Assignment 2 — Solutions

Text Representation & Word Embeddings

Part A — Foundations · 23 questions with answers

Q1. What is a vector space model in NLP?

Answer

A **vector space model** represents every word or document as a numeric vector in a multi-dimensional space. Each dimension is a feature (a vocabulary word, or a learned latent feature). Texts with similar meaning lie close together, so we can measure similarity with cosine similarity or distance, and feed the vectors to ML models. Examples: BoW and TF-IDF (sparse, dimension = vocabulary size) and embeddings (dense, e.g. 300 or 768 dimensions).

Q2. What is the Bag-of-Words model? What are its limitations? ★

Answer

Bag-of-Words (BoW) represents a document as the **count (or presence) of each vocabulary word**, ignoring grammar and word order.

Example – vocabulary [I, love, NLP, hate]: "I love NLP" → [1, 1, 1, 0].

Limitations:

- Loses word order: "dog bites man" = "man bites dog".
- No semantics: "good" and "great" are unrelated dimensions.
- High-dimensional and sparse (vocab can be 100k+).
- Cannot handle OOV words.
- Frequent words dominate unless weighted (→ TF-IDF).

Q3. What is a Bag of N-grams model? What problem does it solve over plain BoW?

Answer

A **Bag of N-grams** counts sequences of N consecutive words instead of single words. Bigrams of "not good movie" → ["not good", "good movie"].

It solves part of BoW's **word-order problem**: "not good" becomes one feature, so the model can detect negation and short phrases ("New York", "machine learning").

Downside: the vocabulary grows very quickly → even sparser vectors; usually limited to n = 1–3.

Q4. What is TF-IDF? How do you calculate TF, IDF and TF-IDF? Work out a small example by hand. ★ · Asked: NLP interview rounds

Answer

TF-IDF weights a word by how important it is to a document relative to the whole corpus.

- **TF(t, d)** = (count of t in d) / (total words in d)
- **IDF(t)** = log(N / df(t)), where N = number of documents, df = number of documents containing t
- **TF-IDF = TF × IDF**

Worked example (log base 10) – 3 documents: D1 "the cat sat", D2 "the dog sat", D3 "the cat ran".

- Word "cat" in D1: TF = 1/3; df = 2 → IDF = log(3/2) = 0.176 → TF-IDF = 0.059
- Word "the" in D1: TF = 1/3; df = 3 → IDF = log(3/3) = 0 → TF-IDF = 0 (common word, no information)
- Word "ran" in D3: TF = 1/3; df = 1 → IDF = log(3) = 0.477 → TF-IDF = 0.159 (rare word, high weight)

(scikit-learn uses a smoothed version: idf = ln((1+N)/(1+df)) + 1, then L2-normalizes the vector.)

Q5. What are word embeddings? How are they better than one-hot / BoW vectors? ★

Answer

Word embeddings are dense, low-dimensional vectors (e.g. 100–1024 numbers) learned from data, where words with similar meaning have similar vectors.

Better than one-hot/BoW because:

- **Dense** (300 dims vs 100k sparse dims) → less memory, better generalization.
- **Capture semantics:** cos(king, queen) is high; famous analogy king − man + woman ≈ queen.
- Similar words share information, so the model generalizes to words it saw less often.
- Can be pre-trained on huge corpora and reused (transfer learning).

Q6. Which algorithms are used to train word embeddings?

Answer

- **Word2Vec** (CBOW and Skip-gram) – Google, 2013
- **GloVe** – Stanford, global co-occurrence matrix factorization
- **FastText** – Facebook, Word2Vec with character n-grams
- **Doc2Vec** – paragraph/document vectors
- **ELMo** – contextual embeddings from bi-LSTMs
- **BERT / Transformer-based** contextual embeddings
- **Sentence embedding models** – Sentence-BERT, OpenAI text-embedding-3, BGE, E5 (used in RAG)

Q7. Explain Word2Vec and its two architectures (CBOW and Skip-gram). What is the difference? ★ · Asked: NLP interview rounds

Answer

Word2Vec is a shallow neural network (one hidden layer) that learns word vectors by predicting words from their neighbours within a context window. The learned hidden-layer weights are the embeddings.

