Naïve Semantic Text Similarity Model

Zachary Parent

UPC

November 30, 2024

Outline

Introduction

- Semantic Text Similarity (STS) is crucial for many NLP tasks
- Challenge: Which features best capture semantic similarity?
- Our approach: Unbiased feature analysis using Random Forests

Methodology

- Approach
- Feature extraction
- Feature selection
- Model training
- Model evaluation

Approach

- Naïve approach which requires no knowledge of the corpus
- Use categorized steps to process sentences in every permutation
  - 521 permutations
  - e.g. sentence_to_doc, chunk_NEs, remove_stopwords, lemmatize_tokens, get_characters, get_2grams
- Apply 4 similarity metrics to each permutation
- Used Random Forest's feature importance capabilities
- Let the data guide feature selection

Feature Extraction

```python
# Generate all valid permutations of sentence processing steps
def generate_valid_permutations(
    functions: List[Callable] = all_functions,
) -> List[Tuple[Callable, ...]]:
    valid_permutations = []
    for n in range(1, len(functions) + 1):
        for perm in itertools.permutations(functions, n):
            if _is_valid_permutation(perm):
                valid_permutations.append(perm)
    # Add sentence_to_doc to the beginning of each permutation
    valid_permutations = (
        [tuple([sentence_to_doc]) + perm for perm in valid_permutations])
    # Add final step to each permutation (e.g. get_2grams)
    valid_permutations = (
        [new_perm for perm in valid_permutations for new_perm in add_final_step(perm)])
    return valid_permutations
```

Feature Extraction

```
# Dictionary to hold function names and their input/output types
function_input_output_types: Dict[str, Tuple[Tuple[type, ...], type]] = {}

# Populate the dictionary with function names and their input/output types
for func in all_functions:
    input_types, output_type = _extract_input_output_types(func)
    function_input_output_types[func.__name__] = (input_types, output_type)


# Function to check if a permutation is valid based on input/output types
def _is_valid_permutation(perm: Tuple[Callable]) -> bool:
    if function_input_output_types[perm[0].__name__][0] != spacy.tokens.doc.Doc:
        return False
    if function_input_output_types[perm[-1].__name__][1] not in [
        Tuple[Word, ...],
        Tuple[PosTag, ...],
        Tuple[Character, ...],
    ]:
        return False
    for i in range(len(perm) - 1):
        _, current_func_output_type = function_input_output_types[perm[i].__name__]
        next_func_input_type, _ = function_input_output_types[perm[i + 1].__name__]
        if current_func_output_type != next_func_input_type:
            return False
    return True
```

Feature Extraction

```
class PosTag(str): pass
class Word(str): pass
class Character(str): pass
class Ngram(Tuple[Word | Character | PosTag, ...]): pass
class WordPair(Tuple[Word, Word]): pass

def get_characters(words: Tuple[Word, ...]) -> Tuple[Character, ...]:
def get_word_pairs(words: Tuple[Word, ...]) -> Tuple[WordPair, ...]:
def sentence_to_doc(sentence: str) -> spacy.tokens.doc.Doc:
def get_tokens(doc: spacy.tokens.doc.Doc) -> Tuple[spacy.tokens.token.Token, ...]:
def get_pos_tags(tokens: Tuple[spacy.tokens.token.Token, ...]) -> Tuple[PosTag, ...]:
def lemmatize_tokens(tokens: Tuple[spacy.tokens.token.Token, ...]) -> Tuple[Word, ...]:
def get_token_text(tokens: Tuple[spacy.tokens.token.Token, ...]) -> Tuple[Word, ...]:
def get_2grams(words: Tuple[Word | Character | PosTag, ...]) -> Tuple[Ngram, ...]:
def get_3grams(words: Tuple[Word | Character | PosTag, ...]) -> Tuple[Ngram, ...]:
def get_4grams(words: Tuple[Word | Character | PosTag, ...]) -> Tuple[Ngram, ...]:
def chunk_NEs(doc: spacy.tokens.doc.Doc) -> Tuple[spacy.tokens.token.Token, ...]:
def get_synsets(tokens: Tuple[spacy.tokens.token.Token, ...]) -> Tuple[Word, ...]:
def remove_non_alnum(words: Tuple[Word, ...]) -> Tuple[Word, ...]:
def remove_stopwords(tokens: Tuple[spacy.tokens.token.Token, ...])
    -> Tuple[spacy.tokens.token.Token, ...]:
def get_stopwords(tokens: Tuple[spacy.tokens.token.Token, ...])
    -> Tuple[spacy.tokens.token.Token, ...]:
```

Feature Extraction

```
lexical_functions = [
    get_characters,        # Character-level patterns
    get_tokens,            # Word tokenization
    get_token_text,        # Raw word forms
    remove_non_alnum,      # Character filtering
    get_word_pairs,        # Word co-occurrences
]
semantic_functions = [
    lemmatize_tokens,      # Normalize to base meaning
    get_synsets,           # Word meanings/concepts
    chunk_NEs,             # Named entity grouping
    get_pos_tags,          # Part of speech (bridges lexical/semantic)
]
ngram_functions = [
    get_2grams,            # Bigrams
    get_3grams,            # Trigrams
    get_4grams,            # 4-grams
]
preprocessing_functions = [
    remove_stopwords,      # Filter non-content words
    get_stopwords,         # Identify non-content words
]
```

Top Features

- Jaccard similarity dominates (7 of top 10)
- Common pipeline steps: lemmatization, stopwords, n-grams
- Top feature accounts for 20% importance

Feature Pipeline	Importance
score_jaccard_165	0.197
score_cosine_257	0.089
score_cosine_165	0.069
score_jaccard_258	0.033
score_cosine_258	0.022

Figure: Top 5 Features by Importance

Conclusions

- Simple features can be highly effective
- Pipeline complexity isn't always better
- Character-level analysis with n-grams shows promise