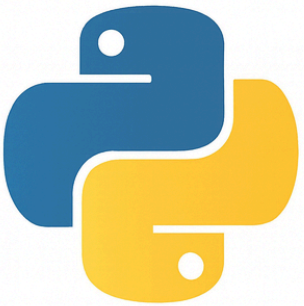
Bonham Lab

Tufts UNIVERSITY | School of Medicine

M2F Microbiome to Function



A practical pipeline for mining UniProt, cleaning annotations, and turning biology into machine-learnable features

Author
Yehor Mishchyriak
Summer Research Intern, Bonham Lab, Tufts University School of Medicine
Undergraduate Student, Wesleyan University

Affiliations
Bonham Lab, Tufts University School of Medicine
Wesleyan University, Middletown, CT, USA

Contact
ymishchyriak@wesleyan.edu

Contents:

Executive Overview

Problem. Functional annotation projects need clean, machine-ready representations of proteins at scale. Raw UniProt text is messy (free-form prose, PubMed debris, duplicated tokens), and annotations are heterogeneous (GO terms, EC numbers, domains, pathways). Researchers end up reinventing the same brittle scripts over and over.

Solution (M2F). M2F is a modular toolkit that:

- **Mines** UniProtKB at scale with rate-limited, batched REST calls (HPC-friendly).
- **Cleans & normalizes** key text fields with targeted regex extractors.
- **Encodes features** into numerical tensors:
    - **Dense embeddings** for amino acid sequences (ESM-2) and for free-text fields (OpenAI, with SQLite+LRU caching).
    - **Structured label encodings** for GO and EC (auto depth selection / class-budgeting).
- **Persists datasets** compactly in a **single Zarr ZipStore** per split (restorable losslessly).
- Works well with **SLURM**; Some example scripts are included in the package code.

End state. You get a reproducible path from HUMAnN outputs → UniProt records → a tidy, numeric table (vectors + multi-hot tuples) ready for GNNs or any downstream ML.

Architecture & Modules (what each piece does)

Public API

The package's `__init__.py` re-exports the core user-facing functions and classes so end-users can `import M2F` and reach everything directly.

A. Mining layer — `mining_utils.py`

- **`extract_accessions_from_humann(path)` / `extract_all_accessions_from_dir(dir, pattern)`**
  Parses HUMAnN gene-families files, extracts **UniRef90** accessions, and filters out IDs that can't be resolved by UniProt.
- **`fetch_uniprotkb_fields(uniref_ids, fields, request_size, rps, ...)`**
  Sends **batched, rate-limited** queries to UniProtKB (`/uniprotkb/search?format=tsv&fields=...`), with **retry logic that halves batch size** after HTTP errors and continues. Returns a tidy DataFrame (or empty with the requested columns).
- **`fetch_save_uniprotkb_batches(...)`**
  Cuts a huge accession list into **coarse batches**; for each batch, it calls the function above, then **saves to Parquet (or CSV fallback)**. Perfect for SLURM: predictable memory, clear progress logs, and resumable outputs.

B. Cleaning layer — `cleaning_utils.py`

- A small **regex DSL** per column (e.g., **Domain [FT]**, **GO**, **EC**, **Pathway**, **Cofactor**, etc.).
- **`clean_col(df, col_name, ...)`**:
  - Strips PubMed references (inline and braced).
  - Extracts structured tokens via the column's pattern.
  - Optionally normalizes (lowercase, punctuation sanitize, whitespace collapse).
  - **Always returns tuples** (deduped, order-preserving). NaNs become `()`.
- **`clean_cols(df, col_names, ...)`**: Orchestrates the above for multiple columns without repeated copying.

Design choice: outputs are **tuples** so the next stage can treat every cell as "a bag of items" without having to re-parse or special-case NaNs.

C. Embedding & Encoding layer — `embedding_utils.py`

- **Sequence embeddings — `AAChainEmbedder`**
  - Wraps **ESM-2** checkpoints from Hugging Face; you pick the representation layer (`last`, `second_to_last`, or layer index).
  - Carefully **masks BOS/EOS and padding**, pools **only residue tokens** (mean-pooling).
  - Runs on CPU or CUDA; supports batching and automatic device mapping. Returns **`np.ndarray (hidden_dim,)`** per sequence.
- **Free-text embeddings — `FreeTXTEmbedder`**
  - Uses OpenAI embeddings (e.g., `text-embedding-3-large`).
  - **Two-tier cache**: a small in-RAM **LRU** and an **SQLite** on-disk store.
  - On exit, any LRU entries are **flushed** to SQLite. You can budget cache size (KB).
  - Interface: `embed_sequences(list[str]) -> list[np.ndarray]`.
- **Multi-label encoders**
  - **`MultiHotEncoder`**: general "tuple of labels → tuple of integer indices" using `MultiLabelBinarizer`, but **returns indices, not a dense multi-hot matrix**, minimizing memory.
  - **`GOEncoder`**: collapses GO IDs to a **fixed depth** (either user-specified or **auto** from the depth distribution to hit a coverage target), then encodes with `MultiHotEncoder`.
  - **`ECEncoder`**: collapses EC strings to a chosen depth; if depth is omitted, it **auto-chooses** a depth so the number of distinct classes roughly matches a **target class budget** (derived from `N / examples_per_class`). Encodes to tuples of indices.

