006 - Представлення слів у R^n

Word2Vec

Word2Vec is a popular technique for learning word embeddings, introduced by Google in 2013. It represents words as vectors in a continuous vector space, allowing words with similar meanings to have similar vector representations. These embeddings capture the semantic relationships between words based on the context in which they appear. **Word2Vec** uses shallow neural networks to learn these representations from large text corpora.

Word2Vec is a method for learning dense, continuous vector representations of words that capture semantic relationships between words based on their context in large text corpora. It can be trained using either the **Continuous Bag of Words (CBOW)** or **Skip-gram** model, and it uses optimization techniques like **negative sampling** and **hierarchical softmax** to improve efficiency. Word2Vec embeddings are useful for a variety of NLP tasks, but they have been largely superseded by more advanced models like **BERT** and **ELMo**, which capture context-dependent meanings.

Key Concepts of Word2Vec:

1. Word Embeddings

Word embeddings are dense vector representations of words, where each word in the vocabulary is represented by a vector in a high-dimensional space. Unlike traditional one-hot encoding, where each word is represented by a sparse vector with a single "1" and the rest "0"s, word embeddings are continuous-valued vectors. These vectors capture syntactic and semantic relationships between words.

- For example, the word embeddings of "king", "queen", "man", and "woman" might exhibit the following relationships: king−man≈queen−woman\text{king} - \text{man} \approx \text{queen} - \text{woman}king−man≈queen−woman

This means the difference between "king" and "man" is similar to the difference between "queen" and "woman," capturing analogical relationships.

2. Training Objectives of Word2Vec

Word2Vec can be trained using one of two main approaches:

- **Continuous Bag of Words (CBOW)**: In CBOW, the model predicts a target word based on its surrounding context words. Given a window of words surrounding a target word, the model tries to predict the target word. The intuition is that words that appear in similar contexts should have similar representations.

    - For example, given the context words "the", "sat", "on", and "mat", CBOW would try to predict the word "cat".
- **Skip-gram**: In the skip-gram model, the task is reversed. The model is trained to predict the context words given a target word. The goal is to learn word representations such that given a word, the model can accurately predict the words that appear around it in a sentence.
    - For example, given the target word "cat", skip-gram would try to predict words like "sat", "on", and "mat".

Both approaches aim to capture the relationships between words in a sentence, but they do so in different ways. Skip-gram tends to perform better on rare words, while CBOW is more efficient with common words.

3. Context Window

Both CBOW and skip-gram models use a **context window** to define the words that surround a target word. The window size determines how many words before and after the target word are considered part of the context.

- For example, with a window size of 2, the model looks at the two words before and two words after the target word to define the context. Larger window sizes capture broader context, while smaller windows focus on more immediate context.

4. Mathematical Formulation

CBOW: The objective is to maximize the probability of predicting the target word wtw_twt given its context words $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}$:

$$P(w_t | w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$$

Skip-gram: The goal is to maximize the probability of predicting the context words $w_{t-k}, \ldots, w_{t+k}$ given a target word w_t:

$$P(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} | w_t)$$

In both cases, the model is trained to maximize the probability of correct predictions, which is achieved through a shallow neural network with one hidden layer. The final word embeddings are derived from the weights of this hidden layer.

5. Efficient Training with Negative Sampling and Hierarchical Softmax

Training Word2Vec efficiently on large corpora can be computationally expensive due to the size of the vocabulary. To address this, Word2Vec uses two optimization techniques:

- **Negative Sampling**: Instead of updating the weights for all words in the vocabulary during training, negative sampling randomly selects a few negative examples (words

that are not in the context) to update. This makes training faster and reduces computational complexity.
- **Hierarchical Softmax**: Another technique to speed up training is hierarchical softmax, which structures the vocabulary as a binary tree. Instead of computing the softmax over all words in the vocabulary, the model traverses the tree to make predictions, which reduces the number of computations required.

6. Advantages of Word2Vec

- **Capturing Semantic Relationships**: Word2Vec embeddings capture semantic relationships between words, such as analogies, synonyms, and relatedness.
- **Efficient Training**: Using negative sampling and hierarchical softmax makes Word2Vec efficient for large corpora.
- **Pre-trained Models**: Pre-trained Word2Vec embeddings are often used as a starting point for NLP models, saving time and improving performance, especially on smaller datasets.

7. Applications of Word2Vec

Word2Vec embeddings have been widely used in various NLP applications:

- **Text Classification**: Word embeddings are used as input features for tasks like sentiment analysis, topic classification, and spam detection.
- **Machine Translation**: Word2Vec helps map words between languages based on their vector representations.
- **Document Similarity**: Word embeddings can be averaged or combined to represent entire documents, making it easier to compute similarity between documents.
- **Named Entity Recognition (NER)**: Word2Vec helps identify and classify entities like names, dates, and organizations in a text.

