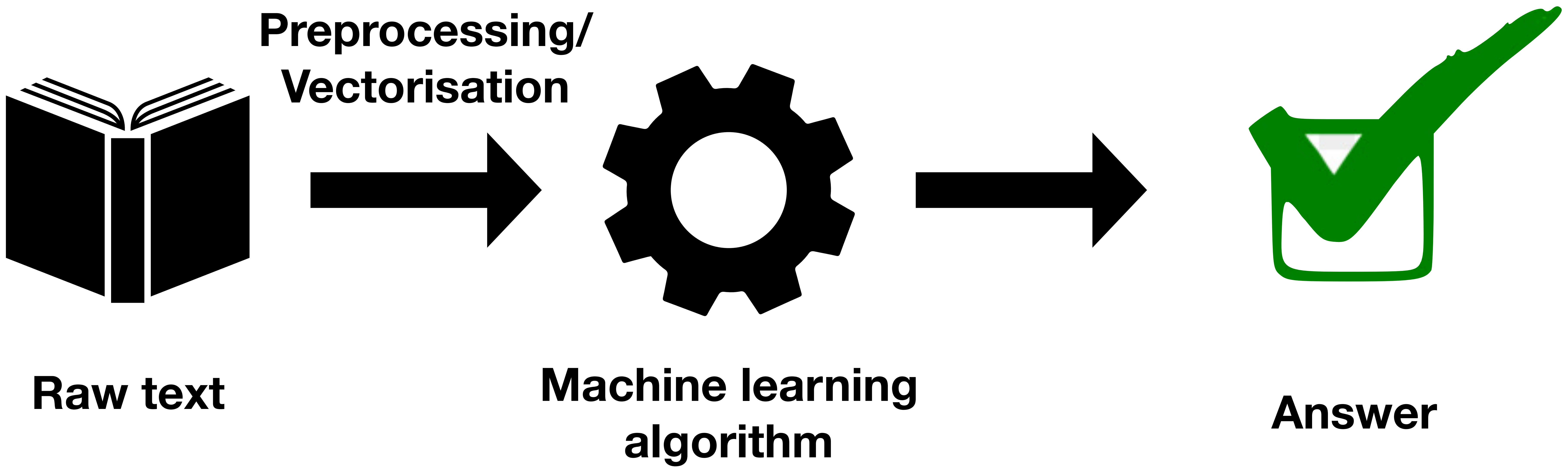
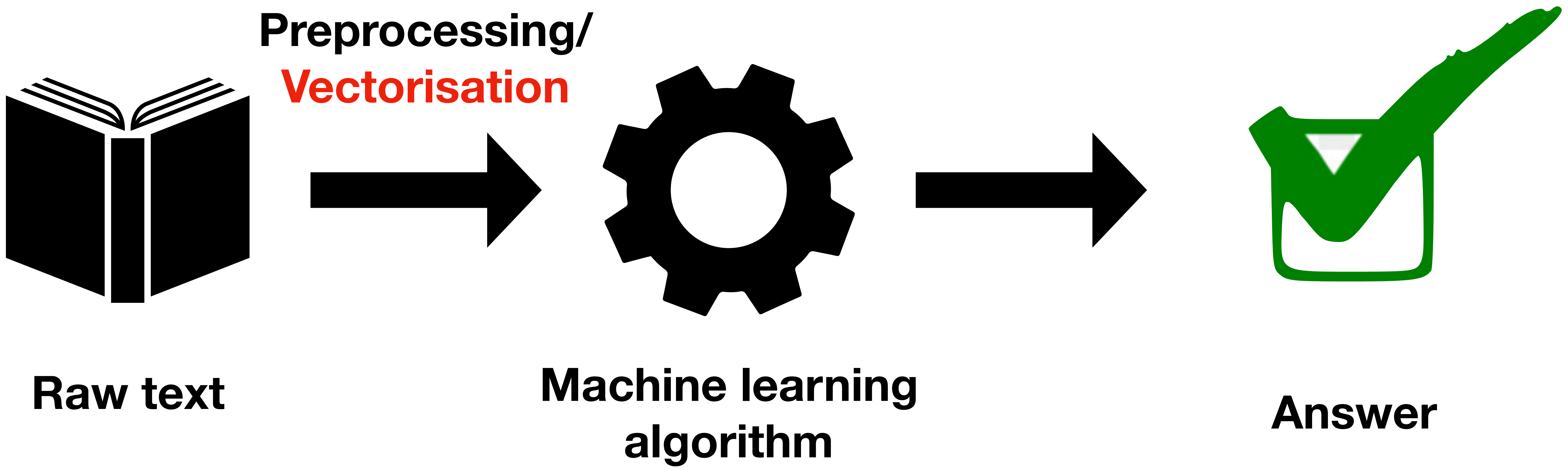
Vector embeddings

Eugeny Malyutin

So we want to solve NLP classification task:



So we want to solve NLP classification task:



Previously:

motel [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] AND
hotel [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0] = 0

$$TF - IDF(w, d, C) = \frac{count(w, d)}{count(d)} * log(\frac{\sum_{d' \in C} 1(w, d')}{|C|})$$

One-hot encoding drawback:

- “monkeys eat bananas” and “apes consume fruits” - similarity equals to 0
- «Pouteria is widespread throughout the tropical regions of the world and monkeys eat their fruits»(c). What is Pouteria? Is it a tree?
- «a word is characterized by the company it keeps» – John Rupert Firth
- Ideally, we want vector representations where ***similar words*** end up with ***similar vectors***. **Dense** vectors. And when I say similar a mention some **similarity measure** (cosine).
- Even better, we’d want more similar representations when the words share some properties such as if they’re both plural or singular, verbs or adjectives or if they both reference to a male.

