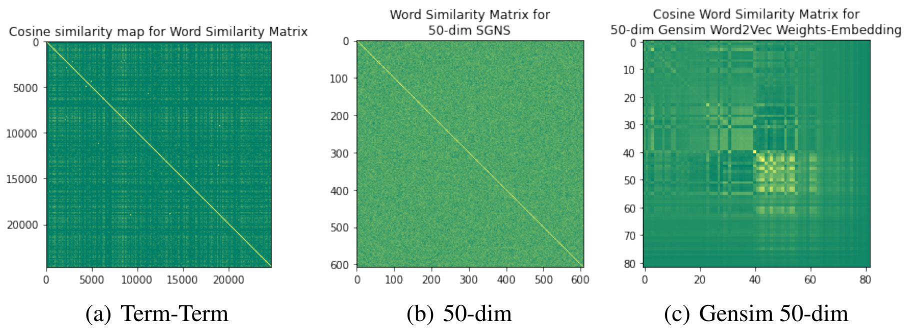


Figure 7: Word-similarity matrices for word2vec dense embeddings of the tweets dataset, compared to the raw word-word similarity matrix (left) and Gensim SGNS (right).

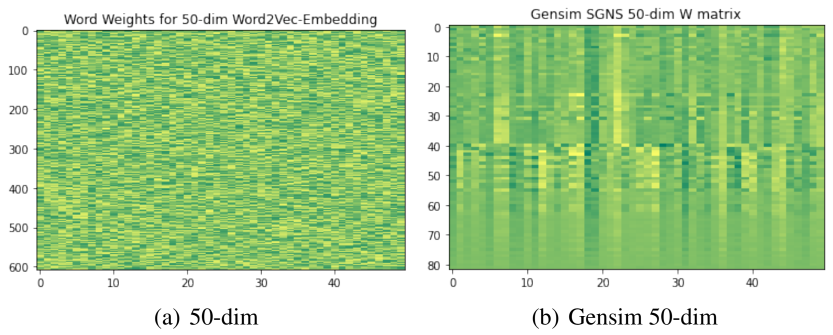


Figure 8: Word embeddings for SGNS of tweets dataset, compared to Gensim SGNS.

5 Part-of-Speech Tagging

Part-of-speech tagging (POS-tagging) is, given a sentence, we want to correctly predict their parts of speech. This is not easy, as there can be multiple choices for the same words; thus when words are in a sequence, the total possibilities then scale exponentially. For example, in the sentence "Fed raises interest rates" [2], every word can be a noun or verb.

5.1 HMM Model

One model for POS-tagging is the Hidden Markov Model (HMM). The HMM is a graphical model composed of a sequence of n hidden state variables $\mathbf{x}$, each hidden state x_i having an output variable y_i that depends on it (thus n output variables in $\mathbf{y}$). The joint probability of the states and observations, $P(\mathbf{y}, \mathbf{x})$, can then be simplified by the 1st order Markov assumptions encoded in the graphical model: 1. all observations are independent of each other: $P(y_i)$ and $P(y_j)$ are independent for $i \neq j$; 2. each observation y_i depends *only* on the hidden state x_i to which it is connected, so that $P(y_i|x_1, x_2, ..., x_n) = P(y_i|x_i)$; 3. each hidden state x_i depends only on its previous hidden state x_{i-1}: $P(x_i|x_{i-1}, ...x_2, x_1) = P(x_i|x_{i-1})$. These simplify the joint probability $P(\mathbf{y}, \mathbf{x})$:

$$\begin{aligned}
P(\mathbf{y},\mathbf{x}) &= P(y_1, y_2, ..., y_n, x_1, x_2, ..., x_n) \\
&= P(\mathbf{y}|\mathbf{x})P(\mathbf{x}) \\
&= P(y_1|\mathbf{x})P(y_2|\mathbf{x})...P(y_n|\mathbf{x})P(\mathbf{x}) \\
&= P(y_1|x_1)P(y_2|x_2)...P(y_n|x_n)P(x_1, x_2, ..., x_n) \\
&= P(y_1|x_1)P(y_2|x_2)...P(y_n|x_n)P(x_n|x_{n-1})P(x_{n-1}|x_{n-2})...P(x_2|x_1)P(x_1) \\
&= \prod_{i=1}^{n} P(y_i|x_i) \prod_{i=2}^{n} P(x_i|x_{i-1}) \;\; P(x_1)
\end{aligned}$$

Then, the probabilities $P(y_i|x_i)$ are called the *emission probabilities*, $P(x_i|x_{i-1})$ are called the *transition probabilities*, and $P(x_1)$ is the *prior probability*. The emission probabilities are the CPTs for the observation nodes (variables) y_i; the transition probabilities are the CPTs for the hidden state nodes (variables) x_i; and the prior probability x_1 is just a (C)PT of the probabilities for each possible state of variable x_1.

Applied to POS-tagging, the observations y_i are the given (observed) words, and the POS-tags assigned to each word are the unobserved, hidden states.

5.1.1 Three Questions that HMM can Answer [2; 3]

The HMM model breaks down the entire joint probability into simple products of probabilities (CPTs of a Bayesian network). But how do we actually use it? We summarize the three main queries [2]:

Below, *path* refers to a sequence of hidden states $\mathbf{x} = x_1...x_n$– i.e., the POS-tags; *sentence* refers to the sequence of observations $\mathbf{y} = y_1, .., y_n$– i.e., the word sequence; *under model M* refers to using/given the parameters of model M (for HMM, the parameters are all CPTs and all prior probabilities).

1. **Scoring**: basically, calculating a probability 'score'.
   (a) **One-path:** when given a path (hidden states) and a sentence (observations), what is the probability of having observed those observations (that sentence)?
   This answers: $P(y_1, ..., y_n, x_1, ..., x_n)$ under model M; i.e., the joint probability of a sentence and a path, given the CPTs (model).
   Algorithm: 'Run' the model: multiply emission and transition probabilities together (since this yields the joint probability as shown earlier).
   (b) **All-paths:** when given just a sentence (observations), what is the probability of having observed this sentence regardless of paths (i.e., across all paths)?
   This answers: $P(y_1, ..., y_n)$ under model M, by marginalizing-out path variables $(x_1, ..., x_n)$ from the joint probability.
   Algorithm: 'Run' the model: multiply emissions and probabilties, then marginalize-out the hidden state $\mathbf{x}$ (i.e., sums over all paths). Or can run *forward algorithm* which does the equivalent.
2. **Decoding (Parsing)**: finding instantiations of hidden-state variables that are most likely.
   (a) **One-path:** when given a sentence (observations), what is the most likely path (hidden states)?
   This answers: $\mathbf{x}^* = argmax_{\mathbf{x}} P(y_1, ...y_n, x_1, .., x_n) = argmax_{\mathbf{x}} P(\mathbf{y}, \mathbf{x})$
   Algorithm: Viterbi decoding, MPE
   (b) **All-paths:** when given a sentence (sequence of observations), what are the most likely states for each of these observations, independent of all other observations?
   This answers: $(x_1^*, ..., x_n^*) = [argmax_k \sum_{\mathbf{x}} P(x_i = k|\mathbf{y})$ for i in $1..n]$
   Intuitively, the above is saying: for each sequence position i, get the hidden state value x_i that yields the highest hidden state probability $P(x_i|\mathbf{y})$; the x_i in the hidden state probability is isolated from the joint probability by summing out all other hidden states x_j.
   Algorithm: Posterior decoding (forwards-backwards once at a single node/variable; equivalent to Posterior marginal on a single hidden state variable).

3. **Learning**: want to fit (find) the CPTs and prior probabilities.
   Denote $\mathbf{\Theta} = (\text{emission}, \text{transmission}, \text{priors})$. Then the goal is to maximize $\mathbf{\Theta}$. *Supervised* means every example (say, observation) comes with the labels (hidden states); *Unsupervised* means only the observations are given, with no labels (unknown hidden states); then, learning in such conditions requires finding the maximizing hidden state probabilties as well.
   (a) One-path:
      i. **Supervised:** given obs. *and* hidden states, find: $\mathbf{\Theta}^* = argmax_{\mathbf{\Theta}} P(\mathbf{y}, \mathbf{x}|\mathbf{\Theta})$
      Algorithm: MLE (count frequencies of emissions, transitions in the observed data)
      ii. **Unsupervised:** given only observations, find: $\mathbf{\Theta}^* = argmax_{\mathbf{\Theta}}\ max_{\mathbf{x}} P(\mathbf{y}, \mathbf{x}|\mathbf{\Theta})$
      Algorithm: Viterbi training, best path.
   (b) All-paths:
      i. **Unsupervised:** given observations only, find: $\Theta^* = argmax_{\Theta} \sum_{\mathbf{x}} P(\mathbf{y}, \mathbf{x}|\Theta)$
      Algorithm: Baum-Welch (E-M).

Thus, to answer question one (Scoring) or two (Decoding), the HMM parameters (namely: the CPTs and prior probabilities) need to be known; they can be learned by answering question three (Learning). To learn, training-data (i.e., given observations and their hidden states) is used to learn these CPTs (emission, transition, and prior probabilities).

Further explanation of Posterior decoding: Succinctly, the Posterior decoding is just the Posterior marginal of a single hidden state variable x_i of the HMM.

Then, the *posterior decoding* is saying: for each observation y_i, we can find the most likely state x_i for it, using all the other y_j we have observed. This breaks down the joint probability into two halves at the position y_i:

$$\begin{aligned}
argmax_k P(x_i = k|\mathbf{y}) &= argmax_k \frac{P(x_i = k, \mathbf{y})}{P(\mathbf{y})} \\
&= argmax_k \frac{P(y_1, ..., y_i, x_i = k) P(y_{i+1}, ...y_N | x_i = k)}{P(\mathbf{y})} \\
&= argmax_k P(y_1, ..., y_i, x_i = k)\ \ argmax_k P(y_{i+1}, ...y_N | x_i = k)
\end{aligned}$$

The latter two maximized terms can be calculated by running the, repsectively, *forward algorithm* and *backward algorithm* once.

The posterior decoding is assigning the most likely state to each hidden state in the sequence, but *independent* of the most likely hidden states we would also calculate for the other positions as well. Intuitively, at each position i, it wants to maximally leverage the known observations from sequence position 1 to i (forwards algorithm) but also from position n 'backwards' to position i (backwards algorithm). The backwards-pass leverages the reversibility of Bayes rule [4] to allow position i to maximize its probability using information that occurred 'after' it (i.e. probability that flows/influences backwards from position n to i). Intuitively, if you are able to know the future outcome of some, say, coin flips, then it would probably influence yor current prediction of the hidden state.

Comparison to Bayesian Network queries and methods [5]:

- **MPE:** *Most Probable Explanation* (MPE) is the maximum posterior (joint) marginal of **all** variables ($N_1, N_2, ..., N_n$), given any evidence. The *MPE instantiation* is the instantiation of all variables corresponding to the MPE. Different algorithms for MPE can be used, such as Variable-Elimination or (Heuristic) search-based. *Variable-elimination MPE* with all evidence variables being the observation variables $y_1, ..., y_n$ in an HMM and variable elimination order $\pi = y_1, x_1, y_2, x_2, ..., y_n, s_n$ *is* the *Viterbi algorithm*[5]. Then, the MPE instantiation of states $x_1, ...x_n$ is the *most probable path* in an HMM [5].
- **MAP:** *Maximum a Posteriori Hypothesis* (MAP) is the maximum posterior marginal for a subset of variables (i.e., with $N_{m+1}, N_{m+2}, ...N_n$ summed-out), given any evidence. The *MAP instantiation* then is the instantiation of this subset of variables corresponding to this MAP score. MAP is like MPE but with some hidden states x_i summed-out (but with the caveat that it is in a worse complexity class).

