A Technical Appendices and Supplementary Material

A.1 Memory Bank Design

Our lightweight Memory Bank stores only pointers and short notes from third-party repositories, never full code, and runs in strict No-External-Repo or Library-Only modes. The Coding Agent queries it via a Search Agent (`GPT-5`), which returns Top-K compressed hits—URLs, paths, commit hashes, and brief usage notes—as reuse candidates or hints. Top-K results are ranked by the Deep-Research model with deterministic tie-breaking based on compliance and licenses. Unlike generic RAG agents, the Memory Bank is task-specific, created on demand, and accessed through MCP in Camel AI for precise, context-aware retrieval.

Lightweight store & compliance. We *do not store or import full code bodies* from third-party repos; the Bank only keeps pointers and short notes. Unless a task explicitly permits otherwise, we run in one of two compliant modes: (i) **No-External-Repo** (strict): no non-library code imports; general-purpose libraries are installed via package managers with version pins; (ii) **Library-Only**: identical to strict mode but may reference external repos as *specification hints* for re-implementation. We keep the blacklist identical to the official task blacklist (unmodified).

Lookup flow. When the Coding Agent needs a specific method, it asks the Memory Bank for prior art or hints. A Searching Agent (instantiated with `GPT-5`) performs simple keyword/fuzzy search over titles/summaries/tags and returns Top-K (K=10 by default) *compressed hits*: each hit includes the repo/page URL, optional file/function path, commit hash, and a one-sentence usage note. If a plausible prior implementation exists elsewhere, the hit is returned as a *reuse candidate* (to guide re-implementation in compliant modes); otherwise, only brief textual hints are surfaced to conserve the Coding Agent's context.

Top-K to Top-1 selection. When a single candidate must be chosen (e.g., a base parser or test harness to *re-implement*), we use a two-step rule: (i) the Deep-Research model proposes a priority ranking over the Top-K; (ii) deterministic tie-breaking by *hard* constraints (blacklist/whitelist compliance, license allowlist, version pinning, and dry-run compilation when applicable). Prompts and default K are provided in the supplement.

Why not generic RAG? Unlike generic RAG agents [5, 4, 2] that rely on large Web-crawled corpora (often noisy and low-relevance), our Bank is *lightweight* and *task-specific*: its contents are created on demand by Deep Research and tightly aligned with the reproduction goal. This targeted retrieval keeps the Coding Agent focused on precise, context-aware knowledge rather than being distracted by irrelevant text, improving both efficiency and correctness.

Search Agent. When the Coding Agent attempts to access content in the memory bank, the Search Agent queries the bank and forwards the required materials to the Coding Agent. Through the Model Context Protocol (MCP) [1] tooling in the Camel AI framework[3], the Search Agent can follow links stored in the memory bank, perform targeted retrieval, and—after LLM-based reading and analysis—return precise information. In our implementation, we use `o3-mini` for tool invocation and for generating the responses.

B Prompts

Using the paper "Semantic Self-Consistency: Enhancing Language Model Reasoning via Semantic Weighting" as an example, we present a complete prompt.

B.1 Breakdown

System Prompt:

You are a research assistant specialized in generating high-precision search queries for academic paper reproduction. Your expertise lies in creating comprehensive, copy-paste ready search strings that maximize discovery of relevant resources across data, code, papers, and implementation details.

TASK STRUCTURE: Break down any research paper reproduction into 4 core modules: 1. Data Component - datasets, benchmarks, preprocessing pipelines 2. Supporting Component - libraries,

frameworks, dependencies, tools 3. Core Algorithm Component - method implementations, architectures 4. Experimental Component - evaluation metrics, baselines, studies

SEARCH QUERY REQUIREMENTS:

- Detailed description so that I can use deep research effectively, it's better to have some examples - Target multiple resource types: repositories, papers, datasets, docs - Balance specificity (precision) with coverage (recall) - Prioritize recent and well-maintained resources

User Prompt

Human: I'm reproducing a research paper and have structured my approach into 4 modules: Data Component | Supporting Component | Core Algorithm Component | Experimental Component.

For each module, I need a set of high-precision search queries that I can directly copy-paste into deep search engines. These queries should include advanced search operators like site:, filetype:, intitle:, stars:, etc., synonymous expressions to maximize resource discovery.

The goal is to quickly find:

- Relevant datasets and preprocessing code - Required libraries and implementation frameworks - Configuration files and hyperparameter settings - Core algorithm implementations and model architectures - Experimental setups and evaluation benchmarks

Please generate "high-hit" search queries for each of the 5 modules that will help me gather all necessary resources, code repositories, papers, and implementation details before I start the actual reproduction work.

Give me the detailed description of the prompts so that I can use deep research effectively. It's better to have some examples.

The paper I'm reproducing is: [PAPER_LINK_HERE]

B.2 Deep-Research

After breaking down the paper, the 4 searchable corresponding modules will each generate prompts for deep research, which will be input into the deep research system.

B.2.1 Data Deep Research prompt

Deep Research Brief — Datasets (AQuA-RAT / SVAMP / StrategyQA)

Important: No direct URLs are provided in this prompt. You must independently discover, verify, and record the canonical sources and mirrors.

Goal

For AQuA-RAT, SVAMP, and StrategyQA, collect:

official and mirror download sources,

evaluation splits and commonly cited test counts,

exact licenses,

official and general-purpose evaluation scripts,

CoT few-shot exemplars (from paper appendices),

robust answer-extraction rules for CoT outputs (e.g., patterns like “The answer is …”).

Deliverables

A concise Markdown table:

| Dataset | Official Download | Mirror(s) | Split & Counts | License (verbatim) | Official Eval (script/entrypoint) | Harness Task (if any) | CoT Exemplars (appendix ref) | Parsing Notes |

Command recipes (runnable after you resolve file names/paths):
a) downloading data via wget/curl or datasets.load_dataset("<owner>/<name>"),
b) running official evaluation scripts (CLI, required args),
c) running lm-evaluation-harness for aqua-rat, svamp, strategyqa (include --num_fewshot, --seed, and --output_path).

