Simulation Kaggle Problem: Santa 2024

Barrera Mosquera Jairo Arturo 20222020142
Martinez Silva Gabriela 20231020205

June 2025

1 Normalize input

1.1 Data Preparation

The dataset is expected to be retrieved from a Kaggle competition used in Workshop #1. In our simulation, the CSV file `sample_submission.csv` contains a column named `text` that holds the raw textual data.

- **Cleaning:** Text is normalized using regular expressions to extract alphabetical tokens only.
- **Preprocessing:** The text is converted to lowercase to ensure uniformity in token matching.
- **Summary:** Each row of the CSV file is analyzed to extract and count words.

1.2 Simulation Planning

System Design Mapping

This component aligns with the "Normalize input" module from Workshop #2's system design.

Scenario Definition

We simulate a preprocessing pipeline that would be part of a data ingestion microservice in a larger Kaggle competition system. This module ensures:

- Input text normalization.
- Conversion of raw inputs into structured data (word counts).

Flow Diagram of the Process

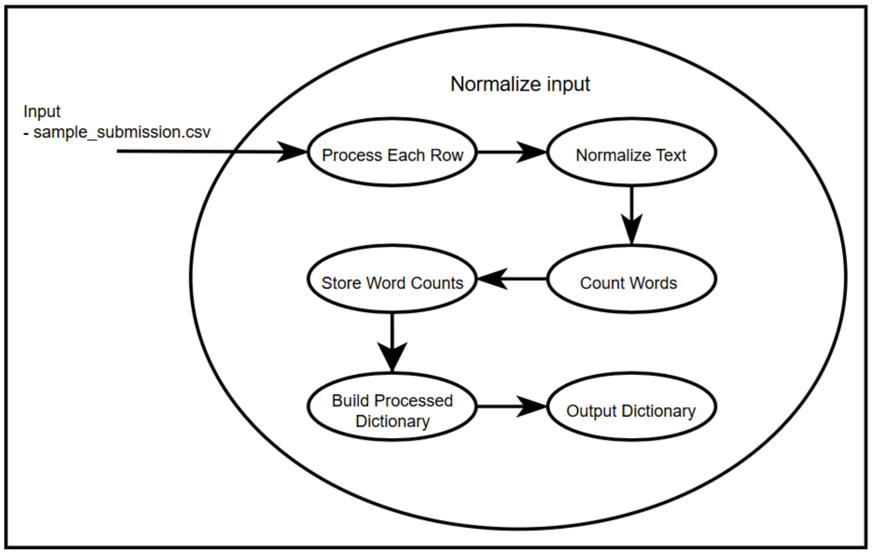

Figure 1: Data Preprocessing Flow – Normalize Input

1.2.1 Constraints and Metrics

- Constraints: The current simulation only extracts and processes words containing English alphabetic characters (A-Z, a-z).

  It ignores numbers, special characters, punctuation, and words with accents or diacritics (e.g., "Café", "résumé").

  Text in other languages or scripts (such as Japanese, Mandarin, or any non-Latin alphabet) will not be recognized or counted as words. Words with hyphens, underscores, or other non-alphabetic symbols are also excluded from processing.

  In short: The simulation only handles plain English words and cannot process non-alphabetical characters or non-English languages/scripts.

- Metrics;In the provided NormalizedInput class, metrics such as processing time and memory usage are collected during the execution of the process method. This practice is justified in systems engineering know the effencyency. For the sample submission the delay was 0.0453 seconds, and the use of memory was 0.10 MB.

1.3 Simulation Implementation

Diagram Description

Each block in the diagram maps to specific parts of the Python code:

- **Read CSV File:** (`read_csv()`) loads the raw CSV file into a pandas DataFrame.
- **Normalize Text:** (`normalize_text()`) converts text to lowercase and extracts alphabetic words using regex.
- **Count Words:** (`count_words()`) creates frequency dictionaries for each row's words.
- **Build Processed Dictionary:** (`process()`) aggregates word counts into a dictionary mapping row IDs to word frequency dictionaries.
- **Output:** (`get_processed_dictionary()`) provides the final dictionary row_id: word: count.

