st125171_A1_That_s_What_I_LIKE

January 19, 2025

1 Load Libraries

[1]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import pickle
import math
import time
import os

from collections import Counter
```

2 Task 1: Preparation and Training

2.0.1 Objective:

Build upon the code discussed in class to enhance understanding and implementation of Word2Vec and GloVe algorithms. The task emphasizes creating and modifying these algorithms without relying on pre-built solutions from the internet.

2.0.2 1. Read and Understand:

- **Word2Vec Paper**: Study the foundational concepts and techniques outlined in the original Word2Vec paper.

- **GloVe Paper**: Comprehend the methodology and innovations introduced in the GloVe paper.

2.0.3 2. Code Modifications:

a. Modify the Word2Vec (with and without negative sampling) and GloVe algorithms as discussed in the lab lecture.

- **Implementation Details**:
  - Use a real-world corpus for training, such as categorizing news data from the **nltk dataset**.
  - Source the dataset from reputable public databases or repositories and include proper citations in the documentation.

b. Create a Function for Dynamic Window Size Modification:

- Develop a function to enable the dynamic adjustment of the window size during training.
- **Default Window Size**: Set the default window size to 2.

2.1 Additional Notes:

- **Documentation**: Ensure that all dataset sources and citations are included in the documentation to maintain academic integrity.
- **Evaluation**: Implement and validate the modifications on the selected corpus to verify the functionality of the updated algorithms.

2.2 Load data - Corpus and Tokenization

[2]:
```
import nltk

#download nltk corpus
nltk.download()
```

```
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml

2025-01-18 22:53:27.259 python[14899:607162] +[IMKClient subclass]: chose
IMKClient_Modern
2025-01-18 22:53:27.259 python[14899:607162] +[IMKInputSession subclass]: chose
IMKInputSession_Modern
```

[2]:
```
True
```

In this assignment we are asked to use a real world corpus from the nltk. Hence for this assignment I am using the brown corpus with the category of news as suggested in the instruction.

BROWN CORPUS (source: The below text has been copied from Wikipedia) The Brown University Standard Corpus of Present-Day American English, better known as simply the Brown Corpus, is an electronic collection of text samples of American English, the first major structured corpus of varied genres. This corpus first set the bar for the scientific study of the frequency and distribution of word categories in everyday language use. Compiled by Henry Kučera and W. Nelson Francis at Brown University, in Rhode Island, it is a general language corpus containing 500 samples of English, totaling roughly one million words, compiled from works published in the United States in 1961.

Manual of Brown Corpus: http://clu.uni.no/icame/manuals/
NLTK Corpora (12. Brown Corpus): https://www.nltk.org/nltk_data/
id: brown; size: 3314357; author: W. N. Francis and H. Kucera; copyright: ; license: May be used

for non-commercial purposes.;
Download Brown Corpus: https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/brown.zip
Source of Brown Corpus: http://www.hit.uib.no/icame/brown/bcm.html

Categorizing and Tagging Words in nltk: https://www.nltk.org/book/ch05.html

Importing the corpus and tokenization

[3]:
```
from nltk.corpus import brown
corpus = brown.sents(categories='news')

#Here from the corpus we are selecting the first 1000000 words
corpus = corpus[:1000000]

#lowercase the word in corpus
corpus = [[word.lower() for word in sent] for sent in corpus]
```

2.3 Numeralization

[4]:
```
#find unique words
flatten = lambda l: [item for sublist in l for item in sublist]

#assign unique integer
vocabs = list(set(flatten(corpus))) #all the words we have in the system - <UNK>

#append UNKNOWN token to vocabs
vocabs.append('<UNK>')
```

[5]:
```
#create handy mapping between integer and word
word2index = {v:idx for idx, v in enumerate(vocabs)}
word2index['dog']
```

[5]:
```
11852
```

[6]:
```
index2word = {v:k for k, v in word2index.items()}
index2word[5]
```

[6]:
```
'0'
```

Prepare train data random batch function with window size

[7]:
```
#create pairs of center word, and outside word

def random_batch(batch_size, corpus, window_size):

    skipgrams = []