How can we build our vectors?

- Ok, let us imagine that this matrix can be de-composed into two separ… Stop, it's another story (look at pLSA/LDA)

- The co-occurrence number alone is not a good number to measure the co-occurrence probability of two words because it does not take into account how many times each of them occur. («the monkey»)

- This PMI method leads to many log(0) (i.e. $-\infty$) entries (every time two words do not co-occur).

- The computation time in order to count all this is very expensive, especially if it's done naively. Fortunately, there are ways to do this requiring just a single pass through the entire corpus to collect the statistics.

Co-occurrence matrix	**I**	**love**	**monkeys**	**and**	**apes**	**bananas**
I	0	1	1	0	0	0
love	1	0	2	1	0	1
monkeys	1	2	0	1	1	1
and	0	1	1	0	1	0
apes	0	0	1	1	0	0
bananas	0	1	1	0	0	0

$$PMI(w,c) = \log \frac{\hat{P}(w,c)}{\hat{P}(w)\hat{P}(c)} = \log \frac{\#(w,c)\cdot|D|}{\#(w)\cdot\#(c)}$$

How can we build our vectors?

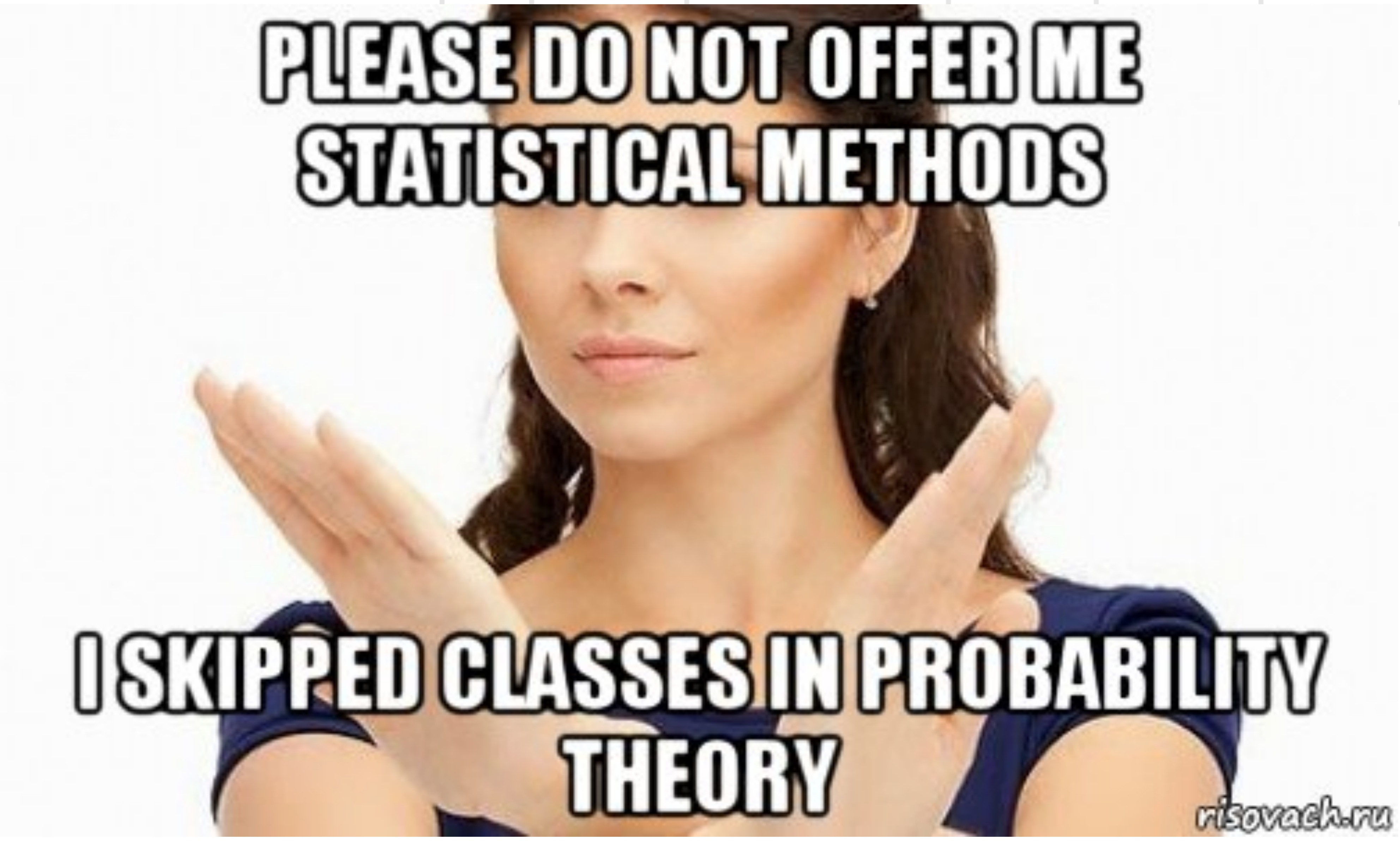
- This PMI method leads to many log(0) (i.e. $-\infty$) entries (every time two words do not co-occur).

