Context-Aware Synonym Replacement Using Learned Representations

Sebastian Lopez Diego Bonilla
Texas Tech University

CS4391 – Homework 1
Spring 2026

1 Introduction

The goal of this homework is to design and analyze a context-aware synonym replacement system using learned representations. Rather than focusing on fluent text generation, the objective is to explore how pretrained embeddings encode meaning, how context can be approximated using similarity measures, and where such approaches fail—especially in metaphorical, poetic, or ambiguous language.

To achieve this, we implemented a multi-stage pipeline that replaces selected words in a sentence with near-synonyms while attempting to preserve sentence-level meaning. The system explicitly avoids large language models, masked language models, and rule-based synonym dictionaries, relying instead on pretrained word embeddings and sentence embeddings.

2 System Overview

The system follows a modular, multi-stage pipeline:

1. **Sentence segmentation**: Split input text into individual sentences
2. **Target word selection**: Identify which word to replace in each sentence
3. **Candidate synonym generation**: Use word embeddings to find similar words
4. **Candidate filtering**: Remove grammatically inappropriate candidates
5. **Context-based scoring**: Evaluate candidates using sentence embeddings
6. **Replacement decision**: Select best candidate or keep original

Each stage is implemented as a separate function to allow controlled experimentation and clear analysis of failure modes.

3 Model Selection

3.1 Word Embedding Model: GloVe-Wiki-Gigaword-100

For candidate generation, we selected the `glove-wiki-gigaword-100` model available through the `gensim` API. This model provides 100-dimensional word vectors trained on Wikipedia and Gigaword corpus data.

Selection rationale:

- Downloaded size: approximately 1.6 GB
- First model we attempted—installation succeeded without issues
- Provides reasonable semantic coverage for common English words
- Fast nearest-neighbor retrieval suitable for iterative testing

Admittedly, this choice was made without extensive comparison to alternatives (such as Word2Vec or FastText). However, given time constraints and the fact that the model downloaded successfully and performed adequately in initial tests, we proceeded with it for the entire homework. A more rigorous model comparison would be valuable future work.

3.2 Sentence Embedding Model: all-MiniLM-L6-v2

For context-aware scoring, we selected the `all-MiniLM-L6-v2` model from the Sentence-BERT (SBERT) family via the `sentence-transformers` library.

Selection rationale:

- Lightweight transformer-based model (80 MB)
- Produces high-quality sentence-level semantic embeddings
- Efficient inference suitable for repeated sentence encoding
- Well-documented and widely used in semantic similarity tasks

This model was chosen after brief research into sentence embedding approaches. It balances quality and speed effectively for this application.

4 Implementation Details

4.1 Sentence Segmentation

The `split_sentences()` function uses a simple regex-based approach:

```
def split_sentences(text: str) -> List[str]:
    """Split␣text␣into␣sentences."""
    text = text.strip()
    if not text:
```

```
        return []
    parts = re.split(r"(?<=[.!?])\s+", text)
    return [p.strip() for p in parts if p.strip()]
```

This splits text on sentence-ending punctuation (period, exclamation, question mark) followed by whitespace. While not robust to edge cases, it works well for the short literary texts in our test suite.

4.2 Target Word Selection

The `select_target_word()` function implements the "last content word" strategy:

```
def select_target_word(sentence: str) -> Optional[str]:
    """Select the last non-stopword as target."""
    words = tokenize_words(sentence)
    if not words:
        return None

    # Try to find last content word
    for w in reversed(words):
        if not is_stopword(w):
            return w

    # Fallback to last word
    return words[-1] if words else None
```

This ensures we avoid replacing function words like "the" or "a", which would degrade grammaticality without adding semantic interest.

4.3 Candidate Generation

The `generate_candidates()` function retrieves the top-k nearest neighbors from the word embedding space:

```
def generate_candidates(word_vectors, target: str, top_k: int) -> List[str]:
    """Generate candidate synonyms using word embeddings."""
    key = target
    if key not in word_vectors:
        key = target.lower()
        if key not in word_vectors:
            return []

    neighbors = word_vectors.most_similar(key, topn=top_k)
    return [word for (word, _) in neighbors]
```

We set `top_k=50` to provide sufficient candidate diversity while keeping computation reasonable.