- **Sum-Product Belief Propagation:** Is used to compute marginals (i.e., probabilities/scores). It uses an algorithmic process of multiplying factors together and summing-out, repeatedly– the main observation being to sum-out early to reduce the exponential increase in factor size as factors are multiplied. Sum-product BP applied to an HMM is thus the *forwards-backwards algorithm*.
- **Max-Product Belief Propagation:** Is essentially Sum-Product BP with the sum (marginalization) over the messages-from-senders replaced with a max. Intuitively, the max now picks the message-sender with highest score rather than computing the combined sums from all message-senders. Applied to an HMM, it *is* the *Viterbi algorithm*.
- **Message-Passing algorithms:** Message-passing refers to the process of multiplying one factor into another factor, i.e., 'passing a message'. If factor A is multiplied into factor B, then A is passing a message to B. Note that A itself could be the product of multiple messages passed to it; and so the messages are passed on in an accumulated fashion from node to node. Sum-Product and Max-Product BP algorithms are thus message-passing algorithms. The *forward* and *backward* algorithms for HMMs are therefore also message-passing algorithms; *forward* referring to messages passed 'forward' from a parent to child and *backward* referring to messages passed 'backward' from child to parent.

5.1.2 Implementation

We planned to implement a direct Viterbi algorithm, and also an MPE with the appropriate Variable-Elimination order, and compared both results to the Viterbi-decoder tagger in NLTK. Unfortunately, I only got as far as implementing general Factor multiplication, sum-out, max-out, and VE_MPE, but did not have time to write the code to train the CPTs from the corpus data in order to run MPE with given evidence variables (observations $\mathbf{y}$). All completed code mentioned just now has been attached below.

5.1.3 Results

Here are results for our graphical model Factor multiplication and marginalization. We were very close to implmenting MPE, since these two operations are basically the most important and difficult parts.

```
factor a:
a
0 0.8000
1 0.2000
factor b:
a b
0 0 1.0000
0 1 0.0000
1 0 0.2000
1 1 0.8000
factor ab:
[['0', '0', '1', '1'], ['0', '1', '0', '1'], ['0.8000', '0.0000', '0.0400', '0.1600']]
a b
0 0 0.8000
0 1 0.0000
1 0 0.0400
1 1 0.1600
factor c:
b c
0 0 1.0000
0 1 0.0000
1 0 0.2000
1 1 0.8000
factor d:
b d
0 0 1.0000
0 1 0.0000
1 0 0.2000
1 1 0.8000
factor cd:
[['0', '0', '0', '0', '1', '1', '1', '1'], ['0', '0', '1', '1', '0', '0', '1', '1'], ['0', '1', '0', '1', '0', '1', '0', '1'], ['1.0000', '0.0000', '0.0000', '
b c d
0 0 0 1.0000
0 0 1 0.0000
0 1 0 0.0000
0 1 1 0.0000
1 0 0 0.0400
1 0 1 0.1600
1 1 0 0.1600
1 1 1 0.6400
factor cd with cd marginalized out:
```

```
b
0 1.0000
1 1.0000
```

The following are some results for the default tagger in NLTK (averaged-perceptron-tagger) versus the NLTK hmm-tagger.

Sentence: 'And now for something completely different'
avg-perceptron-tagger:
('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')
hmm-tagger:
('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')

The two taggers give the same results. However, for a longer sentence, there is some difference:

Sentence: 'This is supposed to be a super duper long and hard sentence that will create different results in both taggers.'
avg-perceptron-tagger:
('This', 'DT'), ('is', 'VBZ'), ('supposed', 'VBN'), ('to', 'TO'), ('be', 'VB'), ('a', 'DT'), ('super', 'JJ'), ('duper', 'NN'), ('long', 'RB'), ('and', 'CC'), ('hard', 'JJ'), ('sentence', 'NN'), ('that', 'WDT'), ('will', 'MD'), ('create', 'VB'), ('different', 'JJ'), ('results', 'NNS'), ('in', 'IN'), ('both', 'DT'), ('taggers', 'NNS'), ('.', '.')
hmm-tagger:
('This', 'DT'), ('is', 'VBZ'), ('supposed', 'NNP'), ('to', 'NNP'), ('be', 'NNP'), ('a', 'NNP'), ('super', 'NNP'), ('duper', 'NNP'), ('long', 'NNP'), ('and', 'NNP'), ('hard', 'NNP'), ('sentence', 'NNP'), ('that', 'NNP'), ('will', 'NNP'), ('create', 'NNP'), ('different', 'NNP'), ('results', 'NNP'), ('in', 'NNP'), ('both', 'NNP'), ('taggers.', 'NNP')

Notice that after the first 2 words, the POS-tags become completely different.

5.1.4 Conclusions

It is hard to assess just how differently the above two parses / POS-taggings are, without being able to visualize the parse-tree. For example, from lecture, even a small sentence with a parse difference of 'with chopsticks' being coupled to 'sushi' or 'eat' can result in drastically different meaning (and, parse-tree). Here, it maybe be just one small initial difference in a POS-tag early-on, that lead to different subsequent parses.

However, the take-away is that POS-tagging is certainly a difficult task, as even the built-in NLTK taggers performly very differently on even a very simple, regular sentence.

6 Classification

This section explains the binary classifiers I built and compares their results against ScikitLearn.

6.1 Naive Bayes

The Naive Bayes is a graphical model which assumes that every word is conditioned only on a single variable, the class to which the word belongs. Thus, conditioned on the class, every word becomes independent (by d-separation), and so the probability of a document (tweet) being 'disaster' or 'not disaster' is just the multiplication of all the probabilities of the words in that document, conditioned on each class. The class with the higher probability will be chosen as the predicted class for that document.

	My own NB	Scikit Multinomial NB
Test Acc	46%	80.6%

Table 1: Comparison of custom NB versus ScikitLearn

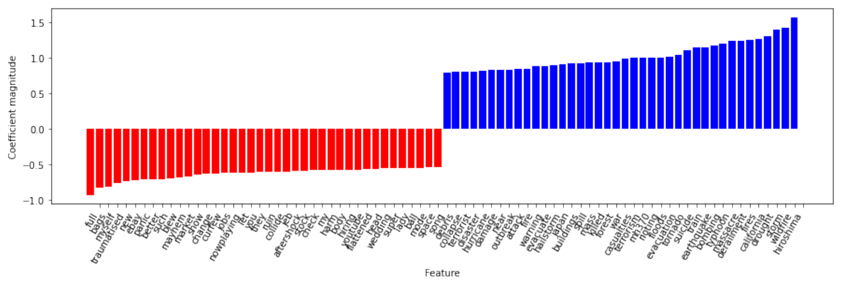


Figure 9: Most influential words: logistic regression coeffs.

Clearly, my implementation has an error, since the prediction is no different than random. I attempted to debug it, but I could not find the error in time. The source code is appended at the end of this document, for reference.

6.2 Logistic Regression

Logistic regression can be viewed as a standard linear regression ($Wx + b$) (i.e., a weighted sum of features) but passed through a sigmoid (logistic) function to create a probability score (0,1) from the weighted sum [6; 7],. The sigmoid (logistic) function can be understood as creating an "S shaped" curve such that, when plotted against its input ($Wx + b$), becomes linear (with outliers squashed to 0 or 1)– allowing us to predict on it. The logit function is the inverse sigmoid and has the form $logit(p) = log\frac{p}{1-p}$, mapping its input (a probability p) to log odds. Its inverse, the sigmoid (logistic), thus maps from the log-odds back into the original input (p), a Pr score: we can solve this equation $logit(p) = log\frac{p}{1-p} = Wx + b$ for p, to get the logistic: $p = P(Y = 1) = \frac{exp(Wx+b)}{1+exp(Wx+b)}$.

Thus the model assumes we have modeled the log-odds using predictors $Wx + b$, and uses the sigmoid to map (invert) these log-odds into (i.e., solve for) probability p. The coefficients of a logistic regression can then be interpreted as: for a one unit increase in that particular input component (component of input vector x), how much does it change the log-odds? The percentage change in the odds is exp(coeff). For example, if the coefficient (weight) of x_2 is 0.12, then for a one unit increase in that input component x_2, we will see a $exp(0.12) = 1.13$ or 13% increase in the odds, holding all other input components constant.

I did not do my own implementation of logistic regression. I applied ScikitLearn's logistic regression to classify the tweets, and got the following result:

	Scikit Logistic Regression
Test Acc	80.3%

Table 2: ScikitLearn Logistic Regression

The result makes sense; it is almost the same as Naive Bayes. Both of these classifiers intuitively should not be that much better or worse than the other.

The following figure shows the logistic regression coeffs for the most influential words.

So, Hiroshima is the most likely indicator for a true disaster, with a weight of 1.6, corresponding to $exp(1.6) = 4.95$ times increase in the odds of a tweet being truly about a disaster, if the tweet contains "Hiroshima" versus if it doesn't contain "Hiroshima". At the other end, the word "traumatised" leads to a decrease in the odds of the tweet being about a real disaster: this also makes sense, as people use this word alot in other contexts.

7 Conclusion

For language modeling, we compared results using a poor dataset meant only for classification (disaster-tweets) versus a more suitable language-modeling corpus such as Austen's 'Emma'. Our custom implementation of n-gram model was a great learning experience because it forced us to consider why the result was different than NLTK; in particular, we hypothesize that our method of doing out-of-band sampling is not fully correct, resulting in our bigram implementation being more closer to unigram. Indeed, it seems like increasing the robustness using unigram (i.e., everygram) is very performant.

For sparse embeddings, we used the disaster-tweets datasets, seeking to examine the document similarities to gain intuition on how it will classify. We saw a visual effect of TFIDF in making some scores less extreme. PMI-matrices for a small corpus intended for classification, not modeling, resulted in very sparse signals.

For the SGNS dense-embeddings, we attempted our own implementation. We are not sure on how correct it is. We can consider doing further low-dimensional PCA comparison with Gensim's SGNS, as well as incorporating the dense embedding into a simple neural model for another task (classification, tagging, etc) to see how well it works.

For POS-tagging, we reviewed fundamental understandings of HMM model, and compared it with general methods for Bayesian networks. We implemented custom implementation of Factor multiplication and marginalization (tested to be working), and Variable-Elimination MPE, but did not have enough time to fully test the VE_MPE to see if it can decode similarly to NLTK's hmm-tagger. Another issue was that I ran out of time to write the code to parse a corpus and train the HMM to initialize its CPTs.

For classification, I attempted a custom NaiveBayes model. I used the logistic regression and NaiveBayes models from SkLearn. While my own NaiveBayes model requires further debugging (it did not work properly), the ScikitLearn results for NaiveBayes and LogisticRegression show that the *disaster-tweets* dataset can be classified to a test-accuracy of roughy 80% using just these simple binary classifiers.

Finally, all attempts at my own implementations took a lot of effort for each one (even if some were unsuccessful). The codes are all fully attached and can be easily run to see the results.

