AI4SE Assignment 1 — If-Condition Prediction

Student: Yijia Shi
Course: AI for Software Engineering
Date: Oct 2025

Abstract

This report describes a reproducible pipeline to build two datasets required to train a Transformer that completes masked `if` conditions within Python functions. The pipeline pulls Python code from GitHub, parses and cleans it, removes semantic duplicates, and produces (i) a large, general pre-training corpus augmented to emphasize `if` structures and (ii) a task-specific fine-tuning set where one `if` condition per function is masked and becomes the target. The design strictly trains a custom tokenizer first and uses it consistently across pre-training and fine-tuning, per assignment rules. We document selection criteria, quality gates, augmentation strategies, data splits, and anti-leakage controls, plus deliverable formats and scripts to allow end-to-end replication.

---

0. Data source and repository selection

Goal. Build a high-quality, diverse, and license-safe pool of Python repositories and extract function-level code for downstream datasets.

0.1 Repository discovery & selection criteria

- **Source.** GitHub repositories with substantial Python content. *SEART GHS* is used interactively to shortlist candidates by stars/activity; automated harvesting relies on the GitHub API.
- **Query (example, adjustable):** `language:Python stars:>20 forks:false archived:false` and **recent activity** within the last 12–18 months. Exclude archived or mirror repos.
- **Minimum Python footprint.** ≥ **30** `.py` files **and** ≥ **1,000** Python LOC; Python should account for **≥60%** of repo LOC (estimated via `cloc`).
- **License compatibility.** Accept **MIT, BSD-2/3-Clause, Apache-2.0, MPL-2.0**. Exclude unknown or restrictive licenses; record SPDX id. Store `LICENSE` content for traceability.
- **Code quality heuristics.** Prefer repos with: (a) a tests folder or files named `test_*.py`/`*_test.py`; (b) CI config present; (c) linter config (`ruff.toml`, `.flake8`, `pyproject.toml`). Optionally compute quick metrics (sampled files): **radon** cyclomatic

complexity histogram, **ruff/flake8** error density (< 0.5 issues / 100 LOC threshold), and **docstring coverage** (fraction of functions with docstrings > 10%).
- **Domain diversity.** Bucket repos by topics (e.g., *web*, *data*, *ML*, *CLI/tools*) using GitHub topics/readme heuristics; stratified sampling ensures coverage across domains.

0.2 Repository extraction ("how we fetch repos")

1. **Manifest build (`discover_repos.py`).** Use the search query to fetch candidates and write a manifest `data/manifests/repos_v3.csv` with columns: `{repo_url, default_branch, commit_sha, license_spdx, stars, last_push, topics, py_file_count, py_loc, python_ratio, bucket}`.
2. **Snapshot (`fetch_latest_snapshot.py`).** For each accepted repo, clone **latest default-branch commit** with `--depth 1` and record the resolved `commit_sha`. If available, download the repository archive to speed up.
3. **Language/size check.** Run `cloc` (or a light LOC counter) on checkout; re-enforce thresholds and drop outliers (e.g., >200k Python LOC, or Python ratio <60%).
4. **License verification.** Parse `LICENSE` to confirm SPDX id matches allowlist; exclude if ambiguous.
5. **Path exclusions.** Skip vendor/third-party directories (`vendor/`, `site-packages/`, `third_party/`, `examples/large_data/`) and generated artifacts.
6. **Provenance.** Store per-repo metadata in `data/provenance/{owner}__{name}__{sha}.json` and keep a **global manifest** to enable repo-level splits later.

0.3 Function extraction process

- **Method.** Walk the repo for `.py` files; for each file, use `ast.parse` (Python 3). Extract every `ast.FunctionDef`/`ast.AsyncFunctionDef` with source slices.
- **Extraction criteria.**
    - **Syntax validity.** Must be `ast.parse`-able (files and extracted functions).
    - **Minimum complexity.** Discard trivial bodies (only `pass`, only a constant return) and functions with < **5 lines**.
    - **Maximum length.** Discard functions with > **4,000 characters** to control sequence length.
    - **Optional tests filter.** If desired, exclude functions under `tests/` to avoid assertion-heavy patterns.

- **Output.** Append to `data/functions_v2.jsonl` with `{repo, path, sha, func_src}`.