- The computation time in order to count all this is very expensive, especially if it's done naively. Fortunately, there are ways to do this requiring just a single pass through the entire corpus to collect the statistics.

- And then (in 2013) Tomas Mikolov came and saved everyone.
  //Mikolov T. et al. Distributed representations of words and phrases and their compositionality //Advances in neural information processing systems. – 2013. – C. 3111-3119.

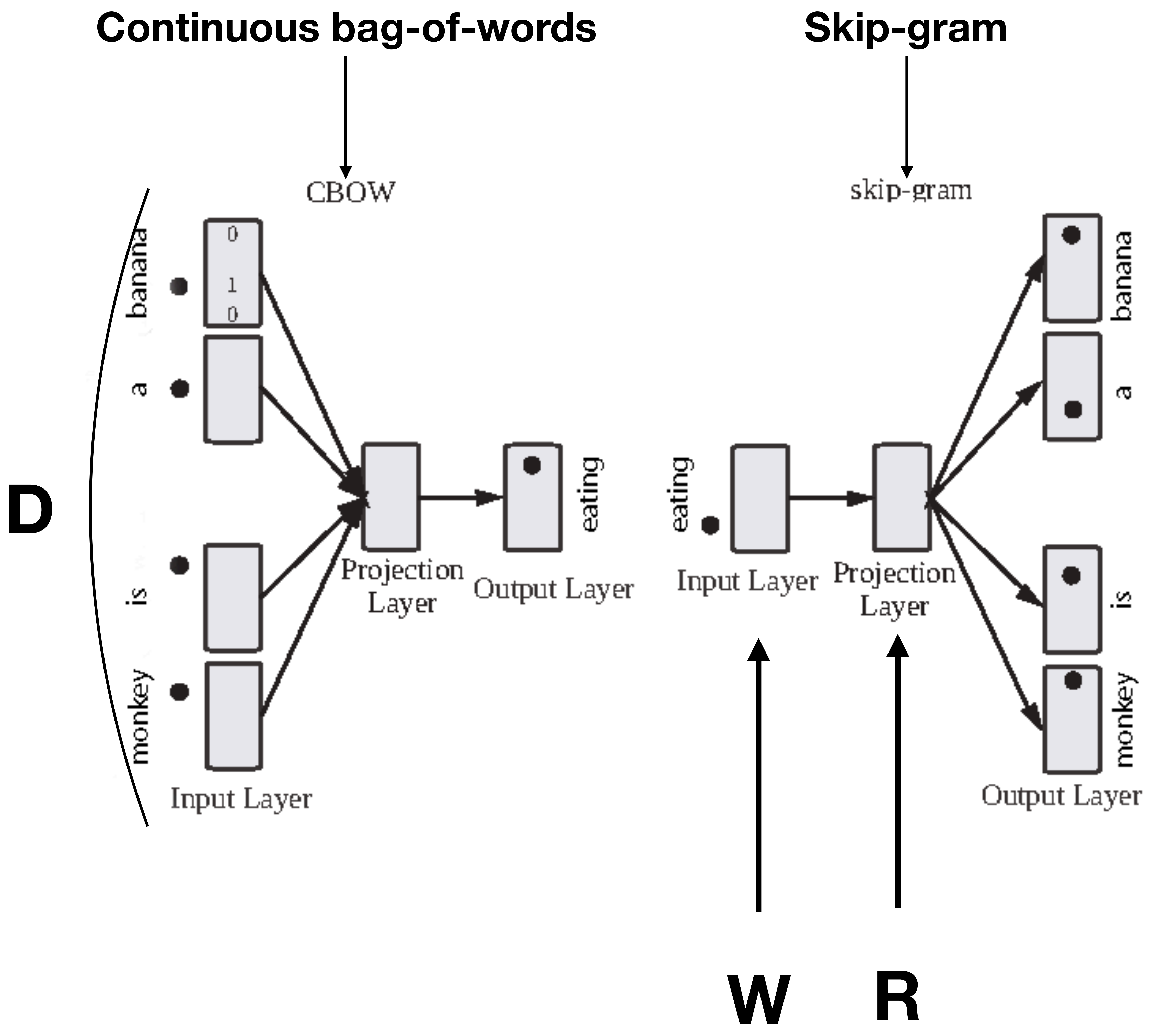
Co-occurrence matrix	**I**	**love**	**monkeys**	**and**	**apes**	**bananas**
I	0	1	1	0	0	0
love	1	0	2	1	0	1
monkeys	1	2	0	1	1	1
and	0	1	1	0	1	0
a	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
b	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

STATISTICAL WAYS



Word2vec scheme:

- It has an input layer that receives **D** one-hot encoded words which are of dimension **V** (the size of the vocabulary).
- It «averages» them, creating a single input vector.
- That input vector is multiplied by a weights matrix ***W*** (that has size VxD, being D nothing less than the dimension of the vectors that you want to create). That gives you as a result a D-dimensional vector.
- The vector is then multiplied by another matrix (**R** - reverse W), this one of size DxV. The result will be a new V-dimensional vector.
- That V-dimensional vector is normalized to make all the entries a number between 0 and 1, and that all of them sum 1, using the softmax function, and that's the output. It has in the i-th position the predicted probability of the i-th word in the vocabulary of being the one in the middle for the given context.