CoT Examples: 2–3 few-shot exemplars per dataset; cite the appendix section/page (title, venue, year).
Parsing Rules: 2–3 regex patterns per answer type (numeric, multiple-choice A–E, yes/no), with precedence and fallbacks.
Verification log: short notes describing how split counts and licenses were cross-checked across primary sources and how disagreements were resolved.

Precision Constraints
Prefer primary sources (official repos, dataset cards, original papers/PDFs).
Quote license strings verbatim as shown on the authoritative page.
When multiple mirrors exist, prefer the canonical origin; record mirrors as backups.

Cross-check and report the commonly cited counts (list disagreements explicitly and state which you adopt for reproduction):
AQuA-RAT: dev/test ≈ 254 each
SVAMP: evaluation set 1000
StrategyQA: test 687
Output Format (exactly in this order)

Table
Download & Setup (runnable commands)
CoT Examples (2–3 per dataset with appendix references)
Parsing Rules (regex + precedence + fallbacks)

Verification Notes (1–2 paragraphs)

Answer-Extraction & Self-Consistency Rules (include verbatim in your output)

Primary pattern (numeric):
r"[Tt]he answer is[:\\s]*(-?\\d[\\d,\\.]*?)\\b"

Multiple-choice (A–E):
r"[Tt]he answer is[:\\s]*\\(?([A-Ea-e])\\)?"

Yes/No:
r"[Tt]he answer is[:\\s]*(yes|no)\\b"

Fallbacks:

scan the last 1–2 sentences for a standalone value after Answer:/Ans:/Final:;

if a standardized marker like #### appears (GSM8K-style), prefer the token following it;

normalize commas/casing before comparison;

if multiple candidates appear, apply self-consistency (majority vote over k sampled CoT paths) and return the mode.

B.2.2 Supporting Components Deep Research prompt

Deep Research Brief — Supporting Components (Embeddings • Similarity • Outliers • Vector Index • Inference & Prompting • Parsing)

Important: This brief intentionally contains no direct URLs. You must independently discover, verify, and record canonical sources and mirrors.

Goal
Identify, compare, and provision the core supporting components for our reproduction stack:

Semantic embeddings (SciBERT / RoBERTa family; optionally MathBERT / Sentence-Transformers).
Similarity computation (cosine / pairwise batched).
Outlier detection on embedding space (IsolationForest / One-Class SVM / KNN-distance).
Vector index / store (FAISS as baseline; note alternatives).
Multi-sample CoT & self-consistency (majority vote).
LLM inference & APIs (OpenAI; OSS via vLLM / TGI for Llama/Mistral).
Prompting/orchestration & parsing (temperature sweeps, queues, multi-sample runners, regex extractors).

Deliverables

A comparison table:

| Component | Shortlist (name + maintainer) | Version & License | Minimal Install | 10–20 line Usage Snippet | Perf Notes (CPU/GPU/RAM) | Tuning Knobs (top-k, thresholds, temps) | Integration Notes |

Ready-to-run code blocks (no external links), for each area:

Embedding inference (tokenization, pooling, normalization).
Cosine similarity (batched) in scikit-learn and PyTorch.
FAISS index build/search (IP & L2; IVF/HNSW variants).
Outlier detectors with IsolationForest, OneClassSVM, KNN distance; show thresholding.
CoT multi-sample runner with majority vote (+ deterministic seeding).
LLM inference recipes: OpenAI API; vLLM for Llama 3 8B; TGI for Mistral 7B.
Parsing utilities: regex for “The answer is …” (A–E / numeric / yes–no) with fallbacks.

A compatibility matrix (Python/CUDA/torch/transformers/faiss bindings; CPU vs GPU).

Reference hyperparameter ranges (from papers/repos) for each detector and index type; justify defaults.

Repro checklist (seed control, dtype, device, tokenizer/version pinning, locale/BLAS determinism).

B.2.3 Core Algorithms Deep Research prompt

Deep Research Brief — Core Algorithms (SCW / CPW • Semantic Filtering • Baselines)

Important: This prompt contains **no URLs**. You must independently discover, verify, and record canonical sources.
Constraint: **Do not** use/return the target paper's official codebase or any direct unofficial re-implementation of that paper. Focus on **adjacent or foundational repos** we can reuse as the **base repo** and stitch SCW/CPW on top.

Objective

Locate and evaluate resources to implement the paper's **core algorithms**:

1. **Semantic Consensus Weighting (SCW)** — sum/aggregate of pairwise cosine similarities over rationale embeddings.
2. **Centroid Proximity Weighting (CPW)** — inverse-distance (or kernel) weighting by proximity to the centroid of rationale embeddings.
3. **Semantic filtering / outlier removal** — three-way comparator: **IsolationForest**, **One-Class SVM**, **KNN-distance** in embedding space.
4. **Self-consistency baselines** — unweighted majority vote; **weighted self-consistency** variants that use SCW/CPW.
5. **N-gram/ROUGE weighting** — to serve as **negative control** / ablation (expected to underperform semantic weighting).
6. **Matrix forms, pseudocode, complexity** — vectorized (NumPy/PyTorch) formulations, big-O, and numerically stable variants (ε-regularization).

Return **≥10 strongly related code repositories** (not the target paper's official or faithful re-impl) and select **one base repo** to build upon.

Deliverables

A) Repository Shortlist (≥10 items; no official or faithful re-impls)

Provide a table:

Rank	Repo Name	Maintainer/Org	Focus Area (why relevant)	License	Lang	Last Commit	Stars	Reuse Targets (files/APIs)	Risks/Limitations

In "Focus Area," explain which part(s) maps to SCW/CPW/semantic filtering/self-consistency. Include **code paths** (e.g., `src/…/similarity.py: pairwise_cosine`, `…/voting.py: weighted_vote`) and a **10–20-line excerpt** of the function signatures or usage snippets (no URLs).

Base repo selection: choose **one** repo as the **base**, justify with:

* Modularity (clean APIs for embeddings/similarity/voting)
* Activity (recent commits, maintenance signals)
* License compatibility (permissive preferred)
* Fit (closest to SCW/CPW with minimal glue code)

B) Algorithms (to extract/confirm)

1) SCW — Semantic Consensus Weighting

* **Embeddings:** rationales → vectors $E \in \mathbb{R}^{K \times D}$, L2-normalized row-wise.
* **Pairwise cosine matrix:** $S = E E^\top \in \mathbb{R}^{K \times K}$.
* **Consensus weights:** $w = S \mathbf{1}$ (optionally set diagonal to 0), then normalize: $\tilde{w} = \frac{w}{\sum_i w_i}$.
* **Weighted vote:** parse each rationale into candidate answer a_i; output $\arg\max_{a} \sum_{i:\,a_i=a} \tilde{w}_i$.

