WORD2VEC

A DETAILED REPORT

- Suhas K M

INTRODUCTION:

This report presents an in-depth analysis of the construction of Word2Vec models, highlighting the hurdles encountered during the preprocessing phases. It addresses the intricacies of managing text data for training models using GENSIM, PyTorch/TensorFlow, and leveraging Python APIs and methods with suitable data types to ensure the accuracy of Word2Vec models.

This precision is critical for capturing the full spectrum of semantic relationships within the text. Additionally, the work introduces the fundamentals of Continuous Bag of Words (CBOW) and Skip-Gram methods as strategies to grasp the context embedded in the text corpus. Moreover, the report delves into the computation of similarity indices and the visualization of frequently occurring words within the corpus, offering insights into the models' effectiveness in understanding and representing the semantic nuances of language.

Word2vec:

Word2Vec is a pivotal approach in the domain of natural language processing (NLP), designed to uncover semantic relationships within extensive text corpora. This method relies on analyzing texts collected from the internet, offering a comprehensive dataset for grasping language usage and meanings across various contexts. By leveraging such a vast

corpus, Word2Vec can accurately identify the intricate ways words interact and relate to each other in human language.

Notably, Word2Vec encompasses more than a single algorithm; it includes a range of model architectures and optimization techniques aimed at learning word embeddings. These embeddings are high-dimensional vectors that map words into a continuous vector space based on their semantic relationships, as deduced from their contextual appearances in the text. This approach ensures that words are represented in a manner that reflects their true meanings and the subtleties of language use.

Within the Word2Vec suite, models like Skip-gram and Continuous Bag of Words (CBOW) utilize distinct methods to learn these embeddings. Skip-gram predicts the context surrounding a target word, whereas CBOW focuses on predicting the target word from its surrounding context. Both are designed to deepen our understanding of word semantics, thereby enhancing machines' comprehension of natural language.

Word2Vec's effectiveness is underscored by its empirical success across numerous downstream NLP tasks, such as sentiment analysis, text classification, and more. This success is largely due to the embeddings' capacity to capture semantic nuances, allowing algorithms to process text with increased awareness and sophistication.

In essence, Word2Vec is a foundational technology in NLP, offering robust means for exploring the semantic depth of language. Through its innovative models and learning techniques, Word2Vec enables more nuanced text interaction, improving machines' performance on various language-related tasks.

DATASET:

The dataset in question is a comprehensive collection of Wikipedia articles, meticulously curated and made available through Hugging Face, a platform renowned for its contributions to machine learning and natural language processing. This dataset stands out for its broad coverage of Wikipedia content across all languages, making it an invaluable asset for various research and development endeavors.

Dataset Summary:

Built from the Wikipedia dump, which is publicly accessible at the Wikimedia dump site, the dataset hosted by Hugging Face encompasses cleaned articles from Wikipedia in all languages. The cleaning process entails removing markdown and extraneous sections like references, resulting in a dataset that predominantly consists of the articles' main content. This refined nature of the dataset renders it particularly beneficial for tasks necessitating clean, unformatted text.

Data Processing and Structure:

Utilizing the mwparserfromhell tool, a specialized utility for parsing Wikipedia's intricate markup, the articles within the dataset are processed. This tool, which can be installed via pip, aids in extracting clean text from Wikipedia's markup language. Organized with a separate split for each language, the dataset facilitates access to content tailored to specific linguistic preferences. It encompasses several fields in each entry:

id: The unique identifier for the article.

url: The URL of the original Wikipedia article.

title: The article's title.

text: The clean text content of the article.

This structured approach enhances the dataset's utility across a variety of uses, from language modeling and text analysis to machine translation and educational resources.

Accessing the Dataset:

The datasets library, part of Hugging Face's ecosystem designed for efficient dataset loading and processing, offers a straightforward interface for accessing Wikipedia dataset subsets.

The load_dataset function also accommodates parallel processing through the num_proc parameter, facilitating rapid data loading and processing.

