From N-Grams to Embeddings

Thiemo Fetzer

University of Warwick & University of Bonn

April 30, 2025

Why Move Beyond N-grams?

Traditional **N-gram language models** represent words as *discrete symbols* and learn from observed *count-based co-occurrences* of word sequences. These models assume a fixed context size and rely on the Markov assumption:

$$P(w_t \mid w_{t-1}, \ldots, w_1) \approx P(w_t \mid w_{t-1}, \ldots, w_{t-n+1})$$

While effective in capturing local dependencies, N-gram models suffer from:

- **Data sparsity**: Most N-grams are rare or never observed.
- **Vocabulary explosion**: Representations are one-hot, leading to high-dimensional and sparse matrices.
- **No notion of similarity**: "apple" and "pear" are completely unrelated unless seen in the exact same contexts.

Plan

Language Models And Classifiers

Learning Embeddings via Gradient Descent

Visualisation of Embeddings

Text as Data

- Text is a rich, underexploited source of information.
- **Key steps in representing text quantitatively:**
  1. **Selecting and Conversion**: Define corpus, make machine-readable.
  2. **Defining Features**: Tokens, n-grams, dictionaries.
  3. **Matrix Construction**: Convert to quantitative matrix.
  4. **Statistical Modeling**: Apply classifiers, clustering, embeddings.

From Text to a Quantitative Matrix

Indian security forces shot another militant in Kupwara district. A group of 100 armed Naxalites attacked Asarali police station, about 250 km from the district headquarters of Gadchiroli, on Maharashtra's border with Andhra Pradesh early on Monday. The Vanni Commander of the Sri Lankan Army (SLA), at a meeting at the Vavuniya Headquarters, asked Tamil paramilitary groups to surrender any firearms in their possession before , tomorrow morning. As per the cease-fire agreement reached between the two, , all Tamil paramilitary groups would have either surrender their arms or join the SLA as regulars to be posted in areas away from the North and the East. According to the prosecution, , , Lahori and his associates killed Ramzan Ali and injured two others in the Saddar area of Karachi.

From Text to a Quantitative Matrix

From Text to a Quantitative Matrix

	docs				
terms	1	2	3	4	5
another	1	0	0	0	0
district	1	1	0	0	0
forces	1	0	0	0	0
in	1	0	1	1	1
indian	1	0	0	0	0
kupwara	1	0	0	0	0
militant	1	0	0	0	0
security	1	0	0	0	0
shot	1	0	0	0	0
	0	1	1	1	1
a	0	1	1	0	0
about	0	1	0	0	0
andhra	0	1	0	0	0
armed	0	1	0	0	0
asarali	0	1	0	0	0
attacked	0	1	0	0	0
border	0	1	0	0	0
early	0	1	0	0	0
from	0	1	0	1	0
gadchiroli	0	1	0	0	0
group	0	1	0	0	0
headquarters	0	1	1	0	0
km	0	1	0	0	0
maharashtra	0	1	0	0	0
monday	0	1	0	0	0
naxalites	0	1	0	0	0
of	0	1	1	0	1
on	0	1	0	0	0
police	0	1	0	0	0
pradesh	0	1	0	0	0

From Text to a Quantitative Matrix

	docs				
terms	1	2	3	4	5
another	1	0	0	0	0
district	1	1	0	0	0
forces	1	0	0	0	0
in	1	0	1	1	1
indian	1	0	0	0	0
kupwara	1	0	0	0	0
militant	1	0	0	0	0
security	1	0	0	0	0
shot	1	0	0	0	0
	0	4	7	3	5
a	0	1	1	0	0
about	0	1	0	0	0
andhra	0	1	0	0	0
armed	0	1	0	0	0
asarali	0	1	0	0	0
attacked	0	1	0	0	0
border	0	1	0	0	0
early	0	1	0	0	0
from	0	1	0	1	0
gadchiroli	0	1	0	0	0
group	0	1	0	0	0
headquarters	0	1	1	0	0
km	0	1	0	0	0
maharashtra	0	1	0	0	0
monday	0	1	0	0	0
naxalites	0	1	0	0	0
of	0	2	1	0	1
on	0	2	0	0	0
police	0	1	0	0	0
pradesh	0	1	0	0	0

Bernoulli Language Model

- a document is represented by a feature vector with binary elements taking value 1 if the corresponding word is present in the document and 0 if the word is not present.
- Let p be the number of words considered, an individual document D_i can be represented as a binary vector $\mathbf{d}_i = (b_{i1}, ..., b_{ip})$.
- Then we can represent a collection of n documents as a **term document incidence matrix**.

$$\mathbf{D} = \begin{pmatrix} b_{11} & ... & b_{1p} \\ b_{21} & ... & b_{2p} \\ ... & & \\ b_{n1} & ... & b_{np} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & ... & 1 \\ 0 & 0 & 1 & ... & 1 \\ 1 & 1 & 0 & ... & 0 \\ & ... & & & \end{pmatrix}$$

Bernoulli Language Model

- Let $P(w_j|Y=k)$ be the probability of word w_j occurring in a document of class k; the probability of w_j not occurring in a document of this class is given by $(1-P(w_j|Y=k))$.
- We can write the document likelihood $P(D_i|k)$ in terms of the individual word likelihoods $P(w_j|Y=k)$:

$$P(D_i|Y=k) = \prod_{j=1}^{p} P((b_{i1},...,b_{ip})|k)$$

$$\overbrace{=}^{\text{NB Assumption}} \prod_{j=1}^{p} b_{ij}P(w_j|k) + (1-b_{ij})(1-P(w_j|k))$$

- If word w_j is present, then $b_{ij}=1$ with probability $P(w_j|Y=k)$
- We can imagine this as a model for generating document feature vectors of class y, where the document feature vector is modelled as a collection of p weighted coin tosses, the j-th having a probability of success equal to $P(w_j|k)$