0.4 Data quality assurance on the raw corpus

- **AST-based deduplication.** Canonicalize ASTs (strip comments/whitespace, literal placeholders, alpha-rename identifiers) → compute a fingerprint → keep the first

occurrence globally.
Filtering recap. Remove too short/too long/malformed items; drop non-parsable or anomalous encodings; optional removal of vendored/generated files.

- **Validation.** Track **parsability rate** and random-sample manual reviews. Keep counters/metrics in a build log.

After this stage, the pipeline proceeds with §4 (augmentation) and §4.3 (fine-tuning set construction) unchanged.

---

1. Problem framing and data plan

Task. Given a Python function that contains a special token `<IFMASK>` in place of exactly one `if` condition, the model must generate the missing condition.

Two datasets.

- **Pre-training corpus (≥150k instances):** broad Python code distribution to learn syntax, identifiers, and control-flow patterns via **causal language modeling (CLM)**; lightly biased toward `if` statements through targeted augmentation (§4.2).
- **Fine-tuning dataset (≥50k instances):** examples purpose-built for the `if`-condition prediction task. Each input contains `<IFMASK>`; each label is the masked condition string.

Tokenizer constraint. Train a **custom byte-level BPE tokenizer** first and use it everywhere (no pre-made Hugging Face tokenizers). All special tokens are included at tokenizer-training time to guarantee stable segmentation.

---

2. Sources and collection

- **Scope.** Only Python repositories from GitHub. Use *SEART GHS* (https://seart-ghs.si.usi.ch) to shortlist repos by primary language, stars, recent activity, and size. DataHub is **not** used (per assignment).
- **Snapshot.** Clone the repo and check out the **latest default-branch commit**; record `{repo_url, commit_sha, path}` for traceability.
- **Extraction unit.** Functions from `.py` files. Parse with Python's `ast` to walk `ast.FunctionDef` / `ast.AsyncFunctionDef`, capturing function source spans and metadata.
- **Exclusions.** Drop vendor/third-party directories, generated code, mega files (>200 KB), and non-parsing files. Only Python-3-parsable files are kept.

Raw function store. A JSONL file `data/functions_v2.jsonl` with one record per function: `{repo, path, sha, func_src}`.

---

3. Parsing, cleaning, and quality gates

3.1 Structural parse as the first gate

If `ast.parse` fails, the function is discarded. For valid functions we keep **verbatim source** (including comments) for language modeling and build a **normalized AST** for analysis/deduplication.

3.2 Filtering criteria

- **Length bounds.** `min_lines = 5` (very short functions add noise), `max_chars = 4000` (controls sequence length and GPU memory).
- **Trivial or noisy bodies.** Remove functions that are essentially stubs (e.g., only `pass`, `return NotImplemented`, or `raise NotImplementedError`). Remove cases with >80% docstring/comments or anomalous non-printable characters.
- **`if` presence policy.**
    - Pre-training: keep all valid functions (including those **without** `if`) to retain natural distribution.
    - Fine-tuning: require at least one `if` statement.

3.3 Semantic deduplication (AST-level)

Goal: reduce memorization and cross-repo clones.

1. **Canonicalization.** Remove comments/whitespace; normalize literals (strings → stable hash token, numbers → placeholder), alpha-rename variables/params to `VAR_i`/`ARG_i`, and sort order-insensitive literal sets/maps where safe.
2. **Fingerprint.** Compute `SHA1(ast.dump(canonical_ast, include_attributes=False))`.
3. **De-duplicate.** Retain only the first occurrence per fingerprint globally (across all repos/paths).

This procedure eliminates near-identical functions while preserving diverse control-flow shapes.

---

4. Augmentation and corpus formation

4.1 Special tokens (inserted during tokenizer training)

<CODE>, </CODE> delimit code blocks; <IFMASK> marks a hidden condition; <ANS> prefixes targets in certain formats; <TASK=IF_COND> can steer the model during supervised examples. These tokens are **part of the tokenizer vocabulary** from the outset.

4.2 Pre-training corpus (CLM)

- **Wrapping.** Each function becomes a block:
  `\n<CODE>\n{function_source}\n</CODE>\n`.
- **Targeted augmentation (8% of `if`-bearing functions).** With probability 0.08 on functions that contain `if`:
  1. **Mask mode:** replace exactly one condition with `<IFMASK>` *in situ*; or
  2. **Answer mode:** append a line `\n<ANS> {condition}` after the function.
- **Rationale.** Keeps overall distribution realistic while gently increasing exposure to `if` semantics before task supervision.
- **Output.** Single text file `data/pretrain_corpus_v3.txt` (example scale: ~165k code blocks).

