Word Representation in Biomedical Domain

Report of Group 3

1 Methodology and Underlying Principles

1.1 Parse the Data

Initially, we performed data extraction. Given the large scale of the COVID-19 dataset, we focused on extracting only the titles and abstracts of the papers to form our dataset. The specific extraction process is outlined in the code below.

```
import csv

# Open and read the metadata.csv file
with open(r"D:\DOWNLOAD\wetransfer_winter-school-nlp-2025_2025-02-03_1458\all_sources_metadata_2020-03-13.csv", encoding='utf-8') as f_in:
    reader = csv.DictReader(f_in)

    # Open a txt file for writing
    with open('titles_abstracts.txt', 'w', encoding='utf-8') as f_out:
        for row in reader:
            title = row['title'].strip() if row['title'] else "-"
            abstract = row['abstract'].strip() if row['abstract'] else "-"

            # Combine title and abstract
            text = f"{title}\n{abstract}\n\n"

            # Write to file
            f_out.write(text)
```

1.2 Tokenization

Our tokenization process is implemented in four methods, as described below:

1. Use split()
2. Use NLTK or SciSpaCy
3. Use Byte-Pair Encoding (BPE)
4. Build custom Byte-Pair Encoding (BPE)

1.2.1 Use split()

In this approach, we perform tokenization by using the simple split() method together with regex, which splits the text based on spaces and newline characters. The implementation is as follows:

```
import re
import random
from collections import Counter

def Tokenization_1(content):
    filename = './titles_abstracts.txt'
    with open(filename, 'r', encoding='utf-8') as file:
        content = file.read()
        words = []
```

```
        lines = content.split("\n")
    for line in lines:
        words.extend(re.split(r"\W+", line))  # Use regular expressions to split text
    # Filter out empty strings to ensure only valid words are extracted
    words = [word for word in words if word]

    # Use the Counter class to count word occurrences and perform weighted sampling
    word_counts = Counter(words)
    weighted_sample = random.choices(
        list(word_counts.keys()), weights=word_counts.values(), k=20
    )
    print("Weighted sample:", "-".join(weighted_sample))

    result = "-".join(words)
    return result
```

1.2.2 Use NLTK

In this section, we use NLTK [1] for tokenization. NLTK is a library optimized for processing scientific texts. Its tokenization approach is tailored to handle the specific challenges posed by scientific and technical language. Below are the principles underlying NLTK's tokenization process:

- **Rule-Based Tokenization**: Utilizes predefined rules to handle punctuation, abbreviations, and special symbols common in scientific writing.
- **Regular Expressions**: Implements regex-based patterns to split text efficiently while preserving meaningful units.
- **Treebank Tokenizer**: Uses patterns derived from the Penn Treebank dataset to ensure consistency in breaking down complex sentences.
- **Word and Sentence Segmentation**: Differentiates between sentence boundaries and word boundaries, ensuring accurate parsing of multi-sentence documents.
- **Handling Special Cases**: Accounts for numbers, dates, chemical formulas, and technical notation, preventing incorrect segmentation.

The details of the implementation are as follows.

```
import nltk
from nltk import word_tokenize, sent_tokenize

def Tokenization_2(content):
    # Sentence segmentation
    sents = sent_tokenize(content)
    interpunctuations = [",", ".", ":", ";", "?", "(", ")", "[", "]", "&", "!", "*", "@", "#", "$", "%", "/", "\\", "    "]

    words = []
    for sent in sents:
        sent_t = word_tokenize(sent)  # Tokenization
        sent_t = [word.lower() for word in sent_t if word not in interpunctuations]  # Remove punctuation marks
        words.append(sent_t)

    # Save tokenization results
    result = ["-".join(sent) for sent in words]
```

```
    with open("./nlkt_tokenized.txt", "w", encoding="utf-8") as f:
        f.write("\n".join(result))

    return result
```

1.2.3 Use Byte-Pair Encoding (BPE)

In this section we use Byte-Pair Encoding to tokenize. Byte-Pair Encoding (BPE) is a tokenization method that iteratively merges the most frequent adjacent character or subword pairs in a text until a predefined vocabulary size is reached. It starts by splitting words into individual characters and counting the frequency of adjacent pairs. The most common pair is merged into a new subword, and this process repeats, gradually forming longer tokens. By doing this, BPE efficiently reduces the size of the vocabulary while preserving the ability to handle rare words by breaking them into familiar subunits. This makes it especially useful for natural language processing, as it balances flexibility and efficiency in text representation.

The implementation is as follows:

```
from transformers import AutoTokenizer

def Tokenization_3(content):
    bert_tokenizer = AutoTokenizer.from_pretrained("bert-base-
        uncased")

    # Split text into paragraphs to avoid excessively long inputs
    max_length = 512
    result = []

    for i in range(0, len(content), max_length):
        chunk = content[i : i + max_length]
        tokens = bert_tokenizer.tokenize(chunk)
        result.append(" ".join(tokens))  # Join tokens with spaces

    # Save tokenization results
    with open("bert_tokenized.txt", "w", encoding="utf-8") as f:
        f.write("\n".join(result))

    return result
```

1.2.4 Build custom Byte-Pair Encoding (BPE)

In this section we use build custom Byte-Pair Encoding (BPE). Custom Byte-Pair Encoding (BPE) follows the same iterative merging process as standard BPE but allows modifications for specific tasks or datasets. It starts with character-level tokenization, repeatedly merges the most frequent adjacent pairs into subwords, and continues until a target vocabulary size is reached. Custom implementations can adjust merge rules, handle special characters differently, or optimize tokenization for domain-specific applications, enhancing adaptability while preserving the efficiency of BPE in reducing vocabulary size and handling rare words.

The implementation is as follows:

```
from tokenizers import Tokenizer, models, trainers, pre_tokenizers

def Tokenization_4(content):
    # Train the BPE model
    print("Training BPE tokenizer...")
    bpe_tokenizer = Tokenizer(models.BPE())
    bpe_tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()

```

```
    trainer = trainers.BpeTrainer(special_tokens=["<unk>", "<pad>",
        "<s>", "</s>"])
    bpe_tokenizer.train(files=["titles_abstracts.txt"], trainer=
        trainer)

    # Save the trained BPE model
    bpe_tokenizer.save("biomedical_bpe.json")

    # Reload the BPE tokenizer and perform tokenization
    bpe_tokenizer = Tokenizer.from_file("biomedical_bpe.json")
    bpe_tokens = bpe_tokenizer.encode(content)

    # Save BPE tokenization results
    with open("bpe_tokenized.txt", "w", encoding="utf-8") as f:
        f.write(" ".join(bpe_tokens.tokens))  # Join tokens with
            spaces

    return bpe_tokens
```

1.3 Build Word Representations

1.3.1 Use N-gram Language Modeling

N-gram language modeling [2] represents words using statistical cooccurrence. Given a sequence $w_1, w_2, \ldots, w_t$, the probability of a word based on the previous $n-1$ words is:

$$P(w_t|w_{t-(n-1)}, \ldots, w_{t-1}) = \frac{C(w_{t-(n-1)}, \ldots, w_t)}{C(w_{t-(n-1)}, \ldots, w_{t-1})}$$

Embeddings are learned by constructing a co-occurrence matrix M, where:

$$M_{ij} = P(w_j|w_i)$$

Embeddings are obtained by singular value decomposition (SVD) of M:

$$M = USV^T$$

where U, V are low-dimensional word embeddings that capture semantic relationships. U captures the embedding of words by focusing on their co-occurrence patterns across contexts (rows of M), while V captures the embedding of contexts by focusing on their co-occurrence patterns with words (columns of M).