- **CBOW (Continuous Bag of Words):** predicts the **target word from the surrounding context words**. Context "the cat __ on the mat" → "sat". Faster, works well for frequent words.
- **Skip-gram:** predicts the **context words from the target word**. "sat" → "the", "cat", "on", "mat". Slower, but better for rare words and small datasets.

Training tricks: **negative sampling** and **hierarchical softmax** avoid computing softmax over the whole vocabulary.

Q8. What are the key hyperparameters of Word2Vec (vector_size, window, min_count, sg, negative) and their role?

Answer

- **vector_size** – embedding dimension (100–300). Larger = more expressive but needs more data.
- **window** – max distance between target and context words. Small (2–5) → syntactic similarity; large (5–10) → topical similarity.
- **min_count** – ignore words with frequency below this; removes noise/typos.
- **sg** – 0 = CBOW, 1 = Skip-gram.
- **negative** – number of negative samples (5–20); 0 means use hierarchical softmax (`hs=1`).
- **epochs** – passes over the corpus.
- **alpha** – learning rate; **sample** – down-sampling of very frequent words; **workers** – CPU threads.

Q9. Compare Word2Vec vs FastText vs GloVe vs Doc2Vec. · Asked: NLP interview rounds

Answer

- **Word2Vec** – predictive model using local context windows; one vector per word; cannot handle OOV.
- **GloVe** – count-based; factorizes a **global** word co-occurrence matrix; combines global statistics with local context; also no OOV support.
- **FastText** – like Word2Vec but each word = sum of its **character n-grams** ("where" → "<wh", "whe", "her", "ere", "re>"); handles **OOV and misspellings**, good for morphologically rich languages.
- **Doc2Vec** – extends Word2Vec with a **document/paragraph vector** (PV-DM and PV-DBOW) to produce a fixed vector for a whole document.

All four are **static** (one vector per word regardless of context).

Q10. What is the difference between static embeddings (Word2Vec) and contextual embeddings (BERT)? Give an example with the word "bank".

Answer

- **Static embeddings** (Word2Vec, GloVe) give **one fixed vector per word**, whatever the sentence.
- **Contextual embeddings** (ELMo, BERT, GPT) produce a **different vector for each occurrence** depending on surrounding words.

Example: "I deposited money in the **bank**" vs "We sat on the river **bank**". Word2Vec gives the same vector for both; BERT gives two different vectors – one close to "finance", the other close to "river". This is why contextual models handle ambiguity much better.

Q11. How do you handle out-of-vocabulary (OOV) words? ★

Answer

- **Subword tokenization** (BPE/WordPiece/SentencePiece) – any word can be built from known pieces; this is the standard solution today.
- **FastText** – builds vectors for unknown words from character n-grams.
- **Character-level models.**
- Replace with an **<UNK> token** (simple but loses information).
- Spelling correction / lemmatization to map to known words.

- Use a larger vocabulary or fine-tune on domain data.

Q12. Explain cosine similarity and why it is important in NLP. ★

Answer

Cosine similarity measures the cosine of the angle between two vectors:

$\cos(\theta) = (A \cdot B) / (\|A\| \times \|B\|)$, ranging from -1 (opposite) through 0 (unrelated) to 1 (same direction).

Why important in NLP:

- It measures **direction (meaning), not magnitude (length)**, so a long and a short document on the same topic still score high.
- Works well for high-dimensional sparse and dense vectors.
- It is the default metric for semantic search, RAG retrieval, recommendation, duplicate detection and clustering.

Q13. What similarity / distance metrics are used (cosine, Euclidean, dot product, Jaccard)?

Answer

- **Cosine similarity** – angle between vectors (most common for text).
- **Dot product** – cosine × magnitudes; equals cosine for normalized vectors; used by many vector DBs.
- **Euclidean distance (L2)** – straight-line distance.
- **Manhattan distance (L1)** – sum of absolute differences.
- **Jaccard similarity** – $|A \cap B| / |A \cup B|$ for sets of words (good for near-duplicate detection).
- **Levenshtein (edit) distance** – for string/spelling similarity.
- **BM25** – ranking function for keyword search.