4.3 Fine-tuning set (supervised)

- **`if` detection.** Use a conservative regex to find candidate headers:
  `r"(?ms)^\s*if\s+(.+?):\s*(?:#.*)?$"`
  We ignore `elif` during extraction and treat it as an `if` with a preceding branch when masked.
- **Single-mask policy.** If multiple `if` are present, uniformly sample one eligible `if` per function.
- **Masking & targets.** Replace the chosen condition with `<IFMASK>` in the header, capture the original condition as the **label**. Strip trailing colons, inline comments, and normalize whitespace.
- **AST validity check.** Run the masked function through a small repair pass to ensure the remaining code is parsable (e.g., maintain indentation, leave the body untouched). A post-processing script (`prepare_eval_inputs.py`) rejects items that become unparsable.
- **Windowing for long contexts.** When the function exceeds model length, **left-truncate** outside the `if` region while preserving the `<IFMASK>` line and (for formats that include it) a terminal `<ANS>` marker.
- **Splits & counts.** Repository-level split to avoid leakage: `train: 72k`, `val: 9k`, `test: 9k` (total ~90k). Blackboard's additional test set is held out entirely.

Output. JSONL files `data/finetune_v3_{train,val,test}_prepped.jsonl` with fields:

{"id": "...", "input": "<CODE>...<IFMASK>...</CODE>", "expected_condition": "...", "repo": "...", "path": "...", "sha": "..."}

---

5. Handling edge cases

- **Short functions.** We exclude functions with <5 lines from both datasets to reduce degenerate pattern learning (many are trivial getters/setters where `if` is rare or stylistically uninformative).
- **Functions without `if`.** Useful for language modeling; retained in pre-training but **not** in fine-tuning (no label to predict).

- **Compound conditions.** Preserve the exact logical form (e.g., `a and (b or c)`), but remove trailing colon and trailing comments from the label. No normalization (like De Morgan) is applied to avoid altering semantics.
- **Inline comments.** Stripped from labels; kept in inputs, because models often learn comment-code correlations.
- **`elif`/`else`.** Only `if` headers are candidates for masking. `elif` is treated as `if <cond>` at its line for masking purposes; `else` is ignored.

---

6. Anti-leakage and split hygiene

- **Repo-level splitting.** All functions from the same repository go to the **same** split.
- **Fingerprint hold-out.** After the repo split, re-check fingerprints across splits and drop any duplicates to prevent near-duplicate leakage.
- **Pre-training vs fine-tuning contamination.** The fine-tuning *test/val* repos are excluded from pre-training where feasible. When complete exclusion is impractical, we rely on fingerprint filtering to remove overlapping functions.
- **Prompt-label isolation.** Inputs never contain the answer explicitly (e.g., `<ANS>` lines are **not** present at inference-time inputs).

---

7. Tokenizer training (required for masking consistency)

- **Type.** Byte-level BPE (GPT-2-style), `add_prefix_space=True`, `lowercase=False` to preserve case and punctuation in code.
- **Vocab size.** 50,257 (room for identifiers and the special tokens).
- **Corpus.** Train on the pre-training text (`pretrain_corpus_v3.txt`) **including** the special tokens so that `<CODE>`, `</CODE>`, `<IFMASK>`, `<ANS>`, and `<TASK=IF_COND>` are atomic tokens.
- **Artifacts.** `artifacts/tokenizer_v6_gpt2style/` (vocab.json, merges.txt, tokenizer.json). These are then used to encode both datasets and to perform masking by token indices if needed.

---

8. File formats and scripts

- **Scripts.**
  - `build_pretrain_corpus.py` — loads `functions_v2.jsonl`, applies filters, AST-dedup, targeted augmentation, and emits `pretrain_corpus_v3.txt`.

