Sparse vectors recap

Previous lectures:

- How to represent a word as a sparse vector with dimensions corresponding to the words in the vocabulary
- The values in the vector were a function of the count of the word co-occurring with each neighbouring word
- Each word is thus represented with a vector that is long (with vocabularies of 20,000 to 50,000)
  - and sparse (with most elements of the vector for each word equal to zero)

Today’s Lecture

- How to represent a word with vectors that are short (with length of 50 – 1,000)
  - and dense (most values are non-zero)
- Why short vectors?
  - Easier to include as features in machine learning systems
  - Because they contain fewer parameters, they generalize better and are less prone to overfitting
  - Sparse vectors are better at capturing synonymy

Before density, another approach to normalisation

- Raw frequency moves common and uncommon words away from each other in vector space
- Cosine (length-weighted dot-product) normalises length
- PPMI is one way of trying to detect important co-occurrences based on divergence between observed and predicted (from unigram MLEs) bigram probabilities
- Another intuition is that a word that’s common in only some contexts carries more information than one that’s common everywhere
Common here, but not elsewhere...

- This is formalised under the name **tf-idf**
  - **tf** Term frequency
  - **idf** Inverse document frequency

- Originally from Information Retrieval, where there a lots of documents, often with lots of words in them

---

**Tf-idf: combine two factors**

- **tf**: term frequency:
  \[
  tf_{t,d} = \begin{cases} 
  \log_{10} \text{count}(t,d) & \text{if } \text{count}(t,d) > 0 \\
  0 & \text{otherwise}
  \end{cases}
  \]
  frequency count of term \(i\) in document \(d\)

- **Idf**: inverse document frequency:
  \[
  idf_i = \log \left( \frac{N}{df_i} \right)
  \]
  \(N\) is total # of docs in collection
  \(df_i\) is # of docs that have term \(i\)

- Terms such as *the* or *good* have very low idf
  - because \(df_i \approx N\)

- **tf-idf value for word \(t\) in document \(d\):**
  \[
  w_{t,d} = tf_{t,d} \times idf_t
  \]

---

**Summary: tf-idf**

- Compare two words using tf-idf cosine to see if they are similar

- Compare two documents
  - Take the centroid of vectors of all the terms in the document
  - Centroid document vector is:
    \[
    d = \frac{t_1 + t_2 + \cdots + t_k}{k}
    \]
  - And yes, that’s vector addition but sort-of arithmetic division

---

**Tf-idf and PPMI are sparse representations**

- tf-idf and PPMI vectors are
  - long (length \(|V| = 20,000\) to 50,000)
  - sparse (most elements are zero)

- Alternative: dense vectors
  - vectors which are
    - short (length 50–1000)
    - dense (most elements are non-zero)
Neural network-inspired dense embeddings

- Methods for generating dense embeddings inspired by neural network models
- Neural network language models are trained to predict contexts from words – this process amounts to learning word embeddings
- The intuition is that words with similar meanings tend to occur near each other in text
- The process for learning these embeddings has a strong relationship with dot-product similarity metrics
- We’ll look at one particular version of this: skip-grams

Word2vec

- Instead of counting how often each word $w$ occurs near “apricot”
- Train a classifier on a binary prediction task:
  - Is $w$ likely to show up near “apricot”?
- We don’t actually care about this task
  - But we’ll take the learned classifier weights as the word embeddings

Use running text as implicitly supervised training data!

- A word $c$ near apricot
  - Acts as gold ‘correct answer’ to the question
  - “Is word $w$ likely to show up near apricot”
- No need for hand-labeled supervision
- The idea comes from neural language modeling

skip-grams (and related approaches such as continuous bag of words (CBOW)) are often referred to as word2vec

- Code available on the web
- Popular embedding methods
- Very fast to train
- Idea: predict rather than count
**Prediction with Skip-grams**

- Skip-gram model predicts each neighbouring word in a context window of \( L \) words, e.g. context window \( L = 2 \) the context is \([w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]\).
- We want to predict each of the context words from word \( w_j \).
- Skip-gram calculates the probability \( p(w_k | w_j) \) by computing dot product between context vector \( c_k \) of word \( k \) and target vector \( v_j \) for word \( w_j \).
- The higher the dot product between two vectors, the more similar they are.

**Skip-gram with negative sampling**

- Faster than using the softmax function.
- In the training phase, for each target word the algorithm chooses surrounding context words as positive examples.
- For each positive example, the algorithm samples \( k \) noise examples, or negative examples, according to their weighted unigram probability, from non-neighbour words.
- The goal is to move the embeddings towards the neighbour words and away from noise words.
- We’re basically training a logistic regression classifier with two groups of features: positive and negative.

