Quantitative Text Analysis
Meeting 10: Language Models

Petro Tolochko

Outline

What is a language model? (Intuition)

- A **language model (LM)** assigns a probability to text.
- Given a sequence of words $w_1, w_2, \ldots, w_T$, an LM answers:

$$p(w_1, w_2, \ldots, w_T)$$

- Answers:
  - How *likely* is this sentence?
  - What is a *plausible next word*?
  - Which of two texts *fits better* with the language?
- Conceptually: an LM is a **probability distribution over strings**.

Language models as probability distributions

Goal: model the probability of a sequence of tokens

$$p(w_1, w_2, \ldots, w_T)$$

Language models as probability distributions

Goal: model the probability of a sequence of tokens

$$p(w_1, w_2, \ldots, w_T)$$

Using the chain rule of probability:

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

Language models as probability distributions

Goal: model the probability of a sequence of tokens

$$p(w_1, w_2, \ldots, w_T)$$

Using the chain rule of probability:

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

- At each position t, the LM predicts the probability of the next token given all previous tokens:

$$p(w_t \mid w_{<t})$$

- **Autoregressive view:** generate text one token at a time, always conditioning on the past.

An example

Consider the prefix: *"I drank a cup of"*

A language model might assign:

$$p(\text{coffee} \mid \text{"I drank a cup of"}) \approx 0.40$$
$$p(\text{tea} \mid \text{"I drank a cup of"}) \approx 0.30$$
$$p(\text{water} \mid \text{"I drank a cup of"}) \approx 0.10$$
$$p(\text{elephant} \mid \text{"I drank a cup of"}) \approx 0.000001$$

An example

Consider the prefix: *"I drank a cup of"*

A language model might assign:

$$p(\text{coffee} \mid \text{"I drank a cup of"}) \approx 0.40$$
$$p(\text{tea} \mid \text{"I drank a cup of"}) \approx 0.30$$
$$p(\text{water} \mid \text{"I drank a cup of"}) \approx 0.10$$
$$p(\text{elephant} \mid \text{"I drank a cup of"}) \approx 0.000001$$

- The LM encodes which continuations are **plausible** in the language.
- Good LMs capture:
  - Grammar and syntax
  - Word meanings and typical combinations
  - Some world knowledge (cups of coffee are common, cups of elephants are not)

What can we *do* with a language model?

Once we have $p(w_t \mid w_{<t})$, we can:

- **Generate text**
  - Sample the next token from the distribution repeatedly.
- **Score text**
  - Compute how likely a sentence or document is.
  - Compare alternative versions of a text.
- **Use as a component** in downstream NLP tasks:
  - Machine translation, summarization, question answering, etc.
  - Feature extraction (embeddings) for text-as-data applications.

What can we *do* with a language model?

Once we have $p(w_t \mid w_{<t})$, we can:

- **Generate text**
  - Sample the next token from the distribution repeatedly.
- **Score text**
  - Compute how likely a sentence or document is.
  - Compare alternative versions of a text.
- **Use as a component** in downstream NLP tasks:
  - Machine translation, summarization, question answering, etc.
  - Feature extraction (embeddings) for text-as-data applications.

LM is a **general-purpose engine** that turns text into probabilities and useful representations we can exploit.

Different kinds of language models

Many architectures can implement this same idea:

- **Count-based models**
  - n-gram models (Markov chains over words)
  - Simple, interpretable, but limited context.
- **Neural models**
  - Feed-forward neural LMs, RNNs, LSTMs, GRUs
  - Can use longer context, learn continuous representations.
- **Transformer-based models**
  - Self-attention, parallel computation
  - Scales to very large models (GPT, BERT, etc.)

Different kinds of language models

Many architectures can implement this same idea:

- **Count-based models**
  - n-gram models (Markov chains over words)
  - Simple, interpretable, but limited context.
- **Neural models**
  - Feed-forward neural LMs, RNNs, LSTMs, GRUs
  - Can use longer context, learn continuous representations.
- **Transformer-based models**
  - Self-attention, parallel computation
  - Scales to very large models (GPT, BERT, etc.)

While the complexity of **Makrov** LMs is drastically lower than e.g., **transformer** LMs, the core probabilistic idea stays the same.

Unigram (Multinomial) Model: Intuition

Simplest LM: assume tokens are independent:

$$p(w_{1:T}) = \prod_{t=1}^{T} p(w_t)$$

- Equivalent to a **bag-of-words**: order is ignored.
- Learning = estimating word frequencies.
- Useful as a baseline, but extremely limited.

$$\hat{p}(w) = \frac{\text{count}(w)}{N}$$

This model motivates why we need word *dependencies*.

Why the Unigram Model Fails

- Ignores word order:
$$P(\text{cat sat}) = P(\text{sat cat})$$
- Cannot capture syntax, semantics, phrases, or dependencies.
- Predicts text that is locally plausible but globally nonsensical.

This motivates n-gram models (Markov assumption).

From Unigrams to n-grams

Language is sequential:

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

n-gram idea: use only the last $n-1$ tokens:

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

- $n = 1$: unigram
- $n = 2$: bigram
- $n = 3$: trigram

This is a **Markov assumption**.

Markov Assumption

$$P(w_{1:T}) \approx \prod_{t=1}^{T} P(w_t \mid w_{t-n+1:t-1})$$

Example for bigrams:

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

This captures **local** word dependencies, improving realism and syntax.

Bigram Example with Counts

Mini-corpus: "the cat sat on the mat"

$$c(\text{the} \rightarrow \text{cat}) = 1, \quad c(\text{the}) = 2$$

Bigram probability:

$$P(\text{cat} \mid \text{the}) = \frac{1}{2}$$

General formula:

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

Avoiding Zero Probabilities: Smoothing

Raw n-gram estimates give many zeros. For unseen bigrams:

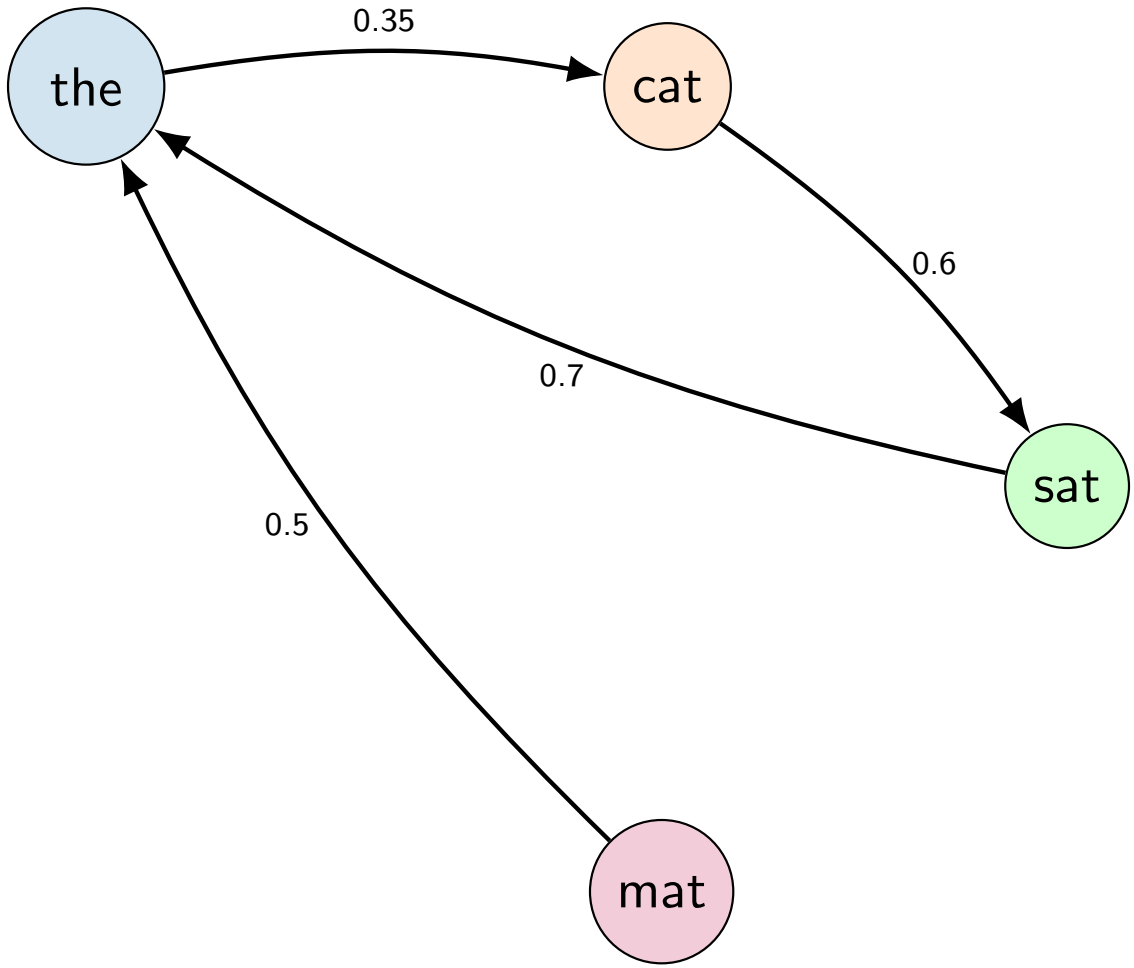
$$c(w_{t-1}, w_t) = 0 \Rightarrow P = 0$$

Add-α (Laplace) smoothing:

$$P_\alpha(w_t \mid w_{t-1}) = \frac{c(w_{t-1}, w_t) + \alpha}{c(w_{t-1}) + \alpha|V|}$$

- Ensures non-zero probabilities
- Helps generalization

Bigram Model as a Markov Chain



Each arrow encodes a transition probability $P(w_t \mid w_{t-1})$.