    - `build_finetune_dataset.py` — extracts `if` headers, masks one per function, creates labels, validates with AST, and writes `finetune_v3_{split}_prepped.jsonl`.
    - `prepare_eval_inputs.py` — final sanity pass ensuring parsability and windowing invariants.
    - `train_tokenizer.py` — learns the byte-level BPE tokenizer with special tokens.
- **Pre-training text.** One code block per paragraph, wrapped by `<CODE> ... </CODE>`, with ~8% of `if`-bearing blocks augmented.
- **Fine-tuning JSONL.** Fields: `id`, `input`, `expected_condition`, and provenance (`repo`, `path`, `sha`).

---

9. Quality checks and acceptance tests

Before freezing each dataset version:

1. **Parsability rate.** >99% of inputs must be `ast.parse`-able after masking/windowing.
2. **Leakage audit.** No shared repo or fingerprint across `train/val/test` for fine-tuning; spot-check overlaps between pre-training and fine-tuning test.
3. **Distribution sanity.** Plot histograms of function length, number of `if` per function, and label length (tokens/chars); ensure no pathologies (e.g., labels empty or excessively long).
4. **Regex false positives.** Manually sample matches near decorators, multiline conditions, and `if` inside strings to verify the extractor.
5. **Spot evaluations.** Dry-run 50 samples end-to-end (mask → label recovery) to confirm formatting and target extraction.

---

10. Policy, ethics, and licensing

- **Licenses.** Prefer permissive licenses (MIT/BSD/Apache). Store license text and include repository attributions in a manifest. If a license is absent or restrictive, exclude the repo.
- **PII & secrets.** Strip files matching secret patterns (keys, passwords). Exclude known credential files and any commit history beyond the latest snapshot.
- **Reproducibility.** Record tool versions (Python, `ast`, regex), hash of the script bundle, and random seeds used for sampling/masking.

---

11. Limitations and future improvements

- **Extractor coverage.** Regex can miss exotic `if` headers (e.g., backslash-continued lines). Future work: CST/AST alignment to capture all headers reliably.
- **Augmentation balance.** The 8% rate is a pragmatic default; an ablation could tune this knob or add AST-aware negative sampling (e.g., perturb logical operators) for harder pre-training noise.
- **Beyond text.** Consider enriching inputs with lightweight structure (e.g., `<AST=...>` summaries) if allowed, or at least block-level tags for docstring/code separation.

---

12. Deliverables (for this assignment)

- **Datasets**
  - `data/pretrain_corpus_v3.txt` (≈165,886 blocks)
  - `data/finetune_v3_train_prepped.jsonl` (≈72k)
  - `data/finetune_v3_val_prepped.jsonl` (≈9k)
  - `data/finetune_v3_test_prepped.jsonl` (≈9k)
- **Tokenizer artifacts**: `artifacts/tokenizer_v6_gpt2style/`
- **Provenance manifests**: list of repos, commits, and licenses included in each split.

This dataset plan produces clean, diverse, and leakage-controlled corpora aligned with the assignment's constraints and with the downstream objective of predicting masked `if` conditions in Python functions.

---

2. Model training procedure (consolidated)

2.1 Tokenizer training — `train_tokenizer.py`

- **Type:** Byte-level BPE (GPT-2 aligned)
- **Config:** `add_prefix_space=True`, `lowercase=False`
- **Vocab size:** 50,257
- **Special tokens:** `<CODE>`, `</CODE>`, `<IFMASK>`, `<ANS>`, `<TASK=IF_COND>` included **during** tokenizer training
- **Artifacts:** `artifacts/tokenizer_v6_gpt2style/`

2.2 Pre-training — `pretrain_clm.py`

- **Model:** GPT-2 Medium (355M)
- **Objective:** Causal Language Modeling (CLM)
- **Data:** `data/pretrain_corpus_v3.txt` (wrapped functions, targeted augmentation enabled)

- **Config:** Epochs=1; Batch size=4; LR=5e-5; Warmup steps=1000; Max length=512
- **Output:** `artifacts/pretrained_gpt2_medium_v6_*/`

2.3 Fine-tuning — `finetune_if_condition.py`

- **Base:** checkpoint from pre-training
- **Task:** predict the masked condition for a single `<IFMASK>` per function
- **Data:** `data/finetune_v3_{train,val,test}_prepped.jsonl`
- **Config:** Epochs=3; Batch size=4; LR=5e-5; Warmup steps=1000; Max length=512; Eval & Save every 2000 steps
- **Output:** `artifacts/ifrec_finetuned_v6_*/`