Code Listing

Listing 1: Python Class for Text Preprocessing

```python
import pandas as pd
import re
from collections import Counter

class NormalizedInput:
    def __init__(self, csv_path, text_column):
        self.csv_path = csv_path
        self.text_column = text_column
        self.df = None
        self.processed_df = None

    def read_csv(self):
        print("[NormalizedInput] Reading CSV file:", self.csv_path)
        self.df = pd.read_csv(self.csv_path)
        print("[NormalizedInput] CSV loaded. Shape:", self.df.shape)

    def normalize_text(self, text):
        words = re.findall(r'\b[a-zA-Z]+\b', str(text).lower())
        return words

    def count_words(self, words):
        return dict(Counter(words))

    def process(self):
        print("[NormalizedInput] Starting processing of DataFrame.")
        if self.df is None:
            self.read_csv()
        print("Original CSV:")
        print(self.df)
        processed_dict = {}
        for idx, row in self.df.iterrows():
            words = self.normalize_text(row[self.text_column])
            word_count = self.count_words(words)
            row_id = row['id'] if 'id' in row else idx
            processed_dict[row_id] = word_count
        self.processed_df = processed_dict
        print("[NormalizedInput] Processing complete.")

    def get_processed_dictionary(self):
        if self.processed_df is None:
            self.process()
        return self.processed_df

    def get_processed_df(self):
        return self.processed_df
```

1.4 Executing the Simulation

To observe the performance and verify outputs, the simulation is executed as follows:

1. The CSV is loaded.
2. Each text row is normalized and counted.
3. The resulting word count matrix is printed.

During execution, varying text entries help evaluate the system's handling of punctuation, casing, and empty inputs. Work in tandem with the mwtrics of time and memory previously exposed.

1.5 Results and Discussion

Results

- A DataFrame is dictionary is created with id's, words and appearances of these words for each unique text

Discussion

- **Strengths:** Modular structure, robust against missing or malformed text.
- **Limitations:** Assumes English alphabet only; does not remove stopwords or perform stemming.
- **Improvement Ideas:** Extend the pipeline with NLP preprocessing (e.g., NLTK or spaCy integration), stopword removal, or lemmatization.

2 Templates generation

2.1 Data Preparation

This simulation extends the text normalization pipeline:

- The **NormalizedInput** module reads raw text from a CSV file, normalizes, splits, and counts words.
- Its **output**, a dictionary of word frequencies per row, serves as the **input** for the new **TemplatesGeneration** module.
- The combined pipeline moves from data cleaning to advanced linguistic generation.

2.2 Simulation Planning

Scenario Definition

The new scenario simulates sentence generation from the normalized text. It mimics a component that builds coherent sentences from labeled parts of speech (POS) for downstream tasks.

System Design Mapping

- **NormalizedInput:** Handles ingestion, cleaning, word count output.
- **TemplatesGeneration:** Uses SpaCy to classify words by POS, maps them to grammatical templates, and generates valid sentence candidates.

Flow Connection Diagram

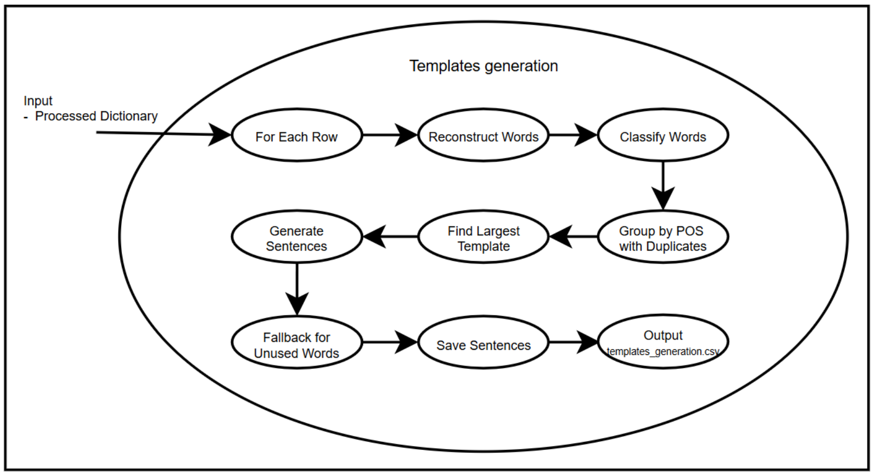

Figure 2: Data Preprocessing Flow – The output of NormalizedInput feeds TemplatesGeneration

Constraints and Metrics

- **Constraints:** Computational load of NLP tagging, combinatorial explosion in sentence permutations.
- **Constraints:** The generated text dont use all the words so the generated sentence in the final must be colapse for complete all the required words
- **Metrics:** Number of valid sentences generated, coverage of words, grammatical coherence.

2.3 Simulation Implementation

2.3.1 Combined Process Description

- **Input Processing:** Receives row_id: word_counts from `NormalizedInput`.
- **Word Reconstruction:** Converts word counts to duplicate-inclusive lists (e.g., "the":2 → "the","the").
- **POS Tagging:** (`classify_words_in_row()`) uses spaCy to tag each word occurrence, handling ambiguous words.
- **Grouping with Duplicates:** (`group_words_by_pos_with_duplicates()`) organizes words into POS groups, tracking duplicates as unique (word, index) pairs.