The details of the implementation are as follows.

```
import collections
from collections import defaultdict, Counter
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import svds
import numpy as np

class NGramLanguageModel:
    def __init__(self, n, vocab_size=10000, embedding_dim=100):
        self.n = n
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim

        # Vocabulary
        self.vocab = {'<UNK>', '<START>', '<END>'}
        self.word_to_id = {}
        self.id_to_word = {}

```

```
        # N-gram statistics
        self.ngram_counts = defaultdict(lambda: defaultdict(float))
        self.context_counts = defaultdict(float)

        # Word embeddings
        self.word_embeddings = None
        self.context_embeddings = None

        # Smoothing parameter
        self.alpha = 0.1

    def build_vocab(self, text):
        """Build vocabulary from text"""
        words = text.split()
        word_counts = Counter(words)
        top_words = sorted(word_counts.items(), key=lambda x: x[1],
            reverse=True)
        top_words = top_words[:self.vocab_size - 3]  # Reserve space
             for special tokens

        self.vocab.update(word for word, _ in top_words)
        self.word_to_id = {word: idx for idx, word in enumerate(self
            .vocab)}
        self.id_to_word = {idx: word for word, idx in self.
            word_to_id.items()}

    def build_cooccurrence_matrix(self, text):
        """Build co-occurrence matrix"""
        words = ['<START>'] + text.split() + ['<END>']
        vocab_size = len(self.vocab)

        # Initialize sparse matrix components
        rows, cols, data = [], [], []

        # Compute co-occurrence
        for i in range(len(words) - self.n + 1):
            context_words = words[i:i + self.n - 1]
            target_word = words[i + self.n - 1]

            # Get word IDs
            context_ids = [self.word_to_id.get(w, self.word_to_id['<
                UNK>'])
                           for w in context_words]
            target_id = self.word_to_id.get(target_word, self.
                word_to_id['<UNK>'])

            # Update n-gram statistics
            context = tuple(context_ids)
            self.ngram_counts[context][target_id] += 1
            self.context_counts[context] += 1

            # Update co-occurrence counts
            for context_id in context_ids:
                rows.append(context_id)
                cols.append(target_id)
                data.append(1.0)

        # Construct sparse matrix
```

```
        M = csr_matrix((data, (rows, cols)), shape=(vocab_size,
            vocab_size))

        # Compute conditional probability P(wj|wi)
        row_sums = M.sum(axis=1).A.flatten()
        row_sums[row_sums == 0] = 1  # Avoid division by zero
        M = csr_matrix(M / row_sums[:, np.newaxis])

        return M

    def factorize_matrix(self, M):
        """Perform SVD on the co-occurrence matrix"""
        U, S, Vt = svds(M, k=self.embedding_dim)

        # Scale singular values
        S_sqrt = np.sqrt(S)
        self.word_embeddings = U * S_sqrt  # Scale each column of U
        self.context_embeddings = Vt.T * S_sqrt  # Scale each row of
            Vt.T

    def train(self, text):
        """Train the model"""
        self.build_vocab(text)
        M = self.build_cooccurrence_matrix(text)
        self.factorize_matrix(M)

    def get_word_vector(self, word):
        """Retrieve word vector"""
        word_id = self.word_to_id.get(word, self.word_to_id['<UNK>'
            ])
        return self.word_embeddings[word_id]

    def get_context_vector(self, context):
        """Retrieve context vector"""
        context_words = context.split()[-self.n + 1:]
        context_ids = [self.word_to_id.get(w, self.word_to_id['<UNK>
            '])
                       for w in context_words]
        return np.mean([self.context_embeddings[i] for i in
            context_ids], axis=0)

    def predict_next(self, context):
        """Predict the next word"""
        context_vec = self.get_context_vector(context)

        # Compute similarity with all words
        similarities = np.dot(self.word_embeddings, context_vec)

        # Find the most similar word
        best_id = np.argmax(similarities)
        best_word = self.id_to_word[best_id]

        # Compute probability (using softmax)
        exp_similarities = np.exp(similarities - np.max(similarities
            ))
        probability = exp_similarities[best_id] / exp_similarities.
            sum()

```

```
        return best_word, float(probability)

    def generate(self, seed, length=10):
        """Generate text"""
        current = ['<START>'] + seed.split()
        result = current.copy()

        for _ in range(length):
            context = '-'.join(current[-(self.n - 1):])
            next_word, _ = self.predict_next(context)

            if next_word == '<END>':
                break

            result.append(next_word)
            current = result[-self.n + 1:]

        return '-'.join(result[1:])  # Remove <START> token

    def get_similar_words(self, word, k=5):
        """Find the top-k similar words"""
        word_vec = self.get_word_vector(word)
        similarities = np.dot(self.word_embeddings, word_vec)
        top_k = np.argsort(similarities)[-k-1:-1][::-1]  # Exclude
            the word itself
        return [(self.id_to_word[i], similarities[i]) for i in top_k
            ]

def Representations_1(result):
    # Tokenized text input
    tokenized_text = result

    model = NGramLanguageModel(3)
    model.train(tokenized_text)

    # Output word vectors
    for word, vector in model.get_word_vector().items():
        print(f"Word: {word}, Vector: {vector[:5]}...")  # Display
            first 5 dimensions

    return model
```

1.3.2 Use Skip-gram with Negative Sampling

Skip-gram with Negative Sampling (SGNS) [3] is an efficient method of learning word embeddings by training a neural network to predict context words given a target word, while also distinguishing real context words from randomly sampled negative words.

Given a word sequence $w_1, w_2, \ldots, w_T$, the skip-gram model maximizes the probability of context words w_c within a window around a target word w_t:

$$P(w_c|w_t) = \sigma(v_c^T v_t)$$

where v_t and v_c are the embeddings of the target and context words, and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

Negative sampling replaces the full softmax with a binary classification task, optimizing:
$\log \sigma(v_c^T v_t) + \sum_{n=1}^{k} \mathbb{E}_{w_n \sim P_n} \log \sigma(-v_n^T v_t)$
where k negative samples w_n are drawn from a noise distribution. This approach significantly reduces computational cost while preserving meaningful semantic structure in the learned embeddings.

