An introduction to word embeddings

W4705: Natural Language Processing

Fei-Tzin Lee

September 23, 2019

Today

1. What are these word embedding things, anyway?
2. Distributional semantics
3. word2vec
4. Analogies with word embeddings

Today

1. What are these word embedding things, anyway?
2. Distributional semantics
3. word2vec
4. Analogies with word embeddings

Representing knowledge

Humans have rich internal representations of words that let us do all sorts of intuitive operations, including (de)composition into other concepts.

- “parent’s sibling” = ‘aunt’ - ‘woman’ = ‘uncle’ - ‘man’
- The attribute of a banana that is ‘yellow’ is the same attribute of an apple that is ‘red’.

But this is obviously impossible for machines. There’s no numerical representation of words that encodes these sorts of abstract relationships.

Representing knowledge

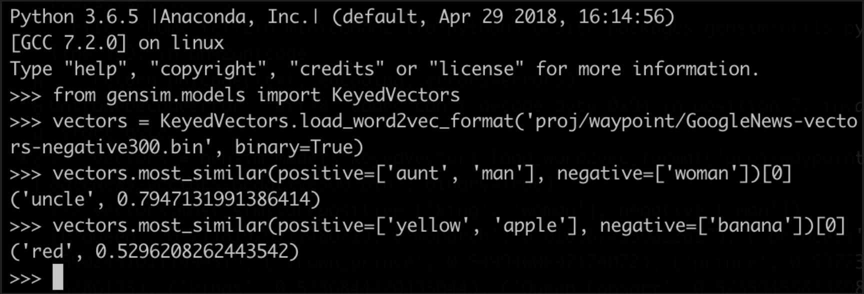
Humans have rich internal representations of words that let us do all sorts of intuitive operations, including (de)composition into other concepts.

- “parent’s sibling” = ‘aunt’ - ‘woman’ = ‘uncle’ - ‘man’
- The attribute of a banana that is ‘yellow’ is the same attribute of an apple that is ‘red’.

But this is obviously impossible for machines. There’s no numerical representation of words that encodes these sorts of abstract relationships.

...Right?

A bit of magic



```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 29 2018, 16:14:56)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gensim.models import KeyedVectors
>>> vectors = KeyedVectors.load_word2vec_format('proj/waypoint/GoogleNews-vecto
rs-negative300.bin', binary=True)
>>> vectors.most_similar(positive=['aunt', 'man'], negative=['woman'])[0]
('uncle', 0.7947131991386414)
>>> vectors.most_similar(positive=['yellow', 'apple'], negative=['banana'])[0]
('red', 0.5296208262443542)
>>> 
```

Figure: Output from the gensim package using word2vec vectors pretrained on Google News.

- This is not a fancy language model
- No external knowledge base
- Just vector addition and subtraction with cosine similarity

A bit of ~~magic?~~ math

Where did these magical vectors come from? This trick works in a few different flavors:

- SVD-based vectors
- word2vec, from the example above, and other neural embeddings
- GloVe, something akin to a hybrid method

Word embeddings

The semantic representations that have become the de facto standard in NLP are **word embeddings**, vector representations that are

- Distributed: information is distributed throughout indices (rather than sparse)
- Distributional: information is derived from a word's distribution in a corpus (how it occurs in text)

These can be viewed as an *embedding* from a discrete space of words into a continuous vector space.

Applications

- Language modeling
- Machine translation
- Sentiment analysis
- Summarization
- etc...

Basically, anything that uses neural nets can use word embeddings too, and some other things besides.

Today

1. What are these word embedding things, anyway?
2. Distributional semantics
3. word2vec
4. Analogies with word embeddings

Words.

What is a word?

Words.

What is a word?

- A composition of characters or syllables?

Words.

What is a word?

- A composition of characters or syllables?
- A pair - usage and meaning.

Words.

What is a word?

- A composition of characters or syllables?
- A pair - usage and meaning.

These are independent of representation. So we can choose what representation to use in our models.

