lab4_word_embeddings_part2

Harito ID

2025-10-14

Lab 4: Word Embeddings with Word2Vec

Setup

Before starting this lab, ensure you have installed all necessary project dependencies by running:

```
pip install -r requirements.txt
```

This lab uses the `gensim` library for word embeddings. The `WordEmbedder` class will automatically download the `glove-wiki-gigaword-50` model the first time it is used. This download is approximately 65MB and may take some time depending on your internet connection.

Objective

To move from sparse, high-dimensional representations (like TF-IDF) to dense, low-dimensional semantic representations. You will learn to load and use pre-trained word embeddings to find semantic similarities between words and to create document embeddings.

Theory

Word embeddings are dense vectors that represent words in a way that captures their meaning, context, and semantic relationships. Unlike TF-IDF, where similar words are treated as completely different features, word embeddings place similar words close to each other in the vector space.

Word2Vec is a popular model used to learn these embeddings from large text corpora. It works on the principle that "a word is characterized by the company it keeps."

For this lab, we will use a pre-trained model, which is a common and effective approach.

Task 1: Setup

1. **Install `gensim`:** This library provides easy access to word embedding models. Add `gensim` to your `requirements.txt` file and install it.
2. **Download Pre-trained Model:** We will use a pre-trained model from `gensim`'s data repository. The model `glove-wiki-gigaword-50` is a good starting point (50-dimensional vectors trained on Wikipedia).

Task 2: Word Embedding Exploration

1. **Create the file:** `src/representations/word_embedder.py`.
2. **Implement the `WordEmbedder` class:**
    - The constructor `__init__(self, model_name: str)` should accept a model name (e.g., `'glove-wiki-gigaword-50'`).
    - Inside the constructor, use `gensim.downloader.load(model_name)` to load the model and store it in an attribute.
3. **Implement exploration methods:**
    - `get_vector(self, word: str)`: Returns the embedding vector for a given word. Handle cases where the word is not in the vocabulary (Out-of-Vocabulary or OOV words).
    - `get_similarity(self, word1: str, word2: str)`: Returns the cosine similarity between the vectors of two words.
    - `get_most_similar(self, word: str, top_n: int = 10)`: Uses the model's built-in `most_similar` method to find the top N most similar words.

Task 3: Document Embedding

Representing a whole document can be done in many ways. A simple but effective baseline is to average the word vectors of all the words in the document.

1. **Implement `embed_document(self, document: str)`:**
    - This method should take a string document as input.
    - Use a `Tokenizer` (from Lab 1) to split the document into tokens.
    - For each token, get its vector. Ignore OOV words.
    - If the document contains no known words, return a zero vector of the correct dimension.
    - Otherwise, compute the element-wise mean of all the word vectors to get a single document vector.

Evaluation

- Create a new test file: `test/lab4_test.py`.
- Instantiate your `WordEmbedder`.
- Perform and print the results of the following operations:
    - Get the vector for the word 'king'.
    - Get the similarity between 'king' and 'queen', and between 'king' and 'man'.
    - Get the 10 most similar words to 'computer'.
    - Embed the sentence "The queen rules the country." and print the resulting document vector.

Bonus Task: Training a Word2Vec Model from Scratch

While using pre-trained models is very effective, it's also insightful to train your own embeddings on a specific domain. The script `test/lab4_embedding_training_demo.py` was created to demonstrate this process.

This script performs the following steps:

1. **Streams Data**: It reads the raw text from `data/UD_English-EWT/en_ewt-ud-train.txt` in a memory-efficient way.
2. **Trains a Model**: It uses `gensim` to train a new `Word2Vec` model on this data.
3. **Saves the Model**: The resulting trained model is saved to `results/word2vec_ewt.model`.
4. **Demonstrates Usage**: It shows how to use the newly trained model to find similar words and solve analogies, providing a complete end-to-end example.

To run this demonstration, execute the following command from the project root:

```
python test/lab4_embedding_training_demo.py
```

Advanced Task: Scaling Word2Vec with Apache Spark

While `gensim` is excellent for training on datasets that fit into a single machine's memory, real-world scenarios often involve massive text corpora (terabytes of data). In such cases, `gensim` becomes impractical.

This is where **Apache Spark** comes in. Spark is a distributed computing framework that can parallelize tasks across a cluster of machines. Its machine learning library, `Spark MLlib`, provides a scalable implementation of Word2Vec.

Why use Spark for Word2Vec?

- **Scalability:** Train on datasets far larger than a single machine's RAM.
- **Speed:** Distributed training significantly speeds up the process on large datasets.
- **Ecosystem:** Integrates seamlessly with other big data tools and Spark-based data processing pipelines.

Implementation with PySpark

Below is an example of how to train a Word2Vec model using PySpark on a larger dataset.

1. **Setup Spark:** First, you need to install `pyspark`. Add `pyspark` to your `requirements.txt` and install it.

   ```
   pip install pyspark
   ```

2. **Training Script:** The following script demonstrates how to set up a Spark session, load data, and train a Word2Vec model. We will use the `c4-train.00000-of-01024-30K.jso` dataset, which contains text data.

   Create a file `test/lab4_spark_word2vec_demo.py`:

   ```python
   import re
   from pyspark.sql import SparkSession
   from pyspark.ml.feature import Tokenizer, Word2Vec
   from pyspark.sql.functions import col, lower, regexp_replace, split

   def main():
       # Initialize Spark Session
       # ...

       # Load the dataset
       # The C4 dataset is in JSON format, with each line being a JSON object.
   ```

```
    # We are interested in the 'text' field.
    # ...

    # Preprocessing
    # 1. Select the text column and convert to lowercase
    # 2. Remove punctuation and special characters
    # 3. Split the text into an array of words
    # ...

    # Configure and train the Word2Vec model
    # ...

    # Demonstrate the model
    # Find synonyms for a word
    # ...

    # Stop the Spark session
    spark.stop()

if __name__ == "__main__":
    main()
```

3. **Run the Spark Job:** Execute the script from your terminal. This will start a local Spark job.

   ```
   python test/lab4_spark_word2vec_demo.py
   ```

   This script will:

   - Read the JSON data.
   - Perform basic text cleaning and tokenization using Spark's built-in functions.
   - Train a 100-dimensional Word2Vec model.
   - Find and display the top 5 words most similar to "computer".

This advanced task shows how you can leverage distributed computing to handle NLP tasks at scale, moving beyond the limitations of single-machine libraries.