D. Feature engineering & persistence — `feature_engineering_utils.py`

- **Pooling & mapping**
  - `max_pool` picks the vector with **max L2 norm** among candidates (robust signal selector for sets of embeddings).
  - `vals2embs_map` builds a **value→embedding map** for any column whose cells are tuples of items (dedup + batch embed once).
- **Turn text & sequences into vectors**
  - `embed_freetxt_cols(df, cols, FreeTXTEmbedder)`: for each listed column of tuples, embed all unique strings, then **max-pool per row**.
  - `embed_ft_domains(df, AAChainEmbedder)`:
    - Parses **`Domain [FT]`** ranges like `"5..10"` to slice `Sequence`,
    - Embeds each **domain subsequence**,
    - **Max-pools** into a single vector per entry.
  - `embed_AAsequences(df, AAChainEmbedder)`: embeds **full sequences** (expects `Sequence` to be a singleton tuple).
- **Hierarchical label encoders**
  - `encode_go(df, col, depth|coverage_target)` and `encode_ec(df, col, depth|examples_per_class)` return the **modified DataFrame** and the **class label mapping** (term → index).
  - `encode_multihot(df, col)`: generic one-shot wrapper for non-hierarchical label columns.
- **Persistence**
  - `save_df(df, path.zip, metadata=None)`: writes a **single Zarr ZipStore** containing:
    - **`accessions`**: fixed-width ASCII bytes for `Entry`.
    - For each other column:
      - **Tuples of ints** → `flat_vals` (uint16) + `offsets` (uint16).
      - **Dense vectors** → `data` (float32 2-D array).
      - An `is_empty` flag if the column has only NaNs.
    - Optional **root attrs** to stash label vocabularies, etc.
    - Compression tuned per array (Blosc with zlib/lz4/zstd where sensible).
  - `load_df(path)` reconstructs the original DataFrame (and **restores attrs**).
  - `empty_tuples_to_NaNs(df)` convenience helper.
- **GO OBO** is bundled/loaded once and the encoder functions are exposed as module-level callables for convenience.

E. Logging — `logging_utils.py`

- **`configure_logging(logs_dir, file_level=DEBUG, console_level=WARNING)`** sets up a **timed rotating file handler** (daily, keep 7) and a console handler with your levels. No duplicate handlers on repeat calls. Filenames are timestamped.

F. Utilities — `util.py`

- `files_from(dir, pattern)` (ordered generator),
- `compose(*funcs)` (simple functional pipeline helper),
- `suppress_warnings(*WarningTypes)` decorator used to silence harmless third-party noise (e.g., transformers head, goatools warnings, zarr dtype warnings).

Data Flow (end-to-end narrative)

Input: HUMAnN gene-families files (TSV) with IDs in `READS_UNMAPPED`.

1. **Accession mining**
    - Walk a directory of HUMAnN outputs, extract **UniRef90** IDs, drop bad prefixes.
    - This yields **unique accessions** (tens of thousands to millions).
2. **UniProtKB retrieval**
    - Batched REST queries (TSV), with **rps** throttling.
    - Errors trigger **smaller batch retries**; non-recoverable IDs are skipped.
    - Results are saved per **large batch** to Parquet/CSV (HPC-friendly).
3. **Cleaning**
    - For each column of interest (domains, GO, EC, pathways, cofactors, functions):
        - Strip PubMed clutter.
        - Extract tokens with a dedicated regex.
        - Normalize and deduplicate.
        - Everything becomes a **tuple** of tokens (or `()` for missing).
4. **Feature engineering**
    - **Sequence vectors**: ESM-2 pooled embeddings for `Sequence` and per-domain subsequences (`Domain [FT]`).
    - **Text vectors**: OpenAI embeddings for `Domain [CC]`, `Function [CC]`, `Catalytic activity`, `Pathway` (with **SQLite+LRU cache**).
    - **Structured labels**:
        - GO (MF/BP) collapsed to a chosen depth (or auto by coverage) → **indices**.
        - EC collapsed to a chosen/auto depth by class-budget → **indices**.
        - Any other multi-label column → **indices**.
5. **Serialization**
    - One **`.zip` Zarr** per processed chunk.
    - Restorable with `load_df` to the same shape & column semantics, plus `df.attrs` carrying vocabularies/metadata.

