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 (June-July of 2025). Undergraduate Student, Wesleyan University (2022-2026)

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

Contact:
ymishchyriak@wesleyan.edu

Contents:

Overview

Problem. Functional annotation projects need clean, machine-ready representations of proteins at scale. Raw UniProt text is messy (free-form prose, PubMed debris, a lot of metadata), and annotations are heterogeneous (GO terms, EC numbers, functional 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.
- **Cleans & normalizes** key text fields with targeted regex extractors.
- **Encodes features** into numerical tensors:
    - **Dense embeddings** for amino acid sequences and for free-text fields.
    - **Structured label encodings** for GO and EC.
- **Persists datasets** compactly in a **single Zarr ZipStore** and provides functionality for easy dataset reconstruction from the ZipStore.

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

Typical Data Flow

*The following is given in broad strokes; full API explanation is provided in API Reference.

1. **Accession mining**

   HUMAnN gene-families TSV files are supplied either iteratively individually or as a single directory to **extract_accessions_from_humann** or **extract_all_accessions_from_dir**, respectively. In either case, the output will be two iterables: UniRef and UniClust accession IDs.

2. **UniProtKB retrieval**

   Important note: currently there is no way to mine data on UniClust accessions from UniProtKB. Hence, one may simply stash those IDs until a workaround is added to M2F. For now, we proceed with just the UniRef IDs without much data loss as typically the amount of UniClust accessions is relatively marginal compared to UniRef accessions. So, we supply the IDs to **fetch_uniprotkb_fields** or **fetch_save_uniprotkb_batches**, which retrieve the data of our interest from UniProtKB.

3. **Cleaning**

   The data retrieved in the last step is then cleaned using **clean_cols**, extracting valuable information with regex, getting rid of metadata, and normalizing text (by throwing away punctuation, making everything lower case, etc.) for columns that require this step (typically free text fields). This leaves us with a dataframe containing only valuable data of predictable format.

4. **Feature engineering**

   The clean dataframe is then supplied to various functions from **feature_engineering_utils** (available through the single M2F API), which encode entries of each column as either multihot or dense vectors – suitable for training ML models.

5. **Persistence**

   Following the aforementioned steps, we call **save_df** function to save our prepared data, which can later be loaded using **load_df**. This stored data can be wrapped in a PyTorch Dataset class for later training using a Dataloader.

API Reference

Logging:

```
configure_logging(
      logs_dir: str,
      file_level: int = logging.DEBUG,
      console_level: int = logging.WARNING
)
```

Description

Configures logging with a timed rotating file handler (rotates at midnight, keeps 7 files) and a console handler. Safe to call multiple times; does not duplicate handlers.

Parameters

str: **logs_dir** – directory where log files should be written.
int: **file_level** – log level for the file handler (default logging.DEBUG).
int: **console_level** – log level for the console handler (default logging.WARNING).

Returns

None

Data Mining:

```
extract_accessions_from_humann(
      file_path: str,
      out_type: type = list
)-> Tuple[Iterable, Iterable]
```

Description

Extracts accessions from a HUMAnN gene-families file.

Parses the **READS_UNMAPPED** column, filters out accessions starting with **UNK** or **UPI** (as UniProtKB queries fail on them), and returns two iterable collections: **unirefs**, **uniclusts**.

Parameters

str: **file_path** – path to a HUMAnN gene-families .tsv file

type: **out_type** – iterable type to return the data in (e.g. set, list, tuple, etc.). Defaults to list.

Returns

tuple[Iterable, Iterable] – (unirefs, uniclusts) of type out_type.

```
extract_all_accessions_from_dir(
   dir_path: str,
   pattern: Optional[re.Pattern] = None,
   out_type: type = list)
-> Tuple[Iterable, Iterable]
```

Description

Aggregates UniRef90 and UniClust90 accessions from all HUMAnN files in a directory. Iterates files (optionally filtered by regex), calls extract_accessions_from_humann per file, and unions results.

Parameters

str: **dir_path** – directory containing HUMAnN outputs.

re.Pattern|None: **pattern** – optional filename pattern to filter inputs.

type: **out_type** – iterable type for the return; defaults to list.