**Prediction with Skip-grams**

- Dot product \( c_k \times v_j \) is a number ranging from -inf. to +inf
- We use softmax function to normalize the dot product into probabilities:

\[
p(w_k | w_j) = \frac{\exp(c_k \cdot v_j)}{\sum_{i \in |V|} \exp(c_i \cdot v_j)}
\]
- Computing the denominator requires computing dot product between each word in \( V \) and the target word \( w_i \), which may take a long time.

**Skip-gram with negative sampling**

Training sentence for example word apricot:

lemon, a tablespoon of apricot preserves or jam

\( c_1 \quad c_2 \quad w \quad c_3 \quad c_4 \)
- goal - learn an embedding whose dot product with each context word is high.
- We select 2 noise words for each of the context words:

\( n_1 \quad n_2 \quad n_3 \quad n_4 \quad w \quad n_5 \quad n_6 \quad n_7 \quad n_8 \)
- We want noise words \( n_i \) to have a low dot-product with target embedding \( w \).
Skip-Gram Goal

- Given a pair \((t, c) = \text{target, context}\)
  - (apricot, jam)
  - (apricot, aardvark)
- Return probability that \(c\) is a real context word:
  - \(P(+|t, c)\)
  - \(P(-|t, c) = 1 - P(+|t, c)\)

How to compute \(p(+|t, c)\)?

Intuition:

- Words are likely to appear near similar words
- Model similarity with dot-product!

Similarity \((t, c) \propto t \cdot c\)

Problem:

- Dot product is not a probability!
  - (Neither is cosine)

Turning dot product into a probability

Solution: use a sigmoid

- The sigmoid lies between 0 and 1:
  \[
  \sigma(x) = \frac{1}{1 + e^{-x}}
  \]

Skip-gram with negative sampling

So, given the learning objective is to maximise:

\[
\log P(+|t, c) + \sum_{i=1}^{k} \log P(-|t, n_i)
\]

We use \(\sigma\) and dot product to get:

\[
\log \sigma (c \cdot w) + \sum_{i=1}^{k} \log \sigma (-n_i \cdot w)
\]

- where \(\sigma\) is a sigmoid function of the dot product
  - \(c\) is our ‘good’ context vector, and the \(n_i\) are the vectors for our \(k\) ‘bad’ context words
  - We want to maximise the dot product of \(w\) with \(c\)
  - and minimise over all the dot products of \(w\) with all the \(n_i\)
Reminder: We have two matrices

- We’re learning two separate embeddings for each word $w$: word embedding $v$ and context embedding $c$
- Embeddings encoded in two matrices: word matrix $W$ and context matrix $C$
- Each row $i$ of word matrix $W$ is a $1 \times d$ vector embedding $v_i$ for word $i$ in vocabulary $V$
- Each column $i$ of the context matrix $C$ is a $d \times 1$ vector embedding $c_i$ for word $i$ in vocabulary $V$

Learning skip-gram weights

We can approach this as a logistic regression problem

- Use stochastic gradient descent
  - Starts with randomly initialized $W$ and $C$ matrices
  - Then iterates over the training corpus to maximize the objective function

Skip-gram as neural network

Or we can use a neural network to find the weights

- We have input vector $x$ of word $w_j$ represented as a one-hot vector
  - One element = 1, and all the others equal 0
- Predict probability of each of the output words in 3 steps:
  1. Select embedding from $W$: $x$ is multiplied by $W$ to give the hidden (projection) layer
  2. Compute dot product $c_k \cdot v_j$: for each of the context words, multiply the projection vector by context matrix $C$. This produces a $1 \times |V|$ dimensional output vector with a score for each word in $V$
  3. Normalize dot products into probabilities: using soft-max

Skip-gram as neural network, cont’d

The resulting network looks like this:

If we look at just one output unit, we have a yes-no classifier

- Just like the one from lecture 19 (on slide 13)
  - For the ‘good’ context words we want ‘yes’ results
  - For the ‘bad’ context words we want ‘no’ results
What we actually want

A reminder that we’re not really interested in the (logistic regression) classifier
• or the neural network context predictor

We just want their weights to use as word embeddings
We can either just use the weights from $W$
• discarding $C$ altogether
or
• combine $W$ and $C$

As usual, whether/how we do this is a matter for tuning to the particular application in view
• Which is also the case for the context window size $L$

Some real embeddings

Examples of the closest tokens to some target words using a phrase-based extension of the skip-gram algorithm (Mikolov et al. 2013):

<table>
<thead>
<tr>
<th>Redmond</th>
<th>Havel</th>
<th>ninjutsu</th>
<th>graffiti</th>
<th>capitatele</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redmond</td>
<td>Vaclav Havel</td>
<td>ninja</td>
<td>spray paint</td>
<td>capitated</td>
</tr>
<tr>
<td>Wash</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redmond</td>
<td>President Vaclav Havel</td>
<td>Martial arts</td>
<td>graffiti</td>
<td>capitated</td>
</tr>
<tr>
<td>Washington</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>Velvet Revolution</td>
<td>swordsmanship</td>
<td>taggers</td>
<td>capitating</td>
</tr>
</tbody>
</table>