Training a Bernoulli Naive Bayes Classifier

- The parameters of the likelihoods are the probabilities of each word given the document class $P(w_j|Y = k)$; the model is also parameterised by the prior probabilities, $P(Y = ks)$.
- Using a labeled training set of documents we can estimate these parameters. Let n_k be the number of documents of class $Y = k$ in which w_j is observed
- Let N be the total number of documents, N_k be the total number of documents of class k. Then we can estimate the parameters of the word likelihoods as, for all $j = 1, ..., p$ and all $k \in \mathcal{C}$.

$$\hat{P}(w_j|k) = \frac{n_k}{N_k}$$

$$\hat{P}(k) = \frac{N_k}{N}$$

Training a Bernoulli Naive Bayes Classifier

Algorithm (*Training a Bernoulli Naive Bayes Classifier*)

1. *Define the vocabulary V, where p is the number of words in the vocabulary, i.e. the number of columns.*
2. *Count the following in the training set:*
   - *N the total number of documents*
   - *N_k the number of documents labelled with class k,for $k = 1, ..., |\mathcal{C}|$*
   - *$n_k(w_j)$ the number of documents of class $Y = k$ containing word w_j for every class and for each word in the vocabulary.*
3. *Estimate the likelihoods $P(w_j|Y = k)$ using*

$$\hat{P}(w_j|Y = k) = \frac{n_k(w_j)}{N_k}$$

4. *Estimate the priors $P(Y = k)$ using*

$$\hat{P}(Y = k) = \frac{N_k}{N}$$

Class Assignment using a Bernoulli Naive Bayes Classifier

Once you have trained the Bernoulli Naive Bayes Classifier, you can compute posterior probabilites for a document $D_i = (b_{i1}, ...b_{ip})$ using

$$P(Y = k|D_i) = P(Y = k|(b_{i1}, ..., b_{ip})) \propto \prod_{j=1}^{p} P((b_{i1}, ..., b_{ip})|k)P(k)$$

$$\overbrace{=}^{\text{NB Assumption}} P(k)[\prod_{j=1}^{p} b_{ij}P(w_j|k) + (1 - b_{ij})(1 - P(w_j|k))]$$

for each class $k = 1, ..., |\mathcal{C}|$ and assign the document to class k that yields the largest posterior probability.

Multinomial Distribution

- We now introduce another document model, which explicitly takes into account the number of words, so documents are represented as a collection of word counts.
- The most common distribution to assume is a multinomial distribution, which is a generalization of a Bernoulli distribution.
- Using all the letters, how many distinct sequences can you make from the word "Mississippi"? There are 11 letters to permute, but "i" and "s" occur four times and "p" twice.
- If each letter was distinct, you would have 11 choices for first, 10 for second, 9 for third, ... so a total of 11!.
- However, 4! permutations are identical as the letter "i" is repeated four times; similarly, 4! for "s" and 2! for "p" and 1! for "m".
- So the total number of distinct arrangements of the letters that form the word "Mississippi" is:

$$\frac{11!}{4!4!2!1!} = 34650$$

Reminder: Multinomial Distribution

- Generally if we have n items of p types (letters or words), with n_1 of type 1, n_2 of type 2 and n_p of type p

$$n_1 + ... + n_p = n$$

- then the number of distinct permutations is given by:

$$\frac{n!}{n_1!n_2!...n_p!}$$

- Now suppose a population contains items of $p \geq 2$ different types and that the proportion of items that are of type j is p_j for $(j = 1, ..., p)$, with

$$\sum_{j=1}^{p} p_j = 1$$

- Suppose n items are drawn at random (with replacement) and let x_j denote the number of items of type j.
- The vector $\mathbf{x} = (x_1, ..., x_p)$ has a multinomial distribution with parameters n and $p_1, ..., p_p$.

Multinomial Language Model

- In the multinomial language model, you model documents as being collections of word counts.
- Let p be the number of words considered, an individual document D_i can be represented as a vector $\mathbf{x}_i = (x_{i1}, ..., x_{ip})$.
- $n_i = \sum_{j=1}^{p} x_{ij}$ is the total number of words of document D_i.
- Then we can represent a collection of n documents as a **term document count matrix**.

$$\mathbf{X} = \begin{pmatrix} x_{11} & \dots & x_{1p} \\ x_{21} & \dots & x_{2p} \\ \dots & & \\ x_{n1} & \dots & x_{np} \end{pmatrix} = \begin{pmatrix} 3 & 0 & 0 & \dots & 1 \\ 0 & 0 & 5 & \dots & 1 \\ 7 & 2 & 1 & \dots & 3 \\ & \dots & & & \end{pmatrix}$$

Multinomial Language Model

- We can now write the joint probability of observing a document $D_i = (x_{i1}, ..., x_{ip})$ of class k as:

$$P(D_i|Y = k) = P((x_{i1}, ..., x_{ip})|k) = \frac{n_i!}{\prod_{j=1}^{p} x_{ij}!} \prod_{j=1}^{p} P(w_j|k)^{x_{ij}}$$

- $n_i = \sum_{j=1}^{p} x_{ij}$ is the total number of words of document i.
- So we need to estimate $P(w_j|k)$ from our training data.
- We generally ignore the scaling factor $\frac{n_i!}{\prod_{j=1}^{p} x_{ij}!}$, because it is not a function of the class k! So we can savely ignore it, as it will not affect the optimal class assignment.