Markov Transition Matrix

$$M_{uv} = P(v \mid u)$$

	the	cat	sat	mat
the	0	0.35	0.10	0.55
cat	0.20	0	0.60	0.20
sat	0.70	0.10	0	0.20
mat	0.50	0.10	0.10	0

- Each row is a probability distribution.
- Use it to generate sentences or score sequences.

Evaluating Language Models: Entropy

Entropy measures the **uncertainty** or **average surprise** of a probability distribution.

$$H(p) = -\sum_{w \in V} p(w) \log p(w)$$

- Shannon (1948): quantifies how many bits are needed on average to encode outcomes from p.
- High entropy $\rightarrow$ more uncertainty (uniform distribution).
- Low entropy $\rightarrow$ more predictability (peaked distribution).

Example:

$$p = (0.5, 0.5): \quad H = 1 \text{ bit}$$

$$p = (0.99, 0.01): \quad H \approx 0.08 \text{ bits}$$

Entropy defines a limit on how well any language model can compress text drawn from p.

Cross Entropy

Cross entropy measures how well a model q can encode data generated from the true distribution p:

$$H(p, q) = -\sum_{w \in V} p(w) \log q(w)$$

- If $q = p$, then $H(p, q) = H(p)$ (optimal case).
- If q assigns low probability to events common in p, cross entropy increases.
- Used to train language models: model tries to **minimize cross-entropy**.

Key relationship:

$$H(p, q) = H(p) + KL(p||q)$$

Thus cross entropy is minimized exactly when $q = p$. In neural LMs, cross-entropy $=$ the standard training loss.

Why Cross Entropy in Language Modeling?

Given a corpus with empirical distribution $\hat{p}$, minimizing cross-entropy:

$$H(\hat{p}, q) = -\sum_{w} \hat{p}(w) \log q(w)$$

is equivalent to **maximizing the likelihood** of the model q.

$$\arg\min_{q} H(\hat{p}, q) \quad \Longleftrightarrow \quad \arg\max_{q} \prod_{t=1}^{T} q(w_t)$$

- Cross-entropy is just negative log-likelihood, averaged.
- This makes it the natural objective for both n-gram and neural models.

Perplexity

Perplexity measures how "surprised" a model is:

$$\text{PP} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(w_t \mid w_{<t})\right)$$

- Lower perplexity = better LM.
- Standard evaluation for n-gram models.
- Still used today, although less meaningfully for LLMs.

Why Go Beyond n-grams?

n-gram models suffer from:

- **Fixed context window** (local dependencies only).
- **Data sparsity:** number of parameters grows as $O(|V|^n)$.
- **No notion of similarity**: "cat" and "cats" unrelated.

Why Go Beyond n-grams?

n-gram models suffer from:

- **Fixed context window** (local dependencies only).
- **Data sparsity:** number of parameters grows as $O(|V|^n)$.
- **No notion of similarity**: "cat" and "cats" unrelated.

Motivation:

- Learn **continuous representations** that generalize.
- Capture semantic and syntactic structure.
- Feed numeric vectors into neural networks.

What Are Embeddings?

- Language consists of **discrete symbols** (words/tokens).
- Neural networks require **numeric vectors** as inputs.
- An **embedding** maps each word to a point in a continuous space:

$$w_i \rightarrow \mathbf{v}_i \in \mathbb{R}^d$$

- Words with similar meanings or contexts learn similar vectors.

Embeddings connect discrete language with continuous computation.

From One-Hot to Dense Vectors

One-hot representation:

$$w_i \to \mathbf{e}_i = [0, \ldots, 1, \ldots, 0] \in \mathbb{R}^{|V|}$$

- Sparse & high-dimensional
- No similarity between words

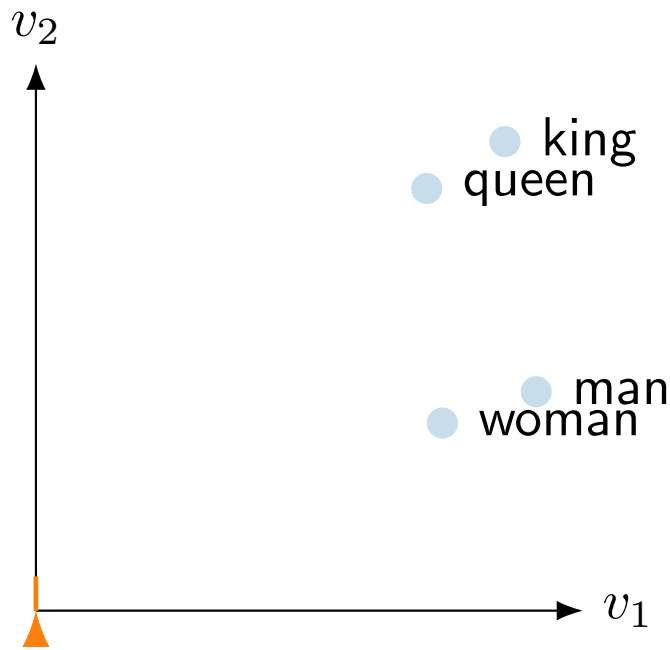
Embedding representation:

$$\mathbf{v}_i = E[i] \in \mathbb{R}^d$$

where $E \in \mathbb{R}^{|V| \times d}$ is the **embedding matrix**.

Key fact: Embedding lookup = matrix row selection. No computation, just indexing.

Embedding Space: Geometric Intuition



Differences between vectors encode interpretable relations (e.g., gender, tense, semantic fields).

How Do We Compare Word Vectors?

Cosine similarity:

$$\text{sim}(w_i, w_j) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\| \, \|\mathbf{v}_j\|}$$

- Close to 1 $\rightarrow$ highly similar
- Around 0 $\rightarrow$ unrelated
- Negative $\rightarrow$ opposite contexts

Examples:

$$\text{sim}(\text{happy}, \text{joyful}) \approx 0.9, \qquad \text{sim}(\text{happy}, \text{sad}) \approx -0.4$$

Why Do Embeddings Capture Meaning?

Embeddings are learned by **predictive objectives**.
Example: Skip-gram (Word2Vec):

$$\max_{E} \sum_{(w,c)\in D} \log \sigma(\mathbf{v}_w^\top \mathbf{v}_c)$$

Words that co-occur frequently get vectors with large dot products.

In neural LMs (RNNs, Transformers):

$$\max_{E} \sum_{t} \log P(w_t \mid w_{<t})$$

Embedding matrix is updated through backpropagation.

Conclusion: Co-occurrence $\rightarrow$ geometry $\rightarrow$ meaning.

Types of Embeddings

1. **Static embeddings** (Word2Vec, GloVe)
   - One vector per word.
   - Does not depend on context.
2. **Subword embeddings** (BPE, SentencePiece)
   - Handle rare words, morphology.
3. **Contextual embeddings** (ELMo, BERT, GPT)
   - Representation *changes* with context.
   - "bank" in "river bank" vs. "central bank" $\rightarrow$ different vectors.

These are the embeddings used in modern neural language models.

Why Neural Networks?

Embeddings give us continuous vectors, but we need a function that:

- Processes these vectors,
- Learns non-linear transformations,
- Combines information from different inputs,
- Predicts probabilities for language modeling.

Why Neural Networks?

Embeddings give us continuous vectors, but we need a function that:

- Processes these vectors,
- Learns non-linear transformations,
- Combines information from different inputs,
- Predicts probabilities for language modeling.

Neural networks provide:

$$f_\theta(\mathbf{x}) = \text{nonlinear}(W\mathbf{x} + b)$$

They are universal function approximators.