Returns

tuple[Iterable, Iterable] – (all_unirefs, all_uniclusts) of type out_type.

```
fetch_uniprotkb_fields(
      uniref_ids: List[str],
      fields: List[str],
      request_size: int = 100,
      rps: float = 10,
      max_retry: Optional[int | float] = float("inf")
) -> pd.DataFrame
```

Description

Fetches selected UniProtKB fields for a list of accessions using batched, rate-limited REST calls. On HTTP errors, halves batch size and retries up to max_retry; concatenates non-empty responses into a DataFrame.

Parameters

list[str]: **uniref_ids** – accessions to fetch.
list[str]: **fields** – UniProtKB field names to request (TSV).
int: **request_size** – IDs per HTTP request (default 100).
float: **rps** – max requests per second (default 10).
int|float: **max_retry** – max recursive retries on error (default inf).

Returns

pd.DataFrame – table with the requested columns (empty if nothing returned).

Notes

Respects basic rate-limiting via sleep between calls.
Uses TSV endpoint; treats empty fields as NaN.

```
fetch_save_uniprotkb_batches(
      uniref_ids: List[str],
      fields: List[str],
      batch_size: int,
      single_api_request_size: int = 100,
      rps: float = 10,
      save_to_dir: Optional[str] = None
) -> str
```

Parameters

Processes very large ID lists by splitting into coarse batches, fetching each batch via fetch_uniprotkb_fields, and saving results to Parquet (CSV fallback). Designed for HPC/SLURM.

Parameters

list[str]: **uniref_ids** – accessions to process.
list[str]: **fields** – UniProtKB fields to fetch.
int: **batch_size** – number of IDs per coarse batch (each batch becomes a file).
int: **single_api_request_size** – per-HTTP request size inside a batch (default 100).
float: **rps** – requests per second throttle (default 10).
str|None: **save_to_dir** – output directory; created if missing (default: cwd).

Returns

str – path to the directory where batch files were saved.

Data Cleaning:

```
clean_col(
      df: pd.DataFrame,
      col_name: str,
      apply_norm: bool = True,
      apply_strip_pubmed: bool = True,
      inplace: bool = True
) -> pd.DataFrame
```

Description

Cleans a single column: optionally strips PubMed references, extracts information via a column-specific regex rule (if available), normalizes the extracted information, de-duplicates, and represents each cell as a tuple. NaNs become empty tuples ().

Parameters

pd.DataFrame: **df** – input table.
str: **col_name** – column to clean.
bool: **apply_norm** – whether to normalize the extracted information (default True).
bool: **apply_strip_pubmed** – whether to remove PubMed refs (default True).
bool: **inplace** – mutate df in place (default True).

Returns

pd.DataFrame – cleaned DataFrame (same object if inplace=True).

```
clean_cols(
      df: pd.DataFrame,
      col_names: List[str],
      apply_norms: Optional[Dict[str, bool]] = None,
      apply_strip_pubmeds: Optional[Dict[str, bool]] = None,
      inplace: bool = False
) -> pd.DataFrame
```

Description

Cleans multiple columns using clean_col under the hood. Supports per-column toggles for normalization and PubMed stripping; avoids repeated deep copies.

Parameters

pd.DataFrame: **df** – input table.
list[str]: **col_names** – columns to process.
dict[str,bool]|None: **apply_norms** – per-column normalization flags (default all True).
dict[str,bool]|None: **apply_strip_pubmeds** – per-column strip flags (default all True).
bool: **inplace** – mutate df in place (default False).

Returns

pd.DataFrame – cleaned DataFrame.

Numerical Data Encoding:

```
AAChainEmbedder(
      model_key: str = "esm2_t6_8M_UR50D",
      device: str = "cpu",
      dtype: Union[torch.dtype, None] = None,
      representation_layer: Union[int, str] = "second_to_last"
)
```

Description

Computes mean-pooled ESM-2 embeddings for amino-acid sequences. Masks out special tokens and pads; pools only residue tokens. Supports CPU/CUDA.

Parameters (constructor)

str: **model_key** – one of ESM-2 repo keys (e.g. esm2_t30_150M_UR50D).
str: **device** – "cpu" or "cuda:idx".
torch.dtype|None: **dtype** – optional weight precision override.
int|str: **representation_layer** – layer index, or "last" / "second_to_last".

