In [1]:
```python
import os

os.environ["CUDA_LAUNCH_BLOCKING"] = "1"


import random
import time
import math
from collections import Counter
import matplotlib.pyplot as plt


import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import nltk
from nltk.corpus import reuters

# Device Configuration
device = torch.device("mps" if torch.backends.mps.is_available() else ("cuda" if torch.cuda.is_available() else "cpu"))
print(f"Using device: {device}")

# Download necessary NLTK datasets
nltk.download("reuters")
nltk.download("punkt")


# Build Reuters Corpus
def build_corpus(sample_size):
    corpus = []
    # Iterate through the first 'sample_size' files in the Reuters corpus
    for file_id in reuters.fileids()[:sample_size]:
        # Extract words from the file, convert to lowercase, and filter out non-alphabetic tokens
        words = [word.lower() for word in reuters.words(file_id) if word.isalpha()]
        # Append the processed words to the corpus
        corpus.append(words)
    return corpus


# Vocabulary Building
def build_vocab(corpus, min_freq=5):
    # Flatten the corpus to get a list of all words
    words = [word for sentence in corpus for word in sentence]

    # Count the frequency of each word in the corpus
    word_counts = Counter(words)

    # Create a vocabulary list with words that have a frequency greater than or equal to 'min_freq'
    vocab = [word for word, count in word_counts.items() if count >= min_freq]

    # Add a special token for unknown words
    vocab.append("<UNKNOWN>")

    # Create a mapping from words to their indices
    word2index = {word: idx for idx, word in enumerate(vocab)}

    # Ensure the unknown token is mapped to index 0
    word2index["<UNKNOWN>"] = 0

    return vocab, word2index, word_counts


# Generate Skip-grams
def build_skipgrams(corpus, word2index, window_size):
    skip_grams = []
    # Iterate through each sentence in the corpus
    for sentence in corpus:
        # Iterate through each word in the sentence
        for idx, word in enumerate(sentence):
            # Get the index of the center word, default to <UNKNOWN> if not found
            center = word2index.get(word, word2index["<UNKNOWN>"])
            # Define the context window around the center word
            context_window = sentence[max(0, idx - window_size) : idx] + sentence[idx + 1 : idx + window_size + 1]
            # Iterate through each context word in the context window
            for context_word in context_window:
                # Get the index of the context word, default to <UNKNOWN> if not found
                context = word2index.get(context_word, word2index["<UNKNOWN>"])
                # Append the (center, context) pair to the skip_grams list
                skip_grams.append((center, context))
    return skip_grams


def weighting_function(x_ij, x_max=100, alpha=0.75):
    return (x_ij / x_max) ** alpha if x_ij < x_max else 1


# Skipgram Model
class Skipgram(nn.Module):
    def __init__(self, vocab_size, embed_size):
        super(Skipgram, self).__init__()
        self.embedding_v = nn.Embedding(vocab_size, embed_size)
        self.embedding_u = nn.Embedding(vocab_size, embed_size)

    def forward(self, center_words, context_words):
        # Get the embeddings for the center and context words
        center_embed = self.embedding_v(center_words)
        context_embed = self.embedding_u(context_words)

        # Compute the scores by taking the dot product of center and context embeddings
        scores = torch.matmul(center_embed, context_embed.T)

        # Apply log softmax to the scores to get the log probabilities
        log_probs = torch.log_softmax(scores, dim=1)
        return log_probs


# Skipgram Model with Negative Sampling
class SkipgramNegSampling(nn.Module):
    def __init__(self, vocab_size, embed_size):
        super(SkipgramNegSampling, self).__init__()
        self.embedding_v = nn.Embedding(vocab_size, embed_size)
        self.embedding_u = nn.Embedding(vocab_size, embed_size)
        self.logsigmoid = nn.LogSigmoid()

    def forward(self, center_words, pos_context, neg_context):
        # Get the embeddings for the center, positive context, and negative context words
        center_embed = self.embedding_v(center_words)
        pos_embed = self.embedding_u(pos_context)
        neg_embed = self.embedding_u(neg_context)

        # Compute the positive score by taking the dot product of center and positive context embeddings
        pos_score = self.logsigmoid(torch.bmm(pos_embed.unsqueeze(1), center_embed.unsqueeze(2))).squeeze()

        # Compute the negative score by taking the dot product of center and negative context embeddings
        neg_score = self.logsigmoid(-torch.bmm(neg_embed, center_embed.unsqueeze(2))).squeeze()

        # Calculate the loss as the negative sum of positive and negative scores
        loss = -(pos_score.sum() + neg_score.sum())
        return loss
```