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

- Bag-of-words? (no word order)

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

- Bag-of-words? (no word order)
- Sequences of numerical indices? (relatively uninformative)

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

- Bag-of-words? (no word order)
- Sequences of numerical indices? (relatively uninformative)
- One-hot vectors? (space-inefficient; curse of dimensionality)

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

- Bag-of-words? (no word order)
- Sequences of numerical indices? (relatively uninformative)
- One-hot vectors? (space-inefficient; curse of dimensionality)
- Scores from lexicons, or hand-engineered features? (expensive and not scalable)

So, how?

How do we represent the words in some segment of text in a machine-friendly manner?

- Bag-of-words? (no word order)
- Sequences of numerical indices? (relatively uninformative)
- One-hot vectors? (space-inefficient; curse of dimensionality)
- Scores from lexicons, or hand-engineered features? (expensive and not scalable)

Plus: none of these tell us how the word is used, or what it actually means.

What *does* a word mean?

Let's try a hands-on exercise.

What *does* a word mean?

Let's try a hands-on exercise.

Obviously, word meaning is really hard for even humans to quantify. So how can we possibly generate representations of word meaning automatically?

We approach it obliquely, using what is known as the distributional hypothesis.

The distributional hypothesis

Borrowed from linguistics: the meaning of a word can be determined from the contexts it appears in.[1]

- Words with similar contexts have similar meanings (Harris, 1954)
- "You shall know a word by the company it keeps" (Firth, 1957)

Example: "My homework was no archteryx of academic perfection, but it sufficed."

[1] https://aclweb.org/aclwiki/Distributional_Hypothesis

Context?

Most static word embeddings use a simple notion of context - a word is a "context" for another word when it appears close enough to it in the text.

- But we can also use sentences or entire documents as contexts.

In the most basic case, we fix some number of words as our 'context window' and count all pairs of words that are less than that many words away from each other as co-occurrences.

Example time!

Let's say we have a context window size of 2.

no raw meat pants, please.
please do not send me some raw
vegetarians.

What are the co-occurrences of 'send'?

Example time!

Let’s say we have a context window size of 2.

no raw meat pants, please.
please do not send me some raw
vegetarians.

What are the co-occurrences of ‘send’?

- ‘do’
- ‘not’
- ‘me’
- ‘some’

The co-occurrence matrix

We can collect the context occurrences in our corpus into a *co-occurrence matrix* M_{ij}, where each row corresponds to a word and each column to a context word.

Then the entry M_{ij} represents how many times word i appeared within the context of word j.

Example II: the example continues

no raw meat pants, please.
please do not send me some raw
vegetarians.

What does the co-occurrence matrix look like?

Other count-based representations

We can generalize this concept a bit further.

- More general contexts - sentences or documents
- Weighted context windows
- Other measures of word-context association (e.g. pointwise mutual information)

This gives us the more general notion of a word-context matrix.

Embedding with matrix factorization

Goal: find a vector for each word w_i and context c_j such that $\langle \vec{w}, \vec{c} \rangle$ approximates M_{ij} - that is, the association between w_i and c_j.

Of course, this is easy if we get to use arbitrary-length vectors. But we ideally want low-dimensional representations.

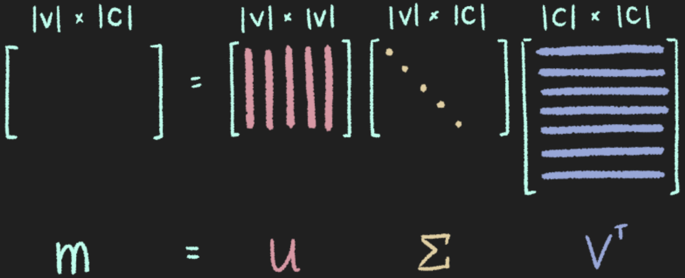
How can we do this? Use the singular value decomposition.

Truncated SVD

Recall that the SVD of a matrix M gives us

$$M = U\Sigma V^T.$$

To approximate M, we truncate to the top k singular values.