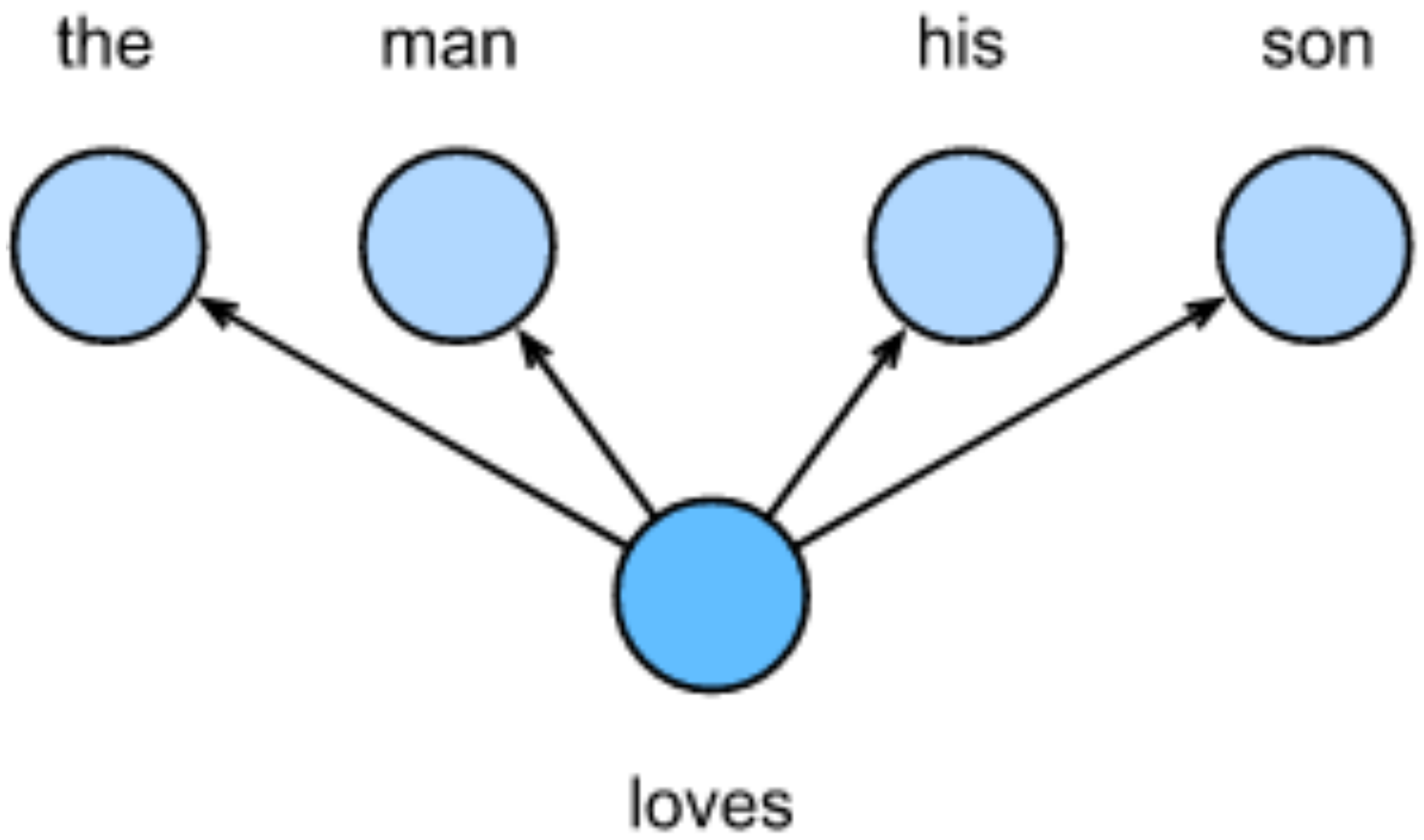
The skipgram model

- We assume that, given the central target word, the context words are generated independently of each other.
  P(the, man, his, son | loves) = P(the | loves) * P(man | loves) * P(his | loves) * P(son | loves)
- And $p(w_o | w_c) = \dfrac{exp(u_o^T v_c)}{\sum_{i \in V} exp(u_i^T v_c)}$ **cond. probability**, u and v — vector representations.

  u_0 - context,
  v_c — central target.

- The **likelihood function** of the skip-gram model:

$$\prod_{i=1}^{T} \prod_{-m \leq j \leq m} P(w^{(t+j)} | w^t)$$



Skipgram model training

- Loss function $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \textbf{log}\, \mathbb{P}(w^{(t+j)} \mid w^{(t)})$

- If we want to SGD it - we need to compute gradient of conditional probability:

$$\log \mathbb{P}(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i \in \mathcal{V}} \textbf{exp}(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- Through differentiation, we can get the gradient from the formula above.

$$\begin{aligned}\frac{\partial \textbf{log}\, \mathbb{P}(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\textbf{exp}(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \textbf{exp}(\mathbf{u}_i^\top \mathbf{v}_c)}\right)\mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid w_c)\mathbf{u}_j .\end{aligned}$$

- Any problems?