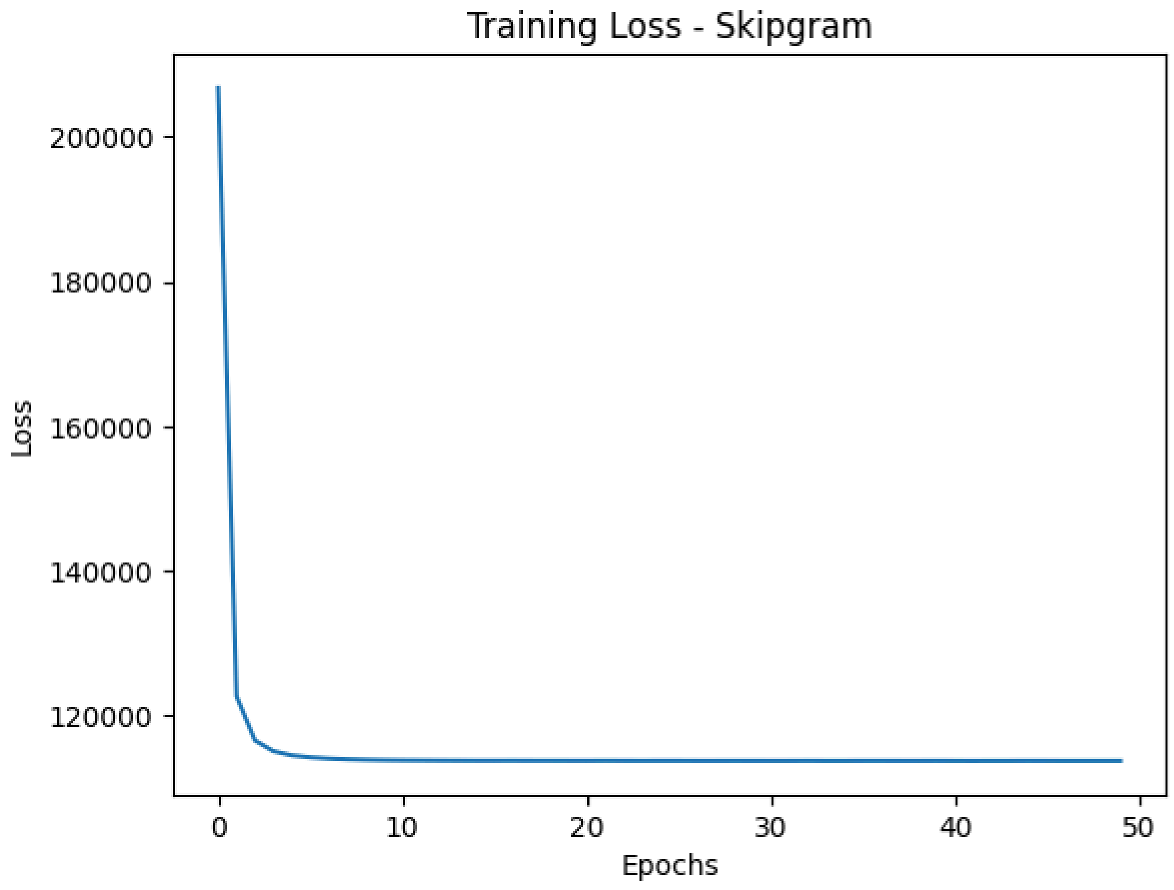
```python
# GloVe Model
class GloVe(nn.Module):
    def __init__(self, vocab_size, embed_size):
        super(GloVe, self).__init__()
        self.embedding_v = nn.Embedding(vocab_size, embed_size)
        self.embedding_u = nn.Embedding(vocab_size, embed_size)
        self.v_bias = nn.Embedding(vocab_size, 1)
        self.u_bias = nn.Embedding(vocab_size, 1)

    def forward(self, center_words, target_words, co_occurrences, weightings):
        # Get the embeddings for the center and target words
        center_embed = self.embedding_v(center_words)
        target_embed = self.embedding_u(target_words)

        # Get the biases for the center and target words
        center_bias = self.v_bias(center_words).squeeze(1)
        target_bias = self.u_bias(target_words).squeeze(1)

        # Compute the inner product of the center and target embeddings
        inner_product = (center_embed * target_embed).sum(dim=1)

        # Calculate the loss using the weighting function and co-occurrences
        loss = weightings * torch.pow(inner_product + center_bias + target_bias - co_occurrences, 2)
        return loss.mean()


# Plotting Loss Function
def plot_losses(losses, model_name):
    plt.plot(losses)
    plt.title(f"Training Loss - {model_name}")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.show()


# Training Function for Skipgram Models
def train_skipgram(model, skip_grams, epochs, batch_size, learning_rate, word2index, num_neg_samples=5):
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    loss_history = []

    for epoch in range(epochs):
        start_time = time.time()
        total_loss = 0

        # Shuffle the skip-grams for each epoch
        random.shuffle(skip_grams)

        for i in range(0, len(skip_grams), batch_size):
            # Extract the current batch from the skip-grams
            batch = skip_grams[i : i + batch_size]

            # Unzip the batch into separate lists for center and context words
            center_words, context_words = zip(*batch)

            # Convert the lists to tensors and move them to CPU/GPU
            center_words = torch.LongTensor(center_words).to(device)
            context_words = torch.LongTensor(context_words).to(device)

            # If the batch is smaller than the batch size, pad with zeros
            if len(batch) < batch_size:
                padding_size = batch_size - len(batch)
                center_words = torch.cat([center_words, torch.zeros(padding_size, dtype=torch.long).to(device)])
                context_words = torch.cat([context_words, torch.zeros(padding_size, dtype=torch.long).to(device)])

            # Zero the gradients
            optimizer.zero_grad()

            assert context_words.max().item() < len(word2index), "Context word index out of bounds."
            assert context_words.min().item() >= 0, "Negative index found in context_words."

            # Compute the loss based on the model type
            if isinstance(model, Skipgram):
                log_probs = model(center_words, context_words)
                # print(f"log_probs shape: {log_probs.shape}, context_words shape: {context_words.shape}")

                # Clip context_words to avoid index errors
                context_words = torch.clamp(context_words, max=log_probs.shape[1] - 1)
                assert context_words.max().item() < log_probs.shape[1], "Index out of bounds in context_words"

                # Replace manual loss with NLLLoss
                loss_fn = torch.nn.NLLLoss()
                loss = loss_fn(log_probs, context_words)
                # loss = -torch.mean(log_probs[range(batch_size), context_words])
            else:
                # neg_samples = torch.LongTensor([np.random.choice(len(word2index), num_neg_samples) for _ in range(batch_size)]).to(device)
                neg_samples = torch.from_numpy(np.random.choice(len(word2index), (batch_size, num_neg_samples))).long().to(device)
                loss = model(center_words, context_words, neg_samples)

            # Backpropagate the loss and update the model parameters
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
        print(
            f"Model: {model.__class__.__name__:20s}| Epoch: {epoch + 1:-3d}/{epochs}  Loss: {total_loss:12.4f}  Time: {time.time() - start_time:6.2f}s"
        )
        loss_history.append(total_loss)
    plot_losses(loss_history, model.__class__.__name__)


def train_glove_model(model, training_data, epochs, batch_size, learning_rate):
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    loss_history = []
    model.train()

    for epoch in range(epochs):
        start_time = time.time()
        total_loss = 0

        # Shuffle the training data for each epoch
        random.shuffle(training_data)

        for i in range(0, len(training_data), batch_size):
            # Extract the current batch from the training data
            batch = training_data[i : i + batch_size]

            # Unzip the batch into separate lists for centers, contexts, co-occurrences, and weights
            centers, contexts, coocs, weights = zip(*batch)

            # Convert the lists to tensors and move them to CPU/GPU
            centers = torch.LongTensor(centers).to(device)
            contexts = torch.LongTensor(contexts).to(device)
            coocs = torch.FloatTensor(coocs).to(device)
            weights = torch.FloatTensor(weights).to(device)

            optimizer.zero_grad()

            loss = model(centers, contexts, coocs, weights)

            loss.backward()
```

```python
            optimizer.step()

            total_loss += loss.item()
        print(
            f"Model: {model.__class__.__name__:20s}| Epoch: {epoch + 1:-3d}/{epochs}  Loss: {total_loss:12.4f}  Time: {time.time() - start_time:6.2f}s"
        )
        loss_history.append(total_loss)
    plot_losses(loss_history, model.__class__.__name__)


# Hyperparameters
# SAMPLE_SIZE = 1000  # Number of documents to sample from the Reuters corpus
SAMPLE_SIZE = len(reuters.fileids())
WINDOW_SIZE = 2  # Context window size for skip-grams
EMBEDDING_DIMENSION = 100  # Dimension of the embedding vectors
TOTAL_EPOCHS = 50  # Number of epochs to train the models
BATCH_SIZE = 256  # Batch size for training
LEARNING_RATE = 0.001  # Learning rate for the optimizer

# Build the corpus from the Reuters dataset
corpus = build_corpus(SAMPLE_SIZE)

# Build the vocabulary from the corpus with a minimum frequency threshold
vocab, word2index, word_counts = build_vocab(corpus)

# Generate skip-grams from the corpus using the built vocabulary
skip_grams = build_skipgrams(corpus, word2index, WINDOW_SIZE)

# Train Models
print("--------------------------------------------------------------------------------")
model_skipgram = Skipgram(len(vocab), EMBEDDING_DIMENSION).to(device)
train_skipgram(model_skipgram, skip_grams, TOTAL_EPOCHS, BATCH_SIZE, LEARNING_RATE, word2index)
print("--------------------------------------------------------------------------------")
model_skipgram_neg = SkipgramNegSampling(len(vocab), EMBEDDING_DIMENSION).to(device)
train_skipgram(model_skipgram_neg, skip_grams, TOTAL_EPOCHS, BATCH_SIZE, LEARNING_RATE, word2index)
print("--------------------------------------------------------------------------------")
model_glove = GloVe(len(vocab), EMBEDDING_DIMENSION).to(device)
co_occurrence_matrix = Counter(skip_grams)  # Prepare co-occurrence matrix
training_data_glove = [(center, context, math.log(count + 1), weighting_function(count)) for (center, context), count in co_occurrence_matrix.items()]
train_glove_model(model_glove, training_data_glove, TOTAL_EPOCHS, BATCH_SIZE, LEARNING_RATE)
print("--------------------------------------------------------------------------------")

# Save Models
torch.save({"model_state_dict": model_skipgram.state_dict(), "word2index": word2index, "vocab": vocab}, "skipgram_model.pth")
torch.save({"model_state_dict": model_skipgram_neg.state_dict(), "word2index": word2index, "vocab": vocab}, "skipgram_neg_model.pth")
torch.save({"model_state_dict": model_glove.state_dict(), "word2index": word2index, "vocab": vocab}, "glove_model.pth")
print("All models have been saved.")
```

