Name: Chia Bing Xuan

Matriculation Number: A0259419R

Title: DSA4213 Assignment 1 – Word Embedding Exploration

1. Introduction

In this assignment, I conducted experiments on various word embedding algorithms. I considered three techniques:

- Skip-Gram with Negative Sampling (SGNS)
- Shifted Positive Pointwise Mutual Information - Singular Value Decomposition (SPPMI-SVD)
- GloVe

I implemented each of the aforementioned techniques, utilising both qualitative and quantitative metrics to evaluate and compare their performances on a selected corpus.

2. Explanation of Algorithms

2.1. SGNS

The skip-gram model is a specific model variant of the Word2Vec family. The fundamental working principle of skip-gram models lies in distributional semantics – the idea that the meaning of a word can be deduced from its neighbouring words, within a preset context window. Ultimately, our goal is to ensure that words appearing in similar contexts will have similar vector embeddings.

Suppose we have a large corpus of length T. We also set the size of the context window, m. For each position $t = 1, 2, \dots, T$, we consider the word at the centre of the window, $\boldsymbol{w_t}$, as well as the surrounding words (context words) within the window, $\{\boldsymbol{w_{t+j}}\}_{\substack{-m \le j \le m, \\ j \neq 0}}$. For skip-gram models in particular, we aim to maximise the conditional probability of each context word, given the centre word. In other words, if $\boldsymbol{\theta}$ is the concatenation of all word embeddings to be obtained, we want to maximise

$$L(\boldsymbol{\theta}) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m, \\ j \neq 0}} P\big(\boldsymbol{w_{t+j}} \,\big|\, \boldsymbol{w_t}\,;\, \boldsymbol{\theta}\big)$$

which is equivalent to minimising the objective function

$$J(\boldsymbol{\theta}) = -\frac{1}{T} \log L(\boldsymbol{\theta}) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m, \\ j \neq 0}} \log P\big(\boldsymbol{w_{t+j}} \,\big|\, \boldsymbol{w_t}\,;\, \boldsymbol{\theta}\big)$$

Note that each word will either be a context word or a centre word, and this role changes as the window shifts. Thus, we associate each word w with two embeddings – $\boldsymbol{u_w}$ for when it is a context word and $\boldsymbol{v_w}$ for when it is a centre word. For a centre word c and a context word o, the vanilla skip-gram algorithm defines the conditional probability using the softmax function:

$$P(o \mid c) = \frac{e^{\boldsymbol{u_o^T v_c}}}{\sum_{w \in V} e^{\boldsymbol{u_w^T v_c}}}$$

where V is the set of unique words in the corpus. However, this is computationally expensive to determine, as we would have to sum over all the words in V to do so. In my implementation of skip-gram Word2Vec, I made use of negative sampling. For each centre word c, this involves selecting K words that are outside of the context window, but still within the vocabulary. In essence, we aim to maximise the similarities between actual context words and c, whilst minimising the similarities between negative samples and c. Hence, instead of minimising $-P(o \mid c) = -\frac{e^{\boldsymbol{u_o^T v_c}}}{\sum_{w \in V} e^{\boldsymbol{u_w^T v_c}}}$ in the definition of $J(\boldsymbol{\theta})$, we minimise

$$J_{neg}(\boldsymbol{u_o}, \boldsymbol{v_c}, U) = -\log \sigma(\boldsymbol{u_o^T v_c}) - \sum_{k \in \{K \text{ negative samples}\}} \log \sigma(-\boldsymbol{u_k^T v_c})$$

where σ is the sigmoid function and U is the unigram distribution present in the distribution $P(w) = \frac{U(w)^{\frac{3}{4}}}{Z}$, from which the negative words are sampled.

Optimisation techniques can then be used to find θ that corresponds to minimum loss.

2.2. SPPMI-SVD

The SPPMI-SVD algorithm makes use of the pointwise mutual information (PMI) metric, in the context of word co-occurrences.

Suppose we have a large corpus of many documents. We also set the size of the context window, m, which is to be shifted throughout each document in the corpus. For a word w and a context c, w co-occurs with c if c appears in the context window centred at w. In this way, we can construct a co-occurrence matrix, in which the (i, j)-th entry corresponds to the number of co-occurrences of word i with context j, throughout the corpus. With this, each word in the vocabulary corresponds to a single row in the co-occurrence matrix, which is a vector that encodes information about how often other words appear near that word.

Then the PMI of each (i, j) pair is defined by

$$PMI(i, j) = \log \frac{\#(i, j)|D|}{\#(i)\#(j)}$$

where $|D|$ is the sum of all the elements of the co-occurrence matrix and $\#(i,j)$ is the number of co-occurrences of word i with context j. $\#(i)$ and $\#(j)$ are respectively the number of times word i and word j are the centre words.

The shifted positive PMI metric is then defined by

$$SPPMI_k(i,j) = \max(PMI(i,j) - \log k, 0)$$

where k is a hyperparameter that dictates some shift applied to the PMI value. The SPPMI function can then be applied element-wise to the co-occurrence matrix, obtaining a SPPMI matrix $\boldsymbol{S}$. Singular Value Decomposition can then be applied to $\boldsymbol{S}$ for dimensionality reduction:

$$\boldsymbol{S} = \boldsymbol{U\Sigma V}^T$$

Keeping the n largest singular values, the reduced word embeddings are (based on my implementation):

$$\boldsymbol{E} = \boldsymbol{U_n \Sigma_n}$$

where $\boldsymbol{U_n}$ corresponds to the n leftmost columns of $\boldsymbol{U}$ and $\boldsymbol{\Sigma_n}$ is the diagonal matrix containing the n largest singular values.

2.3. GloVe

The intuition for GloVe arises from how a word i can be distinguished from another word j by considering the ratios of conditional probabilities, with respect to some probe words. For instance, we consider the ratios $\frac{P(k_1|i)}{P(k_1|j)}$ and $\frac{P(k_2|i)}{P(k_2|j)}$ for probe words k_1 and k_2. As an example, in the case of

$$i = \text{"ice"}, j = \text{"steam"}, k_1 = \text{"solid"}, k_2 = \text{"gas"}$$

we might have $\frac{P(k_1|i)}{P(k_1|j)} \gg 1$ and $\frac{P(k_2|i)}{P(k_2|j)} \ll 1$, which would allow us to differentiate between the two target words “ice” and “steam” in that way.

With this in mind, we can make use of the log-bilinear model to define the GloVe word embeddings $\boldsymbol{w_i}$, $\boldsymbol{w_j}$ and $\boldsymbol{w_k}$, ensuring that the difference $\boldsymbol{w_i} - \boldsymbol{w_j}$ can reproduce the ratio of conditional probabilities, thereby encoding meaning components:

$$\boldsymbol{w_k} \cdot (\boldsymbol{w_i} - \boldsymbol{w_j}) = \log \frac{P(k|i)}{P(k|j)}$$

Note that the conditional probability $P(k|i)$ (and similarly $P(k|j)$) can be expressed in terms of co-occurrences, using the co-occurrence matrix $\boldsymbol{X}$:

$$P(k|i) = \frac{\boldsymbol{X}_{ik}}{\sum_{w \in V} \boldsymbol{X}_{iw}}$$

where $\boldsymbol{X}_{ab}$ is the (a,b)-th entry of $\boldsymbol{X}$.

After derivation, the loss function for GloVe embeddings is given as

$$J = \sum_{i,j=1}^{V} f\left(\boldsymbol{X}_{ij}\right) \left(\boldsymbol{w}_{\boldsymbol{i}}^{T} \widetilde{\boldsymbol{w}}_{\boldsymbol{j}} + \boldsymbol{b}_{\boldsymbol{i}} + \widetilde{\boldsymbol{b}}_{\boldsymbol{j}} - \log \boldsymbol{X}_{ij}\right)^2$$

with weighting function f. We associate each word w with two embeddings – $\boldsymbol{w}$ for when it is a target word and $\widetilde{\boldsymbol{w}}$ for when it is a probe word.

3. Methodology

3.1. Selection of Corpus and Data Processing

For this assignment, I investigated the effectiveness of the aforementioned embedding techniques, on a corpus of IMDB movie reviews offered by the NLTK library. This dataset consists of 1000 positive reviews and 1000 negative reviews compiled by Bo Pang and Lillian Lee. Note that each review can be loaded as a list of raw tokens, along with its associated sentiment label (either positive or negative). Throughout the workflow, a seed of 42 was set for reproducibility.

A round of data pre-processing was carried out on this dataset:

1. Convert the class labels "pos" and "neg" into their corresponding integer labels (1 and 0 respectively)
2. For each review (list of raw tokens):
    a. Convert the tokens to lowercase
    b. Remove tokens that do not consist entirely of alphabets (eg. punctuation marks and numerals)
    c. Remove stop words (eg. function words such as "a", "the", "it", etc.)
    d. Lemmatise the remaining tokens (convert to root word / base form)

The most common words in the processed dataset are the following:

Word	**Count**
film	11053
movie	6977
one	6028
character	3879
like	3789
time	2979
get	2814
scene	2671
make	2634
even	2568

3.2. Experimentation

I started by performing a train-test split of 80%-20% on the processed dataset, before using the train set for my embedding exploration.

Firstly, I applied each embedding technique on the train set using the following set of parameters. Note that the set of parameters used are consistent across the techniques (the shift k is analogous to the number of negative samples in SGNS).

Embedding Technique	**Parameters**
SGNS	Number of epochs: 20 Minimum frequency of words to consider: 1 Vector size: 50 Window size: 3 Number of negative samples: 5
SPPMI-SVD	Minimum frequency of words to consider: 1 Vector size: 50 Window size: 3 Shift (k): 5
GloVe	Number of epochs: 20 Minimum frequency of words to consider: 1 Vector size: 50 Window size: 3 Number of negative samples: 5

I subsequently carried out hyperparameter tuning by iterating through various sets of hyperparameters, attempting to find the one that leads to optimal performance for each embedding technique. To quantify performance, I made use of the Spearman correlation coefficient. This was calculated between human-labelled similarity scores in the WordSim-353 dataset, as well as the corresponding cosine similarity values produced by each fitted model.

The following are the hyperparameter values that were experimented with, which were consistent across the three techniques:

Parameter	**Values Considered**
Vector size	50, 100, 150
Window size	3, 5, 10
Number of negative samples (SGNS, GloVe) / Shift (k) (SPPMI-SVD)	3, 5, 10

For each embedding technique, I then compared the quality of the new embeddings with the original embeddings. This was done qualitatively, by analysing the nearest neighbours of a list of selected words. If hyperparameter tuning did not lead to any significant improvement in the quality of the nearest neighbours, the original embeddings were chosen – otherwise, I selected the embeddings produced from hyperparameter tuning.

3.3. Sentiment Analysis

After selecting a set of word embeddings for each technique, I conducted sentiment analysis on the movie reviews dataset, as part of a downstream task. Using a set of word embeddings $\boldsymbol{E}$, it is possible to convert each review in the dataset (both train and test) to a feature vector. The steps taken were as follows:

1. For each review:
    a. Ignore words that are not present in the vocabulary of $\boldsymbol{E}$
    b. Convert all remaining words to their corresponding vector embeddings
    c. Determine the mean of all these vector embeddings element-wise to obtain a feature vector for the review
2. Using the embeddings and class labels in the train set, train a classifier to identify the sentiment of a given review.
    a. Models used: Random Forests, Support Vector Machines (SVMs), XGBoost
3. Evaluate the classifier on the embeddings and class labels in the test set

4. Evaluation

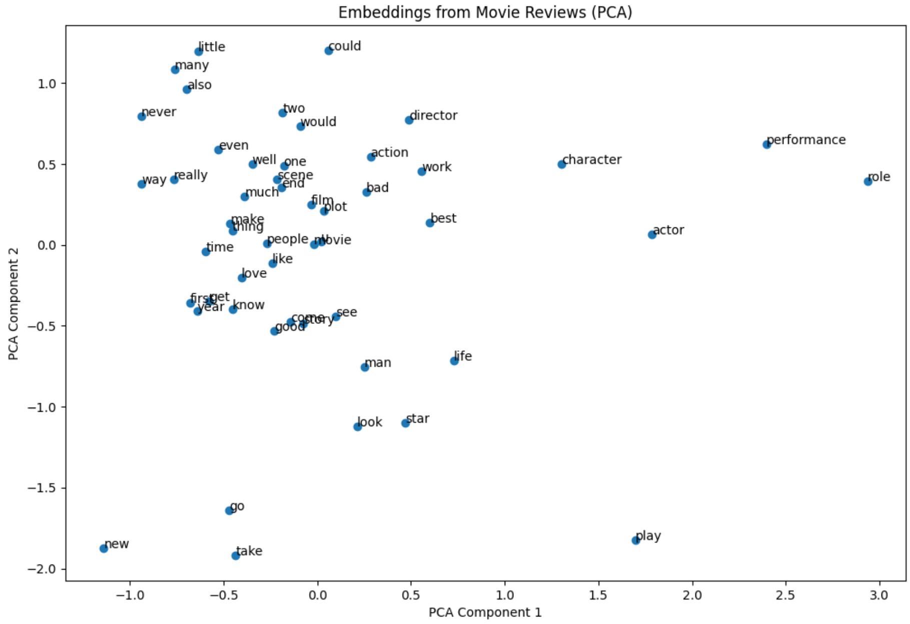
4.1. SGNS

4.1.1. Fixed Set of Parameters

A Word2Vec model was fitted on the train set with the gensim library in Python. A fixed set of parameters was used:

- Number of epochs: 20
- Minimum frequency of words to consider: 1
- Vector size: 50
- Window size: 3
- Number of negative samples: 5

Upon obtaining the word embeddings, principal component analysis (PCA) was used to reduce them to vectors of size 2. The corresponding PCA scatter plot of the 50 most common words in the corpus is shown below.



It can be seen that the SGNS model was able to separate out words of certain categories. For example, words related to individual cast members – such as “character”, “actor”, “role” and “performance” are significantly further away from the other word clusters. Moreover, words describing the segments of a movie (in part or whole) – such as “movie”, “film”, “scene” and “plot” are located near one another, although “story” is positioned further away. Words describing quantity – such as “little” and “many” – are also within close proximity of one another. However, this may not be ideal since “little” and “many” are opposite in meaning. Generally, the embeddings are part of one large cluster, with smaller clusters located further away from the main one. Some of these smaller clusters – such as the one containing “new”, “go” and “take” – are less reasonable, given that these words do not have similar meanings.

I also analysed the nearest neighbours for a list of ten common words (“film”, “like”, “good”, “time”, “story”, “character”, “life” and “scene”), checking if they are valid or otherwise.

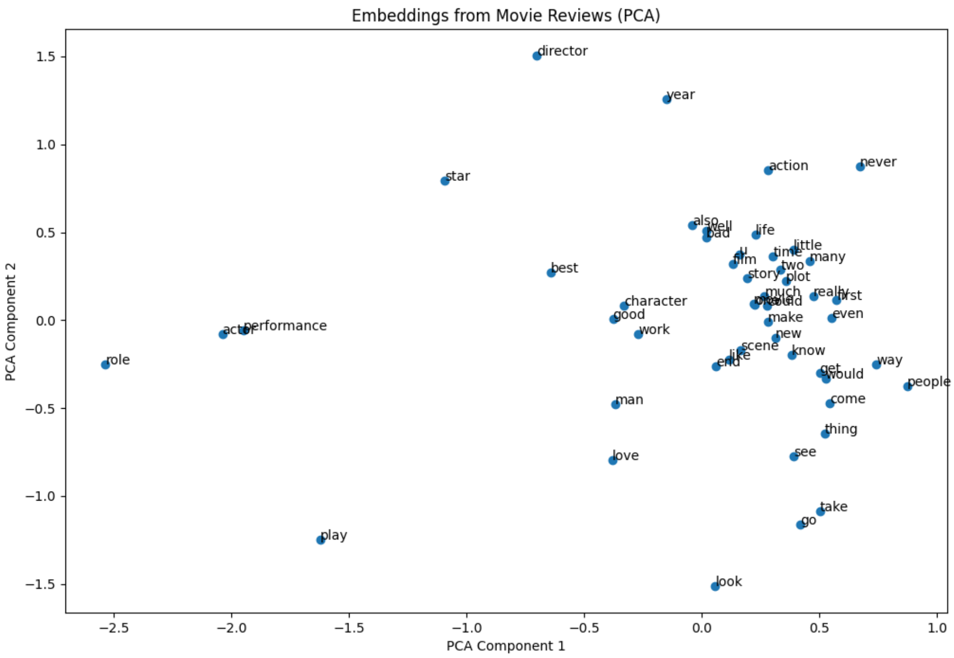
Test Word	**Nearest Neighbours (in descending order of cosine similarity)**	**General Semantic Meaning**
film	movie, unsatisfactory, expanded, mant, emphatically	
like	synch, embarrassed, sake, kinship, xerox	
good	decent, eas, great, terrible, paled	Adjectives that evaluate quality
time	booted, waaaay, heartbreaker, percent, scarce	
story	parallel, storyline, overlap, analogy, linear	Describes the plot of a movie

character	personality, tangential, role, incorrectly, attachment	Describes the traits of individuals in a movie (eg. personality, role)
life	comfort, harmony, miracle, live, sacrificing	Intangible aspects of life
scene	moment, sequence, confrontation, straw, gunfight	Various movie scenes

As detailed in the "General Semantic Meaning" column, the nearest neighbours of some words are logical. For example, the word most similar to "film" to "movie", as expected. The word "sacrificing" is similar to "life", since they are likely to appear in similar contexts, such as in the phrase "sacrificing [one's] life". The word "good" is most similar to other words with positive connotations, such as "decent" and "great". While it makes less sense for "good" to be similar to "terrible", this can be explained by how these two words have the same function – both are used to evaluate the quality of something or someone. However, the closest neighbours for other words are less reasonable, such as the words "like" and "time".