References

[1] D. Jurafsky, *Speech & language processing*. 2022.

[2] K. Chang, *CS263 Lectures, UCLA*. 2022.

[3] user34790, Morat, and F. Dernoncourt, "What is the difference between the forward-backward and viterbi algorithms?." Cross Validated Stack Exchange, 2012. [Online:] https://stats.stackexchange.com/questions/31746/what-is-the-difference-between-the-forward-backward-and-viterbi-algorithms, url = https://stats.stackexchange.com/questions/31746/what-is-the-difference-between-the-forward-backward-and-viterbi-algorithms, urldate = 2012-07-06.

[4] C. Mavroforakis, "Lecture 08: Hmms ii- posterior decoding and learning," December 2012.

[5] P. A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. USA: Cambridge University Press, 1st ed., 2009.

[6] Wikipedia, "Logistic regression — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Logistic%20regression&oldid=1090775162, 2022. [Online; accessed 17-June-2022].

[7] Wikipedia, "Multinomial logistic regression — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Multinomial%20logistic%20regression&oldid=1040245748, 2022. [Online; accessed 17-June-2022].

Appendix: Code

All the code below is written by myself (Nurrachman Liu), except for:

- plotting code for ScikitLearn Logistic Regression coefficients (attribution: https://amueller.github.io/aml/05-advanced-topics/13-text-data.html)

Language Modeling

NGram

```
import re
from io import StringIO
from collections import defaultdict

from DefaultDictUtils import DefaultDictUtils

"""
@author rliu 2022/05
"""


class NGram:
    """
    Implementation notes:
    - Special tokens:  begin: <b>, end: <e>
    - Only 1 end-marker representation (<e>); ellipsis are treated as single periods (ie, a sentence end marker)
    - All other punctuation is ignored (stripped)
    - All words are lower-cased for simplicity (see:  https://stackoverflow.com/questions/45855160/nlp-when-to-lowercase-text-during-preprocessing)
    """

    BEG_TOKEN = " <b> "
    END_TOKEN = " <e> "

    def __init__(self, n=2):
        self.n = n
        # for each (n-1)-gram, record a count of the nth-word
        self.ngrams = defaultdict(lambda: defaultdict(lambda: 0))

    def strip_punctuation(self, str):
        """
        Removes all punctuation from the line.
        """
        # to do it properly, do this: (but requires installing package 'regex')
        # https://stackoverflow.com/questions/58986617/python-3-regex-remove-all-punctuation-except-special-word-pattern

        # custom hack: replace all special tokens with ZZZZZ, strip all punctuation, then put the special tokens back
        UNIQTOK = 'ZA109X930ZB'
        str2 = str
        SPECIAL_TOKENS = [NGram.BEG_TOKEN, NGram.END_TOKEN]
        for SP_TOK in SPECIAL_TOKENS:
            str2 = re.sub(r'' + SP_TOK, UNIQTOK + SP_TOK, str2)
        str2 = re.sub(r'[^\w\s]', '', str2)
        for SP_TOK in SPECIAL_TOKENS:
            str2 = re.sub(r'' + UNIQTOK + re.sub(r'[^\w\s]', '', SP_TOK), SP_TOK, str2)
        return str2

    def replace_periods(self, str):
        """
        Replaces any period or ellipsis with <e><b><b>. Thus, we treat ellipsis as a period.
        """
        return re.sub(r'[.]+',
                      r'' + NGram.END_TOKEN + (NGram.BEG_TOKEN * (self.n - 1)),
                      str)

    def remove_double_begins(self, str):
        """
        Removes <b><b> when there is nothing after it except possibly whitespace or punctuation.
        """
        return re.sub(r'' + (NGram.BEG_TOKEN * (self.n - 1)) + r'\s*$',
                      r'',
                      str)

    def clean_and_add_special_tokens(self, str):
        """
        ONLY adds the special tokens BEG and END to periods found, but does not touch the beginning or end of the line.
        This handles correctly the cases when newline's can appear randomly in a corpus without necessarily being tied
        to the end of a line (the usual case).
        """
        str2 = str
        str2 = str2.lower()
        str2 = self.replace_periods(str2)
        str2 = self.remove_double_begins(str2)
        str2 = self.strip_punctuation(str2)
        return str2

    def word_tokenize(self, str):
        """
        Tokenizes by removing any whitespace characters.
        """
        return re.findall(r'\S+', str)

    def read(self, corpus):
        """
        Reads the entire corpus into an n-gram window.
        """

        self.toks = [] # store the entire token stream for debugging

        # file-like stream over the text
        cont = []  # the previously parsed (n-1) tokens, for continuation.
        corpus_io = StringIO(corpus)
        for i, line in enumerate(corpus_io):
            line2 = re.sub(r'https?://\S+', '', line) # remove urls (http://..) as they flood the vocab with useless words
            line2 = self.clean_and_add_special_tokens(line2)
            toks = self.word_tokenize(line2)

            # at the beginning of every \n-delimited line, add BEG
            toks = [NGram.BEG_TOKEN.strip()] * (self.n - 1) + toks

            self.toks += toks

            toks = cont + toks
            cont = toks[-(self.n - 1):]   # get last n elements

            # form a list of the overlapping windows:
            # https://stackoverflow.com/questions/38151445/iterate-over-n-successive-elements-of-list-with-overlapping
            toks2 = [tuple(toks[i:i+self.n]) for i in range(len(toks)-self.n+1)]
            for ngr in toks2:
                # index into the (n-1)-word by the nth word and incr by 1
                self.ngrams[ngr[:-1]][ngr[-1:]] += 1

    def pprint(self):
        return DefaultDictUtils.pformat(self.ngrams)
```

SampleNGram

```
from ComputeNGramProbs import ComputeNGramProbs
from NGram import *
import numpy as np

"""
@author rliu 2022/05
"""


class SampleNGram:
    """
    Samples sentences from a n-gram language model.
    """

    def __init__(self, ngram):
        self.ngram = ngram

    def sample(self):
        """
        Samples a sentence, beginning with (n-1) start-tokens. For the unigram case,
        there are no start tokens (it passes in []).
        """
        tok = None
        gen = []
        prev = [NGram.BEG_TOKEN.strip()] * (self.ngram.n - 1)
        while tok != NGram.END_TOKEN.strip() and tok != NGram.BEG_TOKEN.strip():
            tok = self._sample(prev)
            gen.append(tok)
            # shift prev to the left
            prev.pop(0)
            prev.append(tok)
            if len(gen) > 10:
                break
        return gen

    def _sample(self, prev):
        probs, mars, vocab = ComputeNGramProbs.compute_pr_addone(self.ngram)

        # restrict the sample space (ie, conditional pr space) to the set of prev
        pr = probs
        if prev:
            pr = pr[tuple(prev)]   # keys of probs are tuples (ex: ('a', 'b')), since list keys are not allowed.

        pr_keys = set()
        for tup_key in pr.keys():
            pr_keys.update(list(map(str, tup_key)))

        out_of_key_words = list(vocab - pr_keys)

        p = np.random.uniform()
        if p <= sum(pr.values()):
            cdf = ComputeNGramProbs.make_cdf(pr)
            for wd, accum_pr in cdf.items():
                if p < accum_pr:
                    tok = wd[0]
                    break
        else:
            p1 = p - sum(pr.values())       # left-over pr mass
            p2 = p1 / (1 - sum(pr.values()))  # normalize by pr mass of out-of-key words
            idx = int(np.floor(p2 * len(out_of_key_words)))
            tok = out_of_key_words[idx]

        return tok


if __name__ == "__main__":
    import pandas as pd

    bigram = NGram()
    df = pd.read_csv('../data/disaster-tweets/train.csv')

    for doc in df['text']:
        bigram.read(doc + ".")

    bigram_sampler = SampleNGram(bigram)

    for _ in range(10):
        sentence = bigram_sampler.sample()
        print(' '.join(sentence))
```

ComputeNGramProbs

```
from collections import defaultdict
from collections import OrderedDict

"""
@author rliu 2022/05
"""


class ComputeNGramProbs:
    """
    Computes the NGram probabilities.
    """

    def __init__(self):
        pass

    @classmethod
    def compute_pr(cls, ngram):
        """
        Computes the probabilities without any smoothing.
        """
        marginals = {}
        for pre, wd_cnts in ngram.ngrams.items():
            marginals[pre] = sum(wd_cnts.values())

        probs = defaultdict(lambda: {})
        for pre, wd_cnts in ngram.ngrams.items():
            for wd, cnt in wd_cnts.items():
                probs[pre][wd] = cnt / marginals[pre]

        return probs, marginals

    @classmethod
    def compute_pr_addone(cls, ngram):
        """
        Computes the probabilities using add-one smoothing.
        The simplest way to implement add-1 smoothing is to just add 1 to every count across the entire ngram.
        This thus also handles non-unigram's properly, after normalizing. However, we must also assign a
        default pr of 1/|V| for the words in the vocab that don't appear in the prefix (for non-unigram's);
        for unigram, the default pr of 1/|V| also applies, but the prefix is empty.
        """

        vocab = set()
        for pre in ngram.ngrams:
            vocab.update(pre)

        marginals = {}
        for pre, wd_cnts in ngram.ngrams.items():
            marginals[pre] = sum(wd_cnts.values())

        probs = {}
        for pre, wd_cnts in ngram.ngrams.items():
            if pre not in probs:
                # assign the default 1/|V| pr for all words conditioned on the prefix 'pre'
                probs[pre] = defaultdict(lambda: 1/(marginals[pre]+len(vocab)))
            for wd, cnt in wd_cnts.items():
                probs[pre][wd] = (cnt + 1) / (marginals[pre]+len(vocab))

        return probs, marginals, vocab

    @classmethod
    def make_cdf(cls, probs):
        """
        Generates an interval along 0-1 for all the probabilities, such that they can be sampled
        with a uniform number in (0,1).
        """
        accum = None
        od = OrderedDict()
        for wd, pr in sorted(probs.items()):
            if accum is None:
                accum = pr
            else:
                accum += pr
            od[wd] = accum

        return od
```

