Recruitment Tasks Report and Findings

Task1: Word Similarity Scores

Objective:
To find the semantic similarity between word pairs that exists in SimLex-999 data

Task 1.a

Data Used:
With the given data constraint of not to use and pre-trained models, I opted to use *OSCAR 2019 en_part2.txt* data. The original file consists of more than 1M tokens. Hence, the training data used to run the experiment is the truncated version of OSCAR data which is done using the following linux command:

cat en_p2_2k19.txt | head -n 19000 > train_data.txt

The test data used is SimLex-999 data

Code:

```
!pip install nltk
!pip install pandas
import nltk
import re
import pandas as pd
import sklearn

from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

nltk.download('stopwords')
nltk.download('wordnet')

def data_preprocessing(tr_data):
  with open(tr_data, "r") as f:
    f_read = f.read()
  f_lower = f_read.lower()

  stop_words = set(stopwords.words('english'))
  lemmatizer = WordNetLemmatizer()
```

```
  w = f_lower.split()
  filtered_tokens = [word for word in w if word not in stop_words]
  lemmatized_tokens = [lemmatizer.lemmatize(t) for t in
filtered_tokens]
  lemmatized_text = ' '.join(lemmatized_tokens)

  clean_text = [re.sub("[^a-z]", " ", str(lemmatized_text))]

  return clean_text


def get_test_vec(word, v_dict):
  if word in v_dict:
    return v_dict[word]
  else:
    return None

doc1 = data_preprocessing('/content/train_a.txt')
doc2 = data_preprocessing('/content/train_b.txt')

train_docs = [doc1, doc2]
# print(train_docs)

flat_train_docs = []
for i in train_docs:
  flat_train_docs.append(i[0])
# print(flat_train_docs)

vec = TfidfVectorizer(lowercase=False)
transfrmd_vec = vec.fit_transform(flat_train_docs).toarray()
# print(transfrmd_vec)
print(transfrmd_vec.shape)

vocab = vec.get_feature_names_out()
# print(vocab)
# print(vec.vocabulary_)
vocab_dict = {v: transfrmd_vec[:, n] for n,v in enumerate(vocab)}
frst = next(iter(vocab_dict.items()))
# print(frst)


read_data = pd.read_csv('/content/test_data.txt',
sep='\t',encoding='UTF-8')
df = pd.DataFrame(read_data)
df_words = df[df.columns[0:2]]
```

```
semantic_sim = []

for i in range(len(df_words.index)):
  word1 = df_words.iloc[i, 0]
  vec1 = get_test_vec(word1, vocab_dict)
  word2 = df_words.iloc[i, 1]
  vec2 = get_test_vec(word2, vocab_dict)

  if vec1 is not None and vec2 is not None:
    sim = cosine_similarity([vec1], [vec2])[0][0]
    print(word1, word2, sim)
    semantic_sim.append(sim)
  else:
    print("oov")

ground_truth_values = df['SimLex999'].tolist()

simlex_threshold = 5
cosine_sim_threshold = 0.5

ground_truth_labels = [1 if val > simlex_threshold else 0 for val in
ground_truth_values]
cosine_sim_labels = [1 if val > cosine_sim_threshold else 0 for val in
semantic_sim]

preds = sum(1 for g, c in zip(ground_truth_labels, cosine_sim_labels)
if g == c)
acc = (preds / len(ground_truth_values))
print(acc)
```

Design Choices and Code WorkFlow:

1. While there are many ways(like one hot encoding, BOW etc) to convert words to numerics, I chose to go with the TF-IDF approach to convert text into vectors
2. The reason behind going with TF-IDF is because it takes the relevance of rare words into account by giving such words high scores which can be done by other methods like one hot encoding/ BOW
3. The function `data_preprocessing()` takes care of all the data cleaning and tokenizing before converting text into vectors
4. While `get_test_vec()` is used to get the vector value of corresponding text from the vocabulary dictionary generated with the help of TfidfVectorizer from scikit library
5. The train data has been split into two documents because, TF-IDF approach's usage is limited to TF if only we use it on single document
6. After that, the data is fit and transformed thru vectorizer and then a dictionary is created which has all words from the documents with its corresponding word vectors