4.1.2. Hyperparameter Tuning

After hyperparameter tuning with Spearman correlation coefficient (using WordSim-353 dataset), the optimal parameters of vector size = 150, context window = 10 and negative samples = 5 were obtained. The results are shown below.



Test Word	Nearest Neighbours (in descending order of cosine similarity)
film	movie, rejuvenates, godforsaken, unconventionally, horor
like	ewwwww, unflushed, interferred, glisten, cagney
good	expended, commensurate, imaganitive, bregman, faulted
time	wayyy, bullsh*tting, rewound, buddying, lamanna
story	ascribe, comprehendably, brining, unsurprising, tangentially
character	tangential, logistical, recaptured, unfetching, thorougly
life	fullest, bottlecap, predetermined, overtaken, touchingly
scene	impart, storyboarded, choppily, humping, unusable

The nearest neighbours here are generally less logical. From this qualitative analysis, the performance of the word embeddings appears to have worsened after hyperparameter tuning. Thus, the original embeddings were chosen for SGNS.

4.2. SPPMI-SVD

4.2.1. Fixed Set of Parameters

SPPMI-SVD was conducted on the train set. A fixed set of parameters was used:

- Minimum frequency of words to consider: 1
- Vector size: 50
- Window size: 3
- Shift (k): 5

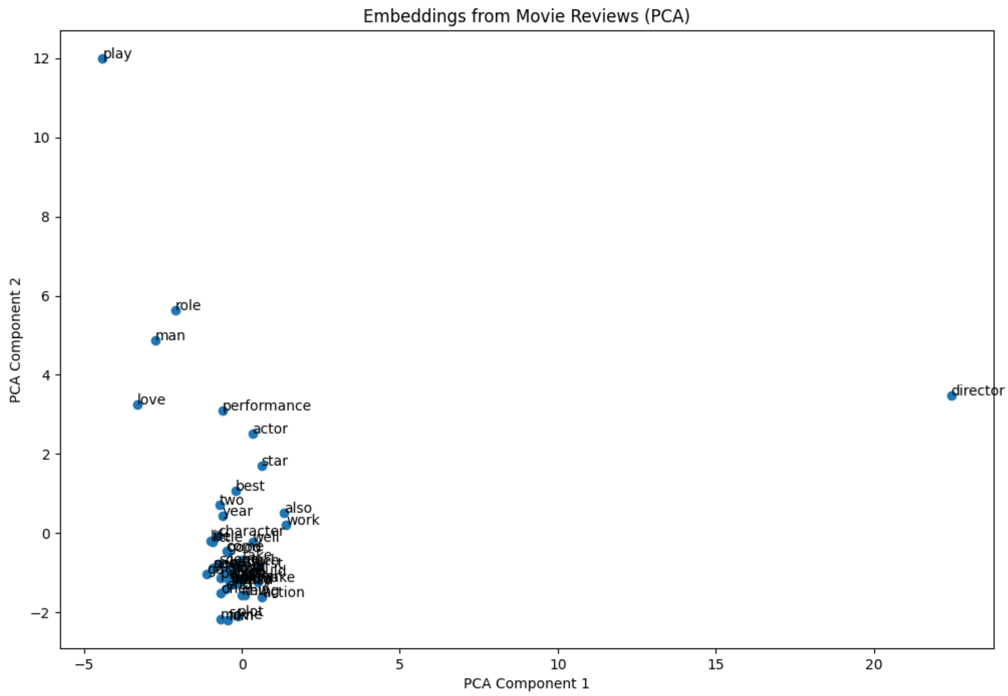
The results are shown below.

Test Word	Nearest Neighbours (in descending order of cosine similarity)
film	interferred, aspired, script, movie, look
like	come, think, watching, get, see
good	time, bad, much, little, make
time	much, even, really, better, could
story	time, little, made, interesting, give
character	never, even, good, interesting, really
life	story, come, actually, much, even
scene	much, never, time, audience, enough

Note that the nearest neighbours for SPPMI-SVD are less reasonable. For example, “time”, “little” and “made” do not have much in common with the word “story”. Hence, I looked to improve this performance through hyperparameter tuning.

4.2.2. Hyperparameter Tuning

After hyperparameter tuning with Spearman correlation coefficient, the optimal parameters of vector size = 150, context window = 3 and shift = 5 were obtained. The results are shown below.



The positions of the SPPMI-SVD word embeddings are much more varied (over a larger area), compared to that of SGNS. Note that words describing the segments of a movie (in part or whole) – such as "movie", "film" and "plot" are correctly located near one another, much like in SGNS. Words related to individual cast members – such as "actor", "star", "role" and "performance" are also detached from the main word cluster. For SPPMI-SVD, the words that are positioned outside of the main cluster (eg. "play", "director", "role", "man", "love" and "performance") were also singled out in SGNS.

Test Word	**Nearest Neighbours (in descending order of cosine similarity)**	**General Semantic Meaning**
film	movie, made, many, could, much	
like	one, really, know, look, even	
good	well, one, really, time, much	
time	see, one, know, much, even	
story	plot, character, many, time, however	Describes the plot of a movie
character	much, even, however, good, one	
life	people, even, real, one, time	
scene	one, see, even, well, film	

Based on the nearest neighbour analysis, the quality of SPPMI-SVD embeddings has improved after hyperparameter tuning. For instance, the nearest neighbours for "story" now better reflect the plot of a movie (eg. "plot" and "character"). The nearest neighbour to "film" is now more reasonable as well ("movie" instead of "interferred"). Some pairs of similar words also make sense, as they tend to be used together (eg. "good time" and "real life"). This is characteristic of the SPPMI-SVD algorithm, which is based on pairwise word co-occurrences. However, the nearest neighbours generally do not reflect a common semantic meaning related to each test word, with the exception of "story". This pales in comparison to SGNS, where the nearest neighbours of "good", "story", "character", "life" and "scene" each collectively describe a general idea relevant to the word in question. Since there is greater value in gathering words which convey similar meanings (rather than merely identifying which words are used alongside one another), it can be concluded that the SPPMI-SVD algorithm performs worse than SGNS.

Regardless, since hyperparameter tuning has improved the quality of the SPPMI-SVD word embeddings, the new embeddings were chosen for SPPMI-SVD.

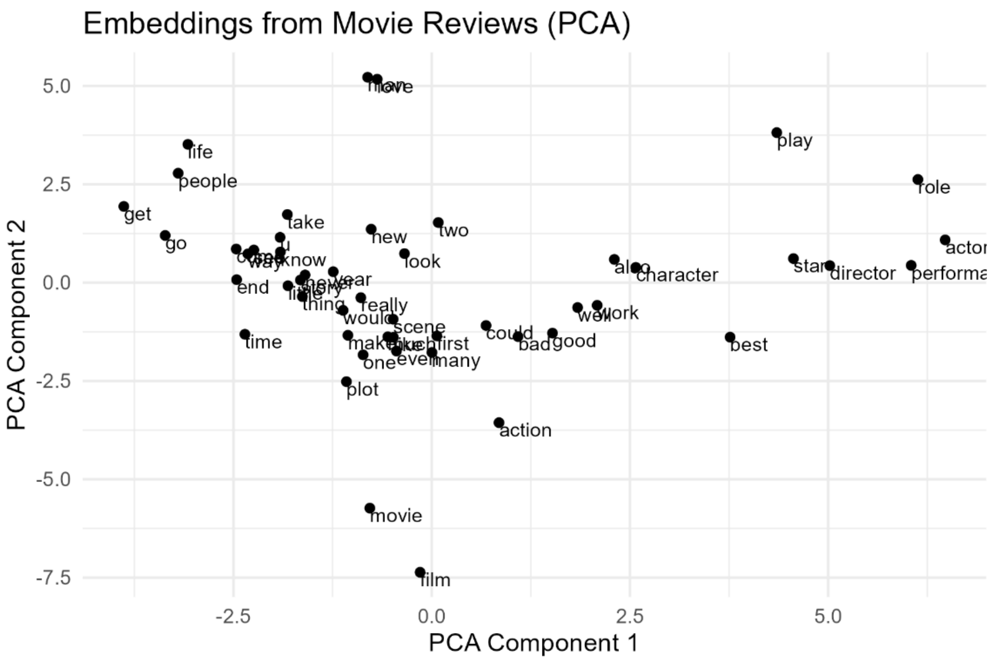
4.3. GloVe

4.3.1. Fixed Set of Parameters

A GloVe model was fitted on the train set with the text2vec library in R. A fixed set of parameters was used:

- Number of epochs: 20
- Minimum frequency of words to consider: 1
- Vector size: 50
- Window size: 3
- Number of negative samples: 5

The results are shown below.

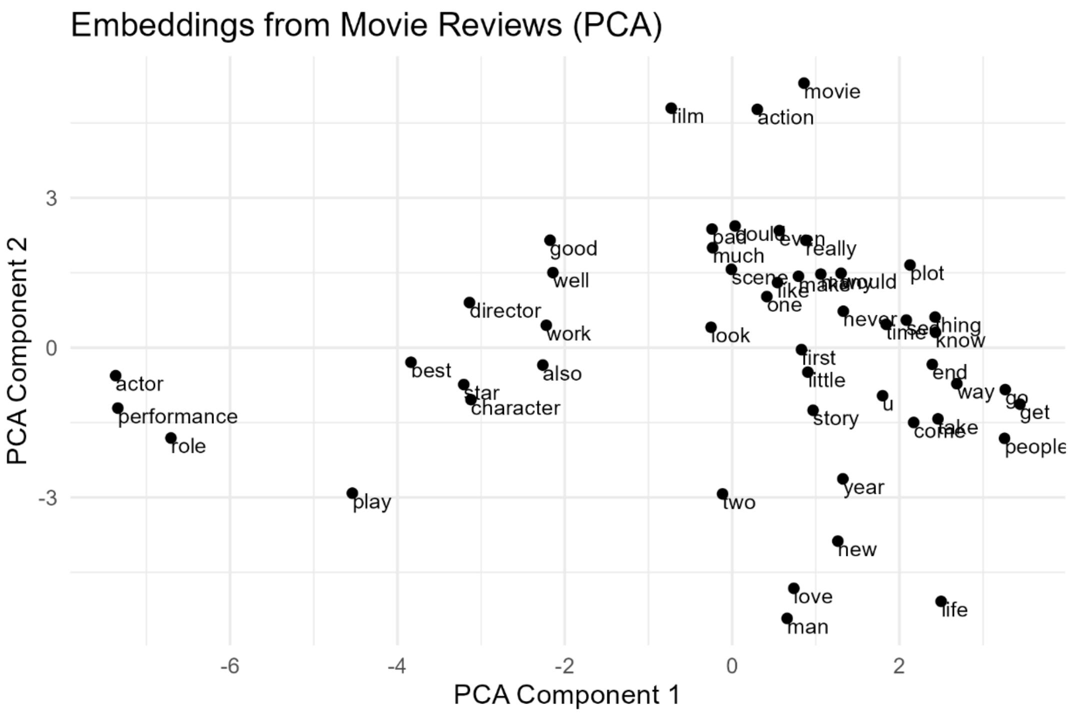


Test Word	Nearest Neighbours (in descending order of cosine similarity)
film	movie, one, however, since, even
like	movie, look, kind, actually, even
good	bad, also, making, make, look
time	long, since, movie, one, much
story	plot, rather, tell, character, way
character	seems, main, also, rather, interesting
life	real, find, world, take, way
scene	sequence, one, moment, particularly, also

Note that the nearest neighbours for GloVe have some room for improvement. For example, "movie" and "look" do not have much in common with the word "like". Hence, I looked to enhance this performance through hyperparameter tuning.

4.3.2. Hyperparameter Tuning

After hyperparameter tuning with Spearman correlation coefficient, the optimal parameters of vector size = 50, context window = 10 and negative samples = 10 were obtained. The results are shown below.



Note that the scatter plot after hyperparameter tuning contains more distinct clusters, which is an improvement. The GloVe embeddings of "actor", "performance" and "role" – all of which pertain to individual cast members – are located further away from the main cluster, much like in SGNS and SPPMI-SVD. In addition, the words "movie", "film" and "action" (as in "action movie") are also singled out, positioned within close proximity of each other. Notably, words with positive connotations – such as "good", "best", "well" and "star" are also grouped more closely together, which is an enhancement from the previous two embedding techniques. Another new development is that "character", "director" and "star" (as in "movie star") are all located near each other – which can be explained by the fact that these words are related to the people involved in a movie. However, the pair of "man" and "love" is less reasonable, as these words are not very similar to each other from a semantic standpoint.

Test Word	**Nearest Neighbours (in descending order of cosine similarity)**	**General Semantic Meaning**
film	movie, one, many, made, even	
like	one, movie, even, good, really	
good	bad, even, like, also, really	
time	one, first, long, two, however	Elements of time
story	plot, film, character, way, also	Describes the plot of a movie
character	also, main, one, story, interesting	
life	real, world, people, find, come	Intangible aspects of life
scene	moment, sequence, another, one, also	Sections of a movie

Based on the nearest neighbour analysis, the quality of GloVe embeddings has slightly improved after hyperparameter tuning. For instance, the word "good" is now one of the nearest neighbours for "like" (and vice versa), which better reflects the idea of a preference for something or someone. Like the previous techniques, GloVe correctly identifies "movie" as the most similar word to "film". As mentioned previously, it is also somewhat reasonable for "bad" to be similar to "good", despite them being polar opposites. Furthermore, like SGNS, certain test words have nearest neighbours that accurately reflect a collective semantic meaning. For example, "first" and "long" possess temporal elements (test word: "time"), while "sequence" and "moment" both describe segments of a movie (test word: "scene"). Some pairs of similar words are also valid, as they tend to be used together (eg. "main character"). This is unsurprising as the GloVe algorithm is based on pairwise word co-occurrences, like SPPMI-SVD. However, there are several cases where the nearest neighbours are less representative of the given word's meaning. Examples include some of the neighbours for "film" – such as "one", "many", "made" and "even".

Since hyperparameter tuning has improved the quality of the GloVe word embeddings, the new embeddings were chosen for GloVe.

4.4. Sentiment Analysis

Using the chosen word embeddings from SGNS (without hyperparameter tuning), SPPMI-SVD (with hyperparameter tuning) and GloVe (with hyperparameter tuning), I conducted sentiment analysis on the movie reviews dataset. This was done by training various scikit-learn classifiers on the train set, before evaluating their performances on the test set. The evaluation results, in terms of accuracy, are shown below.

	RandomForestClassifier	**SVM**	**XGBoost**
SGNS	73.0	**77.5**	74.8
SPPMI-SVD	72.0	**76.0**	74.8
GloVe	66.0	**70.0**	65.5

For this sentiment classification task, SVMs consistently performs better across all the embedding techniques. Among all <embedding, classifier> pairs, SGNS with SVM attains the strongest performance, with an accuracy of 77.5%.

5. Conclusion

An investigation was conducted on three embedding techniques – SGNS, SPPMI-SVD and GloVe – using the NLTK movie reviews dataset. The embedding performance was evaluated in two ways. Firstly, I analysed the nearest neighbours of a list of test words. Furthermore, I also used the embeddings for a downstream sentiment analysis task (extrinsic word vector evaluation). In sum, SGNS is the algorithm that achieves the best performance on this corpus.

6. Appendix

AI Tool Declaration: I used GPT-5 to assist in the creation of code and improve the phrasing of the report. I am responsible for the content and quality of the submitted work.

The repository for this assignment can be found at https://github.com/chiabingxuan/Word-Embeddings-Exploration.

Attached below are the code snippets used.