Skipgram model training

- Loss function $-\sum_{t=1}^{T} \sum_{-m\leq j\leq m,\ j\neq 0} \mathbf{log}\, \mathbb{P}(w^{(t+j)} \mid w^{(t)})$

- If we want to SGD it - we need to compute gradient of conditional probability:

$$\log \mathbb{P}(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i\in\mathcal{V}} \mathbf{exp}(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- Through differentiation, we can get the gradient from the formula above:

$$\begin{aligned}\frac{\partial \mathbf{log}\, \mathbb{P}(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j\in\mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i\in\mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\\ &= \mathbf{u}_o - \sum_{j\in\mathcal{V}} \left(\frac{\mathbf{exp}(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i\in\mathcal{V}} \mathbf{exp}(\mathbf{u}_i^\top \mathbf{v}_c)}\right)\mathbf{u}_j\\ &= \mathbf{u}_o - \sum_{j\in\mathcal{V}} \mathbb{P}(w_j \mid w_c)\mathbf{u}_j\,.\end{aligned}$$

- Its computation obtains the conditional probability for **all the words in the dictionary** given the central target word w_c
We then use **the same method** to obtain the gradients **for other word vectors**.

Negative sampling:

- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from
$$\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c),$$
- Now we consider maximizing the joint probability $\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)})$.
- However, the events included in the model only consider positive examples. We need to sample additional negative events (never occurred in the same context) and then:

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k) \,.$$

Negative sampling:

- Now we consider maximizing the joint probability $$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \,.$$
- However, the events included in the model only consider positive examples. We need to sample additional negative K events (never occurred in the same context) and then:

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k) \,.$$

- The logarithmic loss for the conditional probability above is $$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \mathbb{P}(D = 0 \mid w^{(t)}, w_k)$$
- Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K

$$= -\log\, \sigma\left(\mathbf{u}_{i_{t+j}}^{\top} \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^{\top} \mathbf{v}_{i_t}\right)\right)$$

$$= -\log\, \sigma\left(\mathbf{u}_{i_{t+j}}^{\top} \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^{\top} \mathbf{v}_{i_t}\right) .$$

Negative sampling:

- The logarithmic loss for the conditional probability above is

$$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D=1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \mathbb{P}(D=0 \mid w^{(t)}, w_k)$$

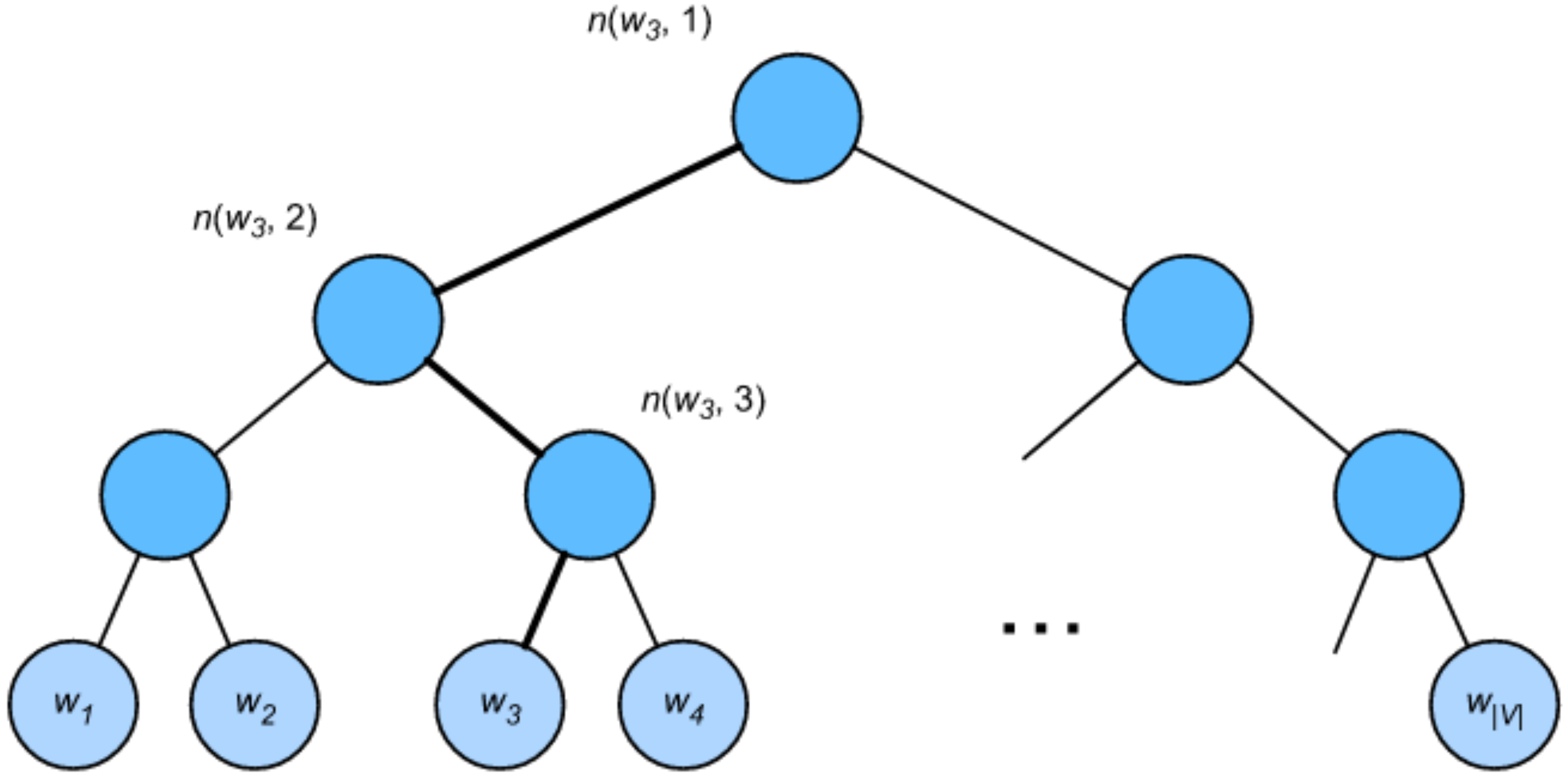
$$= -\log\, \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right)\right)$$