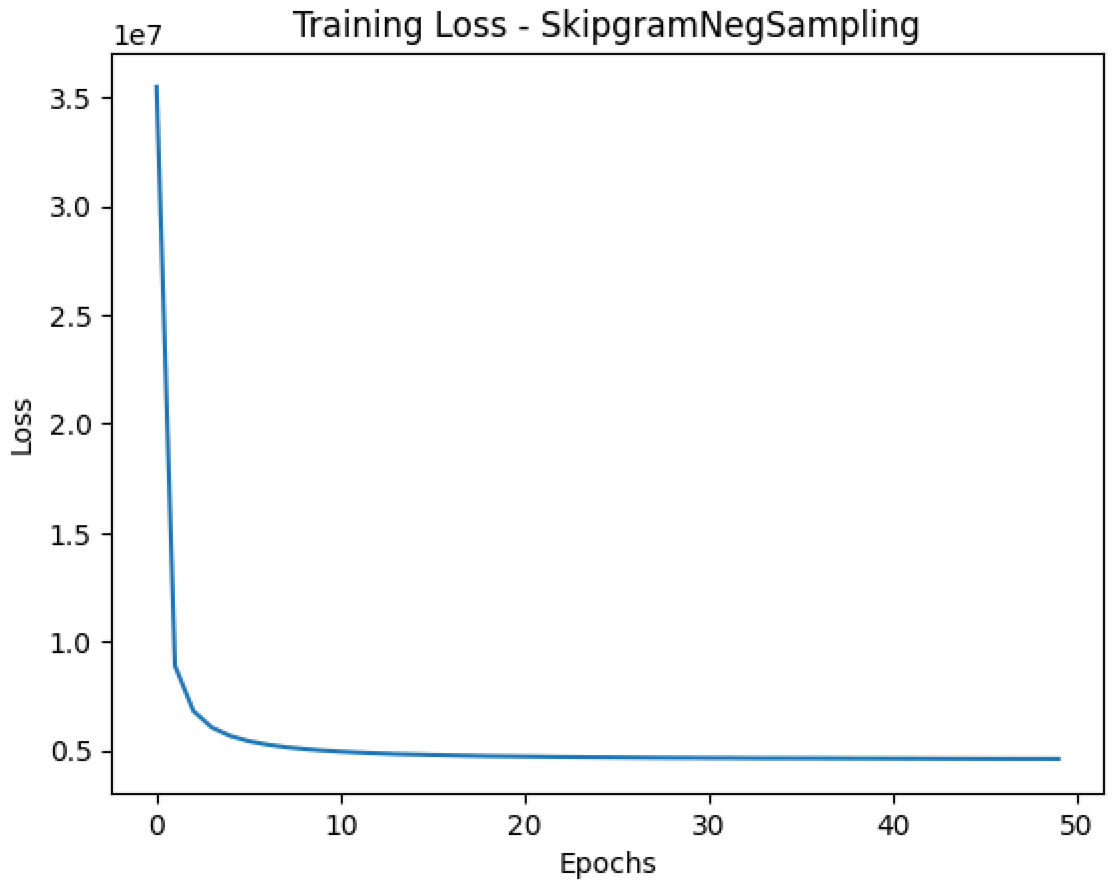
```
Using device: cuda
```

```
[nltk_data] Downloading package reuters to /home/jupyter-
[nltk_data]     st125127/nltk_data...
[nltk_data]   Package reuters is already up-to-date!
[nltk_data] Downloading package punkt to /home/jupyter-
[nltk_data]     st125127/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

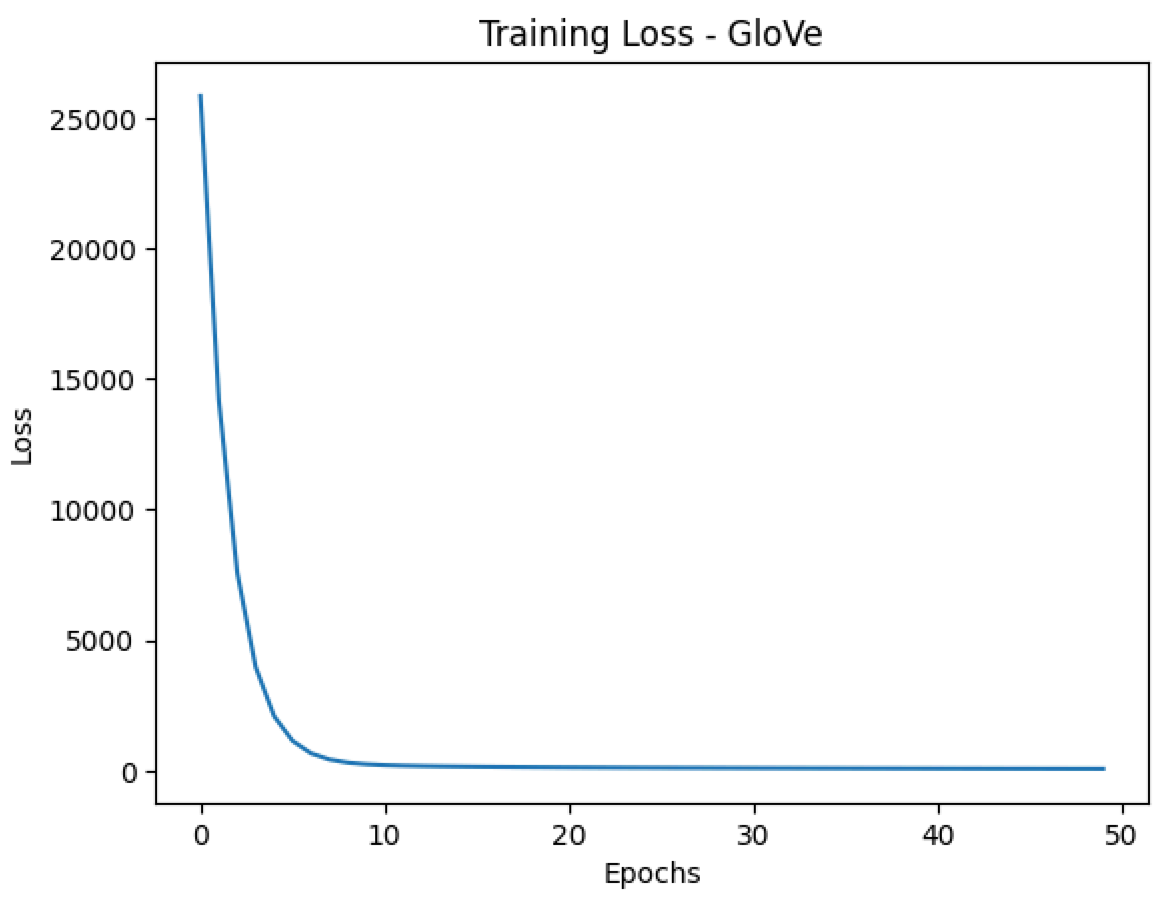
```
--------------------------------------------------------------------------------
Model: Skipgram            | Epoch:   1/50  Loss:  206793.8349  Time:  51.73s
Model: Skipgram            | Epoch:   2/50  Loss:  122697.5201  Time:  50.23s
Model: Skipgram            | Epoch:   3/50  Loss:  116577.0100  Time:  56.89s
Model: Skipgram            | Epoch:   4/50  Loss:  115097.3579  Time:  52.81s
Model: Skipgram            | Epoch:   5/50  Loss:  114510.7330  Time:  50.54s
Model: Skipgram            | Epoch:   6/50  Loss:  114239.5806  Time:  50.02s
Model: Skipgram            | Epoch:   7/50  Loss:  114071.3658  Time:  50.21s
Model: Skipgram            | Epoch:   8/50  Loss:  113966.9763  Time:  56.55s
Model: Skipgram            | Epoch:   9/50  Loss:  113900.2349  Time:  50.55s
Model: Skipgram            | Epoch:  10/50  Loss:  113856.8528  Time:  50.30s
Model: Skipgram            | Epoch:  11/50  Loss:  113832.0894  Time:  50.31s
Model: Skipgram            | Epoch:  12/50  Loss:  113818.9138  Time:  54.42s
Model: Skipgram            | Epoch:  13/50  Loss:  113804.5176  Time:  50.47s
Model: Skipgram            | Epoch:  14/50  Loss:  113793.5932  Time:  50.18s
Model: Skipgram            | Epoch:  15/50  Loss:  113784.5499  Time:  53.90s
Model: Skipgram            | Epoch:  16/50  Loss:  113783.5260  Time:  51.73s
Model: Skipgram            | Epoch:  17/50  Loss:  113787.0366  Time:  50.46s
Model: Skipgram            | Epoch:  18/50  Loss:  113779.8843  Time:  50.24s
Model: Skipgram            | Epoch:  19/50  Loss:  113778.8513  Time:  50.29s
Model: Skipgram            | Epoch:  20/50  Loss:  113774.6845  Time:  51.15s
Model: Skipgram            | Epoch:  21/50  Loss:  113771.6251  Time:  54.03s
Model: Skipgram            | Epoch:  22/50  Loss:  113778.6493  Time:  50.53s
Model: Skipgram            | Epoch:  23/50  Loss:  113772.0127  Time:  57.74s
Model: Skipgram            | Epoch:  24/50  Loss:  113766.1785  Time:  50.38s
Model: Skipgram            | Epoch:  25/50  Loss:  113771.5025  Time:  50.20s
Model: Skipgram            | Epoch:  26/50  Loss:  113769.9036  Time:  50.19s
Model: Skipgram            | Epoch:  27/50  Loss:  113761.2431  Time:  50.40s
Model: Skipgram            | Epoch:  28/50  Loss:  113761.3268  Time:  50.20s
Model: Skipgram            | Epoch:  29/50  Loss:  113765.0748  Time:  50.02s
Model: Skipgram            | Epoch:  30/50  Loss:  113763.0451  Time:  50.00s
Model: Skipgram            | Epoch:  31/50  Loss:  113760.2598  Time:  52.74s
Model: Skipgram            | Epoch:  32/50  Loss:  113760.9852  Time:  50.23s
Model: Skipgram            | Epoch:  33/50  Loss:  113769.2522  Time:  50.25s
Model: Skipgram            | Epoch:  34/50  Loss:  113757.3656  Time:  50.31s
Model: Skipgram            | Epoch:  35/50  Loss:  113758.1626  Time:  50.15s
Model: Skipgram            | Epoch:  36/50  Loss:  113763.8537  Time:  50.01s
Model: Skipgram            | Epoch:  37/50  Loss:  113765.3965  Time:  53.50s
Model: Skipgram            | Epoch:  38/50  Loss:  113761.8824  Time:  50.45s
Model: Skipgram            | Epoch:  39/50  Loss:  113760.1840  Time:  53.79s
Model: Skipgram            | Epoch:  40/50  Loss:  113764.8393  Time:  50.07s
Model: Skipgram            | Epoch:  41/50  Loss:  113757.9337  Time:  50.28s
Model: Skipgram            | Epoch:  42/50  Loss:  113752.3064  Time:  50.30s
Model: Skipgram            | Epoch:  43/50  Loss:  113759.2773  Time:  50.04s
Model: Skipgram            | Epoch:  44/50  Loss:  113758.9017  Time:  50.23s
Model: Skipgram            | Epoch:  45/50  Loss:  113764.8812  Time:  52.66s
Model: Skipgram            | Epoch:  46/50  Loss:  113761.0172  Time:  50.04s
Model: Skipgram            | Epoch:  47/50  Loss:  113758.3588  Time:  49.96s
Model: Skipgram            | Epoch:  48/50  Loss:  113759.1755  Time:  50.00s
Model: Skipgram            | Epoch:  49/50  Loss:  113756.9309  Time:  50.06s
Model: Skipgram            | Epoch:  50/50  Loss:  113757.2915  Time:  53.12s
```