Imports

In [1]:
```
from gensim.models import Word2Vec
import matplotlib.pyplot as plt
import nltk
from nltk.corpus import movie_reviews, stopwords, wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import numpy as np
import os
import pandas as pd
import pickle
from scipy.stats import spearmanr
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split
from typing import Callable

nltk.download("punkt_tab")
nltk.download("movie_reviews")
nltk.download("stopwords")
nltk.download("wordnet")
```

```
[nltk_data] Downloading package punkt_tab to
[nltk_data]     C:\Users\bxchi\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package movie_reviews to
[nltk_data]     C:\Users\bxchi\AppData\Roaming\nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\bxchi\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\bxchi\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Out[1]: True

Setup

In [2]:
```
# Set seed for reproducibility
seed = 42

# Create folders
os.makedirs(os.path.normpath(os.path.join("..", "data")), exist_ok=True)
os.makedirs(os.path.normpath(os.path.join("..", "embedding_outputs")), exist_ok=True)
os.makedirs(os.path.normpath(os.path.join("..", "embedding_plots")), exist_ok=True)
```

Helpers

Visualisation

In [3]:
```
def visualise_embeddings(embeddings: np.ndarray, words: list[str], filename: str, seed: int = seed) -> None:
  # Use PCA to reduce dimensionality
  pca = PCA(n_components=2, random_state=seed)
  embeddings = pca.fit_transform(embeddings)

  # Plot the embeddings
  plt.figure(figsize=(12, 8))
  plt.scatter(embeddings[:, 0], embeddings[:, 1], marker="o")

  for i, word in enumerate(words):
      plt.annotate(word, xy=(embeddings[i, 0], embeddings[i, 1]), fontsize=10)

  plt.title("Embeddings from Movie Reviews (PCA)")
  plt.xlabel("PCA Component 1")
  plt.ylabel("PCA Component 2")

  # Save the plot
  plt.savefig(os.path.normpath(os.path.join("..", "embedding_plots", f"{filename}.png")))

  plt.show()
```

WordSim-353 with Spearman Coefficient (For Hyperparameter Tuning)

In [4]:
```python
# Load the WordSim-353 dataset into a dictionary of pairs to actual similarities
def load_wordsim353() -> dict[tuple[str, str], float]:
    wordsim353_df = pd.read_csv(os.path.normpath(os.path.join("..", "data", "wordsim353crowd.csv")))

    wordsim353_pairs_to_scores = dict()
    for _, row in wordsim353_df.iterrows():
      word_1, word_2 = sorted([row["Word 1"].lower(), row["Word 2"].lower()])
      score = row["Human (Mean)"]
      wordsim353_pairs_to_scores[(word_1, word_2)] = score

    return wordsim353_pairs_to_scores


# Get cosine similarity of two vectors
def cosine(vec_1: np.ndarray, vec_2: np.ndarray) -> float:
   return float(cosine_similarity([vec_1], [vec_2])[0][0])


# Get Spearman coefficient for a given model (set of hyperparams)
def eval_wordsim353(is_in_vocab: Callable, get_vector: Callable) -> dict[str, float | None]:
   wordsim353_pairs_to_scores = load_wordsim353()

   actual_sims, cos_sims = list(), list()
   num_pairs_in_vocab = 0

   # Loop through each word pair in WordSim-353
   for pair, actual_sim in wordsim353_pairs_to_scores.items():
      word_1, word_2 = pair

      # Check if the pair is present in model's vocab
      if is_in_vocab(word_1) and is_in_vocab(word_2):
         word_1_vec, word_2_vec = get_vector(word_1), get_vector(word_2)

         # Cosine similarity of the two word vectors
         cos_sim = cosine(vec_1=word_1_vec, vec_2=word_2_vec)

         # Update both the actual similarity from WordSim-353 and the cosine similarity
```

```
            actual_sims.append(actual_sim)
            cos_sims.append(cos_sim)

            num_pairs_in_vocab += 1

    if actual_sims:
        spearman_coeff, _ = spearmanr(actual_sims, cos_sims)

    else:
        spearman_coeff = None

    return {"coeff": spearman_coeff, "coverage": num_pairs_in_vocab / len(wordsim353_pairs_to_scores)}
```

Prepare Data

Obtain Data

In [ ]:
```
# Get data from NLTK movie reviews
raw_reviews_with_labels = [
    {
        "tokens": list(movie_reviews.words(fileid)),
        "category": category
    }
    for category in movie_reviews.categories()
    for fileid in movie_reviews.fileids(category)
]

reviews_with_labels_df = pd.DataFrame(raw_reviews_with_labels)
reviews_with_labels_df["category"] = reviews_with_labels_df["category"].apply(lambda cat: 1 if cat == "pos" else 0)
```

In [6]:
```
reviews_with_labels_df
```

Out[6]:

	tokens	category
0	[plot, :, two, teen, couples, go, to, a, churc...	0
1	[the, happy, bastard, ', s, quick, movie, revi...	0
2	[it, is, movies, like, these, that, make, a, j...	0
3	[", quest, for, camelot, ", is, warner, bros, ...	0
4	[synopsis, :, a, mentally, unstable, man, unde...	0
...	...	...
1995	[wow, !, what, a, movie, ., it, ', s, everythi...	1
1996	[richard, gere, can, be, a, commanding, actor,...	1
1997	[glory, --, starring, matthew, broderick, ,, d...	1
1998	[steven, spielberg, ', s, second, epic, film, ...	1
1999	[truman, (, ", true, -, man, ",), burbank, is...	1

2000 rows × 2 columns

Data Processing

In [ ]:

```
def process_raw_data(reviews_with_labels_df: pd.DataFrame) -> None:
  # Get the list of raw tokens for each review
  raw_tokenised_reviews = list(reviews_with_labels_df["tokens"])

  # Process each token list
  stop_words = set(stopwords.words("english"))
  lemmatiser = WordNetLemmatizer()
  processed_tokenised_reviews = list()
  for tokens in raw_tokenised_reviews:
    processed_tokens_in_review = list()

    for token in tokens:
      # Convert to lowercase
```

```
      token = token.lower()

      # Only consider token if it completely consists of alphabets, and is not a stopword
      # Also ignore "br" - may correspond to html <br> tag
      if token.isalpha() and token not in stop_words and token != "br":
        # Lemmatise the word (change to root form)
        token = lemmatiser.lemmatize(token)
        processed_tokens_in_review.append(token)

    processed_tokenised_reviews.append(processed_tokens_in_review)

  # Modify "tokens" column to the processed version
  reviews_with_labels_df["tokens"] = processed_tokenised_reviews
```

In [8]:
```
process_raw_data(reviews_with_labels_df=reviews_with_labels_df)
```

In [9]:
```
reviews_with_labels_df
```

Out[9]:

	tokens	category
0	[plot, two, teen, couple, go, church, party, d...	0
1	[happy, bastard, quick, movie, review, damn, b...	0
2	[movie, like, make, jaded, movie, viewer, than...	0
3	[quest, camelot, warner, bros, first, feature,...	0
4	[synopsis, mentally, unstable, man, undergoing...	0
...	...	...
1995	[wow, movie, everything, movie, funny, dramati...	1
1996	[richard, gere, commanding, actor, always, gre...	1
1997	[glory, starring, matthew, broderick, denzel, ...	1
1998	[steven, spielberg, second, epic, film, world,...	1
1999	[truman, true, man, burbank, perfect, name, ji...	1

2000 rows × 2 columns

Data Exploration

In [10]:

```
# Check word counts
def get_common_words(reviews_with_labels_df: pd.DataFrame, num_words: int) -> tuple[dict[str, int], list[str]]:
  # Count occurrences of words in corpus
  word_counts = dict()
  for review in list(reviews_with_labels_df["tokens"]):
    for word in review:
      if word not in word_counts:
        word_counts[word] = 0
      word_counts[word] += 1

  most_common_words = sorted(word_counts.keys(), key=lambda x: word_counts[x], reverse=True)[:num_words]
  return word_counts, most_common_words
```

In [11]:
```
word_counts, most_common_words = get_common_words(reviews_with_labels_df=reviews_with_labels_df, num_words=50)
print("Most common words:")
for word in most_common_words:
  print(f"{word}: Count = {word_counts[word]}")
```

```
Most common words:
film: Count = 11053
movie: Count = 6977
one: Count = 6028
character: Count = 3879
like: Count = 3789
time: Count = 2979
get: Count = 2814
scene: Count = 2671
make: Count = 2634
even: Count = 2568
good: Count = 2429
story: Count = 2345
would: Count = 2109
much: Count = 2049
also: Count = 1967
well: Count = 1921
life: Count = 1913
two: Count = 1911
see: Count = 1885
way: Count = 1882
first: Count = 1836
go: Count = 1760
year: Count = 1732
thing: Count = 1661
take: Count = 1579
plot: Count = 1574
really: Count = 1558
come: Count = 1510
little: Count = 1505
know: Count = 1494
people: Count = 1470
could: Count = 1427
man: Count = 1404
bad: Count = 1395
work: Count = 1379
never: Count = 1374
director: Count = 1347
best: Count = 1334
end: Count = 1328
performance: Count = 1317
```

```
new: Count = 1292
look: Count = 1278
many: Count = 1268
action: Count = 1260
actor: Count = 1252
u: Count = 1225
love: Count = 1209
play: Count = 1205
star: Count = 1160
role: Count = 1155
```

In [12]:
```
# Conduct train-test split before experimentation (avoid data leakage)
TESTSET_SIZE = 0.2

reviews_train, reviews_test, labels_train, labels_test = train_test_split(reviews_with_labels_df["tokens"], reviews_with_label
reviews_train, reviews_test, labels_train, labels_test = list(reviews_train), list(reviews_test), list(labels_train), list(lab
```

In [13]:
```
# Save processed tokens and most common word list (can be loaded for use in Glove workflow)
with open(os.path.normpath(os.path.join("..", "data", "reviews_train.pkl")), "wb") as f:
  pickle.dump(reviews_train, f)

with open(os.path.normpath(os.path.join("..", "data", "reviews_test.pkl")), "wb") as f:
  pickle.dump(reviews_test, f)

with open(os.path.normpath(os.path.join("..", "data", "most_common_words.pkl")), "wb") as f:
  pickle.dump(most_common_words, f)

# Save labels
with open(os.path.normpath(os.path.join("..", "data", "labels_train.pkl")), "wb") as f:
  pickle.dump(labels_train, f)

with open(os.path.normpath(os.path.join("..", "data", "labels_test.pkl")), "wb") as f:
  pickle.dump(labels_test, f)
```

In [6]:
```
# Select common words for analysis later
TEST_WORDS = ["film", "like", "good", "time", "story", "character", "life", "scene"]
```

Skip-Gram (Word2Vec)

In [15]:
```
WORD2VEC_MIN_COUNT = 1
WORD2VEC_EPOCHS = 20
```

Fixed Set of Parameters

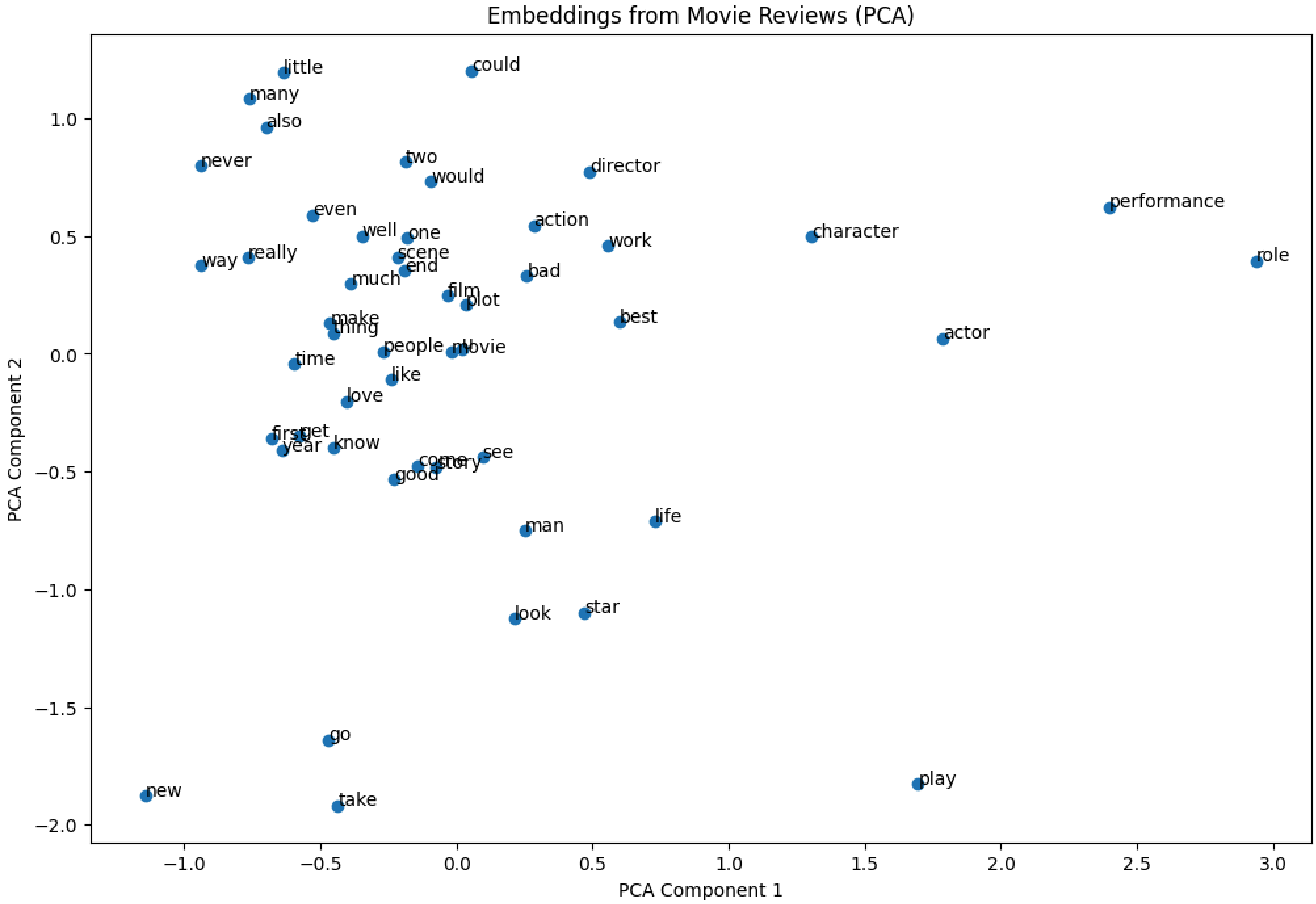
Algorithm

In [16]:
```
WORD2VEC_VECTOR_SIZE = 50
WORD2VEC_WINDOW = 3
WORD2VEC_NEGATIVE = 5
```

In [17]:
```
# Fit the model and get the embeddings
word2vec_model = Word2Vec(sentences=reviews_train, vector_size=WORD2VEC_VECTOR_SIZE, window=WORD2VEC_WINDOW, min_count=WORD2VE
word2vec_embeddings = word2vec_model.wv

# Save model
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "word2vec_model.pkl")), "wb") as f:
    pickle.dump(word2vec_model, f)
```

Visualisation

In [18]:
```
# Only visualise the embeddings of the most common words in the corpus
most_common_word2vec_embeddings = np.array([word2vec_embeddings[word] for word in most_common_words])
visualise_embeddings(embeddings=most_common_word2vec_embeddings, words=most_common_words, filename="word2vec_pca_visualisation
```



Nearest Neighbours

In [19]:
```
# Check nearest words
WORD2VEC_TOPN = 10

for word in TEST_WORDS:
  print(f"{WORD2VEC_TOPN} nearest neighbours to {word}:")
  print(word2vec_model.wv.most_similar(word, topn=WORD2VEC_TOPN))
  print()
```

```
10 nearest neighbours to film:
[('movie', 0.9044820666313171), ('unsatisfactory', 0.8606497645378113), ('expanded', 0.8437156677246094), ('mant', 0.8396050333
976746), ('emphatically', 0.8351638913154602), ('putrid', 0.8332487940788269), ('unmistakable', 0.8327082991600037), ('shoo',
0.8282871842384338), ('voted', 0.8277428150177002), ('quieter', 0.8270278573036194)]

10 nearest neighbours to like:
[('synch', 0.7804538607597351), ('embarrassed', 0.7312526106834412), ('sake', 0.7308354377746582), ('kinship', 0.72693204879760
74), ('xerox', 0.7237459421157837), ('foodstuff', 0.7236295938491821), ('bloodbath', 0.7207787036895752), ('yammering', 0.72048
63429069519), ('humourous', 0.7190214991569519), ('culp', 0.7170007824897766)]

10 nearest neighbours to good:
[('decent', 0.7957882881164551), ('eas', 0.7387107610702515), ('great', 0.7342360019683838), ('terrible', 0.7264418601989746),
('paled', 0.7233241200447083), ('bad', 0.7233129143714905), ('mastering', 0.7232654690742493), ('decieving', 0.717977881431579
6), ('recommends', 0.7157704830169678), ('marketable', 0.7105687260627747)]

10 nearest neighbours to time:
[('booted', 0.7493131160736084), ('waaaaay', 0.7414796948432922), ('heartbreaker', 0.7301492094993591), ('percent', 0.724713027
4772644), ('scarce', 0.7244104743003845), ('metre', 0.7204680442810059), ('slab', 0.7185456156730652), ('wayyyy', 0.71555370092
39197), ('craziest', 0.7141170501708984), ('glowering', 0.7127245664596558)]

10 nearest neighbours to story:
[('parallel', 0.7386681437492371), ('storyline', 0.7369586825370789), ('overlap', 0.7327081561088562), ('analogy', 0.7029218673
706055), ('linear', 0.7024115920066833), ('historical', 0.6973534226417542), ('aftermath', 0.6968250870704651), ('framework',
0.6814592480659485), ('tale', 0.6793993711471558), ('maugham', 0.6793735027313232)]

10 nearest neighbours to character:
[('personality', 0.76336669921875), ('tangential', 0.754920244216919), ('role', 0.7286948561668396), ('incorrectly', 0.72792500
25749207), ('attachment', 0.7243849635124207), ('stereotype', 0.7196766138076782), ('consequently', 0.7184745669364929), ('cari
cature', 0.7180594801902771), ('recaptured', 0.7118464708328247), ('nitwit', 0.7085890173912048)]

10 nearest neighbours to life:
[('comfort', 0.6640452146530151), ('harmony', 0.6545533537864685), ('miracle', 0.6508592963218689), ('live', 0.650464177131652
8), ('sacrificing', 0.6433071494102478), ('bliss', 0.6416030526161194), ('everlasting', 0.6349601745605469), ('seclusion', 0.63
24366927146912), ('scion', 0.6294461488723755), ('ordinary', 0.6265143752098083)]