The details of our implementation are as follows:

```
import numpy as np
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess

class SkipGramWithNegativeSampling:
    def __init__(self, vocab_size, vector_dim, window_size, negative_samples):
        self.vocab_size = vocab_size
        self.vector_dim = vector_dim
        self.window_size = window_size
        self.negative_samples = negative_samples
        # Initialize word vectors randomly
        self.word_vectors = np.random.uniform(-0.5, 0.5, (vocab_size, vector_dim))
        self.context_vectors = np.random.uniform(-0.5, 0.5, (vocab_size, vector_dim))

    def _sigmoid(self, x):
        # Activation function
        return 1 / (1 + np.exp(-x))

    def _loss(self, central_vec, context_vec, negative_vecs):
        # Loss function
        positive_score = np.dot(context_vec, central_vec)
        positive_loss = -np.log(self._sigmoid(positive_score))
        negative_scores = np.dot(negative_vecs, central_vec)
        negative_loss = -np.sum(np.log(self._sigmoid(-negative_scores)))
        return positive_loss + negative_loss

    def _gradient_update(self, central_vec, context_vec, negative_vecs, learning_rate):
        # Positive sample gradient
        positive_score = self._sigmoid(np.dot(context_vec, central_vec))
        positive_grad = (positive_score - 1) * context_vec
        # Negative sample gradient
        negative_scores = self._sigmoid(np.dot(negative_vecs, central_vec))
        negative_grad = np.sum(negative_scores[:, np.newaxis] * negative_vecs, axis=0)
        # Update vectors
        central_vec -= learning_rate * (positive_grad + negative_grad)
        context_vec -= learning_rate * (positive_score - 1) * central_vec
        for neg_vec in negative_vecs:
            neg_vec -= learning_rate * negative_scores * central_vec

    def train(self, dataset, epochs, learning_rate):
        # Train model using gradient descent
        for epoch in range(epochs):
            total_loss = 0
            for sentence in dataset:
                for i, central_word in enumerate(sentence):
                    # Set window
```

```
                    start = max(0, i - self.window_size)
                    end = min(len(sentence), i + self.window_size +
                        1)
                    # Get sentence
                    context_words = [sentence[j] for j in range(
                        start, end) if j != i]
                    for context_word in context_words:
                        # Positive sample
                        central_vec = self.word_vectors[central_word
                            ]
                        context_vec = self.context_vectors[
                            context_word]
                        # Negative samples
                        negative_indices = random.sample(
                            [idx for idx in range(self.vocab_size)
                                if idx not in sentence],
                            self.negative_samples,
                        )
                        negative_vecs = self.context_vectors[
                            negative_indices]
                        # Calculate loss function
                        loss = self._loss(central_vec, context_vec,
                            negative_vecs)
                        total_loss += loss
                        # Update gradients
                        self._gradient_update(central_vec,
                            context_vec, negative_vecs, learning_rate
                            )
            print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss}")

    def get_word_vectors(self):
        # Get word vectors
        return self.word_vectors

def Representations_2(result):
    # Preprocess sentences
    tokens = [sent.split(' ') for sent in result]
    # Set model parameters
    model = Word2Vec(
        sentences=tokens[:20_000],  # Input preprocessed sentences
        vector_size=256,            # Embedding vector dimension
        window=5,                   # Window size (consider up to 5
            words before and after)
        sg=1,                       # Skip-gram mode (sg=1 for Skip-
            gram; sg=0 for CBOW)
        negative=10,                # Number of negative samples (
            number of negative samples per positive sample)
        min_count=2,                # Ignore words that appear less
            than 2 times
        workers=4,                  # Number of CPU cores to use
        epochs=10                   # Number of training iterations
    )
    # Output word vectors
    vocab = model.wv.index_to_key
    for word in vocab:
        vector = model.wv[word]
        print(f"Word: {word}, Vector: {vector[:5]}...")  # Only
            display first 5 dimensions
```

```
    return model
```

1.3.3 Use Contextualised Word Representation by Masked Language Model (MLM)

Masked Language Model (MLM) [4] is a technique used to learn **contextualized word representations** by training a model to predict randomly masked words in a sentence based on their surrounding context. This approach enables embeddings to capture dynamic word meanings depending on context, unlike static word embeddings (e.g. Word2Vec).

Given an input sequence $w_1, w_2, \ldots, w_T$, some tokens are randomly masked (e.g. w_m), and the model learns to predict them:

$$P(w_m | w_1, \ldots, w_{m-1}, \texttt{[MASK]}, w_{m+1}, \ldots, w_T)$$

A transformer-based model (e.g., BERT) processes the full sequence bidirectionally, generating deep contextualized word embeddings. The loss function is typically the **cross-entropy loss** over the masked tokens:

$$\mathcal{L} = -\sum_{m} \log P(w_m | \mathbf{h}_m)$$

where $\mathbf{h}_m$ is the hidden representation of the masked token. This method enables embeddings to capture word sense variations and syntactic dependencies based on the given context.