Methods

```
<AAChainEmbedder>.embed_sequences(
      seqs: List[str],
      batch_size: int = 32)
-> List[np.ndarray]  – returns float32 vectors per sequence.
```

Notes

Sequences longer than model max length are truncated (logs a warning)

```
FreeTXTEmbedder(
      api_key: str, model: str,
      cache_file_path: Union[str, None] = None,
      caching_mode: str = "NOT_CACHING",
      max_cache_size_kb: int = 1000
)
```

Description

Embeds free-text strings with OpenAI embeddings. Provides a two-tier cache: in-RAM LRU and on-disk SQLite. Flushes LRU entries to SQLite on interpreter exit.

Parameters (constructor)

str: **api_key** – OpenAI API key.
str: **model** – "SMALL_OPENAI_MODEL" or "LARGE_OPENAI_MODEL".
str|None: **cache_file_path** – SQLite DB path for caching (optional).
str: **caching_mode** – "NOT_CACHING" | "APPEND" | "CREATE/OVERRIDE".
int: **max_cache_size_kb** – approximate RAM budget for LRU (default 1000 KB).

Methods

```
<FreeTXTEmbedder>.embed_sequences(
      seqs: List[str],
      batch_size: int = 1000
) -> List[np.ndarray]
```
– returns float32 vectors in input order, reusing cache hits.

Notes

When the LRU exceeds max_cache_size_kb, the least-recently-used entry is spilled to SQLite.

`MultiHotEncoder()`

Description

Encodes multi-label cells (tuples of strings) into tuples of integer indices using sklearn.MultiLabelBinarizer, without materializing dense multi-hot arrays.

Parameters (constructor)

(none)

Methods

`<MultiHotEncoder>.encode(self, sequences: pd.Series)->dict` – fits on the provided series of tuples and returns:

encodings: list[tuple[int,...]] – per-row index tuples,

class_labels: dict[str,int] – label to index mapping.

Notes

All elements in the series must be tuples; raises ValueError otherwise.

`GOEncoder(obo_path: str)`

Description

Collapses GO IDs to a fixed depth (or auto-selects depth by coverage of the observed depth distribution), then encodes them via MultiHotEncoder.

Parameters (constructor)

str: **obo_path** – path to go-basic.obo.

Methods

```
<GOEncoder>.encode_go(
    df: pd.DataFrame,
    col_name: str,
    depth: Union[None, int] = None,
    coverage_target: Union[float, None] = None,
    inplace: bool = False
) -> (pd.DataFrame, dict)
```

– collapses per-row GO sets to depth k (explicit or auto), encodes to index tuples, and returns updated df + term→index map.

Notes

Exactly one of depth or coverage_target must be provided.

`ECEncoder()`

Description

Collapses EC strings to a specified depth (1–4) or auto-selects a depth whose unique class count is close to N / examples_per_class. Encodes results as tuples of integer indices.

Parameters (constructor)

(none)

Methods

```
<ECEncoder>.encode_ec(
      df: pd.DataFrame,
      col_name: str,
      *,
      depth: Optional[int] = None,
      examples_per_class: int = 30,
      inplace: bool = False
) -> Tuple[pd.DataFrame, Dict[str, int]]
```

– collapses per-row EC sets to target depth and encodes with MultiHotEncoder.

```
embed_freetxt_cols(
      df: pd.DataFrame,
      cols: List[str],
      embedder: FreeTXTEmbedder,
      batch_size: int = 1000,
      inplace: bool = False) -> pd.DataFrame
```

Description

Embeds specified free-text tuple columns using FreeTXTEmbedder.
Builds a value→vector map for unique strings, then max-pools per row.

Parameters

pd.DataFrame: **df** – input table.
list[str]: **cols** – columns to embed (each cell is a tuple of strings).
FreeTXTEmbedder: **embedder** – text embedder instance.
int: **batch_size** – embedding batch size (default 1000).
bool: **inplace** – mutate df in place (default False).

Returns

pd.DataFrame – with listed columns replaced by vectors.

```
embed_ft_domains(
    df: pd.DataFrame,
    embedder: AAChainEmbedder,
    batch_size: int = 128,
    inplace: bool = False
) -> pd.DataFrame
```

Description

Embeds domain-level subsequences from Domain [FT] and Sequence.
Parses ranges like "5..10", slices the full sequence, embeds each domain with AAChainEmbedder, and max-pools to a single vector per row.