Juptyer Notebook: ngrams.ipync

```
## n-gram models
@author rliu 2022-05
#%% md
### Setup
#%%
#%load_ext autoreload
#%autoreload 2
#%%
import pandas as pd
from functools import partial
from itertools import chain
flatten = chain.from_iterable

import nltk
from nltk.corpus import gutenberg
from nltk.util import bigrams
from nltk.util import ngrams
from nltk.util import pad_sequence
from nltk.lm.preprocessing import pad_both_ends
from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.lm import MLE

from NGram import NGram
from SampleNGram import SampleNGram

#%% md
### Load datasets
#%%
nltk.download('gutenberg')
files = gutenberg.fileids()
print(files)
#%%
GUTEN_DOC = 'austen-emma.txt'
# the document is nicely parsed and tokenized for us

# read it into a list for convenience
guten_sents = []
for sent in gutenberg.sents(GUTEN_DOC):
    guten_sents.append(sent)
print(len(guten_sents))
print(guten_sents[0:2])
#%%
tweet_df = pd.read_csv('../data/disaster-tweets/train.csv')
tweet_df['text'].head(5)
#%% md
### Build lm's
#%% md
#### My Model: Tweets dataset
#%%
bigram = NGram()
df = pd.read_csv('../data/disaster-tweets/train.csv')

for doc in df['text']:
    bigram.tokenize_raw(doc + ".")  # append period separator to the end of every tweet.
bigram.build_ngram()

bigram_sampler = SampleNGram(bigram)

for _ in range(2):
    sent = bigram_sampler.sample()
    print(' '.join(sent))
    sent = bigram_sampler.sample('the')
    print(' '.join(sent))
    sent = bigram_sampler.sample('he')
    print(' '.join(sent))
    sent = bigram_sampler.sample('she')
    print(' '.join(sent))

#%% md
#### My Model: Austen "Emma"
#%%
bigram = NGram()

for sent in gutenberg.sents(GUTEN_DOC):
    sent.insert(0, NGram.BEG_TOKEN.strip())  # insert BEG token at the start of every sent
    sent = [NGram.END_TOKEN.strip() if x == '.' else x for x in sent]  # replace . with END token
    bigram.toks += sent
bigram.build_ngram()

bigram_sampler = SampleNGram(bigram)

for _ in range(3):
    sent = bigram_sampler.sample()
    print(' '.join(sent))
    sent = bigram_sampler.sample('the')
    print(' '.join(sent))
    sent = bigram_sampler.sample('he')
    print(' '.join(sent))
    sent = bigram_sampler.sample('she')
    print(' '.join(sent))

#%% md
#### NLTK Model: Austen "Emma"

#%%
print(list(bigrams(gutenberg.words(GUTEN_DOC)))[0:5])
print(list(bigrams(gutenberg.sents(GUTEN_DOC)))[0:3])
#%% md
Let's pad the bigrams:
#%%
sents = pad_sequence(gutenberg.sents(GUTEN_DOC),
                     pad_left=True, left_pad_symbol='<b>',
                     pad_right=True, right_pad_symbol='<e>', n=2)
print(list(sents)[5:8])


#s0 = pad_sequence(guten_sents[0], pad_left=True, left_pad_symbol='<b>', pad_right=True,
## right_pad_symbol='<e>', n=2)#
#s1 = pad_sequence(guten_sents[1], pad_left=True, left_pad_symbol='<b>', pad_right=True, # right_pad_symbol='<e>', n=2)
##
#print(list(s0))
#print(list(s1))

#%%
padding_fn = partial(pad_both_ends, n=2)
train = [bigrams(list(padding_fn(sent))) for sent in gutenberg.sents(GUTEN_DOC)]
vocab = flatten(map(padding_fn, gutenberg.sents(GUTEN_DOC)))
# note: upon reading from train, it is consumed. we will have to call this later.
out = []
for x in list(train):
    out.append(list(x))
print(out[0:2])
print(list(vocab)[0:15])
#%%
padding_fn = partial(pad_both_ends, n=2)
train = [bigrams(list(padding_fn(sent))) for sent in gutenberg.sents(GUTEN_DOC)]
vocab = flatten(map(padding_fn, gutenberg.sents(GUTEN_DOC)))
bigram_lm = MLE(2)
bigram_lm.fit(train, vocab)
print(bigram_lm.vocab)
len(bigram_lm.vocab)

#%%
for _ in range(10):
    print(' '.join(bigram_lm.generate(10)))

#%% md
It seems that there are some robustness problems; it will try to generate a word given some
previously generated word, but that sequence may not have occurred.
#%%
padding_fn = partial(pad_both_ends, n=1)
train_unigram = [ngrams(list(padding_fn(sent)), n=1) for sent in gutenberg.sents(GUTEN_DOC)]
vocab_unigram = flatten(map(padding_fn, gutenberg.sents(GUTEN_DOC)))
unigram_lm = MLE(2)
unigram_lm.fit(train_unigram, vocab_unigram)
print(unigram_lm.vocab)
len(unigram_lm.vocab)

#%%
for _ in range(10):
    print(' '.join(unigram_lm.generate(10)))

#%% md
##### padded everygram
#%% md
Padded everywhere is equivalent to applying a padding of 2, and then doing 1-gram, 2-gram, ...
n-gram for you.
#%% md
First, what do padded everywhere-grams look like?
(Once we materialize it into list, it cannot be used to train.)
#%%
train, vocab = padded_everygram_pipeline(2, gutenberg.sents(GUTEN_DOC))
train_cache = []
for x in train:
    train_cache.append(list(x))
print(train_cache[0:3])
print(list(vocab)[0:20])  # vocab is equivalent to:   list(flatten(pad_both_ends(sent, n=2) for sent in text))

#%% md
```

```
Notice how the everygram first padded the sentences, then applied unigrams and bigrams to them.
Every token in the padded sentence (including the pads of course) are listed with their unigrams
first, then their 2-grams.
#%%
train, vocab = padded_everygram_pipeline(2, gutenberg.sents(GUTEN_DOC))
lm = MLE(2)
lm.fit(train, vocab)
print(lm.vocab)
len(lm.vocab)
#%%

for _ in range(10):
    print(' '.join(lm.generate(10)))
```

Sparse Embeddings

CoccurMatrix

```
from collections import defaultdict

"""
@author rliu 2022/05
"""


class CoccurMatrix:
    """
    Computes the co-occurrence matrix between words, given a specified context window.
    """

    def __init__(self, vocab, half_window_size):
        self.vocab = vocab
        self.half_window_size = half_window_size

    def build(self, docs):
        coccur = defaultdict(lambda: defaultdict(lambda: 0))

        # todo: there's a bug here with the window initialization: see the way i do it in SGNS
        window = [None] * (self.half_window_size * 2 + 1)    # [half_window_size] wd [half_window_size]
        cidx = int(len(window)/2)
        for doc in docs:
            for s in (doc.split() + [None]*self.half_window_size):  # todo: probably should pre-tokenize the docs
                window = window[1:] + [s]
                cwd = window[cidx]
                if cwd is not None:
                    for wd in window:
                        if wd is not None:
                            coccur[self.vocab[cwd]][self.vocab[wd]] += 1

        return coccur
```

DocTermMatrix

```
from TextFormatter import TextFormatter


"""
@author rliu 2022/05
"""


class DocTermMatrix:
    """
    Represents a Document-Term matrix, using non-sparse representation.
    The Document-Term matrix can also be viewed as a co-occurrence matrix where the context window
    is the entire document.

    The input documents are expected to be Pandas series.
    Notationally, the first symbol is the rows of the matrix; so Document-Term has documents as the
    rows; for Term-Document, we take the transpose, where the rows are the Terms now.
    """

    def __init__(self, docs):
        self.docs = docs

    def build_vocab(self):
        """
        Builds the vocabulary out of the given documents.
        """
        all_text = self.docs.tolist()
        vocab = {}   # >= 3.7, dicts maintain *insertion order*. using this as an ordered set.
        for doc in all_text:
            for s in doc.split():
                if s not in vocab:
                    vocab[s] = len(vocab.keys())

        self.vocab = vocab
    def build_full(self):
        """
        Builds a full term-document matrix.
        """
        docterm = [[0] * len(self.vocab.keys()) for _ in self.docs]

        for i, doc in enumerate(self.docs):
            for j, wd in enumerate(doc.split()):
                docterm[i][self.vocab[wd]] += 1

        # store as np.array
        self.full = np.array([np.array(docvec) for docvec in docterm])


    def build_csr(self):
        """
        Builds a sparse term-document matrix.
        """
        # csr handles repeats. So if in the same row-vector, a same index appears, it will count that.
        indptr = [0]   # points to indexes in indices that indicate start of new row in csr_array
        indices = []  # index in the row vector for each corresponding data
        data = []

        for i, doc in enumerate(self.docs):
            for j, wd in enumerate(doc.split()):
                idx = self.vocab[wd]
                indices.append(idx)
                data.append(1)  # count as 1 occurrence; multiple occurrences will map via same idx, then sums to wd cnt
            indptr.append(len(indices))  # advance row-vector ptr past current indices, to next row-vector

        self.csr = csr_array((data, indices, indptr), dtype=int)
        return data, indices, indptr


if __name__ == "__main__":
    df = pd.read_csv('../data/disaster-tweets/train.csv')

    tf = TextFormatter()
    df['text'] = [tf.format_text(x) for x in df['text']]

    dt = DocTermMatrix(df['text'])

    dt.build_vocab()
    dt.build_full()
    dt.build_csr()

    # verify if the two representations are exactly equal:
    # print('Full and CSR matrices are equal: %s' % np.array_equal(dt.csr.toarray(), np.array(dt.full)))

    pass
```

PMIScorer

```
import numpy as np
from collections import defaultdict

"""
@author rliu 2022/05
"""


class PMIScorer:
    """
    This class ascribes PMI scores to a given co-occurrence matrix.
    """

    def __init__(self):
        pass

    def build(self, coccur_matrix):
        """
        Given a co-occurrence matrix (i.e., term-term either windowed or full-document), returns the
        PMI score between every word.
        """

        # Compute the occurrences of a word in every (marginalized-over) context.
        context_cnt = defaultdict(lambda: 0)
        for wd1, occurs in coccur_matrix.items():
            for wd2, cnt in occurs.items():
                context_cnt[wd2] += cnt

        # Compute the word counts: this is the number of times the word appears in all contexts
        word_cnt = {}
        for wd1, occurs in coccur_matrix.items():
            word_cnt[wd1] = sum(occurs.values())

        tot = 0
        for _, occurs in coccur_matrix.items():
            for _, cnt in occurs.items():
                tot += cnt
        pmi_matrix = defaultdict(lambda: {})
        for wd1, occurs in coccur_matrix.items():
            for wd2, cnt in occurs.items():
                pmi = (cnt/tot) / ((word_cnt[wd1]/tot) * (context_cnt[wd2]/tot))
                pmi_matrix[wd1][wd2] = np.log2(pmi)

        return pmi_matrix, word_cnt, context_cnt, tot
```