    #loop each corpus
    for doc in corpus:
```

```
        #look from the 2nd word until second last word
        for i in range(window_size, len(doc)-window_size):
            #center word
            center = word2index[doc[i]]
            #outside words = 2 words
            outside = []
            for j in range(i-window_size, i+window_size+1):
                outside.append(word2index[doc[j]])
            #for each of these two outside words, we gonna append to a list
            for each_out in outside:
                skipgrams.append([center, each_out])
                #center, outside1;   center, outside2

    random_index = np.random.choice(range(len(skipgrams)), batch_size,␣
↪replace=False)

    inputs, labels = [], []
    for index in random_index:
        inputs.append([skipgrams[index][0]])
        labels.append([skipgrams[index][1]])

    return np.array(inputs), np.array(labels)
```

3 Task 2. Model Comparison and Analysis

1) Compare Skip-gram, Skip-gram negative sampling, GloVe models on training loss, training time. (1points)

2) Use Word analogies dataset 3 to calucalte between syntactic and semantic accuracy, similar to the methods in the Word2Vec and GloVe paper. (1 points)

- **Note** : using only capital-common-countries for semantic and past-tense for syntactic.
- **Note** : Do not be surprised if you achieve 0% accuracy in these experiments, as this may be due to the limitations of our corpus. If you are curious, you can try the same experiments with a pre-trained GloVe model from the Gensim library for a comparison.

3) Use the similarity dataset4to find the correlation between your models' dot product and the provided similarity metrics. (from scipy.stats import spearmanr) Assess if your embeddings correlate with human judgment. (1 points)

4 Word2Vec (with and without negative sampling)

Unigram distribution

[8]:
```
z = 0.001
```

[9]:
```python
#count
word_count = Counter(flatten(corpus))
word_count

#get the total number of words
num_total_words = sum([c for w, c in word_count.items()])
num_total_words
```

[9]: 100554

[10]:
```python
unigram_table = []

for v in vocabs:
    uw = word_count[v] / num_total_words
    uw_alpha = int((uw ** 0.75) / z)
    unigram_table.extend([v] * uw_alpha)
```

Model

[11]:
```python
def prepare_sequence(seq, word2index):
    idxs = list(map(lambda w: word2index[w] if word2index.get(w) is not None␣
 ↪else word2index["<UNK>"], seq))
    return torch.LongTensor(idxs)
```

[12]:
```python
import random

def negative_sampling(targets, unigram_table, k):
    batch_size = targets.shape[0]
    neg_samples = []
    for i in range(batch_size):  #(1, k)
        target_index = targets[i].item()
        nsample      = []
        while (len(nsample) < k):
            neg = random.choice(unigram_table)
            if word2index[neg] == target_index:
                continue
            nsample.append(neg)
        neg_samples.append(prepare_sequence(nsample, word2index).reshape(1, -1))

    return torch.cat(neg_samples) #batch_size, k
```

4.0.1 Word2Vec (without negative sampling)

[13]:
```python
class Skipgram(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(Skipgram, self).__init__()
        self.embedding_center  = nn.Embedding(voc_size, emb_size)
```

```
        self.embedding_outside = nn.Embedding(voc_size, emb_size)

    def forward(self, center, outside, all_vocabs):
        center_embedding     = self.embedding_center(center)  #(batch_size, 1,␣
↪emb_size)
        outside_embedding    = self.embedding_center(outside) #(batch_size, 1,␣
↪emb_size)
        all_vocabs_embedding = self.embedding_center(all_vocabs) #(batch_size,␣
↪voc_size, emb_size)

        top_term = torch.exp(outside_embedding.bmm(center_embedding.
↪transpose(1, 2)).squeeze(2))
        #batch_size, 1, emb_size) @ (batch_size, emb_size, 1) = (batch_size, 1,␣
↪1) = (batch_size, 1)

        lower_term = all_vocabs_embedding.bmm(center_embedding.transpose(1, 2)).
↪squeeze(2)
        #batch_size, voc_size, emb_size) @ (batch_size, emb_size, 1) =␣
↪(batch_size, voc_size, 1) = (batch_size, voc_size)

        lower_term_sum = torch.sum(torch.exp(lower_term), 1)  #(batch_size, 1)

        loss = -torch.mean(torch.log(top_term / lower_term_sum))  #scalar

        return loss
```

4.0.2 Word2Vec (with negative sampling)

[14]:
```
class SkipgramNeg(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(SkipgramNeg, self).__init__()
        self.embedding_center  = nn.Embedding(voc_size, emb_size)
        self.embedding_outside = nn.Embedding(voc_size, emb_size)
        self.logsigmoid        = nn.LogSigmoid()

    def forward(self, center, outside, negative):
        #center, outside:  (bs, 1)
        #negative       :  (bs, k)

        center_embed   = self.embedding_center(center) #(bs, 1, emb_size)
        outside_embed  = self.embedding_outside(outside) #(bs, 1, emb_size)
        negative_embed = self.embedding_outside(negative) #(bs, k, emb_size)

        uovc           = outside_embed.bmm(center_embed.transpose(1, 2)).
↪squeeze(2) #(bs, 1)
```