Complexity:

* Embedding: $O(KD)$ (per encoder pass).
* Pairwise cosine: $O(K^2 D)$ (matrix multiply; GPU-friendly).
* Vote: $O(K)$.

Edge cases & stability: clip negatives in S (optional), zero out diagonal, add ε to denominators.

2) CPW — Centroid Proximity Weighting

* **Centroid:** $c = \frac{1}{K}\sum_{i=1}^K E_i$ (optionally L2-normalize).
* **Distances:** $d_i = \|E_i - c\|_2$.
* **Weights:** inverse or kernelized, e.g. $w_i = \frac{1}{d_i + \varepsilon}$ or $w_i \propto \exp(-d_i^2 / (2\sigma^2))$; normalize.
* **Vote:** same as SCW with w_i.

Complexity: centroid $O(KD)$, distances $O(KD)$, vote $O(K)$.

3) Semantic Filtering (three comparators)

* **IsolationForest** (contamination $\in [0.01, 0.1]$, $n\_estimators \in [100,500]$).
* **One-Class SVM** (RBF kernel; $\nu \in [0.01, 0.2]$; $\gamma = 1/D$ or `auto`).
* **KNN distance cutoff** (k $\in [5,50]$, threshold via ROC-Youden on a dev set).
* Produce a **binary mask** $m \in \{0,1\}^K$ and re-normalize weights over kept samples.

4) Baselines

* **Self-consistency (SC)**: unweighted majority vote over parsed answers.
* **Weighted self-consistency (W-SC)**: SCW/CPW weights applied to vote.
* **N-gram/ROUGE weighting**: compute pairwise n-gram overlap or ROUGE-N scores across rationales; derive weights as sums; use as negative/ablation.

C) Pseudocode & Vectorized Forms

Provide **copy-pasteable** pseudocode + vectorized PyTorch/NumPy snippets (no imports from external repos). Include batched cosine, centroid, weights, mask application, and voting. Add comments on precision (fp16 vs fp32), normalization, and ε choices.

D) Complexity & Memory Notes

For typical $K \in [5, 50]$, $D \in [384, 1536]$, quantify: flops, bandwidth, and GPU/CPU viability. Provide scaling strategies (block matmul when K grows, or use ANN for large K).

E) Ablations & Sanity Checks

* **With/without filtering**; **SC vs W-SC**; **SCW vs CPW**; **ROUGE-W (negative)**.
* Report accuracy deltas; include tie-break policy (e.g., frequency then max-weight, then earliest index).

Search Strategy & Queries (no links; use advanced operators)

Exclusion Rules

* Exclude the target paper’s **official repo** and any **explicit unofficial re-implementation** of that exact paper.
* Use **paper title/acronym** negative filters (once identified) in queries, e.g., `-in:name "<paper_acronym>" -in:readme "<paper_exact_title>"`.

1) SCW (pairwise-cosine consensus / weighted voting)

* `("consensus weighting" OR "semantic consensus" OR "cosine consensus") embeddings rationale NOT "

<paper_acronym>" site:github.com in:readme`
* `"pairwise cosine" "weighted vote" embeddings site:github.com in:path`
* `"affinity matrix" "cosine similarity" "majority vote" embeddings site:github.com`

2) CPW (centroid proximity / kernel density style weighting)

* `"centroid distance" weighting embeddings "inverse distance" site:github.com`
* `"rationale embeddings" centroid proximity vote site:github.com in:readme`
* `"kernel density weighting" embeddings vote site:github.com`

3) Semantic Filtering in Embedding Space

* `embeddings outlier filtering (IsolationForest OR "One-Class SVM" OR "KNN") code site:github.com`
* `"semantic filtering" "hallucination removal" embeddings site:github.com in:readme`
* `"pyod" "embedding" outlier detection example site:github.com`

4) Weighted Self-Consistency / CoT Voting

* `"self-consistency" "majority vote" chain-of-thought implementation site:github.com in:readme`
* `"weighted self-consistency" (cosine OR centroid) site:github.com`
* `"answer aggregation" embeddings vote site:github.com`

5) N-gram / ROUGE Weighting (negative/ablation)

* `"ROUGE-N" "rationale overlap" weighting site:github.com in:readme`
* `"n-gram overlap" "weighted vote" rationales site:github.com`

6) Matrix/Vectorized Implementations (NumPy/PyTorch)

* `"pairwise cosine" torch einsum example site:github.com in:readme`
* `"centroid" embeddings numpy vectorized site:github.com`
* `"weighted voting" torch implementation site:github.com in:path`

What to Return for Each Candidate Repo

* **Name, maintainer/org, license, language, last commit, stars.**
* **Why relevant** to SCW/CPW/filtering/self-consistency.
* **Code locations** (files/functions) enabling: pairwise cosine building, centroid computation, weighted voting, outlier detectors, and CoT parsing/voting.
* **Integration sketch** (5–10 lines) showing how to call those functions from our pipeline.
* **Gaps** (what we must implement) and **risks** (stale, no tests, unclear license).

Base Repo Selection Rubric (apply and show scores 1–5)

* **Fit to SCW/CPW**: nearest abstraction (pairwise cosine/centroid + voting hooks).
* **Modularity**: clean APIs, minimal side effects, test coverage.
* **Maintenance**: recent commits, issue responsiveness.
* **License**: permissive and compatible.
* **Performance**: vectorized on CPU/GPU; benchmarks or examples provided.
* **Complexity of adaptation**: LoC changes expected to integrate SCW/CPW + filters.