In our implementation, we utilize BERT for embedding generation. Given the large number of parameters in BERT and the high computational cost of training, we employ LoRA [5] fine-tuning to optimize the training process. The details of our implementation are as follows:

```python
from transformers import BertTokenizer, BertForMaskedLM, Trainer, TrainingArguments
from datasets import Dataset
import torch
from transformers import DataCollatorForLanguageModeling
from peft import get_peft_model, LoraConfig, TaskType

def Representations_3():
    # Load BERT (Masked Language Model version)
    tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
    model = BertForMaskedLM.from_pretrained("bert-base-uncased")
    # Configure LoRA
    lora_config = LoraConfig(
        r=4,  # Low-rank adaptation dimension
        lora_alpha=32,  # LoRA alpha parameter
        lora_dropout=0.1,  # Dropout probability
        target_modules=["query", "value"]  # Adapt transformer's Q/V matrices
    )
    model = get_peft_model(model, lora_config)
    total_params = sum(p.numel() for p in model.parameters())  # All parameters
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)  # Only trainable parameters
    print(f"Total model parameters: {total_params:,}")
    print(f"Trainable parameters: {trainable_params:,}")
    print(f"Frozen parameters: {total_params - trainable_params:,}")
    # Read text data
    with open("titles_abstracts.txt", "r", encoding="utf-8") as f:
        lines = [line.strip() for line in f.readlines() if line.strip()]
    # Create Hugging Face dataset
    dataset = Dataset.from_dict({"text": lines})
```

```
    # Split dataset
    dataset = dataset.train_test_split(test_size=0.1)
    train_dataset = dataset["train"]
    eval_dataset = dataset["test"]
    # Preprocessing: Tokenization + MLM masking
    def tokenize_function(examples):
        return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=256)
    train_dataset = train_dataset.map(tokenize_function, batched=True)
    eval_dataset = eval_dataset.map(tokenize_function, batched=True)
    # Data collator
    data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer,
        mlm=True,  # Perform MLM training
        mlm_probability=0.15  # 15% of tokens are masked
    )
    # Training arguments
    training_args = TrainingArguments(
        output_dir="./bert_lora_finetuned",
        per_device_train_batch_size=16,
        per_device_eval_batch_size=16,
        num_train_epochs=3,
        save_strategy="epoch",
        evaluation_strategy="epoch",
        logging_dir="./logs",
        logging_steps=500,
    )
    # Trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        data_collator=data_collator,
    )
    # Start training
    trainer.train()
    # Set to evaluation mode
    model.eval()
    # Read text and perform tokenization
    with open("titles_abstracts.txt", "r", encoding="utf-8") as f:
        lines = f.readlines()[:500]
    # Process line by line
    all_embeddings = []
    for line in lines:
        inputs = tokenizer(line.strip(), return_tensors="pt", truncation=True, padding="max_length", max_length=256)
        with torch.no_grad():
            outputs = model(inputs)  # Direct BERT call
        embeddings = outputs.logits  # Extract token predictions
        all_embeddings.append(embeddings.squeeze(0))
    # Print information
    print(f"Total embeddings: {len(all_embeddings)}")
    print(f"Shape of single embedding: {all_embeddings[0].shape}")
    # Save token embeddings
    torch.save(all_embeddings, "bert_lora_finetuned_embeddings.pt")
    print("LoRA fine-tuned token embeddings have been saved!")
```

```
    return all_embeddings
```

1.4 Explore the Word Representations

1.4.1 Visualise the word representations by t-SNE

In this section, we employ t-SNE to transform high-dimensional embeddings into a 2D visual representation, enabling effective embedding visualization.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from random import sample

kimi_BME={
    "viruses": ["virus", "coronavirus", "covid-19", "influenza", "2019-ncov", "sars-cov-2", "h5n1", "mers-cov", "norovirus", "hiv-1", "h1n1"],
    "diseases": ["infection", "disease", "pneumonia", "syndrome", "mortality", "cancer", "pathogen", "illness"],
    "proteins": ["protein", "proteins", "antibodies", "spike", "glycoprotein", "capsid", "membrane", "antigen", "antigens", "epitopes"],
    "genes": ["gene", "genes", "genome", "rna", "dna", "receptor", "genetic", "mutation", "mutations"],
    "cells": ["cells", "cell", "host", "cellular", "blood", "serum", "tissue", "antibody", "antibodies"],
    "treatments": ["treatment", "therapy", "vaccine", "vaccines", "antiviral", "care", "diagnosis", "diagnostic", "drug", "drugs"],
    "epidemiology": ["outbreak", "epidemic", "pandemic", "transmission", "quarantine", "prevention", "public", "surveillance", "population", "spread"],
    "research": ["study", "analysis", "data", "research", "model", "models", "sequencing", "clinical", "review", "findings"],
    "symptoms": ["symptoms", "fever", "cough", "dyspnea", "fatigue", "headache", "sore-throat", "shortness-of-breath"],
    "methods": ["methods", "assay", "pcr", "rt-pcr", "sequencing", "imaging", "tools", "test", "tests", "diagnostic"],
    "public_health": ["health", "public", "hospital", "hospitals", "healthcare", "prevention", "quarantine", "epidemiology", "surveillance"],
    "miscellaneous": ["cases", "patients", "human", "novel", "background", "severity", "risk", "control", "management", "response"]
}

def Exploration_1(model, BME=kimi_BME, max_words=1000, seed=1022):
    random.seed(seed)
    # Extract vocabulary and corresponding word vectors
    BMEs = [word for words in list(BME.values()) for word in words]
    vocab = model.wv.index_to_key
    vocab_1 = list(filter(lambda x: x in BMEs, vocab))
    vocab_2 = sample(vocab, max_words-len(vocab_1))

    vocab = vocab_1 + vocab_2  # Extract vocabulary
    word_vectors = model.wv[vocab]  # Extract word vectors

```

```
    # Limit the maximum number of words for visualization
    if len(vocab) > max_words:
        np.random.seed(seed)
        indices = np.random.choice(len(vocab), max_words, replace=
            False)
        vocab = [vocab[i] for i in indices]
        word_vectors = word_vectors[indices]

    # Use t-SNE for dimensionality reduction to 2D
    tsne = TSNE(n_components=2, random_state=seed, perplexity=75,
        init='pca', n_iter=1000)
    reduced_embeddings = tsne.fit_transform(word_vectors)

    # Plot t-SNE visualization
    plt.figure(figsize=(14, 10))
    for i, word in enumerate(vocab):
        x, y = reduced_embeddings[i]
        plt.scatter(x, y, color='blue', s=10)
        plt.text(x + 0.2, y + 0.2, word, fontsize=8, color='gray')
    plt.title("t-SNE Visualization of Word Embeddings")
    plt.show()