PMIScorer

```
import numpy as np
from scipy.sparse import csr_array
from scipy.sparse import lil_matrix
from collections import defaultdict

"""
@author rliu 2022/05
"""


class TFIDFScorer:
    """
    Accepts a corpus and vocabulary, and creates the td-idf scores for the vocabulary within
    that corpus. The vocabulary is separate from the corpus to allow for different vocabs.
    """

    def __init__(self, corpus, vocab):
        """
        :param corpus: a pandas Series of the documents in the corpus.
        :param vocab: the assigned mapping of the words in 'vocab' to integer indices.
        """
        self.corpus = corpus
        self.vocab = vocab

    def build(self):
        """
        Builds the tf-idf scores (support counts) of all words of the vocab, within the corpus.
        """

        support = defaultdict(lambda: 0)
        for doc in self.corpus:
            seen = {}
            for wd in doc.split():
                if wd not in seen:
                    support[wd] += 1
                    seen[wd] = 1

        N = len(self.corpus)
        idf = {}
        for wd, supp in support.items():
            idf[wd] = np.log10(N / supp)

        self.df = support
        self.idf = idf

    def weight(self, termdoc_full):
        """
        Accepts a raw word-document matrix and returns the TF-IDF weighted matrix.
        :param termdoc_full:  the term-document representation as a full (non-sparse) np array/matrix.
        """
        inv_vocab = {v: k for k, v in self.vocab.items()}

        tfidf = []
        for wd_idx, docvec in enumerate(termdoc_full):
            wd = inv_vocab[wd_idx]
            idf = self.idf[wd]
            docvec2 = docvec * idf    # apply idf-weighting
            tfidf.append(docvec2)

        return np.array(tfidf)

    def weight2(self, termdoc_csr):
        """
        Accepts a raw word-document matrix and returns the TF-IDF weighted matrix.
        :param termdoc_csr:  the term-document representation as sparse np array/matrix.
        """
        inv_vocab = {v: k for k, v in self.vocab.items()}

#        termdoc_csr.sort_indices()

#        termdoc_lil = lil_matrix(termdoc_csr.shape)
#        for wd_idx in range(0, termdoc_csr.shape[0]):
#            docvec = termdoc_csr.getrow(wd_idx)
#            wd = inv_vocab[wd_idx]
#            idf = self.idf[wd]
#            termdoc_lil[wd_idx] = docvec * idf   # apply idf-weighting
#
```

```
#        return termdoc_lil.tocsr()

        data, indices, indptr = np.array([]), np.array([]), np.array([0])
        for wd_idx in range(0, termdoc_csr.shape[0]):
            docvec = termdoc_csr.getrow(wd_idx)
            wd = inv_vocab[wd_idx]
            idf = self.idf[wd]
            docvec2 = docvec * idf    # apply idf-weighting
            data = np.concatenate((data, docvec2.data))
            indices = np.concatenate((indices, docvec2.indices))
            indptr = np.concatenate((indptr, [len(indices)]))

        return csr_array((data, indices, indptr))


if __name__ == "__main__":

    import pandas as pd
    import numpy as np
    from DocTermMatrix import DocTermMatrix
    from TextFormatter import TextFormatter
    from sklearn.metrics.pairwise import cosine_similarity
    from timeit import timeit
    from itertools import islice

    df = pd.read_csv('../data/disaster-tweets/train.csv')
    tf = TextFormatter()
    df['text'] = [tf.format_text(x) for x in df['text']]
    dt = DocTermMatrix(df['text'])

    dt.build_vocab()
    dt.build_full()
    data, indices, indptr = dt.build_csr()
    for i in enumerate(dt.csr):
        pass
    tfidf_scorer = TFIDFScorer(df['text'], dt.vocab)
    tfidf_scorer.build()
    print(dict(islice(tfidf_scorer.idf.items(), 0, 5)))

    # Compute the tfidf-weighted td's using the full td representation.
    td_full = dt.csr.T.toarray()
    td_tfidf_full = tfidf_scorer.weight(td_full)

    # Compute the tfidf-weighted td's using the sparse td representation.
    td_csr = dt.csr.T
    td_tfidf_csr = tfidf_scorer.weight2(td_csr)

    pass
```

TextFormatter

```
import numpy as np
from scipy.sparse import csr_array
from scipy.sparse import lil_matrix
from collections import defaultdict

"""
@author rliu 2022/05
"""


class TFIDFScorer:
    """
    Accepts a corpus and vocabulary, and creates the td-idf scores for the vocabulary within
    that corpus. The vocabulary is separate from the corpus to allow for different vocabs.
    """

    def __init__(self, corpus, vocab):
        """
        :param corpus: a pandas Series of the documents in the corpus.
        :param vocab: the assigned mapping of the words in 'vocab' to integer indices.
        """
        self.corpus = corpus
        self.vocab = vocab

    def build(self):
        """
        Builds the tf-idf scores (support counts) of all words of the vocab, within the corpus.
        """

        support = defaultdict(lambda: 0)
        for doc in self.corpus:
            seen = {}
            for wd in doc.split():
                if wd not in seen:
                    support[wd] += 1
                    seen[wd] = 1

        N = len(self.corpus)
        idf = {}
        for wd, supp in support.items():
            idf[wd] = np.log10(N / supp)

        self.df = support
        self.idf = idf

    def weight(self, termdoc_full):
        """
        Accepts a raw word-document matrix and returns the TF-IDF weighted matrix.
        :param termdoc_full:  the term-document representation as a full (non-sparse) np array/matrix.
        """
        inv_vocab = {v: k for k, v in self.vocab.items()}

        tfidf = []
        for wd_idx, docvec in enumerate(termdoc_full):
            wd = inv_vocab[wd_idx]
            idf = self.idf[wd]
            docvec2 = docvec * idf    # apply idf-weighting
            tfidf.append(docvec2)

        return np.array(tfidf)

    def weight2(self, termdoc_csr):
        """
        Accepts a raw word-document matrix and returns the TF-IDF weighted matrix.
        :param termdoc_csr:  the term-document representation as sparse np array/matrix.
        """
        inv_vocab = {v: k for k, v in self.vocab.items()}

#        termdoc_csr.sort_indices()

#        termdoc_lil = lil_matrix(termdoc_csr.shape)
#        for wd_idx in range(0, termdoc_csr.shape[0]):
#            docvec = termdoc_csr.getrow(wd_idx)
#            wd = inv_vocab[wd_idx]
#            idf = self.idf[wd]
#            termdoc_lil[wd_idx] = docvec * idf   # apply idf-weighting
#
#        return termdoc_lil.tocsr()

        data, indices, indptr = np.array([]), np.array([]), np.array([0])
        for wd_idx in range(0, termdoc_csr.shape[0]):
            docvec = termdoc_csr.getrow(wd_idx)
            wd = inv_vocab[wd_idx]
            idf = self.idf[wd]
            docvec2 = docvec * idf    # apply idf-weighting
            data = np.concatenate((data, docvec2.data))
            indices = np.concatenate((indices, docvec2.indices))
            indptr = np.concatenate((indptr, [len(indices)]))

        return csr_array((data, indices, indptr))


if __name__ == "__main__":

    import pandas as pd
    import numpy as np
    from DocTermMatrix import DocTermMatrix
    from TextFormatter import TextFormatter
    from sklearn.metrics.pairwise import cosine_similarity
    from timeit import timeit
    from itertools import islice

    df = pd.read_csv('../data/disaster-tweets/train.csv')
    tf = TextFormatter()
    df['text'] = [tf.format_text(x) for x in df['text']]
    dt = DocTermMatrix(df['text'])

    dt.build_vocab()
    dt.build_full()
    data, indices, indptr = dt.build_csr()
    for i in enumerate(dt.csr):
        pass
    tfidf_scorer = TFIDFScorer(df['text'], dt.vocab)
    tfidf_scorer.build()
    print(dict(islice(tfidf_scorer.idf.items(), 0, 5)))

    # Compute the tfidf-weighted td's using the full td representation.
    td_full = dt.csr.T.toarray()
    td_tfidf_full = tfidf_scorer.weight(td_full)

    # Compute the tfidf-weighted td's using the sparse td representation.
    td_csr = dt.csr.T
    td_tfidf_csr = tfidf_scorer.weight2(td_csr)

    pass
```

VocabBuilder

```
import pandas as pd

"""
```

```
@author rliu 2022/05
"""


class VocabBuilder:
    """
    Builds a vocabulary, which is a mapping from words to indices.
    """

    def __init__(self):
        pass

    def build(self, docs):
        """
        Builds the vocabulary out of the given documents. Returns both the vocab
        and the inverse_mapping.
        """
        all_text = docs.tolist() if isinstance(docs, pd.Series) else docs
        vocab = {}   # >= 3.7, dicts maintain *insertion order*. using this as an ordered set.
        for doc in all_text:
            for s in doc.split():
                if s not in vocab:
                    vocab[s] = len(vocab.keys())

        inv_vocab = {v: k for k, v in vocab.items()}

        return vocab, inv_vocab
```

SGNSEmbedder

```
import numpy as np
import random
import scipy.special

"""
A raw Word2Vec implementation.
@author rliu 2022/05
"""


class SGNSEmbedder:
    """
    Trains a Skip-Gram with Negative Sampling embedding.
    """

    def __init__(self, vocab, d, k, hws, eta=0.1, std=1e-4):
        """
        :param v: the size of the vocabulary.
        :param d: the size of the dense embedding.
        :param k: hyper-parameter representing the ratio of positive to negatively sampled words.
        :param hws: half-window size: 2*hws+1 is the number of words in each window.
        """
        self.k = k
        self.vocab = vocab
        self.v_size = len(vocab.keys())
#        self.W = np.zeros((self.v_size, d))   # weight matrix for input-words (centered word)
#        self.C = np.zeros((self.v_size, d))   # weight matrix for neighbor-words (context word)
#        self.W = std * np.random.randn(self.v_size, d)   # weight matrix for input-words (centered word)
#        self.C = std * np.random.randn(self.v_size, d)   # weight matrix for neighbor-words (context word)
#        self.W = np.random.randn(self.v_size, d)   # weight matrix for input-words (centered word)
#        self.C = np.random.randn(self.v_size, d)   # weight matrix for neighbor-words (context word)
        self.W = np.random.uniform(-0.8, 0.8, (self.v_size, d))
        self.C = np.random.uniform(-0.8, 0.8, (self.v_size, d))
        self.hws = hws
        self.eta = eta

#    def xavier(self):
#        """
#        Xavier initialization on the weights.
#        """
#        lb = -1/np.sqrt(self.v_size)
#        up = 1/np.sqrt(self.v_size)

    def preformat(self, docs):
        """
        Given a corpus, generates the positive and negative training examples. A window of size
        2*self.hws+1 is slid over every document, to generate 2*self.hws positive examples for
        each centered word, and k*2*self.hws negative examples. Each positive example and its
        k negatively-sampled examples are grouped into one training example on which the loss
        is calculated on.
        """

        samples = []

        vocab_words = list(self.vocab.keys())

        for i, doc in enumerate(docs):
            if i % 1000 == 0:
                print('Document %d' % i)
            window = [None] * self.hws        # pad the window for the initial words
            # todo: consider adding tokenization customizability here
            doc = doc.split() + [None] * self.hws     # pad the context for the final words
            cidx = int((2*self.hws+1)/2)
            for s in doc:
                # pad the window until it is the proper size.
                if len(window) < (2*self.hws)+1:
                    window = window + [s]
                    continue
                cwd = window[cidx]
                if cwd is not None:
                    cwds = [cwd] * (self.k + 1)
                    for j, wd in enumerate(window):
                        if j != cidx and wd is not None:
                            pos = wd
                            neg = random.sample(vocab_words, self.k)
                            samp = list(zip(cwds, [pos] + neg))
                            samples.append(samp)
                window = window[1:] + [s]

        return samples

    def sigmoid(self, x):
        return 1/(1+np.exp(x))

    def softmax(self, x):
        """Compute softmax values for each sets of scores in x."""
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum()

    def train_one(self, x_train):
        """
        Trains for one-epoch.
        :param x_train: list of input-pairs: (input-word, neighbor-word).
        :param y_train: labels (1 or 0) for each input-pair: whether it is a pos (1) or neg (0) sample.
        """
        import math

        dps = []
        for center_wd, ctx_wd in x_train:
            dps.append(self.W[center_wd] @ self.C[ctx_wd])
        #dps = [self.W[center_wd] @ self.C[ctx_wd] for center_wd, ctx_wd in x_train]

        pos_dp = np.array([dps[0]])          #  (c dot w)
        neg_dp = np.array(dps[1:]) * -1    # -(c dot w)
#        sigmoids = scipy.special.expit(np.concatenate((pos_dp, neg_dp)))
        sigmoids = self.sigmoid(np.concatenate((pos_dp, neg_dp)))
        #loss = -sum(sigmoids)
        loss = -sum(np.log(sigmoids[np.nonzero(sigmoids)]))  # skip zero's to avoid inf
        return loss, sigmoids[0], sigmoids[1:]

    def train(self, x_train, num_epochs):
        """
        Returns the trained embedding, given a set of training examples.
        """

        loss_arr = []

        import math

        for i in range(num_epochs):
            loss_ep = 0
            for x_tr in x_train:
                loss, pos_sigmoid, neg_sigmoids = self.train_one(x_tr)

                cwd, nwd = x_tr[0]
                w = self.W[cwd]
                d_cpos = (pos_sigmoid - 1) * w
                self.C[nwd] = self.C[nwd] - self.eta * d_cpos

                d_cnegs = []
                for j, (cwd, nwd) in enumerate(x_tr[1:]):
                    w = self.W[cwd]
                    d_cneg = neg_sigmoids[j] * w
                    self.C[nwd] = self.C[nwd] - self.eta * d_cneg
                    d_cnegs.append(d_cneg)

                self.W[cwd] = self.W[cwd] - self.eta * d_cpos + sum(d_cnegs)

                loss_ep += loss
            loss_arr.append(loss_ep/len(x_train))

        return loss_arr
```