The Perceptron: The Basic Computational Unit

A single artificial neuron computes:

$$y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

Components:

- Input vector $\mathbf{x} \in \mathbb{R}^d$
- Weights $\mathbf{w}$
- Bias b
- Activation function $\sigma(\cdot)$

Interpretation: A perceptron takes a weighted sum of inputs and passes it through a nonlinearity.

Activation Functions

Neural networks rely on nonlinear activations.

Sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Tanh:

$$\tanh(z)$$

ReLU:

$$\text{ReLU}(z) = \max(0, z)$$

Softmax (output layer):

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Nonlinearities allow networks to learn complex mappings that linear models cannot capture.

Stacking Perceptrons into a Layer

A neural network layer applies multiple perceptrons in parallel:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$$

Where:

- $W \in \mathbb{R}^{m \times d}$ maps inputs to m hidden units.
- $\mathbf{b} \in \mathbb{R}^m$ is a bias vector.
- $\mathbf{h}$ is the hidden representation.

Stacking Perceptrons into a Layer

A neural network layer applies multiple perceptrons in parallel:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$$

Where:

- $W \in \mathbb{R}^{m \times d}$ maps inputs to m hidden units.
- $\mathbf{b} \in \mathbb{R}^m$ is a bias vector.
- $\mathbf{h}$ is the hidden representation.

Idea: Each neuron extracts a different feature of the input.

Stacking Layers: Multi-Layer Perceptrons

A basic neural network stacks layers:

$$h_1 = \sigma(W_1 x + b_1)$$

$$h_2 = \sigma(W_2 h_1 + b_2)$$

$$\hat{y} = W_3 h_2 + b_3$$

Stacking Layers: Multi-Layer Perceptrons

A basic neural network stacks layers:

$$h_1 = \sigma(W_1 x + b_1)$$
$$h_2 = \sigma(W_2 h_1 + b_2)$$
$$\hat{y} = W_3 h_2 + b_3$$

Why stack layers?

- Lower layers learn simple features.
- Higher layers learn abstract patterns.

This hierarchy of representations is essential for neural LMs.

Training Neural Networks

Neural networks learn by minimizing a loss function.
Goal:

$$\min_{\theta} \mathcal{L}(f_\theta(x), y)$$

Backpropagation:

- Compute gradients of loss w.r.t. each parameter.
- Propagate errors backward through layers.
- Update weights via gradient descent.

This is how embeddings, hidden layers, and output layers are learned simultaneously.

Neural Networks as Feature Learners

Neural networks automatically learn hierarchical representations:

$$\mathbf{x} \to h_1 \to h_2 \to \cdots \to \hat{y}$$

This is ideal for language modeling:

- Embed each word into $\mathbb{R}^d$
- Combine embeddings using learned nonlinear functions
- Predict next-word probabilities

Next: Feedforward neural language models (Bengio et al., 2003).

Why Neural Language Models?

Embeddings give us continuous word vectors. Now we want a model that:

- **Uses** these vectors as input,
- **Learns** non-linear functions of context,
- **Generalizes** beyond exact n-gram counts,
- **Predicts** the next word probabilistically.

Why Neural Language Models?

Embeddings give us continuous word vectors. Now we want a model that:

- **Uses** these vectors as input,
- **Learns** non-linear functions of context,
- **Generalizes** beyond exact n-gram counts,
- **Predicts** the next word probabilistically.

Idea (Bengio et al., 2003):

$$P(w_t \mid w_{t-(n-1):t-1}) = \text{softmax}(W_o\, h + b_o)$$

where h is a hidden layer applied to embeddings of previous words.

Architecture of a Feedforward Neural LM

Given context words $w_{t-1}, \ldots, w_{t-(n-1)}$:

1. Look up embeddings:

$$\mathbf{x} = [E(w_{t-1}); E(w_{t-2}); \ldots; E(w_{t-(n-1)})]$$

2. Apply a non-linear hidden layer:

$$h = \tanh(W_h \mathbf{x} + b_h)$$

3. Compute next-word distribution:

$$P(w_t) = \text{softmax}(W_o h + b_o)$$

This replaces discrete count tables with a smooth, parametric function.

How Feedforward NNs Use Embeddings

Where do embeddings come from? They are the first layer of the neural LM: an embedding matrix $E \in \mathbb{R}^{|V| \times d}$, learned from scratch.

How are they used?

1. Take previous $n-1$ words: $w_{t-1}, \ldots, w_{t-(n-1)}$
2. Look up embeddings from E:

$$\mathbf{v}_{t-k} = E[w_{t-k}]$$

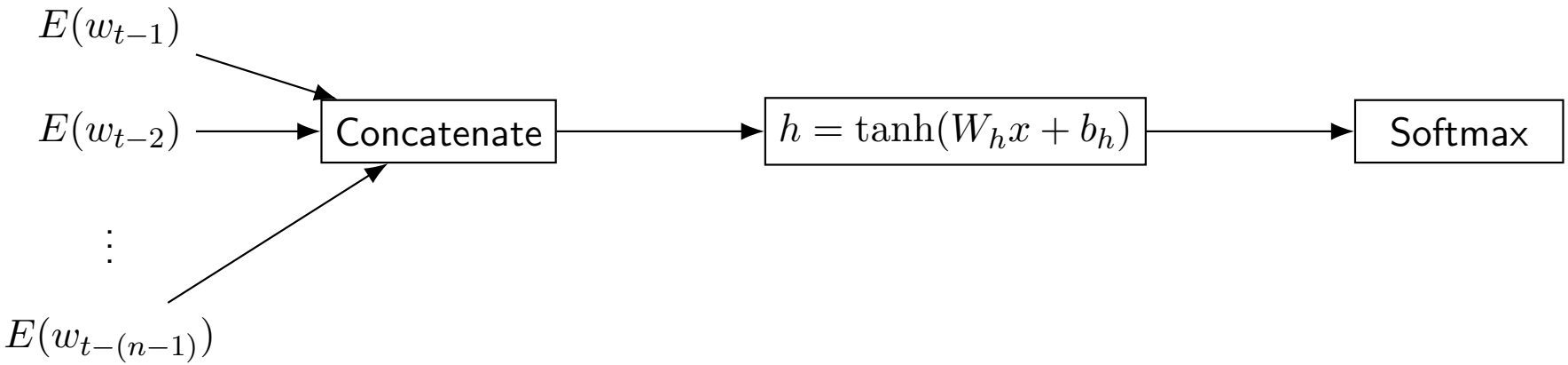
3. Concatenate into input vector $\mathbf{x}$
4. Pass through hidden layers:

$$h = \sigma(W_h \mathbf{x} + b_h)$$

5. Output probabilities via softmax.

How are embeddings learned? Backpropagation updates the rows of E just like any other parameters.

Visual: Feedforward Neural LM



Predicts $P(w_t \mid w_{t-1}, \ldots, w_{t-(n-1)})$.

Why Feedforward Neural LMs Outperform n-grams

- **Generalization:** Similar contexts produce embeddings close in vector space.
- **Parameter sharing:** Same parameters used regardless of specific words.
- **Smoothness:** Small changes in input $\rightarrow$ small changes in output.
- **Reduces sparsity:** No need for $O(|V|^n)$ tables.
- **Learns non-linear patterns** via hidden layers.

Why Feedforward Neural LMs Outperform n-grams

- **Generalization:** Similar contexts produce embeddings close in vector space.
- **Parameter sharing:** Same parameters used regardless of specific words.
- **Smoothness:** Small changes in input $\rightarrow$ small changes in output.
- **Reduces sparsity:** No need for $O(|V|^n)$ tables.
- **Learns non-linear patterns** via hidden layers.

But: still limited to a fixed window $\rightarrow$ motivates RNNs.

Softmax Output Layer

Given hidden state h, produce distribution over vocabulary V:

$$P(w_t = v) = \frac{\exp((W_o h + b_o)_v)}{\sum_{v' \in V} \exp((W_o h + b_o)_{v'})}$$

Properties:

- Ensures valid probabilities (non-negative, sum to 1).
- Scores all words in vocabulary simultaneously.
- Interpret output as the model's predicted next-word distribution.

Training the Model

Training uses **cross-entropy loss**:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(w_t \mid w_{t-1:t-(n-1)})$$

Gradients propagate through:

1. Output layer (softmax)
2. Hidden layer
3. Embedding matrix

Training the Model

Training uses **cross-entropy loss**:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(w_t \mid w_{t-1:t-(n-1)})$$

Gradients propagate through:

1. Output layer (softmax)
2. Hidden layer
3. Embedding matrix

Thus embeddings, hidden weights, and output weights are learned jointly.

Limitations of Feedforward Neural LMs

- Fixed context size (e.g. last 3–5 words).
- Context does not grow with sequence length.
- Cannot capture long-range dependencies.
- Computational cost increases with larger window.
- Not position-invariant (just concatenation of fixed slots).