```
------------------------------------------------------------------------------
Model: SkipgramNegSampling | Epoch:   1/50  Loss: 35490448.5621  Time:  98.24s
Model: SkipgramNegSampling | Epoch:   2/50  Loss: 8895835.6872  Time:  97.62s
Model: SkipgramNegSampling | Epoch:   3/50  Loss: 6815674.2596  Time:  92.62s
Model: SkipgramNegSampling | Epoch:   4/50  Loss: 6058098.8755  Time:  92.35s
Model: SkipgramNegSampling | Epoch:   5/50  Loss: 5670921.4602  Time:  92.69s
Model: SkipgramNegSampling | Epoch:   6/50  Loss: 5430236.0637  Time:  92.71s
Model: SkipgramNegSampling | Epoch:   7/50  Loss: 5270409.5954  Time:  91.98s
Model: SkipgramNegSampling | Epoch:   8/50  Loss: 5152779.3537  Time:  95.48s
Model: SkipgramNegSampling | Epoch:   9/50  Loss: 5068606.2010  Time:  96.08s
Model: SkipgramNegSampling | Epoch:  10/50  Loss: 5000201.6832  Time:  91.55s
Model: SkipgramNegSampling | Epoch:  11/50  Loss: 4946154.0013  Time:  92.40s
Model: SkipgramNegSampling | Epoch:  12/50  Loss: 4903049.6545  Time:  97.51s
Model: SkipgramNegSampling | Epoch:  13/50  Loss: 4868929.1121  Time:  98.27s
Model: SkipgramNegSampling | Epoch:  14/50  Loss: 4836983.6913  Time:  95.27s
Model: SkipgramNegSampling | Epoch:  15/50  Loss: 4814809.9045  Time:  95.41s
Model: SkipgramNegSampling | Epoch:  16/50  Loss: 4791652.5036  Time:  95.30s
Model: SkipgramNegSampling | Epoch:  17/50  Loss: 4774211.4023  Time:  95.75s
Model: SkipgramNegSampling | Epoch:  18/50  Loss: 4757116.1124  Time:  94.68s
Model: SkipgramNegSampling | Epoch:  19/50  Loss: 4741089.9543  Time:  99.97s
Model: SkipgramNegSampling | Epoch:  20/50  Loss: 4731496.1444  Time: 101.75s
Model: SkipgramNegSampling | Epoch:  21/50  Loss: 4721718.0607  Time:  84.50s
Model: SkipgramNegSampling | Epoch:  22/50  Loss: 4710923.8855  Time:  51.65s
Model: SkipgramNegSampling | Epoch:  23/50  Loss: 4701309.4241  Time:  51.69s
Model: SkipgramNegSampling | Epoch:  24/50  Loss: 4694604.9067  Time:  99.55s
Model: SkipgramNegSampling | Epoch:  25/50  Loss: 4686311.5633  Time: 101.11s
Model: SkipgramNegSampling | Epoch:  26/50  Loss: 4682442.2797  Time: 101.83s
Model: SkipgramNegSampling | Epoch:  27/50  Loss: 4675707.9523  Time:  97.83s
Model: SkipgramNegSampling | Epoch:  28/50  Loss: 4668543.2730  Time:  99.43s
Model: SkipgramNegSampling | Epoch:  29/50  Loss: 4663283.5550  Time:  97.53s
Model: SkipgramNegSampling | Epoch:  30/50  Loss: 4660312.3248  Time:  97.41s
Model: SkipgramNegSampling | Epoch:  31/50  Loss: 4654341.4692  Time:  97.69s
Model: SkipgramNegSampling | Epoch:  32/50  Loss: 4654255.7836  Time:  97.59s
Model: SkipgramNegSampling | Epoch:  33/50  Loss: 4646798.5071  Time:  96.30s
Model: SkipgramNegSampling | Epoch:  34/50  Loss: 4642924.2044  Time:  97.67s
Model: SkipgramNegSampling | Epoch:  35/50  Loss: 4642534.0558  Time:  97.78s
Model: SkipgramNegSampling | Epoch:  36/50  Loss: 4641483.7262  Time:  97.92s
Model: SkipgramNegSampling | Epoch:  37/50  Loss: 4639276.4671  Time:  97.41s
Model: SkipgramNegSampling | Epoch:  38/50  Loss: 4635902.8608  Time:  94.53s
Model: SkipgramNegSampling | Epoch:  39/50  Loss: 4632345.9391  Time:  97.06s
Model: SkipgramNegSampling | Epoch:  40/50  Loss: 4631959.4249  Time:  97.82s
Model: SkipgramNegSampling | Epoch:  41/50  Loss: 4628359.9640  Time:  97.74s
Model: SkipgramNegSampling | Epoch:  42/50  Loss: 4626505.6184  Time:  96.39s
Model: SkipgramNegSampling | Epoch:  43/50  Loss: 4624475.0277  Time:  96.55s
Model: SkipgramNegSampling | Epoch:  44/50  Loss: 4619339.7402  Time:  97.21s
Model: SkipgramNegSampling | Epoch:  45/50  Loss: 4618475.7701  Time:  96.88s
Model: SkipgramNegSampling | Epoch:  46/50  Loss: 4616495.7107  Time:  97.45s
Model: SkipgramNegSampling | Epoch:  47/50  Loss: 4614299.9071  Time:  94.62s
Model: SkipgramNegSampling | Epoch:  48/50  Loss: 4612837.6267  Time:  94.61s
Model: SkipgramNegSampling | Epoch:  49/50  Loss: 4613706.5779  Time:  97.88s
Model: SkipgramNegSampling | Epoch:  50/50  Loss: 4610617.7234  Time:  95.79s
```