- **Template Selection:** (`get_largest_template()`) identifies the most complex fillable template from predefined options.
- **Sentence Generation:** (`generate_sentences_for_template_with_duplicates()`) computes all valid word occurrence combinations for the template.
- **Fallback Handling:** Applies fallback templates to unused words to maximize word utilization.
- **Output:** Saves all generated sentences to `templates_generation.csv` with row IDs.

Code Listing: TemplatesGeneration

Listing 2: Template-Based Sentence Generator

```
import pandas as pd
import spacy
from spacy.tokens import Doc
from itertools import product
from collections import defaultdict

# POS and templates definitions...

class TemplatesGeneration:
    def __init__(self, normalized_input):
        self.normalized_input = normalized_input
        self.processed_dict = self.normalized_input.get_processed_dictionary()
        self.all_sentences = []
        self.nlp = spacy.load("en_core_web_sm")

    def classify_words_in_row(self, row_id, words):
        doc = Doc(self.nlp.vocab, words=words)
        for name, proc in self.nlp.pipeline:
            doc = proc(doc)
        results = []
        for token in doc:
            pos_list = [POS_CATEGORIES.get(token.pos_, token.pos_.lower())]
            if token.text in AMBIGUOUS_WORDS:
                for amb_pos in AMBIGUOUS_WORDS[token.text]:
                    if amb_pos not in pos_list:
                        pos_list.append(amb_pos)
            results.append({'id': row_id, 'word': token.text, 'pos_list': pos_li
        return results

    # Remaining methods for grouping, template selection, generation...
```

```
def generate(self):
    for row_id, word_counts in self.processed_dict.items():
        # Build words list, classify, group, generate sentences...
        pass
    sentences_df = pd.DataFrame(self.all_sentences)
    sentences_df.to_csv("sentence_generation.csv", index=False)
```

2.4 Executing the Simulation

1. Run `NormalizedInput` to produce word count dictionaries.
2. Feed the dictionary to `TemplatesGeneration`.
3. Use SpaCy to tag parts of speech.
4. Generate sentences by filling grammatical templates.
5. Store results in `sentence_generation.csv`.

2.5 Results and Discussion

- **Results:** Each row's normalized words are grouped and used to build grammatically valid sentence variations.
- **Discussion:** Shows how word frequency info supports sentence construction, improving data-driven text generation.

3 Sentences Generation Module

3.1 Data Preparation

This final module takes as input:

- **Sentence candidates:** The file `templates_generation.csv` produced by the `TemplatesGeneration` module.
- **Word bag:** The word frequency dictionary from the `NormalizedInput` module, ensuring that all original words are reused.

3.2 Simulation Planning

Scenario: Combine generated sentences for each row such that:

- The combined text covers the entire original bag of words (or as much as possible).
- Sentences are selected in a greedy manner, prioritizing longer ones that do not exceed word counts.

Design Alignment: This step validates the flow from normalized input, to sentence generation, to final output text.

Flow Connection Diagram

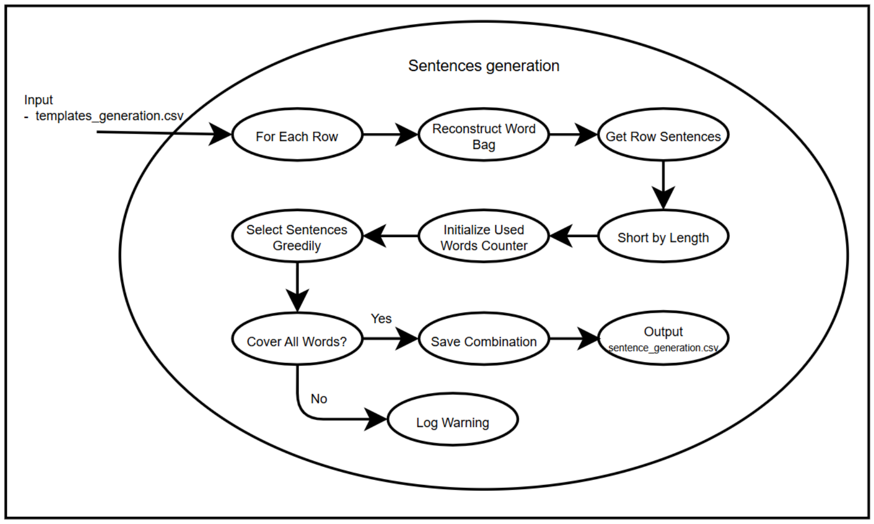

Figure 3: Data Preprocessing Flow – The output of TemplatesGeneration feeds SentencesGeneration

Constraints and Metrics:

- **Constraints:** Overlapping word usage must not exceed original frequencies.
- **Metrics:** Final submission a feedback by Perplexity.