Jupyter Notebook: termdoc.ipynb

```
### Term-Document Matrix explorations
@author rliu 2022-05
#%%
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
%autoreload 2
```

```
#%%
import pandas as pd
import numpy as np
from DocTermMatrix import DocTermMatrix
from TextFormatter import TextFormatter

from sklearn.metrics.pairwise import cosine_similarity

from timeit import timeit
from itertools import islice

import matplotlib.pyplot as plt

#%%
df = pd.read_csv('../data/disaster-tweets/train.csv')
tf = TextFormatter()
df['text'] = [tf.format_text(x) for x in df['text']]
dt = DocTermMatrix(df['text'])

dt.build_vocab()
dt.build_full()
dt.build_csr()
#%%
# test that getting row0 is correct:
a = np.array(dt.full[0]) # shape: (24k,)
b = dt.csr.getrow(0).toarray().reshape(-1)  # shape:1x24k to shape:(24k,)
print(np.array_equal(a, b))
#%%

#docterm = td.termdoc_csr.transpose()
#a = docterm.getrow(0)

#%%
a = dt.csr.getrow(0)
b = dt.csr.getrow(1)

# of course, dot product between any two random docs will be 0. most docs will not overlap in words.
a.dot(b.transpose()).toarray()

#%%
# how about dot-products between all docs *within* a class?

#doc_similarity = dt.full
#%time type(dt.full)
#%time dt.full.T @ dt.full

# word similarity matrix
wsm_csr = dt.csr.T @ dt.csr
wsm_full = wsm_csr.toarray()
#%%

plt.imshow(wsm_full, cmap='summer', interpolation='nearest')
plt.title("Word Similarity Matrix Heat Map")
plt.show()

#%%

%time word_sim_matrix = cosine_similarity(wsm_csr)

#%%

plt.imshow(word_sim_matrix, cmap='summer', interpolation='nearest')
plt.title("Word Similarity Matrix Heat Map")
plt.show()

#%%

dsm = dt.csr @dt.csr.T
dsm2 = dsm.toarray()
#%%

plt.imshow(dsm2, cmap='summer', interpolation='nearest')
plt.title("Document Similarity Matrix Heat Map")
plt.show()

#%%

%time doc_sim_matrix = cosine_similarity(dsm2)
#%%
plt.imshow(doc_sim_matrix, cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for Document Similarity Matrix")
plt.show()
#%%
from sklearn.metrics.pairwise import cosine_similarity

#%time word_sim_matrix = cosine_similarity(wsm2)

#%%
#plt.imshow(word_sim_matrix, cmap='summer', interpolation='nearest')
#plt.title("Cosine similarity map for Word Similarity Matrix")
#plt.show()
```

```
#%%
from TFIDFScorer import TFIDFScorer
from itertools import islice

tfidf_scorer = TFIDFScorer(df['text'], dt.vocab)

tfidf_scorer.build()
print(dict(islice(tfidf_scorer.idf.items(), 0, 5)))
#%%
# Compute the tfidf-weighted td's using the full td representation.
td_full = dt.csr.T.toarray()
td_tfidf_full = tfidf_scorer.weight(td_full)

#%%
# compute the document similarity matrix using the full representation
dsm_tfidf = td_tfidf_full.T @ td_tfidf_full
%time dsm_tfidf_simm = cosine_similarity(dsm_tfidf)

plt.imshow(dsm_tfidf_simm, cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for TFIDF-weighted Document Similarity Matrix")
plt.show()

#%%
# Compute the tfidf-weighted td's using the sparse td representation.
td_csr = dt.csr.T
td_tfidf_csr = tfidf_scorer.weight2(td_csr)

#%%
# compute the document similarity matrix using the csr representation
dsm_tfidf2 = td_tfidf_csr.T @ td_tfidf_csr
%time dsm_tfidf2_simm = cosine_similarity(dsm_tfidf2)

plt.imshow(dsm_tfidf2_simm, cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for TFIDF-weighted Document Similarity Matrix")
plt.show()

#%%
# compute document similarities using document-vectors (average of all word vectors)




#%%
# word similarity matrix
wsm_tfidf = td_tfidf_full @ td_tfidf_full.T
%time wsm_tfidf_simm = cosine_similarity(wsm_tfidf)
#%%

plt.imshow(wsm_tfidf_simm, cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for TFIDF-weighted Word Similarity Matrix")
plt.show()

#%%
# word similarity matrix
wsm_tfidf2 = td_tfidf_csr @ td_tfidf_csr.T
%time wsm_tfidf2_simm = cosine_similarity(wsm_tfidf2)
#%%

plt.imshow(wsm_tfidf2_simm, cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for TFIDF-weighted Word Similarity Matrix")
plt.show()

#%%

plt.imshow(wsm_tfidf2_simm[0:5000,0:5000], cmap='summer', interpolation='nearest')
plt.title("Cosine similarity map for TFIDF-weighted Word Similarity Matrix")
plt.show()

#%%
from CoccurMatrix import CoccurMatrix
from VocabBuilder import VocabBuilder
from DictUtils import DictUtils

vb = VocabBuilder()
vocab, inv_vocab = vb.build(df['text'])
co = CoccurMatrix(vocab, 2)

coccur = co.build(df['text'])
coccur2 = DictUtils.remap(coccur, inv_vocab, 2)

#%%
from PMIScorer import PMIScorer

pmiscorer = PMIScorer()

pmi, word_cnt, context_cnt, tot = pmiscorer.build(coccur2)
print('PMI score of "our, to": %f' % pmi['our']['to'])
print('PMI score of "are, the": %f' % pmi['are']['the'])
print('PMI score of "deeds, our": %f' % pmi['deeds']['our'])
print('PMI score of "our, deeds": %f' % pmi['our']['deeds'])

pmi_mat = DictUtils.tomatrix(pmi)
```

```
plt.imshow(pmi_mat, cmap='summer', interpolation='nearest')
plt.title("PMI scores for a 5-word context")
plt.show()

#%%
pmi_mat[0:3,0:3]

plt.imshow(pmi_mat[0:100,0:100], cmap='summer', interpolation='nearest')
plt.title("PMI scores for a 5-word context")
plt.show()

#%%
# Try a X-word context window:
X = 30
vb = VocabBuilder()
vocab, inv_vocab = vb.build(df['text'])
co = CoccurMatrix(vocab, X)
coccurX = co.build(df['text'])
coccurX_2 = DictUtils.remap(coccurX, inv_vocab, 2)


#%%
from PMIScorer import PMIScorer
pmiscorer = PMIScorer()
pmiX, _, _, _ = pmiscorer.build(coccurX_2)
pmiX_mat = DictUtils.tomatrix(pmiX)
#%%

plt.imshow(pmiX_mat, cmap='summer', interpolation='nearest')
plt.title("PMI scores for a %d-word context" % (2*X+1))
plt.show()

#%%

plt.imshow(pmiX_mat[0:10,0:10], cmap='summer', interpolation='nearest')
plt.title("PMI scores for a %d-word context" % (2*X+1))
plt.show()

#%%
print(list(coccurX_2.keys())[0:19])

for wd1 in list(coccurX_2.keys())[0:19]:
    d2 = coccurX_2[wd1]
    print(list(d2.keys())[0:19])
```

Jupyter Notebook: embeddings.ipynb

```
#%% md
### Embeddings exploration

@author rliu 2022/05
#%%
#%load_ext autoreload
#%autoreload 2

#%%
import numpy as np
import pandas as pd

from TextFormatter import TextFormatter
from VocabBuilder import VocabBuilder
from ListUtils import ListUtils

from SGNSEmbedder import SGNSEmbedder

import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import cosine_similarity
from timeit import timeit

#%%

df = pd.read_csv('../data/disaster-tweets/train.csv')
tf = TextFormatter()
df['text'] = [tf.format_text(x) for x in df['text']]

vb = VocabBuilder()
vocab, inv_vocab = vb.build(df['text'])

#%%
vocab_words = list(vocab.keys())
words = np.random.choice(vocab_words, 5)
words

#%%

w2v_dim = 50
sgns = SGNSEmbedder(vocab, w2v_dim, 3, 2)
samples = sgns.preformat(df['text'])
# encode samples using vocab map
samples2 = ListUtils.remap(samples, vocab)
```

```
#%%
print(samples2[0:3])
print(len(samples2))
#%%

loss = sgns.train(samples2, 10)
print(loss)

#%%

plt.plot(loss)
plt.title('Training Loss for word2vec %d-dim' % w2v_dim)
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()

#%%

print(type(sgns.W))
print(sgns.W.shape)

#%%

#plt.imshow(sgns.W[:100,0:10], cmap='summer', interpolation='nearest')
plt.imshow(sgns.W, cmap='summer', interpolation='nearest', aspect='auto')
plt.title('Word2vec %d-dim Trained Weights' % w2v_dim)
plt.show()

#%%
# word similarity matrix
wsm = sgns.W @ sgns.W.T
%time wsm_simm = cosine_similarity(wsm)

#%%

%time wsm_simm2 = cosine_similarity(sgns.W)

#%%

plt.imshow(wsm_simm, cmap='summer', interpolation='nearest')
plt.title('Cosine Word Similarity Matrix for %d-dim Word2Vec-Embedding' % w2v_dim)
plt.show()

#%% md
### Gensim Word2Vec
#%%
import gensim
from gensim.models import Word2Vec

#%%
# Prepare data for gensim
data = vocab_words

#%%
# Train gensim model(s)
gs_sgns = gensim.models.Word2Vec(data, min_count=1, vector_size=50, window=5, sg=3)

#%%
# W matrix
gs_sgns.wv.vectors

#%%

plt.imshow(gs_sgns.wv.vectors, cmap='summer', interpolation='nearest', aspect='auto')
plt.title('Gensim SGNS %d-dim W matrix' % w2v_dim)
plt.show()

#%%

plt.imshow(gs_sgns.syn1neg, cmap='summer', interpolation='nearest', aspect='auto')
plt.title('Gensim SGNS %d-dim C matrix' % w2v_dim)
plt.show()

#%%
gs_wsm = gs_sgns.wv.vectors @ gs_sgns.wv.vectors.T

plt.imshow(gs_wsm, cmap='summer', interpolation='nearest')
plt.title('Cosine Word Similarity Matrix for\n50-dim Gensim Word2Vec Weights-Embedding')
plt.show()

#%%
# C matrix
gs_sgns.syn1neg

#%%


data = ' '.join(df['text'])
```