10 nearest neighbours to scene:
[('moment', 0.7549512386322021), ('sequence', 0.7423784136772156), ('confrontation', 0.6936034560203552), ('straw', 0.693128526
2107849), ('gunfight', 0.6924886703491211), ('gratuitous', 0.6819995641708374), ('unsuspenseful', 0.6769528985023499), ('especi
ally', 0.6742717623710632), ('incomplete', 0.6722208261489868), ('reccurs', 0.6666862964630127)]
```

Trying out Hyperparameter Tuning

Algorithm

In [20]:
```
WORD2VEC_VECTOR_SIZES = [50, 100, 150]
WORD2VEC_WINDOWS = [3, 5, 10]
WORD2VEC_NEGATIVES = [3, 5, 10]
```

In [21]:
```
word2vec_max_spearman_coeff = -10
word2vec_best_vector_size, word2vec_best_window, word2vec_best_negative = None, None, None

# Iterate through each possible set of hyperparameters, finding the best set (metric: Spearman coefficient + WordSim-353)
for vector_size in WORD2VEC_VECTOR_SIZES:
  for window in WORD2VEC_WINDOWS:
    for negative in WORD2VEC_NEGATIVES:
      # Run algorithm
      print(f"Vector size: {vector_size} | Window: {window} | Negative: {negative}")
      word2vec_model_ht = Word2Vec(sentences=reviews_train, vector_size=vector_size, window=window, min_count=WORD2VEC_MIN_COU
      word2vec_embeddings_ht = word2vec_model_ht.wv

      # Evaluate by getting Spearman coefficient using WordSim-353
      eval_output = eval_wordsim353(is_in_vocab=lambda word: word in word2vec_embeddings_ht.key_to_index, get_vector=lambda wo
      spearman_coeff, coverage = eval_output["coeff"], eval_output["coverage"]
      print(f"Spearman coefficient: {spearman_coeff} | Coverage: {coverage}\n")

      if spearman_coeff is not None and spearman_coeff > word2vec_max_spearman_coeff:
        # Best hyperparams so far
        word2vec_max_spearman_coeff = spearman_coeff
        word2vec_best_vector_size, word2vec_best_window, word2vec_best_negative = vector_size, window, negative

        # Save best model
        with open(os.path.normpath(os.path.join("..", "embedding_outputs", "word2vec_model_ht.pkl")), "wb") as f:
          pickle.dump(word2vec_model_ht, f)

print(f"Max Spearman coefficient: {word2vec_max_spearman_coeff} | Best vector size: {word2vec_best_vector_size} | Best window:
```

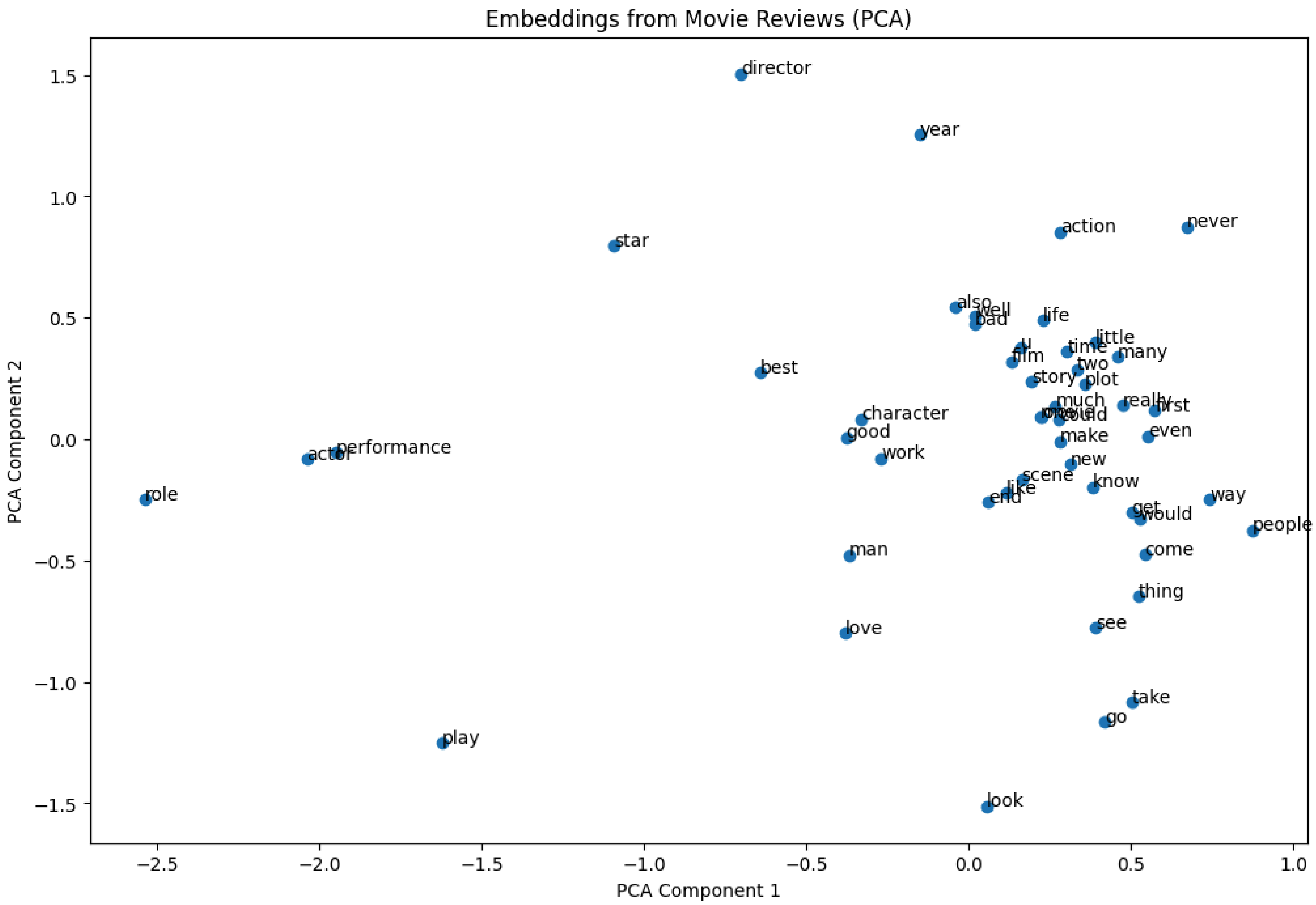
```
Vector size: 50 | Window: 3 | Negative: 3
Spearman coefficient: 0.22995677899214922 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 3 | Negative: 5
Spearman coefficient: 0.22946362369808587 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 3 | Negative: 10
Spearman coefficient: 0.23191778675525399 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 3
Spearman coefficient: 0.2302448331147119 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 5
Spearman coefficient: 0.2354567670299349 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 10
Spearman coefficient: 0.24764815486513364 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 10 | Negative: 3
Spearman coefficient: 0.256373561022586 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 10 | Negative: 5
Spearman coefficient: 0.265381629360782 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 10 | Negative: 10
Spearman coefficient: 0.2711862621113427 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 3
Spearman coefficient: 0.22511502262104022 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 5
Spearman coefficient: 0.2156807834967634 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 10
Spearman coefficient: 0.27634199540222126 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 5 | Negative: 3
Spearman coefficient: 0.23679666457196324 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 5 | Negative: 5
Spearman coefficient: 0.2656593143644619 | Coverage: 0.8746438746438746
```

```
Vector size: 100 | Window: 5 | Negative: 10
Spearman coefficient: 0.28730360573792646 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 3
Spearman coefficient: 0.2723782960181031 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 5
Spearman coefficient: 0.2815045723115624 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 10
Spearman coefficient: 0.2920147112111129 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 3
Spearman coefficient: 0.22358827355674485 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 5
Spearman coefficient: 0.22673073872534957 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 10
Spearman coefficient: 0.24850443670247074 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 3
Spearman coefficient: 0.22605487955657225 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 5
Spearman coefficient: 0.2760779976354666 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 10
Spearman coefficient: 0.25915974326637964 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 3
Spearman coefficient: 0.2808979788569218 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 5
Spearman coefficient: 0.2951063676971921 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 10
Spearman coefficient: 0.28520095581088334 | Coverage: 0.8746438746438746

Max Spearman coefficient: 0.2951063676971921 | Best vector size: 150 | Best window: 10 | Best negative: 5
```

In [22]:
```
# Load best model and embeddings
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "word2vec_model_ht.pkl")), "rb") as f:
  word2vec_model_ht = pickle.load(f)

word2vec_embeddings_ht = word2vec_model_ht.wv
```

Visualisation

In [23]:
```
# Only visualise the embeddings of the most common words in the corpus
most_common_word2vec_embeddings_ht = np.array([word2vec_embeddings_ht[word] for word in most_common_words])
visualise_embeddings(embeddings=most_common_word2vec_embeddings_ht, words=most_common_words, filename="word2vec_ht_pca_visuali
```



Nearest Neighbours

In [24]:
```
# Check nearest words
WORD2VEC_TOPN = 10

for word in TEST_WORDS:
  print(f"{WORD2VEC_TOPN} nearest neighbours to {word}:")
  print(word2vec_model_ht.wv.most_similar(word, topn=WORD2VEC_TOPN))
  print()
```

```
10 nearest neighbours to film:
[('movie', 0.8113678097724915), ('rejuvenates', 0.7720246315002441), ('godforsaken', 0.7709106206893921), ('unconventionality',
0.7676049470901489), ('horor', 0.7661639451980591), ('circled', 0.7618758082389832), ('chequered', 0.7570934891700745), ('voi
l', 0.7546131610870361), ('unsentimental', 0.7523300051689148), ('kafkaism', 0.75039142370224)]

10 nearest neighbours to like:
[('ewwwww', 0.6563647985458374), ('unflushed', 0.6562364101409912), ('interferred', 0.6544751524925232), ('glisten', 0.65132325
88768005), ('cagney', 0.648890495300293), ('anothergreat', 0.6481334567070007), ('prefered', 0.6474140882492065), ('peacenik',
0.6471408009529114), ('shalit', 0.644629180431366), ('amateurism', 0.6434406638145447)]

10 nearest neighbours to good:
[('expended', 0.6557424664497375), ('commensurate', 0.6325111985206604), ('imaganitive', 0.6225097179412842), ('bregman', 0.618
292510509491), ('faulted', 0.6158546209335327), ('decieving', 0.6107174158096313), ('extremel', 0.6106644868850708), ('rekindli
ng', 0.6048842072486877), ('pitifully', 0.6048626899719238), ('stammer', 0.5995452404022217)]

10 nearest neighbours to time:
[('wayyyy', 0.6849040985107422), ('bullshitting', 0.6718259453773499), ('rewound', 0.646697461605072), ('buddying', 0.642084658
1459045), ('lamanna', 0.6403175592422485), ('slab', 0.6367695927619934), ('glossing', 0.6279914379119873), ('replayed', 0.62275
66003799438), ('seperation', 0.6183134913444519), ('sidestep', 0.6159421801567078)]

10 nearest neighbours to story:
[('ascribe', 0.6475194096565247), ('comprehendably', 0.6337887644767761), ('brining', 0.5999462008476257), ('unsurprising', 0.5
978795886039734), ('tangentially', 0.5937677025794983), ('overdramaticizes', 0.5931387543678284), ('incorporation', 0.592058777
8091431), ('unmotivated', 0.5821845531463623), ('structuring', 0.5754712224006653), ('realisticaly', 0.5741592645645142)]

10 nearest neighbours to character:
[('tangential', 0.6973575949668884), ('logistical', 0.6929374933242798), ('recaptured', 0.690814197063446), ('unfetching', 0.68
60516667366028), ('thorougly', 0.6798093318939209), ('ungloriously', 0.6606026291847229), ('unplayable', 0.659747838973999),
('murkily', 0.6592525243759155), ('muddling', 0.6538572311401367), ('uncountable', 0.6519955992698669)]

10 nearest neighbours to life:
[('fullest', 0.5262770652770996), ('bottlecap', 0.5251275897026062), ('predetermined', 0.5235081315040588), ('overtaken', 0.521
6379165649414), ('touchingly', 0.5184016227722168), ('disapproving', 0.5163471102714539), ('quicksand', 0.5134742856025696),
('disatisfaction', 0.5102584362030029), ('swirl', 0.5080655813217163), ('privelege', 0.5065743923187256)]

10 nearest neighbours to scene:
[('impart', 0.6290867924690247), ('storyboarded', 0.6162256002426147), ('choppily', 0.5996971130371094), ('humping', 0.59938657
28378296), ('unusable', 0.5969494581222534), ('bebe', 0.5927467942237854), ('bungled', 0.590409517288208), ('caffeinated', 0.58
6901068687439), ('spacewalk', 0.5847108960151672), ('fiftieth', 0.5833877325057983)]
```

SPPMI-SVD

In [25]:
```
### Functions for Algorithm ###
def get_co_occurrence_matrix(reviews: list[list[str]], window_size: int) -> tuple[np.ndarray, dict[str, int]]:
  # Get all unique words in the corpus, mapped to a unique int id
  vocab = {word: id for id, word in enumerate(set(word for review in reviews for word in review))}

  # Initialise co-occurrence matrix
  vocab_size = len(vocab)
  co_occurrence_matrix = np.zeros((vocab_size, vocab_size), dtype=np.float32)

  # Populate the co-occurrence matrix with the window
  for review in reviews:
      review_length = len(review)
      for current_word_index, current_word in enumerate(review):
          current_word_id = vocab[current_word]

          # Find the endpoints of the context window (using window size)
          start = max(0, current_word_index - window_size)
          end = min(review_length, current_word_index + window_size + 1)

          # Update co-occurrence counts for words in the window
          for context_word_index in range(start, end):
              if current_word_index != context_word_index:  # Skip the word itself
                  context_word_id = vocab[review[context_word_index]]
                  co_occurrence_matrix[current_word_id, context_word_id] += 1

  return co_occurrence_matrix, vocab


def conduct_sppmi_svd(reviews: list[list[str]], window_size: int, negative: int, vector_size: int, seed: int = seed) -> tuple[
  # Get co-occurence matrix
  co_occurrence_matrix, vocab = get_co_occurrence_matrix(reviews=reviews, window_size=window_size)
  print("Co-occurrence matrix populated!")

  # Initialise SPPMI matrix
  sppmi_matrix = np.zeros_like(co_occurrence_matrix)
```

```
  # Populate SPPMI matrix
  # Find indices where entries of co-occurrence matrix are positive
  row_indices_non_zero, col_indices_non_zero = np.nonzero(co_occurrence_matrix)

  co_occurrence_matrix_sum = np.sum(co_occurrence_matrix)
  marginal_probs = np.sum(co_occurrence_matrix, axis=1) / co_occurrence_matrix_sum
  for i, j in zip(row_indices_non_zero, col_indices_non_zero):
    if co_occurrence_matrix[i, j] > 0:
        pmi = np.log((co_occurrence_matrix[i, j] / co_occurrence_matrix_sum) / (marginal_probs[i] * marginal_probs[j]))
        sppmi_matrix[i, j] = max(pmi - np.log(negative), 0)

  print("SPPMI matrix populated!")