Multinomial Language Model

- In the Multinomial language model, we assume that word counts x_{ij} are generated following a multinomial distribution.
- Multinomial distribution is a generalization of the Binomial distribution.
- Let x_i indicate the number of times outcome number i is observed over the n trials, the vector $x_i = (x_{i1}, ..., x_{ip})$ follows a multinomial distribution with parameters n and p, where $p = (p_1, ..., p_p)$

$$P(X_1 = x_{i1} \cap ... \cap X_p = x_{ip}) = \frac{n!}{x_{i1}! \cdots x_{ip}!} p_1^{x_{i1}} \cdots p_p^{x_{ip}}$$

For example:

$$\mathbf{X} = \begin{pmatrix} x_{11} & ... & x_{1p} \\ x_{21} & ... & x_{2p} \\ ... & & \\ x_{n1} & ... & x_{np} \end{pmatrix} = \begin{pmatrix} 3 & 0 & 0 & ... & 1 \\ 0 & 0 & 5 & ... & 1 \\ 7 & 2 & 1 & ... & 3 \\ & ... & & & \end{pmatrix}$$

Multinomial Language Model

- We can now write the joint probability of observing a document $D_i = (x_{i1}, ..., x_{ip})$ of class k as:

$$P(D_i|Y = k) = P((x_{i1}, ..., x_{ip})|k) = \frac{n_i!}{\prod_{j=1}^{p} x_{ij}!} \prod_{j=1}^{p} P(w_j|k)^{x_{ij}}$$

$$\propto \prod_{j=1}^{p} P(w_j|k)^{x_{ij}}$$

- $n_i = \sum_{j=1}^{p} x_{ij}$ is the total number of words of document i.
- So we need to estimate $P(w_j|k)$ from our training data.
- We generally ignore the scaling factor $\frac{n_i!}{\prod_{j=1}^{p} x_{ij}!}$, because it is not a function of the class k! So we can savely ignore it, as it will not affect the optimal class assignment.

Multinomial Model and Estimation of Likelihoods

- For the Multinomial distribution we need to estimates of the vector of $P(w_j|k)$ for all $j = 1, ..., p$ and all $k \in \mathcal{C}$.
- I.e. there is a different vector $\mathbf{p} = (p_1, ..., p_p)$ for every possible class k.
- The maximum likelihood estimator turns out to be slightly more tricky:

$$P(w_j|k) = \frac{\text{No. of times word } w_j \text{ appears in all documents of class } k}{\text{Total No. of words in all documents of class } k}$$

- A more formal notation is

$$\hat{P}(w_j|Y = k) = \frac{\sum_{i=1}^{N} x_{ij} z_{ik}}{\sum_{s=1}^{p} \sum_{i=1}^{N} x_{is} z_{ik}}$$

where z_{ik} is a dummy variable that is 1, in case document i has class k.

Training a Multinomial Naive Bayes Classifier

Algorithm (*Training a Multinomial Naive Bayes Classifier*)

1. *Define the vocabulary V, where p is the number of words in the vocabulary, i.e. the number of columns.*
2. *Count the following in the training set:*
   - *N the total number of documents*
   - *N_k the number of documents labelled with class k,for $k = 1, ..., |\mathcal{C}|$*
   - *x_{ij} the frequency of word w_j in document D_i, computed for every word w_j in V.*
3. *Estimate the likelihoods $P(w_j|Y = k)$ using*

$$\hat{P}(w_j|Y = k) = \frac{\sum_{i=1}^{N} x_{ij} z_{ik}}{\sum_{s=1}^{p} \sum_{i=1}^{N} x_{is} z_{ik}}$$

4. *Estimate the priors $P(Y = k)$ using*

$$\hat{P}(Y = k) = \frac{N_k}{N}$$

Class Assignment using a Multinomial Naive Bayes Classifier

We compute the posterior probability of a document D_i, represented as a word count vector $\mathbf{x_i}$, belonging to some class k as:

$$P(Y = k|D_i) = P(Y = k|\mathbf{x_i}) \overbrace{\propto}^{\text{Bayes Law}} P((x_{i1}, ..., x_{ip})|k)P(k)$$

$$\overbrace{\propto}^{\text{NB Assumption}} P(k) \prod_{j=1}^{p} P(w_j|k)^{x_{ij}}$$

Note that, unlike the Bernoulli model, words that do not occur in a document (i.e., for which $x_{ij} = 0$) do not affect the probability (since $p^0 = 1$).
Thus we can write the posterior probability in terms of the set of words U that appear in document i, i.e. the set of words U defined by $x_{ij} > 0$.

Limitations of Traditional Representations

- **Sparse**: Most entries are zero.
- **High-Dimensional**: Number of features = vocabulary size.
- **No Semantic Relationship**: *"apple"* and *"pear"* are orthogonal.

→ We need compact, dense representations that capture meaning!

Topic models? Last week discussed topic modeling where every word has a probability distribution (likelihood) across a topic space K

- Each word can be represented as its probabilities across many soft topics.
- Given a large number of topics (e.g., 1000), the word embedding becomes:

$$\text{Embedding of word } w = (P(z_1|w), P(z_2|w), \ldots, P(z_{1000}|w))$$

→ We turned a word into a vector. This is what embeddings do.

Plan

Language Models And Classifiers

Learning Embeddings via Gradient Descent

Visualisation of Embeddings

Word2Vec

Building on what we know:

- We already represent documents with matrices (Bernoulli, Multinomial models).
- We already reduce words to feature vectors.
- We can represent words as vectors over a latent topic space through topicmodelling

Word2Vec introduces a key change:

- **Instead of counting co-occurrences, we predict them.**
- Words are placed close together if they appear in similar contexts.
- Embeddings for text generation can induce us to not have to worry about grammar, on average...

The Intuition Behind Word2Vec

Distributional Hypothesis: "You shall know a word by the company it keeps."

Word2Vec operationalizes this:

- Words that appear in similar contexts should have similar vectors.
- Training task: Given a word, predict its context (Skip-gram), or given context, predict the word (CBOW).