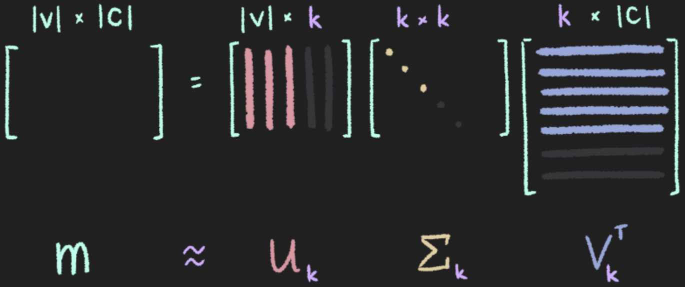


Truncated SVD

Recall that the SVD of a matrix M gives us

$$M = U\Sigma V^T.$$

To approximate M, we truncate to the top k singular values.



Why truncated SVD?

The truncation

$$M_k = U_k \Sigma_k V_k^T$$

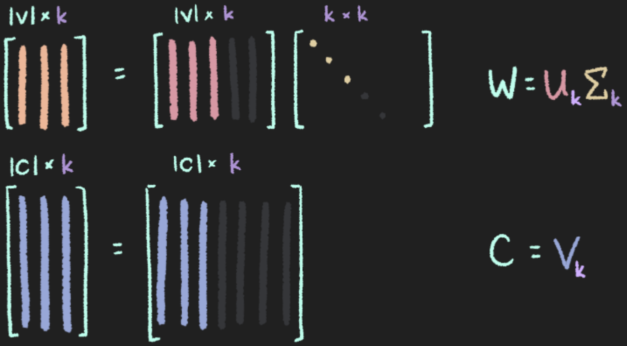
is the best approximation to M under Frobenius norm.

We can view word vectors $\vec{w}_i$ and context vectors $\vec{c}_j$ as rows of matrices W and C.

- For W, C with rows of length k, their product WC^T can't approximate M better than M_k.

Embedding with truncated SVD

Of course, truncated SVD has three components, but we only want two matrices, W and C. Traditionally, we set $W = U_k\Sigma_k$ and $C = V_k$.



Variations on SVD

Note that this preserves inner products between word vectors (rows of M). But W and C are asymmetric. There is no a priori reason that only C should be orthogonal.

Instead, using symmetric word and context matrices works better in practice.

- Split Σ: $W = U_k \Sigma_k^{1/2}$, $C = V_k \Sigma_k^{1/2}$
- Omit Σ altogether: $W = U_k, C = V_k$

In conclusion?

Simple matrix factorization can actually give us decent word representations.

Today

1. What are these word embedding things, anyway?
2. Distributional semantics
3. word2vec
4. Analogies with word embeddings

Neural models of semantics

In essence, word2vec approximates distributional semantics with a neural objective.

Two flavors: skipgram and continuous-bag-of-words (CBOW).
Today, we will focus on *skipgram with negative sampling.*

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.

humans are topologically equivalent to
several donuts in a trenchcoat

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.

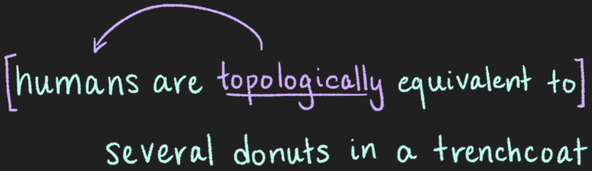
[humans are topologically equivalent to]

several donuts in a trenchcoat

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.



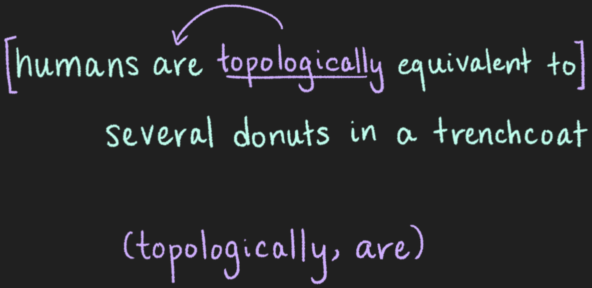
[humans are topologically equivalent to] several donuts in a trenchcoat

(topologically, humans)

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.