Properties of Embeddings

Offsets between embeddings can capture relations between words, e.g. number

\[
\text{vector}(\text{king}) + (\text{vector}(\text{woman}) - \text{vector}(\text{man})) \approx \text{vector}(\text{queen})
\]

Singular Value Decomposition (SVD)

[Optional background from here on]

• SVD is a method for finding the most important dimensions of a dataset

• SVD belongs to a family of methods that can approximate an N-dimensional dataset using fewer dimensions
  • Such as Principle Component Analysis (PCA) or Factor Analysis

• First applied in Latent Semantic Analysis (LSA) to tasks generating embeddings from term-document matrices
Singular Value Decomposition cont’d

- Dimensionality reduction methods first rotate the axes of the original dataset into a new space
- The new space is chosen so that the highest order dimension captures the most variance in the original dataset
  - The next dimension captures the next most variance, and so on
- While some information about the relationship between the original points is necessarily lost in the new transformation
  - the remaining dimensions preserve as much as possible of the original setting

Principal Component Analysis

Here’s an example of PCA:

Latent Semantic Analysis (LSA)

- LSA is a particular application of SVD to a $|V| \times c$ term-document matrix $X$ representing $|V|$ words and their co-occurrence with $c$ documents
  - Cell values are some version of normalised counts, e.g. PPMI or tf-idf
- SVD factorises matrix $X$ into the product of three matrices $W$, $\Sigma$ and $C$:
  
  $$
  X = \begin{bmatrix}
  \sigma_1 & 0 & 0 & \ldots & 0 \\
  0 & \sigma_2 & 0 & \ldots & 0 \\
  0 & 0 & \sigma_3 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & 0 & \ldots & \sigma_m \\
  \end{bmatrix}
  \begin{bmatrix}
  W \\
  \end{bmatrix}
  \begin{bmatrix}
  C \\
  \end{bmatrix}
  $$

- $W$: a $|V| \times m$ matrix, where each row $w$ represents a word and each column represents $m$ dimensions in a latent space
  - The $m$ column vectors are orthogonal to each other and are ordered by the amount of variance in the original dataset
  - $m = \text{rank of } X$ (number of linearly independent rows)
- $\Sigma$: a diagonal $m \times m$ matrix with singular values along the diagonal, expressing the importance of each dimension
- $C^T$: a diagonal $m \times c$ matrix, where each row now represents one of the latent dimensions and the $m$ row vectors are orthogonal to each other
LSA: Dimensionality reduction

- By using only the top $k$ dimensions of $W$, $\Sigma$ and $C$, the product of these 3 matrices becomes a least-squares approximation to the original $X$.
- Since the higher dimensions encode the most variance, SVD models the most important information in the original $X$.
- That is, $W$ gives us lower-dimension embeddings (fewer rows) for all the words (same number of columns).

LSA: Dimensionality reduction, cont’d

- Taking only the top $k \leq m$ dimensions after SVD is applied to the co-occurrence matrix $X$:

$$
\begin{bmatrix}
X \\
|V| \times c
\end{bmatrix} =
\begin{bmatrix}
W_k \\
|V| \times k
\end{bmatrix} \begin{bmatrix}
\sigma_1 & 0 & 0 & \ldots & 0 \\
0 & \sigma_2 & 0 & \ldots & 0 \\
0 & 0 & \sigma_3 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \ldots & \sigma_k \\
k \times k & k \times c
\end{bmatrix} \begin{bmatrix}
C \\
k \times c
\end{bmatrix}
$$

Word embedding vs. document classification

- Notice that because what we care about is words, in both $X$ and $W$ each row corresponds to a distinct word type.
  - This is in contrast to last week’s lab, where we were interested in classifying documents, and each column corresponded to a different word type.

SVD and LSA

- Using only the top $k$ dimensions leads to a reduced $W$ matrix, with one $k$-dimensioned row per word.
- This row acts as a dense $k$-dimensional vector (embedding) representing that word.
- LSA embeddings generally set $k = 300$.
- LSA applies a particular weighting for each co-occurrence cell that multiplies two weights: local and global.
**LSA term weighting**

- The local weight of each term $i$ in document $j$ is its log frequency: $\log f(i,j) + 1$

- The global weight of term $i$ is a version of its entropy with respect to the entire document collection:

$$1 + \frac{\sum_j p(i,j) \log p(i,j)}{\log D}$$

- where $D$ is the number of documents

---

**SVD and word-context**

- In LSA, SVD is applied to a term-document matrix

- An alternative is to apply SVD to a word-word or word-context matrix – the context dimensions are words (rather than documents as in LSA)

- Relies on PPMI-weighted word-word matrix

- Only top dimensions are used – truncated SVD

---

**Word embeddings in truncated SVD**

1) SVD:

$$X = W \Sigma C$$

2) Truncation:

$$W'$$

3) Embeddings:

$$W_k$$