    return None
```

1.4.2 Visualise the Word Representations of Biomedical Entities by t-SNE

In this section, we selected the biomedical entities from the tokenized data, assigned colors based on their categories, and visualized them using t-SNE to observe their distribution in the high-dimensional space.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from random import sample

kimi_BME={
    "viruses": ["virus", "coronavirus", "covid-19", "influenza", "
        2019-ncov", "sars-cov-2", "h5n1", "mers-cov", "norovirus", "
        hiv-1", "h1n1"],
    "diseases": ["infection", "disease", "pneumonia", "syndrome", "
        mortality", "cancer", "pathogen", "illness"],
    "proteins": ["protein", "proteins", "antibodies", "spike", "
        glycoprotein", "capsid", "membrane", "antigen", "antigens", "
        epitopes"],
    "genes": ["gene", "genes", "genome", "rna", "dna", "receptor", "
        genetic", "mutation", "mutations"],
    "cells": ["cells", "cell", "host", "cellular", "blood", "serum",
         "tissue", "antibody", "antibodies"],
    "treatments": ["treatment", "therapy", "vaccine", "vaccines", "
        antiviral", "care", "diagnosis", "diagnostic", "drug", "drugs
        "],
    "epidemiology": ["outbreak", "epidemic", "pandemic", "
        transmission", "quarantine", "prevention", "public", "
        surveillance", "population", "spread"],
    "research": ["study", "analysis", "data", "research", "model", "
        models", "sequencing", "clinical", "review", "findings"],
    "symptoms": ["symptoms", "fever", "cough", "dyspnea", "fatigue",
         "headache", "sore throat", "shortness of breath"],
```

```
    "methods": ["methods", "assay", "pcr", "rt-pcr", "sequencing", "imaging", "tools", "test", "tests", "diagnostic"],
    "public_health": ["health", "public", "hospital", "hospitals", "healthcare", "prevention", "quarantine", "epidemiology", "surveillance"],
    "miscellaneous": ["cases", "patients", "human", "novel", "background", "severity", "risk", "control", "management", "response"]
}

def Exploration_2(model, BME=kimi_BME, max_words=1000, seed=1022):
    random.seed(seed)

    # Extract vocabulary and corresponding word vectors
    BMEs = [word for words in list(BME.values()) for word in words]
    BME_classes = [c for c in list(BME.keys())]
    vocab = model.wv.index_to_key
    vocab_1 = list(filter(lambda x: x in BMEs, vocab))
    vocab_2 = sample(vocab, max_words - len(vocab_1))

    vocab = vocab_1 + vocab_2  # Extract vocabulary
    word_vectors = model.wv[vocab]  # Extract word vectors

    # Limit the number of words to visualize
    if len(vocab) > max_words:
        np.random.seed(seed)
        indices = np.random.choice(len(vocab), max_words, replace=False)
        vocab = [vocab[i] for i in indices]
        word_vectors = word_vectors[indices]

    # Use t-SNE to reduce dimensionality to 2D
    tsne = TSNE(n_components=2, random_state=seed, perplexity=75, init='pca', n_iter=1000)
    reduced_embeddings = tsne.fit_transform(word_vectors)

    # Plot the t-SNE visualization
    colors = ['pink', 'blue', 'green', 'yellow', 'purple', 'magenta', 'red', 'orange', 'cyan', 'lawngreen', 'olive', 'lightblue']
    colors = dict(zip(BME_classes, colors[:len(BME_classes)]))

    plt.figure(figsize=(14, 10))
    l = {}  # Store data for the legend

    for i, word in enumerate(vocab):
        x, y = reduced_embeddings[i]
        for c in BME_classes:
            if word in BME[c]:
                color = colors[c]
                size = 30
                l[c] = (x, y, color)
                break
            else:
                color = 'gray'
                size = 5
        plt.scatter(x, y, color=color, s=size)
        plt.text(x + 0.2, y + 0.2, word, fontsize=0.3 * size, color=color)
```

```
65
66      for key, value in l.items():
67          plt.scatter(value[0], value[1], color=value[2], s=size,
                label=key)
68
69      plt.title("t-SNE Visualization of Word Embeddings")
70      plt.legend()
71      plt.show()
72
73      return None
```

1.4.3 Co-occurrence

In this section, we demonstrate the biomedical entities which frequently cooccur with "coronavirus". The words are selected in order from highest to lowest according to the statistical frequency of co-occurrence with the target words in the corpus.

```
1  import pandas as pd
2
3  def Exploration_3(model, target_word='coronavirus', k=10,
       output_file='.\cooccurrence_words.xlsx'):
4      if target_word in model.vocab:
5          # Get similar words
6          cooccurrence_words = model.get_similar_words(target_word, k)
7          print("Top 10 similar words:")
8          for word, similarity in cooccurrence_words:
9              print('{0}: {1:.5f}'.format(word, similarity))
10
11         # Save to Excel
12         df = pd.DataFrame(cooccurrence_words, columns=['Word', '
               Similarity'])
13         df.to_excel(output_file, index=False)
14         print(f"Co-occurrence words saved to {output_file}")
15         return True
16     else:
17         print(f"{target_word} not in vocabulary.")
18         return False
```

1.4.4 Semantic Similariy

In this section, we compare the cosine distances of the embeddings obtained from word2vec in the high-dimensional vector space to identify the tokenized results most semantically similar to "coronavirus".

```
1  from sklearn.metrics.pairwise import cosine_similarity
2  import pandas as pd
3
4  def Exploration_4(model, target_word = "coronavirus", output_file='
       .\semantic_similar_words.xlsx'):
5      if target_word.lower() in model.wv:
6          target_vector = model.wv[target_word.lower()]
7          all_words = model.wv.index_to_key
8          similarities = []
9
10         for word in all_words:
11             if word != target_word.lower():
12                 similarity = cosine_similarity([target_vector], [
                       model.wv[word]])[0][0]
```

```
                similarities.append((word, similarity))

        sorted_similarities = sorted(similarities, key=lambda x: x
            [1], reverse=True)
        top_similar_words = sorted_similarities[:10]

        # Save to Excel
        df = pd.DataFrame(top_similar_words, columns=['Word', '
            Similarity'])
        df.to_excel(output_file, index=False)
        print(f"Top similar words saved to {output_file}")

        print("Top 10 similar words:")
        for word, similarity in top_similar_words:
            print('{0}:{1:.5f}'.format(word, similarity))
        return True

    else:
        print(f"{target_word.lower()} not in vocabulary.")

        return False
```

2 Results and Analysis

2.1 Tokenization

2.1.1 Use split()

Algorithm 1 Use split() for Tokenization

1: **Input:** Text file
2: **Output:** List of sampled words
3: Initialize empty list `words`
4: **for** each line in the file **do**
5: Tokenize line using regex `r"\W+"`
6: Append tokens to `words`
7: **end for**
8: Remove empty strings from `words`
9: Compute word frequencies using `Counter(words)`
10: Set `sample_size` to 20
11: Perform weighted sampling using word frequencies
12: **Return:** List of sampled words

- Tokenization and weighted sampling procedure:
  - Reads file line-by-line, tokenizing using regex `r"\W+"`
  - Filters out empty strings from `words` list
- Frequency-based weighted sampling:
  - Counts word occurrences with `Counter(words)`
  - Uses `random.choices()` for weighted random sampling based on frequency
  - `weights=word_counts.values()` ensures higher frequency words have greater probability of selection
- Random selection:
  - Selects 20 words using weighted probability distribution

– Resulting words are printed as output

source	for	acetate	number	rate
enzymes	of	respiratory	peptide	fruit
disease	virus	MERS	the	with
investigation	that	method	in	activation

Table 1: Weighted Random Sample of 20 Words

2.1.2 Use NLTK

Algorithm 2 Tokenization with NLTK

1: **Input:** `content of file`
2: **Output:** `words` (List of tokenized sentences)
3: Split content into sentences using `sent_tokenize()`
4: Define list of punctuation marks `interpunctuations`
5: Initialize empty list `words`
6: **for** each sentence in sentences **do**
7: Tokenize sentence into words using `word_tokenize()`
8: Remove punctuation words from tokenized sentence
9: Convert all words to lowercase
10: Append processed words to `words`
11: **end for**
12: **Return:** `words`