[humans are topologically equivalent to]

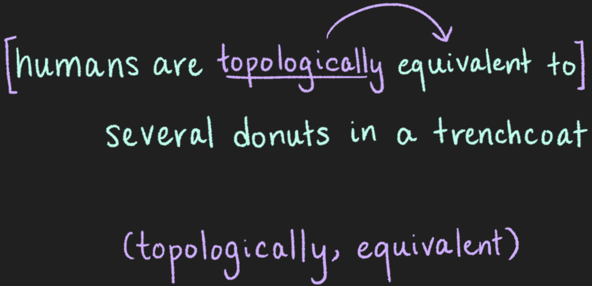
several donuts in a trenchcoat

(topologically, are)

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

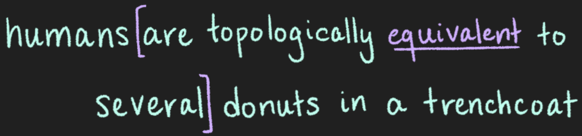
But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.


[humans are topologically equivalent to]
several donuts in a trenchcoat
(topologically, equivalent)

Context, revisited

When using the matrix-factorization approach, we collect all co-occurrences globally into the same matrix M.

But word2vec takes a slightly different approach - it slides a window over the corpus, essentially looking at one isolated co-occurrence pair at a time.



humans [are topologically equivalent to several] donuts in a trenchcoat

The setting

We start with the following things:

- A corpus D of co-occurrence pairs (w, c)
- A word vocabulary V
- A context vocabulary C

Skipgram, intuitively

Idea: given a word, predict what context surrounds it.

More specifically, we want to model the probabilities of context words appearing around any specific word.

Skipgram, concretely

Our global learning objective is to maximize the probability of the observed corpus under our model of co-occurrences:

$$\max \prod_{(w,c)\in D} P(w,c)$$

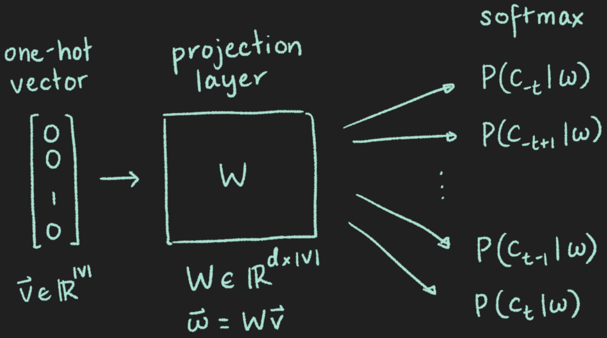
Equivalently, we may maximize the log of this quantity, which gives us the more tractable

$$\max \sum_{(w,c)\in D} \log P(w,c).$$

So how exactly do we calculate $P(w,c)$?

Skipgram's modeling assumption

Vanilla skipgram models log $P(w, c)$ as proportional to $\langle \vec{w}, \vec{c} \rangle$.



What the model actually learns are the vector representations for words and contexts that best fit the distribution of the corpus.

Negative sampling

Since probabilities must sum to 1, we must normalize $e^{\langle w,c\rangle}$ by some factor. In plain skipgram, we use a softmax over all possible context words:

$$\log P(w,c) = \log \frac{e^{\langle \vec{w},\vec{c}\rangle}}{\sum_{c'\in C} e^{\langle \vec{w},\vec{c}'\rangle}}.$$

This denominator is computationally intractable. For efficiency's sake, negative sampling (based on noise-contrastive estimation) replaces this with a sampled approximation:

$$\log P(w,c) \approx \log \sigma(\vec{w}\cdot\vec{c}) + \sum_{i=1}^{k} \mathbb{E}_{c_i'\sim P_n(c)}[\log \sigma(-\vec{w}\cdot\vec{c}_i')].$$

The negative sampling objective

So the SGNS objective for each pair (w, c) is

$$\log P(w,c) = \log \sigma(\vec{w} \cdot \vec{c}) + \sum_{i=1}^{k} \mathbb{E}_{c_i' \sim P_n(c)}[\log \sigma(-\vec{w} \cdot \vec{c_i'})]$$

$$= \log \sigma(\vec{w} \cdot \vec{c}) + \sum_{i=1}^{k} \mathbb{E}_{c_i' \sim P_n(c)}[\log(1 - \sigma(\vec{w} \cdot \vec{c_i'})].$$

Tricks of the trade

In practice, SGNS also uses a couple other tricks to make things run more smoothly.