Result:

- A dense vector space where semantic similarity is geometric proximity.

Deriving the Skip-Gram Objective

General idea: We model a sequence of words $w_1, w_2, \ldots, w_T$.

- Suppose for a given center word w_t, our model generates *all* context words:

$$P(w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c} \mid w_t)$$

- We make a **conditional independence assumption**:

$$P(\text{context} \mid w_t) = \prod_{\substack{-c \leq j \leq c \\ j \neq 0}} P(w_{t+j} \mid w_t)$$

- Then, the full corpus likelihood becomes:

$$P(w_1, \ldots, w_T) \approx \prod_{t=1}^{T} \prod_{\substack{-c \leq j \leq c \\ j \neq 0}} P(w_{t+j} \mid w_t)$$

- Each word contributes $2 \times c$ training pairs.

A simple example with $c = 1$

Illustrating for ease of less notification clutter with $c = 1$:

- Suppose for *any given center word* w_t, our model generates *all* context words:

$$P(w_{t-1}, w_{t+1} \mid w_t)$$

  To train this we would need to know $P(w_{t-1}, w_t, w_{t+1})$ for all arrangements of words.

- So we only worry about a model of the *joint* distribution *within context* window $c = 1$.
- We make a **conditional independence assumption**:

$$P(w_{t-1}, w_{t+1} \mid w_t) = P(w_{t-1} \mid w_t)P(w_{t+1} \mid w_t)$$

- Then, the full corpus likelihood becomes:

$$P(w_1, \ldots, w_T) \approx \prod_{t=1}^{T} P(w_{t-1} \mid w_t)P(w_{t+1} \mid w_t)$$

From General Likelihood to Softmax Modeling

Take logs to convert products into sums:
The general Skip-Gram objective:

$$\log \mathcal{L} = \sum_{t=1}^{T} \sum_{\substack{-c \le j \le c \\ j \neq 0}} \log P(w_{t+j} \mid w_t)$$

Key question: How do we model $P(w_O \mid w_I)$?

Idea:

- Represent each word with a vector $v_w \in \mathbb{R}^d$ (input) and $v'_w \in \mathbb{R}^d$ (output).
- Make $P(w_O \mid w_I)$ large when v_{w_I} and v'_{w_O} are similar.
- Use the dot product to measure similarity.

From General Likelihood to Softmax Modeling

Model: Softmax over all vocabulary words:

$$P(w_O \mid w_I) = \frac{\exp({v'_{w_O}}^\top v_{w_I})}{\sum_{w=1}^{V} \exp({v'_w}^\top v_{w_I})}$$

where vector $v_w \in \mathbb{R}^d$ (input) and $v'_w \in \mathbb{R}^d$ (output). We get a $P(w_O \mid w_I)$ for every word in the vocabulary V.

We **interpret** the dot-product as a measure of *similarity*:

$${v'_w}^\top v_{w_I}$$

Why?

Link to Cosine Similarity

The dot product between two vectors a and b is:

$$a^\top b = \|a\|\|b\| \cos(\theta)$$

where θ is the angle between a and b.
Thus, the dot product is a **linear transformation** of the cosine similarity, scaled by the vector norms, i.e..

$$v'_w{}^\top v_{w_I} \propto \cos(\theta)$$

And if we normalize embeddings such that $\|v'_{w_O}\| = \|v_{w_I}\| = 1$, then:

$$v'_{w_O}{}^\top v_{w_I} = \cos(\theta)$$

In this case, softmax scoring is directly proportional to the cosine similarity.

Modeling the Conditional Probability

Let:

- w_I = input (center) word
- w_O = output (context) word
- $v_{w_I} \in \mathbb{R}^{d \times 1}$ = "input" vector of w_I
- ${v'_{w_O}}^\top \in \mathbb{R}^{1 \times d}$ = "output" vector of w_O
- d = dimensionality of the embedding space (e.g., 100, 300)

The probability of observing w_O given w_I is modeled using a softmax over the vocabulary:

$$P(w_O \mid w_I) = \frac{\exp({v'_{w_O}}^\top v_{w_I})}{\sum_{w=1}^{V} \exp({v'_w}^\top v_{w_I})}$$

This assigns high probability when v'_{w_O} and v_{w_I} are similar (i.e., their dot product is high).

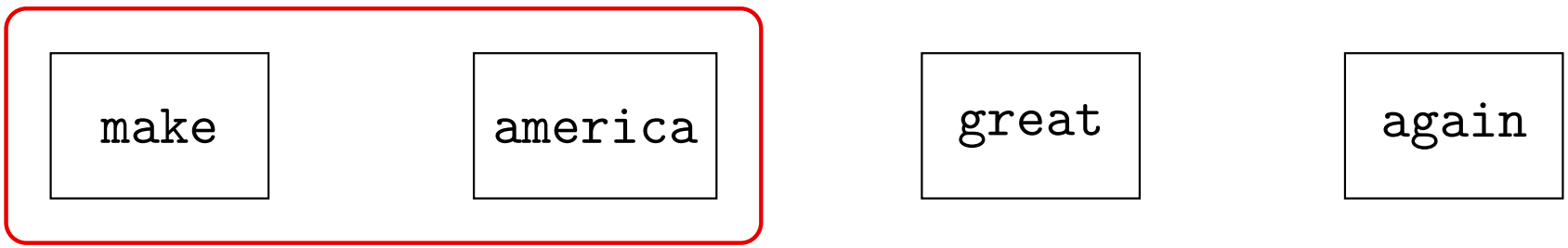
How Skip-Gram Generates Training Pairs

Sentence: `make america great again` **Window size = 1** (context of 1 word on either side)