- Sentence segmentation:
  - Use `sent_tokenize()` to split input text into sentences.
  - Each sentence is processed separately.
- Tokenization:
  - For each sentence, `word_tokenize()` is used to split the sentence into words.
  - Punctuation marks are filtered out from tokenized words.
  - All words are converted to lowercase for uniformity.
- Output:
  - Returns a list of sentences, each containing a list of tokenized words (with punctuation removed).

covid-19	sara-cov-2	2019-ncoV	infection	pneumonia
transmission	symptoms	outbreak	wuhan	china
case	virus	fever	cough	diagnosis
treatment	epidemic	prevention	control	clinical

Table 2: Top 20 Representative High-Frequency Words of Tokenized Result of NLTK Tokenizer

2.1.3 Use Byte-Pair Encoding (BPE)

Algorithm 3 Tokenization with usage of Byte-Pair Encoding (BPE)

Input: content of file
Output: `tokens_list` (List of tokenized text chunks)
Load pre-trained BERT tokenizer `AutoTokenizer.from_pretrained("bert-base-uncased")`
Set `max_length` to 512
Initialize empty list `tokens_list`
for each chunk of text with length `max_length` **do**
    Tokenize chunk using BERT tokenizer `bert_tokenizer.tokenize(chunk)`
    Append tokenized chunk to `tokens_list`
end for
Write `tokens_list` to file `"bert_tokenized.txt"`
Return: `tokens_list`

- BERT Tokenizer:
  - Load the pre-trained BERT tokenizer `AutoTokenizer.from_pretrained("bert-base-uncased")`.
  - Tokenizer converts raw text into tokens suitable for BERT-based models.
- Text Chunking:
  - Split the input text into chunks of size `max_length` (512 tokens), ensuring that the input size does not exceed BERT's maximum token limit.
- Tokenization:
  - For each chunk, tokenize using BERT tokenizer.
  - Tokens are joined into strings using spaces and appended to `tokens_list`.
- File Output:
  - Save the tokenized text in a file `"bert_tokenized.txt"` for later use.
- Output:
  - Return the list `tokens_list` containing tokenized text chunks.

COVID-19	SARS-CoV-2	2019-nCoV	virus	infection
pneumonia	transmission	symptoms	patients	diagnosis
treatment	research	outbreak	cases	China
Wuhan	genome	epidemiology	clinical	prevention

Table 3: Top 20 Representative High-Frequency Words of Tokenized Result of BERT Tokenizer

2.1.4 Build new Byte-Pair Encoding (BPE)

Algorithm 4 Tokenization using BPE (Byte Pair Encoding)

Input: content of file
Output: result of tokenization
Initialize BPE model
Set up the trainer for the model
Training BPE model with content
Save the trained BPE model
Tokenize content using trained BPE model (as Track 2.3)
Write result to file
Return: result of tokenization

- Initialization of the BPE tokenizer:
  - Load a BPE tokenizer using `Tokenizer(models.BPE())`.
  - Pre-tokenization is set to whitespace-based, using `pre_tokenizers.Whitespace()`.
- BPE Model Training:
  - Define special tokens like `"<unk>", "<pad>", "<s>", "</s>"` for unknown, padding, start, and end tokens.
  - Train the tokenizer using the `BpeTrainer` with input data from `"titles_abstracts.txt"`.
- Model Saving:
  - Save the trained BPE tokenizer to a file named `"biomedical_bpe.json"` for future use.
- Tokenization:
  - Reload the saved BPE tokenizer and use it to tokenize the input `content`.
  - Output tokens are joined into a space-separated string.
- Return:
  - The tokenized result is written to `"bpe_tokenized.txt"` and returned as a list of tokens.

corona	virus	infection	COVID	pneumonia
epidemic	transmission	cases	patients	symptoms
treatment	control	quarantine	diagnosis	outbreak
SARS	MERS	vaccine	mortality	epidemiology

Table 4: Top 20 Representative High-Frequency Words of Tokenized Result of BERT Tokenizer

2.2 Explore the Word Representations

2.2.1 Visualise the Word Representations by t-SNE

Code Analysis

Algorithm 5 t-SNE Word Embedding Visualization

Input: Word embedding model *model*, Maximum words *max_words*
Output: 2D t-SNE visualization of word embeddings
Extract vocabulary *vocab* and word vectors *word_vectors* from *model*
if $|vocab| > max\_words$ **then**
    Randomly sample *max_words* words and their vectors
end if
Apply t-SNE to reduce *word_vectors* to 2D
Initialize plot for visualization
for all word in *vocab* **do**
    Plot word at its reduced 2D coordinates
end for
Display the visualization

- Random sampling is used to control visualization density. Only `max_words=1000` are randomly selected from the vocabulary to ensure the plot remains interpretable without overcrowding.
- t-SNE parameters are carefully chosen to optimize for the preservation of local structure:
  - `perplexity=75` strikes a balance between capturing local relationships (similar words) and global relationships (overall structure).

  - The algorithm starts with random initialization, which encourages a broad exploration of the word embedding space before settling into a meaningful visualization.
  - t-SNE runs for 1000 iterations to ensure sufficient convergence for accurate 2D representation of the word vectors.

- The visualization maintains a uniform and consistent style to ensure clarity:
  - Words are represented by blue dots with a size of `s=10`, making the points distinct without overlapping too much.
  - Text labels are offset by (0.2, 0.2) from the corresponding points and colored gray to avoid visual clutter and overlap.
  - The figure size is fixed at 14x10 to maintain an appropriate aspect ratio and visualization scale for better readability.

