Kunitz Domain Detection Using Profile Hidden Markov Models (HMM)

Author: **Martina Castellucci**

This notebook implements a bioinformatics pipeline to identify the Kunitz-type protease inhibitor domain (PF00014) in protein sequences using HMM-based detection.

We will:

- Collect and preprocess positive and negative datasets.
- Reduce redundancy using CD-HIT and extract structural alignments from PDB.
- Build an HMM profile using `hmmbuild`.
- Filter sequences to avoid bias in evaluation.
- Run `hmmsearch` on test sets.
- Evaluate model performance with multiple thresholds.

Linux Command-Line Tools: Summary of Usage

This section summarizes the most commonly used Linux shell commands in this notebook:

- **`cat`**: Concatenates and displays the content of files.
- **`awk`**: A powerful text-processing tool used for pattern scanning and field-based extraction.
- **`grep`**: Searches for lines matching a pattern within a file.
- **`cut`**: Extracts specific columns or fields from each line of a file.
- **`sort`**: Sorts lines in a file, alphabetically or numerically.
- **`comm`**: Compares two sorted files and shows common and distinct lines.
- **`makeblastdb`**: Creates a searchable BLAST database from a FASTA file.
- **`blastp`**: Compares protein sequences using BLAST against a protein database.
- **`cd-hit`**: Clusters sequences to reduce redundancy based on sequence identity thresholds.
- **`hmmbuild`**: Builds a profile Hidden Markov Model (HMM) from a multiple sequence alignment.
- **`hmmsearch`**: Searches sequence databases for matches to a profile HMM.
- **`head` / `tail`**: Outputs the first or last N lines of a file (useful for dataset splitting).
- **`wc`**: Word count; used with `-l` to count lines (e.g., number of sequences).
- **`less`**: Paginates the display of long files (useful for inspection).
- **`clstr2txt.pl`**: Converts .clstr output from CD-HIT into a readable tabular format that lists each sequence and identifies the representative of each cluster.

Python Script Logic and Key Rules

The following Python scripts automate the core evaluation steps of the pipeline, from data preparation to performance analysis.
Each script is built around a simple but powerful rule, and contributes to the robustness and reproducibility of the model.

`get_seq.py` – Extract sequences by UniProt ID

Input:

- A text file of UniProt accession IDs (e.g. `pos_1.ids`)
- A full FASTA file from SwissProt (e.g. `uniprot_sprot.fasta`)

Output:

- A filtered FASTA file (e.g. `pos_1.fasta`)

```python accession = record.id.split('|')[1] if '|' in record.id else record.id if accession in id_set: SeqIO.write(record, f_out, 'fasta')

---

## `performance.py` – Compute classification metrics from .class file

**Input**:

- A `.class` file from `hmmsearch` output
- An e-value threshold (e.g., `1e-2`)

**Output**:

- Confusion matrix (TP, FP, TN, FN)
- Metrics: Accuracy, MCC, Precision, Recall

**Rule**:
“If the e-value is less than or equal to the threshold, predict Kunitz (1); else predict non-Kunitz (0).”

```python pred_label = 1 if evalue <= threshold else 0 cm[pred_label][true_label] += 1

---

## `MCC_plot.py` – Visualize MCC values and optimal threshold

**Input**:

- A list of MCC values for different e-value thresholds (for Set 1 and Set 2)
- Corresponding list of thresholds

**Output**:

- Line plot showing MCC per threshold
- Highlighted best threshold on the plot

**Rule**:
“Find the threshold with the highest MCC and mark it in the plot.”

```python best_index_mean = mean_mcc.index(max(mean_mcc)) best_threshold_mean = thresholds[best_index_mean] best_mcc_mean = mean_mcc[best_index_mean]
```

`ROC_curve.py` – Build ROC curves using e-values

Input:

- Two `.class` files (Set 1 and Set 2), each containing:
  `ID`, `real_class`, `evalue`, `pred_class`

Output:

- ROC plot (`roc_curve_evalue_sets.png`)
- AUC values for Set 1 and Set 2

Rule:
“Convert e-values to -log10 scores to use them as classification scores, then compute ROC.”

```python score = -np.log10(df["evalue"].replace(0, 1e-300)) fpr, tpr, _ = roc_curve(y_true, score)
```

Confusion Matrices – Interpretation and Generation

Confusion matrices provide a detailed view of how well the model classified positive and negative examples in each fold.
They show how many sequences were correctly or incorrectly classified as containing the Kunitz domain.