Limitations of Feedforward Neural LMs

- Fixed context size (e.g. last 3–5 words).
- Context does not grow with sequence length.
- Cannot capture long-range dependencies.
- Computational cost increases with larger window.
- Not position-invariant (just concatenation of fixed slots).

Solution: Introduce **Memory (RNNs)** or **Attention (Transformers)** to maintain a dynamic memory over the entire sequence.

Why Feedforward Networks Cannot Capture Long-Range Dependencies

1. Fixed context window: The input is a concatenation of the last n word embeddings. Anything beyond that is invisible to the model.
2. Dimensionality explosion: Increasing the window increases input dimensionality linearly and parameters quadratically.
3. No memory: A feedforward network processes each window independently. It has no internal state that persists across time.
4. No sequence structure: The model treats the window as a static vector, not as a temporal sequence.

Thus feedforward LMs can only capture very local dependencies.

Why Word2Vec?

Feedforward neural language models learn embeddings as a by-product.

Word2Vec (Mikolov et al., 2013) simplifies this idea:

- Use a shallow neural network,
- Focus entirely on learning word embeddings,
- Use clever optimization (negative sampling),
- Train on billions of tokens efficiently.

Why Word2Vec?

Feedforward neural language models learn embeddings as a by-product.

Word2Vec (Mikolov et al., 2013) simplifies this idea:

- Use a shallow neural network,
- Focus entirely on learning word embeddings,
- Use clever optimization (negative sampling),
- Train on billions of tokens efficiently.

Two architectures:

1. **CBOW** (Continuous Bag-of-Words): predict target from context.
2. **Skip-gram**: predict context words from target.

CBOW: Predict Target from Context

Given context window $C = \{w_{t-k}, \ldots, w_{t+k}\}$: predict the center word w_t.

$$P(w_t \mid C) = \text{softmax}\big(W \cdot \tfrac{1}{|C|} \sum_{c \in C} E[c]\big)$$

- Embed each context word.
- Average the embeddings.
- Feed into softmax to predict target.

CBOW: Predict Target from Context

Given context window $C = \{w_{t-k}, \ldots, w_{t+k}\}$: predict the center word w_t.

$$P(w_t \mid C) = \text{softmax}\big(W \cdot \tfrac{1}{|C|} \sum_{c \in C} E[c]\big)$$

- Embed each context word.
- Average the embeddings.
- Feed into softmax to predict target.

Intuition: Words that share similar contexts get similar embeddings.

Skip-gram: Predict Context from Target

Given target word w_t, predict each context word $c \in C$:

$$P(c \mid w_t) = \text{softmax}(E[c]^\top E[w_t])$$

- Input: embedding of target word.
- Output: probability of each context word.
- Multiple predictions per training token.

Skip-gram: Predict Context from Target

Given target word w_t, predict each context word $c \in C$:

$$P(c \mid w_t) = \text{softmax}(E[c]^\top E[w_t])$$

- Input: embedding of target word.
- Output: probability of each context word.
- Multiple predictions per training token.

Advantages:

- Performs well on rare words.
- Captures rich semantic structure.

The Softmax Problem

For each training example, skip-gram requires:

$$\text{softmax}(E[c]^\top E[w_t]) \quad \text{over all } |V| \text{ words}$$

This is extremely expensive:

- Vocabulary size often $10^5 - 10^6$.
- Billions of predictions during training.

The Softmax Problem

For each training example, skip-gram requires:

$$\text{softmax}(E[c]^\top E[w_t]) \quad \text{over all } |V| \text{ words}$$

This is extremely expensive:

- Vocabulary size often $10^5 - 10^6$.
- Billions of predictions during training.

Solution: Replace full softmax with a faster approximation. This leads to **negative sampling**.

Negative Sampling

Instead of predicting a full distribution over vocabulary: Train the model to **distinguish observed context words from random negatives**.
For each true pair (w_t, c):

$$\log \sigma(E[c]^\top E[w_t])$$

For K negative samples $n_1, \ldots, n_K$:

$$\sum_{j=1}^{K} \log \sigma\big(- E[n_j]^\top E[w_t]\big)$$

Negative Sampling

Instead of predicting a full distribution over vocabulary: Train the model to **distinguish observed context words from random negatives**.
For each true pair (w_t, c):

$$\log \sigma(E[c]^\top E[w_t])$$

For K negative samples $n_1, \ldots, n_K$:

$$\sum_{j=1}^{K} \log \sigma\big(- E[n_j]^\top E[w_t]\big)$$

Goal: Push real context words closer in embedding space, push random words farther away.
Result: Massive speedup; high-quality embeddings.

Why Word2Vec Learns Semantic Structure

Skip-gram with negative sampling optimizes:

$$\max_{E} \sum_{(w,c)\in D} \left[\log \sigma(E[c]^\top E[w]) + \sum_{n\sim P_n} \log \sigma(-E[n]^\top E[w]) \right]$$

This objective ensures:

- **Co-occurring** words get similar vectors.
- **Non-co-occurring** words get pushed apart.
- Linear directions encode relationships (e.g., gender, plural, tense).

Why Word2Vec Learns Semantic Structure

Skip-gram with negative sampling optimizes:

$$\max_{E} \sum_{(w,c)\in D} \left[\log \sigma(E[c]^\top E[w]) + \sum_{n\sim P_n} \log \sigma(-E[n]^\top E[w]) \right]$$

This objective ensures:

- **Co-occurring** words get similar vectors.
- **Non-co-occurring** words get pushed apart.
- Linear directions encode relationships (e.g., gender, plural, tense).

Hence analogies like:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}.$$

Word2Vec: Limitations and Legacy

Limitations:

- One vector per word (no polysemy).
- No context.
- Ignores word order inside window.
- Shallow architecture (no hierarchy).

Word2Vec: Limitations and Legacy

Limitations:

- One vector per word (no polysemy).
- No context.
- Ignores word order inside window.
- Shallow architecture (no hierarchy).

Legacy:

- Demonstrated power of representation learning.
- Scaled neural embeddings to huge corpora.
- Inspired fastText, contextual embeddings (ELMo, BERT).
- Crucial stepping stone toward modern LMs.

From Embeddings to Measurement

Static word embeddings (Word2Vec, GloVe) give each word:

$$w \longrightarrow \mathbf{v}_w \in \mathbb{R}^d$$

Key idea: If meaning is encoded geometrically, then *cultural or political dimensions* should correspond to *directions* in the space.

From Embeddings to Measurement

Static word embeddings (Word2Vec, GloVe) give each word:

$$w \longrightarrow \mathbf{v}_w \in \mathbb{R}^d$$

Key idea: If meaning is encoded geometrically, then *cultural or political dimensions* should correspond to *directions* in the space.
Two influential applications:

- **Latent Semantic Scaling (Watanabe, 2020):** measure ideology or sentiment by projecting onto a semantic direction.
- **The Geometry of Culture (Kozlowski et al., 2019):** uncover cultural schemas using geometric axes in embedding spaces.

Latent Semantic Scaling (LSS)

LSS uses static embeddings to place words, documents, or actors on a *latent semantic dimension*.

Core idea:

$$\text{score}(w) = \mathbf{v}_w \cdot (\mathbf{v}_+ - \mathbf{v}_-)$$

Where:

- $\mathbf{v}_+$ = average embedding of positive/pole words
- $\mathbf{v}_-$ = average embedding of negative/opposing words

This defines a semantic direction.

Semantic Directions in Embedding Space

Given seed words:

$$D = \mathbf{v}_+ - \mathbf{v}_-$$

Any word gets:

$$\text{LSS}(w) = \mathbf{v}_w \cdot D$$

Interpretation:

- Positive score $\rightarrow$ closer to the positive pole
- Negative score $\rightarrow$ closer to the negative pole
- Magnitude $\rightarrow$ strength of association

Examples of Latent Semantic Scaling

Ideology:

$$+\text{pole} : \{\text{freedom}, \text{ market}, \text{tradition}\}$$

$$-\text{pole} : \{\text{equality}, \text{ rights}, \text{redistribution}\}$$

Sentiment:

$$+\text{pole} : \{\text{good}, \text{love}, \text{happy}\}$$

$$-\text{pole} : \{\text{bad}, \text{hate}, \text{sad}\}$$

Documents scored by averaging their words:

$$\text{LSS}(d) = \frac{1}{|d|} \sum_{w \in d} \text{LSS}(w)$$

Why LSS?

- Extremely fast, only requires dot products.
- Works with pretrained static embeddings.
- Uncovers *latent ideological or cultural bias*.
- Useful for political ideology estimation, framing, discourse analysis.