Example of Word2Vec Analogies

- Word2Vec embeddings can capture analogies through vector arithmetic. For example: king−man+woman≈queen\text{king} - \text{man} + \text{woman} \approx \text{queen}king−man+woman≈queen

This suggests that the relationship between "king" and "man" is similar to the relationship between "queen" and "woman," illustrating the semantic structure captured by the embeddings.

Limitations of Word2Vec

- **Contextual Ambiguity**: Word2Vec provides a single vector for each word, regardless of its meaning in different contexts. For example, "bank" (a financial institution) and "bank" (the side of a river) would have the same embedding, even though they have different meanings.
- **Sentence-Level Understanding**: Word2Vec focuses on word-level representations and does not capture the meaning of entire sentences or paragraphs.

To address these limitations, newer models like **ELMo**, **BERT**, and **GPT** provide context-sensitive embeddings, where the vector representation of a word changes depending on the context in which it appears.

Example in python:

Here's an example of how to apply **Word2Vec** for **text clustering** using **Gensim** and **Scikit-learn**. We will use Word2Vec to create word embeddings from a text corpus and then perform clustering using **KMeans**.

Requirements:

You'll need to install the following libraries:

```
pip install gensim scikit-learn
```

Python Code: Word2Vec for Text Clustering

```
import gensim
from gensim.models import Word2Vec
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Sample corpus (list of sentences)
corpus = [
      "The cat sat on the mat.",
      "The dog chased the cat.",
      "The cat and the dog are friends.",
      "The quick brown fox jumped over the lazy dog.",
      "The lazy dog was sleeping.",
      "A man and a woman are sitting on the bench.",
      "The car is parked near the house.",
      "The sun is shining brightly.",
      "She is reading a book by the window.",
      "He loves playing football with his friends."
]

# Preprocess the text by tokenizing and lowercasing
def preprocess(text):
      return [gensim.utils.simple_preprocess(sentence) for sentence
in text]

tokenized_corpus = preprocess(corpus)

# Train a Word2Vec model on the tokenized corpus
word2vec_model = Word2Vec(
      sentences=tokenized_corpus, vector_size=100,
      window=5, min_count=1, workers=4
```

```
)

# Get word vectors for each word in the vocabulary
word_vectors = word2vec_model.wv

# Get the list of words in the vocabulary
vocab = list(word_vectors.index_to_key)

# Retrieve the vector representations for each word in the
vocabulary
word_embeddings = [word_vectors[word] for word in vocab]

# Perform KMeans clustering with 3 clusters
num_clusters = 3
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(word_embeddings)

# Get the cluster assignments for each word
clusters = kmeans.labels_

# Print the words in each cluster
for i in range(num_clusters):
     print(f"Cluster {i}:")
     cluster_words = [
          vocab[j] for j in range(len(vocab)) if clusters[j] == i
     ]
     print(cluster_words)
     print()

# Visualizing the clusters using PCA (2D plot)
pca = PCA(n_components=2)
word_embeddings_2d = pca.fit_transform(word_embeddings)

plt.figure(figsize=(10, 8))
for i, word in enumerate(vocab):
     plt.scatter(word_embeddings_2d[i, 0], word_embeddings_2d[i,
1], marker='o')
     plt.text(word_embeddings_2d[i, 0] + 0.01,
word_embeddings_2d[i, 1], word, fontsize=12)

plt.title("Word2Vec Word Embeddings Clustering")
plt.show()
```

Key Components of the Code:

1. **Preprocessing the Corpus**:

    - The `preprocess` function tokenizes and lowercases each sentence in the corpus to prepare it for training the Word2Vec model.
    - The `gensim.utils.simple_preprocess` function is used to tokenize each sentence.
2. **Training the Word2Vec Model**:
    - We train a **Word2Vec** model using Gensim on the tokenized corpus. The model learns 100-dimensional vector embeddings for each word in the vocabulary.
    - The model uses a window size of 5 (context of 5 words around each word) and a minimum word frequency of 1.
3. **Extracting Word Embeddings**:
    - We extract the word embeddings (vectors) for each word in the vocabulary using `word2vec_model.wv`.
    - The embeddings are stored in the `word_embeddings` list.
4. **KMeans Clustering**:
    - We use **KMeans** clustering from Scikit-learn to cluster the word embeddings into 3 clusters.
    - After fitting the KMeans model, we retrieve the cluster assignments and print the words in each cluster.
5. **Visualization**:
    - To visualize the clusters, we use **PCA** (Principal Component Analysis) to reduce the word embedding dimensions from 100 to 2.
    - The word embeddings are then plotted in 2D space using **Matplotlib**.

