Creating code embedding matrix

Midterm project for PHS7045

Yidan Zhang

Introduction

Word embedding has become a significant topic in natural language processing, especially when working with context data. The key concept behind word embedding is based on the idea that a word can be characterized by "the company it keeps," or its context. Word embeddings capture this contextual information by creating a lower-dimensional space (compared to traditional one-hot encoding) where words that appear in similar contexts are positioned close to one another. This allows for more efficient and effective word representation in models.

One of the most widely recognized methods for generating word embeddings is Word2Vec, developed by Google in 2013. This neural network-based approach is trained on large corpora of text, learning to predict the context of a word based on its surrounding words within a local window. While Word2Vec focuses on local context, another prominent method, GloVe (Global Vectors for Word Representation), expands this concept by incorporating global context. GloVe takes advantage of the overall word-to-word co-occurrence patterns across the entire corpus. By using a precomputed, count-based co-occurrence matrix, GloVe leverages global statistical information about how often words appear together to generate its embedding matrix.

With the increasing digitization of patient information, electronic health records (EHR) have become essential for efficiently tracking medical histories and facilitating the exchange of data between healthcare providers. As structured data within EHR systems, ICD (International Classification of Diseases) and CPT (Current Procedural Terminology) codes have long been used to record diagnoses and procedures, respectively, even before the widespread adoption of EHRs. Embedding these medical codes offers several key benefits. It can enhance disease prediction and diagnosis modeling by allowing models to recognize similarities between related conditions, improving predictive performance. Additionally, ICD code embeddings support patient clustering, enabling algorithms like k-means or DBSCAN to group patients based on diagnostic patterns and improve clustering effectiveness. They also help identify complex comorbidity patterns and hidden phenotypes that traditional methods may miss. Finally, code

embedding aids in healthcare data integration, which is especially valuable given the challenges of accessing protected data due to data safety concerns.

The goal of this project is to implement the GloVe algorithm to create a package that generates word embedding matrices from EHR data. The package will be tested on the MIMIC-III clinical dataset to generate ICD-9 code embeddings.

Solution Plan

- STEP 1:

  The first step in generating embeddings using the GloVe approach is to create a code co-occurrence matrix, $\mathbf{X}$, where each element of $\mathbf{X}_{\mathbf{ij}}$ represents how often word i appears in context of word j. The function's arguments will include the user's dataset (`data`), along with parameters to specify which column corresponds to the subject in dataset (`id`), which column represents the time or sequence information for defining the context window (`time`), and the user-defined size of the context window (`window`). EHR data typically consists of large datasets with vast amounts of patient information, and each patient have multiple diagnosis codes. To compute the co-occurrences, the function must iterate through each patient and count the co-occurrence of ICD codes within a user-defined context window,with approximately 13,000 ICD-9 codes, this process can be both time-consuming and computationally expensive. To address these challenges, the project will optimize performance by coercing the data to a `data.table` structure for faster reference handling, applying vectorization for efficient matrix generation, and incorporating parallel computing to further improve time efficiency.

- STEP 2:

  The second step is to use the co-occurrence matrix to create the pointwise mutual information (PMI) matrix, which measures the association between a word and a context word. PMI is calculated as $\text{PMI}(w, c) = \frac{p(w,c)}{p(w) \cdot p(c)}$, where $p(w, c)$ is the count of how often word w and context code c occur in the same context window, divided by the total count of code pairs within the window; and $p(w)$ and $p(c)$ are singleton frequency of the word w and c, respectively, within the context window. In machine learning literature, the shifted positive pointwise mutual information (SPPMI) matrix is commonly used, where PMI is shifted by a user-defined constant $log(k)$. To accommodate user-specific needs, another function in the package will be aim to calculate SPPMI, which is defined as $\text{SPPMI}(w, c) = max(\text{PMI}(w, c) - log(k), 0)$. The computation starts by obtaining the joint and singleton frequencies for the codes using the function `get_SG`, where vectorization is applied. The two functions will then be used to compute both PMI and SPPMI, respectively.

- STEP 3:

  In this step, truncated singular value decomposition (SVD) is applied to factorize the PMI/SPPMI matrix and generate lower-dimensional embeddings for the codes. The function allows the user to specify the number of top p-dimensional singular values and vectors to retain, as well as the maximum number of iterations for the SVD process. Once executed, the function will return the code embedding matrix $\mathbf{W} = u \cdot \sqrt{d}$, where the matrix has a dimensionality reduced to the user-defined vector length p. To optimize the code's performance, vectorization techniques and the use of `data.table` will be incorporated to accelerate the process and ensure efficient computation.