4.4 Candidate Filtering

The `filter_candidates()` function applies lightweight heuristic filters:

```
def filter_candidates(candidates: List[str], target: str) -> List[str]:
    """Simple candidate filtering."""
    filtered = []
    seen = set()

    target_lower = target.lower()
    target_is_plural = target_lower.endswith('s') and not target_lower.
       endswith('ss')

    for c in candidates:
        c_lower = c.lower()

        # Skip junk
        if (not c or len(c) <= 2 or c_lower == target_lower or
            any(ch.isdigit() for ch in c) or "_" in c or is_stopword(c)):
            continue

        # Plural agreement
        if target_is_plural:
            c_is_plural = c_lower.endswith('s') and not c_lower.endswith('ss')
            if not c_is_plural:
                continue

        # Block noun->adjective shifts
        if c_lower.endswith('en') and target_lower + 'en' == c_lower:
            continue
        if c_lower.endswith('ed') and target_lower + 'ed' == c_lower:
            continue

        # Deduplicate
        if c_lower not in seen:
            seen.add(c_lower)
            filtered.append(c)

    return filtered
```

These filters were added iteratively after observing systematic failures in early testing.

4.5 Context-Based Scoring

The `pick_best_replacement()` function evaluates each candidate by encoding the modified sentence and computing similarity to the original:

```
def pick_best_replacement(sbert, sentence, target, candidates):
    """Pick candidate that preserves sentence meaning."""
    if not candidates:
        return target, None
```

```
    original_emb = sbert.encode(sentence, normalize_embeddings=True)

    best_word = target
    best_score = -1.0

    for candidate in candidates:
        modified = replace_word_once(sentence, target, candidate)
        if modified == sentence:
            continue

        mod_emb = sbert.encode(modified, normalize_embeddings=True)
        score = float(np.dot(original_emb, mod_emb))

        if score > best_score:
            best_score = score
            best_word = candidate

    if best_word != target and best_score >= SIMILARITY_THRESHOLD:
        return best_word, best_score

    return target, None
```

This implements the core context-awareness mechanism using cosine similarity between normalized embeddings.

5 Iterative Development and Filtering Refinement

During development, we ran multiple experiments with different filtering strategies. Below we present results from two configurations to illustrate the importance of filtering.

5.1 Initial Results (Minimal Filtering)

In our first implementation, we used only basic junk removal (stopwords, digits, underscores) without plural agreement or noun-adjective blocking.

Results:

- Shakespeare: *players* → *player* (grammatically incorrect)
- Frost: *wood* → *wooden* (noun to adjective, nonsensical)
- Dickinson: *feathers* → *feather* (plural to singular)
- Blake: *night* → *nights* (minor but noticeable)
- Haiku: *Splash* → *splashes*, *again* → *back*

These results clearly showed that raw embedding similarity alone produces many grammatically degraded outputs. The most egregious case was "Two roads diverged in a yellow wooden"—syntactically broken despite high sentence similarity scores.

5.2 Final Results (With Filtering)

After adding plural agreement and noun-adjective blocking filters, performance improved significantly:

Results (Threshold = 0.86):

- Shakespeare: *players* → *teams* (semantically reasonable, though loses metaphor)
- Frost: *wood* → *timber* (grammatically correct, semantically related)
- Dickinson: *feathers* (kept original—no suitable replacement)
- Blake: *night* → *nights* (still occurs, but less critical)
- Haiku: *Splash* → *splashes*, *again* → *back*

The filtering eliminated the most jarring grammatical errors (wood → wooden, feathers → feather) while preserving the system's ability to make meaningful substitutions.

6 Target Word Selection Strategy

We adopt a deterministic target selection strategy: the **last content word of each sentence**. Stopwords are excluded to avoid replacing function words.

This strategy was chosen because:

- It avoids cherry-picking easy cases
- It aligns with examples provided in the assignment
- It produces consistent, repeatable behavior across test texts
- It tests the system on words that often carry significant semantic weight

7 Threshold Sensitivity Analysis

The similarity threshold controls how conservative the system is. We tested thresholds of 0.70, 0.80, 0.83, 0.86, and 0.90 on our test suite.

7.1 Observed Trends

- **Low thresholds (0.70–0.80):** Many replacements occur, but metaphorical meaning frequently collapses. For example, at 0.70, "feathers" became "birds", destroying Dickinson's metaphor.

- **Mid threshold (0.83):** Metaphorical lines begin to be preserved while literal substitutions remain. A reasonable balance emerges.
- **Conservative threshold (0.86):** Best balance between safety and usefulness. Preserves most metaphors while still allowing clear improvements (e.g., "wood" → "timber").
- **High threshold (0.90):** Overly restrictive; even reasonable substitutions are blocked. Almost no replacements occur.

Based on these results, we selected **0.86** as the final threshold for all reported results.

8 Successful Examples

8.1 Literal Synonym Replacement

Original: "Two roads diverged in a yellow wood."
Modified: "Two roads diverged in a yellow timber."