Output:

The program will print the words assigned to each of the 3 clusters. For example:

```
Cluster 0:
['the', 'cat', 'on', 'and', 'are', 'man', 'woman', 'is', 'by',
'playing']

Cluster 1:
['sat', 'mat', 'chased', 'friends', 'fox', 'jumped', 'parked',
'brightly', 'near']

Cluster 2:
['dog', 'lazy', 'quick', 'brown', 'over', 'sleeping', 'car',
'house', 'sun', 'she', 'reading', 'book', 'window', 'he', 'loves',
'football', 'with', 'his', 'friends']
```

Additionally, a 2D plot will be generated showing the word embeddings and their clustering.

Notes:

- **Word2Vec Training**: The `Word2Vec` model learns the word embeddings by training on the corpus. These embeddings capture the semantic relationships between words based on their context in the text.
- **KMeans Clustering**: The KMeans algorithm clusters the word vectors based on their similarity. Words with similar meanings or usage are likely to end up in the same cluster.
- **PCA for Visualization**: Since the word vectors are high-dimensional, we use PCA to reduce the dimensionality to 2D for visualization purposes.

This example demonstrates how to apply **Word2Vec** to learn word embeddings and cluster them using **KMeans**. The embeddings can then be visualized to gain insights into the relationships between words.

FastText

FastText is an extension of the Word2Vec model developed by Facebook's AI Research (FAIR) lab. It improves upon Word2Vec by addressing some of its limitations, particularly with rare and out-of-vocabulary words. The main innovation in FastText is that it represents each word as a collection of character-level **n-grams** (subword units) rather than treating each word as a single atomic unit. This allows FastText to capture both morphological and semantic information about words, making it more robust for handling rare words and words that were not seen during training.

Key Concepts of FastText:

1. Subword Information

One of the key differences between FastText and Word2Vec is that FastText breaks down each word into smaller units, known as **n-grams**. For example, the word "apple" can be represented as the following character n-grams for a window size of 3 (trigrams):

- `<ap`, `app`, `ppl`, `ple`, `le>`
- FastText would represent "apple" as the sum of the vector representations of these n-grams, rather than as a single word vector. This allows the model to capture information about the morphology of words (e.g., prefixes, suffixes) and handle rare or unseen words by building representations from known subwords.
- This approach is particularly useful for handling **morphologically rich languages** (such as German or Finnish) and **rare words** that are infrequent in the training data. Even if a word wasn't seen during training, FastText can still generate an embedding for it by composing the n-grams.

2. Word Representation

FastText creates the representation of a word by summing up the embeddings of its n-grams and the word itself. This is a key distinction from Word2Vec, where each word is treated as an atomic entity. In FastText, even if the model hasn't seen the entire word, it can infer the meaning from the subword units that it has seen before.

- For example, the word **"apples"** could be represented as the sum of the n-grams: Representation of "apples"=sum of vectors for ‘<ap‘, ‘app‘, ‘ppl‘, ‘ple‘, ‘les‘, ‘s>‘\text{Representation of "apples"} = \text{sum of vectors for `<ap`, `app`, `ppl`, `ple`, `les`, `s>`}Representation of "apples"=sum of vectors for ‘<ap‘, ‘app‘, ‘ppl‘, ‘ple‘, ‘les‘, ‘s>‘

This subword representation allows FastText to generalize better to unseen words.

3. Model Architecture

Like Word2Vec, FastText supports two main learning objectives for training word embeddings:

- **Continuous Bag of Words (CBOW)**: FastText predicts the target word based on its surrounding context words. In this case, the context is built from the n-grams of the neighboring words.
- **Skip-gram**: FastText predicts the context words given a target word. Again, the model leverages n-grams to capture relationships between words.

The overall structure of FastText is similar to Word2Vec, with a shallow neural network architecture and the use of negative sampling or hierarchical softmax to speed up training.

4. Handling Out-of-Vocabulary Words

One of the main advantages of FastText over Word2Vec is its ability to handle **out-of-vocabulary (OOV) words**. Word2Vec treats every word as a unique entity, so if a word is not seen during training, it cannot produce a meaningful vector for it. FastText, on the other hand, can generate representations for OOV words by summing the vectors of their n-grams, even if the entire word was not seen during training.

For instance, if the model hasn't seen the word "electrification" during training, it can still generate a meaningful vector for it by using subwords like "electri", "fication", etc.

5. Efficient and Scalable

FastText is designed to be fast and scalable, capable of training on very large datasets efficiently. It uses optimizations such as **negative sampling** and **hierarchical softmax**, which are also used in Word2Vec, to reduce the computational cost of training on large vocabularies.