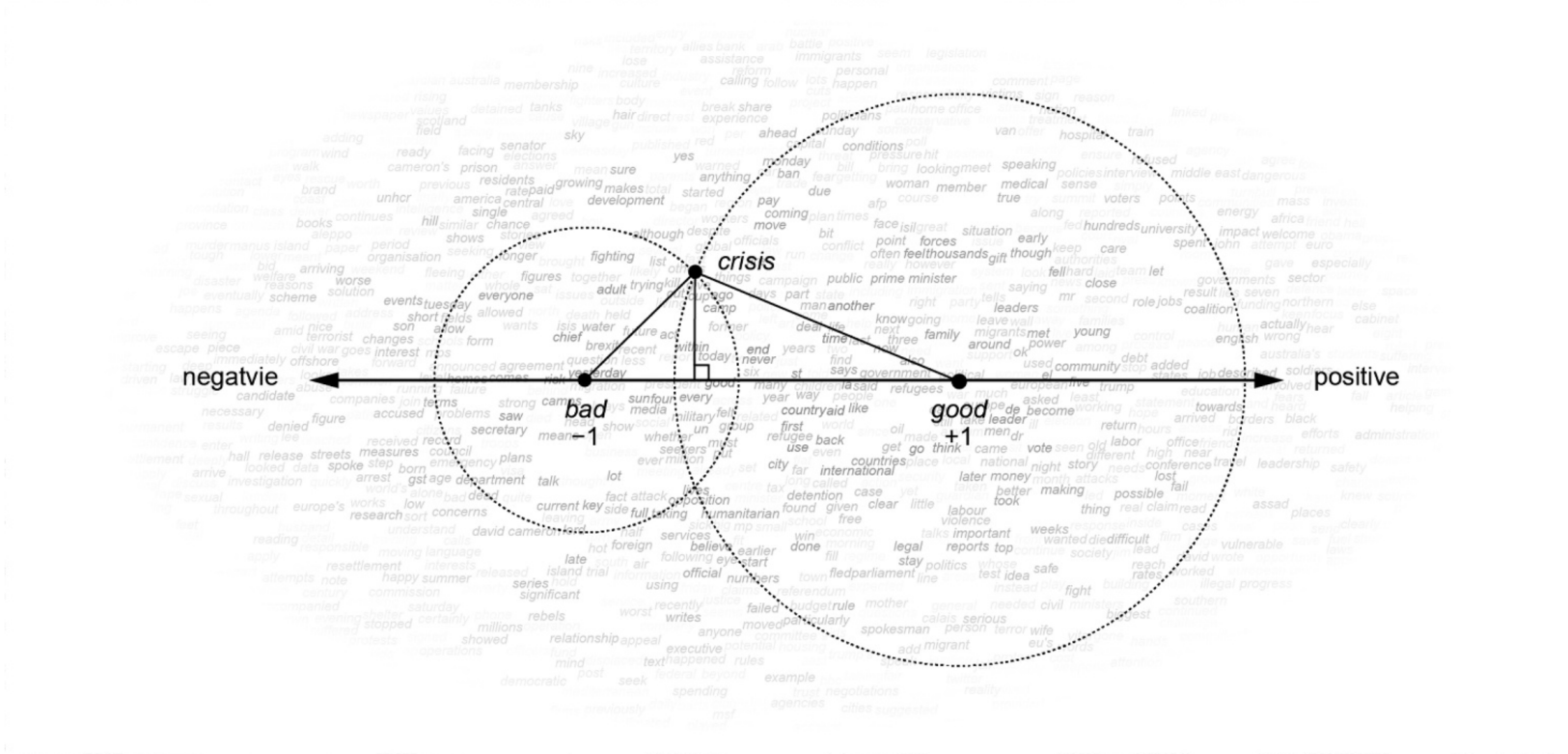


Figure 1. Conceptual illustration of word weighting by seed words in a latent semantic space. The arrow is the sentiment dimension in the semantic space and circles are proximities of positive seed ("good") words and negative seed words ("bad") to "crisis", which is projected on the sentiment dimension and receives a negative score.

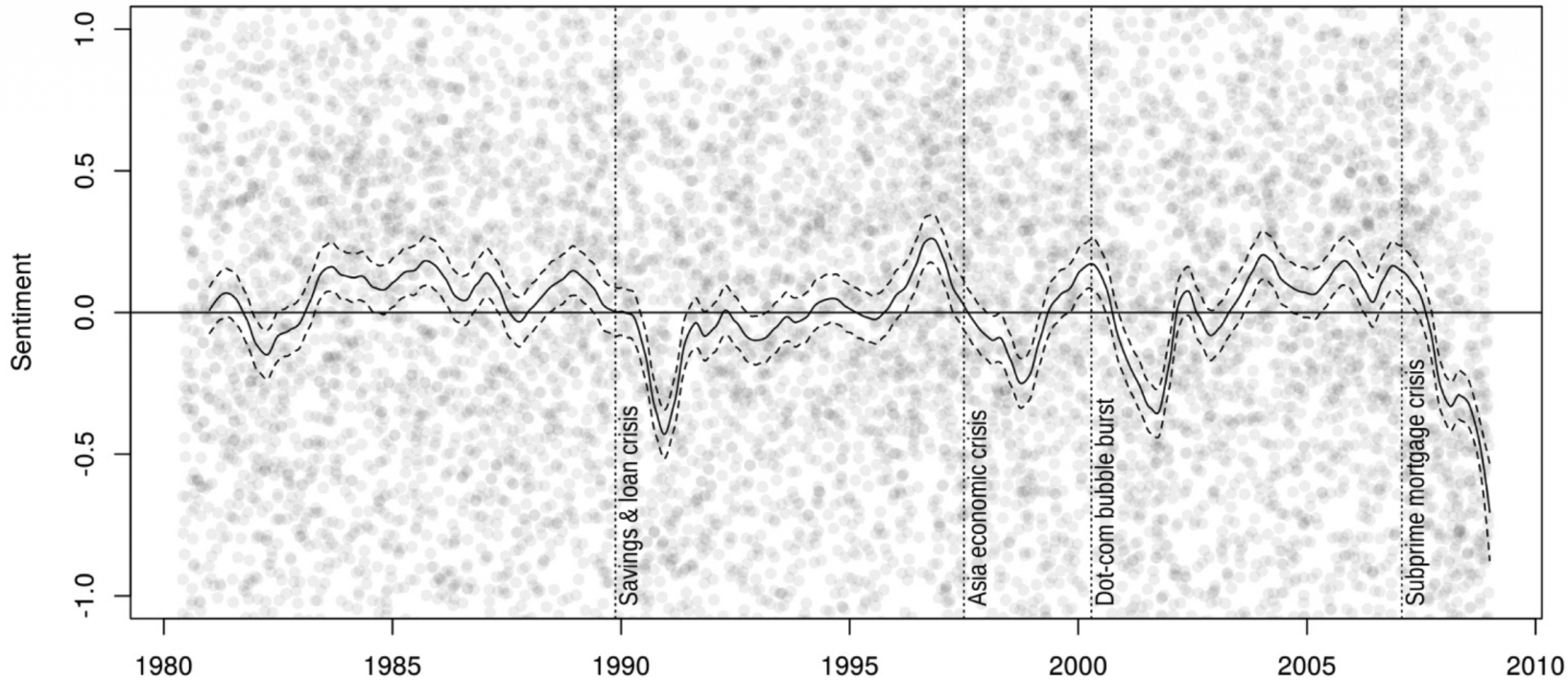


Figure 7. Longitudinal analysis of news articles on the economy (English) in the *New York Times* corpus by LSS. Curves are LOESS smoothed sentiment scores with 95% confidence intervals. Circles are individual sentiment scores of 10,000 news articles.

AMERICAN SOCIOLOGICAL ASSOCIATION

American Sociological Review
2019, Vol. 84(5) 905–949

DOI: 10.1177/0003122419877135
journals.sagepub.com/home/asr

SAGE

The Geometry of Culture: Analyzing the Meanings of Class through Word Embeddings

Austin C. Kozlowski,[a] Matt Taddy,[b] and James A. Evans[a,c]

The Geometry of Culture (Kozlowski et al., 2019)

Premise: Culture is encoded in geometric relationships in embedding space.

They model cultural schemas as:

- high-dimensional structures
- represented by semantic axes
- learned from large corpora (Google Books Ngrams)

The Geometry of Culture (Kozlowski et al., 2019)

Premise: Culture is encoded in geometric relationships in embedding space.