Parameters

pd.DataFrame: **df** – input table with Domain [FT] and Sequence.
AAChainEmbedder: **embedder** – sequence embedder instance.
int: **batch_size** – embedding batch size (default 128).
bool: **inplace** – mutate df in place (default False).

Returns

pd.DataFrame – with Domain [FT] replaced by vectors.

```
embed_AAsequences(
    df: pd.DataFrame,
    embedder: AAChainEmbedder,
    batch_size: int = 128,
    inplace: bool = False
) -> pd.DataFrame
```

Description

Embeds full amino-acid sequences (expects Sequence as a singleton tuple per row) with AAChainEmbedder.

Parameters

pd.DataFrame: **df** – input table with Sequence.
AAChainEmbedder: **embedder** – sequence embedder instance.
int: **batch_size** – embedding batch size (default 128).
bool: **inplace** – mutate df in place (default False).

Returns

pd.DataFrame – with Sequence replaced by vectors.

```
encode_go(
    df: DataFrame,
    col_name: str,
    depth: int | None = None,
    coverage_target: float | None = None,
    inplace: bool = False
) -> tuple[DataFrame, Any]
```

Description

Convenience wrapper bound to the packaged GO DAG.
Collapses GO terms to a fixed or auto-selected depth and encodes them to index tuples.

Parameters

pd.DataFrame: **df** – input table.
str: **col_name** – column with tuples of GO IDs.
int|None: **depth** – fixed depth; mutually exclusive with coverage_target.
float|None: **coverage_target** – percentile (0–1) to auto-select depth.
bool: **inplace** – mutate df in place (default False).

Returns

tuple[pd.DataFrame, dict[str,int]] – encoded DataFrame and GO term→index map.

```
encode_ec(
      df: DataFrame,
      col_name: str,
      *,
      depth: int | None = None,
      examples_per_class: int = 30,
      inplace: bool = False
) -> Tuple[DataFrame, Dict[str, int]]
```

Description

Convenience wrapper for ECEncoder.encode_ec.
Collapses EC numbers to a depth (fixed or auto) and encodes them to index tuples.

Parameters

pd.DataFrame: **df** – input table.

str: **col_name** – column with tuples of EC strings.

int|None: **depth** – explicit collapse depth (1–4).

int: **examples_per_class** – target density for auto depth (default 30).

bool: **inplace** – mutate df in place (default False).

Returns

tuple[pd.DataFrame, dict[str,int]] – encoded DataFrame and EC→index map.

```
encode_multihot(
    df: pd.DataFrame,
    col: str,
    inplace: bool = False
) -> Tuple[pd.DataFrame, Dict[str, int]]
```

Description

One-shot convenience wrapper around MultiHotEncoder.

Fits on df[col], replaces each tuple of labels with a tuple of integer indices, and returns the label vocabulary.

Parameters

pd.DataFrame: **df** – input table.

str: **col** – column containing tuples of labels.

bool: **inplace** – mutate df in place (default False).

Returns

tuple[pd.DataFrame, dict[str,int]] – encoded DataFrame and label→index map.

Data Persistence:

```
save_df(
      df: pd.DataFrame,
      pth: str,
      metadata: Optional[dict] = None
) -> None
```

Description

Persists a heterogeneous DataFrame to a single Zarr ZipStore (.zip). Stores strings as fixed-width ASCII, tuples of ints as (flat_vals, offsets), and vectors as a 2-D float32 array. Optionally writes metadata to root attrs.

Parameters

pd.DataFrame: **df** – table to persist (must include Entry strings).
str: **pth** – output filename ending with .zip.
dict|None: **metadata** – optional dictionary of metadata to store as attributes.

Returns

None

Notes

Column types supported: tuple of ints (or empty tuple), numpy ndarray of floats. Others raise ValueError.

```
load_df(path: str) -> pd.DataFrame
```

Description

Loads a DataFrame previously saved with save_df from a Zarr ZipStore. Reconstructs columns from stored layouts and restores root attributes into df.attrs.