```
        ukvc            = -negative_embed.bmm(center_embed.transpose(1, 2)).
↪squeeze(2) #(bs, k)
        ukvc_sum        = torch.sum(ukvc, 1).reshape(-1, 1) #(bs, 1)

        loss            = self.logsigmoid(uovc) + self.logsigmoid(ukvc_sum)

        return -torch.mean(loss)
```

Training Since, I am doing this in MacBook hence I am using the MAC 'mps' | 'cpu' whichever is available.

[15]:
```
# Set device (MPS if available)
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
device
```

[15]:
```
device(type='mps')
```

[16]:
```
batch_size = 2
emb_size = 2
window_size = 2 #default window_size
voc_size = len(vocabs)
```

[17]:
```
#prepare all vocabs
all_vocabs = prepare_sequence(list(vocabs), word2index).expand(batch_size,␣
↪voc_size).to(device)
all_vocabs
```

[17]:
```
tensor([[    0,     1,     2,  …, 13110, 13111, 13112],
        [    0,     1,     2,  …, 13110, 13111, 13112]], device='mps:0')
```

4.0.3 Word2Vec Skipgram

[18]:
```
model_skipgram = Skipgram(voc_size, emb_size).to(device)
optimizer = optim.Adam(model_skipgram.parameters(), lr=0.001)
```

[19]:
```
num_epochs = 10000
start_time = time.time()

for epoch in range(num_epochs):

    #getbatch
    input_batch, label_batch = random_batch(batch_size, corpus, window_size)
    input_tensor = torch.LongTensor(input_batch).to(device)
    label_tensor = torch.LongTensor(label_batch).to(device)

    #predict
    loss = model_skipgram(input_tensor, label_tensor, all_vocabs)
```

```
    #backprogate
    optimizer.zero_grad()
    loss.backward()

    #update alpha
    optimizer.step()

    #print the loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1:6.0f} | Loss: {loss:2.6f}")

print(f"Training time: {time.time()-start_time}")
```

```
Epoch   1000 | Loss: 9.694761
Epoch   2000 | Loss: 8.396974
Epoch   3000 | Loss: 9.138665
Epoch   4000 | Loss: 9.502065
Epoch   5000 | Loss: 9.731103
Epoch   6000 | Loss: 9.530214
Epoch   7000 | Loss: 9.079416
Epoch   8000 | Loss: 9.076571
Epoch   9000 | Loss: 9.572384
Epoch  10000 | Loss: 9.767000
Training time: 642.6175730228424
```

4.0.4 Word2Vec Skipgram Negative

[20]:
```
model_skipgram_neg = SkipgramNeg(voc_size, emb_size).to(device)
optimizer = optim.Adam(model_skipgram_neg.parameters(), lr=0.001)
```

[21]:
```
num_epochs = 10000
k = 5
start_time = time.time()

for epoch in range(num_epochs):

    #get batch
    input_batch, label_batch = random_batch(batch_size, corpus, window_size)
    input_tensor = torch.LongTensor(input_batch).to(device)
    label_tensor = torch.LongTensor(label_batch).to(device)

    #predict
    neg_samples = negative_sampling(label_tensor, unigram_table, k).to(device)
    loss = model_skipgram_neg(input_tensor, label_tensor, neg_samples)

    #backprogate
```

```
    optimizer.zero_grad()
    loss.backward()

    #update alpha
    optimizer.step()

    #print the loss
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1:6.0f} | Loss: {loss:2.6f}")

print(f"Training time: {time.time()-start_time}")
```

```
Epoch   1000 | Loss: 2.038834
Epoch   2000 | Loss: 3.213029
Epoch   3000 | Loss: 0.510042
Epoch   4000 | Loss: 1.598014
Epoch   5000 | Loss: 1.543426
Epoch   6000 | Loss: 1.063593
Epoch   7000 | Loss: 2.304073
Epoch   8000 | Loss: 2.273893
Epoch   9000 | Loss: 2.072499
Epoch  10000 | Loss: 1.027903
Training time: 651.9084329605103
```

4.0.5 GloVe (Scratch)

Let's work on implementation of GloVE.

Build Co-occurence Matrix X Here, we need to count the co-occurence of two words given some window size. We gonna use window size of 2.

[22]:
```
X_i = Counter(flatten(corpus))
```

[23]:
```
skip_grams = []

for doc in corpus:
    for i in range(1, len(doc)-window_size):
        center = doc[i]
        outside = [doc[i-window_size], doc[i-1], doc[i+1], doc[i+window_size]]
        for each_out in outside:
            skip_grams.append((center, each_out))
```

[24]:
```
X_ik_skipgrams = Counter(skip_grams)
```

Weighting function GloVe includes a weighting function to scale down too frequent words.

[25]:
```
def weighting(w_i, w_j, X_ik):
```

```
    #check whether the co-occurences between w_i and w_j is available
    try:
        x_ij = X_ik[(w_i, w_j)]
        #if not exist, then set to 1 "laplace smoothing"
    except:
        x_ij = 1

