Design Kaggle Problem: Santa 2024

Barrera Mosquera Jairo Arturo 20222020142
Martinez Silva Gabriela 20231020205

May 2025

1 Review

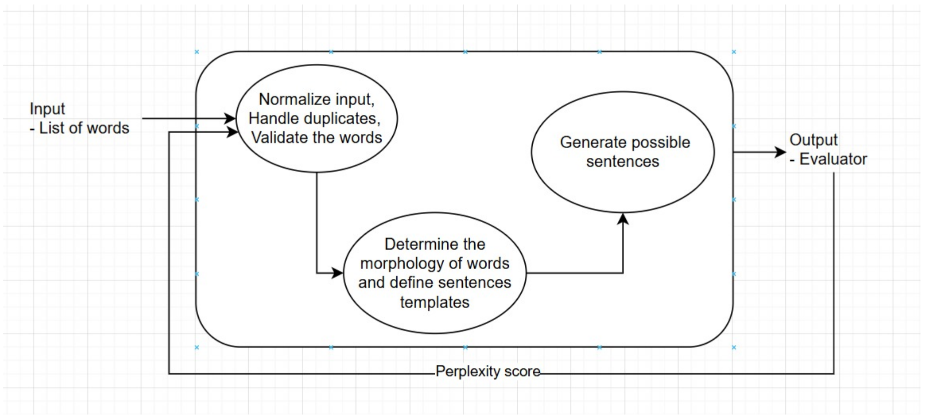

Input
- List of words
Normalize input, Handle duplicates, Validate the words
Determine the morphology of words and define sentences templates
Generate possible sentences
Output
- Evaluator
Perplexity score

Figure 1: System Structure

1.1 Data

The data ingestion is resume with the Santa sequense of words supose to be organized in a passage.

The first interaction with this data is going to be with the element 'Normalized', in this element the data is transform in a way that the words can be permutate.

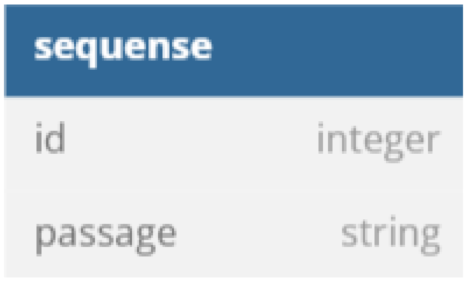
1.1.1 Data structure

Multiple sequense of words attach to an id

1.2 Out comes

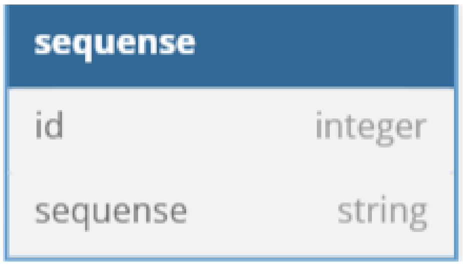
Each passage with his id

1.3 Data figures



sequense	
id	integer
passage	string

Figure 2: Outcome



sequense	
id	integer
sequense	string

Figure 3: Input Data

1.4 Constraints

- **Input Format:** All sequences must follow the competition's prescribed format exactly—no insertions, deletions or spelling changes. The system must support variable-length sequences without failure.
- **Normalization Stage:** Internally you may standardize cases or whitespace for analysis, but outputs must preserve each word's original case and punctuation.
- **Morphological Analysis:** Accurately infer part-of-speech tags and syntactic dependencies. Analysis errors should be minimized to avoid cascading mistakes downstream.
- **Sentence Generation:** Permutations must respect both grammar and semantics. Algorithms should be optimized to handle factorial growth in sequence length efficiently.

1.5 Chaos Attractors

- **Input Variability:** Fluctuations in sequence length or rare word distributions can unpredictably steer permutation search.
- **Normalization Edge Cases:** Repeated tokens, punctuation marks, and mixed capitalization may slip through normalization checks, causing downstream token-matching errors.
- **Morphological Ambiguity:** Inherent ambiguity in parts of speech or dependencies can mislead the generator, compounding errors in later stages.
- **Combinatorial Explosion:** The factorial number of possible permutations amplifies sensitivity: minor reordering can dramatically change perplexity scores or semantic coherence, creating a "butterfly effect."

2 System Requieriments

Functional Requirements (FRs): The functional requirements related to input and data usage are as follows:

- **Sentence Usage:** The system must operate only on the shuffled sentences provided by Kaggle.
- **Word Constraints:** The system must not add, modify, or remove any words from the original input.
- **Sequence Length Handling:** The system must be capable of processing sequences of varying lengths.

Non-Functional Requirements (NFRs) The non-functional requirements are related to the efficiency of permutation generation and evaluation.

- **Perplexity:** The final sentence should have a low perplexity score.
- **Semantic Coherence:** The resulting sentence must be grammatically and semantically correct.

3 Architecture

The system architecture is designed to efficiently process shuffled word lists and reconstruct coherent passages. It follows a modular approach, breaking down the complex problem of permutation and evaluation into manageable, interacting components. This structure, depicted in Figure 1, is justified by principles of Systems Engineering aimed at managing complexity, ensuring data integrity, and optimizing performance.