Q14. Euclidean distance vs cosine similarity — when would you use which? ★

Answer

- **Euclidean distance** depends on vector **magnitude**. A long document has a larger count vector than a short one, so they look far apart even if they are about the same topic.
- **Cosine similarity** depends only on the **angle** (direction), so it ignores length.

Use cosine for text, embeddings, semantic search (most cases). **Use Euclidean** when magnitude is meaningful (e.g. K-means on normalized numeric features, image features). Note: for L2-normalized vectors, ranking by Euclidean distance and by cosine give the same order.

Q15. For measuring similarity, why do we use cosine and not sine or tan?

Answer

Because cosine behaves exactly like a similarity score should:

- $\cos(0°) = 1 \rightarrow$ vectors in the same direction = most similar.
- $\cos(90°) = 0 \rightarrow$ orthogonal = unrelated.
- $\cos(180°) = -1 \rightarrow$ opposite.

It is **monotonically decreasing** from 0° to 180°, bounded in $[-1, 1]$, and is cheap to compute from the dot product.

Sine is 0 for both identical (0°) and opposite (180°) vectors, so it cannot distinguish them, and it increases as vectors diverge (a distance-like measure). **Tangent** is unbounded (infinite at 90°) and undefined there, so it is unusable as a score.

Q16. How do you check the similarity between two words or two documents?

Answer

1. Convert each word/document into a vector – TF-IDF, Word2Vec (average word vectors for a document), or better, a **sentence-embedding model**.
2. Compute **cosine similarity** between the vectors.

```
from sentence_transformers import SentenceTransformer, util
m = SentenceTransformer("all-MiniLM-L6-v2")
e = m.encode(["I love machine learning", "I enjoy studying AI", "The weather is hot"])
print(util.cos_sim(e[0], e[1]))   # high  (~0.6)
print(util.cos_sim(e[0], e[2]))   # low   (~0.05)

# Word similarity with Word2Vec
# w2v.wv.similarity("king", "queen")
```

Other options: Jaccard similarity for word overlap, BM25 for keyword relevance.

Q17. Create vectors using CountVectorizer (unigrams and bigrams). [CODE]

Answer

```
from sklearn.feature_extraction.text import CountVectorizer
docs = ["I love NLP", "I love deep learning", "NLP is fun"]

# Unigrams
cv1 = CountVectorizer()
X1 = cv1.fit_transform(docs)
print(cv1.get_feature_names_out())
print(X1.toarray())

# Unigrams + bigrams
cv2 = CountVectorizer(ngram_range=(1, 2))
X2 = cv2.fit_transform(docs)
print(cv2.get_feature_names_out())
# ['deep' 'deep learning' 'fun' 'is' 'is fun' 'learning' 'love' 'love deep' 'love nlp' 'nlp' 'nlp
    is']
print(X2.toarray())

# Only bigrams: CountVectorizer(ngram_range=(2, 2))
```

Note: the default token pattern drops 1-character tokens like "I".

Q18. Create vectors using TfidfVectorizer and print the top-weighted terms of a document. [CODE]

Answer

```
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

docs = ["the cat sat on the mat",
        "the dog sat on the log",
        "cats and dogs are great pets"]
tfidf = TfidfVectorizer(stop_words="english", ngram_range=(1, 2))
X = tfidf.fit_transform(docs)
vocab = tfidf.get_feature_names_out()

# Top-3 weighted terms of document 0
row = X[0].toarray().ravel()
top = np.argsort(row)[::-1][:3]
print([(vocab[i], round(row[i], 3)) for i in top])
```

Parameters to know: `max_features`, `min_df`, `max_df`, `ngram_range`, `sublinear_tf`, `stop_words`.