7. Then cosine similarity of each word pair is calculated
8. After that to find the accuracy of predictions using this method, I have taken the threshold values for both simplex-999 data and calculated cosine similarity values and created a classification like logic to calculate accuracy (the idea of using threshold values has been taken from chatgpt)

Results and Observations:

1. Accuracy = 0.443
2. Using this approach is not so useful because the results are entirely dependent on the vocabulary of the document used and there are high chances of encountering out of vocabulary words in test data that are not present in train data at the time of generating vectors. It affects the accuracy of the predictions
3. A better improvement could be using a knowledge base like WordNet along with TF-IDF approach to handle out of vocabulary words and also help with the understanding of meaning of words and assigning vectors accordingly instead of depending on word order.

Task 1.b

Data Used:

With the constraint on data removed, I chose to go with Word2Vec pre-trained model and used Word2Vec embeddings pre-trained on Google News data.

Code:

```
!pip install gensim
!pip install pandas
!pip install nltk


import pandas as pd
import sklearn
import numpy as np

from gensim.downloader import load
from sklearn.metrics.pairwise import cosine_similarity


w2v_model = load('word2vec-google-news-300')

def get_word_vec(word):
  try:
    return w2v_model[word]
  except KeyError:
```

```
 return [0] * w2v_model.vector_size

read_data = pd.read_csv('/content/test_data.txt',
sep='\t',encoding='UTF-8')
df = pd.DataFrame(read_data)
df_words = df[df.columns[0:2]]

semantic_sim = []

for i in range(len(df_words.index)):
  word1 = df_words.iloc[i, 0]
  vec1 = get_word_vec(word1)

  word2 = df_words.iloc[i, 1]
  vec2 = get_word_vec(word2)

  if (np.all(vec1!= 0)) and (np.all(vec2 != 0)):
    sim = cosine_similarity([vec1], [vec2])[0][0]
    semantic_sim.append(sim)
    print(word1,word2,sim)
  else:
    print("oov")


ground_truth_values = df['SimLex999'].tolist()

simlex_threshold = 5
cosine_sim_threshold = 0.5

ground_truth_labels = [1 if val > simlex_threshold else 0 for val in
ground_truth_values]
cosine_sim_labels = [1 if val > cosine_sim_threshold else 0 for val in
semantic_sim]

preds = sum(1 for g, c in zip(ground_truth_labels, cosine_sim_labels)
if g == c)
acc = (preds / len(ground_truth_values))
print(acc)
```

Results and Observations:

1. Accuracy = 0.63
2. As we are using Word2Vec pre-trained embeddings, it captures the semantic relationship between word pairs much better than using the TF-IDF approach.

Task 2: Phrase and Sentence Similarity

Objective:
To find the semantic phrase similarity and semantic sentence similarity between pairs of phrases and pairs of sentences respectively

Task 2.a:

Code:

```
!pip install gensim
!pip install datasets


from huggingface_hub import login
login()

from datasets import load_dataset
from gensim.downloader import load
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

import random
import pandas as pd
import numpy as np
import re

ds = load_dataset("PiC/phrase_similarity")

word2vec_model = load('word2vec-google-news-300')


train_1 = ds['train']
dev_1 = ds['validation']
test_1 = ds['test']


train1_df = pd.DataFrame(train_1)
train_df = train1_df.drop(columns=['sentence1', 'sentence2', 'idx'])
# print(train_df)

dev1_df = pd.DataFrame(dev_1)
dev_df = dev1_df.drop(columns=['sentence1', 'sentence2', 'idx'])
# print(dev_df)
```

```
test1_df = pd.DataFrame(test_1)
test_df = test1_df.drop(columns=['sentence1', 'sentence2', 'idx'])
# print(test_df)


def mean_word_embeds(phrase, w2v_model):

  tokens = phrase.lower().split()

  word_embeds = []
  for word in tokens:
    if word in w2v_model:
      word_embeds.append(w2v_model[word])

  if not word_embeds:
    return np.zeros(w2v_model.vector_size)

  return np.mean(word_embeds, axis=0)



def cosine_sim(phrase1, phrase2, w2v_model):

  phrase_embed1 = mean_word_embeds(phrase1, w2v_model)
  phrase_embed2 = mean_word_embeds(phrase2, w2v_model)

  phrase_similarity = cosine_similarity([phrase_embed1],
[phrase_embed2]) [0][0]
  # print(phrase_similarity)

  return phrase_similarity


train_df['cal_sim'] = train_df.apply(lambda i:
cosine_sim(i['phrase1'], i['phrase2'], word2vec_model), axis=1)
train_sim = train_df['cal_sim'].values
train_labels = train_df['label'].values
train_sim_2d = np.array(train_sim).reshape(-1, 1)

dev_df['cal_sim'] = dev_df.apply(lambda j: cosine_sim(j['phrase1'],
j['phrase2'], word2vec_model), axis=1)
dev_sim = dev_df['cal_sim'].values
dev_labels = dev_df['label'].values
dev_sim_2d = np.array(dev_sim).reshape(-1, 1)
```

```
classifier = LogisticRegression()
classifier.fit(train_sim_2d, train_labels)

dev_pred = classifier.predict(dev_sim_2d)
dev_acc = accuracy_score(dev_labels, dev_pred)
print("dev acc: ", dev_acc)


test_df['cal_sim'] = test_df.apply(lambda j: cosine_sim(j['phrase1'],
j['phrase2'], word2vec_model), axis=1)
test_sim = test_df['cal_sim'].values
test_labels = test_df['label'].values
test_sim_2d = np.array(test_sim).reshape(-1, 1)

test_pred = classifier.predict(test_sim_2d)
test_acc = accuracy_score(test_labels, test_pred)
print("test_acc: ",test_acc)
```

Design Choices and Code WorkFlow:

1. To get phrase embeddings, I chose to use averaging of individual word vectors of the corresponding phrase and using these mean vectors of phrases to calculate similarity between them.
2. Word2Vec pre-trained model is being used to get the vectors of individual tokens and function `mean_word_embeds()` if the text is present in the word2vec model, if not that text will have zero vector as its numerical value. Finally the mean of these vectors is done to get the final phrase embeddings
3. The `cosine_sim()` function then calculates the similarity scores of the phrase pairs
4. To classify whether the pairs are similar or not, we logistic regression and training the model on train data to eventually predict the phrase similarity on test data

Results and Observations:

1. Dev Acc = 0.548, Test Acc = 0.5175
2. Though this method of averaging the embeddings works to a certain extent, it may not be the most efficient way to perform this task. While we can agree that it works faster, using RNN or LSTM could improve the performance of this task

Task 2.b:

Code:

```
from datasets import load_dataset
from gensim.downloader import load
from sklearn.metrics.pairwise import cosine_similarity
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

import random
import pandas as pd
import numpy as np
import spacy

ds = load_dataset("google-research-datasets/paws", "labeled_final")
word2vec_model = load('word2vec-google-news-300')

dp = spacy.load("en_core_web_sm")

random.seed(45)

train_1 = ds['train'].shuffle(seed=45).select(range(int(1.0 *
len(ds['train']))))
dev_1 = ds['validation'].shuffle(seed=45).select(range(int(1.0 *
len(ds['validation']))))
test_1 = ds['test'].shuffle(seed=45).select(range(int(1.0
*len(ds['test']))))

train_df = pd.DataFrame(train_1)
dev_df = pd.DataFrame(dev_1)
test_df = pd.DataFrame(test_1)


def dp_embed(sent, w2v_model, dp):

  dp_sent = dp(sent)
  dp_embeds = []

  for word in dp_sent:
    if word.dep_ in {"nsubj", "dobj", "ROOT"} and word.text in
w2v_model:
      dp_embeds.append(w2v_model[word.text])

  if not dp_embeds:
    return np.zeros(w2v_model.vector_size)

  return np.mean(dp_embeds, axis=0)


def cosine_sim(sent1, sent2, w2v_model, dp):
```

```
  sent_embed1 = dp_embed(sent1, w2v_model, dp)
  sent_embed2 = dp_embed(sent2, w2v_model, dp)

  sent_similarity = cosine_similarity([sent_embed1], [sent_embed2])
[0][0]

  return sent_similarity

train_df['cal_sim'] = train_df.apply(lambda i:
cosine_sim(i['sentence1'], i['sentence2'], word2vec_model, dp),
axis=1)
train_sim = train_df['cal_sim'].values
train_labels = train_df['label'].values
train_sim_2d = np.array(train_sim).reshape(-1, 1)

dev_df['cal_sim'] = dev_df.apply(lambda j: cosine_sim(j['sentence1'],
j['sentence2'], word2vec_model, dp), axis=1)
dev_sim = dev_df['cal_sim'].values
dev_labels = dev_df['label'].values
dev_sim_2d = np.array(dev_sim).reshape(-1, 1)


classifier = LogisticRegression()
classifier.fit(train_sim_2d, train_labels)

dev_pred = classifier.predict(dev_sim_2d)
dev_acc = accuracy_score(dev_labels, dev_pred)
print("dev acc: ", dev_acc)

test_df['cal_sim'] = test_df.apply(lambda j:
cosine_sim(j['sentence1'], j['sentence2'], word2vec_model, dp),
axis=1)
test_sim = test_df['cal_sim'].values
test_labels = test_df['label'].values
test_sim_2d = np.array(test_sim).reshape(-1, 1)

test_pred = classifier.predict(test_sim_2d)
test_acc = accuracy_score(test_labels, test_pred)
print("test_acc: ",test_acc)
```

Design Choices and Code WorkFlow:

1. Instead of averaging the vectors of individual tokens, I wanted to try and check if adding dependency parsing would give more information about meaning of sentences along

with using Word2Vec pre-trained model and then averaging the vectors. Function `dp_embed()` does the same
2. We are using spacy's dependency parser
3. The `cosine_sim()` function then calculates the similarity scores of the sentence pairs
4. To classify whether the pairs are similar or not, we logistic regression and training the model on train data to eventually predict the sentence similarity on test data

Results and Observations:

1. Dev Acc = 0.557, Test Acc = 0.558
2. When I used Task2.a code for sentence similarity task, it showed the same dev and test accuracies. Though the initial thought was that using a dependency parser might improve the performance, it did not show any expected improvement and the time taken for the overall execution of the code on the entire dataset has increased
3. Using Transformer based models like Bert and its classification head will definitely show much improvement because they use contextual embeddings unlike Word2Vec or GloVe whose embeddings are static

Reading Task: BERTScore - Evaluating Text Generation with BERT

Strengths:

1. Unlike other metrics like BLEU, METEOR that uses n-gram exact matching approaches, BERTScore computes similarity using contextual embeddings to effectively capture distant dependencies and ordering
2. It is reliable and robust than other metrics when it comes to adversarial paraphrase detection task
3. It is adaptable to various domains like SciBERT and various languages

Weaknesses:

1. There is no proper understanding of when to apply IDF to the data to improve the performance of metric
2. There is no one configuration of BERTScore that outperforms others. They alternate in different settings
3. It could be computationally intensive as it is a transformer based model

Improvements:

1. Understanding and experiment more on when to apply weighting techniques like IDF could help making the metric more reliable

References:

https://medium.com/@igniobydigitate/a-beginners-guide-to-measuring-sentence-similarity-f3c78b9da0bc

https://datascience.stackexchange.com/questions/107012/how-to-calculate-the-mean-average-of-word-embedding-and-then-compare-strings-usi

https://stackoverflow.com/questions/47727078/what-does-a-weighted-word-embedding-mean