POS-Tagging

Factor Operations (Multiply, Marginalize, Print Table)

```
from functools import reduce
from itertools import product
from collections import defaultdict

from list_utils import flatten

"""
@author rliu 2022/06
"""


def build_val2idx_map(x, join_key_indices):
    """
    Builds a map of values in factor x to their row indices. The join_keys are expected to be indexed with respect to the given table.
    """
    x_map = defaultdict(lambda: [])
    x_tbl = factor_to_table(x)
    x_subset = [x_tbl[i] for i in join_key_indices]  # subset of a of only the join_key columns
    x_subset_as_rows = list(map(list, zip(*x_subset)))  # convert to by-row so we can do row-value matching
    for i, v in enumerate(x_subset_as_rows):
        x_map[tuple(v)].append(i)
    return x_map


def get_vars_list(f):
    """
    Returns a flattened list of variables, starting first with its declared parents' variables (in the order of declaration for parents),
    and then with the declared-order of variables for the factor.
    """
    return list(flatten([p['vars'] for p in f['parents']])) + f['vars']


def multiply_factors(a, b):
    """
    Multiplies the two factors together.
    A factor multiplication is analogous to a relational join on the common attributes (variables) between the two factors. Just like a relational join,
    if there are no attributes(variables) in common, then it becomes a cartesian-product.
    We can do a simple relational join algorithm: block-nested join where the outer relation is larger and inner relation is smaller.
       - the outer larger relation miimizes disk io (seeks) if it is actually blocked.
       - none of this is applicable/relevant here, of course, as we assume both a, b, are in memory.
       - But this can be an optimization/consideration when factors become non-trivially large.
    """
    # todo: note that this join_key uses object comparison; may need to instead compare by object name
#    join_keys = [(a.index(var), b.index(var)) for var in a['parents'] if var in b['parents']]

    # gather all variables in a factor: both in its vars and in its parents (consider: both show up in the CPT).
    vars_a = list(flatten([p['vars'] for p in a['parents']])) + a['vars']
    vars_b = list(flatten([p['vars'] for p in b['parents']])) + b['vars']

    join_keys = [var for var in vars_a if var in vars_b]
    join_key_indices_a = [vars_a.index(var) for var in join_keys]
    join_key_indices_b = [vars_a.index(var) for var in join_keys]

    if not join_keys:
        a_row_indices = list(range(len(a['val'])))
        b_row_indices = list(range(len(b['val'])))
        pairs = product(a_row_indices, b_row_indices)
    else:
        a_val2idx_map = build_val2idx_map(a, join_key_indices_a)
        b_val2idx_map = build_val2idx_map(b, join_key_indices_b)
        pairs = []
        for k, a_v in a_val2idx_map.items():
            if k not in b_val2idx_map:
                # todo: what/how do we handle BNs with different levels for nodes? Here, this is a join on 2 same nodes but with different levels.
                raise Exception('Join key value in factor 1 but not in factor 2')
            b_v = b_val2idx_map[k]
            pairs = pairs + list(product(a_v, b_v))

    # very importantly, parents are listed first (leftwards), before the current factor's variables.
    pairs = sorted(pairs)

    prod_vals = [a['val'][a_idx] * b['val'][b_idx] for a_idx, b_idx in pairs]

    merged_parents = []
    _ = [merged_parents.append(e) for e in (a['parents']+b['parents']) if e not in merged_parents]  # merge both but exclude parents to either.

    merged_vars = a['vars'] + b['vars']
    for x in join_keys:
        if x in merged_vars:
            merged_vars.remove(x)

    f = {'vars': merged_vars, 'parents': merged_parents, 'val': prod_vals}
    return f


def sum_out(f, sum_vars):
    """
    Sums out the list of variables ’sum_vars’ from the factor ’f’.
    This is implemented by partitioning by all unique combos of values in sum_vars; then, for the rows left-over in each partition,
    they are combined cross-partition based on if they are equal or not.
    """
    vars = get_vars_list(f)
    # it may be simpler (for now) to just build the full table and match using the built row values.
    arr = factor_to_table(f)

    sum_vars_indices = [vars.index(sv) for sv in sum_vars]

#    sub_arr = [arr[i] for i in sum_vars_indices]
#    sub_arr_as_rows = list(map(list, zip(*sub_arr)))  # convert the by-column format to by-row

    # submatrix with the sum_vars projected-out
    sub_arr = [arr[i] for i in range(0, len(arr)) if i not in sum_vars_indices]
    sub_arr = sub_arr[:-1]
    sub_arr_as_rows = list(map(list, zip(*sub_arr)))  # convert the by-column format to by-row

    # collect indices of all same row-values in sub_arr (sub_arr is the columns of the vars to be summed-out)
    idx_map = defaultdict(lambda: [])
    [idx_map[tuple(v)].append(i) for i, v in enumerate(sub_arr_as_rows)]

    new_vals = []
    vals = f[’val’]
    for k, indices in idx_map.items():
        sum_val = 0
        for i in indices:
            sum_val += vals[i]
        new_vals.append(sum_val)

    new_vars = [v for v in f[’vars’] if v not in sum_vars]
    new_pars = [v for v in f[’parents’] if v not in sum_vars]

    new_f = {’vars’: new_vars, ’parents’: new_pars, ’val’: new_vals}

    return new_f


def factor_to_table(a):
    """
    Returns a 2d col,row array for the factor, using the order of parents listed in the factor.
    The factor itself is the final column.
    """
    vars = list(flatten([p[’vars’] for p in a[’parents’]])) + a[’vars’]
    num_rows = reduce(lambda x, y: x*y, [v[’levels’] for v in vars])
    # nc x nr
    arr = [[’0’] * num_rows] * (len(vars)+1)
    arr[len(vars)] = list(map(lambda s: f’{s:.4f}’, a[’val’]))   # fill the last column using its actual values.

    # proceed to fill each column’s rows all at once. prev_levels tracks the number of items the
    #   current column will need to repeat in sequence to handle each different case of all the columns
    #   previously filled, ie, to its right (since we fill the columns in reverse order-- right to left).
    #   so, the first column will be the factor itself (since vars = a[’parents’+[a]), and so on.
    prev_levels = 1
    for j, v in enumerate(reversed(vars)):
        v2 = [x for x in v[’level_names’] for _ in range(prev_levels)]    # [0,0,,1,1,2,2...]
        v2 = v2 * int(num_rows/len(v2))
        arr[len(vars)-1-j] = v2
        prev_levels *= v[’levels’]
    return arr


def factor_to_string(a):
    """
    Returns a string format of the given factor
    """
    vars = list(flatten([p[’vars’] for p in a[’parents’]])) + a[’vars’]
    allnames = list(flatten(f[’name’] for f in vars))  # flatten all names
    allnames_len = list(map(len, allnames))
    alllvlnames_len = [max(map(len, f[’level_names’])) for f in vars]  # no flatten needed as the max reduces the inner f[’level_names]’ from a list of level_n
    col_lens = [max(l1, l2) for l1, l2 in zip(allnames_len, alllvlnames_len)]
#    col_lens = [max(max(map(len, p[’level_names’])), len(p[’name’])) for p in factors]

    arr = factor_to_table(a)
    num_rows = len(arr[0])

    if not arr[-1]:
        arr[-1] = [’0’] * num_rows
    col_lens.append(max(map(len, arr[-1])))  # add the max-width of the actual CPT’s values (factor a’s values)

    out = []
    out.append(’ ’.join(name.ljust(col_lens[j]) for j, name in enumerate(allnames)))
    for i in range(num_rows):
        row = []
        for j, _ in enumerate(arr):
            row.append(arr[j][i].ljust(col_lens[j]))
        out.append(’ ’.join(row))

    return ’\n’.join(out)


if __name__ == "__main__":
    level_names_01 = [str(i) for i in list(range(2))]
    level_names_012 = [str(i) for i in list(range(3))]
    # todo if val length does not match actual factor size, then factor_to_string() has an error
    va = {'name': 'a', 'levels': 2, 'level_names': level_names_01}
    vb = {'name': 'b', 'levels': 2, 'level_names': level_names_01}
    vc = {'name': 'c', 'levels': 2, 'level_names': level_names_01}
    vd = {'name': 'd', 'levels': 2, 'level_names': level_names_01}
    fa = {'vars': [va], 'val': [0.8, 0.2], 'parents': []}
    fb = {'vars': [vb], 'val': [1.0, 0.0, 0.2, 0.8], 'parents': [fa]}
    fc = {'vars': [vc], 'val': [1.0, 0.0, 0.2, 0.8], 'parents': [fb]}
    fd = {'vars': [vd], 'val': [1.0, 0.0, 0.2, 0.8], 'parents': [fb]}

    print(factor_to_string(fa))
    print(factor_to_string(fb))
    fab = multiply_factors(fa, fb)
    print(factor_to_table(fab))
    print(factor_to_string(fab))

    print(factor_to_string(fc))
    print(factor_to_string(fd))
    fcd = multiply_factors(fc, fd)
    print(factor_to_table(fcd))
    print(factor_to_string(fcd))

    f_cd = sum_out(fcd, [vc, vd])
    print(factor_to_string(f_cd))
```

MPE (unfinished)

```
import functools
from factor_operations import multiply_factors

"""
@author rliu 2022/06
"""

"""
Graph:    a -> b -> c
               \-> d
"""
bn0 = {
    'nodes': {'a', 'b', 'c', 'd'},
    'parents': {'a': [], 'b': ['a'], 'c': ['b'], 'd': ['b']},
}

factors = {
    'a': [0.8, 0.2],
    'b': [1.0, 0.0, 0.5, 0.5],
    'c': [0.5, 0.5, 0.0, 1.0],
    'd': [0.7, 0.3, 0.25, 0.75],
}


def reverse_dict(dic):
    """
    Reverses the given dictionary.
    """
    # return {v: k for k, v in dic.items()}
    new_dic = {}
    for k, v in dic.items():
        for x in v:
            new_dic.setdefault(x, []).append(k)
    return new_dic


class VE_MPE:
    """
    Calculates the Most Probable Explanation for a given Bayesian Network. Uses extended-faactors to
    track the MPE instantiations.
    """

    def __init__(self):
        pass

    def prune_edges(self, N):
        return N

    def apply_evidence(self, f, e):
        """
        :param f: the factors over which to apply evidence e.
        :param e: the evidence to apply.
        :return: the factors reduced by the evidence e.
        """
        return f


    def run(self, N, f, e, ve):
        """
        :param N: Bayesian network over which to compute the MPE.
        :param f: factors of the nodes in N.
        :param e: any evidence variables.
        :param ve: variable elimination order.
        :return: the final extended factor corresponding to the MPE.
        """
        N2 = self.pruneEdges(N)
        S = self.applyEvidence(f, e)

        children = reverse_dict(N2.parents)

        for var in ve:
            fset = [S[fname] for fname in children[var]]   # all factors that mention 'var'
            fres = functools.reduce(multiply_factors, fset)
            fres2 = self.max_out(fres, var)   # max-out 'var' from resultant factor.
            # replace all factors in S by resultant factor.
            for fname in fset:
                S[fname] = fres2
        pass


if __name__ == "__main__":
    pass
```