They model cultural schemas as:

- high-dimensional structures
- represented by semantic axes
- learned from large corpora (Google Books Ngrams)

Example schemas:

- gender
- class
- race
- modernity/tradition

Semantic Axes Represent Cultural Schemas

Given a binary concept:

$$\text{axis} = \mathbf{v}_A - \mathbf{v}_B$$

Example:

$$\text{gender axis} = \mathbf{v}_{\text{man}} - \mathbf{v}_{\text{woman}}$$

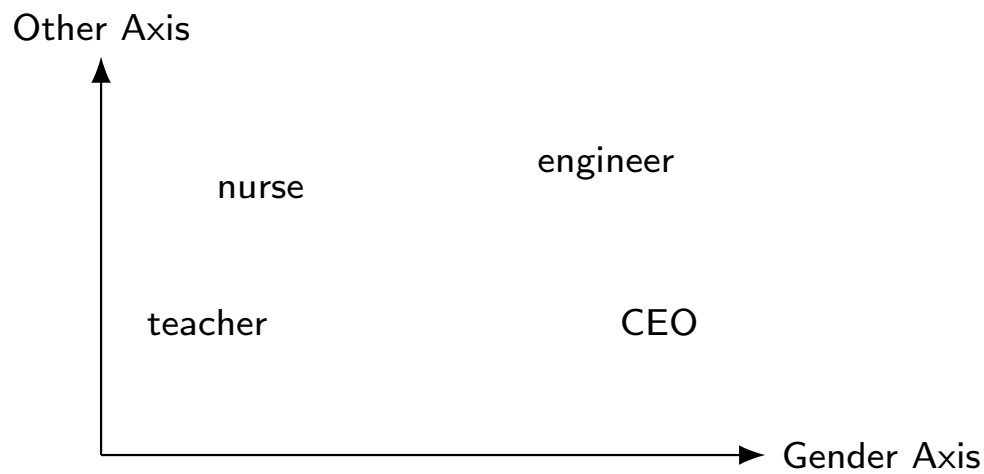
Then any concept c has projection:

$$\text{score}(c) = \mathbf{v}_c \cdot \text{axis}$$

This reveals cultural associations:

- occupations
- adjectives
- political concepts

Projections Reveal Cultural Structure



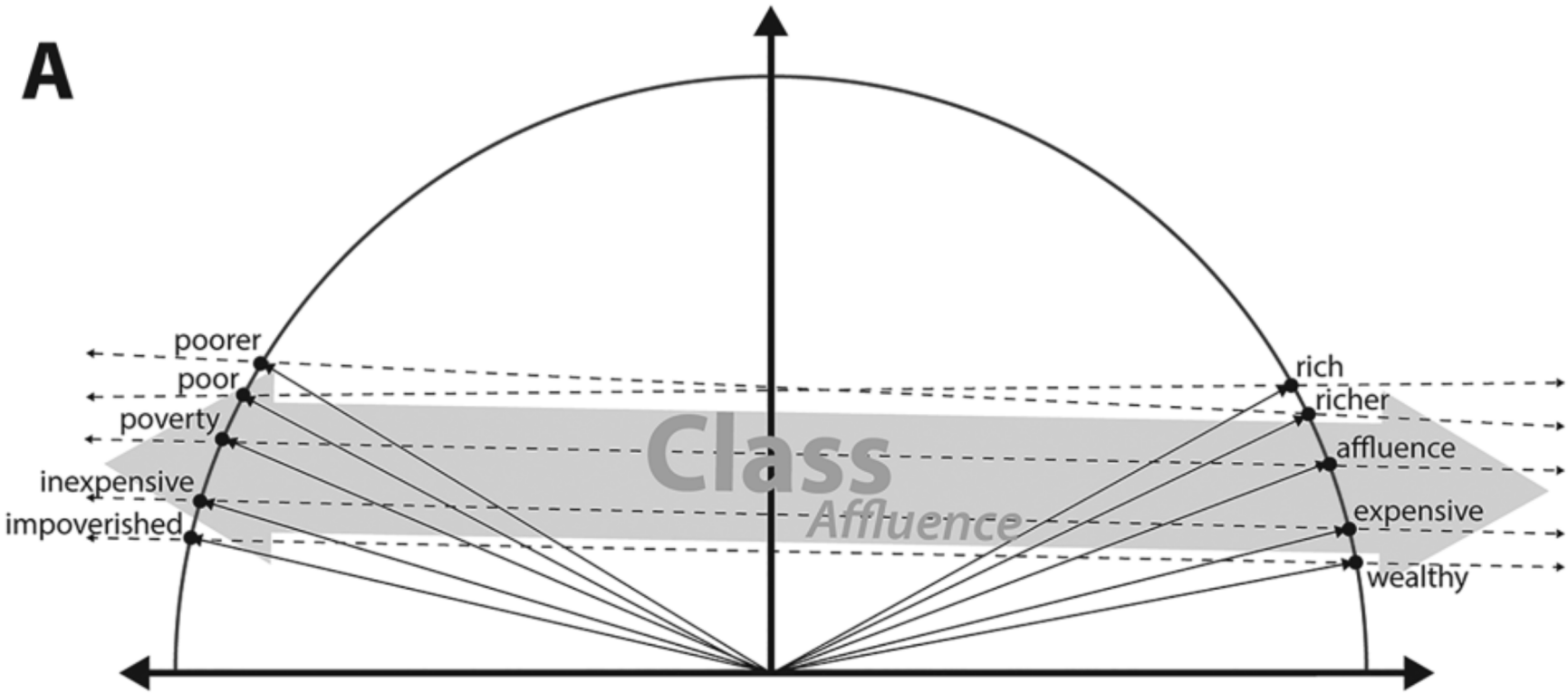
Cultural schemas emerge as geometric patterns.