Generated (center, context) pairs:

How Skip-Gram Generates Training Pairs

Sentence: `make america great again` **Window size = 1** (context of 1 word on either side)

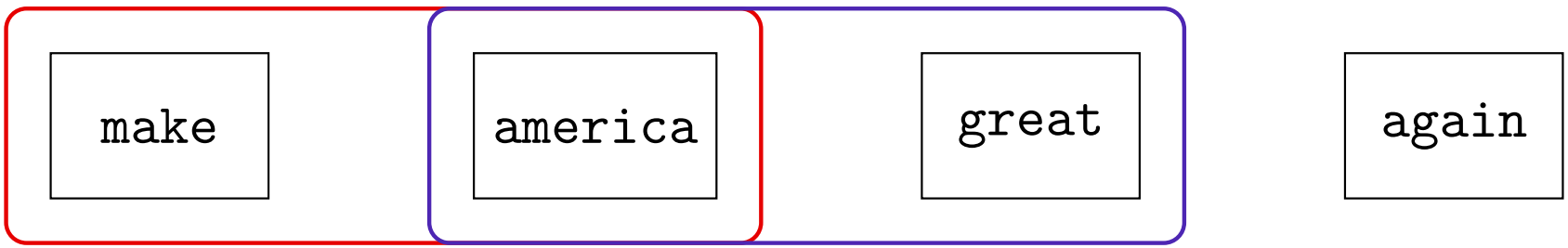


Generated (center, context) pairs:

- (make, america)

How Skip-Gram Generates Training Pairs

Sentence: `make america great again` **Window size = 1** (context of 1 word on either side)

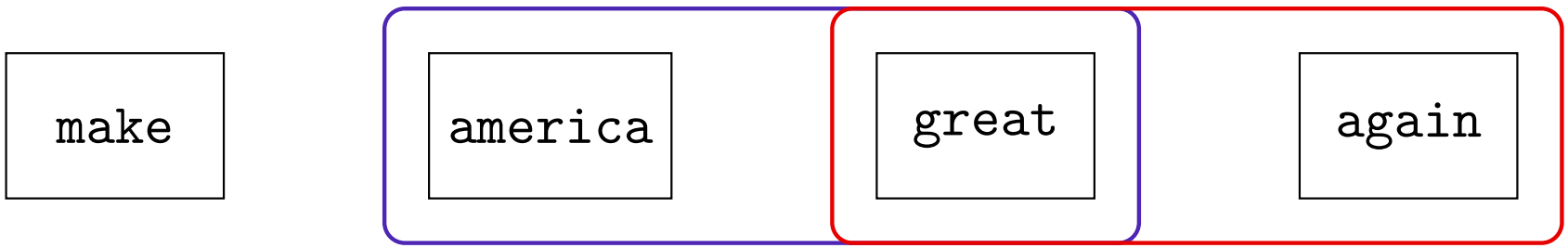


Generated (center, context) pairs:

- (make, america)
- (america, make), (america, great)

How Skip-Gram Generates Training Pairs

Sentence: `make america great again` **Window size = 1** (context of 1 word on either side)

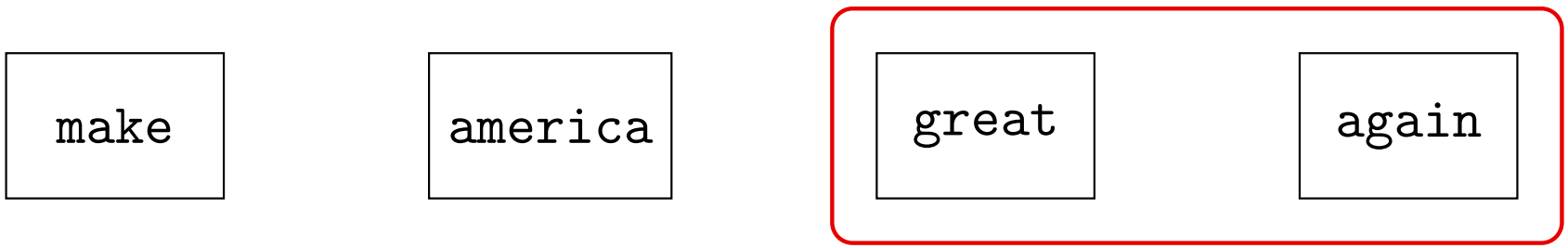


Generated (center, context) pairs:

- (make, america)
- (america, make), (america, great)
- (great, america), (great, again)

How Skip-Gram Generates Training Pairs

Sentence: `make america great again` **Window size = 1** (context of 1 word on either side)



Generated (center, context) pairs:

- (make, america)
- (america, make), (america, great)
- (great, america), (great, again)
- (again, great)

Gradient Descent in Skip-Gram

At each training step:

- Take each center word w_I and each context word w_O.
- Compute scores: $\text{score}_w = {v'_w}^\top v_{w_I}$ for all words w.
- Apply softmax to get probabilities.
- Compute loss: $-\log P(w_O|w_I)$.
- Compute gradient with respect to parameters.
- Update parameters using **stochastic gradient descent**.

Stochastic Gradient Descent to Train Embeddings

We now use the (center, context) pairs to train the embeddings.