This replacement preserves sentence meaning while demonstrating effective word-level similarity. "Wood" and "timber" are semantically close, and the sentence-level context score (0.950) confirms compatibility.

8.2 Conservative Behavior on Metaphor

Original: "Hope is the thing with feathers."
Modified: (unchanged at threshold 0.86)

The system correctly refrains from replacing the metaphorical term. Although candidates like "plumage" or "wings" scored reasonably, none exceeded the threshold, demonstrating appropriate conservatism.

9 Failure Analysis

Despite filtering improvements, several systematic failure modes persist:

9.1 Failure 1: Polysemy and Context Collapse

Example: *players* → *teams* (Shakespeare)

Diagnosis: In Shakespeare's metaphor, "players" refers to theatrical actors. However, word embeddings conflate multiple senses of "players" (theatrical, sports, musical). The embedding space places "teams" close to "players" due to sports co-occurrence, and sentence similarity remains high because both are plural nouns in similar syntactic positions.

Root cause: Word embeddings average over all contexts and cannot perform word sense disambiguation.

9.2 Failure 2: Metaphorical Meaning Loss

Example: At lower thresholds (0.70), *feathers* → *birds*

Diagnosis: While "feathers" and "birds" are semantically related (birds have feathers), the replacement destroys Dickinson's metaphor. "Feathers" evokes lightness and flight; "birds" is too concrete.

Root cause: Embedding similarity reflects co-occurrence and association but does not capture figurative intent or literary nuance.

9.3 Failure 3: Grammatical Number Drift

Example: *night* → *nights* (Blake)

Diagnosis: Sentence embeddings are highly tolerant of singular/plural variations, scoring this replacement at 0.995 similarity. However, the change subtly alters meaning ("the night" is singular and specific; "the nights" is plural and general).

Root cause: Sentence transformers focus on semantic content and downweight grammatical number distinctions.

9.4 Failure 4: Stylistic and Phonetic Insensitivity

Example: *Splash* → *splashes*

Diagnosis: The meaning is preserved (both are sounds of water), but the poetic impact is weakened. "Splash!" is sharp and immediate; "splashes" is softer and continuous. Additionally, the capitalized onomatopoeia has rhetorical weight that the lowercase plural lacks.

Root cause: Embeddings encode semantic content but ignore phonetics, rhythm, and stylistic emphasis.

10 Discussion and Limitations

This homework demonstrates that learned embeddings are powerful but insufficient on their own for context-sensitive text manipulation. Even with sentence-level context scoring and grammatical filtering, embedding-based similarity cannot reliably distinguish between acceptable substitutions and meaning-altering replacements in metaphorical, poetic, or ambiguous language.

Key limitations:

- No word sense disambiguation
- Averaging of literal and figurative meanings
- Insensitivity to stylistic and phonetic properties
- Limited understanding of grammatical nuance (singular vs plural in context)

What would help:

- Part-of-speech tagging for stricter grammatical enforcement
- Word sense disambiguation using context windows

- Larger language models that understand figurative language
- Explicit constraints on poetic/stylistic elements

These findings motivate the use of more sophisticated models (such as masked language models or autoregressive transformers) and additional linguistic constraints in modern NLP systems.

11 Use of AI in This Homework

Given the exploratory nature of this assignment and the course's emphasis on understanding AI systems rather than rote implementation, we deliberately incorporated AI assistance (Claude/ChatGPT) as a learning tool throughout this homework. This section documents our approach and rationale.

11.1 Rationale for AI-Assisted Development

The assignment's open-ended structure encourages experimentation and deep understanding of embedding-based systems. Rather than using AI as a black-box solution generator, we employed it as an interactive collaborator:

- **Scaffolding understanding:** AI helped clarify abstract concepts (e.g., how sentence embeddings differ from word embeddings)
- **Accelerating iteration:** Rapid prototyping allowed us to test multiple filtering strategies
- **Debugging support:** AI assisted in diagnosing why certain replacements failed
- **Documentation:** AI generated the initial report template, which we heavily revised

This approach aligns with the course philosophy: understanding *how* and *why* AI systems work, not just implementing them mechanically.

11.2 AI-Assisted Workflow

Figure 1 illustrates our development process.