- STEP 4 (Time Permitting):

  One approach to help users determine the optimal embedding vector length is to evaluate the performance of the embedding matrix by predicting phecode-based classification accuracy. Phecodes group ICD codes into clinically meaningful categories, and an ideal embedding matrix would accurately capture diagnostic information, allowing for precise classification aligned with phecodes. By assessing the AUC (Area Under the Curve) for different vector lengths, users can identify the length that maximizes performance, minimizing information loss and avoiding overfitting.

Preliminary Result

- Data

  The data used to test the package is sourced from the MIMIC-III clinical dataset, a freely available database containing de-identified health-related data from patients who were admitted to the ICU at Beth Israel Deaconess Medical Center between 2001 and 2012. The dataset includes diagnoses recorded using the 9th version of the ICD coding system, specifically in a table named "DIAGNOSES_ICD." A brief summary of this data is provided below:

Statistic	Value
Number of Patients	46520
Number of ICD9 Codes	6985
Diagnosis/Patient-min	1
Diagnosis/Patient-median	9
Diagnosis/Patient-max	540

Figure 1: Summary of the table

- Function

- The execution of the functions

```
create_cooccurrence_matrix <- function(data,id, code, time, window = NA) {
  # Get unique ICD codes
  unique_codes <- unique(data[[code]])
  code_count <- length(unique_codes)

  # Initialize an empty co-occurrence matrix
  cooccurrence_matrix <- matrix(0, nrow = code_count, ncol = code_count)
  rownames(cooccurrence_matrix) <- unique_codes
  colnames(cooccurrence_matrix) <- unique_codes

  # Iterate through each patient
  patients <- unique(data[[id]])
  for (patient in patients) {
    # Get data for the specific patient and sort by time column (SEQ_NUM)
    patient_data <- data %>% filter(data[[id]] == patient) %>% arrange(data[[time]])

    # If window is NA, set window to be the number of rows for this patient
    if (is.na(window)) {
      patient_window <- nrow(patient_data)
    } else {
      patient_window <- window
    }

    # For each ICD code in patient data, compare with others within the window
    for (i in 1:(nrow(patient_data) - 1)) {
      for (j in (i + 1):nrow(patient_data)) {
        if (patient_data[[time]][j] - patient_data[[time]][i] > patient_window) break

        code_i <- patient_data[[code]][i]
        code_j <- patient_data[[code]][j]

        # Update the co-occurrence matrix
        cooccurrence_matrix[code_i, code_j] <- cooccurrence_matrix[code_i, code_j] + 1
        cooccurrence_matrix[code_j, code_i] <- cooccurrence_matrix[code_j, code_i] + 1
      }
    }
  }

  return(cooccurrence_matrix)
}
```

```r
# Example usage
# Assuming 'data' is your dataset containing custom column names

# Static window example
#cooccurrence_matrix_static <- create_cooccurrence_matrix(data, code = "ICD9_CODE", id = "SU

# Dynamic window example
#cooccurrence_matrix_dynamic <- create_cooccurrence_matrix(data, icd_col = "ICD9_CODE", id_c
```

```r
#   code vectorization
create_cooccurrence_matrix_dt <- function(data, window = NA) {
  # Ensure the input data is a data.table
  setDT(data)

  # Get unique ICD codes
  unique_codes <- unique(data$ICD9_CODE)
  code_count <- length(unique_codes)

  # Initialize an empty co-occurrence matrix
  cooccurrence_matrix <- matrix(0, nrow = code_count, ncol = code_count)
  rownames(cooccurrence_matrix) <- unique_codes
  colnames(cooccurrence_matrix) <- unique_codes

  # Iterate through each patient
  patients <- unique(data$SUBJECT_ID)
  for (patient in patients) {
    # Get data for the specific patient and sort by SEQ_NUM
    patient_data <- data[SUBJECT_ID == patient][order(SEQ_NUM)]

    # If window is NA, set window to be the number of rows for this patient
    if (is.na(window)) {
      patient_window <- nrow(patient_data)
    } else {
      patient_window <- window
    }

    # For each ICD code in patient data, compare with others within the window
    for (i in 1:(nrow(patient_data) - 1)) {
      for (j in (i + 1):nrow(patient_data)) {
        if (patient_data$SEQ_NUM[j] - patient_data$SEQ_NUM[i] > patient_window) break

        code_i <- patient_data$ICD9_CODE[i]
```

```
        code_j <- patient_data$ICD9_CODE[j]

        # Update the co-occurrence matrix
        cooccurrence_matrix[code_i, code_j] <- cooccurrence_matrix[code_i, code_j] + 1
        cooccurrence_matrix[code_j, code_i] <- cooccurrence_matrix[code_j, code_i] + 1
      }
    }
  }

  return(cooccurrence_matrix)
}
```