Large-scale Text Processing in Core Python

Overview

This report describes the design and implementation of a series of text-processing tasks carried out on large Wikipedia and Twitter corpora using core Python only. The aim is not to present coursework-style answers, but to explain the logic, structure, and efficiency considerations behind each solution, as if handing the work over to another engineer or data scientist.

Across all tasks, the guiding principles were:

- streaming large JSON files line by line to minimise memory usage
- using appropriate core data structures (sets, dictionaries, Counters) for fast lookup and aggregation
- writing modular, readable code that scales to larger datasets
- avoiding unnecessary passes over the data

Data handling and common utilities

All datasets are read using streaming I/O '(with open(...): for line in fh:)' to avoid loading entire files into memory. Text normalisation is handled centrally through a small utility function that lowercases text, removes punctuation, and splits on whitespace. This ensures consistent tokenisation across tasks while keeping the logic easy to audit and modify.

File paths are resolved once at the start of the notebook using a small path-detection step. This allows the same code to run both in a university JupyterHub environment (where data files reside in the working directory) and in a GitHub repository layout (where files are expected under a data/ directory). Data files themselves are intentionally not included in the repository due to size and licensing constraints.

---

1. Wikipedia: Most frequent unique words

The first task identifies the most frequently occurring word across Wikipedia articles, counting each word at most once per article. The key design choice here is to use a set for each article's tokens before updating a global Counter. This prevents long articles with repeated terms from dominating the result and instead highlights words that appear broadly across the corpus.

The algorithm processes the file in a single pass. For each article, its text is normalised, converted into a set of unique tokens, and used to update the global counter. This results in linear time complexity with respect to the total number of tokens processed and constant-time updates for each word.

---

2. Wikipedia: Longest capitalised phrase

This task finds the longest contiguous sequence of capitalised words in Wikipedia article body text. The solution scans tokens sequentially while maintaining a running count of the current capitalised sequence and tracking the longest sequence seen so far.

By iterating through tokens once and updating simple counters and indices, the algorithm avoids any need for backtracking or multiple passes. This approach is efficient, easy to reason about, and well suited to large inputs. The final output includes both the length of the sequence and the phrase itself, making the result interpretable.

---

3. Wikipedia: Largest anagram set (filtered)

To identify the largest anagram set among words of length six or greater, each word is mapped to an anagram signature defined by its sorted characters. A dictionary maps these signatures to sets of words, ensuring uniqueness regardless of how often a word appears.

Processing is again performed in a streaming manner. Words are normalised and filtered by length before computing their signatures. Using sets as dictionary values prevents duplicates and keeps memory usage under control. After processing all articles, the largest anagram group is selected by comparing set sizes.

This approach runs in linear time relative to the number of tokens processed, with the dominant cost being character sorting per word, which is acceptable given the length constraint.

---

4. Twitter: Top mentioned users

This task extracts user mentions from tweets and identifies the five most frequently mentioned users. Mentions are detected by preserving the @ character during tokenisation while removing other punctuation.

A Counter is used to aggregate mention frequencies efficiently. Each tweet is processed independently, and only tokens starting with @ are considered. The final result is obtained by selecting the top five entries from the counter, which is efficient even for large datasets.

---

5. Twitter: Word with most case variations

Here the objective is to find the word that appears with the greatest number of distinct casing variants (e.g. what, What, WHAT). Tokens are grouped by their lowercase form, while preserving the original casing.

A dictionary maps each lowercase base form to a set of observed variants. Sets ensure that repeated occurrences of the same casing do not inflate counts. The best candidate is tracked incrementally as new variants are discovered, avoiding an additional pass over the data at the end.

This design balances clarity with performance and demonstrates careful handling of text normalisation edge cases.

---

6. Twitter: Longest dictionary-word sequence

This task identifies the longest contiguous sequence of dictionary words that appears in at least ten distinct tweets. Dictionary words are loaded into a set to enable constant-time membership checks.

Each tweet is tokenised and scanned to extract maximal runs of dictionary words. From each run, all contiguous subsequences are generated and stored in a per-tweet set so that each sequence is counted at most once per tweet. A global Counter tracks how many distinct tweets each sequence appears in.

After processing all tweets, the longest sequence meeting the frequency threshold is selected, with ties broken by overall frequency. This approach requires only a single pass over the tweets dataset and avoids unnecessary duplication of work.

---

7. Matching tweets to Wikipedia articles using Jaccard similarity

In this task, tweets are matched to Wikipedia articles based on Jaccard similarity between sets of unique tokens. Both tweet and article texts are converted into sets, and similarity is computed as the size of the intersection divided by the size of the union.

To reduce unnecessary computation, a simple upper-bound pruning strategy is used: if the maximum possible Jaccard score for a candidate article cannot exceed the current best score, it is skipped. This significantly reduces the number of exact similarity computations while preserving correctness.

The result is an interpretable baseline similarity method that scales well and is easy to extend with more advanced techniques if needed.

---

8. Inverted index and wildcard search

The final task builds a shared positional inverted index across both corpora. The index maps each token to document identifiers and the positions at which the token appears. This enables efficient phrase and pattern matching without scanning full documents.

Wildcard queries are parsed into word sequences and gap sizes, representing how many tokens may appear between words. Candidate documents are first filtered using set intersections on document IDs, then verified using positional checks to confirm that the pattern matches.

This two-stage approach (candidate filtering followed by positional verification) keeps the search efficient while supporting flexible wildcard-style queries.

Conclusion and future improvements

This project demonstrates how core Python data structures and careful algorithmic design can be used to process large text datasets efficiently. Streaming I/O, judicious use of sets and dictionaries, and modular functions result in code that is both scalable and maintainable.

Potential extensions include improved tokenisation for social media text, TF–IDF or cosine similarity for document matching, and persisting the inverted index to disk for larger-than-memory corpora. These enhancements could be layered onto the existing structure without major refactoring, reflecting the flexibility of the current design.