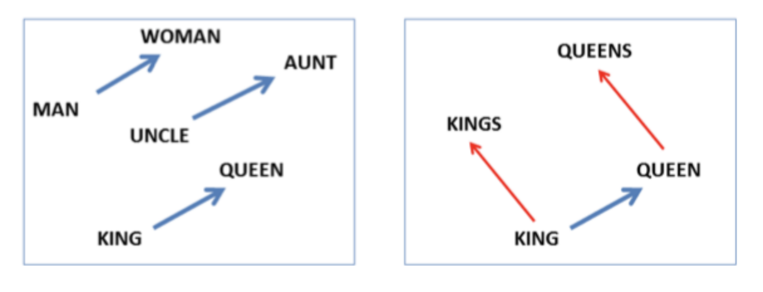
- Instead of the empirical unigram distribution, use the unigram distribution raised to the 3/4th power for noise
- Frequent word subsampling: discard words from the training set with probability

  $$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}},$$

  where t is a small threshold.

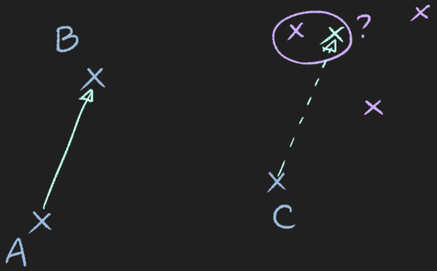
Linear composition, more formally

One of the most exciting results from word2vec is that vectors seem to exhibit some degree of *additive compositionality* - you can compose words by adding their vectors, and remove concepts by subtracting them.



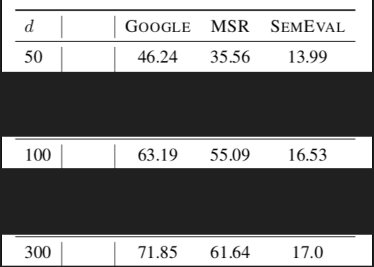
Additive analogy solving

This lets us solve analogies in the form of A:B :: C:? in a very simple way: take $\vec{B} - \vec{A} + \vec{C}$, and find the closest word to the result.



Usually when finding the closest word we use cosine distance.

Additive analogy solving



d		GOOGLE	MSR	SEMEVAL
50		46.24	35.56	13.99
100		63.19	55.09	16.53
300		71.85	61.64	17.0

Figure: From https://www.aclweb.org/anthology/N18-2039.

Why?

Why might this work?

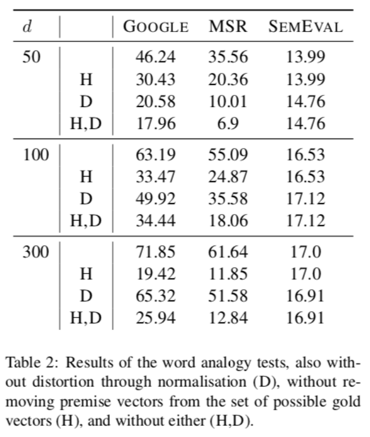
Actually...

Additive analogy completion uses some hidden tricks.

- Leave out original query points
- Embeddings are often normalized in preprocessing

$$d = \operatorname{argmax}_{w \in V} \cos(\vec{w}, \vec{b} - \vec{a} + \vec{c})$$

Additive analogy solving (full)



d		GOOGLE	MSR	SEMEVAL
50		46.24	35.56	13.99
	H	30.43	20.36	13.99
	D	20.58	10.01	14.76
	H,D	17.96	6.9	14.76
100		63.19	55.09	16.53
	H	33.47	24.87	16.53
	D	49.92	35.58	17.12
	H,D	34.44	18.06	17.12
300		71.85	61.64	17.0
	H	19.42	11.85	17.0
	D	65.32	51.58	16.91
	H,D	25.94	12.84	16.91

Table 2: Results of the word analogy tests, also without distortion through normalisation (D), without removing premise vectors from the set of possible gold vectors (H), and without either (H,D).