Matrix Structure

Each matrix is a 2×2 table with:

	Predicted: Kunitz	**Predicted: Non-Kunitz**
Actual: Kunitz	True Positives (TP)	False Negatives (FN)

	Predicted: Kunitz	Predicted: Non-Kunitz
Actual: Non-Kunitz	False Positives (FP)	True Negatives (TN)

Fold 1

Generated from `confusion_matrix.py` using this rule:

```python fold1 = pd.DataFrame([[283999, 0], [2, 181]], index=["Actual Negative", "Actual Positive"], columns=["Predicted Negative", "Predicted Positive"]) sns.heatmap(fold1, annot=True, fmt='d', cmap='Blues')

## Fold2

**Generated from `confusion_matrix.py`** using this rule:

```python fold2 = pd.DataFrame([[283922, 0], [2, 181]], index=["Actual Negative", "Actual Positive"], columns=["Predicted Negative", "Predicted Positive"]) sns.heatmap(fold2, annot=True, fmt='d', cmap='Greens')

---

### `rmsd.py` – Structural Alignment Quality Check

This script assesses the consistency of the structural alignment used to train the profile HMM. It visualizes the Root Mean Square Deviation (RMSD) and Q-score for each PDB entry selected as training input.

**Input**:

- A manually curated list of PDB chains with RMSD and Q-score values (from PDBeFold)

**Output**:

- A scatter plot (`kunitz_structures_scatter.png`) showing alignment quality and highlighting structural outliers

**Rule**:
“Plot RMSD vs Q-score and annotate structures with RMSD > 2 Å as potential outliers.”

```python plt.scatter(df["RMSD"], df["Q-score"], color="mediumseagreen") plt.annotate(row["PDB ID"], (row["RMSD"], row["Q-score"]), color='red')

# Workflow:

# 1. Extract Human and Non-Human Kunitz Sequences

From the file `all_kunitz.fasta`, extract sequences annotated as "Homo sapiens" and separate them from non-human sequences.
```

```
# Extract human sequences
awk '/^>/ {f=($0 ~ /Homo sapiens/)} f' all_kunitz.fasta >
human_kunitz.fasta

# Extract non-human sequences
grep -v "Homo sapiens" all_kunitz.fasta > nothuman_kunitz.fasta

# Count the number of human sequences (headers only)
grep "sp" human_kunitz.fasta | wc -l
```

2. Extract PF00014 sequences from PDB report and cluster with CD-HIT

Download a CSV from RCSB PDB filtered for:

- PFAM = PF00014
- Length 45–80 aa
- Resolution ≤ 3.5 Å

Then extract the sequences in FASTA and reduce redundancy using CD-HIT at 90%.

```
# Extract PF00014 sequences from the CSV
cat rcsb_pdb_custom_report_20250410062557.csv | tr -d '"' \
| awk -F ',' '{if (length($2)>0) {name=$2}; print name ,$3,$4,$5}' \
| grep PF00014 \
| awk '{print ">"$1"_"$3; print $2}' > pdb_kunitz_customreported.fasta

# Remove redundancy with CD-HIT (90% sequence identity)
cd-hit -i pdb_kunitz_customreported.fasta -o
pdb_kunitz_customreported_nr.fasta -c 0.9
```