3.3 Simulation Implementation

Combined Process Description

- **Input Processing:** For each row in the dataset, initialize the process of reconstructing the original word bag and selecting candidate sentences.
- **Reconstruct Word Bag:** Rebuild the list of words for the row using the word counts from the normalization stage, ensuring that duplicates are preserved and each word occurrence is tracked.
- **Get Row Sentences:** Retrieve all candidate sentences generated for this row from `templates_generation.csv:`. Each sentence is split into its constituent words for further processing.
- **Sort by Length:** Sort the candidate sentences in descending order by the number of words they contain. This prioritizes longer sentences, which are more likely to cover more words efficiently.

- **Initialize Used Words Counter:** Initialize a counter to keep track of how many times each word from the original row has been used in the selected sentences so far.
- **Select Sentences Greedily:** Iterate through the sorted candidate sentences:
  - For each sentence, check if adding it would exceed the original word count for any word.
  - If valid, add the sentence to the selection and update the used words counter.
  - Continue until all word occurrences are covered or no more candidates remain.
- **Cover All Words?:** Check if the used words counter matches the original word bag for the row (i.e., every word is used exactly as many times as it appeared in the original input).
- **Save Combination:** If all words are covered, join the selected sentences into a single combined sentence for the row and save it to the output.
- **Log Warning:** If not all words can be covered (due to template limitations or sentence overlap), log a warning for the row, but still save the best possible combination of sentences.
- **Output:** Saves all generated sentences to `sentence_generation.csv` with row IDs.

Updated Process Diagram: This module takes the output of `TemplatesGeneration` and completes the loop:

- **Input:** `sentences_generation.csv` and `NormalizedInput dictionary` (`sentences_generation.csv` is re rewritten)
- **Output:** `sentences_generation.csv`

Code Listing

Listing 3: Combine Sentences Module

```
import pandas as pd
from collections import Counter

class SentencesGeneration:
    def __init__(self, sentence_csv_path, processed_dict):
        self.sentence_csv_path = sentence_csv_path
        self.processed_dict = processed_dict
        self.sent_df = pd.read_csv(sentence_csv_path)
        self.final_sentences = []

    def combine_sentences_per_row(self):
        for row_id, word_count_dict in self.processed_dict.items():
            original_bag = []
            for word, count in word_count_dict.items():
                original_bag.extend([word] * count)
            original_counter = Counter(original_bag)
            n_words = len(original_bag)

            row_sentences = self.sent_df[self.sent_df['id'] == row_id]['sentence
            sentence_word_lists = [s.split() for s in row_sentences]

            used_counter = Counter()
            selected_sentences = []
            for words, sent in sorted(zip(sentence_word_lists, row_sentences), k
                temp_counter = Counter(words)
                if all(used_counter[w] + temp_counter[w] <= original_counter[w]
                    selected_sentences.append(sent)
                    used_counter += temp_counter
                if sum(used_counter.values()) == n_words:
                    break

            if sum(used_counter.values()) < n_words:
                print(f"Warning: Could not cover all words for row {row_id}.")

            self.final_sentences.append({
                "id": row_id,
                "combined_sentence": " ".join(selected_sentences)
            })

    def save(self, output_path="sentences_generation.csv"):
        pd.DataFrame(self.final_sentences).to_csv(output_path, index=False)
        print(f"Combined sentences saved to {output_path}")
```

3.4 Executing the Simulation

1. Ensure that `NormalizedInput` and `TemplatesGeneration` have run.
2. Run `combine_sentences_per_row()`.
3. Save the results.

3.5 Results and Discussion

Results: Each final sentence covers as many original words as possible using valid grammatical structures.

Discussion: The greedy approach works well for short rows. For larger rows or overlapping words, the algorithm issues a warning if full coverage isn't possible.

4 Chaos managment

The chaos atractors seen in the workshop 2 were treated as following:

- **Morphological Ambiguity:** Inherentambiguity in parts of speech is reduce using the templates sytems, in this way the generated sentences follow a defined structure.
- **Combinatorial Explosion:** The use of templates reduce the posibilities of combinations to meaningful sentences describe as:

Listing 4: Templates structure

```
TEMPLATES = [
['determiner', 'adjective', 'noun', 'verb', 'preposition', 'determiner', 'no
['determiner', 'adjective', 'noun'],
['determiner', 'noun', 'verb', 'noun'],
```