- Each pair (`center = great, context = america`) contributes one term to the overall loss:

$$\log P(\texttt{america} \mid \texttt{great}) = \log \frac{\exp({v'_{\texttt{america}}}^{\top} v_{\texttt{great}})}{\sum_{w \in V} \exp({v'_{w}}^{\top} v_{\texttt{great}})}$$

- where V is indicating the total vocabulary and d is the embedding dimensionality.
- The loss for this pair is:

$$\mathcal{L}_{(w_I, w_O)} = -\log P(w_O \mid w_I)$$

- We next derive the first derivative:

$$\frac{\partial \mathcal{L}_{(w_I, w_O)}}{\partial v_{w_I}} = \sum_{w \in V} P(w \mid w_I) v'_w - v'_{w_O} \quad \in \mathbb{R}^d$$

Stochastic Gradient Descent to Train Embeddings

Derivation of the gradient with respect to v_{w_I}:

$$\begin{aligned}\mathcal{L}_{(w_I,w_O)} &= -\log \frac{\exp({v'_{w_O}}^\top v_{w_I})}{\sum_{w\in V}\exp({v'_w}^\top v_{w_I})} \\ &= -{v'_{w_O}}^\top v_{w_I} + \log\left(\sum_{w\in V}\exp({v'_w}^\top v_{w_I})\right)\end{aligned}$$

Differentiating:

$$\frac{\partial \mathcal{L}}{\partial v_{w_I}} = -v'_{w_O} + \sum_{w\in V} P(w \mid w_I) v'_w \in \mathbb{R}^d$$

This gradient nudges v_{w_I} toward v'_{w_O} and away from all other v'_w.
SGD update:

$$v_{w_I} \leftarrow v_{w_I} - \eta \frac{\partial \mathcal{L}_{(w_I,w_O)}}{\partial v_{w_I}} \quad (\text{in } \mathbb{R}^d)$$

Example: Building a Vocabulary

```
# Define example sentences
sentences <- list(
  c("make", "america", "great", "again"),
  c("make", "america", "great"),
  c("america", "is", "great"),
  c("great", "banana", "split"),
  c("eat", "banana", "daily"),
  c("great", "job", "again")
)

# Build vocabulary
words <- unique(unlist(sentences))
V <- length(words)
word2idx <- setNames(1:V, words)
idx2word <- setNames(words, 1:V)
```

Vocabulary and Training Pairs

Unique Words: $|V| = 10$

- `make, america, great, again, is, banana, split, eat, daily, job`

Training Pairs (Window = 2):

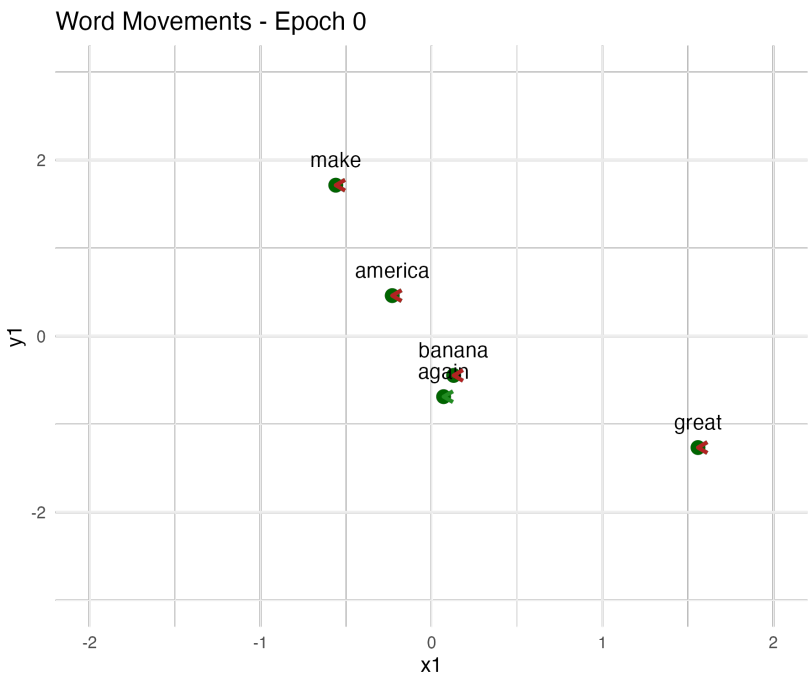
- (make, america), (make, great)
- (america, make), (america, great), (america, again), (america, is)
- (great, make), (great, america), (great, again), (great, is), (great, banana), (great, job)
- (again, america), (again, great), (again, job)
- (is, america), (is, great)
- (banana, great), (banana, split), (banana, eat), (banana, daily)
- (split, banana)
- (eat, banana), (eat, daily)
- (daily, banana)
- (job, great), (job, again)

Why Do We Need Gradient Descent?

- Vocabulary size $|V|$ often large
- The softmax function is nonlinear **non-convex optimization problem** with no closed solution
- Gradient descent provides an iterative method to find good embeddings
- Initialise with assigning random numbers to each vector $v_w \in \mathbb{R}$
- Predicting context words requires assigning a probability to every word in the vocabulary.
- Computing exact probabilities for each pair would be computationally intensive
- We need an efficient method to **adjust** the embeddings step-by-step:
  - Reduce the loss $-\log P(w_O|w_I)$.
  - Improve predictions over time.

Gradient descent provides a scalable way to *iteratively* improve embeddings.