    #set xmax
    x_max = 100
    #set alpha
    alpha = 0.75

    #if co-ocurrence does not exceeed xmax, then just multiply with some alpha
    if x_ij < x_max:
        result = (x_ij / x_max)**alpha
    #otherwise, set to 1
    else:
        result = 1

    return result
```

[26]:
```
from itertools import combinations_with_replacement

X_ik = {} #keeping the co-occurences
weighting_dic = {} #already scale the co-occurences using the weighting function

for bigram in combinations_with_replacement(vocabs, 2):
    if X_ik_skipgrams.get(bigram):  #if the pair exists in our corpus
        co = X_ik_skipgrams[bigram]
        X_ik[bigram] = co + 1 #for stability
        X_ik[(bigram[1], bigram[0])] = co + 1 #basically apple, banana =
 ↪banana, apple
    else:
        pass

    weighting_dic[bigram] = weighting(bigram[0], bigram[1], X_ik)
    weighting_dic[(bigram[1], bigram[0])] = weighting(bigram[1], bigram[0],
 ↪X_ik)
```

Prepare train data

[27]:
```
def random_batch(batch_size, word_sequence, skip_grams, X_ik, weighting_dic):

    random_inputs, random_labels, random_coocs, random_weightings = [], [], [],
 ↪[]

    #convert our skipgrams to id
```

```python
    skip_grams_id = [(word2index[skip_gram[0]], word2index[skip_gram[1]]) for␣
↪skip_gram in skip_grams]

    #randomly choose indexes based on batch size
    random_index = np.random.choice(range(len(skip_grams_id)), batch_size,␣
↪replace=False)

    #get the random input and labels
    for index in random_index:
        random_inputs.append([skip_grams_id[index][0]])
        random_labels.append([skip_grams_id[index][1]])
        #coocs
        pair = skip_grams[index] #e.g., ('banana', 'fruit')
        try:
            cooc = X_ik[pair]
        except:
            cooc = 1
        random_coocs.append([math.log(cooc)])

        #weightings
        weighting = weighting_dic[pair]
        random_weightings.append([weighting])

    return np.array(random_inputs), np.array(random_labels), np.
↪array(random_coocs), np.array(random_weightings)
```

Model

[28]:
```python
class Glove(nn.Module):

    def __init__(self, voc_size, emb_size):
        super(Glove, self).__init__()
        self.embedding_center  = nn.Embedding(voc_size, emb_size)
        self.embedding_outside = nn.Embedding(voc_size, emb_size)

        self.center_bias       = nn.Embedding(voc_size, 1)
        self.outside_bias      = nn.Embedding(voc_size, 1)

    def forward(self, center, outside, coocs, weighting):
        center_embeds  = self.embedding_center(center) #(batch_size, 1,␣
↪emb_size)
        outside_embeds = self.embedding_outside(outside) #(batch_size, 1,␣
↪emb_size)

        center_bias    = self.center_bias(center).squeeze(1)
        target_bias    = self.outside_bias(outside).squeeze(1)
```

```
        inner_product  = outside_embeds.bmm(center_embeds.transpose(1, 2)).
↪squeeze(2)
        #(batch_size, 1, emb_size) @ (batch_size, emb_size, 1) = (batch_size,␣
↪1, 1) = (batch_size, 1)

        loss = weighting * torch.pow(inner_product + center_bias + target_bias␣
↪- coocs, 2)

        return torch.sum(loss)
```

Training

[29]:
```
batch_size     = 10 # mini-batch size
embedding_size = 2 #so we can later plot
model_glove_scratch = Glove(voc_size, embedding_size).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_glove_scratch.parameters(), lr=0.001)
```

[30]:
```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

[31]:
```
num_epochs = 10000
for epoch in range(num_epochs):

    start = time.time()

    input_batch, target_batch, cooc_batch, weighting_batch =␣
↪random_batch(batch_size, corpus, skip_grams, X_ik, weighting_dic)
    input_batch  = torch.LongTensor(input_batch).to(device)        ␣
↪#[batch_size, 1]
    target_batch = torch.LongTensor(target_batch).to(device)       ␣
↪#[batch_size, 1]
    cooc_batch   = torch.FloatTensor(cooc_batch).to(device)        ␣
↪#[batch_size, 1]
    weighting_batch = torch.FloatTensor(weighting_batch).to(device)␣
↪#[batch_size, 1]

    optimizer.zero_grad()
    loss = model_glove_scratch(input_batch, target_batch, cooc_batch,␣
↪weighting_batch)

    loss.backward()
    optimizer.step()
```

```
    end = time.time()

    epoch_mins, epoch_secs = epoch_time(start, end)

    if (epoch + 1) % 1000 == 0:
        print(f"Epoch: {epoch + 1} | cost: {loss:.6f} | time: {epoch_mins}m␣
↪{epoch_secs}s")

print(f"Total Training time: {time.time()-start_time}")
```

```
Epoch: 1000 | cost: 35.283993 | time: 0m 0s
Epoch: 2000 | cost: 21.909081 | time: 0m 0s
Epoch: 3000 | cost: 17.584835 | time: 0m 0s
Epoch: 4000 | cost: 13.305610 | time: 0m 0s
Epoch: 5000 | cost: 4.621394 | time: 0m 0s
Epoch: 6000 | cost: 4.300721 | time: 0m 0s
Epoch: 7000 | cost: 11.993736 | time: 0m 0s
Epoch: 8000 | cost: 6.335316 | time: 0m 0s
Epoch: 9000 | cost: 5.531918 | time: 0m 0s
Epoch: 10000 | cost: 0.672632 | time: 0m 0s
Total Training time: 1214.7566788196564
```