3. Extraction of representative sequences after CD-HIT

After running CD-HIT, we convert the `.clstr` file into a readable format, extract the representative IDs for each cluster, and retrieve the corresponding FASTA sequences. This allows us to work with a non-redundant dataset.

```
# Convert the .clstr file generated by CD-HIT into a readable format
clstr2txt.pl pdb_kunitz_customreported_filtered.clstr >
pdb_kunitz.clusters.txt

# Extract representative sequence IDs (those with value 1 in the fifth
column)
awk '$5 == 1 {print $1}' pdb_kunitz.clusters.txt > pdb_kunitz_rp.ids

# Extract FASTA sequences corresponding to representative IDs
> pdb_kunitz_rp.fasta
for i in $(cat pdb_kunitz_rp.ids); do
```

```
    grep -A 1 "^>$i" pdb_kunitz_customreported.fasta | head -n 2 >>
pdb_kunitz_rp.fasta
done

# Prepare the ID list for PDBefold by converting "_" to ":" as
required
grep ">" pdb_kunitz_rp.fasta | tr -d ">" | tr "_" ":" >
tmp_pdb_efold_ids.txt
```

4. Structural Alignment and HMM Building

Use the representative sequences to submit to PDBeFold and download the `.ali` alignment. Reformat for `hmmbuild`, then build the structural HMM.

```
# Reformat the .ali file into a FASTA-like format
awk '{
  if (substr($1,1,1)==">") {
    print "\n" toupper($1)
  } else {
    printf "%s", toupper($1)
  }
}' pdb_kunitz_rp.ali > pdb_kunitz_rp_formatted.ali

# Build the HMM model from the reformatted alignment
hmmbuild structural_model.hmm pdb_kunitz_rp_formatted.ali
```

5. BLAST Filtering to Remove Overlapping Sequences

To avoid evaluating the same sequences used to build the model, we perform BLAST between the representative PDB Kunitz sequences and the full UniProt Kunitz set.

We retain only sequences with identity < 95% and coverage < 50%.

```
# Create the BLAST database from all Kunitz sequences (human + non-
human)
makeblastdb -in all_kunitz.fasta -dbtype prot -out all_kunitz.fasta

# Run BLAST
blastp -query pdb_kunitz_rp.fasta -db all_kunitz.fasta -out
pdb_kunitz_nr_23.blast -outfmt 7

# Extract UniProt IDs to exclude (identity ≥ 95% and coverage ≥ 50%)
grep -v "^#" pdb_kunitz_nr_23.blast | awk '{if ($3>=95 && $4>=50)
print $2}' | sort -u | cut -d "|" -f 2 > to_remove.ids

# Extract all UniProt IDs from Kunitz sequences
grep ">" all_kunitz.fasta | cut -d "|" -f 2 > all_kunitz.id
```

```
# Get the list of IDs to keep (not too similar to PDB set)
comm -23 <(sort all_kunitz.id) <(sort to_remove.ids) > to_keep.ids

# Extract the final sequences to keep (valid positives for testing)
python3 get_seq.py to_keep.ids all_kunitz.fasta ok_kunitz.fasta
```

6. Construction of the Negative Dataset

Extract from SwissProt all proteins that do NOT contain the Kunitz domain and create the negative FASTA dataset.

```
# Get all UniProt IDs from SwissProt
grep ">" uniprot_sprot.fasta | cut -d "|" -f 2 > sp.id

# Exclude IDs that belong to Kunitz sequences
comm -23 <(sort sp.id) <(sort all_kunitz.id) > sp_negs.ids

# Extract the final negative sequences
python3 get_seq.py sp_negs.ids uniprot_sprot.fasta sp_negs.fasta
```

7. Train/Test Set Construction

Split the positive and negative datasets into training and testing halves using randomization.

```
# Randomize the IDs
sort -R to_keep.ids > random_ok_kunitz.ids
sort -R sp_negs.ids > random_sp_negs.ids

# Split into two halves
head -n 183 random_ok_kunitz.ids > pos_1.ids
tail -n 183 random_ok_kunitz.ids > pos_2.ids

head -n 286417 random_sp_negs.ids > neg_1.ids
tail -n 286417 random_sp_negs.ids > neg_2.ids

# Extract FASTA sequences for the 4 datasets
python3 get_seq.py pos_1.ids uniprot_sprot.fasta > pos_1.fasta
python3 get_seq.py pos_2.ids uniprot_sprot.fasta > pos_2.fasta
python3 get_seq.py neg_1.ids uniprot_sprot.fasta > neg_1.fasta
python3 get_seq.py neg_2.ids uniprot_sprot.fasta > neg_2.fasta
```