Parameters

str: **path** – path to .zip (or base name; .zip is appended if missing).

Returns

pd.DataFrame – reconstructed table with attributes.

Miscellaneous:

```
empty_tuples_to_NaNs(
      df: pd.DataFrame,
      inplace: bool = False

) -> pd.DataFrame
```

Description

Replaces all empty tuple cells () in a DataFrame with np.nan.

Parameters

pd.DataFrame: **df** – input table.

bool: **inplace** – mutate df in place (default False).

Returns

pd.DataFrame – DataFrame with empty tuples converted to NaN.

`util`

Description

Small helper utilities.

Functions

`files_from(`
`dir_path: str,`
`pattern: re.Pattern|None = None`
`)-> Iterator[str]` – Yields full paths of files in dir_path matching pattern (sorted).

`compose(*funcs)` – Simple left-to-right function composition helper.

`suppress_warnings(*warning_types: Type[Warning])`
`-> decorator` – Decorator that suppresses the given warning types inside the wrapped function.

Parameters (per function)

See signatures above.

Returns

As per each function signature.

Extending M2F

- **New free-text column?**
  Add a regex to **`AVAILABLE_EXTRACTION_PATTERNS`** in the **`cleaning_utils.py`** module, 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.

Examples

Data Mining:

```
import M2F
import pandas as pd
import os

gene_fam_data = "/path/to/humann/files"
# note the use of "_" due to UniClust IDs also being returned
unirefs, _ = M2F.extract_accessions_from_humann(gene_fam_data)

df = M2F.fetch_uniprotkb_fields(unirefs,
            fields=["accession", "ft_domain", "cc_function", "go_f",
                   "go_p", "sequence"], request_size=100, max_retry=5)

df.to_csv("my_uniprot_data.csv")
```

Cleaning and Feature Engineering:

```
import M2F
import pandas as pd
import os

# Protein IDs must stay intact, so don't specify the accession column named "Entry"
# Also, note that uniprot uses different names in the TSVs it returns compared to
# the requested column names:

col_names=["Domain [FT]",
       "Gene Ontology (molecular function)",
       "Gene Ontology (biological process)",
       "Function [CC]",
       "Sequence"]

apply_norms={"Domain [FT]" : False,
       "Gene Ontology (molecular function)" : False,
       "Gene Ontology (biological process)" : False,
       "Function [CC]" : True,
       "Sequence" : False}



aa_embedder = M2F.AAChainEmbedder(model_key="esm2_t6_8M_UR50D", device="cuda:0")
```

```
txt_embedder = M2F.FreeTXTEmbedder(api_key="my-openai-api-key",
                                   model="LARGE_OPENAI_MODEL",
                                   cache_file_path="example.db",
                                   caching_mode="CREATE/OVERRIDE",
                                   max_cache_size_kb=20_000)


def process_df_inplace(df: pd.DataFrame, *, col_names: list, apply_norms: dict):
    # clean
    M2F.clean_cols(df, col_names=col_names, apply_norms=apply_norms, inplace=True)
    # encode
    M2F.embed_ft_domains(df, aa_embedder, inplace=True)
    M2F.embed_AAsequences(df, aa_embedder, inplace=True)
    M2F.embed_freetxt_cols(df, ["Function [CC]"], txt_embedder, inplace=True)
    # note: M2F.encode_go returns the updated df and mapping values → integer labels
    # we use underscore unpacking the output to discard the updated df
    # since we modified the original df in place and hence don't need the copy

    _, gomf_meta = M2F.encode_go(df,
                "Gene Ontology (molecular function)",
                coverage_target=0.8, inplace=True)
    _, gobp_meta = M2F.encode_go(df,
                "Gene Ontology (biological process)",
                coverage_target=0.8, inplace=True)

    return {"gomf_meta": gomf_meta, "gobp_meta": gobp_meta}


for file in M2F.util.files_from("/path/to/input/dir"):
    file_name = os.path.basename(file)
    out_pth = os.path.join("/path/to/output/dir", file_name.replace(".csv", ".zip"))
    # load
    df = pd.read_csv(file)
    # process
    meta = process_df_inplace(df, col_names=col_names, apply_norms=apply_norms)
    # save
    M2F.save_df(df, out_pth, metadata=meta)
```