4.0.6 GloVe (Gensim)

For looking at word vectors, we'll use **Gensim**. **Gensim** isn't really a deep learning package. It's a package for for word and text similarity modeling, which started with (LDA-style) topic models and grew into SVD and neural word representations. But its efficient and scalable, and quite widely used. We gonna use **GloVe** embeddings, downloaded at the Glove page. They're inside this zip file

[32]:
```
from gensim.test.utils import datapath
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

#you have to put this file in some python/gensim directory; just run it and it␣
↪will inform where to put....
glove_file = datapath(os.path.abspath('word_test/glove.6B.100d.txt'))  #search␣
↪on the google
model_glove_gensim = KeyedVectors.load_word2vec_format(glove_file,␣
↪binary=False, no_header=True)
```

Use Word analogies dataset 3 to calculate between syntactic and semantic accuracy, similar to the methods in the Word2Vec and GloVe paper.

[39]:
```
def comp_embeddings(model, vocabs):
    embeds = {}
```

```
    device = torch.device("cpu")
    model = model.to(device)

    for word in vocabs:
        try:
            index = word2index[word]
        except:
            index = word2index['<UNK>']

        word_idx = torch.LongTensor([word2index[word]])

        embed_c = model.embedding_center(word_idx)
        embed_o = model.embedding_outside(word_idx)
        embed   = (embed_c + embed_o) / 2
        embed = embed[0][0].item(), embed[0][1].item()
        embeds[word] = np.array(embed)

    return embeds
```

[40]:
```
def get_embed(embeddings, word):
    try:
        index = word2index[word]
    except:
        word = '<UNK>'

    return embeddings[word]
```

[41]:
```
# find the embeddings from each of our model
emb_skipgram = comp_embeddings(model_skipgram, vocabs)
emb_skipgram_neg = comp_embeddings(model_skipgram_neg, vocabs)
emb_glove_scratch = comp_embeddings(model_glove_scratch, vocabs)
```

[67]:
```
from pathlib import Path
import pickle

# Define embeddings dictionary
embeds_dict = {
    "emb_skipgram": emb_skipgram,
    "emb_skipgram_neg": emb_skipgram_neg,
    "emb_glove_scratch": emb_glove_scratch
}

# Define the directory for saving embeddings
output_dir = Path("app/pickle")
output_dir.mkdir(parents=True, exist_ok=True)  # Ensure directory exists

# Save each embedding to a separate file
```

```python
for name, embed in embeds_dict.items():
    file_path = output_dir / f"{name}.pickle"
    try:
        with file_path.open("wb") as f:
            pickle.dump(embed, f)
        print(f"Saved {name} embeddings to {file_path}")
    except Exception as e:
        print(f"Error saving {name}: {e}")
```

```
Saved emb_skipgram embeddings to app/pickle/emb_skipgram.pickle
Saved emb_skipgram_neg embeddings to app/pickle/emb_skipgram_neg.pickle
Saved emb_glove_scratch embeddings to app/pickle/emb_glove_scratch.pickle
```

[68]:
```python
print(f"Skipgram: {get_embed(emb_skipgram, 'greece')}")
print(f"Skipgram NEG: {get_embed(emb_skipgram_neg, 'greece')}")
print(f"GloVe: {get_embed(emb_glove_scratch, 'greece')}")
```

```
Skipgram: [-1.12307346  0.43248087]
Skipgram NEG: [-0.31465432 -0.21734357]
GloVe: [-0.62375081 -0.61236209]
```

Read analogy of "word-test.v1.txt"

[45]:
```python
def read_analogy_dataset(file_path):
    from collections import defaultdict

    analogy_dict = defaultdict(list)  # To store analogies grouped by categories

    try:
        with open(file_path, "r") as f:
            lines = f.read().splitlines()

        current_category = None
        for line in lines:
            line = line.strip()  # Remove surrounding whitespace
            if not line:  # Skip empty lines
                continue

            if line.startswith(': '):  # Category line
                current_category = line[2:].strip()
            elif current_category:  # Analogy line
                analogy_dict[current_category].append(line.split())

        return dict(analogy_dict)

    except FileNotFoundError:
        print(f"Error: File {file_path} not found.")
        return {}
    except Exception as e:
```

```
        print(f"An error occurred: {e}")
        return {}
```