Code Skeletons (to include in your output; fill shapes/params as you verify)

Vectorized SCW (PyTorch):

```python
# E: [K, D] L2-normalized embeddings
S = E @ E.t()             # [K, K] cosine affinity
S.fill_diagonal_(0)       # optional
```

```
w = S.sum(dim=1)          # [K]
w = (w / (w.sum() + 1e-8)).clamp_min(0.)
```

Vectorized CPW (PyTorch):

```python
c = E.mean(dim=0, keepdim=True)     # [1, D]
d = (E - c).norm(dim=1)             # [K]
w = 1.0 / (d + 1e-6)                # or torch.exp(-d**2/(2*sigma**2))
w = w / (w.sum() + 1e-8)
```

Semantic Filtering Mask (sklearn placeholders):

```python
# Fit on E.cpu().numpy()
# iso_mask, ocsvm_mask, knn_mask -> boolean arrays [K]
# final mask = iso_mask & ocsvm_mask & knn_mask    # or tuned union/intersection
```

Weighted Vote:

```python
# answers: list[str] length K; w: torch.FloatTensor [K]
from collections import defaultdict
acc = defaultdict(float)
for ai, wi in zip(answers, w.tolist()):
    acc[ai] += wi
pred = max(acc.items(), key=lambda x: x[1])[0]
```

N-gram/ROUGE Weighting (negative control):

```python
# Build n-gram sets per rationale; weight_i = sum_j Jaccard(grams_i, grams_j) with i≠j
# Expect weaker correlation with correctness vs semantic weighting.
```

Hyperparameters & Reporting (to collect)

* **K** (#samples), **D** (embed dim), normalization, temperature schedule for CoT sampling.
* IsolationForest (`n_estimators`, `contamination`), One-Class SVM (`nu`, `gamma`), KNN (`k`, threshold rule).
* SCW tweaks (zeroing diagonal, clipping negatives), CPW kernel (σ), vote tie-break rules.
* Complexity summaries and wall-clock on dev machine (CPU/GPU).

Verification Notes (what to document)

* Evidence that the base repo cleanly supports pairwise similarity, centroid ops, and voting hooks.
* License due diligence.
* Small, deterministic unit tests on toy data showing SCW/CPW > ROUGE-W in accuracy on a held-out subset.
* Failure cases (mode collapse when embeddings are near-identical; sensitivity to K and σ).

Minimum Repo Count & Diversity

Return **≥10** repos spanning: similarity toolkits, voting/ensembles, CoT self-consistency runners, embedding utilities, outlier detection frameworks, and lightweight orchestration libs. No official or

faithful re-implementation of the target paper.

Reminder: Provide names, maintainers, licenses, code paths, and snippets — **no URLs in this brief**.

B.2.4 Experiments Deep Research prompt

Deep Research Brief — Experiments (Concise)

Constraints

* **No URLs**[90] in this prompt.
* **Exclude** the paper's official repo and any faithful re-implementation.
* Return adjacent, reusable code/libs that can run our experiments.

Objective
Build everything needed to reproduce experimental results on **AQuA-RAT / SVAMP / StrategyQA**:

* Evaluation tables & runners (accuracy).
* **Stats**: bootstrap CIs, McNemar, permutation test.
* **Temperature mixtures** (multi-T, inverse-T weighting).
* **Ablations**: **SC** (majority vote), **SCW**, **CPW**, **N-gram/ROUGE-W**; three outlier masks (IsolationForest / One-Class SVM / KNN).
* **Model & embedder roster** (LLMs + embeddings) with names, sizes, licenses.

Deliverables

1. **Repo shortlist (≥10 items)** — not the paper's impl. Provide a table:
   `Repo | Maintainer | What it provides | License | Lang | Last Commit | Stars | Entry points (files/APIs) | Integration risk`
   → Pick **one base repo** and justify (modularity, maintenance, license, fit).
2. **Runner plan** (brief bullets): dataset eval → sampling (k, temps) → SC/SCW/CPW/N-gram weighting → optional outlier mask → accuracy + CIs → table emit.
3. **Command templates** (placeholders only): dataset eval, temperature grid, ablations, stats.
4. **Tables**:

   * **Table A**: Accuracy (%) per dataset × method (SC / SCW / CPW / ROUGE-W) with 95% CIs.
   * **Table B**: Outlier methods vs accuracy delta and % kept.
   * **Temp grid**: temps × k.
   * **Model/Embedder roster**.
5. **Verification notes** (short): seeds, tokenizer versions, thresholds, tie-break rules.

Minimal Ablation Matrix

* Methods: SC, SCW, CPW, ROUGE-W
* Filters: None, IF, OCSVM, KNN
* Temps: 3 sets; k ∈ {7, 10, 15}
* Datasets: AQuA-RAT, SVAMP, StrategyQA
* Models: Llama-3-8B, Mistral-7B, Llama-2-7B, GPT-3.5, GPT-4o-mini
* Embedders: SciBERT, RoBERTa-base

High-Precision Search Queries (no links)

* **Harness/eval:** `site:github.com in:readme "self-consistency" ("accuracy" OR "evaluation") stars:>100 -in:name:<paper-acronym>`
* **Bootstrap/CIs:** `bootstrap confidence interval accuracy classification code site:github.com in:readme`
* **Paired tests:** `"McNemar test" classification code site:github.com` ; `"permutation test" accuracy paired site:github.com`
* **Temperature mixtures:** `"temperature sweep" "self-consistency" k=10 site:github.com` ; `"inverse temperature weighting" filetype:pdf`
* **Outliers:** `embeddings outlier filtering IsolationForest OneClassSVM KNN code site:github.com in:readme` ; `pyod embeddings example site:github.com`
* **N-gram/ROUGE (negative):** `ROUGE-N rationale overlap weighting site:github.com in:readme`
* **Model roster:** `"Llama 3 8B" "Mistral 7B" "Llama 2 7B" "GPT-3.5" "GPT-4o mini" evaluation` ; `"SciBERT" "RoBERTa base" embeddings reasoning datasets`

Reporting Requirements (one-liners)

* Quote **model names/sizes/licenses/context lengths** verbatim.
* Record **seeds, dtype, device**, rate-limit handling.

* For detectors: ranges tried & chosen thresholds.
* For temps: sets + weights (α in inverse-T).
* Use **paired** tests for all method comparisons.

B.3 Coding Agent Prompt

Based on the deep research recommendations, we used /habedi/cogitator as our base, so all coding prompts will use /habedi/cogitator as the foundational code reproduction framework.