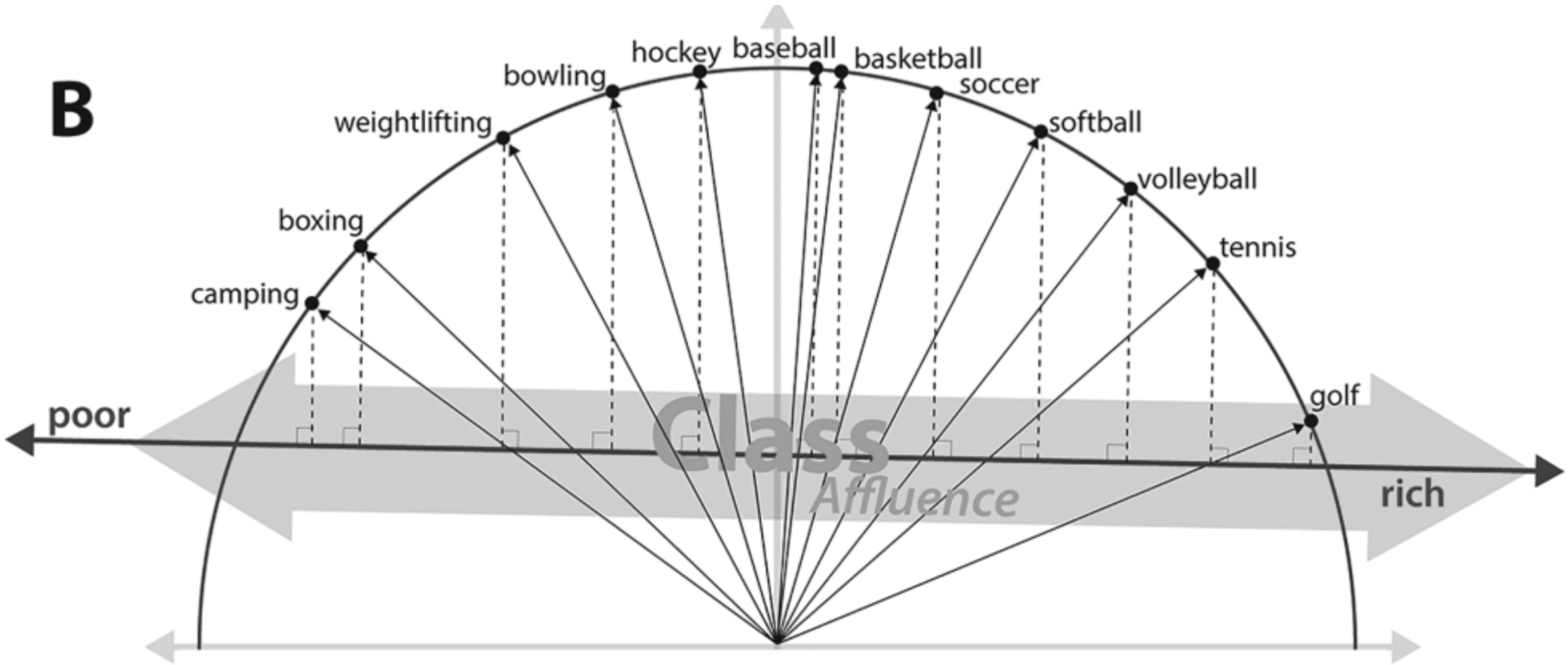
Why the Geometry of Culture Matters

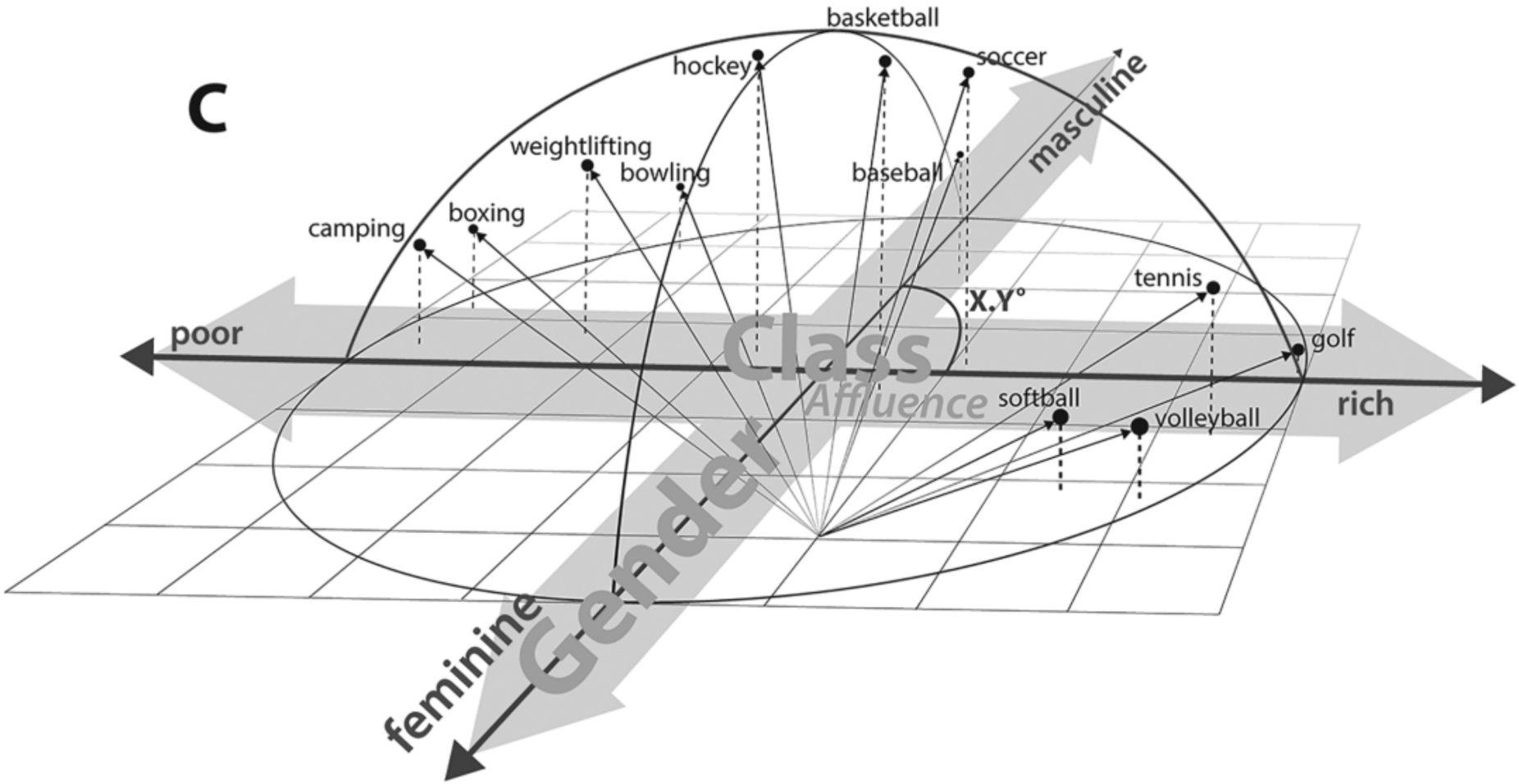
Implication: Cultural meaning can be measured empirically as *locations in vector space*.
Applications:

- cultural sociology
- ideology & belief systems
- bias in language
- representation of social categories

Key insight: Meaning is not discrete, it is geometric.


A
poorer
poor
poverty
inexpensive
impoverished
Class
Affluence
rich
richer
affluence
expensive
wealthy


B
hockey
baseball
basketball
bowling
soccer
weightlifting
softball
volleyball
boxing
tennis
camping
golf
poor
Class
Affluence
rich


C
basketball
hockey
soccer
weightlifting
bowling
baseball
camping
boxing
masculine
tennis
X.Y°
poor
Class
Affluence
golf
softball
rich
volleyball
Gender
feminine

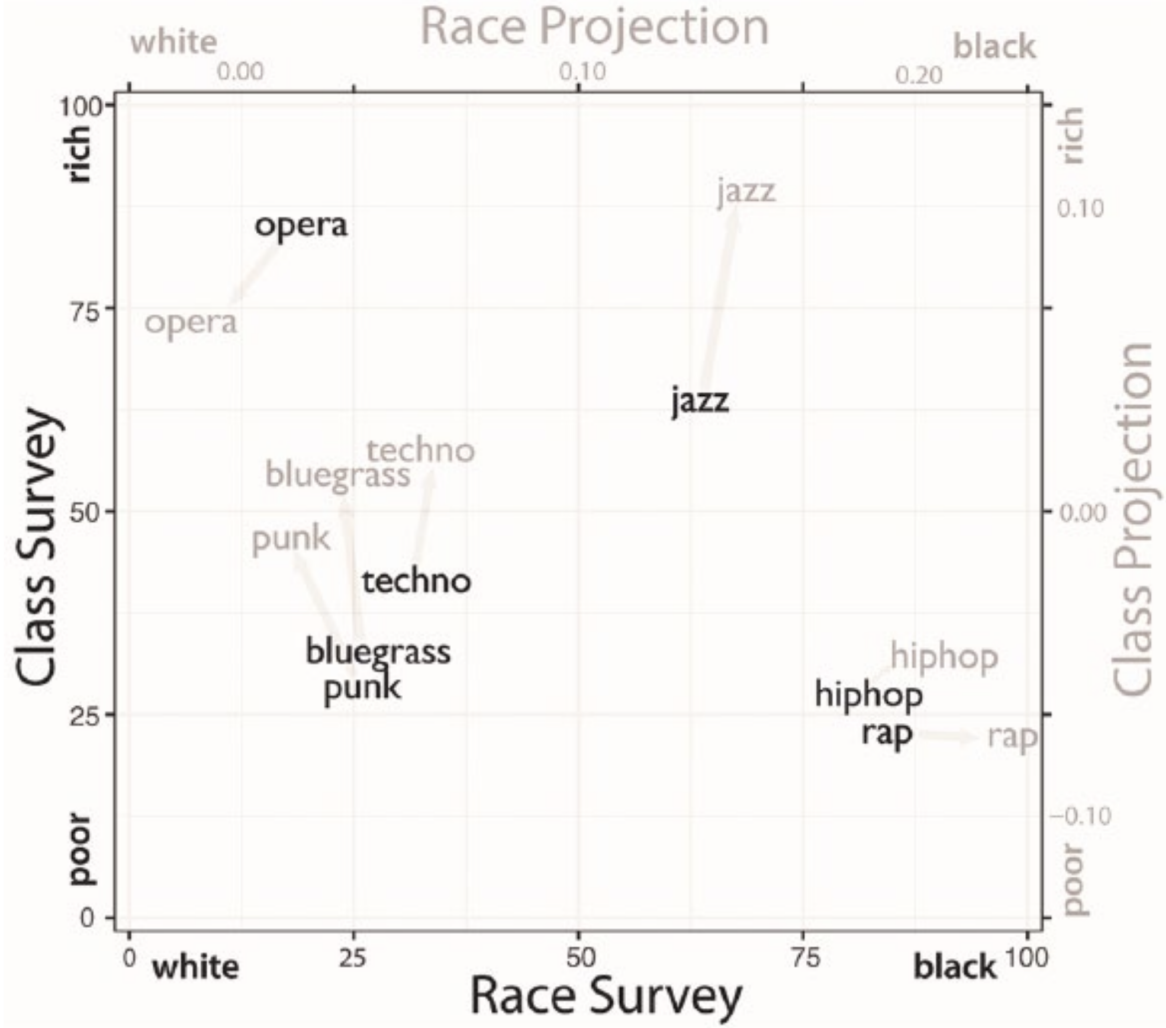


Figure 3. Projection of Music Genres onto Race and Class Dimensions of the Google News Word Embedding (Gray) and Average Survey Ratings for Race and Class Associations (Black)