$$= -\log\, \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right).$$

- Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K

- **Key idea:** sample additional negatives and learn your positive probabilities «opposite to» them.

Hierarchical softmax:



- $L(w)$ — the number of nodes on the path (including the root and leaf nodes)
- n_(w,j) — the jth node on this path, with the context word vector $\mathbf{u}$_$(n(w,j))$
- will approximate the conditional probability in the skip-gram model as:

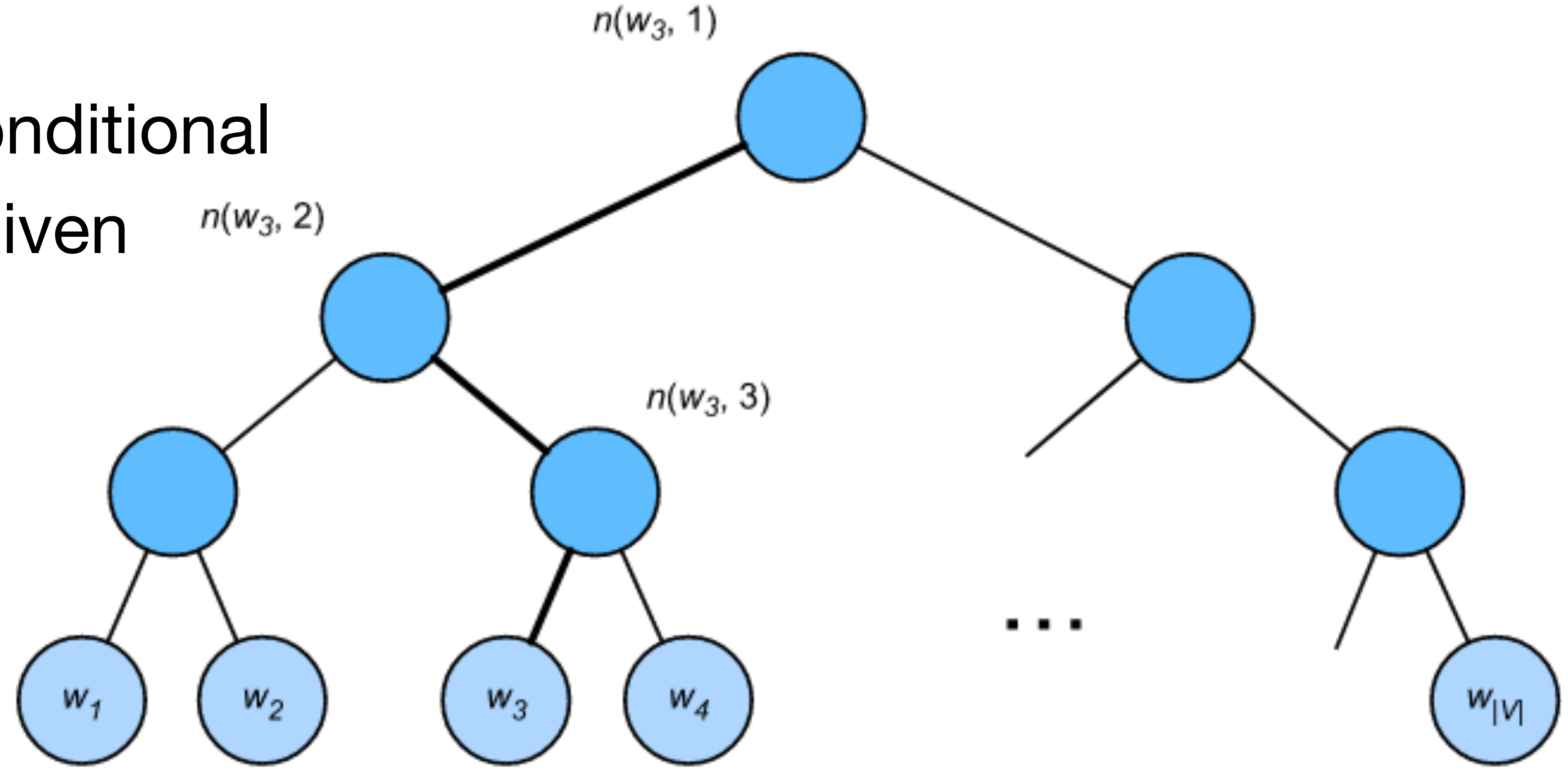
$$\mathbb{P}(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \textbf{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c \right),$$