Below is the final reproduction scaffold, all paths are relative to
./semssc/cogitator

96

1. Directory Tree and File List

```markdown
./semssc/cogitator
├── cogitator
│   ├── __init__.py
│   ├── clustering.py
│   ├── data_loader.py            # Data loading and preprocessing module
│   ├── embedding.py              # Embedding model wrapper module
│   ├── model                   # (Reserved for future expansion)
│   ├── outlier_detection.py      # Outlier detection module wrapper (IsolationForest/One-
ClassSVM/KNN)
│   ├── schemas.py              # Data structure and schema definitions
│   ├── strategies
│   │   ├── __init__.py
│   │   ├── auto_cot.py         # Other generation strategy examples
│   │   ├── cdw_cot.py          # Other generation strategy examples
│   │   ├── graph_of_thoughts.py# Other generation strategy examples
│   │   ├── least_to_most.py    # Other generation strategy examples
│   │   ├── sc_cot.py           # Core algorithm module: implement SCW and CPW weighting and outlier
filtering
│   │   ├── tree_of_thoughts.py # Other generation strategy examples
│   │   └── tree_of_thoughts_with_trace.py  # Other generation strategy examples
│   └── utils.py                # Utility helper functions
├── config
│   └── semssc_config.yaml      # Global configuration file (sampling count, temperature sets,
embedding model, outlier detection parameters, etc.)
├── benches
│   ├── README.md               # Experiment script description and usage methods
│   ├── evaluate.py             # (Optional) existing evaluation script
│   └── evaluate_semssc.py      # End-to-end experiment pipeline: load data, generate CoT, call core
algorithms, evaluate accuracy
├── docs
│   ├── replication_plan.md     # Reproduction project overall planning and module responsibility
descriptions
│   ├── high_hit_search_queries.md  # High-hit search query string list (data/supporting
components/core algorithms/experiment components)
│   └── reproduction_report.md  # Final reproduction report (experiment design, result comparison,
ablation experiment description)
├── tests
│   ├── conftest.py             # pytest configuration file
│   ├── extra_tests             # Other extended tests
│   ├── test_data_loader.py     # Data loading module unit tests
│   ├── test_embedding.py       # Embedding module unit tests
│   ├── test_outlier_detection.py # Outlier detection module unit tests
│   └── test_sc_algorithm.py    # Core algorithm (SCW/CPW) module unit tests
├── Makefile                  # Compilation, testing, running, documentation building command set
└── README.md                 # Project overview, installation and running guide
```

---

2. Responsibility Description and Main Function Signatures for Each Key File

2.1 cogitator/data_loader.py
- **Responsibility**: Responsible for uniformly loading dataset files (JSONL/CSV), preprocessing
data, and extracting final answers from generated chain-of-thought text.
- **Main Functions**:
  - `class DataLoader:`

    - `def __init__(self, dataset_name: str, file_path: str) -> None:`
    - `def load_data(self) -> list:`
    - `@staticmethod def extract_final_answer(cot_output: str) -> Optional[str]:`

2.2 cogitator/embedding.py
- **Responsibility**: Encapsulate calls to pre-trained models on Huggingface (such as SciBERT, RoBERTa), vectorize input text, and return embedding vectors.
- **Main Functions**:
  - `class EmbeddingModel:`
    - `def __init__(self, model_name: str = "allenai/scibert_scivocab_uncased") -> None:`
    - `def embed(self, text: str) -> np.ndarray:`

2.3 cogitator/outlier_detection.py
- **Responsibility**: Uniformly encapsulate outlier detection methods (such as IsolationForest, OneClassSVM, KNN) for filtering abnormal embedding samples.
- **Main Functions**:
  - `class OutlierDetector:`
    - `def __init__(self, method: str = "isolation_forest", **kwargs) -> None:`
    - `def fit(self, X: np.ndarray) -> None:`
    - `def predict(self, X: np.ndarray) -> Union[np.ndarray, list]:`

2.4 cogitator/strategies/sc_cot.py
- **Responsibility**: Core algorithm implementation file, extending from standard self-consistency, making weighted decisions on multi-sample generation results through two weighting strategies:
  - **SCW (Semantic Consensus Weighting)**: Weighted voting based on the sum of cosine similarities of each output embedding
  - **CPW (Centroid Proximity Weighting)**: Weighted voting based on the inverse of centroid distance of all sample embeddings
  Also supports calling outlier detection interfaces to filter abnormal samples.
- **Main Functions**:
  - `def semantic_consensus_weighting(embeddings: np.ndarray, answers: List[str]) -> Tuple[str, dict]:`
  - `def centroid_proximity_weighting(embeddings: np.ndarray, answers: List[str]) -> Tuple[str, dict]:`
  - `def filter_outliers(embeddings: np.ndarray, answers: List[str], method: str, params: dict) -> Tuple[np.ndarray, List[str]]:`
  - `def run_semssc(cot_outputs: List[str], method: str = "scw") -> Tuple[str, dict]:`

2.5 benches/evaluate_semssc.py
- **Responsibility**: End-to-end experiment pipeline script:
  - Load data from specified datasets
  - Generate or simulate multiple chain-of-thought (CoT) outputs
  - Call `run_semssc` to get weighted decisions and detailed voting data
  - Calculate and output evaluation metrics (e.g., accuracy)
- **Main Functions**:
  - `def load_ground_truth(dataset_name: str, data_file: str) -> dict:`
  - `def evaluate_semssc(dataset_name: str, data_file: str, weighting_method: str = "scw") -> None:`

2.6 config/semssc_config.yaml
- **Responsibility**: Configuration file that uniformly manages key project hyperparameters, such as sampling count, temperature sets, top_k, top_p, max_new_tokens, embedding model names, and outlier detection method parameters.
- **Configuration example see below**

2.7 Documentation Files in docs/ Directory
- **replication_plan.md**: Records overall planning of reproduction project, module design, interface descriptions, and experimental process design.
- **high_hit_search_queries.md**: Lists advanced search query strings by module for retrieving data, code, paper principles, and other implementation details.
- **reproduction_report.md**: Final reproduction report containing experimental design, accuracy for each dataset (AQuA-RAT, SVAMP, StrategyQA), ablation experiment comparisons, and discussions.