  # Apply SVD on the SPPMI matrix
  svd = TruncatedSVD(n_components=vector_size, random_state=seed)
  sppmi_svd_embeddings = svd.fit_transform(sppmi_matrix)

  print("SPPMI-SVD embeddings produced!")

  return sppmi_svd_embeddings, vocab
```

In [3]:
```
### Nearest Neighbours ###
def sppmi_svd_nearest_neighbours(embeddings: np.ndarray, words_to_ids: dict[str, int], target_word: str, topn: int) -> list[tu
  # Get embedding of target word
  if target_word not in words_to_ids:
    raise KeyError(f"word {target_word} not in vocabulary")

  target_word_id = words_to_ids[target_word]
  target_word_embedding = np.array([embeddings[target_word_id]])

  # Array of cosine similarities to target word (each element is itself an array of length 1)
  similarities = cosine_similarity(embeddings, target_word_embedding)

  # Organise similarity of each word to the target word
  ids_to_words = {id: word for word, id in words_to_ids.items()}
  word_similarity_pairs = list()
  for i, packed_similarity in enumerate(similarities):
    # Skip the target word itself (don't need to consider similarity with itself)
    if i != target_word_id:
      word = ids_to_words[i]
```

```
        word_similarity_pairs.append((word, packed_similarity.item()))

    # Sort in order of decreasing similarity
    word_similarity_pairs.sort(key=lambda pair: pair[1], reverse=True)

    return word_similarity_pairs[:topn]
```

Fixed Set of Hyperparameters

Algorithm

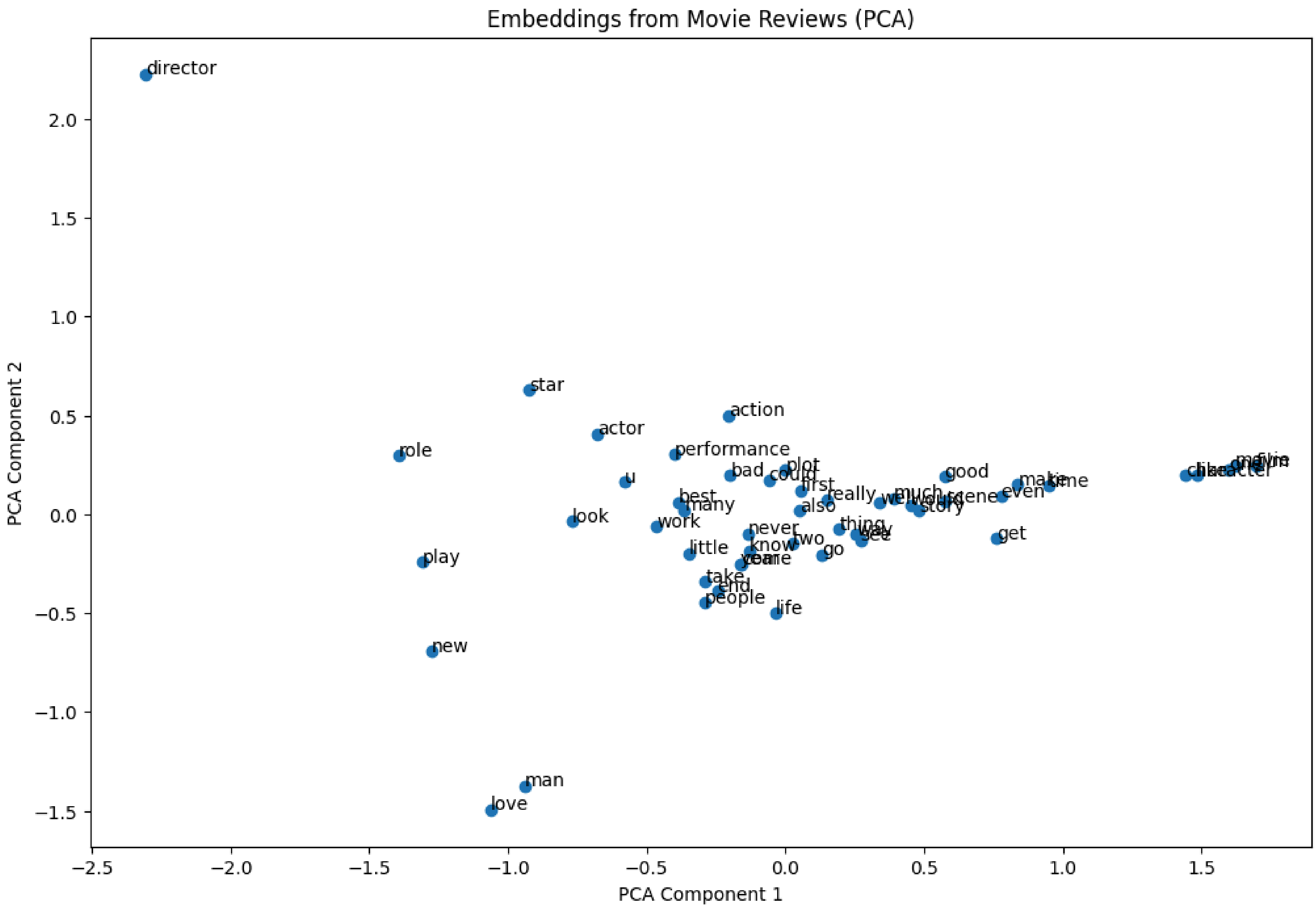
In [27]:
```
SPPMI_SVD_VECTOR_SIZE = 50
SPPMI_SVD_WINDOW = 3
SPPMI_SVD_NEGATIVE = 5
```

In [28]:
```
# Conduct the algorithm
sppmi_svd_embeddings, sppmi_svd_words_to_ids = conduct_sppmi_svd(reviews=reviews_train, window_size=window, negative=negative,

# Save best embeddings and mapping
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings.pkl")), "wb") as f:
    pickle.dump(sppmi_svd_embeddings, f)

with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_mapping.pkl")), "wb") as f:
    pickle.dump(sppmi_svd_words_to_ids, f)
```

```
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
```

Visualisation

In [29]:
```
# Only visualise the embeddings of the most common words in the corpus
most_common_sppmi_svd_embeddings = np.array([sppmi_svd_embeddings[sppmi_svd_words_to_ids[word]] for word in most_common_words]
visualise_embeddings(embeddings=most_common_sppmi_svd_embeddings, words=most_common_words, filename="sppmi_svd_pca_visualisati
```



Nearest Neighbours

In [9]:
```
# Check nearest neighbours
SPPMI_SVD_TOPN = 10

for word in TEST_WORDS:
  print(f"{SPPMI_SVD_TOPN} nearest neighbours to {word}:")
  print(sppmi_svd_nearest_neighbours(embeddings=sppmi_svd_embeddings, words_to_ids=sppmi_svd_words_to_ids, target_word=word, t
  print()
```

```
10 nearest neighbours to film:
[('interferred', 0.6822568774223328), ('aspired', 0.6196185946464539), ('script', 0.5980833172798157), ('movie', 0.588425517082
2144), ('look', 0.583650529384613), ('predictable', 0.5759027004241943), ('anything', 0.57457035779953), ('thriller', 0.5741803
646087646), ('often', 0.5649411678314209), ('could', 0.5646660923957825)]

10 nearest neighbours to like:
[('come', 0.7138094902038574), ('think', 0.7136224508285522), ('watching', 0.711866557598114), ('get', 0.6969590187072754), ('s
ee', 0.6953395009040833), ('every', 0.6942891478538513), ('everyone', 0.6931802034378052), ('time', 0.692459225654602), ('wel
l', 0.6812885403633118), ('something', 0.6792770624160767)]

10 nearest neighbours to good:
[('time', 0.8700373768806458), ('bad', 0.8630803823471069), ('much', 0.858633279800415), ('little', 0.8566007614135742), ('mak
e', 0.848146378993988), ('even', 0.8455625772476196), ('really', 0.8454145193099976), ('better', 0.8443527817726135), ('know',
0.8398519158363342), ('could', 0.8374507427215576)]

10 nearest neighbours to time:
[('much', 0.8910382390022278), ('even', 0.8879128694534302), ('really', 0.8863357305526733), ('better', 0.8859239816665649),
('could', 0.8827092051506042), ('make', 0.8755542039871216), ('come', 0.8754130005836487), ('go', 0.8733460307121277), ('know',
0.8720604777336121), ('plot', 0.8716142773628235)]

10 nearest neighbours to story:
[('time', 0.8397431373596191), ('little', 0.8229535222053528), ('made', 0.8187458515167236), ('interesting', 0.815908432006835
9), ('give', 0.8137189149856567), ('way', 0.8099150061607361), ('although', 0.8090178966522217), ('almost', 0.808367252349853
5), ('work', 0.8080711960792542), ('actually', 0.8025341033935547)]

10 nearest neighbours to character:
[('never', 0.7811121940612793), ('even', 0.7702271938323975), ('good', 0.7698503732681274), ('interesting', 0.768769204616546
6), ('really', 0.7667884230613708), ('could', 0.7643242478370667), ('much', 0.7642346620559692), ('many', 0.7620941400527954),
('although', 0.7619799375534058), ('feel', 0.757488489151001)]

10 nearest neighbours to life:
[('story', 0.7857402563095093), ('come', 0.7829723358154297), ('actually', 0.7781126499176025), ('much', 0.7751485109329224),
('even', 0.7749313116073608), ('feel', 0.7746948599815369), ('year', 0.7727004885673523), ('never', 0.7722012996673584), ('begi
n', 0.7709779739379883), ('something', 0.7697187662124634)]

10 nearest neighbours to scene:
[('much', 0.8583980202674866), ('never', 0.8533055782318115), ('time', 0.8424951434135437), ('audience', 0.8417986035346985),
('enough', 0.8408147096633911), ('make', 0.8341929316520691), ('know', 0.8302638530731201), ('end', 0.8236851096153259), ('wel
l', 0.8223975300788879), ('really', 0.8202391862869263)]
```

Trying out Hyperparameter Tuning

Algorithm

In [31]:
```
SPPMI_SVD_NEGATIVES = [3, 5, 10]
SPPMI_SVD_WINDOWS = [3, 5, 10]
SPPMI_SVD_VECTOR_SIZES = [50, 100, 150]
```

In [32]:
```
sppmi_svd_max_spearman_coeff = -10
sppmi_svd_best_vector_size, sppmi_svd_best_window, sppmi_svd_best_negative = None, None, None

# Iterate through each possible set of hyperparameters, finding the best set (metric: Spearman coefficient + WordSim-353)
for vector_size in SPPMI_SVD_VECTOR_SIZES:
  for window in SPPMI_SVD_WINDOWS:
    for negative in SPPMI_SVD_NEGATIVES:
      # Run algorithm
      print(f"Vector size: {vector_size} | Window: {window} | Negative: {negative}")
      sppmi_svd_embeddings_ht, sppmi_svd_words_to_ids_ht = conduct_sppmi_svd(reviews=reviews_train, window_size=window, negati

      # Evaluate by getting Spearman coefficient using WordSim-353
      eval_output = eval_wordsim353(is_in_vocab=lambda word: word in sppmi_svd_words_to_ids_ht, get_vector=lambda word: sppmi_
      spearman_coeff, coverage = eval_output["coeff"], eval_output["coverage"]
      print(f"Spearman coefficient: {spearman_coeff} | Coverage: {coverage}\n")

      if spearman_coeff is not None and spearman_coeff > sppmi_svd_max_spearman_coeff:
        # Best hyperparams so far
        sppmi_svd_max_spearman_coeff = spearman_coeff
        sppmi_svd_best_vector_size, sppmi_svd_best_window, sppmi_svd_best_negative = vector_size, window, negative

        # Save best embeddings and mapping
        with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_ht.pkl")), "wb") as f:
          pickle.dump(sppmi_svd_embeddings_ht, f)

        with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_mapping_ht.pkl")), "wb") as f
          pickle.dump(sppmi_svd_words_to_ids_ht, f)

print(f"Max Spearman coefficient: {sppmi_svd_max_spearman_coeff} | Best vector size: {sppmi_svd_best_vector_size} | Best windo
```

```
Vector size: 50 | Window: 3 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.18507943985000597 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 3 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.21158995871514277 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 3 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.18540233421201466 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.17981752678176816 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.18508980896888874 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 5 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.12720441018834172 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 10 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.09439506636669258 | Coverage: 0.8746438746438746
```

```
Vector size: 50 | Window: 10 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.08425738621741033 | Coverage: 0.8746438746438746

Vector size: 50 | Window: 10 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.0347888086163652 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.21328655394673754 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.21610467307669 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 3 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.218414705381388 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 5 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.1957461524446859 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 5 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
```

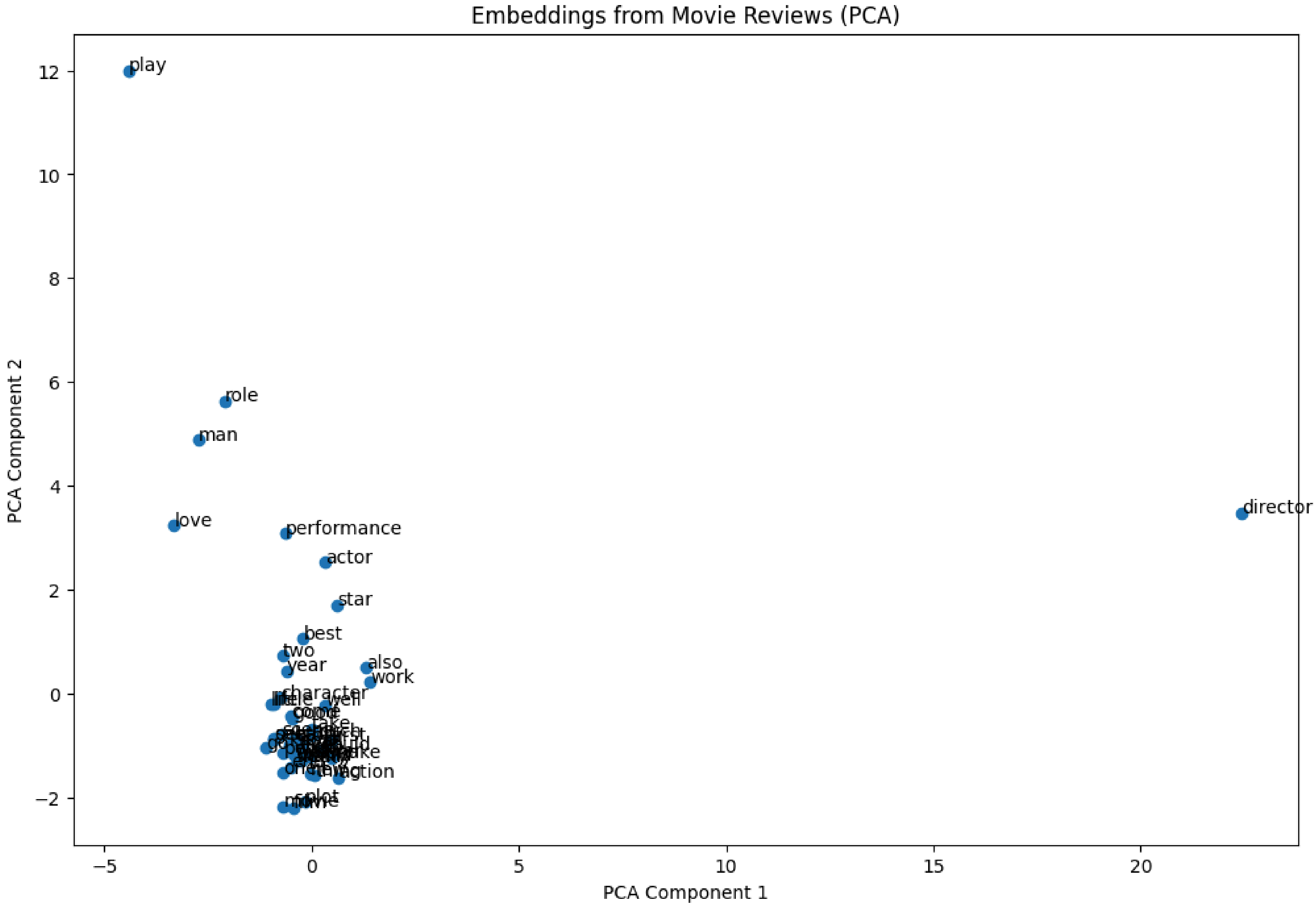
```
Spearman coefficient: 0.20147571277453555 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 5 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.1799915205966206 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.1483020416143079 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.11790704343331715 | Coverage: 0.8746438746438746

Vector size: 100 | Window: 10 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.055129494363891 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.2790860790233509 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.2840080923746124 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 3 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
```

```
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.2526471070785051 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.20907793597460894 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.22061752099683835 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 5 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.186847996766647 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 3
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.1937200266149975 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 5
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.18003672995494938 | Coverage: 0.8746438746438746

Vector size: 150 | Window: 10 | Negative: 10
Co-occurrence matrix populated!
SPPMI matrix populated!
SPPMI-SVD embeddings produced!
Spearman coefficient: 0.1486962755142299 | Coverage: 0.8746438746438746

Max Spearman coefficient: 0.2840080923746124 | Best vector size: 150 | Best window: 3 | Best negative: 5
```

In [33]:
```
# Load best embeddings and mapping
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_ht.pkl")), "rb") as f:
  sppmi_svd_embeddings_ht = pickle.load(f)

with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_mapping_ht.pkl")), "rb") as f:
  sppmi_svd_words_to_ids_ht = pickle.load(f)
```

Visualisation

In [34]:
```
# Only visualise the embeddings of the most common words in the corpus
most_common_sppmi_svd_embeddings_ht = np.array([sppmi_svd_embeddings_ht[sppmi_svd_words_to_ids_ht[word]] for word in most_comm
visualise_embeddings(embeddings=most_common_sppmi_svd_embeddings_ht, words=most_common_words, filename="sppmi_svd_ht_pca_visua
```



Nearest Neighbours

In [11]:
```
# Check nearest neighbours
SPPMI_SVD_TOPN = 10

for word in TEST_WORDS:
  print(f"{SPPMI_SVD_TOPN} nearest neighbours to {word}:")
  print(sppmi_svd_nearest_neighbours(embeddings=sppmi_svd_embeddings_ht, words_to_ids=sppmi_svd_words_to_ids_ht, target_word=w
  print()
```

```
10 nearest neighbours to film:
[('movie', 0.8806698322296143), ('made', 0.8380125761032104), ('many', 0.8197451829910278), ('could', 0.8094170093536377), ('mu
ch', 0.8029585480690002), ('one', 0.7991598844528198), ('even', 0.7987421154975891), ('really', 0.7939698696136475), ('would',
0.7869608998298645), ('first', 0.7846642136573792)]

10 nearest neighbours to like:
[('one', 0.7953758835792542), ('really', 0.7865555286407471), ('know', 0.779260516166687), ('look', 0.7718312740325928), ('eve
n', 0.7717165350914001), ('thing', 0.7710414528846741), ('bad', 0.745314359664917), ('movie', 0.7440394163131714), ('good', 0.7
402405142784119), ('see', 0.7375472784042358)]

10 nearest neighbours to good:
[('well', 0.8275624513626099), ('one', 0.8185980319976807), ('really', 0.8182766437530518), ('time', 0.789311945438385), ('muc
h', 0.7866829037666321), ('bad', 0.786185622215271), ('make', 0.7831035852432251), ('even', 0.77897709608078), ('see', 0.776796
5197563171), ('movie', 0.7681609392166138)]