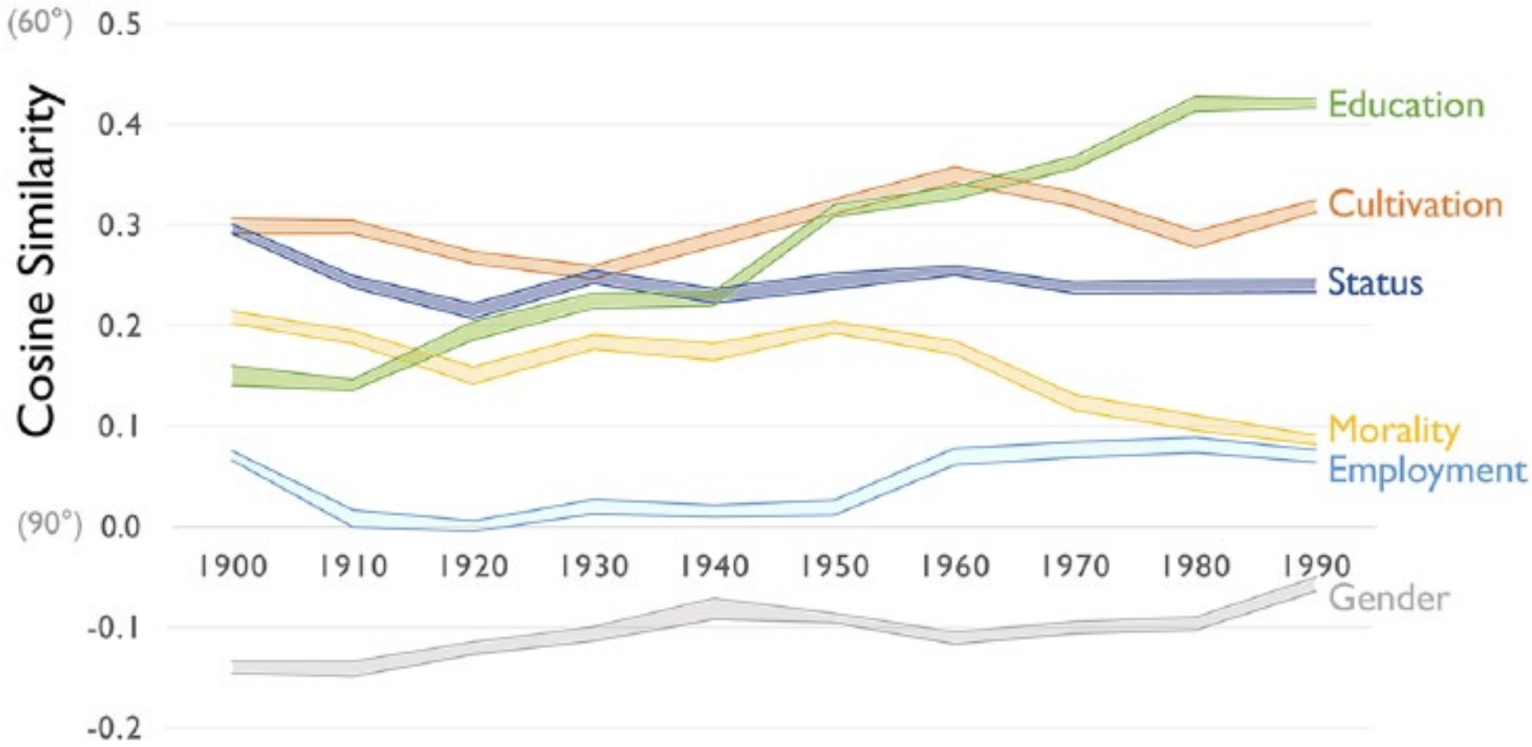
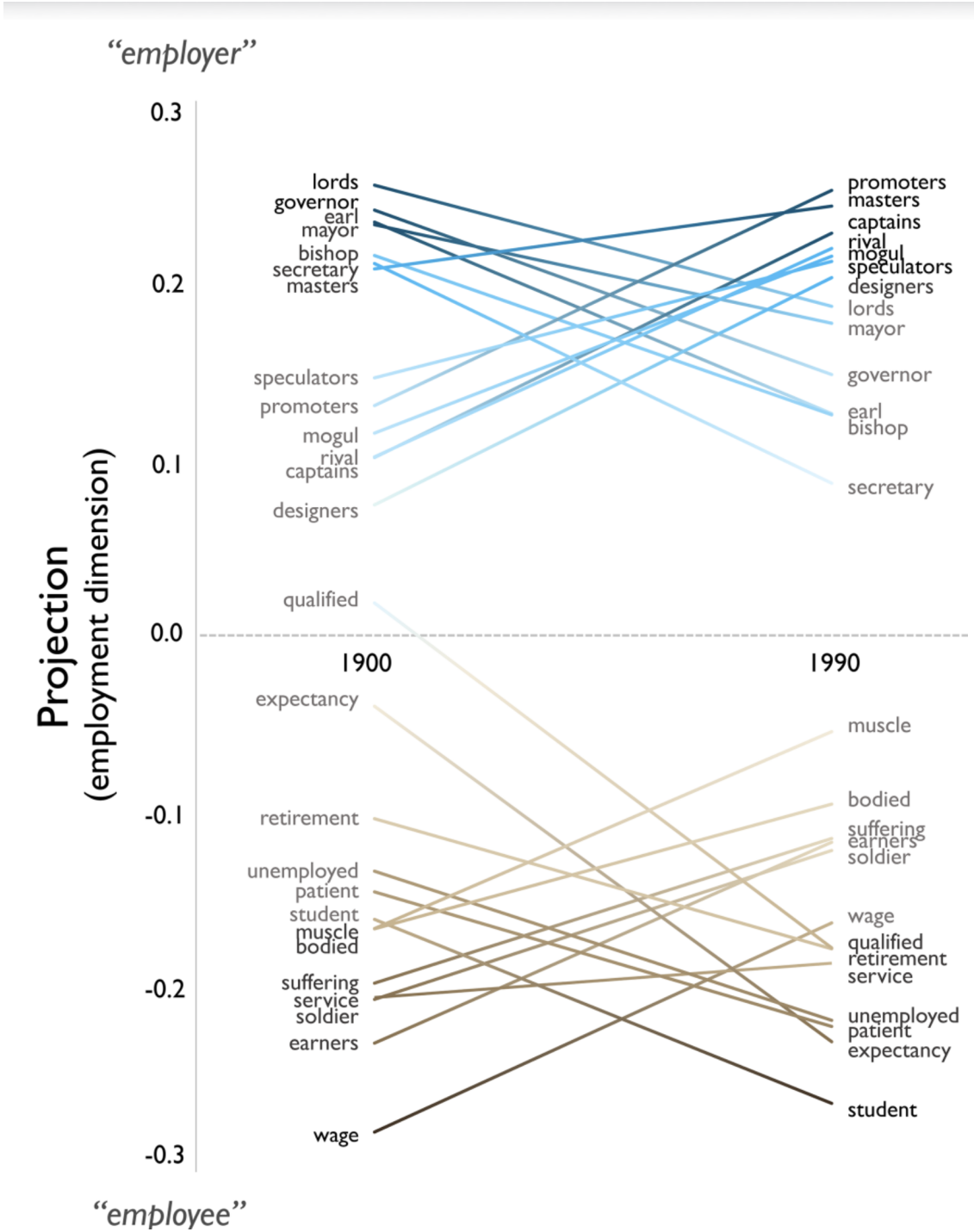


Figure 5. Cosine Similarity between the Affluence Dimension and Six Other Cultural Dimensions of Class by Decade; 1900 to 1999 Google Ngrams Corpus
Note: Bands represent 90 percent bootstrapped confidence intervals produced by subsampling.


"employer"
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
Projection
(employment dimension)
lords
governor
earl
mayor
bishop
secretary
masters
speculators
promoters
mogul
rival
captains
designers
qualified
promoters
masters
captains
rival
mogul
speculators
designers
lords
mayor
governor
earl
bishop
secretary
1900
1990
expectancy
retirement
unemployed
patient
student
muscle
bodied
suffering
service
soldier
earners
wage
muscle
bodied
suffering
earners
soldier
wage
qualified
retirement
service
unemployed
patient
expectancy
student
"employee"

From Static to Contextual Meaning

Static embeddings assume:

$$\mathbf{v}_{\text{bank}} = \text{one vector}$$

But meaning is context-dependent:

- "river bank"
- "central bank"

From Static to Contextual Meaning

Static embeddings assume:

$$\mathbf{v}_{\text{bank}} = \text{one vector}$$

But meaning is context-dependent:

- "river bank"
- "central bank"

Next: Contextual embeddings (BERT, GPT) represent meaning as a function of surrounding words.