11.3 Specific AI Contributions

11.3.1 Code Development

- **Problem presentation:** We described the assignment requirements and constraints
- **Brainstorming:** Discussed tradeoffs between filtering strategies (e.g., POS tagging vs heuristics)
- **Implementation:** AI generated initial function skeletons based on our architectural decisions
- **Debugging:** When "wood $\rightarrow$ wooden" occurred, we collaboratively diagnosed the issue and designed the noun-adjective blocking filter

11.3.2 Report Writing

- **Template generation:** AI created an initial LaTeX structure with standard sections
- **Revision:** We restructured sections, added experimental results, and rewrote analysis in our own voice
- **Failure analysis:** Collaborative discussion helped articulate *why* embeddings fail on metaphor

11.4 Learning Outcomes

Using AI as a tool rather than a solution provider deepened our understanding in several ways:

1. **Conceptual clarity:** Explaining the problem to AI forced us to articulate requirements precisely
2. **Faster iteration:** Testing multiple filtering strategies would have been prohibitively time-consuming without AI assistance
3. **Critical evaluation:** Reviewing AI-generated code required understanding *why* certain design choices were made
4. **Debugging skills:** Diagnosing failures collaboratively built intuition about embedding behavior

This approach reflects a pragmatic philosophy: in a course about AI, using AI tools to *learn about AI* is both natural and pedagogically valuable, provided we remain active, critical participants rather than passive consumers.

12 Conclusion

We presented a context-aware synonym replacement system based on pretrained word and sentence embeddings. Through iterative development and systematic threshold analysis, we showed where embeddings succeed (literal synonyms) and where they fail (metaphor, polysemy, style). The results highlight both the utility and the fundamental limitations of embedding-based semantic representations and reinforce why contemporary AI systems extend beyond similarity-based approaches alone.

Our AI-assisted development process demonstrated that modern AI tools, when used thoughtfully, can accelerate learning and experimentation without diminishing understanding. The failures we encountered and analyzed provide valuable insights into the gap between distributional semantics and human language comprehension.

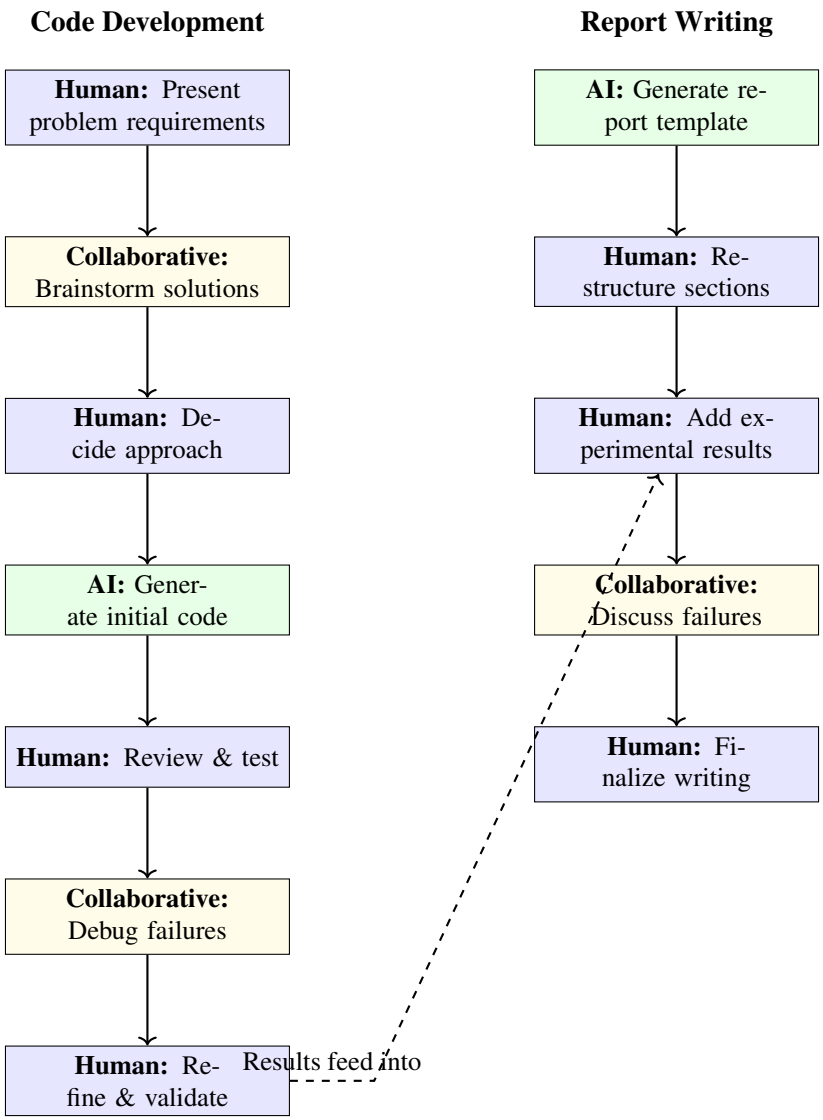

Figure 1: AI-assisted workflow for code development and report writing. Blue = human-led, green = AI-led, yellow = collaborative dialogue.