Figure: From https://www.aclweb.org/anthology/N18-2039, "The word analogy testing caveat", Schluter (2019).

Linear composition (in reality)

Turns out, if we don't rely on these tricks, performance drops by a lot.

- Leave the query points in? Zero accuracy. (Linzen, 2016)
- Performance is best on words with predictable relations (Finley et al., 2017)
- We can actually do a little better on analogies if we allow general linear transformations instead of just translations (Ethayarajh, 2019), although vector addition is the nice intuitive method

Be careful, but curious

We need to be careful not to get carried away.

But still, the fact that we get any regularity of this sort at all is really exciting!

And word embeddings do give us undeniable performance increases on a wide variety of downstream tasks.

Questions?

Further reading

- Distributional semantics and neural embeddings
- Analogy performance

SVD vs word2vec

Neural Word Embedding as Implicit Matrix Factorization. Levy and Goldberg (2014).

Improving Distributional Similarity with Lessons Learned from Word Embeddings. Levy, Goldberg and Dagan (2015).

Analogies

Issues in evaluating semantic spaces using word analogies. Linzen (2016).

What Analogies Reveal about Word Vectors and their Compositionality. Finley, Farmer and Pakhomov (2017).

The word analogy testing caveat. Schluter (2018).

Co-occurrence through a different lens

SVD uses all the co-occurrence information in a corpus, but is sensitive to zero terms and performs worse than word2vec in practice. On the other hand, word2vec fails to use global co-occurrences.

Setting

We'll consider a co-occurrence matrix M_{ij} that collects co-occurrence counts between words i and contexts j.

Again, we'd like to construct matrices W and C containing word and context vectors respectively - but this time we also learn individual bias terms for each word and context. Instead of directly factorizing M, this time we have a different objective.

Co-occurrence ratios

GloVe targets the *ratios* between co-occurrence counts as the target for learning. Intuitively, this captures context *differences*, rather than focusing on similarities.

So what's our model this time?

We specify some desirable properties that will let us hone in on the appropriate model for $\frac{P_{ik}}{P_{jk}}$

- First: *Enforce* linear vector differences - the model should depend only on $w_i - w_j$

$$F(w_i - w_j, c_k) = \frac{P_{ik}}{P_{jk}}$$

- Second: Require that the model be bilinear in $w_i - w_j$ and c_k; choose the dot product for simplicity

$$G((w_i - w_j)^T c_k) = G(w_i^T c_k - w_j^T c_k) = \frac{P_{ik}}{P_{jk}}$$

Constructing the GloVe model

But, wait - in GloVe's setting, words and contexts are symmetric.

- We'd like the *ik* term to depend on the $w_i^T c_k$, and likewise with *jk* in the denominator; we can do this by setting

$$G((w_i - w_j)^T c_k) = \frac{G(w_i^T c_k)}{G(w_j^T c_k)}.$$

- This gives us $G = \exp$, or

$$w_i^T c_k = \log P_{ik} = \log(X_{ik}) - \log(X_i).$$

Then we can symmetrize by adding a free bias term w_k to mop up the $\log(X_i)$.

GloVe's objective

All in all, this works out to a log-bilinear model with least-squares

$$J = \sum_{w,c} f(X_{ij})(\langle \vec{w}, \vec{c} \rangle + b_w + b_c - log(X_{ij}))^2,$$

where f is a weighting function that we can choose.

How do we know if word embeddings are good?

Two main flavors of evaluation: *intrinsic* and *extrinsic* measures.

- Intrinsic evaluation: word similarity or analogy tasks
- Extrinsic evaluation: performance on downstream tasks

Issues with intrinsic evaluation

- Similarity and analogy tasks may be too homogeneous
- Intrinsic and extrinsic measures may not correlate