3.1 Normalized Input

Description: This is the initial processing stage upon receiving a shuffled list of words for a specific passage ID. The key functions are:

- Reading the raw input data.
- Validating the format and content of the input (e.g., ensuring words are strings, handling potential encoding issues).
- Normalizing word representations (e.g., consistent lowercasing if deemed appropriate for internal processing, though original case must be preserved for output).
- Identifying and counting the occurrences of each unique word to correctly handle duplicates as per the competition rules.

The output of this stage is a structured representation of the input word list, including the original words, their exact counts, and potentially basic linguistic annotations (like initial tokenization consistency).

Systems Engineering Justification:

- **Input Validation and Verification:** Ensures that the data entering the system meets required standards and constraints. This prevents errors propagating to downstream components and reduces debugging effort .
- **Data Integrity:** By accurately counting duplicates and preserving the original word forms, this component maintains the integrity of the input data set throughout the process, adhering strictly to competition rules .
- **Clear Interface:** Provides a clean, standardized data structure to the subsequent modules, decoupling them from the specifics of the raw input format .
- **Pre-processing for Efficiency:** Performing tasks like counting duplicates early avoids redundant processing later in the permutation generation phase. Normalization aids consistent linguistic analysis.

3.2 Define Templates

Description: This component takes the normalized word list and performs linguistic analysis to infer possible grammatical structures and relationships within the given set of words. Using techniques like Part-of-Speech (POS) tagging and potentially dependency parsing, it identifies the potential roles of each word (noun, verb, adjective, punctuation, etc.). Based on this analysis, it may propose one or more structural "templates" or constraints that a valid sentence permutation should ideally follow. These templates are not rigid rules but rather probabilistic guides to prune the search space.

Systems Engineering Justification:

- **Abstraction and Modeling:** Creates a higher-level abstraction (linguistic roles, potential structures) from the raw words, allowing the system to reason about grammatical correctness without exhaustively checking every permutation .
- **Complexity Management:** By identifying likely structures and relationships, this component significantly reduces the effective search space for permutations compared to a brute-force approach. It focuses the search on grammatically plausible arrangements.
- **Modularity:** This component encapsulates the linguistic analysis logic, separating it from the permutation generation mechanism. This makes the system easier to understand, develop, and modify .
- **Foundation for Generation:** Provides essential structural information to the Sentence Generation module, guiding it towards producing more meaningful and grammatically sound candidates .

3.3 Generate Permutations

Description: This is the core engine responsible for creating candidate sentence orderings using the words from the normalized input, guided by the templates or structural information provided by the "Define Templates" module. Given the combinatorial explosion ($n!$ permutations for n words), this module cannot simply enumerate all possibilities for longer sequences. It must employ intelligent search strategies (e.g., informed search, local search, genetic algorithms) that prioritize permutations likely to result in a coherent passage. This module interacts directly with the Evaluator module.

Systems Engineering Justification:

- **Functionality Implementation:** Directly addresses the primary function of the system: generating possible solutions to the reordering problem .
- **Iterative Refinement:** Operates in a loop with the Evaluator, using the perplexity score and potentially other metrics to guide its search towards better solutions. This feedback loop is fundamental to optimizing the output quality .
- **Search Space Management:** Employs algorithms designed to navigate a massive potential search space efficiently, rather than relying on brute force, which is infeasible for larger inputs.
- **Interface Definition:** Clearly defines the input (normalized words, templates/constraints) and the output (a candidate sentence permutation sent to the evaluator), facilitating integration with other modules.

4 Sensivity and chaos

4.1 Sensitivity

The model is highly sensitive to input permutations. A small change in word order can significantly affect perplexity scores, other factors affecting system senvity:

- **Text Length:** The greater the number of words in a sentence, the more possible permutations exist, making optimization more difficult.
- **Grammatical Structure:** Some passages may contain ambiguities that impact the perplexity evaluation.
- **Lexical distribution:** Phrases with common words can generate more plausible permutations, while phrases with rare words may have fewer viable options.

4.2 Chaos and Randommess

- **Non-linearity and Chaos:** Small changes in the position of a word can cause a drastic change in perplexity. In some cases, a grammatically incorrect phrase may receive a better score than a correct one.
- **Combinatorial Explosion:** For a sentence with n words, there are $n!$ possible permutations. Evaluating all these combinations is computationally infeasible for high values of n.

5 Implementation

5.1 Normalized Input – Possible Implementation

The implementation of the Normalized Input component primarily involves data loading, basic cleaning, and counting word occurrences.

Libraries/Tools:

- **Pandas:** Essential for reading the input data file (e.g., CSV) into a DataFrame, where each row represents a passage with an ID and a list of shuffled words.
- **Python built-in string methods:** For basic cleaning, such as stripping leading/trailing whitespace from words.
- **collections.Counter:** A highly efficient way to count the occurrences of hashable objects (i.e., words) in a list.

