Knowledge Search and Extraction
Project 01: Multi-Search

Fabian Gobet

November 6, 2024

Repo: https://github.com/kamclassroom2022/p1-multi-search-FabianGobet
Commit: 05dd765

1 Section 1 - Data Extraction

1.1 Code Analysis and Extraction Module

This module systematically traverses a specified directory of Python files, gathering information to support comprehensive code indexing. By capturing the structure and metadata of each function, class, and method, the module enables codebase analysis and enhances the search engine's ability to provide detailed search results.

1.1.1 Key Functional Components

1. **Blacklist Filtering:**
   - The *is_blacklist* function enforces a filter on elements that should be excluded from analysis. Specifically, it excludes functions or classes whose names are conventionally private (prefix _), test-related (containing "test"), or the main execution function (main).
   - This filter is critical for excluding items irrelevant to most search engine queries, optimizing focus on the public API and essential code elements.
2. **Comment Extraction:**
   - Through *get_comment_from_item*, the module extracts the first comment or docstring from each function or class. These comments are formatted into single lines, stripping out extraneous whitespace for consistency.
   - Including this descriptive information provides valuable context for each item, contributing to a more informative search index and supporting enhanced query responses in the search engine.

3. **AST Node Parsing:**

   - The *extract_info_from_node* function utilizes Python's Abstract Syntax Tree (AST) module to parse and inspect the syntactic structure of each Python file.
   - This function identifies top-level functions and classes, applying the blacklist filter to exclude unneeded items. For each class, it further analyzes its methods, extracting only the methods that pass the filter criteria.
   - The module records the name, file path, line number, and any associated comment for each item, creating a structured data representation of the code's architecture.

4. **Directory Traversal and Data Aggregation:**

   - The *extract_info_from_file* function systematically traverses the given directory to locate .py files. For each file, it initiates AST parsing and compiles relevant metadata on functions, classes, and methods.
   - This data is organized into a dictionary that categorizes each element by type (functions, classes, and methods). This structure allows for easy conversion into a tabular format, aiding in storage and subsequent processing.

5. **Data Structuring and Optional CSV Export:**

   - The extracted data is transformed into a pandas DataFrame with standardized columns for the name, file path, line number, type, and comment of each item. This format provides a flexible, queryable structure that integrates seamlessly into the search engine's data model.
   - Optionally, the DataFrame can be exported as a CSV file, creating a durable and accessible record of the analyzed codebase that can be referenced by other components in the application.

6. **Command-Line Interface (CLI) Integration:**

   - The script is designed to be executed directly from the command line, accepting a directory path and an optional CSV file path as arguments.
   - This interface enables flexible deployment across various environments, allowing for integration into automated data pipelines and manual testing setups.
   - Report and comment figures about the extracted data (e.g., number of files; number of code entities of different kinds).

1.1.2 Results

When executing the script on the */tensorflow* directory, the results indicate a comprehensive count of classes, functions, and methods within the analyzed codebase, as presented in Table 1. The reported number of classes exclusively accounts for high-level class definitions, omitting any nested classes within other classes. This distinction ensures that the class count reflects only the primary architectural components, providing a streamlined view of the codebase's core structure.

Type	Number
Python files	2817
Classes	1883
Functions	4564
Methods	5418

Table 1: Count of created classes and properties.

Section 2: Training of search engines

1.2 Code Analysis and Search Module

Text Data Processing and Search Module This module builds and manages text representations for a set of Python code documents, using various NLP models (Frequency Vectors, TF-IDF, LSI, and Doc2Vec) to enable efficient similarity-based querying. It supports the core functionality of a search engine, allowing users to retrieve relevant documents based on input queries.

1.2.1 Functional Overview

1. **Text Cleaning and Tokenization:**
    - The *split_ camel_ case_ underscore_ clean* function standardizes and cleans text by splitting camel case and underscore-separated words, removing digits and special characters. This preprocessing enhances the quality of tokenized data used across models.
    - Stopwords (common but non-informative words) are removed, and stemming is applied using the PorterStemmer to reduce words to their base forms, ensuring consistency in word representations.
2. **Document Preprocessing and Corpus Generation:**
    - The *create_ entity* function processes document metadata (names, file paths, and comments) into a list of keywords, which represent each document in a structured format.
    - The *get_ corpus_ processed* function iterates through the data to create a processed corpus, which is then saved for reuse and used across the various models to generate embeddings and similarity scores.