```
------------------------------------------------------------------------------
Model: GloVe                 | Epoch:   1/50  Loss:   25843.5179  Time:  12.25s
Model: GloVe                 | Epoch:   2/50  Loss:   14167.1556  Time:  12.19s
Model: GloVe                 | Epoch:   3/50  Loss:    7579.4193  Time:  12.14s
Model: GloVe                 | Epoch:   4/50  Loss:    3964.5693  Time:  12.13s
Model: GloVe                 | Epoch:   5/50  Loss:    2085.6386  Time:  12.15s
Model: GloVe                 | Epoch:   6/50  Loss:    1139.3617  Time:  12.13s
Model: GloVe                 | Epoch:   7/50  Loss:     663.7674  Time:  12.15s
Model: GloVe                 | Epoch:   8/50  Loss:     425.6108  Time:  15.00s
Model: GloVe                 | Epoch:   9/50  Loss:     305.3619  Time:  16.08s
Model: GloVe                 | Epoch:  10/50  Loss:     243.3381  Time:  16.07s
Model: GloVe                 | Epoch:  11/50  Loss:     209.0370  Time:  16.26s
Model: GloVe                 | Epoch:  12/50  Loss:     187.9884  Time:  16.65s
Model: GloVe                 | Epoch:  13/50  Loss:     173.7250  Time:  16.09s
Model: GloVe                 | Epoch:  14/50  Loss:     163.2595  Time:  16.48s
Model: GloVe                 | Epoch:  15/50  Loss:     154.9493  Time:  16.65s
Model: GloVe                 | Epoch:  16/50  Loss:     147.8262  Time:  16.63s
Model: GloVe                 | Epoch:  17/50  Loss:     141.9822  Time:  17.32s
Model: GloVe                 | Epoch:  18/50  Loss:     136.4562  Time:  16.61s
Model: GloVe                 | Epoch:  19/50  Loss:     131.3827  Time:  16.67s
Model: GloVe                 | Epoch:  20/50  Loss:     127.0218  Time:  16.63s
Model: GloVe                 | Epoch:  21/50  Loss:     122.9225  Time:  16.64s
Model: GloVe                 | Epoch:  22/50  Loss:     119.3220  Time:  16.63s
Model: GloVe                 | Epoch:  23/50  Loss:     115.9704  Time:  16.62s
Model: GloVe                 | Epoch:  24/50  Loss:     112.6175  Time:  16.37s
Model: GloVe                 | Epoch:  25/50  Loss:     109.7801  Time:  16.44s
Model: GloVe                 | Epoch:  26/50  Loss:     107.0508  Time:  16.46s
Model: GloVe                 | Epoch:  27/50  Loss:     104.4018  Time:  16.51s
Model: GloVe                 | Epoch:  28/50  Loss:     102.2536  Time:  16.89s
Model: GloVe                 | Epoch:  29/50  Loss:      99.9346  Time:  16.90s
Model: GloVe                 | Epoch:  30/50  Loss:      97.9263  Time:  16.53s
Model: GloVe                 | Epoch:  31/50  Loss:      95.6762  Time:  42.42s
Model: GloVe                 | Epoch:  32/50  Loss:      93.7390  Time:  31.16s
Model: GloVe                 | Epoch:  33/50  Loss:      92.0464  Time:  29.07s
Model: GloVe                 | Epoch:  34/50  Loss:      90.2463  Time:  43.15s
Model: GloVe                 | Epoch:  35/50  Loss:      88.5275  Time:  32.39s
Model: GloVe                 | Epoch:  36/50  Loss:      87.0389  Time:  28.95s
Model: GloVe                 | Epoch:  37/50  Loss:      85.1334  Time:  42.57s
Model: GloVe                 | Epoch:  38/50  Loss:      83.9216  Time:  30.54s
Model: GloVe                 | Epoch:  39/50  Loss:      82.2594  Time:  30.73s
Model: GloVe                 | Epoch:  40/50  Loss:      81.1271  Time:  42.78s
Model: GloVe                 | Epoch:  41/50  Loss:      79.4482  Time:  23.09s
Model: GloVe                 | Epoch:  42/50  Loss:      78.4711  Time:  31.30s
Model: GloVe                 | Epoch:  43/50  Loss:      76.7099  Time:  21.75s
Model: GloVe                 | Epoch:  44/50  Loss:      75.8756  Time:  35.56s
Model: GloVe                 | Epoch:  45/50  Loss:      74.4316  Time:  32.83s
Model: GloVe                 | Epoch:  46/50  Loss:      73.4162  Time:  26.91s
Model: GloVe                 | Epoch:  47/50  Loss:      72.2839  Time:  43.89s
Model: GloVe                 | Epoch:  48/50  Loss:      70.9200  Time:  25.23s
Model: GloVe                 | Epoch:  49/50  Loss:      70.1287  Time:  31.65s
Model: GloVe                 | Epoch:  50/50  Loss:      68.9125  Time:  35.32s
```




