OOSD Assignment 2025 - Encoding Words with Suffixes

Author: Jamie Buick
Version: Java 21

Description

This is a console based application that encodes and decodes text given two input files: - A plain text file - An encoding mappings CSV file. This contains the mapping between text and numeric codes

The encodings mapping file contains: - punctuation - numbers - words - word suffixes (e.g. -ing, -ed, -ly)

Each item type above has it's own unique numeric value used for encoding and decoding the text.

Menu

- Use to control the application.
- Full flexibility to enter custom files (As long as formatting and file type correct).
- Inspired by the Student Manager menu system.

Read Encodings File

- The process of reading the text file was influenced by the Vigenere TextFileProcessor Class.
- The encodings CSV file is parsed into an array called **encodingMap** which is a large fixed sized 2D array, this contains the text and its numerical equivalent.
- Reading the full file ensures the file is only opened and read once to preserve resources.

Input file Processing

- The process of reading the text file was influenced by the Vigenere TextFileProcessor Class.
- The input text file is parsed line by line and split into words, the words are trimmed and converted to lowercase to match the format of encoding mapping.
- Line breaks are preserved by adding a special token @@NEWLINE. All words and tokens are added to the **encoderDecoderInput** array.
- When reading is complete, **encoderDecoderInput** is passed to either the encode **EncoderDecoder.encode** or **EncoderDecoder.decode** depending on the menu selection.

- All file reading operations are completed using BufferedReader used in the lab classes.

Encoding Process

Encodes elements of **encoderDecoderInput** based on the **encodingMap**. The encoding process involve: - Detecting newlines using the special token added during parsing - to preserve formatting. - Detecting and separating lone, leading and trailing punctuation. - Attempting a punctuation encoding, e.g. ! → 1. - Attempting a 'full-match' encoding, e.g. love → 338. - Attempting a prefix-suffix encoding, e.g., hello → prefix: hell = 1879 & suffix: @@o = 73, encoded as two parts. - Detecting elements with no match. These are encoded with a 0. - Encoding punctuation and adding it to the **encodings** array relative to its associated word. Punctuation can be handled one its own, at the start of word or up to two characters at the end of a word. - Constructing a clean, trimmed array of encoded words and punctuation ready to be written to a .txt file. - Dynamically increase the size of the **encodings** array for memory efficiency. - Added diagnostics to the encoding process, the user is provided with information regarding the types of items encoded (full match, prefix suffix etc) and unknown encodings. This is presented in console.

Decoding Process

Decodes elements of **encoderDecoderInput** based on the **encodingMap**. The decoding process involve: - Detecting newlines using the special token added during parsing - to preserve formatting. - Find an encoding match from the **encodingMap** for the word or punctuation. - Words with no match will have an encoding of 0. These are decoded as [???]. - Build prefix-suffix words and punctuation around words in the correct format. - Construct a clean, trimmed array of correctly structured words and/or punctuation that can be written to a .txt file - Dynamically increase the size of the **decodings** array for memory efficiency. - Added diagnostics to the decoding process, the user is provided with information regarding the types of items decoded (full match, prefix suffix etc) and unknown decodings. This is presented in console.

Output file Processing

- Writes only valid, non-null or not empty elements to a text file using BufferedWriter in a similar manner to the Vigenere TextFileProcessor Class.
- Each element is printed followed by a space unless it is a newline which is written as just "\n" to preserve original formatting.
- The **encoderDecoderInput** & **encoderDecoderReturn** are cleared after processing to ensure a clean state for next operations.

Main Features

- User can input custom files.

- Encodes and decodes text file using a mapping CSV.
- Dynamic arrays for memory efficiency.
- Handle line breaks to ensure formatting is preserved.
- Handles punctuation and formats correctly during decoding.
- Cleans up variables and arrays after execution to ensure the program runs consistently.
- Encodings and decoding summary printed for the user.

How to use

- Run the console application.
- You MUST specify an Encodings map file (./encodings-10000.csv) that has format (word, encoding). It MUST be a CSV file.
- You MUST specify a text file to encode or decode. It must be a text file and exist.
- You MUST specify an output file. This can have any name or location.
- Select encode or decode options
- The output file will be created and available in the specified location.

Notes

This project follows the single responsibility principle, I had to refactor the TextFileProcessor and EncoderDecoder classes because the original design violated SRP. Instead of passing a full array to the encoder/decoder I was passing single words. This significantly reduced the speed of the application and violated the SRP as I had to process the encodings/ decodings in two classes. Arrays are used for this assignment to avoid the use of array lists. This required a lot of array indexing when encoding and decoding. This required dynamic arrays, as seen in Student Manager to be created to ensure efficient memory usage.