2.8 Test Files in tests/ Directory
- **test_data_loader.py**, **test_embedding.py**, **test_outlier_detection.py**, **test_sc_algorithm.py**
  - Conduct unit tests for data loading, embedding, outlier detection, and core algorithm module functions respectively, ensuring module inputs and outputs meet expectations.

2.9 Makefile
- **Responsibility**: Provides unified command entry points, including dependency installation, testing, code format checking, documentation building, and experiment pipeline execution.
- **Main Targets** (examples):
  - `setup`: Install system dependencies and Poetry
  - `install`: Install Python dependencies
  - `test`: Run all unit tests
  - `format`: Format code
  - `docs`: Generate project documentation
  - `experiment`: Run evaluation scripts (e.g., benches/evaluate_semssc.py)

2.10 README.md
- **Responsibility**: Overall documentation including project introduction, dependency installation, running methods, debugging instructions, and feedback channels.

---

3. Configuration File Example (YAML)

File path: config/semssc_config.yaml

```yaml
sample_k: 10
temperature_set:
  - 0.7
  - 0.8
  - 0.9
top_k: 50
top_p: 1.0
max_new_tokens: 100
embedding_model: "allenai/scibert_scivocab_uncased"
outlier_detection:
  method: "isolation_forest"
  params:
    contamination: 0.1
    random_state: 42
```

---

4. Running/Evaluation Commands and Dependency Installation

4.1 Dependency Installation
- Manage dependencies through Poetry, ensure Python3 is installed.

```bash
cd ./semssc/cogitator
poetry install --all-extras
# Or use Makefile:
make install
```

4.2 Run Unit Tests

```bash
cd ./semssc/cogitator
poetry run pytest
# Or use Makefile:
make test
```

4.3 Run Experiment Evaluation (using AQuA-RAT dataset as example)

```bash
cd ./semssc/cogitator
poetry run python benches/evaluate_semssc.py
# Or use Makefile:
make experiment
```

4.4 Generate Documentation (if using mkdocs to build documentation)

```bash
poetry run mkdocs build
# Or use Makefile:
make docs
```

---

5. Reproduction Metrics

According to paper requirements, reproduction metrics include:
- **AQuA-RAT Dataset**: Accuracy target approximately near paper reported values (example target accuracy between 70%-80%, specific needs to refer to paper results)
- **SVAMP Dataset**: Accuracy target (e.g., 65%-75%, specific to be verified by experiments)
- **StrategyQA Dataset**: Accuracy target (e.g., 60%-70%, specific to be verified by experiments)

(Note: Above accuracy targets need to be adjusted based on actual experiments, final results should be as close as possible to experimental metrics reported in the paper)

---

6. Change Constraints: Add/Modify/Delete Checklist

6.1 New Files
- `cogitator/data_loader.py` → Add data loading and preprocessing module
- `cogitator/embedding.py` → Add embedding model wrapper module
- `cogitator/outlier_detection.py` → Add outlier detection module wrapper
- `cogitator/strategies/sc_cot.py` → Add or extend implementation of core algorithms (SCW/CPW weighting logic and outlier filtering)
- `config/semssc_config.yaml` → Add global configuration file
- `benches/evaluate_semssc.py` → Add end-to-end experiment evaluation script
- `docs/reproduction_report.md` → Add final reproduction report file
- Unit test files: `tests/test_embedding.py`, `tests/test_outlier_detection.py`, `tests/test_sc_algorithm.py` → Add corresponding test modules
- `Makefile` → Modified to include experiment running, testing, documentation building commands (if original project did not contain complete commands, now supplemented)

6.2 Modified Files
- `docs/replication_plan.md`
  → Add detailed descriptions of supporting components, core algorithms, and experiment components (original file content supplemented and updated)
- `docs/high_hit_search_queries.md`
  → Add advanced query strings related to supporting components and core algorithms for convenient research and comparison
- `README.md`
  → Update project overview, installation guide, running and debugging instructions
- (If existing) In original generator strategy files (e.g., auto_cot.py, etc.) if there are content extensions, can note in comments that this round actually did not modify their core logic, keeping them as-is.

6.3 Deleted Files
- No need to delete old files, all new functions are extended on existing structure; if there are redundant test or documentation files, they can be organized and archived in subsequent versions (no deletion operations in this version).

The above is the final reproduction scaffold's project directory structure, key module responsibilities and interfaces, configuration file examples, running commands, and reproduction metric descriptions. Please deploy, debug, and verify experiments according to the above checklist.

_____

【Task Overview and Strong Constraints】

Objective: Reproduce the "Semantic Self-Consistency" framework proposed in the paper "Semantic Self-Consistency: Enhancing Language Model Reasoning via Semantic Weighting", adding semantic weighting (SCW: cosine similarity consensus weighting, CPW: centroid proximity weighting) and outlier detection modules (Isolation Forest / KNN / One-Class SVM) on top of multi-sample CoT self-consistency (majority vote). This reproduction needs to cover five major modules: data components, supporting components, config configuration, core algorithms, and experimental components, ensuring that reproduction results align with target.json metrics (including datasets AQuA-RAT, SVAMP, StrategyQA, corresponding accuracy rates, weighting strategy parameters: k=10, temperature sets, top-k=50/top-p=1, max_new_tokens parameters, and adjustable outlier detection methods).

Strong Constraints:
1. All code modifications are performed on base (./semssc/cogitator) with "add/delete/modify" operations, all paths are relative to base;
2. Must read the following file list (absolute paths) and determine design approach and structure based on content:
   - breakdown.txt (5 major modules and strategy breakdown)
   - camel_dialogue.log (dialogue and thought trajectory reference)
   - cogitator_analysis.md (detailed analysis of base codebase)
   - core_algorithm.docx (supplementary documentation)
   - data.docx, data_summary.md (data documentation)
   - experiment.docx (supplementary documentation)
   - mineru_output_final/full.md and mining output including images/**, layout.json, paper_digest.md (original paper, image resources, other information)
   - scaffold.txt (reproduction framework and directory structure requirements)
   - se_agent_prompt.md (this generation prompt for reference)
   - support.docx (supplementary documentation)
   - target.json (reproduction targets and metric constraints)
3. All operations strictly based on base, no parallel projects should be created.