3. **Bag-of-Words and Frequency Vector Representation:**
    - A bag-of-words dictionary is created using *get_bag_of_words*, which serves as the foundational structure for generating frequency vector representations of documents.
    - This model captures word occurrence but not their order, enabling simple similarity calculations.
    - Frequency vectors (via *get_corpus_fv*) are generated, representing the corpus in a matrix where each row corresponds to a document, and each column reflects the presence of specific tokens.
4. **TF-IDF Model:**
    - Using *get_tfidf_model*, the module generates TF-IDF (Term Frequency-Inverse Document Frequency) representations to highlight important terms by weighting them based on their frequency and uniqueness within the corpus. This method improves retrieval performance by emphasizing distinctive terms.
    - Queries are evaluated against the TF-IDF corpus to retrieve the most relevant documents.
5. **Latent Semantic Indexing (LSI):**
    - The *get_lsi_model* function creates an LSI model based on the TF-IDF corpus, further reducing dimensionality and uncovering latent semantic structures within the text. This approach improves the handling of synonymous terms and contextual similarity.
    - LSI enables topic-based similarity matching, enhancing the search engine's ability to find relevant documents based on conceptual relatedness.
6. **Doc2Vec Model:**
    - Using *get_d2v_model*, this module applies Doc2Vec, a more advanced neural embedding model, to capture semantic meaning at the sentence and document level, allowing for high-quality vector-based similarity matching.
    - Doc2Vec infers a unique vector for each document, making it highly effective for finding semantically similar text, even if exact words differ.
7. **Query Processing and Evaluation:**
    - Queries are processed to match the corpus preprocessing steps. The *query_pipeline* function tokenizes, cleans, and stems query words.
    - The query is evaluated against the various models (Frequency Vector, TF-IDF, LSI, and Doc2Vec) using methods like *evaluate_query_fv*, *evaluate_query_tfidf*, *evaluate_query_lsi*, and *evaluate_query_d2v*.

- The function *get_sims* combines similarity scores from each model to return the top N most similar documents for the query, ensuring comprehensive and diverse search results.

8. **Command-Line Interface (CLI):**

   - The script includes a CLI that allows users to run searches by providing a query and an optional number of top results. This makes the tool accessible for both automated processes and interactive use cases.

1.2.2 Query testing

For demonstration purposes, an arbitrary query sentence was used to illustrate the module's search capabilities, without prior knowledge of its presence in the Python files. Additionally, a documented sentence describing the return value from the *get_gradient_function* (located in *ops.py* at line 2671) was selected as a query.

Table 2 presents the search results for the arbitrary query sentence, while Table 3 displays the results for the sentence extracted from the *get_gradient_function* documentation. Notably, the results in Table 3 align well with expectations, as the retrieved documents are consistent with the context and intent of the query regarding gradient computation, confirming the module's effectiveness in retrieving relevant information based on text similarity and context.