```
------------------------------------------------------------------------------
All models have been saved.
```

In [2]:
```python
def load_model(filepath, model_class, vocab_size, embed_size):
    checkpoint = torch.load(filepath, weights_only=False)
    model = model_class(vocab_size, embed_size)
    model.load_state_dict(checkpoint["model_state_dict"])
    model.eval()
    return model, checkpoint["word2index"], checkpoint["vocab"]


model_skipgram, word2index_skipgram, vocab_skipgram = load_model("skipgram_model.pth", Skipgram, len(vocab), EMBEDDING_DIMENSION)
model_skipgram_neg, word2index_skipgram_neg, vocab_skipgram_neg = load_model("skipgram_neg_model.pth", SkipgramNegSampling, len(vocab), EMBEDDING_DIMENSION)
model_glove, word2index_glove, vocab_glove = load_model("glove_model.pth", GloVe, len(vocab), EMBEDDING_DIMENSION)

from gensim.models import KeyedVectors

model_glove_gensim = KeyedVectors.load_word2vec_format("glove.6B.100d.txt", binary=False, no_header=True)
```

In [3]:
```python
from scipy.stats import spearmanr
from sklearn.metrics import mean_squared_error


# Evaluate Analogies
# TODO: Find Semantic Accuracy score using capital-common-countries section of word-test.v1.txt
# TODO: Find Syntactic Accuracy score using gram7-past-tense section of word-test.v1.txt
def evaluate_analogies(model, word2index, analogy_data, is_gensim=False):
    semantic_correct, semantic_total = 0, 0
    syntactic_correct, syntactic_total = 0, 0
    section = None

    for line in analogy_data:
        if line.startswith(":"):
            section = line.strip().lower()
        else:
            # Split the line into words and convert to lowercase
            words = line.strip().lower().split()
            # Check if all words are in the vocabulary
            if all(word in word2index for word in words):
                if is_gensim:
                    predicted = model.most_similar(positive=[words[1], words[2]], negative=[words[0]], topn=1)[0][0]
                else:
                    # Get the indices of the words
                    a, b, c, expected = [word2index[word] for word in words]
                    # Calculate the vector for the predicted word
                    vec = (
                        model.embedding_v(torch.tensor([b])).detach()
                        - model.embedding_v(torch.tensor([a])).detach()
                        + model.embedding_v(torch.tensor([c])).detach()
                    )
                    # Compute similarities and find the word with the highest similarity
                    similarities = torch.matmul(model.embedding_v.weight, vec.squeeze())
                    predicted = torch.argmax(similarities).item()
```