10 nearest neighbours to time:
[('see', 0.8143904209136963), ('one', 0.8098751902580261), ('know', 0.8082817196846008), ('much', 0.8021175861358643), ('even',
0.8020084500312805), ('really', 0.8010701537132263), ('good', 0.789311945438385), ('first', 0.7764158248901367), ('could', 0.77
61391401290894), ('thing', 0.769415557384491)]

10 nearest neighbours to story:
[('plot', 0.7214222550392151), ('character', 0.7095595598220825), ('many', 0.7037534117698669), ('time', 0.7007693648338318),
('however', 0.7006839513778687), ('much', 0.6973422169685364), ('film', 0.6890939474105835), ('really', 0.680280864238739), ('s
ee', 0.6790071725845337), ('even', 0.6762250065803528)]

10 nearest neighbours to character:
[('much', 0.7573550343513489), ('even', 0.754587709903717), ('however', 0.7533665895462036), ('good', 0.7513739466667175), ('on
e', 0.7449087500572205), ('really', 0.7445036768913269), ('many', 0.7418821454048157), ('also', 0.7351376414299011), ('well',
0.728933572769165), ('seems', 0.7283487319946289)]

10 nearest neighbours to life:
[('people', 0.7202901840209961), ('even', 0.6862643361091614), ('real', 0.6843535900115967), ('one', 0.6812009811401367), ('tim
e', 0.6762433648109436), ('know', 0.6751324534416199), ('however', 0.6720420122146606), ('much', 0.6704410910606384), ('come',
0.662053644657135), ('really', 0.6601817011833191)]

10 nearest neighbours to scene:
[('one', 0.7619084715843201), ('see', 0.7527258396148682), ('even', 0.7184852957725525), ('well', 0.7120426297187805), ('film',
0.7034809589385986), ('many', 0.7008957266807556), ('movie', 0.6948351860046387), ('sequence', 0.6934535503387451), ('much', 0.
6905617117881775), ('really', 0.6856517195701599)]
```

GloVe Experimentation

Chia Bing Xuan

2025-08-21

Imports

```
library(dplyr)
library(ggplot2)
library(lsa)
library(psych)
library(readr)
library(reticulate)
library(text2vec)
```

Setup

```
pickle <- import("pickle")
py_builtin <- import_builtins()
set.seed(42)
```

Helpers

Main Function

```r
# Main GloVe function
conduct_glove <- function(reviews_train, vector_size, window, negative, min_count, epochs) {
  # Create an iterator over the tokens
  it <- itoken(reviews_train, progressbar = FALSE)

  # Build the vocabulary
  vocab <- create_vocabulary(it)

  # Prune the vocabulary - remove infrequent or frequent terms
  vocab <- prune_vocabulary(vocab, term_count_min = min_count)

  # Create a term-co-occurrence matrix
  tcm <- create_tcm(it, vectorizer = vocab_vectorizer(vocab), skip_grams_window = window)

  # Define the GloVe model
  glove <- GlobalVectors$new(rank = vector_size, x_max = 10)  # rank = embedding dimensions

  # Fit the GloVe model
  word_vectors <- glove$fit_transform(tcm, n_iter = epochs, convergence_tol = 0.01)

  # Combine word and context embeddings (optional)
  word_vectors <- word_vectors + t(glove$components)

  return (word_vectors)
}
```

Visualisation

```
# Function to make PCA plot
visualise_embeddings <- function(word_vectors, filename) {
  # Perform PCA to reduce dimensions to 2D
  pca <- prcomp(word_vectors, center = TRUE, scale. = TRUE)
  word_vectors_pca <- data.frame(pca$x[, 1:2])
  word_vectors_pca$word <- rownames(word_vectors)

  # Plot the embeddings
  p <- ggplot(word_vectors_pca, aes(x = PC1, y = PC2, label = word)) +
    geom_point() +
    geom_text(aes(label = word), hjust = 0, vjust = 1, size = 3) +
    theme_minimal() +
    labs(title = "Embeddings from Movie Reviews (PCA)", x = "PCA Component 1", y = "PCA Component 2")
  print(p)

  # Save plot
  ggsave(paste("../embedding_plots/", filename, ".png", sep=""), plot = p, width = 6, height = 4, dpi = 300, bg="white")
}
```

Nearest Neighbours Analysis

```
# Nearest neighbours analysis
find_nearest_neighbours <- function(embeddings, word, topn) {
  if (!(word %in% rownames(embeddings))) {
    stop(paste("word", word, "not in vocabulary"))
  }

  target_word_embedding_mat <- embeddings[word, , drop = FALSE]
  cos_sim <- sim2(x = embeddings[rownames(embeddings) != word, , drop = FALSE], y = target_word_embedding_mat, method = "cos
ine", norm = "l2")
  return (head(sort(cos_sim[, 1], decreasing = TRUE), topn))
}
```

WordSim-353 with Spearman Coefficient (For Hyperparameter Tuning)

```
# Load WordSim-353 dataset
load_wordsim353 <- function(path = "../data/wordsim353crowd.csv") {
  wordsim_data <- read_csv(path, show_col_types = FALSE)

  # Normalise case + store in sorted pairs
  wordsim_data <- wordsim_data %>%
    mutate(
      Word1 = tolower(`Word 1`),
      Word2 = tolower(`Word 2`),
      Pair = apply(cbind(Word1, Word2), 1, function(x) paste(sort(x), collapse = "_"))
    )

  # Create named vector mapping (word1_word2) -> human similarity
  scores <- wordsim_data$`Human (Mean)`
  names(scores) <- wordsim_data$Pair
  return (scores)
}

# Get cosine similarity of two vectors
cosine_sim <- function(vec1, vec2) {
  return (cosine(vec1, vec2)[1, 1])  # lsa::cosine returns a matrix
}

# Get Spearman coefficient for a given model (set of hyperparams)
eval_wordsim353 <- function(is_in_vocab, get_vector, wordsim_scores) {
  actual_sims <- c()
  cos_sims <- c()
  num_pairs_in_vocab <- 0

  for (pair in names(wordsim_scores)) {
    words <- unlist(strsplit(pair, "_"))
    w1 <- words[1]
    w2 <- words[2]

    if (is_in_vocab(w1) && is_in_vocab(w2)) {
      v1 <- get_vector(w1)
```

```
      v2 <- get_vector(w2)

      cos_sim <- cosine_sim(v1, v2)

      actual_sims <- c(actual_sims, ws_scores[[pair]])
      cos_sims <- c(cos_sims, cos_sim)

      num_pairs_in_vocab <- num_pairs_in_vocab + 1
    }
  }

  if (length(actual_sims) > 0) {
    spearman_coeff <- cor(actual_sims, cos_sims, method = "spearman")
  } else {
    spearman_coeff <- NA
  }

  return (
    list(
      coeff = spearman_coeff,
      coverage = num_pairs_in_vocab / length(wordsim_scores)
    )
  )
}
```

Load Processed Data

```
load_data <- function() {
  with(py_builtin$open("../data/reviews_train.pkl", "rb") %as% f, {
    reviews_train_py <- pickle$load(f)
  })

  with(py_builtin$open("../data/most_common_words.pkl", "rb") %as% f, {
    most_common_words_py <- pickle$load(f)
  })

  reviews_train <- py_to_r(reviews_train_py)
  most_common_words <- unlist(py_to_r(most_common_words_py))

  return (list(reviews_train = reviews_train, most_common_words = most_common_words))
}
```

```
loaded_data <- load_data()
reviews_train <- loaded_data$reviews_train
most_common_words <- loaded_data$most_common_words
```

```
most_common_words
TEST_WORDS <- c("film", "like", "good", "time", "story", "character", "life", "scene")
```

Setting General Configs

```
GLOVE_MIN_COUNT <- 1
GLOVE_EPOCHS <- 20
```

Fixed Set of Parameters

Algorithm

```
GLOVE_VECTOR_SIZE <- 50
GLOVE_WINDOW <- 3
GLOVE_NEGATIVE <- 5
```

```
# Fit model
glove_embeddings <- conduct_glove(reviews_train = reviews_train, vector_size = GLOVE_VECTOR_SIZE, window = GLOVE_WINDOW, negative = GLOVE_NEGATIVE, min_count = GLOVE_MIN_COUNT, epochs = GLOVE_EPOCHS)

# Save embeddings
saveRDS(glove_embeddings, file = "../embedding_outputs/glove_embeddings.rds")
```

Visualisation

```
most_common_glove_embeddings <- glove_embeddings[most_common_words,]
visualise_embeddings(word_vectors = most_common_glove_embeddings, filename = "glove_pca_visualisation")
```

Nearest Neighbours

```
GLOVE_TOPN <- 10
for (word in TEST_WORDS) {
  cat(GLOVE_TOPN, " nearest neighbours to ", word, ":\n", sep="")
  print(find_nearest_neighbours(embeddings = glove_embeddings, word = word, topn = GLOVE_TOPN))
  cat("\n")
}
```

Trying out Hyperparameter Tuning

Algorithm

```
GLOVE_VECTOR_SIZES <- c(50, 100, 150)
GLOVE_WINDOWS <- c(3, 5, 10)
GLOVE_NEGATIVES <- c(3, 5, 10)
```

```
glove_max_spearman_coeff <- -10
glove_best_vector_size <- NA
glove_best_window <- NA
glove_best_negative <- NA

ws_scores <- load_wordsim353("../data/wordsim353crowd.csv")

for (vector_size in GLOVE_VECTOR_SIZES) {
  for (window in GLOVE_WINDOWS) {
    for (negative in GLOVE_NEGATIVES) {
      # Run algorithm
      cat(sprintf("Vector size: %d | Window: %d | Negative: %d\n", vector_size, window, negative))
      glove_embeddings_ht <- conduct_glove(reviews_train = reviews_train, vector_size = vector_size, window = window, negati
ve = negative, min_count = GLOVE_MIN_COUNT, epochs = GLOVE_EPOCHS)

      # Evaluate by getting Spearman coefficient using WordSim-353
      is_in_vocab <- function(w) w %in% rownames(glove_embeddings_ht)
      get_vector  <- function(w) glove_embeddings_ht[w, ]
      eval_output <- eval_wordsim353(is_in_vocab, get_vector, ws_scores)
      spearman_coeff <- eval_output$coeff
      coverage <- eval_output$coverage
      cat(sprintf("Spearman coefficient: %.4f | Coverage: %.2f\n\n", spearman_coeff, coverage))

      if (!is.na(spearman_coeff) && spearman_coeff > glove_max_spearman_coeff) {
        glove_max_spearman_coeff <- spearman_coeff
        glove_best_vector_size <- vector_size
        glove_best_window <- window
        glove_best_negative <- negative

        # Save best embeddings
        saveRDS(glove_embeddings_ht, file = "../embedding_outputs/glove_embeddings_ht.rds")
      }
    }
  }
}
```

```
cat(sprintf("Max Spearman coefficient: %.4f | Best vector size: %d | Best window: %d | Best negative: %d\n", glove_max_spear
man_coeff, glove_best_vector_size, glove_best_window, glove_best_negative))
```

```
# Load best embeddings
glove_embeddings_ht <- readRDS(file = "../embedding_outputs/glove_embeddings_ht.rds")
```

Visualisation

```
most_common_glove_embeddings_ht <- glove_embeddings_ht[most_common_words,]
visualise_embeddings(word_vectors = most_common_glove_embeddings_ht, filename = "glove_ht_pca_visualisation")
```

Nearest Neighbours

```
GLOVE_TOPN <- 10
for (word in TEST_WORDS) {
  cat(GLOVE_TOPN, " nearest neighbours to ", word, ":\n", sep="")
  print(find_nearest_neighbours(embeddings = glove_embeddings_ht, word = word, topn = GLOVE_TOPN))
  cat("\n")
}
```

Imports

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
import pickle
import pyreadr
import seaborn as sns
from sklearn.base import ClassifierMixin
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score, precision_score, recall_score
from sklearn.svm import SVC
from xgboost import XGBClassifier
```

Setup

In [2]:
```
# Set seed for reproducibility
seed = 42

# Create folders
os.makedirs(os.path.normpath(os.path.join("..", "sentiment_analysis_outputs")), exist_ok=True)
os.makedirs(os.path.normpath(os.path.join("..", "sentiment_analysis_eval_results")), exist_ok=True)
```

Helpers

In [3]:
```
def make_embeddings(words_to_embeddings_mapping: dict[str, np.ndarray], reviews: list[list[str]]) -> np.ndarray:
    # Initialise final feature array
    num_reviews = len(reviews)
    vector_size = len(list(words_to_embeddings_mapping.values())[0])
    feature_vectors = np.zeros((num_reviews, vector_size))

    for i, review in enumerate(reviews):
        aggregated_feature_vector = np.zeros(vector_size)
        num_words_in_aggregation = 0
```

```
        for word in review:
            # We only aggregate for those words that exist in the vocab
            if word in words_to_embeddings_mapping:
                # Add the word embedding
                word_embedding = words_to_embeddings_mapping[word]
                aggregated_feature_vector += word_embedding
                num_words_in_aggregation += 1

        # To aggregate, we average the word embeddings
        # If no words added to aggregation, just take the aggregation to be a zero vector
        if num_words_in_aggregation != 0:
            aggregated_feature_vector /= num_words_in_aggregation

        feature_vectors[i] = aggregated_feature_vector

    return feature_vectors
```

In [4]:

```
def display_and_save_cm(cm: np.ndarray, filename: str) -> None:
    # Make heat map of confusion matrix
    plt.figure(figsize=(8, 6))
    sns.set_theme(font_scale=0.8)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")

    # Save this heat map
    cm_file_path = os.path.join(os.path.normpath(os.path.join("..", "sentiment_analysis_eval_results")), f"{filename}_cm.png")
    plt.savefig(cm_file_path, bbox_inches="tight")

    # Display heat map
    print("Confusion Matrix:")
    plt.show()


def train_and_eval_classifier(model: ClassifierMixin, X_train: np.ndarray, y_train: np.ndarray, X_test: np.ndarray, y_test: np
    # Train the model
    model.fit(X_train, y_train)

    # Predict on test set
    y_preds = model.predict(X_test)
```

```
    # Calculate evaluation metrics
    acc = accuracy_score(y_test, y_preds)
    precision = precision_score(y_test, y_preds, pos_label=1)
    recall = recall_score(y_test, y_preds, pos_label=1)
    f1 = f1_score(y_test, y_preds, pos_label=1)
    cm = confusion_matrix(y_test, y_preds)

    # Print evaluation metrics
    print(f"Accuracy: {round(acc * 100, 1)}%")
    print(f"Precision: {round(precision * 100, 1)}%")
    print(f"Recall: {round(recall * 100, 1)}%")
    print(f"F1 score: {round(f1 * 100, 1)}%")

    # Display and save confusion matrix
    display_and_save_cm(cm=cm, filename=filename)

    # Save fitted model
    with open(os.path.normpath(os.path.join("..", "sentiment_analysis_outputs", f"{filename}_model.pkl")), "wb") as f:
        pickle.dump(model, f)

    return model
```

Load Models / Embeddings

In [5]:

```
# Word2Vec (no hyperparameter tuning)
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "word2vec_model.pkl")), "rb") as f:
  word2vec_model = pickle.load(f)

word2vec_embeddings = word2vec_model.wv

# Get mapping of words to vector embeddings
word2vec_words_to_embeddings = {word: word2vec_embeddings[word] for word in word2vec_embeddings.key_to_index}
```

In [6]:

```
# SPPMI-SVD (with hyperparameter tuning)
with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_ht.pkl")), "rb") as f:
  sppmi_svd_embeddings = pickle.load(f)

with open(os.path.normpath(os.path.join("..", "embedding_outputs", "sppmi_svd_embeddings_mapping_ht.pkl")), "rb") as f:
  sppmi_svd_embeddings_mapping = pickle.load(f)
```

```
# Get mapping of words to vector embeddings
sppmi_svd_words_to_embeddings = {word: sppmi_svd_embeddings[id] for word, id in sppmi_svd_embeddings_mapping.items()}
```

In [7]:
```
# GloVe (with hyperparameter tuning)
glove_embeddings_df = pyreadr.read_r(os.path.normpath(os.path.join("..", "embedding_outputs", "glove_embeddings_ht.rds")))[Non

# Get mapping of words to vector embeddings
glove_words_to_embeddings = {word: np.array(glove_embeddings_df.loc[word]) for word in glove_embeddings_df.index.tolist()}
```

Load Data

In [8]:
```
with open(os.path.normpath(os.path.join("..", "data", "reviews_train.pkl")), "rb") as f:
  reviews_train = pickle.load(f)

with open(os.path.normpath(os.path.join("..", "data", "reviews_test.pkl")), "rb") as f:
  reviews_test = pickle.load(f)

with open(os.path.normpath(os.path.join("..", "data", "labels_train.pkl")), "rb") as f:
  labels_train = pickle.load(f)

with open(os.path.normpath(os.path.join("..", "data", "labels_test.pkl")), "rb") as f:
  labels_test = pickle.load(f)
```

Skip-Gram (Word2Vec)