Dataset Sizes and Examples:

With several pre-processed subsets, the dataset provides insights into its sizes:

The German subset (20220301.de) comprises 5.34 GB of downloaded files and an 8.91 GB generated dataset, totaling 14.25 GB of disk usage. The English subset (20220301.en) includes 11.69 GB of downloaded files and a 20.28 GB generated dataset, amounting to 31.96 GB in total disk usage.

The French subset (20220301.fr), significantly smaller, consists of 133.89 MB in downloaded files and a 235.07 MB generated dataset, leading to 368.96 MB of total disk usage.

These figures reflect the dataset's variable sizes across different languages, indicative of the vast and diverse content available in Wikipedia's multilingual corpus.

Why is Preprocessing so important:

Preprocessing plays a pivotal role in training machine learning models, including word embeddings like Word2Vec, primarily because it directly influences the quality of the model's output. By refining and cleaning the input data—such as removing noise, handling missing values, normalizing text (e.g., converting to lowercase, removing punctuation), and breaking texts into tokens—preprocessing ensures that the model learns from relevant, clean data.

This step is crucial for models like Word2Vec, which rely on the context of words to generate word embeddings. Without preprocessing, the model might treat different forms of the same word as separate entities, leading to poor representation of word meanings and relationships. Moreover, preprocessing can significantly reduce the dimensionality of the data, which not only improves model training time but also enhances its ability to capture the essence of the text data.

Essentially, effective preprocessing forms the foundation for training robust, accurate, and efficient Word2Vec models, enabling them to learn meaningful word associations and nuances in language.

Challenges & How to Overcome Them:

Understanding the correct format that GENSIM and other Python modules expect the data to be in is crucial. The best way to troubleshoot is to read through the respective APIs' documentation on the official website.

Since we were running on a virtual server (Discovery Cluster in our case), it was running on Python 2.7, but our Jupyter Notebook server was running on Python 3.6.8. So, running into a lot of dependency issues is common, but analyzing why took time. Hence, it is extremely important to keep an eye on what dependencies are allowed on a particular OS.

After a lot of training time, I came to realize how drastically faster CBOW is than the skip-gram approach. The skip-gram approach takes a more brute force approach (window checking). And since I was taking a comparative approach, it was really time-consuming to wait for the Skip Gram model to train while my CBOW model was ready.

It is hard to find the right balance to build a Word2Vec model; sometimes, the preprocessing process took up to 4-5 hours just to prepare the tokens. So, computational issues were definitely major. Overcoming or preventing this is by knowing the resources you have access to.

Word2vec Models:

1) **Skip-Gram Model:** Continuous skip-gram model, predicts words within a certain range before and after the current word in the same sentence. A worked example of this is given below.
2) **CBOW Model:** Continuous bag-of-words model, predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

Comprehensive Comparative Approach:

I sampled out “99999” rows from the Wikipedia dataset - English to train the CBOW model initially on which it performed as good as the Skip-Gram model when which was trained on almost the double the amount of data. And here are the comparative analysis and review of how the Skip-Gram approach performed as compared to CBOW approach.

★ Model similarity between sun and cloth:

CBOW Model:

```
# Words to compare
word1 = 'sun'
word2 = 'cloth'

# Check if both words are in the vocabulary of the model (assuming CBOW model here)
if word1 in cbow_model.wv.key_to_index and word2 in cbow_model.wv.key_to_index:
    similarity_skip_gram = cbow_model.wv.similarity(word1, word2)
    print(f'Skip-gram model similarity between {word1} and {word2}: {similarity_skip_gram}')
else:
    print(f'One or both words not in Skip-gram model vocabulary.')
```

```
Skip-gram model similarity between sun and cloth: 0.22371233999729156
```

Skip Gram Model:

```
# Words to compare
word1 = 'sun'
word2 = 'cloth'

# Check if both words are in the vocabulary of the model (assuming Skip-gram model here)
if word1 in model.wv.key_to_index and word2 in model.wv.key_to_index:
    similarity_skip_gram = model.wv.similarity(word1, word2)
    print(f'Skip-gram model similarity between {word1} and {word2}: {similarity_skip_gram}')
else:
    print(f'One or both words not in Skip-gram model vocabulary.')
```

```
Skip-gram model similarity between sun and cloth: 0.336364209651947
```