```python
                # Check if the predicted word matches the expected word
                if predicted == words[3]:
                    if "capital-common-countries" in section:
                        semantic_correct += 1
                    elif "gram7-past-tense" in section:
                        syntactic_correct += 1
                # Update the total counts for semantic and syntactic sections
                if "capital-common-countries" in section:
                    semantic_total += 1
                elif "gram7-past-tense" in section:
                    syntactic_total += 1

    # Calculate accuracy for semantic and syntactic sections
    semantic_accuracy = semantic_correct / semantic_total if semantic_total else 0
    syntactic_accuracy = syntactic_correct / syntactic_total if syntactic_total else 0
    print(f"  Semantic Accuracy (Capital-Common-Countries): {semantic_accuracy:.4f}")
    print(f"  Syntactic Accuracy (Gram7-Past-Tense): {syntactic_accuracy:.4f}")


# Evaluate Similarity
# TODO: Find the correlation between your models' dot product and the provided similarity metrics (spearmanr). Output is MSE score.
# TODO: Assess if your embeddings correlate with human judgment of word similarity against the wordsim_similarity_goldstandard.txt file.
def evaluate_similarity(model, word2index, similarity_data, is_gensim=False):
    predictions = []
    ground_truth = []
    for line in similarity_data:
        # Split the line into words and the similarity score
        word1, word2, score = line.strip().split()
        # Check if both words are in the vocabulary
        if word1 in word2index and word2 in word2index:
            if is_gensim:
                similarity = model.similarity(word1, word2)
            else:
                vec1 = model.embedding_v(torch.tensor([word2index[word1]])).detach()
                vec2 = model.embedding_v(torch.tensor([word2index[word2]])).detach()
                # Calculate the dot product similarity
                similarity = torch.dot(vec1.squeeze(), vec2.squeeze()).item()
            predictions.append(similarity)
            ground_truth.append(float(score))
    # Calculate Spearman correlation between predictions and ground truth
    spearman_corr, _ = spearmanr(predictions, ground_truth)
    # Calculate Mean Squared Error (MSE) between predictions and ground truth
    mse_score = mean_squared_error(ground_truth, predictions)
    print(f"  Spearman Correlation: {spearman_corr:.4f}")
    print(f"  Mean Squared Error (MSE): {mse_score:.4f}")
```

In [4]:

```python
# Load analogy and similarity datasets
with open("word-test.v1.txt", "r") as file:
    analogy_data = file.readlines()

with open("wordsim_similarity_goldstandard.txt", "r") as file:
    similarity_data = file.readlines()

# Run evaluation
print("--- Skipgram Model ---")
evaluate_analogies(model_skipgram, word2index_skipgram, analogy_data)
evaluate_similarity(model_skipgram, word2index_skipgram, similarity_data)

print("--- Skipgram Negative Sampling Model ---")
evaluate_analogies(model_skipgram_neg, word2index_neg, analogy_data)
evaluate_similarity(model_skipgram_neg, word2index_neg, similarity_data)

print("--- GloVe PyTorch Model ---")
evaluate_analogies(model_glove, word2index_glove, analogy_data)
evaluate_similarity(model_glove, word2index_glove, similarity_data)

print("--- GloVe Gensim Model ---")
evaluate_analogies(model_glove_gensim, model_glove_gensim.key_to_index, analogy_data, is_gensim=True)
evaluate_similarity(model_glove_gensim, model_glove_gensim.key_to_index, similarity_data, is_gensim=True)
```

```
--- Skipgram Model ---
  Semantic Accuracy (Capital-Common-Countries): 0.0000
  Syntactic Accuracy (Gram7-Past-Tense): 0.0000
  Spearman Correlation: 0.3420
  Mean Squared Error (MSE): 24.1058
--- Skipgram Negative Sampling Model ---
  Semantic Accuracy (Capital-Common-Countries): 0.0000
  Syntactic Accuracy (Gram7-Past-Tense): 0.0000
  Spearman Correlation: 0.3238
  Mean Squared Error (MSE): 165.4730
--- GloVe PyTorch Model ---
  Semantic Accuracy (Capital-Common-Countries): 0.0000
  Syntactic Accuracy (Gram7-Past-Tense): 0.0000
  Spearman Correlation: 0.2178
  Mean Squared Error (MSE): 30.8277
--- GloVe Gensim Model ---
  Semantic Accuracy (Capital-Common-Countries): 0.9387
  Syntactic Accuracy (Gram7-Past-Tense): 0.5545
  Spearman Correlation: 0.6019
  Mean Squared Error (MSE): 27.8562
```

In [46]:

```python
import pandas as pd


# Find Top 10 Most Similar Words Based on Dot Product
def find_top_similar_words(model, word2index, index2word, input_word, top_n=10, is_gensim=False):
    if is_gensim:
        if input_word not in model:
            print(f"'{input_word}' not in vocabulary.")
            return
        similar_words = model.most_similar(input_word, topn=top_n)
        print(f"Top {top_n} words similar to '{input_word}':")
        for word, similarity in similar_words:
            print(f"Model: {'Gensim':20s}  Word: {word:15s}  Similarity: {similarity:.4f}")
    else:
        if input_word not in word2index:
            print(f"'{input_word}' not in vocabulary.")
            return
        input_vec = model.embedding_v(torch.tensor([word2index[input_word]])).detach()
        similarities = torch.matmul(model.embedding_v.weight, input_vec.squeeze())
        probabilities = torch.softmax(similarities, dim=0)
        top_indices = torch.topk(probabilities, top_n + 1).indices.tolist()[1:]
        print(f"Top {top_n} words similar to '{input_word}':")
        for idx in top_indices:
            print(f"Model: {model.__class__.__name__:20s}  Word: {index2word[idx]:15s}  Similarity: {similarities[idx]:.4f}")
    print("-----------------------------------------------------------------------")


input_word = random.choice(vocab)

find_top_similar_words(model_skipgram, word2index_skipgram, vocab_skipgram, input_word)
find_top_similar_words(model_skipgram_neg, word2index_neg, vocab_neg, input_word)
find_top_similar_words(model_glove, word2index_glove, vocab_glove, input_word)
find_top_similar_words(model_glove_gensim, model_glove_gensim.key_to_index, None, input_word, is_gensim=True)
```