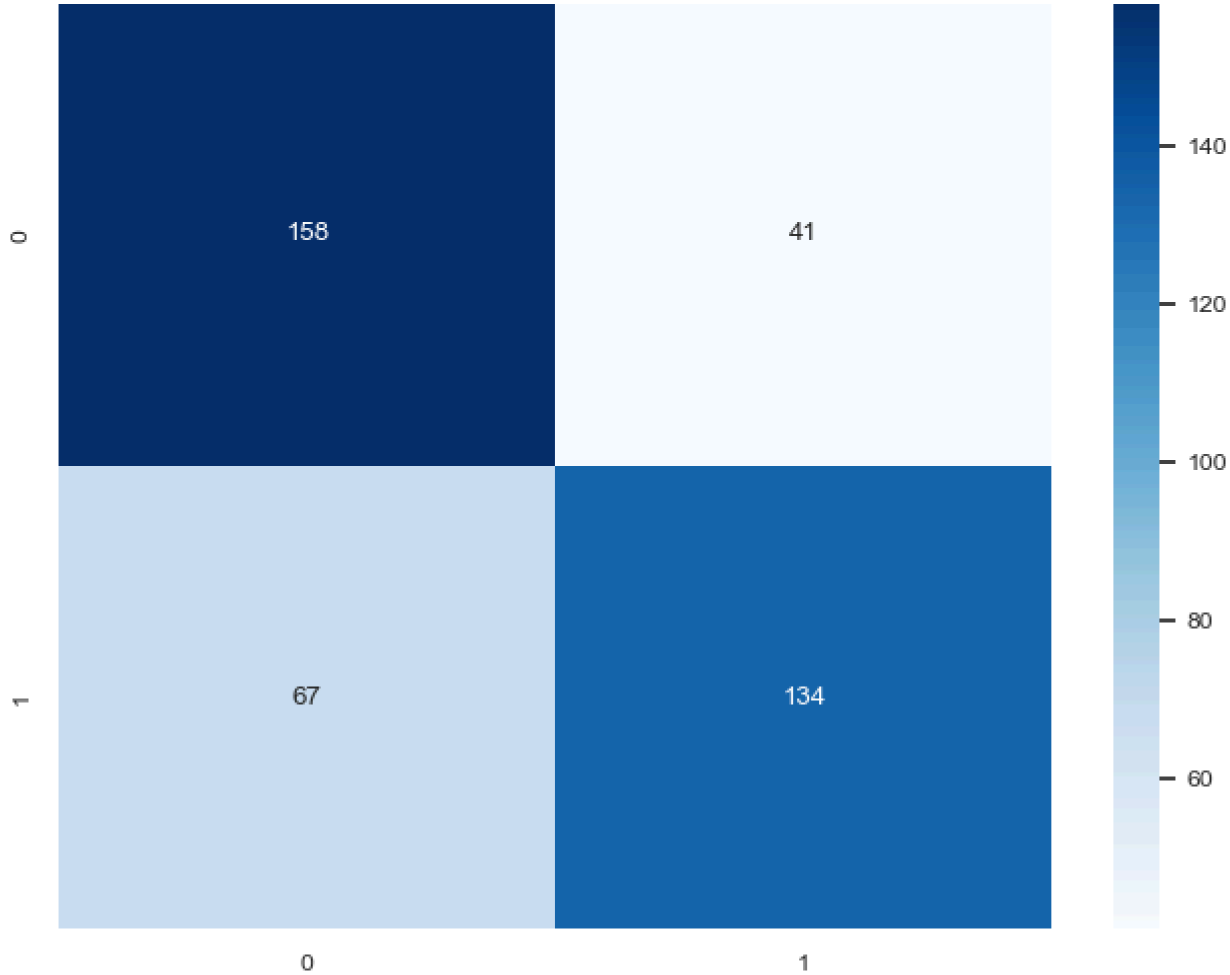
Get Embeddings

In [9]:
```
# Get features
word2vec_embeddings_train = make_embeddings(words_to_embeddings_mapping=word2vec_words_to_embeddings, reviews=reviews_train)
word2vec_embeddings_test = make_embeddings(words_to_embeddings_mapping=word2vec_words_to_embeddings, reviews=reviews_test)
```

Random Forest

In [10]:
```
# Initialise model
word2vec_rf = RandomForestClassifier(random_state=seed)

# Train and evaluate
word2vec_rf_fitted = train_and_eval_classifier(model=word2vec_rf, X_train=word2vec_embeddings_train, y_train=labels_train, X_t
```

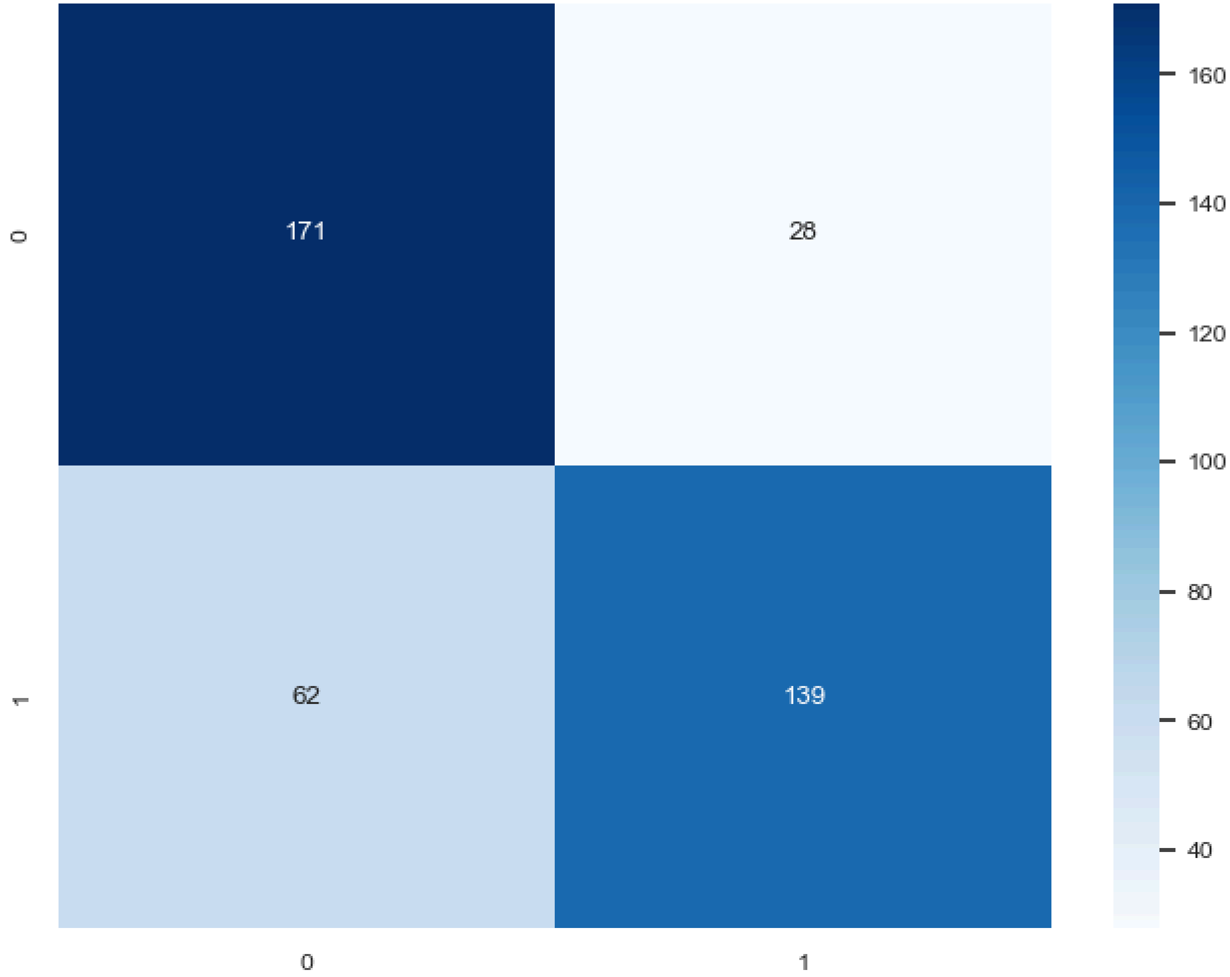
```
Accuracy: 73.0%
Precision: 76.6%
Recall: 66.7%
F1 score: 71.3%
Confusion Matrix:
```



Support Vector Machine

In [11]:
```
# Initialise model
word2vec_svm = SVC(random_state=seed)

# Train and evaluate
word2vec_svm_fitted = train_and_eval_classifier(model=word2vec_svm, X_train=word2vec_embeddings_train, y_train=labels_train, X
```

```
Accuracy: 77.5%
Precision: 83.2%
Recall: 69.2%
F1 score: 75.5%
Confusion Matrix:
```

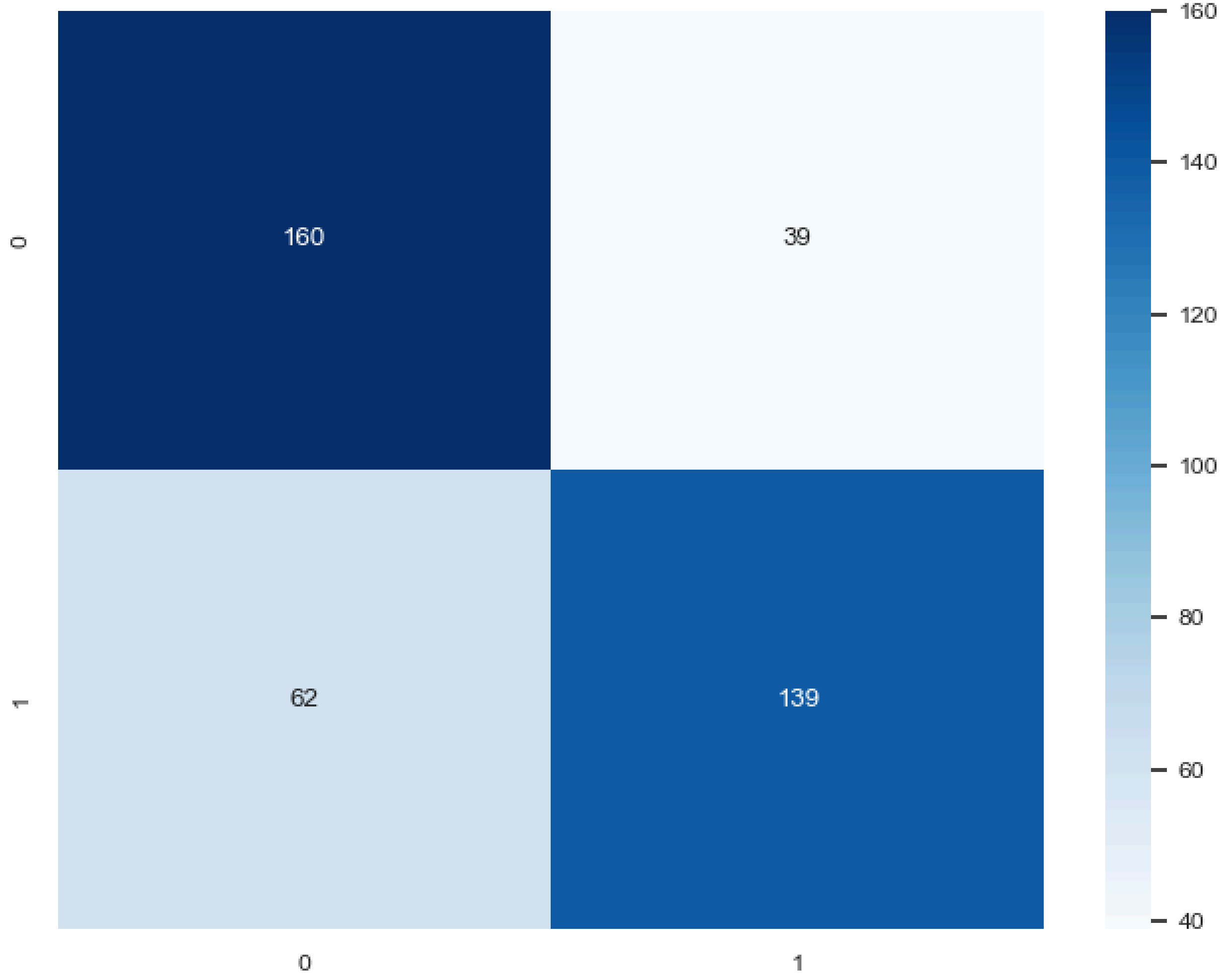


XGBoost

In [12]:
```
# Initialise model
word2vec_xgb = XGBClassifier(random_state=seed)
```

```
# Train and evaluate
word2vec_xgb_fitted = train_and_eval_classifier(model=word2vec_xgb, X_train=word2vec_embeddings_train, y_train=labels_train, X
```

```
Accuracy: 74.8%
Precision: 78.1%
Recall: 69.2%
F1 score: 73.4%
Confusion Matrix:
```



SPPMI-SVD

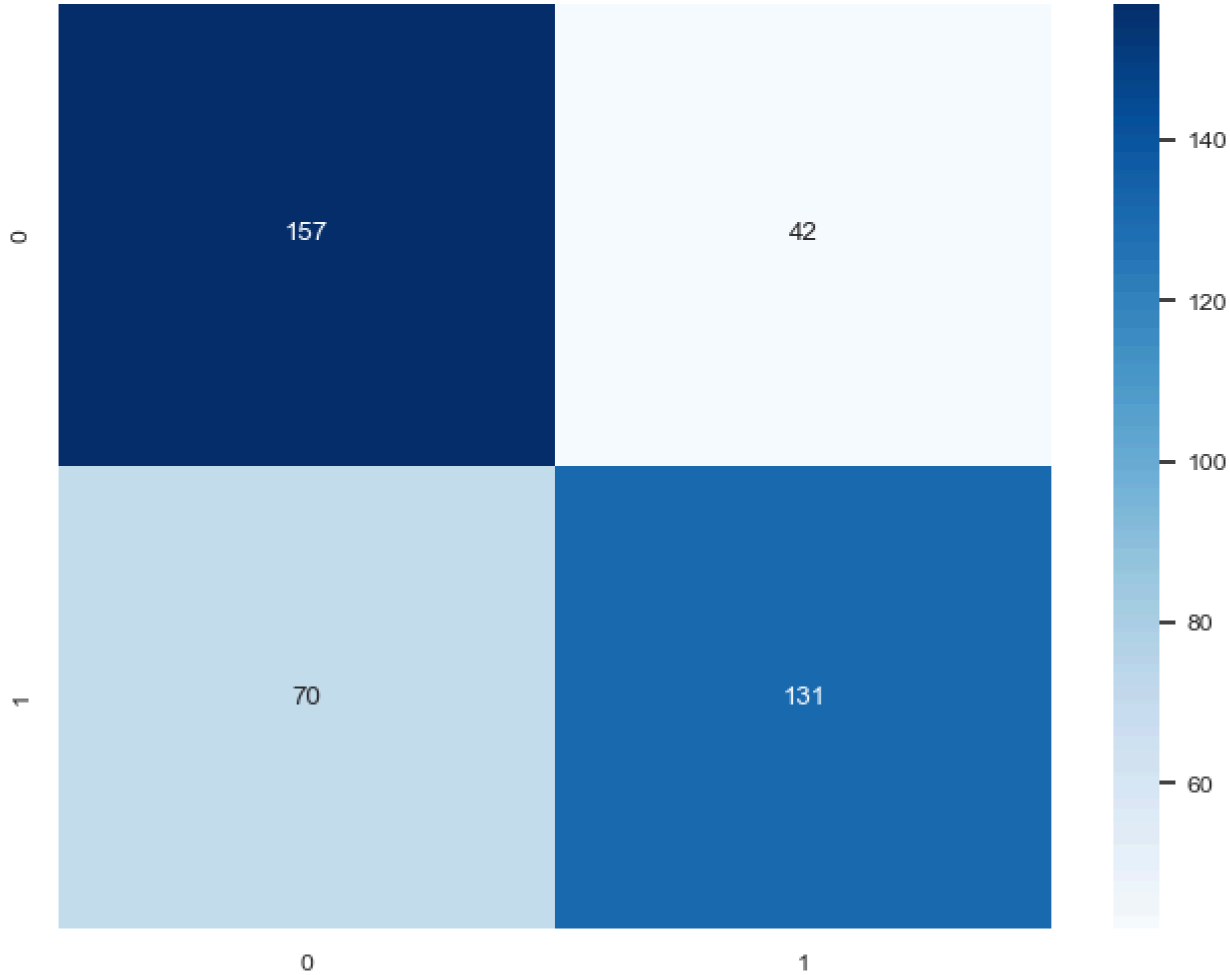
Get Embeddings

In [13]:
```
# Get features
sppmi_svd_embeddings_train = make_embeddings(words_to_embeddings_mapping=sppmi_svd_words_to_embeddings, reviews=reviews_train)
sppmi_svd_embeddings_test = make_embeddings(words_to_embeddings_mapping=sppmi_svd_words_to_embeddings, reviews=reviews_test)
```

Random Forest

In [14]:
```
# Initialise model
sppmi_svd_rf = RandomForestClassifier(random_state=seed)

# Train and evaluate
sppmi_svd_rf_fitted = train_and_eval_classifier(model=sppmi_svd_rf, X_train=sppmi_svd_embeddings_train, y_train=labels_train,
```