- And for w3:

$$\mathbb{P}(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c)\,.$$

Hierarchical softmax:

- $\sigma(x)+\sigma(-x)=1$, the condition that the sum of the conditional probability of any word generated based on the given central target word $$\sum_{w \in \mathcal{V}} \mathbb{P}(w \mid w_c) = 1.$$



- What is u_n(w_3, 2) (for example) — separate vectors we should learn (lurk refs for moar math)

- HSoftmax reduce softmax calculation from O(n) to O(log(n)) where n = |V|

- We can also use Huffman trees to encode more frequent words with shorter paths

So what? (Synonyms)

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

- *get_similar_tokens* — top-K words by cosine measure to the target word;
- glove_6b50d — glove model on some common corpora (Wikipedia?) with 6B of words and vector dimension equals to 50;

So what? (2) (Finding Analogies)

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

"Capital-country" analogy

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

"Adjective-superlative adjective" analogy

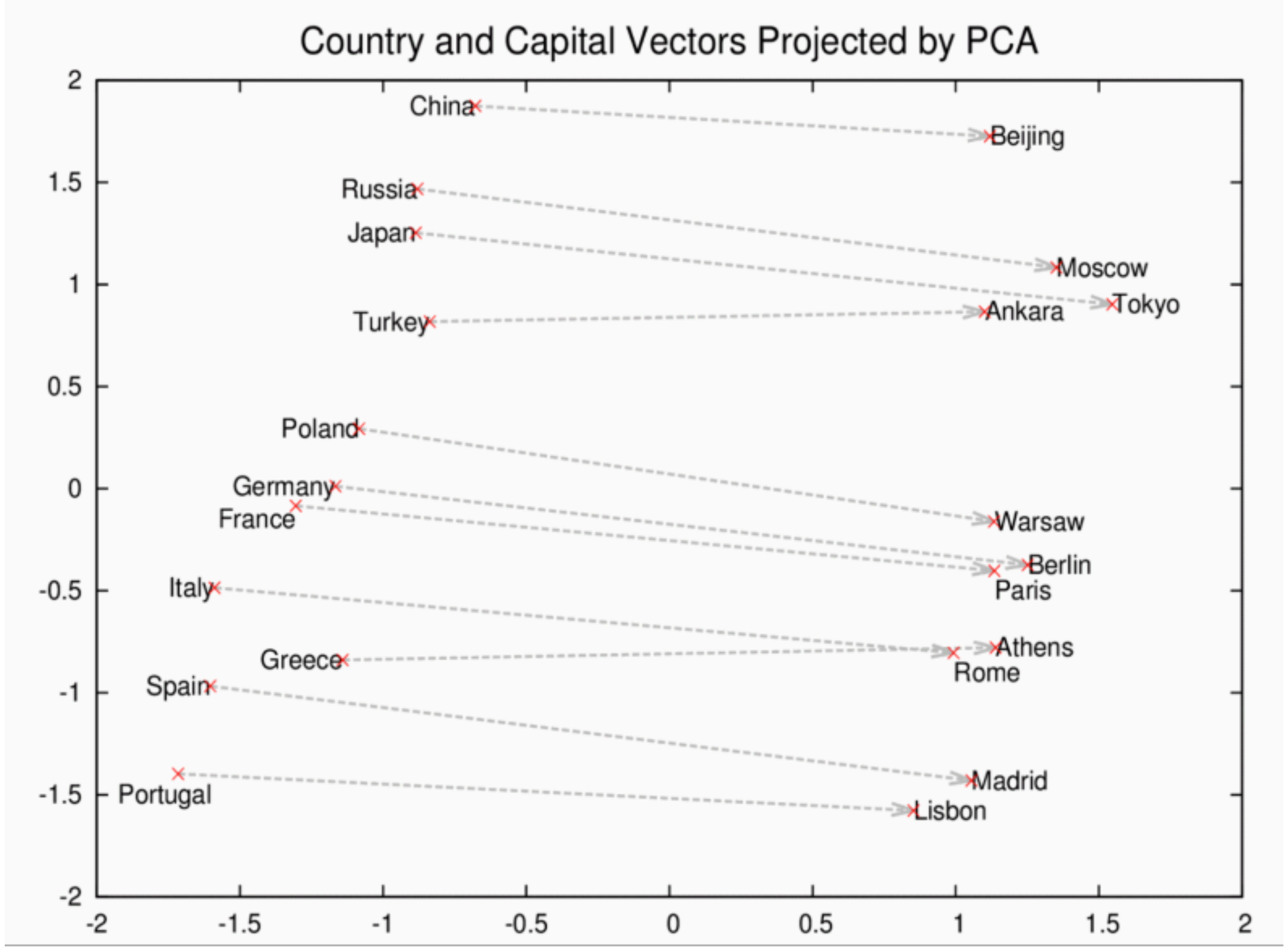
```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

"Present tense verb-past tense verb" analogy

- And it's only x = $\mathbf{vec}(c) + \mathbf{vec}(b) - \mathbf{vec}(a)$
- And then top word for x

So What? (country-capital)



Based on Wikipedia training corpora

Any problems?

- Out-of-vocabulary
- How we can train it? How big our doc’s collection should be?
- Stop, firstly we talk about **text** and word2vec is about **words**

Any problems?

- Out-of-vocabulary

  Yeah, it’s true. But there are few extensions; (fastText)

- Stop, firstly we talk about **text** and word2vec is about **words**

  Ok, average it; Or average with weights; Or do not average and learn some averaging embedding; (look to BERT model)

- How we can train it? How big our doc’s collection should be?

  Really big; Starting from 10+M of symbols; Use pertained vectors;

FastText

- First, we add the special characters “<” and “>” at the beginning and end of the word to distinguish the subwords used as prefixes and suffixes.

- Then, we treat the word as a sequence of characters to extract the n-grams.

$$where = \{\textbf{"<wh"}, \textbf{"whe"}, \textbf{"her"}, \textbf{"ere"}, \textbf{"re>"},\} + \textbf{"<where>"}$$

- $$\mathbf{u}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g \,.$$

- And there rest as in skiagram model;

- Any thoughts?

- It needs a **MUCH MORE** space to store the model (8Gb vs 1-2Gb)

- It needs a **MUCH MORE** corpora to train sufficient model (billions vs millions of symbols)

- It allows us to approximate our unknown word by n-grams it contains;
  For example «ЧЕБУПЕЛИ» by known «ЧЕБУРЕКИ» and «ПЕЛЬМЕНИ»