_____

【Phased Execution Plan】

① 【Phase 1 – Read and Align with Base】
   - Read existing code in the base directory (README.md, pyproject.toml, Makefile, original cogitator module files, etc.), while opening scaffold.txt, cogitator_analysis.md and breakdown.txt to check framework and module responsibilities.
   - Output requirements: Print all key file paths read (e.g., ./semssc/cogitator/README.md, scaffold.txt, etc.), and generate "modification summary" information for each file (e.g., "confirmed

base directory structure, existing modules and configuration files, test command summary").

②【Phase 2 – Generate Final Project Framework Based on Scaffold + Paper】
- Design final directory structure, interface definitions, configuration file locations (e.g., add config/semssc_config.yaml) and commands (Makefile update experiment, test, documentation commands) based on scaffold.txt guidelines and paper key points (semantic self-consistency, SCW/CPW weighting, outlier detection steps).
- Output requirements: List final directory tree (including new files under cogitator: data_loader.py, embedding.py, outlier_detection.py, strategies/sc_cot.py, new config/semssc_config.yaml, new benches/evaluate_semssc.py, and corresponding unit test files under tests), and provide detailed "add/modify/delete" list (each item marked with modification points, e.g., "add data preprocessing module data_loader.py, interface includes __init__(dataset_name, file_path), load_data(), extract_final_answer(cot_output)").

③【Phase 3 – Implement 5 Major Modules Split by Breakdown】
A. Data Components (Datasets)
- New file: cogitator/data_loader.py
- Core objective: Uniformly load datasets (such as JSONL/CSV), preprocess data, and extract final answers from CoT text.
- Minimum runnable test: Write unit tests in tests/test_data_loader.py (test load_data() returns list, extract_final_answer() can match strings).
- Output requirements: Print file paths involved in modifications, interface descriptions, test commands (e.g., poetry run pytest tests/test_data_loader.py) and test result summary.

B. Supporting Components
- New files: cogitator/embedding.py (encapsulate SciBERT, RoBERTa and other model calls, implement embed(text) returning embedding vectors)
cogitator/outlier_detection.py (encapsulate IsolationForest, One-Class SVM, KNN interfaces, implement fit() and predict())
- Minimum runnable test: Write unit tests in tests/test_embedding.py, tests/test_outlier_detection.py to verify embed() return type and outlier detection works properly.
- Output requirements: Print read and modified file lists, interface descriptions, unit test commands (e.g., make test) and test log summary.

C. Config Components (Configs)
- New configuration file: config/semssc_config.yaml (see sample content, including sample_k, temperature_set, top_k, top_p, max_new_tokens, embedding_model, outlier_detection configurations)
- Output requirements: Confirm configuration file path, show configuration file content sample location, and explain each parameter purpose; also confirm call path in Makefile.

D. Core Algorithms Components
- New/extended file: cogitator/strategies/sc_cot.py
  • Implement semantic consensus weighting function semantic_consensus_weighting(embeddings, answers);
  • Implement centroid proximity weighting function centroid_proximity_weighting(embeddings, answers);
  • Implement filter_outliers(embeddings, answers, method, params);
  • Implement run_semssc(cot_outputs, method="scw"), using the above functions to complete overall weighted decision-making.
- Minimum runnable test: Conduct unit tests for SCW, CPW and outlier filtering functions separately in tests/test_sc_algorithm.py, print input and output examples for each method.
- Output requirements: Print modified file paths, detailed descriptions of each function interface, test commands (e.g., poetry run pytest tests/test_sc_algorithm.py) and test result summary.

E. Experiment Components (Experiments)
- New file: benches/evaluate_semssc.py
  • Implement end-to-end experiment pipeline: from loading data (calling data_loader), generating multiple CoT outputs (can simulate or call existing generator strategies), using run_semssc for weighted decision-making, and calculating accuracy.
  • Support evaluation metrics aligned with target.json (datasets AQuA-RAT, SVAMP, StrategyQA).
- Minimum runnable test: After running the experiment script from command line (e.g., poetry run python benches/evaluate_semssc.py), record accuracy for each dataset and save results to benchmark_results.jsonl (or specified output file).

- Output requirements: Print entire evaluation script run log, evaluation metric summary for each dataset, and any exception information.

④【Phase 4 – Overall Integration and Evaluation】
- After completing unit tests for each module, run complete Makefile commands for overall testing (e.g., make test, make experiment, make docs).
- Output requirements: During integration, must print all test commands, log summaries, and documentation build notifications; also save experiment results (including accuracy for each dataset, specific values aligned with target.json descriptions) to specified log files in final evaluation and print summary results.

【Output Checklist Template (Add/Modify/Delete)】

【New Files】
- cogitator/data_loader.py
- cogitator/embedding.py
- cogitator/outlier_detection.py
- cogitator/strategies/sc_cot.py
- config/semssc_config.yaml
- benches/evaluate_semssc.py
- tests/test_data_loader.py
- tests/test_embedding.py
- tests/test_outlier_detection.py
- tests/test_sc_algorithm.py
- docs/reproduction_report.md (new final reproduction report document)

【Modified Files】
- docs/replication_plan.md → Add detailed responsibility descriptions and interface design for each module
- docs/high_hit_search_queries.md → Add high-hit query strings for deep search
- README.md → Update project overview, installation and running guidance
- Makefile → Add experiment running (experiment) and documentation building (docs) commands
- (If existing strategy files like auto_cot.py retain original core without substantial changes, only add sc_cot.py file)

【Deleted Files】
- Current version does not need to delete files; if redundant content exists, organize and archive later.

【Configuration Sample Location】

Please ensure the configuration file is stored at config/semssc_config.yaml, with content referencing the following sample:

```
sample_k: 10
temperature_set:
  - 0.7
  - 0.8
  - 0.9
top_k: 50
top_p: 1.0
max_new_tokens: 100
embedding_model: "allenai/scibert_scivocab_uncased"
outlier_detection:
  method: "isolation_forest"
  params:
    contamination: 0.1
    random_state: 42
```

【Evaluation Commands and Log/Result Saving Rules】

- Execute in sequence: dependency installation (poetry install --all-extras or make install), unit testing (make test), evaluation experiments (make experiment or poetry run python benches/evaluate_semssc.py), documentation building (make docs).
- Each test run must print:
  - File paths read (paths of key files).
  - Modification summary (new, modified files and their interfaces).
  - Test commands (e.g., poetry run pytest tests/xxx.py) and command execution result summary (e.g., number of tests passed, coverage).
  - After saving experiment results to specified output file (e.g., benchmark_results.jsonl), print evaluation summary information, dataset accuracy and other metrics.