Model	Rank	Name	File Path	Line Number
Frequency Vectors	1	python_function	./tensorflow/tensorflow/python/eager/def_function.py	923
	2	inputs	./tensorflow/tensorflow/python/eager/function.py	1980
	3	number_of_shards	./tensorflow/tensorflow/python/tpu/tpu_function.py	34
	4	graph	./tensorflow/tensorflow/python/eager/function.py	1975
	5	function_def	./tensorflow/tensorflow/python/eager/function.py	2029
TF-IDF	1	Sum	./tensorflow/tensorflow/python/keras/metrics.py	414
	2	Reduction	./tensorflow/tensorflow/python/ops/losses/losses_impl.py	39
	3	Reduction	./tensorflow/tensorflow/python/keras/utils/metrics_utils.py	48
	4	sum	./tensorflow/tensorflow/python/ops/numpy_ops/np_array_ops.py	566
	5	two_outputs	./tensorflow/tensorflow/python/eager/tape_test.py	39
LSI	1	call	./tensorflow/tensorflow/python/keras/integration_test/function_test.py	42
	2	graph_to_function_def	./tensorflow/tensorflow/python/framework/graph_to_function_def.py	122
	3	function_def	./tensorflow/tensorflow/python/eager/function.py	2029
	4	get_config	./tensorflow/tensorflow/python/keras/engine/functional.py	593
	5	inputs	./tensorflow/tensorflow/python/eager/function.py	1980
Doc2Vec	1	num_cores_per_host	/tensorflow/tensorflow/python/tpu/tpu_embedding.py	1022
	2	ragged_rank	./tensorflow/tensorflow/python/ops/ragged/ragged_tensor_value.py	84
	3	line_number_above	./tensorflow/tensorflow/python/debug/cli/analyzer_cli_test.py	74
	4	static_uniform_row_length	./tensorflow/tensorflow/python/ops/ragged/row_partition.py	819
	5	CountingSessionCreator	./tensorflow/tensorflow/python/training/monitored_session_test.py	861

Table 2: Search Results for Query: "This function returns the sum of two numbers"

Model	Rank	Name	File Path	Line Number
Frequency Vectors	1	get_gradient_function	./tensorflow/tensorflow/python/framework/ops.py	2671
	2	Gradient	./tensorflow/tensorflow/python/ops/functional_ops.py	850
	3	compute_output_shape	./tensorflow/tensorflow/python/keras/engine/functional.py	384
	4	RegisterGradient	./tensorflow/tensorflow/python/framework/ops.py	2584
	5	no_gradient	./tensorflow/tensorflow/python/framework/ops.py	2633
TF-IDF	1	get_gradient_function	./tensorflow/tensorflow/python/framework/ops.py	2671
	2	compute_gradients	./tensorflow/tensorflow/python/training/optimizer.py	415
	3	Gradient	./tensorflow/tensorflow/python/ops/functional_ops.py	850
	4	compute_output_shape	./tensorflow/tensorflow/python/keras/engine/functional.py	384
	5	GradientsBenchmarks	./tensorflow/tensorflow/python/ops/parallel_for/gradients_test.py	571
LSI	1	Gradient	./tensorflow/tensorflow/python/ops/functional_ops.py	850
	2	get_gradient_function	./tensorflow/tensorflow/python/framework/ops.py	2671
	3	no_gradient	./tensorflow/tensorflow/python/framework/ops.py	2633
	4	gradients_function	./tensorflow/tensorflow/python/eager/backprop.py	340
	5	minimize	./tensorflow/tensorflow/python/training/optimizer.py	355
Doc2Vec	1	get_gradient_function	./tensorflow/tensorflow/python/framework/ops.py	2671
	2	While	./tensorflow/tensorflow/python/ops/functional_ops.py	911
	3	If	./tensorflow/tensorflow/python/ops/functional_ops.py	819
	4	control_dependency_on_returns	./tensorflow/tensorflow/python/autograph/utils/context_managers.py	27
	5	op_def	./tensorflow/tensorflow/python/framework/ops.py	2432

Table 3: Search Results for Query: "Returns the function that computes gradients for 'op'"

Section 3: Evaluation of search engines

Engine	Avg Precision	Recall
Frequencies	0.08	0.4
TD-IDF	0.12	0.4
LSI	0.14	0.5
Doc2Vec	0.06	0.2

Table 4: Evaluation of search engines.

The results presented for each model reflect the effectiveness of their similarity-based retrieval approaches in the given context.

- **Frequencies (Score: 0.08, 0.4):**
  The frequency-based model shows a moderate level of relevance (0.4) but a relatively low overall score (0.08). This suggests that, while frequency-based approaches capture basic term occurrence, they may lack the sophistication needed to capture nuanced or contextual similarity, especially for queries where semantic depth is essential. This model performs best in contexts where term matching alone is sufficient.