[46]:
```
file_path = "word_test/word-test.v1.txt"
analogy_dict = read_analogy_dataset(file_path)

#Print first category and its analogies
if analogy_dict:
    first_category = next(iter(analogy_dict))
    print(f"Category: {first_category}")
    print("Analogies:", analogy_dict[first_category][:5])  # Show first 5␣
 ↪analogies
```

```
Category: capital-common-countries
Analogies: [['Athens', 'Greece', 'Baghdad', 'Iraq'], ['Athens', 'Greece',
'Bangkok', 'Thailand'], ['Athens', 'Greece', 'Beijing', 'China'], ['Athens',
'Greece', 'Berlin', 'Germany'], ['Athens', 'Greece', 'Bern', 'Switzerland']]
```

[47]:
```
# Access the 'capital-common-countries' section
capital = analogy_dict.get('capital-common-countries', [])

# To print the first 5 analogies from the section
capital[:5]
```

[47]:
```
[['Athens', 'Greece', 'Baghdad', 'Iraq'],
 ['Athens', 'Greece', 'Bangkok', 'Thailand'],
 ['Athens', 'Greece', 'Beijing', 'China'],
 ['Athens', 'Greece', 'Berlin', 'Germany'],
 ['Athens', 'Greece', 'Bern', 'Switzerland']]
```

[48]:
```
# Access the 'gram7-past-tense' section from analogy_dict
past_tense = analogy_dict.get('gram7-past-tense', [])

# Display the first 5 analogies
past_tense[:5]
```

[48]:
```
[['dancing', 'danced', 'decreasing', 'decreased'],
 ['dancing', 'danced', 'describing', 'described'],
 ['dancing', 'danced', 'enhancing', 'enhanced'],
 ['dancing', 'danced', 'falling', 'fell'],
 ['dancing', 'danced', 'feeding', 'fed']]
```

[49]:
```
capital[1]
```

[49]:
```
['Athens', 'Greece', 'Bangkok', 'Thailand']
```

[50]:
```
i = 1
embeddings = ['emb_skipgram', 'emb_skipgram_neg', 'emb_glove_scratch']
```

```
# Assuming the embeddings are stored as variables like `emb_skipgram`,␣
↪`emb_skipgram_neg`, and 'emb_glove_scratch'.
for embed_name in embeddings:
    # Dynamically fetch the embedding variable using globals()
    emb_w1 = get_embed(globals()[embed_name], capital[i][1].lower())
    emb_w2 = get_embed(globals()[embed_name], capital[i][0].lower())
    emb_w3 = get_embed(globals()[embed_name], capital[i][2].lower())

    y_pred = emb_w1 - emb_w2 + emb_w3
    print(f"Embedding: {embed_name}")
    print(f"y_pred: {y_pred}")
    print("=================================")
```

```
Embedding: emb_skipgram
y_pred: [-1.12307346  0.43248087]
=================================
Embedding: emb_skipgram_neg
y_pred: [-0.31465432 -0.21734357]
=================================
Embedding: emb_glove_scratch
y_pred: [-0.62375081 -0.61236209]
=================================
```

Cosine Similarity

[51]:
```
def cosine_similarity(A, B):
    dot_product = np.dot(A, B)
    norm_a = np.linalg.norm(A)
    norm_b = np.linalg.norm(B)
    similarity = dot_product / (norm_a * norm_b)
    return similarity
```

[52]:
```
# function to find the most similar word given the input vector
def get_most_similar(vector, embeddings):
    try:
        words = list(embeddings.keys())
    except:
        words = list(embeddings.key_to_index.keys())

    # Precompute norms of all embeddings to avoid redundant calculations
    norms = {word: np.linalg.norm(embedding) for word, embedding in embeddings.
↪items()}

    # Calculate similarities and keep the word with the highest similarity
    similarities = {}
    for word, embedding in embeddings.items():
```

```
        similarity = np.dot(vector, embedding) / (norms[word] * np.linalg.
↪norm(vector))
        similarities[word] = similarity

    return max(similarities, key=similarities.get)
```

[53]:
```
# function to find the most similar word given the input vector cosine_ranking
def cosine_ranking(vector, embeddings):
    try:
        words = list(embeddings.keys())
    except:
        words = list(embeddings.key_to_index.keys())

    # Precompute norms of all embeddings to avoid redundant calculations
    norms = {word: np.linalg.norm(embedding) for word, embedding in embeddings.
↪items()}

    similarities = {}
    for word, embedding in embeddings.items():
        similarity = np.dot(vector, embedding) / (norms[word] * np.linalg.
↪norm(vector))
        similarities[word] = similarity

    # Return the dictionary sorted by similarity values in descending order
    return dict(sorted(similarities.items(), key=lambda item: item[1],␣
↪reverse=True))
```