```
Top 10 words similar to 'responsible':
Model: Skipgram             Word: darby             Similarity: 4.3564
Model: Skipgram             Word: prevailed         Similarity: 4.2553
Model: Skipgram             Word: rouen             Similarity: 4.1849
Model: Skipgram             Word: boveri            Similarity: 4.0725
Model: Skipgram             Word: quantum           Similarity: 3.9758
Model: Skipgram             Word: underwoods        Similarity: 3.9561
Model: Skipgram             Word: gkn               Similarity: 3.9252
Model: Skipgram             Word: herrhausen        Similarity: 3.8366
Model: Skipgram             Word: studios           Similarity: 3.8184
Model: Skipgram             Word: erik              Similarity: 3.8103
----------------------------------------------------------------------
Top 10 words similar to 'responsible':
Model: SkipgramNegSampling   Word: integral          Similarity: 49.4248
Model: SkipgramNegSampling   Word: prevailed         Similarity: 48.7580
Model: SkipgramNegSampling   Word: abiding           Similarity: 47.4386
Model: SkipgramNegSampling   Word: borne             Similarity: 47.1484
Model: SkipgramNegSampling   Word: urgent            Similarity: 46.2675
Model: SkipgramNegSampling   Word: jauppi            Similarity: 46.0135
Model: SkipgramNegSampling   Word: teeth             Similarity: 45.9261
Model: SkipgramNegSampling   Word: confusion         Similarity: 45.5945
Model: SkipgramNegSampling   Word: guesstimating     Similarity: 45.5513
Model: SkipgramNegSampling   Word: additive          Similarity: 45.5133
----------------------------------------------------------------------
Top 10 words similar to 'responsible':
Model: GloVe                Word: tank              Similarity: 21.1896
Model: GloVe                Word: stat              Similarity: 18.7223
Model: GloVe                Word: steinhardt        Similarity: 18.0555
Model: GloVe                Word: detroit           Similarity: 17.9031
Model: GloVe                Word: apples            Similarity: 17.8443
Model: GloVe                Word: alleviate         Similarity: 17.8258
Model: GloVe                Word: aiming            Similarity: 17.7971
Model: GloVe                Word: walker            Similarity: 17.5778
Model: GloVe                Word: clarifies         Similarity: 17.4171
Model: GloVe                Word: reinforced        Similarity: 17.3692
----------------------------------------------------------------------
Top 10 words similar to 'responsible':
Model: Gensim               Word: involved          Similarity: 0.8070
Model: Gensim               Word: responsibility    Similarity: 0.7752
Model: Gensim               Word: committed         Similarity: 0.7292
Model: Gensim               Word: planning          Similarity: 0.7224
Model: Gensim               Word: involvement       Similarity: 0.6700
Model: Gensim               Word: accountable       Similarity: 0.6643
Model: Gensim               Word: actions           Similarity: 0.6632
Model: Gensim               Word: concerned         Similarity: 0.6630
Model: Gensim               Word: overseeing        Similarity: 0.6613
Model: Gensim               Word: charged           Similarity: 0.6611
----------------------------------------------------------------------
```

In [112…

```python
import random


# Analogy Prediction Function
def predict_analogy(model, word2index, word_a, word_b, word_c, is_gensim=False):
    if is_gensim:
        if all(word in model for word in [word_a, word_b, word_c]):
            result = model.most_similar(positive=[word_b, word_c], negative=[word_a], topn=1)[0][0]
            print(f"{'Gensim':>20s}: '{word_a}' is to '{word_b}' as '{word_c}' is to '{result}'")
        else:
            print("One or more input words are not in the vocabulary.")
    else:
        if all(word in word2index for word in [word_a, word_b, word_c]):
            a, b, c = [word2index[word] for word in [word_a, word_b, word_c]]
            vec = (
                model.embedding_v(torch.tensor([b])).detach()
                - model.embedding_v(torch.tensor([a])).detach()
                + model.embedding_v(torch.tensor([c])).detach()
            )
            similarities = torch.matmul(model.embedding_v.weight, vec.squeeze())
            predicted_idx = torch.argmax(similarities).item()
            predicted_word = vocab_glove[predicted_idx]
            print(f"{model.__class__.__name__:>20s}: '{word_a}' is to '{word_b}' as '{word_c}' is to '{predicted_word}'")
        else:
            print("One or more input words are not in the vocabulary.")


first_word = random.choice(vocab)
second_word = random.choice(vocab)
third_word = random.choice(vocab)

predict_analogy(model_skipgram, word2index_skipgram, first_word, second_word, third_word)
predict_analogy(model_skipgram_neg, word2index_neg, first_word, second_word, third_word)
predict_analogy(model_glove, word2index_glove, first_word, second_word, third_word)
predict_analogy(model_glove_gensim, model_glove_gensim.key_to_index, first_word, second_word, third_word, is_gensim=True)
```

```
           Skipgram: 'rebound' is to 'belgian' as 'ottawa' is to 'salina'
SkipgramNegSampling: 'rebound' is to 'belgian' as 'ottawa' is to 'instance'
              GloVe: 'rebound' is to 'belgian' as 'ottawa' is to 'ottawa'
             Gensim: 'rebound' is to 'belgian' as 'ottawa' is to 'quebec'
```

In [111…

```python
# Function to Generate Sentence from a Single Input Word
def generate_sentence(model, word2index, index2word, input_word, length=9, is_gensim=False):
    if is_gensim:
        if input_word not in model:
            print(f"'{input_word}' not in vocabulary.")
            return
        sentence = [input_word]
        current_word = input_word
        previous_word = np.random.choice(list(model.key_to_index.keys()))
        for _ in range(length):
            next_word = model.most_similar(positive=[current_word], negative=[previous_word], topn=1)[0][0]
            sentence.append(next_word)
            previous_word = current_word
            current_word = next_word
        print("Generated Sentence (Gensim):", " ".join(sentence))
    else:
        if input_word not in word2index:
            print(f"'{input_word}' not in vocabulary.")
            return
        sentence = [input_word]
        current_idx = word2index[input_word]
        for _ in range(length):
            input_vec = model.embedding_v(torch.tensor([current_idx])).detach()
            similarities = torch.matmul(model.embedding_v.weight, input_vec.squeeze())
            next_idx = torch.argmax(similarities).item() + 1
            next_word = index2word[next_idx]
            sentence.append(next_word)
            current_idx = next_idx
        print(f"Generated Sentence ({model.__class__.__name__}):", " ".join(sentence))


input_word = random.choice(vocab)

generate_sentence(model_skipgram, word2index_skipgram, vocab_skipgram, input_word)
generate_sentence(model_skipgram_neg, word2index_neg, vocab_neg, input_word)
generate_sentence(model_glove, word2index_glove, vocab_glove, input_word)
generate_sentence(model_glove_gensim, model_glove_gensim.key_to_index, None, input_word, is_gensim=True)
```

```
Generated Sentence (Skipgram): imbalance congress takes castings sample weizsaecker ziering swissair fairfax britoil
Generated Sentence (SkipgramNegSampling): imbalance congress adobe toys welbilt laurentiis whale sooner sanchez sc
Generated Sentence (GloVe): imbalance congress encountered beneath cote beneficiaries equivalents specializes diablo innovex
Generated Sentence (Gensim): imbalance increase 50 125 239.7 zety chevedden john as undesirable
```