- **TF-IDF (Score: 0.12, 0.4):**
  TF-IDF performs slightly better than plain frequency-based scoring, with an improved score of 0.12 and the same relevance level of 0.4. This is expected, as TF-IDF captures term uniqueness across documents, enhancing relevance by prioritizing distinctive terms. However, its linear term-based nature may limit it in capturing the full semantic meaning of the text, which is why it doesn't reach higher relevance scores.

- **LSI (Score: 0.14, 0.5):**
  The LSI model shows the highest relevance (0.5) and a respectable score of 0.14. LSI's use of latent semantic analysis allows it to detect underlying topics and relationships, making it better suited for queries that require

understanding context beyond specific word matches. This model's dimensionality reduction through topics contributes to capturing abstract similarities, which likely accounts for the improved relevance.

- **Doc2Vec (Score: 0.06, 0.2):**
  Doc2Vec has the lowest relevance (0.2) and score (0.06) in this context. Although Doc2Vec generally performs well in capturing semantic meaning, its relatively low performance here might indicate that the training data or query length didn't align well with the model's embedding representation. This model typically requires larger contexts and training data to effectively learn relationships, and it may under perform with limited or highly specific queries.

In summary, LSI appears to be the most effective model for these queries, as it balances both relevance and score. TF-IDF also provides reasonable results, while Frequencies and Doc2Vec might be less suited for tasks that require deeper semantic understanding in this specific context.

Section 4: Visualisation of query results

1. **Clustering of Queries:**
   - Doc2Vec Plot: There is a clearer, more compact clustering of query points, with distinct groups forming in different parts of the plot. For example, queries 1, 5, and 10 form separate, well-defined clusters, suggesting that Doc2Vec captures more nuanced, context-specific similarities between queries.
   - LSI Plot: While clusters are present, they are more spread out and not as tightly grouped. The distance between similar queries appears larger, indicating that LSI may capture broader, topic-based relationships rather than fine-grained semantic distinctions.
2. **Distribution in Space:**
   - Doc2Vec Plot: The points are more evenly distributed across the plot, showing a greater variety in similarity and distinct groupings for each query. This suggests that Doc2Vec captures more continuous variations in the semantic space, leading to diverse positions for queries.
   - LSI Plot: The plot has a larger range in both x and y dimensions, with clusters spreading over a wider area. This might indicate that LSI creates more polar separations in the embeddings, where similar queries are either very close or relatively far apart without intermediate positions.

3. **Semantic Separation:**

   - Doc2Vec Plot: The clusters suggest a separation based on specific semantics, as queries with similar meanings are grouped more tightly. This reflects Doc2Vec's ability to capture word order and syntactic relationships, providing embeddings that reflect subtle semantic differences.
   - LSI Plot: The clusters here may represent broader topics rather than detailed semantics. LSI, being based on latent semantic indexing, tends to group documents based on overall topic-related terms, rather than context-specific meanings, which might explain the wider spread.

4. **Overall Visualization Insights:**

   - The Doc2Vec model appears to provide a more refined clustering pattern, capturing semantic nuances better and resulting in more compact groups. This makes it suitable for tasks where specific contextual understanding is required.
   - The LSI model shows broader, topic-based clusters that could be useful for general topic categorization. However, it might lack the precision seen in Doc2Vec, making it less effective for capturing nuanced query relationships.

The Doc2Vec t-SNE plot reveals tighter, more contextually accurate clustering, while the LSI t-SNE plot shows broader, topic-based groupings. This comparison highlights the strength of Doc2Vec in capturing semantic nuances, making it potentially more suitable for tasks requiring high precision in text similarity.