Q19. Train (1) Word2Vec, (2) FastText and (3) Doc2Vec on a small corpus and find similar words. [CODE]

Answer

```
from gensim.models import Word2Vec, FastText
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from nltk.tokenize import word_tokenize

corpus = ["machine learning is fun", "deep learning is a part of machine learning",
          "nlp uses deep learning", "transformers are deep learning models for nlp"]
tokens = [word_tokenize(s.lower()) for s in corpus]

# 1) Word2Vec (sg=1 -> skip-gram)
w2v = Word2Vec(tokens, vector_size=50, window=3, min_count=1, sg=1, epochs=100)
print(w2v.wv.most_similar("learning", topn=3))

# 2) FastText - works for unseen words
ft = FastText(tokens, vector_size=50, window=3, min_count=1, epochs=100)
print(ft.wv.most_similar("learnings", topn=3))     # OOV word still works

# 3) Doc2Vec
tagged = [TaggedDocument(t, [i]) for i, t in enumerate(tokens)]
d2v = Doc2Vec(tagged, vector_size=50, min_count=1, epochs=100)
vec = d2v.infer_vector(word_tokenize("deep learning for nlp"))
print(d2v.dv.most_similar([vec], topn=2))
```

On a toy corpus the similarities are noisy; use a real corpus (e.g. a few MB of text) for meaningful results.

Q20. Why can averaging Word2Vec vectors to make a sentence vector lose information? What is better? ★ · Asked: NLP interview rounds

Answer

Averaging (mean pooling of word vectors) is a simple baseline, but:

- **Word order is lost:** "dog bites man" and "man bites dog" get the same vector.
- **Negation and composition are lost:** "not good" ≈ average of "not" and "good", which stays close to "good".
- **Important words get diluted** by many common/filler words in long sentences.
- **Each word has one static vector**, so polysemy ("bank") is not resolved.
- OOV words are simply missing.

Better options: TF-IDF–weighted averaging (down-weights common words), SIF embeddings, Doc2Vec, and today mainly **Sentence Transformers / embedding models** (SBERT, BGE, E5, OpenAI embeddings) that are trained specifically to produce sentence-level vectors with context and word order.

Q21. What are Sentence Transformers (SBERT)? Why are they better for semantic search than averaged word vectors or BERT's raw [CLS] vector? ★ · Asked: NLP interview rounds

Answer

Sentence Transformers (SBERT) are Transformer encoders (BERT/MPNet/MiniLM) fine-tuned with a **siamese / bi-encoder** setup on sentence pairs (NLI, paraphrase, question–answer pairs) using **contrastive losses**, so that semantically similar sentences produce embeddings with high cosine similarity.

Why better:

- Raw BERT [CLS] or mean-pooled BERT vectors were **not trained for similarity** – their cosine scores are poorly calibrated (often worse than GloVe averages for similarity tasks).

- Averaged Word2Vec ignores order and context.
- SBERT encodes the **whole sentence in context** into one vector that directly supports cosine search, clustering and RAG retrieval; it is fast because documents are encoded once and compared with dot products.

```
from sentence_transformers import SentenceTransformer, util
model = SentenceTransformer("all-MiniLM-L6-v2")
q = model.encode("How do I get my money back?")
d = model.encode(["Refund policy: refunds within 7 days", "Office timings are 9 to 6"])
print(util.cos_sim(q, d))   # first document scores much higher
```

Q22. Pretrained embeddings vs training embeddings on your own domain data – which would you choose and when? · Asked: NLP interview rounds

Answer

Use pretrained embeddings (default): small or medium datasets; general language; quick start. Models trained on billions of words capture semantics you cannot learn from a few thousand documents.

Train or fine-tune on domain data when: the domain has its own vocabulary or word meanings (medical, legal, banking codes, internal product names), the pretrained model has many OOV/rare terms, and you have a **large** domain corpus (millions of tokens) – or fine-tune a pretrained model (continue training / contrastive fine-tuning of a sentence-embedding model on domain query–document pairs).

Best practice: start with a strong pretrained model, **measure** on your task (e.g. retrieval recall@k), then fine-tune if domain terms are the cause of errors.

Q23. Explain embeddings to a non-technical person without using the word "vector". ★ · Asked: Deloitte

Answer

An embedding is like giving every word or sentence a **position on a giant map of meaning**. Things that mean similar things are placed close together – 'refund', 'money back' and 'return payment' sit in the same neighbourhood, while 'office timings' is far away. The computer learns these positions by reading huge amounts of text and noticing which words appear in similar situations.

So when a customer asks 'How do I get my money back?', we look up where that question lands on the map and pick the documents placed nearest to it – even if they use completely different words like 'refund policy'. That's how the system finds answers by **meaning**, not just by matching exact words.