Visualization Results

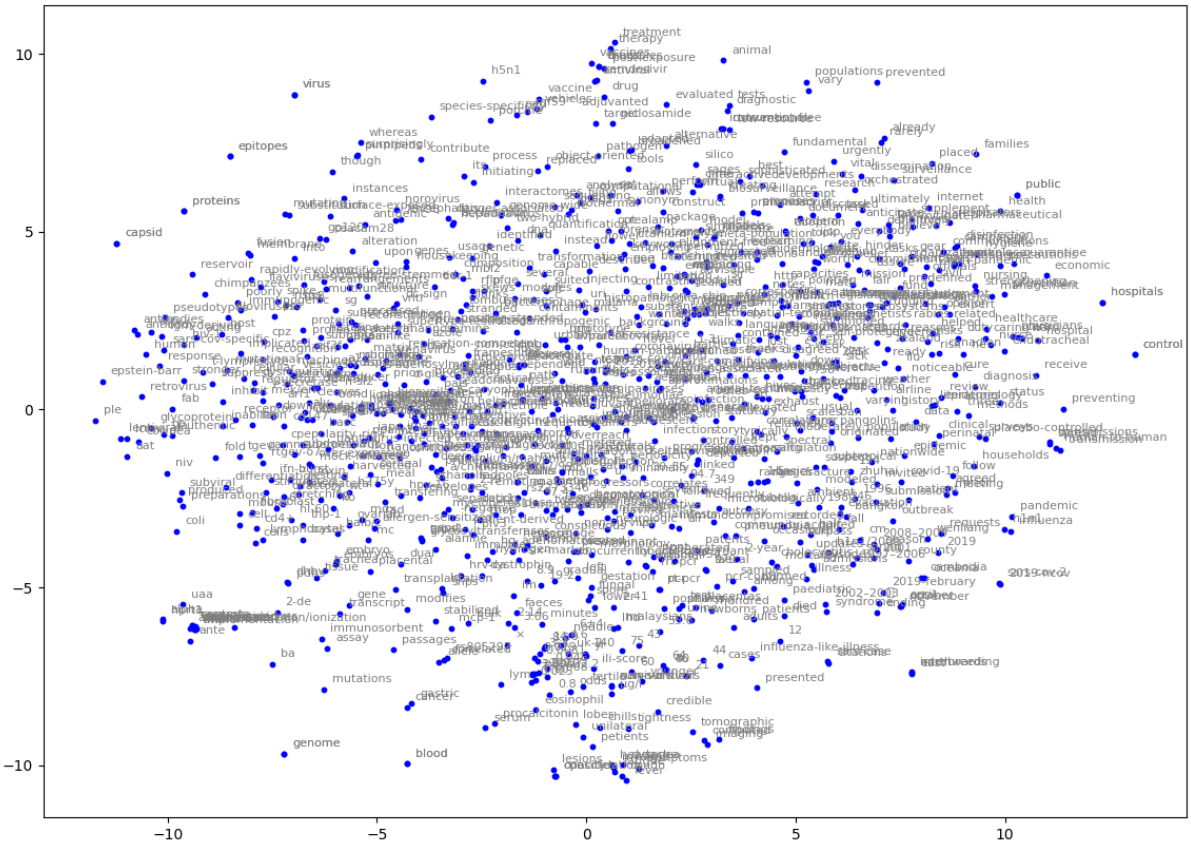

Figure 1: t-SNE Visualization of Word Embeddings

Spatial Topology

- **Dense central cluster** representing core semantic relationships
- **Dispersed peripheral words** indicating specialized or contextual terms
- **Non-linear arrangement** reflecting complex linguistic interconnections

Embedding Dynamics (Enabled by Code Parameters)

- **Randomized sampling** (`max_words=1000`) prevents visual overload
- `perplexity=75` balances local semantic clusters and global word distributions
- **Blue scatter points** with text annotations create an intuitive semantic map

Semantic Clustering Insights

- **Proximity between words** suggests conceptual similarity
- **Scattered blue dots** reveal nuanced semantic networks
- Demonstrates how `word2vec` models capture intricate linguistic contextual relationships

2.2.2 Visualise the Word Representations of Biomedical Entities by t-SNE

Code Analysis

Algorithm 6 t-SNE Word Embedding Visualization

Input: `model` (word embedding model), `vocabulary`, `max_words`
Output: 2D scatter plot of word embeddings
`words` ← `random.sample(vocabulary, max_words)` **if** `len(vocabulary) > max_words` **else** `vocabulary`
`vectors` ← `[model[word] for word in words]`
`embedded_words` ← `TSNE(n_components=2, perplexity=30, random_state=42).fit_transform(vectors)`
Initialize 2D scatter plot
for all `word, (x, y)` **in** `zip(words, embedded_words)` **do**
    Plot point at (x, y)
    Annotate point with corresponding `word`
end for
Show the scatter plot

- Hybrid vocabulary selection: - BME disease entities (vocab_1) - Random 300-word sample (vocab_2)
- Visual differentiation: - Disease terms: Larger points (30px), category-specific colors - General terms: Small gray points (5px) - Font size proportional to point size (0.3×)
- Color mapping: - Distinct colors for each disease class - Gray for non-classified terms

This implementation enables clear visualization of disease term relationships while maintaining context within the broader vocabulary space.

Visualization Results

Semantic Proximity Insights

- **Viral research terms** densely interconnected
- **Close spatial relationships** between epidemiological and clinical concepts
- **Clear differentiation** between molecular, clinical, and systemic research domains

Contextual Categorizations

- **Central region:** Generalized research methodologies
- **Peripheral regions:** Specialized technical terminology