[54]:
```
# function to find semantic and syntactic accuracy
def find_accuracy(dataset, embeddings):
    matched_count = 0

    for data in dataset:
        row = [word.lower() for word in data]

        try:
            pred_y = get_embed(embeddings, row[1]) - get_embed(embeddings,␣
↪row[0]) + get_embed(embeddings, row[2])
            pred_word = get_most_similar(pred_y, embeddings)
        except:
            pred_word = embeddings.most_similar(positive=[row[1], row[2]],␣
↪negative=[row[0]])[0][0]

        if row[3] == pred_word:
            matched_count += 1

    return matched_count / len(dataset)
```

[55]:
```
skipgram_semantic_acc = find_accuracy(capital, emb_skipgram)
skipgram_syntactic_acc = find_accuracy(past_tense, emb_skipgram)

print("== Word2Vec Skipgram ==")
print(f"Semantic accuracy: {skipgram_semantic_acc}")
print(f"Syntactic accuracy: {skipgram_syntactic_acc}")
```

```
== Word2Vec Skipgram ==
Semantic accuracy: 0.0
Syntactic accuracy: 0.000641025641025641
```

[56]:
```
skipgram_neg_semantic_acc = find_accuracy(capital, emb_skipgram_neg)
skipgram_neg_syntactic_acc = find_accuracy(past_tense, emb_skipgram_neg)

print("== Word2Vec Skipgram (NEG) ==")
print(f"Semantic accuracy: {skipgram_neg_semantic_acc}")
print(f"Syntactic accuracy: {skipgram_neg_syntactic_acc}")
```

```
== Word2Vec Skipgram (NEG) ==
Semantic accuracy: 0.0
Syntactic accuracy: 0.000641025641025641
```

[57]:
```
glove_scratch_semantic_acc = find_accuracy(capital, emb_glove_scratch)
glove_scratch_syntactic_acc = find_accuracy(past_tense, emb_glove_scratch)

print("== GloVe (Scratch) ==")
print(f"Semantic accuracy: {glove_scratch_semantic_acc}")
print(f"Syntactic accuracy: {glove_scratch_syntactic_acc}")
```

```
== GloVe (Scratch) ==
Semantic accuracy: 0.0
Syntactic accuracy: 0.0
```

[58]:
```
glove_gensim_semantic_acc = find_accuracy(capital, model_glove_gensim)
glove_gensim_syntactic_acc = find_accuracy(past_tense, model_glove_gensim)

print("== GloVe (Gensim) ==")
print(f"Semantic accuracy: {glove_gensim_semantic_acc}")
print(f"Syntactic accuracy: {glove_gensim_syntactic_acc}")
```

```
== GloVe (Gensim) ==
Semantic accuracy: 0.9387351778656127
Syntactic accuracy: 0.5544871794871795
```

Similarity Correlation Use the similarity dataset 4 to find the correlation between your models' dot product and the provided similarity metrics. (from scipy.stats import spearmanr) Assess if your embeddings correlate with human judgment. (1 points)

[59]:
```python
#load the similarity dataset
word_sim = pd.read_csv('word_test/wordsim_similarity_goldstandard.txt', sep =␣
↪"\t", header = None, names = ['word_1', 'word_2', 'similarities'])
word_sim
```

[59]:
```
          word_1    word_2  similarities
0          tiger       cat          7.35
1          tiger     tiger         10.00
2          plane       car          5.77
3          train       car          6.31
4     television     radio          6.77
..           …         …            …
198      rooster    voyage          0.62
199         noon    string          0.54
200        chord     smile          0.54
201    professor  cucumber          0.31
202         king   cabbage          0.23

[203 rows x 3 columns]
```

[60]:
```python
# Dictionary to map column names to their respective embedding models or methods
embedding_models = {
    'skipgram_dot_product': emb_skipgram,
    'skipgram_neg_dot_product': emb_skipgram_neg,
    'glove_scratch_dot_product': emb_glove_scratch,
    'glove_gensim_dot_product': model_glove_gensim,
}
```

[61]:
```python
# Function to compute dot products
def compute_dot_product(embedding_model, word1, word2):
    if isinstance(embedding_model, dict):  # For custom embeddings like␣
↪skipgram or glove
        return np.dot(
            get_embed(embedding_model, word1.lower()),
            get_embed(embedding_model, word2.lower())
        )
    elif hasattr(embedding_model, '__getitem__'):  # For Gensim models or␣
↪similar
        return np.dot(embedding_model[word1.lower()], embedding_model[word2.
↪lower()])
    else:
        raise ValueError("Unsupported embedding model type.")
```