★ 10 words closest to 'example':

CBOW Model:

```
# Specify your input word
input_word = 'example'  # Replace 'example' with your actual input word

# Assuming 'model' is your Skip-gram trained model
# Find the 10 words closest to the input word
closest_words = cbow_model.wv.most_similar(input_word, topn=10)

print(f"10 words closest to '{input_word}':")
for word, similarity in closest_words:
    print(f"{word}: {similarity}")
```

```
10 words closest to 'example':
instance: 0.8416684865951538
examples: 0.7914485335350037
likewise: 0.7651951313018799
particular: 0.7328760027885437
contrast: 0.7312443852424622
clearly: 0.7232483625411987
similarly: 0.7210996150970459
necessarily: 0.7077968716621399
manner: 0.6973150372505188
rather: 0.695795476436615
```

Skip Gram Model:

```
# Specify your input word
input_word = 'example'

# Assuming 'model' is your Skip-gram trained model
# Find the 10 words closest to the input word
closest_words = model.wv.most_similar(input_word, topn=10)

print(f"10 words closest to '{input_word}':")
for word, similarity in closest_words:
    print(f"{word}: {similarity}")
```

```
10 words closest to 'example':
instance: 0.9114886522293091
examples: 0.8914328217506409
particular: 0.8338468074798584
contrast: 0.8154786229133606
consider: 0.8134335875511169
likewise: 0.8124775886535645
similarly: 0.8118988871574402
therefore: 0.8118278384208679
necessarily: 0.8104322552680969
possible: 0.8036367297172546
```

★ 10 words least similar to 'sun':

CBOW Model:

```
import numpy as np

# Specify your input word
input_word = 'sun'  # Replace 'example' with your actual input word

# Assuming 'model' is your Skip-gram trained model
# Retrieve all words from the model's vocabulary
all_words = list(cbow_model.wv.key_to_index.keys())

# Calculate similarity of input word with all other words in the vocabulary
similarities = [(word, cbow_model.wv.similarity(input_word, word)) for word in all_words]

# Sort the words by similarity in ascending order (least similar first)
least_similar_words = sorted(similarities, key=lambda x: x[1])

# Display the 10 least similar words to the input word
print(f"10 words least similar to '{input_word}':")
for word, similarity in least_similar_words[:10]:
    print(f"{word}: {similarity}")
```

```
10 words least similar to 'sun':
zistersdorf: -0.4427976906299591
arithmomaniac: -0.44182586669921875
mattani: -0.42715543508529663
schünemanns: -0.4065355360507965
hostagetakers: -0.40505439043045044
atiglio: -0.4049690365791321
ollerías: -0.4011055529117584
acalyphus: -0.4006273150444031
ghid: -0.39958053827285767
savene: -0.3926926553249359
```

Skip Gram Model:

```
import numpy as np

# Specify your input word
input_word = 'sun'  # Replace 'example' with your actual input word

# Assuming 'model' is your Skip-gram trained model
# Retrieve all words from the model's vocabulary
all_words = list(model.wv.key_to_index.keys())

# Calculate similarity of input word with all other words in the vocabulary
similarities = [(word, model.wv.similarity(input_word, word)) for word in all_words]

# Sort the words by similarity in ascending order (least similar first)
least_similar_words = sorted(similarities, key=lambda x: x[1])

# Display the 10 least similar words to the input word
print(f"10 words least similar to '{input_word}':")
for word, similarity in least_similar_words[:10]:
    print(f"{word}: {similarity}")
```

```
10 words least similar to 'sun':
id977: -0.3495708703994751
kargo: -0.3426513969898224
sw20: -0.3407485783100128
421778: -0.32830116152763367
manogi: -0.3118351399898529
460836: -0.30719104409217834
id919: -0.2978152334690094
04490: -0.29093706607818604
id971: -0.28943178057670593
846th: -0.2891865372657776
```

★ First 10 words in Skip-gram model's vocabulary:

CBOW Model:

```
# List first 10 words in Skip-gram model's vocabulary
skip_gram_vocab = list(cbow_model.wv.key_to_index.keys())
print("First 10 words in Skip-gram model's vocabulary:")
print(skip_gram_vocab[:10])
```

```
First 10 words in Skip-gram model's vocabulary:
['also', 'first', 'new', 'references', 'one', 'people', 'american', 'two', 'united', 'university']
```

Skip Gram Model:

```
# List first 10 words in Skip-gram model's vocabulary
skip_gram_vocab = list(model.wv.key_to_index.keys())
print("First 10 words in Skip-gram model's vocabulary:")
print(skip_gram_vocab[:10])
```

```
First 10 words in Skip-gram model's vocabulary:
['also', 'first', 'new', 'references', 'one', 'people', 'american', 'two', 'united', 'university']
```

★ Similarity between king and queen:

CBOW Model:

```
similarity_king_queen = cbow_model.wv.similarity('man', 'women')
print(f'Similarity between king and queen: {similarity_king_queen}')
```

```
Similarity between king and queen: 0.46683260798454285
```

Skip Gram Model:

```
similarity_king_queen = model.wv.similarity('man', 'women')
print(f'Similarity between king and queen: {similarity_king_queen}')
```

```
Similarity between king and queen: 0.5139315724372864
```

★ Model similarity Between Canada and Government:

CBOW Model:

```
cbow_model.wv.similarity('laptop'.lower(), 'pc'.lower())
```

```
0.65765923
```

Skip Gram Model:

```
model.wv.similarity('laptop'.lower(), 'pc'.lower())
```

```
0.52500683
```

★ Word Analogy:

CBOW Model:

```
from gensim.models import Word2Vec

# Define the analogy components
positive = ['king', 'woman']
negative = ['man']

# Find the words that complete the analogy
result = cbow_model.wv.most_similar(positive=positive, negative=negative, topn=1)

# Print the result
print(f"man is to king as woman is to {result[0][0]}")
```

```
man is to king as woman is to queen
```

This test proves that the model has been able to learn and performs fine on Analogy tests

Skip Gram Model:

```
# from gensim.models import Word2Vec

# Assuming your Gensim model is loaded in the variable `model`

# Define the analogy components
positive = ['king', 'woman']
negative = ['man']

# Find the words that complete the analogy
result = model.wv.most_similar(positive=positive, negative=negative, topn=1)

# Print the result
print(f"man is to king as woman is to {result[0][0]}")
```

```
man is to king as woman is to throne
```

CONCLUSION:

In my evaluation of Word2Vec model architectures, I observed that the Continuous Bag of Words (CBOW) model adeptly passed the analogy test, unlike the Skip-Gram approach, which struggled to meet the established criteria. Furthermore, in metrics such as the similarity index and cosine similarity, my CBOW models demonstrated a slight superiority over those based on the Skip-Gram methodology. This advantage underscores the nuanced effectiveness of the CBOW architecture in capturing semantic relationships within the textual data under these specific testing conditions.

Had there been additional time for model training, I believe I could have processed the entirety of the Wikipedia document, not just the English text. The preparation functions for text data processing were in place, suggesting that with extended time, I could have developed a much more accurate Word2Vec model. This realization emphasizes the importance of time as a critical resource in the model training process, affecting the scope of text data that can be processed and, consequently, the model's overall quality.

However, it's crucial to acknowledge the limitations I encountered with Skip-Gram models when dealing with large datasets. While Skip-Gram excels in detailed learning, making it suitable for smaller to medium datasets where capturing intricate semantic relationships is paramount, its efficacy diminishes with the scale of the dataset. Conversely, the methodological efficiency and faster processing times of CBOW models make them a more viable option for larger datasets. In such contexts, where computational resources and time are constrained, the ability of CBOW models to effectively manage and process large volumes of data highlights their suitability for extensive textual analyses, marking a distinct preference in my work for handling substantial text corpora.

CITATIONS:

[1] Hugging Face Datasets: Wikipedia. Available at:
https://huggingface.co/datasets/wikipedia

[2] GENSIM Documentation:
https://radimrehurek.com/gensim/auto_examples/index.html

[3]