Key Implementation Details & Rationale

- **Why tuples instead of dense multi-hot?**
  Returning **tuples of indices** keeps memory cheap and avoids sparse/dense bloat. You can materialize multi-hot matrices lazily only when needed (e.g., during model input assembly).
- **GO depth & EC depth auto-selection.**
  - GO: infer a depth **k** from the empirical depth distribution to hit a target **coverage** (default 80%); this stabilizes the ontology granularity across projects.
  - EC: pick a depth in `{4,3,2,1}` that makes the **unique class count** close to `N / examples_per_class` (defaults to 30) — a direct, practical way to avoid over/under-fragmented labels.
- **Domain embeddings.**
  `Domain [FT]` ranges (`start..end`) are parsed into 0-based inclusive-exclusive slices of the full `Sequence`. Each domain subsequence is embedded; **max-norm pooling** selects the most “salient” representation per protein.
- **Transformer pooling correctness.**
  Special tokens **BOS/EOS** and **PAD** are masked out. Only **AA tokens** are averaged — crucial for faithful sequence representations.
- **Caching text embeddings.**
  The LRU+SQLite design avoids re-embedding repeated strings (which is common for cofactors, pathways, families). On shutdown, the LRU flush prevents data loss.

API Cheat Sheet (by task, not by file)

Mining

- `extract_accessions_from_humann(path) -> (list[str], list[str])`
- `extract_all_accessions_from_dir(dir, pattern=None) -> (list[str], list[str])`
- `fetch_uniprotkb_fields(uniref_ids, fields, request_size=100, rps=10, ...) -> pd.DataFrame`
- `fetch_save_uniprotkb_batches(uniref_ids, fields, batch_size, single_api_request_size=100, rps=10, save_to_dir=None) -> str`

Cleaning

- `clean_col(df, col_name, apply_norm=True, apply_strip_pubmed=True, inplace=True) -> df`
- `clean_cols(df, col_names, apply_norms=None, apply_strip_pubmeds=None, inplace=False) -> df`

Embeddings & Encodings

- `AAChainEmbedder(model_key='esm2_t6_8M_UR50D', device='cpu|cuda', dtype=None, representation_layer='second_to_last|last|int')`
  - `embed_sequences(seqs: list[str], batch_size=32) -> list[np.ndarray]`
- `FreeTXTEmbedder(api_key, model='LARGE_OPENAI_MODEL', cache_file_path=None, caching_mode='NOT_CACHING|APPEND|CREATE/OVERRIDE', max_cache_size_kb=1000)`
  - `embed_sequences(texts: list[str], batch_size=1000) -> list[np.ndarray]`
- `encode_go(df, col, depth=None, coverage_target=None, inplace=False) -> (df, dict)`
- `encode_ec(df, col, depth=None, examples_per_class=30, inplace=False) -> (df, dict)`
- `encode_multihot(df, col, inplace=False) -> (df, dict)`

- `embed_freetxt_cols(df, cols, FreeTXTEmbedder, batch_size=1000, inplace=False) -> df`
- `embed_ft_domains(df, AAChainEmbedder, batch_size=128, inplace=False) -> df`
- `embed_AAsequences(df, AAChainEmbedder, batch_size=128, inplace=False) -> df`

Persistence

- `save_df(df, path.zip, metadata: dict|None) -> None`
- `load_df(path|path_without_zip) -> df`
- `empty_tuples_to_NaNs(df, inplace=False) -> df`

Logging

- `configure_logging(logs_dir, file_level=logging.DEBUG, console_level=logging.WARNING) -> None`

Patterns & Gotchas

- **`Sequence` is a singleton tuple** after cleaning (`("ACDE...",)`). The sequence embedders expect that and index with `[0]`. Don't convert it to a raw string later unless you also adjust the embedding helpers.
- **Zarr dtypes:** strings are stored as fixed-width ASCII (`'S{max_len}'`), tuples as **(flat, offsets)**. Keep integers small when possible; you cast to `uint16` for storage efficiency.
- **GPU vs CPU:** `AAChainEmbedder` will move the model appropriately. Long sequences are truncated to the model's `max_position_embeddings` (minus 2 for BOS/EOS) with a warning — that's intentional and safe.
- **Text caching:** If you embed tons of unique strings, increase `max_cache_size_kb` or expect more frequent SQLite writes; both are fine, just a throughput trade-off.
- **GO/EC auto depth:** The auto heuristics are pragmatic, not "theory-pure." They're there to prevent class explosion and keep feature spaces sane for first-pass models.

Extending M2F

- **New free-text column?**
  Add a regex to `AVAILABLE_EXTRACTION_PATTERNS`, list the column in `clean_cols`, then pass it through `embed_freetxt_cols`.
- **New ontology / label set?**
  Subclass `MultiHotEncoder` with your own **collapse-to-depth** (or other) policy; return `(df, vocab)` the same way GO/EC do.
- **Different sequence model?**
  Clone `AAChainEmbedder`, swap the HF repo, and keep the **special-token masking** and pooling semantics identical so the rest of the pipeline stays plug-compatible.
- **Custom serialization?**
  If you need additional data types, mirror the `save_df` / `load_df` pattern: pick a compact array layout + an unambiguous reconstruction path.