Word to text:

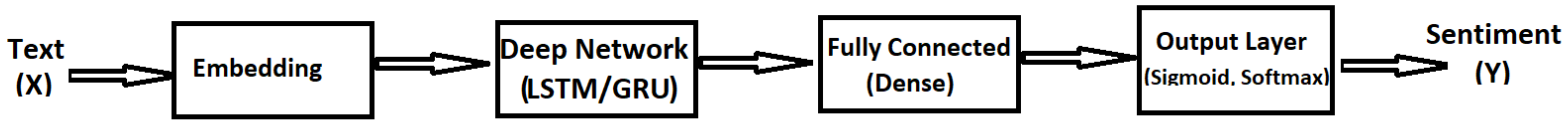
- Average words vectors:
  - More words you averages -> more un-informative representation you get (kind of dimensionality curse). Practically starts from 5-10 words;
- Average words vectors with some weights (TF-IDF):
  - Same problems, yeah. Start 10-15 words;
- Try to «learn» your text’s embeddings from word embeddings:
  - Bring LDA-like approach to word2vec (glove);
  - Use transformers-attention-trillions of TPU’s and spend all money you have on AWC (BERT);

Word2Vec myths:

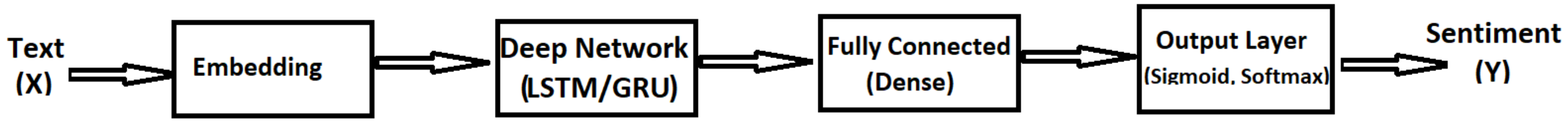
- Word2vec is the best word vector algorithm
- Word vectors are created by deep learning
- Word vectors are used only with deep learning
- Statistical and predictive methods have nothing to do with each other
- There’s a perfect set of word vectors that can be used in every NLP project

Use cases:

- There are some semantic-oriented task (for example topic classification) and you have not a lot of data (you have a need to bring outer semantic info into your corpora)
- You need to build quick (and maybe not so good) similarity measurement for your recommendation system. (works great for a cold start)
- You can use vector embeddings as a simple representation for any kind of classification task (but you should use an algorithm with unlimited dimension as input)



Use cases:



- **Embedding (layer)** — turns your words into vector representation

- **Deep Network** — turns your sentence (with «no limitation» on it's length) into compressed representation.

- **Fully Connected Layer** — performs classification (regression/binary classification etc.)

- **Output Layer** — transform predictions into answers; **Sigmoid** for binary classification or **Softmax** for both binary and multi classification

Refs:

- Word2vec: https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf
- Vectors: http://vectors.nlpl.eu/repository/
- Russian vectors on Russian national corpora: https://rusvectores.org/ru/
- There is word2vec learning and inference in gensim: https://radimrehurek.com/gensim/models/word2vec.html
- …

Refs

- Perfect mxnet tutorial on words vectors:
  http://www.d2l.ai/chapter_natural-language-processing-pretraining/approx-training.html
- Good article «for Dummies»:
  https://monkeylearn.com/blog/word-embeddings-transform-text-numbers/
- Another good article:
  https://towardsdatascience.com/machine-learning-word-embedding-sentiment-classification-using-keras-b83c28087456
- Some articles if u want MOAR MATH (softmax tricks explained):
  http://ruder.io/word-embeddings-softmax/index.html#hierarchicalsoftmax
- And really you can google BERT, ELMO and GloVe by yourself