8. HMMER Search and .class File Generation

We run `hmmsearch` on each of the four FASTA sets (pos_1, pos_2, neg_1, neg_2) using the trained HMM.
To ensure consistency in e-value calculation across datasets of different sizes, we use the `-Z 1000` option.
We convert the output to `.class` format to later evaluate classification performance.

```
# Run hmmsearch with tabular output for each dataset
hmmsearch -Z 1000 --max --tblout pos_1.out structural_model.hmm
pos_1.fasta
hmmsearch -Z 1000 --max --tblout pos_2.out structural_model.hmm
pos_2.fasta
hmmsearch -Z 1000 --max --tblout neg_1.out structural_model.hmm
neg_1.fasta
hmmsearch -Z 1000 --max --tblout neg_2.out structural_model.hmm
neg_2.fasta
```

We now extract useful information from the HMMER output:

- Identifier
- True label (1 for positive, 0 for negative)
- Full-sequence E-value
- Domain E-value (if available)

These are stored in `.class` files to be used with the `performance.py` script.

```
# POSITIVES
grep -v "^#" pos_1.out | awk '{split($1,a,"|"); print a[2]"\t1\t"$5"\
t"$8}' > pos_1.class
grep -v "^#" pos_2.out | awk '{split($1,a,"|"); print a[2]"\t1\t"$5"\
t"$8}' > pos_2.class

# NEGATIVES
grep -v "^#" neg_1.out | awk '{split($1,a,"|"); print a[2]"\t0\t"$5"\
t"$8}' > neg_1.class
grep -v "^#" neg_2.out | awk '{split($1,a,"|"); print a[2]"\t0\t"$5"\
t"$8}' > neg_2.class
```

If `hmmsearch` does not report a match, that sequence is still part of the dataset and should be considered a true negative.
We assign them a default E-value of 10.0 and append them to the `.class` files using `comm`.

```
# UNMATCHED NEGATIVES: add them manually with E-value 10.0
comm -23 <(sort neg_1.ids) <(cut -f1 neg_1.class | sort) | awk '{print
$1"\t0\t10.0\t10.0"}' >> neg_1_hits.class
comm -23 <(sort neg_2.ids) <(cut -f1 neg_2.class | sort) | awk '{print
$1"\t0\t10.0\t10.0"}' >> neg_2_hits.class
```

9. Merge Datasets

We now merge the positive and negative `.class` files into two evaluation sets (`set_1` and `set_2`).

```
# Merge the datasets
cat pos_1.class neg_1_hits.class > set_1.class
cat pos_2.class neg_2_hits.class > set_2.class
```

10. Evaluate Model Performance at Different Thresholds and Analyze Errors

In this step, we evaluate model performance at different E-value thresholds using `performance.py`. The goal is to identify the optimal threshold based on the **Matthews Correlation Coefficient (MCC)**.

We also analyze false negatives—true positive sequences with high E-values that were misclassified as negatives. This helps us identify potential sensitivity issues in the model.

- Run performance evaluation for both `set_1.class` and `set_2.class` across multiple thresholds (from `1e-1` to `1e-10`)
- Extract and save the worst false negatives for comparison and further inspection

```
# Repeat for different E-value thresholds to find the optimal one
(based on MCC)
for i in $(seq 1 10); do
  python3 performance.py set_1.class 1e-$i
done > performance_set1_thresholds_final.txt

for i in $(seq 1 10); do
  python3 performance.py set_2.class 1e-$i
done > performance_set2_thresholds_final.txt

# Error analysis – Identify false negatives with high E-values
sort -grk 3 pos_1.class | less
sort -grk 3 pos_2.class | less

# Extract the worst false negatives into separate files for further
inspection
awk '$2 == 1 && $3 > 1e-5' pos_1.class | sort -grk 3 > fn_pos1.txt
awk '$2 == 1 && $3 > 1e-5' pos_2.class | sort -grk 3 > fn_pos2.txt
```