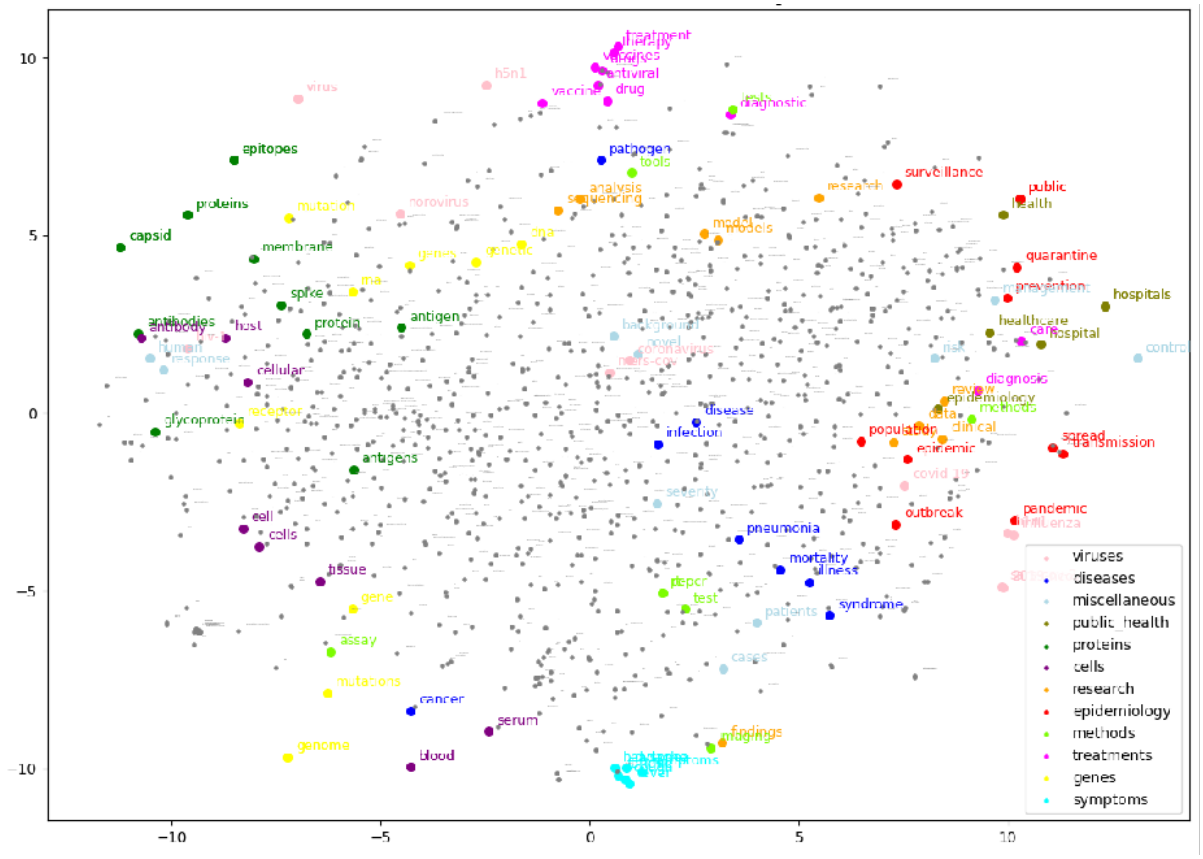

Figure 2: t-SNE Visualization of the Word Representations of Biomedical Entities

viruses diseases
proteins genes
cells treatments
epidemiology research
symptoms methods
public health miscellaneous

2.2.3 Co-occurrence

Algorithm 7 Co-occurrence Word Similarity Analysis

```
function EXPLORATION_3(model, target_word, k, output_file)
    if target_word in model vocabulary then
        cooccurrence_words ← model.get_similar_words(target_word, k)
        Print "Top 10 similar words:" (word, similarity) in cooccurrence_words
        Print word, similarity

        df ← pd.DataFrame(cooccurrence_words, columns=['Word', 'Similarity'])
        df.to_excel(output_file, index=False)
        Print "Co-occurrence words saved to " + output_file
        return True
    end if
    Print target_word + " not in vocabulary."
    return False
end function
```

Cross-Domain Relevance:

- The list contains terms from virology, epidemiology, and broader medical contexts, reflecting interdisciplinary discussions surrounding "coronavirus".

Contextual Associations:

- Many words relate to either the direct effects of COVID-19 (e.g. "persistent," "obstructive") or comparative studies involving other pathogens (e.g., "flaviviral," "SFTSV").

Broad Impact of COVID-19:

- The inclusion of terms like "Alzheimer" demonstrates how COVID-19 research spans beyond respiratory symptoms, touching neurological and systemic effects.

Co-occurring Word	Frequency
Covid	0.009722719
multiplicity	0.006220484
flaviviral	0.005976294
Lyme	0.005395208
productive	0.005248906
obstructive	0.005074524
mouth	0.004548549
persistent	0.004354533
Alzheimer	0.004330223
SFTSV	0.004047467

Table 5: 10 biomedical entities with the highest frequency of co-occurrence with coronavirus

2.2.4 Semantic Similarity

Algorithm 8 Semantic Similarity Analysis

```
function EXPLORATION_4(model, target_word, output_file)
    if target_word.lower() in model vocabulary then
        target_vector ← model.wv[target_word.lower()]
        similarities ← [(w, cosine_similarity([target_vector], [model.wv[w]])[0][0])
            for w in model.wv.index_to_key if w ≠ target_word.lower()]
        top_similar_words ← sorted(similarities, key=lambda x: x[1], reverse=True)[:10]
        pd.DataFrame(top_similar_words, columns=['Word', 'Similarity']).to_excel(output_file, index=False)
        Print "Top similar words saved to " + output_file
        Print top_similar_words
        return True
    end if
    Print "Target word not in vocabulary."
    return False
end function
```

Model Strengths:

- The skip-gram model successfully captures semantic relationships, synonyms, and abbreviations (e.g., "2019-novel," "ncov-2019").
- Contextual and domain-specific terms like "provisionally" and "abstract" are included, showcasing the model's sensitivity to real-world usage.

Error Handling:

- Recognition of typographical variants like "coronovirus" reflects the model's robustness against minor text inconsistencies.

Temporal and Specific References:

- Terms like "2019" and "2019-ncov" highlight the temporal specificity of the dataset and the focus on SARS-CoV-2-related contexts.

Semantic Similar Word	Similarity
novel	0.6320
2019-novel	0.6185
coronovirus	0.6100
abstract	0.5961
provisionally	0.5929
2019-ncov	0.5895
2019	0.5876
ncov-2019	0.5767
cov	0.5766
2019-novel	0.6185

Table 6: 10 biomedical entities with the highest semantic similarity with coronavirus

3 Reflection

3.1 Strengths

1. **Comprehensive Data Handling**
   - The data preprocessing section is well-designed, efficiently handling JSON parsing and extraction.
   - Alternative strategies are considered for handling different data availability scenarios, improving the robustness of the approach.
2. **Utilization of Standard Libraries**
   - The notebook makes effective use of widely adopted NLP libraries, ensuring compatibility, scalability, and performance.
   - The structured approach to data extraction enhances readability and maintainability.
3. **Reproducibility and Environment Setup**
   - Clear instructions are provided for setting up the environment, including system requirements and data acquisition procedures.
   - The use of standardized commands facilitates easy dataset retrieval and preparation.

3.2 Weaknesses

1. **Lack of Robust Exception Handling**
   - Error handling mechanisms are not consistently applied, which may lead to unexpected failures during file operations or data processing.
   - Improved exception handling would enhance fault tolerance and user experience.

2. **Potential Inefficiencies in Large-Scale Processing**

   - The notebook processes a large dataset, but there is no explicit mention of optimization techniques such as parallel processing or batch operations.
   - Leveraging more efficient computation strategies could significantly improve performance.

3. **Limited Data Visualization**

   - While the focus is on text processing, the absence of exploratory data analysis (EDA) limits insights into data distribution and characteristics.
   - Integrating visualization techniques could provide a more comprehensive understanding of the dataset.

3.3 Overall Evaluation

- **Strengths:** Well-structured framework, effective data preprocessing, appropriate library usage, and clear setup instructions.
- **Weaknesses:** Limited portability, lack of robust error handling, inefficiencies in large-scale processing, and insufficient data visualization.

References

[1] S. Bird, "Nltk: The natural language toolkit," in *Annual Meeting of the Association for Computational Linguistics*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:1438450

[2] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Comput. Linguist.*, vol. 18, no. 4, p. 467–479, Dec. 1992.

[3] A. Fonarev, O. Hrinchuk, G. Gusev, P. Serdyukov, and I. Oseledets, "Riemannian optimization for skip-gram negative sampling," *ArXiv*, vol. abs/1704.08059, 2017.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019.

[5] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *ArXiv*, vol. abs/2106.09685, 2021.