- **Optional – NLTK or spaCy (minimal processing):** If robust handling of punctuation or consistent tokenization is needed (e.g., for "word."), basic tokenization using NLTK or spaCy can be applied. However, full linguistic annotation is reserved for the next step.

Implementation Steps:

1. Load the input data file (e.g., `train.csv`) using Pandas: `df = pd.read_csv('train.csv')`
2. Iterate through each row of the DataFrame, where each row corresponds to a unique passage (`for index, row in df.iterrows():`).
3. For the current passage, retrieve the list of words. This may involve splitting a string column if words are provided as a space-separated string: `word_list = row['shuffled_passage'].split()`.
4. Perform basic cleaning on each word if necessary (e.g., `cleaned_words = [word.strip() for word in word_list]`).
5. Use `collections.Counter` to obtain the frequency of each word in the cleaned list: `word_counts = Counter(cleaned_words)`.
6. Store the original word list and the `word_counts` object, associated with the passage ID. This structured data becomes the output for this passage and is ready for the "Define Templates" step.

5.2 Define Templates – Possible Implementation

This component involves linguistic analysis to infer grammatical structure and potential sentence patterns from the jumbled words.

Libraries/Tools:

- **spaCy or Stanza:** Primary tools providing pre-trained models for linguistic annotation.
- **NLTK:** Can be used for POS tagging and access to linguistic resources, though it is often less integrated than spaCy or Stanza.

Implementation Steps:

1. Load a pre-trained English language model from spaCy or Stanza: `nlp = spacy.load("en_core_web_sm")` or `nlp = stanza.Pipeline('en')`.
2. For each passage's normalized word list (from the previous step):

3. Process the list of words using the loaded NLP model. While the words are shuffled, processing them together can still yield useful probabilistic tags and dependencies based on common word co-occurrences: `doc = nlp(" ".join(normalized_word_list))`. (Alternatively, process words individually or in small, likely phrases if the context is too distorted.)

4. Extract linguistic features for each word:
    - Part-of-Speech (POS) tags (`token.pos_` in spaCy, `word.upos` in Stanza).
    - Dependency relations (`token.dep_`, `token.head` in spaCy; `word.deprel`, `word.head` in Stanza).
    - Lemma (`token.lemma_` in spaCy, `word.lemma` in Stanza).

5. Based on the distribution of POS tags and dependency relationships, infer potential sentence templates or constraints. This could involve:
    - Identifying the most likely main verb(s) or noun(s).
    - Recognizing common phrase patterns (e.g., Adjective–Noun, Adverb–Verb).
    - Noting required punctuation marks and their likely positions (e.g., a period at the end).
    - Creating abstract templates like `[NOUN] [VERB] [ADJECTIVE] [NOUN] [PUNCTUATION]`, based on word counts and types.
    - Representing constraints as rules, e.g., "word X must appear before word Y", or "word Z is likely the sentence-initial word".

6. Output the linguistic analysis results (POS tags, dependencies per word) and the inferred templates/constraints, associated with the passage ID, to be used by the "Generate Permutations" module.

5.3 Generate Permutations – Possible Implementation

This is the **search component**. It takes the words, their counts, and linguistic insights to find the best permutation, guided by the Evaluator's score.

Libraries/Tools:

- **Python's itertools:** Useful for generating permutations, though only feasible for very short sequences (n ¡ 10).
- **Custom search algorithms:** Implementation of algorithms like A*, Simulated Annealing, or Genetic Algorithms in Python.
- **Language Models (for evaluation/scoring):** Libraries such as Hugging Face's `transformers` or `tensorflow/pytorch` to load pre-trained language models (e.g., GPT-2, BERT variants) to calculate perplexity. While the calculation is handled by the "Evaluator", the "Generate Permutations" module must call it.

Implementation Steps

1. Receive the normalized word list, word counts, and linguistic templates/constraints for a passage.
2. Initialize a search process. This could be:
    - **Start State:** An empty sequence or a partial sequence based on strong template cues (e.g., a likely starting word).
    - **Candidate Pool:** The set of words available for placement, respecting their counts.
3. Iterative Generation:
    - At each step, select the next word(s) to add to the current partial sequence from the Candidate Pool. This selection is guided by:
        - Adherence to inferred templates or grammatical constraints (e.g., don't place a verb if a subject is clearly missing).
        - Probabilistic likelihood based on the linguistic features of available words and the current sequence's ending (e.g., using trained probabilities or heuristics).
    - Construct potential next sequence states.
    - For each potential state (partial or complete sentence), call the "Evaluator" module to get a score (e.g., perplexity).
    - Use the scores and search algorithm logic to decide which states to explore further

      Search Algorithm Examples:
        - **A* Search:** States are partial sentences. The cost is related to perplexity adding a word to the partial sentence. The path is build based on the potential of remaining words.
        - **Genetic Algorithm:** Individuals are permutations. Fitness is the perplexity score (lower is better). Select parents based on fitness, apply crossover (combine parts of two permutations), and mutation (random swaps) to create new permutations. Repeat across generations.
4. **Termination:** The search stops when a complete sentence using all words is formed and achieves a satisfactory score, or after a predefined number of iterations or evaluations (due to computational limits).