Gradient Descent in Action

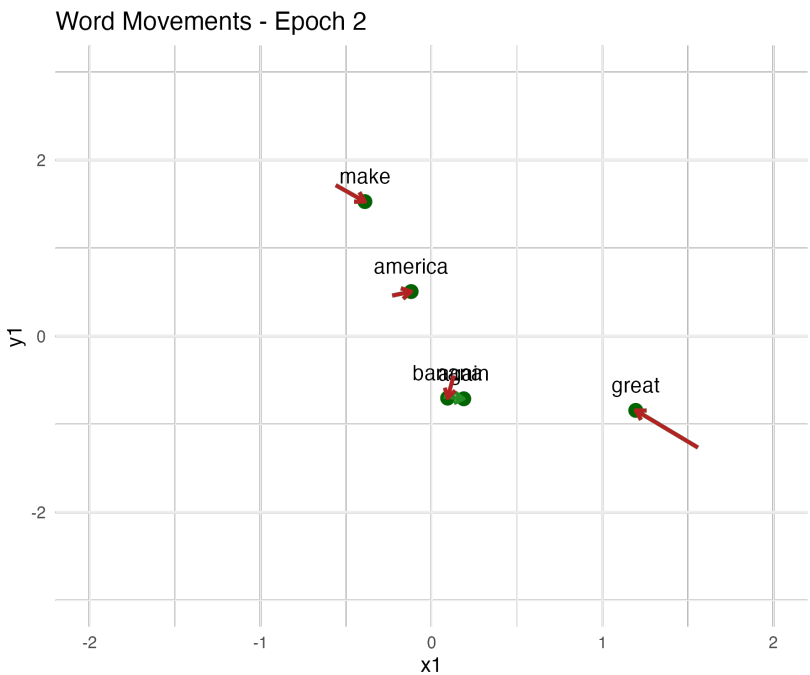


- Initially, embeddings are random and unstructured.
- Positive training pairs (e.g., `great`, `america`) are pulled closer.
- Negative pairs (e.g., `banana`, `great`) are pushed apart.
- Each update step mimics a gradient:

$$v_w \leftarrow v_w \pm \eta(v_{w'} - v_w)$$

- Arrows visualize how vectors shift in response to training pairs.

Gradient Descent in Action

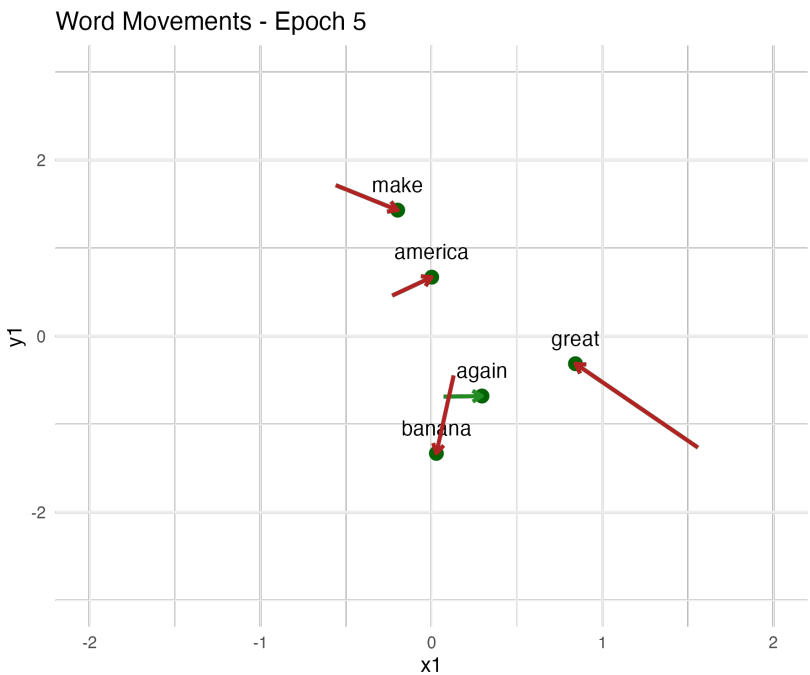


- Initially, embeddings are random and unstructured.
- Positive training pairs (e.g., `great`, `america`) are pulled closer.
- Negative pairs (e.g., `banana`, `great`) are pushed apart.
- Each update step mimics a gradient:

$$v_w \leftarrow v_w \pm \eta(v_{w'} - v_w)$$

- Arrows visualize how vectors shift in response to training pairs.

Gradient Descent in Action

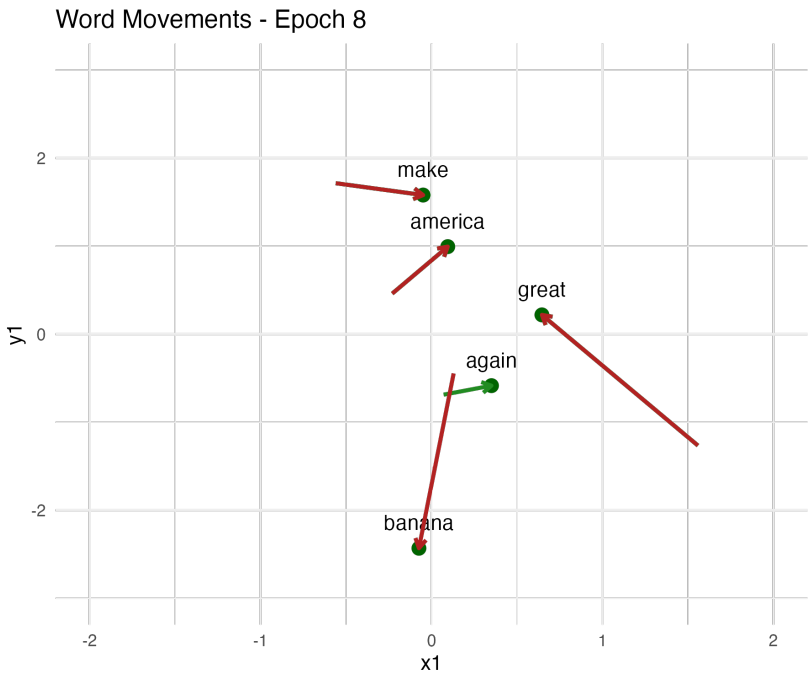


- Initially, embeddings are random and unstructured.
- Positive training pairs (e.g., `great`, `america`) are pulled closer.
- Negative pairs (e.g., `banana`, `great`) are pushed apart.
- Each update step mimics a gradient:

$$v_w \leftarrow v_w \pm \eta(v_{w'} - v_w)$$

- Arrows visualize how vectors shift in response to training pairs.