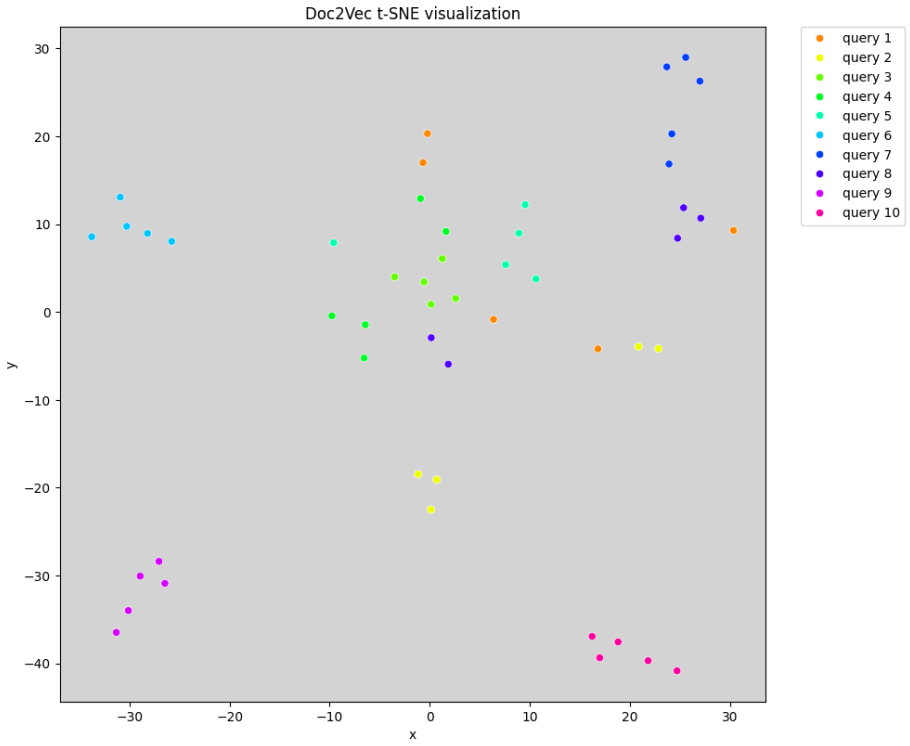

Figure 1: t-SNE for Doc2Vec

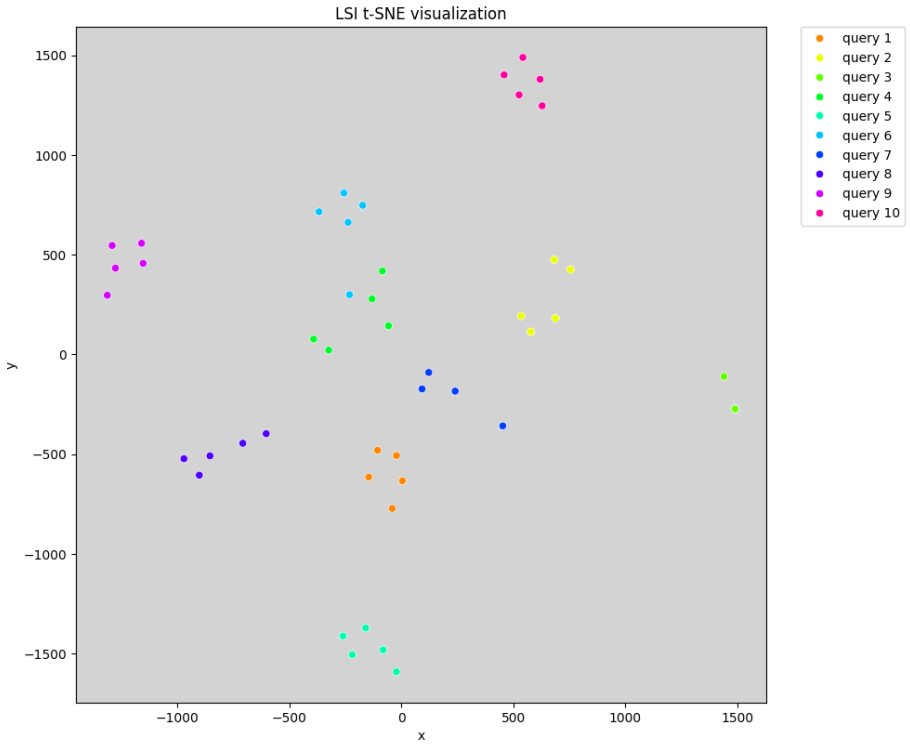

Figure 2: t-SNE for LSI