[62]:
```python
# Loop through each embedding type and compute the dot products
for column_name, model in embedding_models.items():
    word_sim[column_name] = word_sim.apply(
```

```
        lambda row: compute_dot_product(model, row['word_1'], row['word_2']),␣
↪axis=1
    )
```

[63]:
```
word_sim
```

[63]:
```
          word_1    word_2  similarities  skipgram_dot_product  \
0          tiger       cat          7.35              0.374398
1          tiger     tiger         10.00              0.374398
2          plane       car          5.77              0.171009
3          train       car          6.31             -0.100144
4     television     radio          6.77             -0.623952
..           …         …           …                   …
198      rooster    voyage          0.62              0.374398
199         noon    string          0.54             -0.323274
200        chord     smile          0.54              0.068867
201    professor  cucumber          0.31             -0.489723
202         king   cabbage          0.23             -0.174933

     skipgram_neg_dot_product  glove_scratch_dot_product  \
0                    0.448477                   0.485633
1                    0.448477                   0.485633
2                   -1.264918                  -0.161668
3                   -0.349281                   0.635021
4                   -0.150336                   0.366628
..                        …                          …
198                  0.448477                   0.485633
199                 -0.993186                  -0.530243
200                  0.126992                   0.376086
201                 -0.545512                  -0.588388
202                  0.693191                  -1.000759

     glove_gensim_dot_product
0                   15.629377
1                   32.800144
2                   24.047298
3                   25.472923
4                   34.689987
..                        …
198                  1.683646
199                  1.070593
200                  6.762520
201                 -0.230552
202                  1.400288

[203 rows x 7 columns]
```

[64]:
```python
from scipy.stats import spearmanr

# List of embedding similarity columns and their names
embedding_columns = {
    'Word2Vec Skipgram': 'skipgram_dot_product',
    'Word2Vec Skipgram (NEG)': 'skipgram_neg_dot_product',
    'GloVe (Scratch)': 'glove_scratch_dot_product',
    'GloVe (Gensim)': 'glove_gensim_dot_product',
    'Y_true': 'similarities'  # Adding for reference comparison
}

# Convert the wordsim similarities to numpy for efficiency
wordsim_sim = word_sim['similarities'].to_numpy()
```

[65]:
```python
# Compute Spearman correlations for each embedding similarity
print("=== Spearman correlations (MSE) ===")
for name, column in embedding_columns.items():
    if column == 'similarities':  # Skip self-correlation for Y_true
        correlation = 1.0
    else:
        similarity = word_sim[column].to_numpy()
        correlation = spearmanr(wordsim_sim, similarity).statistic
    print(f"{name}: {correlation}")
```

```
=== Spearman correlations (MSE) ===
Word2Vec Skipgram: 0.09847292793281937
Word2Vec Skipgram (NEG): 0.03240032525936758
GloVe (Scratch): 0.0681032757240688
GloVe (Gensim): 0.5430870624672256
Y_true: 1.0
```

Comparision of models on training loss and training time

Model	Window Size	Training Loss	Training Time	Syntactic Accuracy	Semantic Accuracy
Skipgram	2	9.767000	10min 42.5s	0.064%	0%
Skipgram (NEG)	2	1.027903	10min 51.9s	0.064%	0%
GloVe (Scratch)	2	0.672632	7min 15.7s	0%	0%
GloVe (Gensim)	-	-	-	55.44%	93.87%

Similarity correlation between models' dot product and the provided similarity metrics.

Model	Skipgram	NEG	GloVe	GloVe (gensim)	Y_true
MSE	0.09847292	0.03240032	0.06810327	0.54308706	1

5 Task 3. Search similar context - Web Application Development - Develop a simple website with an input box for search queries. (2 points)

1) Implement a function to compute the dot product between the input query and your corpus and retrieve the top 10 most similar context.

2) You may need to learn web frameworks like Flask or Django for this task. Web application can be accessed locally:
To deploy application first download repo from github (https://github.com/sachinmalego/NLP-A1-Thats-What-I-LIKE.git).
Open in VSCode and open terminal.
In the terminal type "python3 app.py". My local deployment address was "http://127.0.0.1:5000/" however your's might be different.
Go to browser and enter your local deployment server address to test the application.

Video of Working application:
https://drive.google.com/file/d/1Y0JF2sfW6w1TMBjQl3zMfmvAWBkNXSc8/view?usp=sharing

Screen shots of the working application is attached here with:
https://drive.google.com/drive/folders/1O_SWxxDTlojha1XfL1cS-F4V4ACpfVCT?usp=sharing

GloVe

GloVe Output

Skipgram

Skipgram Output

Skipgram NEG

Skipgram NEG Output