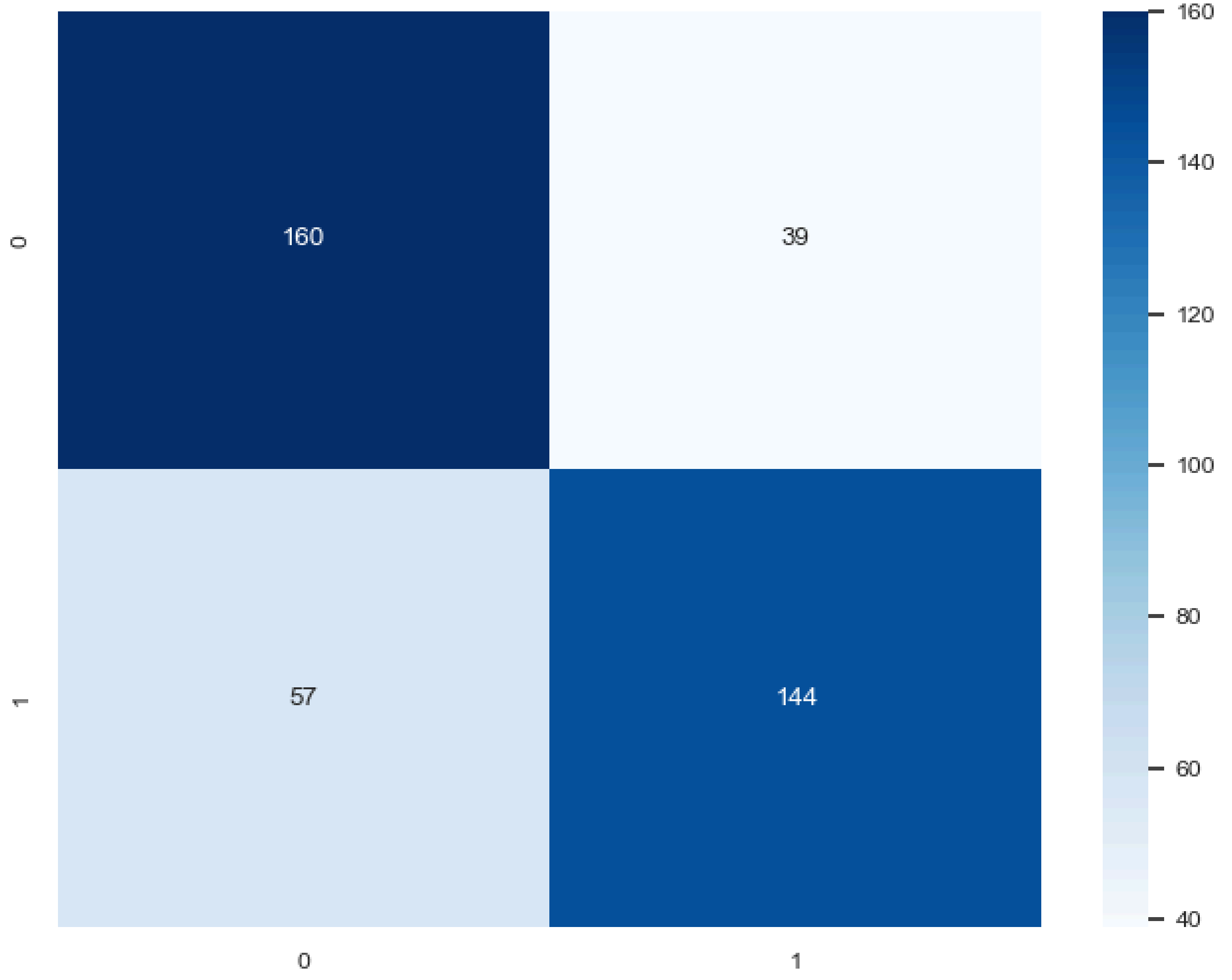
```
Accuracy: 72.0%
Precision: 75.7%
Recall: 65.2%
F1 score: 70.1%
Confusion Matrix:
```



Support Vector Machine

In [15]:
```
# Initialise model
sppmi_svd_svm = SVC(random_state=seed)

# Train and evaluate
sppmi_svd_svm_fitted = train_and_eval_classifier(model=sppmi_svd_svm, X_train=sppmi_svd_embeddings_train, y_train=labels_train
```

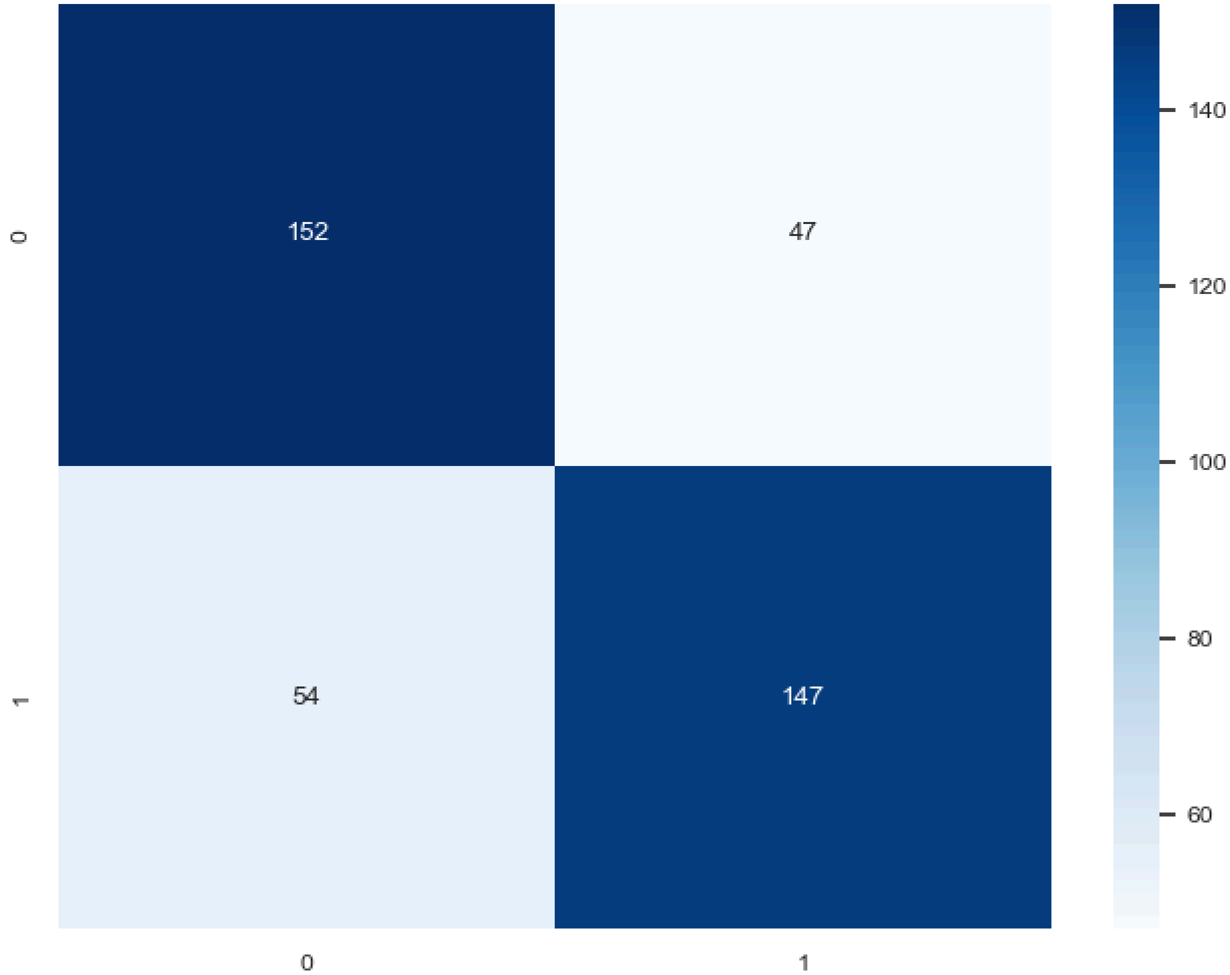
```
Accuracy: 76.0%
Precision: 78.7%
Recall: 71.6%
F1 score: 75.0%
Confusion Matrix:
```



XGBoost

In [16]:
```
# Initialise model
sppmi_svd_xgb = XGBClassifier(random_state=seed)

# Train and evaluate
sppmi_svd_xgb_fitted = train_and_eval_classifier(model=sppmi_svd_xgb, X_train=sppmi_svd_embeddings_train, y_train=labels_train
```

```
Accuracy: 74.8%
Precision: 75.8%
Recall: 73.1%
F1 score: 74.4%
Confusion Matrix:
```



GloVe

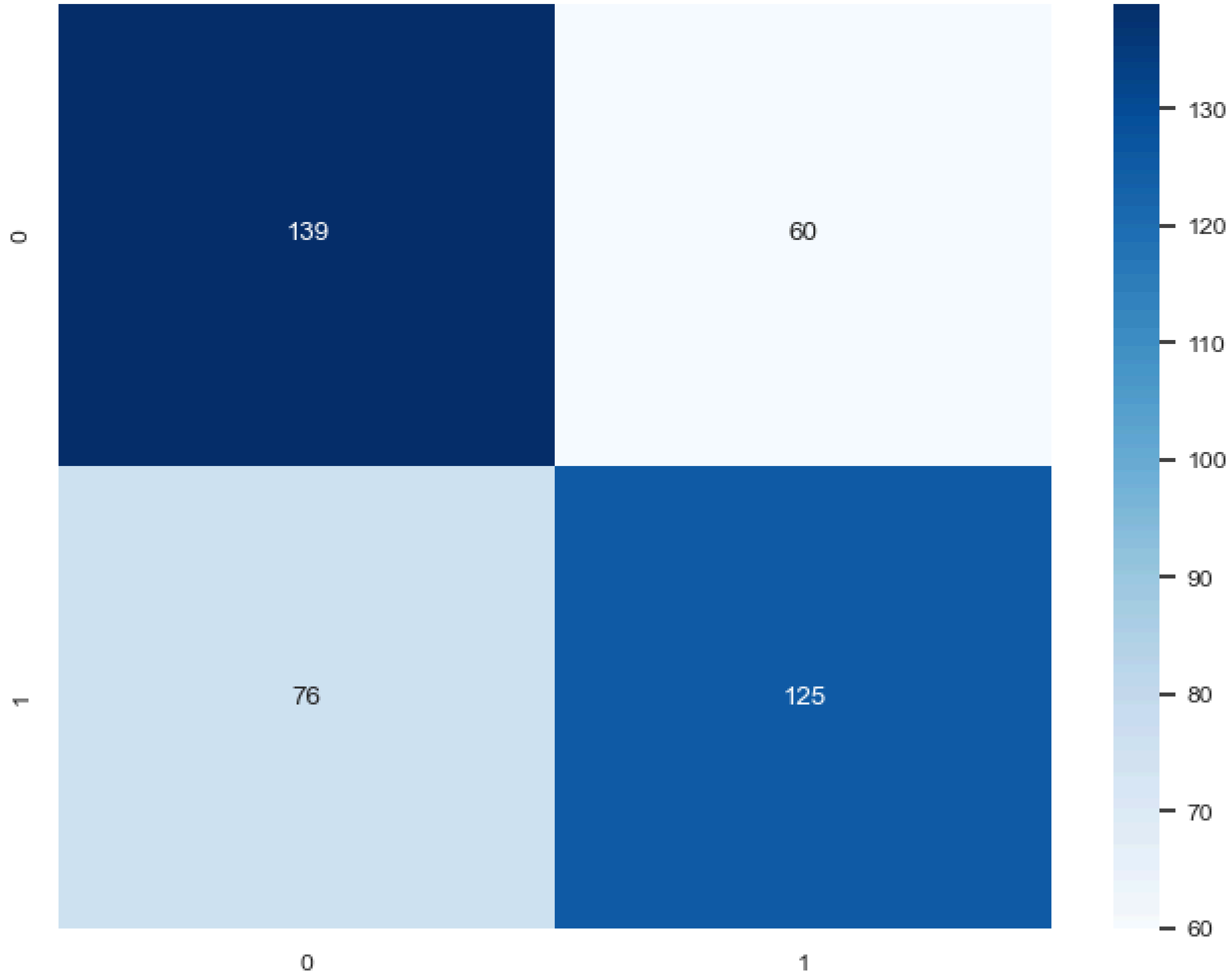
Get Embeddings

In [17]:
```
# Get features
glove_embeddings_train = make_embeddings(words_to_embeddings_mapping=glove_words_to_embeddings, reviews=reviews_train)
glove_embeddings_test = make_embeddings(words_to_embeddings_mapping=glove_words_to_embeddings, reviews=reviews_test)
```

Random Forest

In [18]:
```
# Initialise model
glove_rf = RandomForestClassifier(random_state=seed)

# Train and evaluate
glove_rf_fitted = train_and_eval_classifier(model=glove_rf, X_train=glove_embeddings_train, y_train=labels_train, X_test=glove
```

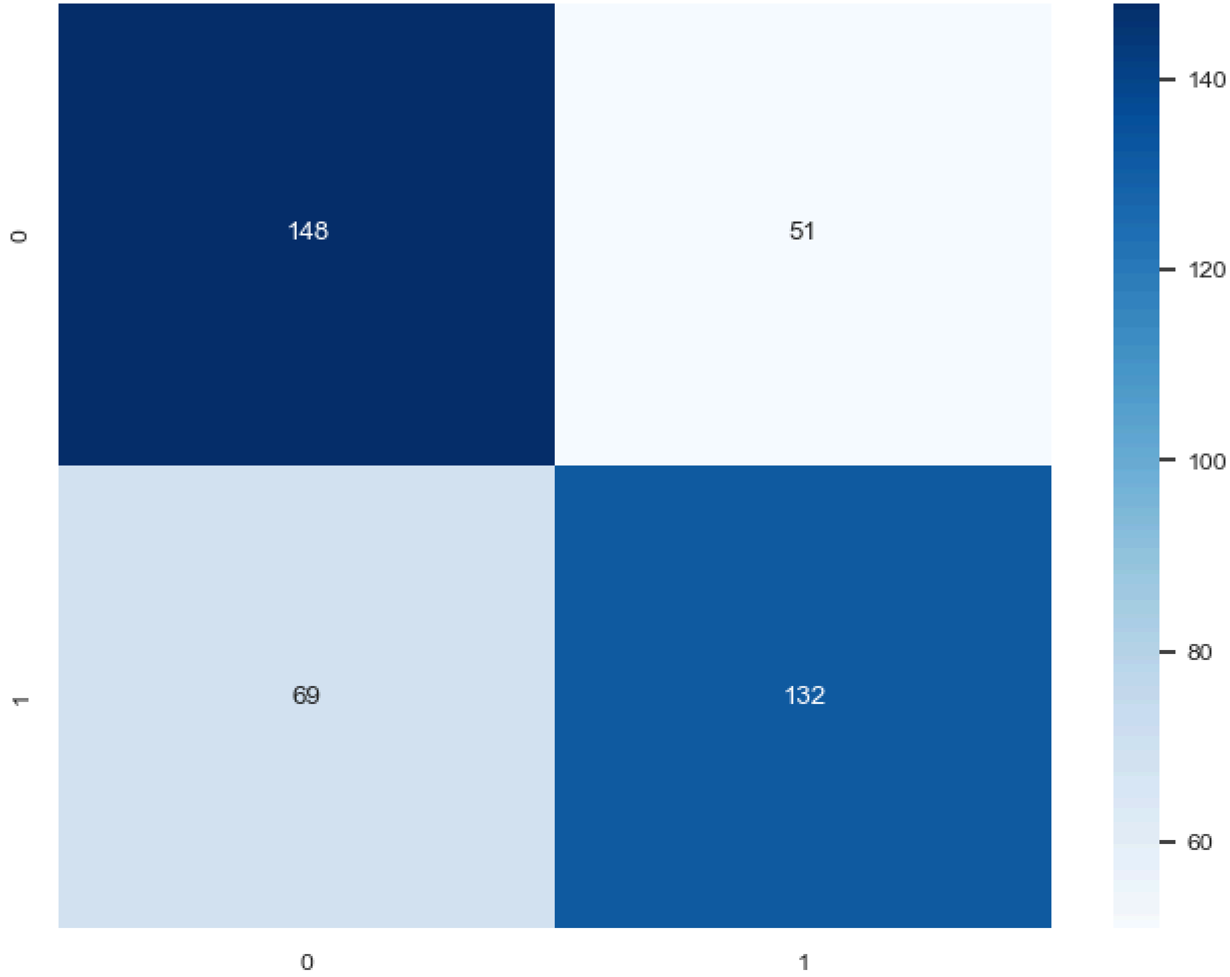
```
Accuracy: 66.0%
Precision: 67.6%
Recall: 62.2%
F1 score: 64.8%
Confusion Matrix:
```



Support Vector Machine

In [19]:
```
# Initialise model
glove_svm = SVC(random_state=seed)

# Train and evaluate
glove_svm_fitted = train_and_eval_classifier(model=glove_svm, X_train=glove_embeddings_train, y_train=labels_train, X_test=glc
```

```
Accuracy: 70.0%
Precision: 72.1%
Recall: 65.7%
F1 score: 68.8%
Confusion Matrix:
```



XGBoost

In [20]:
```
# Initialise model
glove_xgb = XGBClassifier(random_state=seed)
```

```
# Train and evaluate
glove_xgb_fitted = train_and_eval_classifier(model=glove_xgb, X_train=glove_embeddings_train, y_train=labels_train, X_test=glc
```

```
Accuracy: 65.5%
Precision: 67.2%
Recall: 61.2%
F1 score: 64.1%
Confusion Matrix:
```