- Save evaluation logs and result files in corresponding folders under base directory (e.g., recorded in benches/benchmark_results.jsonl).

【Align target.json Metrics and Adjustable Parameter Locations】

- Adjustable parameter areas: config/semssc_config.yaml includes k_samples (sample_k), temperature_set, top_k, top_p, max_new_tokens; additionally, optional outlier detection methods in core algorithms (isolation_forest, knn, one_class_svm).
- Model and dataset alignment with target.json metrics:
  - Datasets: AQuA-RAT, SVAMP, StrategyQA, accuracy targets refer to target.json descriptions (e.g., AQuA-RAT improvement about +3.14%, SVAMP +0.97%, StrategyQA where SCW can improve accuracy).
  - Model or encoder selection: Support GPT-3.5, Llama 2 7B, Llama 3 8B, Mistral 7B, GPT-4o mini, and SciBERT, RoBERTa.
- Output requirements: Each execution output must print current hyperparameters used (including k, temperatures, max_new_tokens, outlier detection methods and their parameters) and corresponding experimental comparison data.

【Required Print Statement for Each Round Output】
During each module implementation and overall integration testing, the following must be printed in logs:
1. File paths read (show specific absolute or relative paths, e.g., ./semssc/cogitator/xxx.md).
2. Summary of current changes (list specific content and interface overview of "new files" and "modified files").
3. Test commands executed (e.g., make test, poetry run pytest tests/xxx.py, make experiment).
4. Test and evaluation result summary, ensuring alignment with target.json metrics.

Please strictly follow the above prompt steps and, after completing each phase, summarize and print all above items (file paths, change summary, test commands and result overview) for subsequent manual review and automatic evaluation. All content must be tightly coupled to base and cannot deviate from the base directory structure, ensuring the reproduction project aligns with the original code structure.

Please start execution and output detailed log information at each stage, thank you!

B.4 Searching Agent Prompt

You are **Searching Agent** for the Deep-Reproducer system.

Identity & Scope
- Your ONLY job is to retrieve and rank entries **from the task-scoped Memory Bank** and return compact, actionable pointers for the Coding Agent.
- You MUST NOT invent facts, visit the open web, or paste full code. Return pointers, short notes, and minimal context only.

Tools (abstract)
- `mb.search(query, filters) -> [entry]`
- `mb.get(id) -> entry`
Each `entry` may include: {id, type ∈ {dataset, code, paper, debug}, title, url_or_path, summary, tags, version/commit, license, timestamp, meta, blacklist_flag}.

Inputs (will be provided in the user message JSON)
- `task_id`: string
- `query_text`: short natural language information need
- `top_k`: integer (default 5)
- `required_types`: optional subset of {dataset, code, paper, debug}
- `mode`: one of {"library_only", "no_external_repo", "free"}; default "library_only"
- `blacklist`: list of blocked url/path patterns (exact or regex)
- `constraints`: {must_include: [kw], must_exclude: [kw], prefer_official: bool}

Retrieval & Ranking
1) Normalize & expand the query:
   - Expand common synonyms/aliases and abbreviations (e.g., "Isolation Forest" ⇔ "iForest"; "OC-SVM" ⇔ "One-Class SVM").
   - Include method, dataset, metric, file/function names if present.
2) Filter:
   - Enforce `required_types` and `mode`.
   - EXCLUDE any entry with `blacklist_flag` or url/path matching `blacklist`.
3) Score each candidate with:
   - Relevance (keyword & field match, title>tags>summary>meta) 0.55
   - Type match to `required_types` 0.15
   - License safety (permissive > unknown > restrictive) 0.10
   - Recency (newer preferred if not contradictory) 0.10
   - Officialness (docs/org sites, original dataset portal) 0.10
4) Deterministic tie-break: prefer (license safer) → (official) → (newer) → (alphabetical url).
5) De-duplicate by canonicalized url/domain and by near-duplicate titles.

Output (STRICT JSON)
Return exactly this schema:
```
{
  "task_id": "...",
  "query": "...",
  "top_k": N,
  "hits": [
    {
      "id": "...",
      "type": "dataset|code|paper|debug",
      "title": "...",
      "url_or_path": "...",
      "version_or_commit": "...",
      "license": "...",
      "why_selected": "≤25 words, concrete reason",
      "usage_hint": "≤25 words, how Coding Agent should use it",
      "risk_flags": ["license|staleness|partial|spec_mismatch|none"]
    }
  ],
  "usage_notes": [
    "Concise cross-hit synthesis, ≤25 words each (e.g., ordering, compatibility, pin versions)."
  ],
  "caveats": [
    "Known gaps or conflicts, ≤20 words each."
```

```
  ],
  "next_actions_for_coder": [
    "Imperative steps, 1 line each, ≤20 words, no code."
  ]
}
```

Style & Safety
- Be concise, factual, and **action-oriented**.
- Do NOT include code blocks, pseudocode, or long prose.
- Do NOT cite or quote large text; summaries ≤25 words.
- Never recommend blacklisted or target-paper repos.
- If zero valid hits: return empty `hits` plus `next_actions_for_coder` with specific fallback (e.g., "request Deep-Research for official dataset portal").

Return ONLY the JSON. No extra text.

References

[1] Anthropic. Introducing the model context protocol (mcp). https://www.anthropic.com/news/model-context-protocol, 2024. Accessed: 2025-08-25.

[2] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024. doi: 10.48550/arXiv.2403.17134. URL https://arxiv.org/abs/2403.17134.

[3] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. doi: 10.48550/arXiv.2303.17760. URL https://arxiv.org/abs/2303.17760. arXiv:2303.17760.

[4] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/.

[5] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, pages 1592–1604, Vienna, Austria, 2024. ACM. doi: 10.1145/3650212.3680384.