Gradient Descent in Action



- Initially, embeddings are random and unstructured.
- Positive training pairs (e.g., `great`, `america`) are pulled closer.
- Negative pairs (e.g., `banana`, `great`) are pushed apart.
- Each update step mimics a gradient:

$$v_w \leftarrow v_w \pm \eta(v_{w'} - v_w)$$

- Arrows visualize how vectors shift in response to training pairs.

Is Word2Vec like K-Means?

- **Both learn spatial structures in vector space.**
- K-Means clusters data points around fixed centroids.
- Word2Vec gradually adjusts word vectors via gradient updates.

Key similarities:

- Words that co-occur are pulled together (like points in the same cluster).
- Embeddings form semantic "regions" in space.

Key differences:

- K-Means has *hard* cluster assignments; Word2Vec is *continuous*.
- Word2Vec is trained via prediction loss, not geometric distance.
- No explicit centroids — similarity emerges through context prediction.

Plan

Language Models And Classifiers

Learning Embeddings via Gradient Descent

Visualisation of Embeddings

Two techniques for visualisation

Embeddings are trained in a **high dimensional space**, typically with $d = \in \{256, 512, 1024\}$.

It often is useful to **inspect embeddings** in a lower dimensional space to assess quality.

Two common techniques:

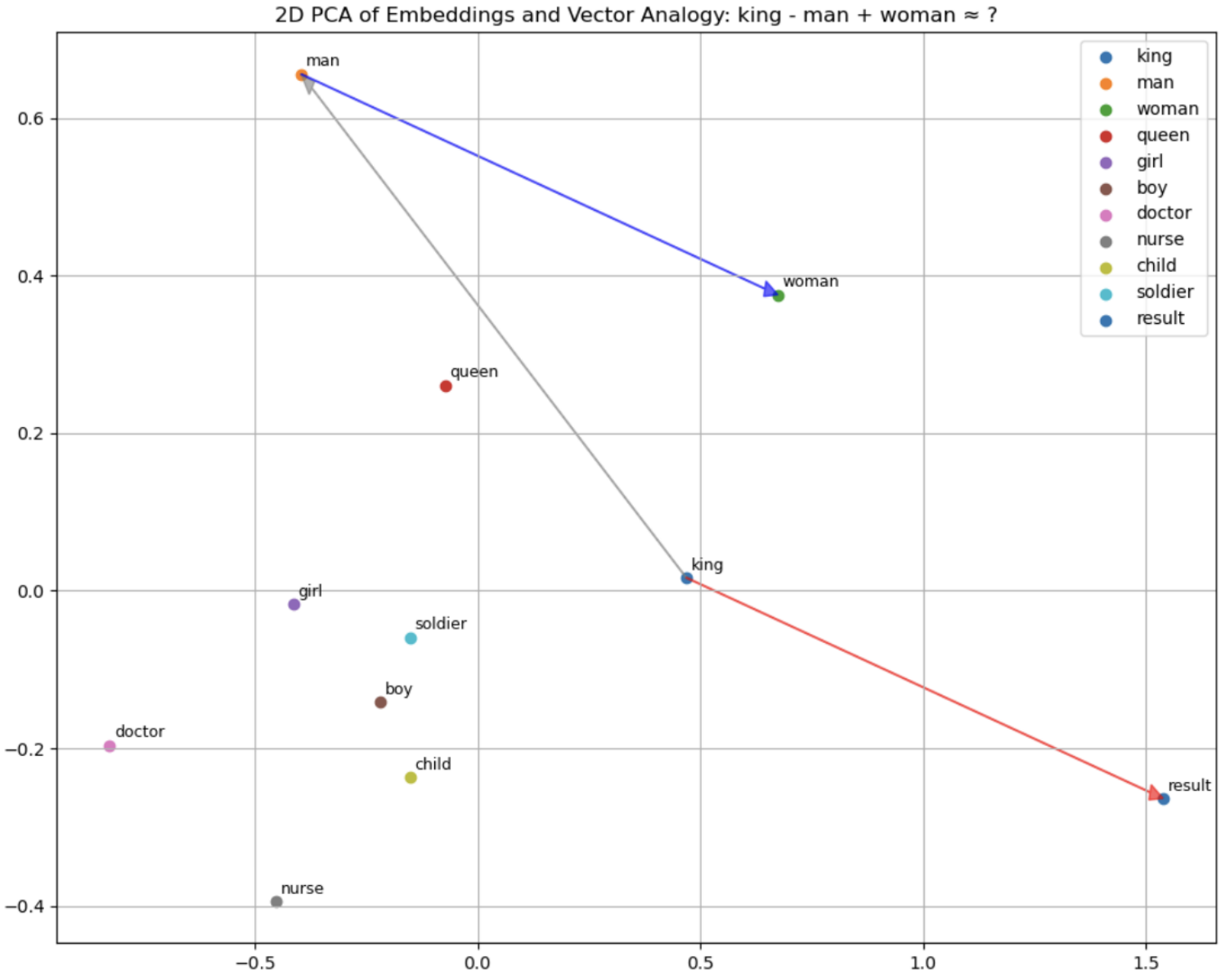
- Principal Component Analysis
- t-SNE

$\rightarrow$ here we introduce both of them briefly

Principal Component Analysis (PCA)

- PCA looks for the directions where the data varies the most.
- These directions (called "principal components") let us project high-dimensional vectors into 2D.
- PCA is linear and fast — it works well when you're interested in structure and analogies.

PCA for Word Embeddings



t-SNE: Neighborhood Preservation

- t-SNE focuses on keeping nearby points close together in the 2D map.
- It helps you see clusters and local groupings.
- Useful when you're interested in similarity and semantic neighborhoods.

t-SNE visualisation