Jupyter Notebook: pos-tagging.ipynb

```
## pos-tagging
@author rliu 2022-05
#%% md
### Setup
#%%
#%load_ext autoreload
#%autoreload 2
#%%
import pandas as pd
from functools import partial
from itertools import chain
flatten = chain.from_iterable

import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import brown
from nltk.corpus import treebank
from nltk.tag import hmm

#%% md
### Load datasets
#%%
nltk.download('brown')
nltk.download('treebank')
#%%
# Train data - pretagged
treebank_tr_data = treebank.tagged_sents()[:3000]
#%% md
### POS-tagging using NLTK
#%%
# Download some models to do tagging.
#     Resource averaged_perceptron_tagger not found.
#       Please use the NLTK Downloader to obtain the resource:
#       >>> import nltk
#       >>> nltk.download('averaged_perceptron_tagger')
#       For more information see: https://www.nltk.org/data.html
#       Attempted to load taggers/averaged_perceptron_tagger/averaged_perceptron_tagger.pickle
nltk.download('averaged_perceptron_tagger')
#%% md
### Setup test sentences
#%%
TE_SENT1 = "And now for something completely different"
TE_SENT2 = "This is supposed to be a super duper long and hard sentence that will create different results in both taggers."

#%% md
### Default tagger (averaged perceptron)
#%%
text = word_tokenize(TE_SENT1)
nltk.pos_tag(text)
#%% md
### HMM tagger
#%%
# Setup a trainer with default(None) values
# And train with the data
trainer = hmm.HiddenMarkovModelTrainer()
hmm_tagger = trainer.train_supervised(treebank_tr_data)
# Prints the basic data about the tagger
print(hmm_tagger)
#%%
hmm_tagger.tag(TE_SENT1.split())
#%% md
#### Test sentence 1
#%%
print('avg-perceptron-tagger:\n\t%s' % nltk.pos_tag(word_tokenize(TE_SENT1)))
print('hmm-tagger:\n\t%s' % hmm_tagger.tag(TE_SENT1.split()))

#%% md
#### Test sentence 2
```

```
#%%
print('avg-perceptron-tagger:\n\t%s' % nltk.pos_tag(word_tokenize(TE_SENT2)))
print('hmm-tagger:\n\t%s' % hmm_tagger.tag(TE_SENT2.split()))
```

Classification

NaiveBayes

```
import pandas as pd
import math
import re
import operator
from collections import defaultdict

"""
@author rliu 2022/05
"""


class Document:
    """
    Represents a document in a corpus.
    """

    def __init__(self, text, cls):
        self.text = text
        self.cls = cls


class NaiveBayes:

    def __init__(self):
        self.docs = {}

    def read_documents(self, docs):
        pass

    @staticmethod
    def format_text(text):
        # using string.translate() is fastest way to remove puncutation
        # see:  https://datagy.io/python-remove-punctuation-from-string/
        # however, we want to use more nuanced ways, that preserve hashtags and exclamation marks after a word.

        text = text.lower()
        #    all_text = [s.translate(str.maketrans('', '', string.punctuation)) for s in all_text]
        #    all_text = [re.sub(r'([.]+)(\w+)', r'\1 \2', s) for s in all_text] # fix all periods immediately followed by text, so they will be correctly token
        text = re.sub(r'[.]{2,}', '.',
                      text)  # change 2 or more consecutive periods to a single period.
        text = re.sub(r'[.]+\s+', ' ', text)  # remove all periods
        text = re.sub(r'[\-]+', ' ', text)  # replace hypthens with spaces
        text = re.sub(r'[?,.)(\[\]{}:]', '', text)  # remove punctuation
        return text

    def train(self, df):
        """
        """

        # iterating over df is not recommended; seems to do with creating Series for every row (ie, to handle arbitrary # of cols).
        # itertuple() is faster than iterrows() if have < 255 columns.
        # see:  https://stackoverflow.com/questions/16476924/how-to-iterate-over-rows-in-a-dataframe-in-pandas

        # finally, to create an object per row, it is faster to simply convert the whole df to a numpy array,
        # then iterate:
        # see:  https://stackoverflow.com/questions/25274270/how-can-i-convert-each-pandas-data-frame-row-into-an-object-including-the-column

        # docs = df.to_records()

        df['text'] = [NaiveBayes.format_text(x) for x in df['text']]

        docs0 = df[df['target'] == 0]
        docs1 = df[df['target'] == 1]

        Ndoc = len(df)
        Nc = [len(docs0), len(docs1)]

        logprior = [math.log(Nc[0]/Ndoc, 2), math.log(Nc[1]/Ndoc, 2)]

        all_text = df['text'].tolist()
        vocab = set()
        [vocab.update(s.split()) for s in all_text]

        #    # for debugging
        #    out_file = open("all_text.txt", "w")
        #    n = out_file.write(" ".join(all_text))
        #    out_file.close()

        docs0_text = docs0['text'].tolist()
        docs1_text = docs1['text'].tolist()

        docs0_cnt = defaultdict(lambda: 0)
        docs1_cnt = defaultdict(lambda: 0)

        for doc in docs0_text:
            for wd in doc.split():
                docs0_cnt[wd] += 1
        for doc in docs1_text:
            for wd in doc.split():
                docs1_cnt[wd] += 1

        count = {0: defaultdict(lambda: 0), 1: defaultdict(lambda: 0)}
        for wd in vocab:
            count[0][wd] += docs0_cnt[wd]
            count[1][wd] += docs1_cnt[wd]

        count0_sorted = sorted(count[0].items(), key=operator.itemgetter(1), reverse=True)
        count1_sorted = sorted(count[1].items(), key=operator.itemgetter(1), reverse=True)

        docs0_sum = sum(docs0_cnt.values())
        docs1_sum = sum(docs1_cnt.values())

        llik = {0: {}, 1: {}}
        for wd in vocab:
            llik[0][wd] = math.log((count[0][wd]+1)/(docs0_sum + len(vocab)), 2)
            llik[1][wd] = math.log((count[1][wd]+1)/(docs1_sum + len(vocab)), 2)

        self.logprior = logprior
        self.llik = llik
        self.vocab = vocab


    def test_one(self, doc):
        """
        """
        sum = [None] * len(self.logprior)
        for c in [0, 1]:
            sum[c] = self.logprior[c]
            for wd in doc:
                if wd in self.vocab:
                    sum[c] += self.llik[c][wd]
        return sum

    def test(self, df):
        preds = [None] * len(df['text'])
        for i, doc in enumerate(df['text']):
            sum = nb.test_one(doc)
            preds[i] = sum.index(max(sum))
        return preds


if __name__ == "__main__":
    nb = NaiveBayes()
    df = pd.read_csv('../data/disaster-tweets/train.csv')

    train = df.sample(frac=0.7, random_state=123)  # random state is a seed value
    test = df.drop(train.index)

    nb.train(train)
    preds = nb.test(test)
    golds = train['target'].tolist()

    corr = 0
    for p, g in zip(preds, golds):
        if p == g:
            corr += 1
    acc = corr / len(preds)

    print(f'Accuracy is: {acc}')
```

ScikitLearn: Logistic Regression

```
### author rliu 2022/05

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

#%%
# Load words
df = pd.read_csv('../data/disaster-tweets/train.csv')
print(df['text'][:5])

text_trainval, y_trainval = df['text'], df['target']

from sklearn.feature_extraction.text import CountVectorizer
text_train, text_val, y_train, y_val = train_test_split(
    text_trainval, y_trainval, stratify=y_trainval, random_state=0)

vect = CountVectorizer()
X_train = vect.fit_transform(text_train)  # fit docs into a word-doc matrix.
X_val = vect.transform(text_val)          # apply fitted word-doc matrix (ie, |V|) to the test docs.
print("Vocabulary size: %d" % len(vect.vocabulary_))
print("Vocabulary: " + str(list(map(str, vect.vocabulary_.items()))[:5]))
X_train.shape
# Vocabulary size: 17715
# Vocabulary: ["('flunkie', 6227)", "('if', 7857)", "('it', 8267)", "('makes', 9787)", "('you', 17443)"]
# (5709, 17715)

from sklearn.linear_model import LogisticRegressionCV
lr = LogisticRegressionCV().fit(X_train, y_train)
#%%
lr.C_

lr.score(X_val, y_val)
# 0.803046218487395

print(lr.coef_, lr.intercept_)
# [[-0.01422598  0.02817988  0.08329446 ...  0.18241662  0.05989239

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# source:  https://amueller.github.io/aml/05-advanced-topics/13-text-data.html
def visualize_coefficients(coefficients, feature_names, n_top_features=25):
    """Visualize coefficients of a linear model.

    Parameters
    ----------
    coefficients : nd-array, shape (n_features,)
        Model coefficients.

    feature_names : list or nd-array of strings, shape (n_features,)
        Feature names for labeling the coefficients.

    n_top_features : int, default=25
        How many features to show. The function will show the largest (most
        positive) and smallest (most negative)  n_top_features coefficients,
        for a total of 2 * n_top_features coefficients.
    """
    coefficients = coefficients.squeeze()
    if coefficients.ndim > 1:
        # this is not a row or column vector
        raise ValueError("coeffients must be 1d array or column vector, got"
                         " shape {}".format(coefficients.shape))
    coefficients = coefficients.ravel()

    if len(coefficients) != len(feature_names):
        raise ValueError("Number of coefficients {} doesn't match number of"
                         "feature names {}.".format(len(coefficients),
                                                    len(feature_names)))
    # get coefficients with large absolute values
    coef = coefficients.ravel()
    positive_coefficients = np.argsort(coef)[-n_top_features:]
    negative_coefficients = np.argsort(coef)[:n_top_features]
    interesting_coefficients = np.hstack([negative_coefficients,
                                          positive_coefficients])
    # plot them
    plt.figure(figsize=(15, 5))
    colors = ['r' if c < 0 else 'b'
              for c in coef[interesting_coefficients]]
    plt.bar(np.arange(2 * n_top_features), coef[interesting_coefficients],
            color=colors)
    feature_names = np.array(feature_names)
    plt.subplots_adjust(bottom=0.3)
    plt.xticks(np.arange(1, 1 + 2 * n_top_features),
               feature_names[interesting_coefficients], rotation=60,
               ha="right")
    plt.ylabel("Coefficient magnitude")
    plt.xlabel("Feature")

plt.figure(figsize=(20, 5), dpi=300)
visualize_coefficients(
    lr.coef_,
    np.array(vect.get_feature_names_out()), n_top_features=40)
```

ScikitLearn: Naive Bayes

```
### author rliu 2022/05

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

#%%
# Load words
df = pd.read_csv('../data/disaster-tweets/train.csv')
print(df['text'][:5])

text_trainval, y_trainval = df['text'], df['target']
```

```
from sklearn.feature_extraction.text import CountVectorizer
text_train, text_val, y_train, y_val = train_test_split(
    text_trainval, y_trainval, stratify=y_trainval, random_state=0)

vect = CountVectorizer()
X_train = vect.fit_transform(text_train)  # fit docs into a word-doc matrix.
X_val = vect.transform(text_val)          # apply fitted word-doc matrix (ie, |V|) to the test docs.


from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
clf.fit(X_train, y_train)
MultinomialNB()
y_pred = clf.predict(X_val)
print(y_pred)

# Compute accuracy
corr = 0
for gold, pred in zip(y_val, y_pred):
    if gold == pred:
        corr += 1
print(f'Acc: {corr/len(y_pred)}')
```