---

3. Evaluation method (consolidated)

3.1 Prediction generation — `predict.py`

- **Inputs:** functions containing `<IFMASK>`
- **Prompting:** use the pre-windowed `input`; prompts end with `<ANS>` for format consistency
- **Decoding:** greedy; `max_new_tokens=24`; `temperature=0.0`; `do_sample=False`

3.2 Correctness & scoring (current implementation)

Normalization. We compare only the model's **first generated line**. For both the expected and predicted conditions we:

- lowercase, strip whitespace, and drop any trailing colon `:`;
- tokenize into `\w+` word tokens;
- remove stop-words: `{is, not, and, or, in, of, the, a, an, to, for, with, by}`.

Correctness (keyword-overlap heuristic).
Let `E` be the expected token set after normalization, and `P` the predicted token set.
Define `overlap = E ∩ P`. We mark a prediction **correct** iff:

- `|overlap| > 0`, **and**
- **coverage** = `|overlap| / |E| > 0.30`.

Fallback (empty → True).
If the decoded first line is empty, we replace it with `"True"` before evaluation (baseline guard).

Confidence score.
From the generation API we take per-step logits, compute per-token log-probabilities for the **generated sequence**, average them, then map to a 0–100 confidence:

$$\text{score} = \text{clip}_{[0,100]}\left(e^{\overline{\log p}} \times 100\right).$$

If no tokens were generated, `score = 0`.

Notes: This heuristic is **not** AST- or semantics-aware; it rewards lexical overlap with the expected condition. It is useful for quick benchmarking but may over/under-count equivalence in cases of synonymy, reordering, or logically equivalent rewrites.

3.3 Required CSV outputs (per assignment)

- **Columns:** `Input` (string fed to model), `Correct` (true/false), `Expected` (ground-truth condition), `Predicted` (model output), `Score` (0–100)
- **Files:** `generated-testset.csv` (Generated) and `provided-testset.csv` (From the provided benchmark)

---

4. Results

The token accuracy from finetune is shown below:

```
{'eval_loss': 2.7090160846710205, 'eval_token_accuracy': 0.5762642740619902, 'eval_runtime': 79.6454, 'eval_samples_per_second': 4.093, 'eval_steps_per_second':
{'loss': 1.5029, 'grad_norm': 2.2375776767730713, 'learning_rate': 1.4732566012186867e-06, 'epoch': 4.92}
{'loss': 1.4889, 'grad_norm': 1.9185185432434082, 'learning_rate': 9.316181448882871e-07, 'epoch': 4.95}
{'loss': 1.554, 'grad_norm': 2.066581964492798, 'learning_rate': 3.8997968855788764e-07, 'epoch': 4.98}
{'train_runtime': 8280.1735, 'train_samples_per_second': 31.616, 'train_steps_per_second': 0.989, 'train_loss': 1.9534632012257143, 'epoch': 5.0}
[sanity] one-batch token_acc = 0.5873 (supervised tokens=63)
```

The generated prediction accuracy:

```
============================================================
Prediction Summary:
  Total examples: 7000
  Correct: 3008
  Accuracy: 42.97%
  Average score: 67.79
============================================================
```

The provided testset accuracy:

```
============================================================
Prediction Summary:
  Total examples: 288
  Correct: 132
  Accuracy: 45.83%
  Average score: 63.40
============================================================
```

Discussion:
Across two evaluations under our current correctness metric, performance is consistent: **43.1%** on the 288-sample benchmark (avg confidence **57.5**) and **43.0%** on the full 7,000-sample set (avg confidence **67.8**). This gap between >50% token-level accuracy during fine-tuning and ~43% sequence-level correctness is expected: a single wrong token can break a condition, and our metric rewards full-line correctness rather than partial matches. The higher confidence on the large set suggests mild over-confidence (some wrong predictions still score confidently), but overall the model is clearly learning non-trivial signal from pretraining + finetuning rather than producing noise.

Likely factors. Strict matching at the condition level, noisy/rough training labels, long contexts, and occasional decoding drift.

Next steps. Tighten inference hygiene (keep <IFMASK> and <ANS> in the encoded window, anchor <ANS> on the mask line, greedy + newline stop, parseable-prefix trimming), report **parse rate** alongside accuracy, and consider focusing training loss on the **condition span**. If desired, add an **AST-equivalence** evaluation to complement the current metric.