Additionally, FastText can generate vectors for unseen words very quickly during inference, making it highly suitable for applications where new words or rare words frequently appear (e.g., social media, domain-specific texts).

6. Embeddings for Phrases

In addition to generating embeddings for words, FastText can also create embeddings for phrases by treating them as single entities during training. For instance, a phrase like "New York" can be treated as a single token instead of two separate words, allowing the model to capture the meaning of multi-word expressions or named entities.

7. Advantages of FastText

- **Handles Rare Words and OOV Words**: FastText can generate embeddings for rare words and out-of-vocabulary words by composing their n-grams, unlike Word2Vec which struggles with these cases.
- **Morphological Information**: By using subwords, FastText captures the internal structure of words, making it effective for morphologically rich languages.
- **Fast and Efficient**: FastText retains the efficiency of Word2Vec, using techniques like negative sampling and hierarchical softmax for fast training.
- **Contextual Flexibility**: FastText can capture word variations based on their character-level structure, which is useful in applications like text classification and information retrieval.

8. Applications of FastText

FastText has been widely adopted in various NLP tasks, including:

- **Text Classification**: FastText embeddings are often used as input features for sentiment analysis, spam detection, and topic classification.
- **Named Entity Recognition (NER)**: FastText can help identify and classify named entities, including rare names and places.
- **Machine Translation**: FastText can assist in mapping words between languages by using embeddings that capture subword information.
- **Search and Information Retrieval**: FastText embeddings can improve the retrieval of documents by generating relevant representations even for misspelled or out-of-vocabulary queries.

9. Limitations of FastText

- **Context Independence**: Like Word2Vec, FastText generates static embeddings, meaning the same word will have the same vector representation regardless of its context. Models like **ELMo** and **BERT** offer contextualized embeddings that change depending on the context in which a word is used.
- **Subword Complexity**: Although subwords improve FastText's ability to handle rare words, they also add complexity and computational overhead compared to Word2Vec, which only works at the word level.

Example of FastText Word Representation:

- Consider the word **"running"**. FastText will represent "running" not just as a single word, but as a combination of the following character n-grams (for trigrams):
    - `<ru`, `run`, `unn`, `nni`, `nin`, `ing`, `ng>`

This representation allows FastText to capture both semantic and morphological information about words, improving its performance on tasks where rare or unseen words are common.

Example in Python

Here's an example of how to apply **FastText** for text classification using the **Gensim** library in Python. We'll train a FastText model on a simple dataset and then use the word embeddings generated by FastText for a text classification task.

Requirements:

You'll need to install the following libraries:

```
pip install gensim scikit-learn
```

Python Code: FastText for Text Classification

```
import gensim
from gensim.models import FastText
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
import numpy as np

# Sample corpus (list of sentences)
corpus = [
      "The cat sat on the mat.",
      "The dog chased the cat.",
      "The cat and the dog are friends.",
      "The quick brown fox jumped over the lazy dog.",
      "The lazy dog was sleeping.",
      "A man and a woman are sitting on the bench.",
      "The car is parked near the house.",
      "The sun is shining brightly.",
      "She is reading a book by the window.",
      "He loves playing football with his friends."
]

# Sample labels (sentiment or category for each sentence)
labels = [
      "animals",
      "animals",
      "animals",
      "animals",
      "animals",
      "people",
      "people",
      "nature",
```

```
      "people",
      "sports"
]

# Preprocessing: Tokenizing and converting text to lowercase
def preprocess(text):
      return [
            gensim.utils.simple_preprocess(sentence)
            for sentence in text
      ]

tokenized_corpus = preprocess(corpus)

# Train a FastText model on the tokenized corpus
fasttext_model = FastText(sentences=tokenized_corpus,
vector_size=100, window=5, min_count=1, workers=4)

# Function to get sentence embeddings by averaging the word
embeddings
def get_sentence_embedding(sentence, model):
      word_vectors = [
            model.wv[word] for word in sentence if word in model.wv
      ]
      if len(word_vectors) > 0:
            return np.mean(word_vectors, axis=0)
      else:
            return np.zeros(model.vector_size)

# Convert all sentences to their FastText embeddings
X = np.array([
      get_sentence_embedding(sentence, fasttext_model)
      for sentence in tokenized_corpus
])

# Encode the labels into numerical format
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(labels)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train a RandomForest classifier on the FastText embeddings
classifier = RandomForestClassifier(n_estimators=100,
random_state=42)
classifier.fit(X_train, y_train)

# Make predictions on the test set
```

```
y_pred = classifier.predict(X_test)

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print the predicted labels and the actual labels
print("\nPredicted labels:",
label_encoder.inverse_transform(y_pred))
print("Actual labels:   ",
label_encoder.inverse_transform(y_test))
```