ChatGPT

Roadmap and TODOs

Purpose: This document outlines a comprehensive plan for improving our phage-bacteria interaction project in terms of engineering, explainability, and biological analysis. The next steps are organized into four tracks (A to D), each with specific goals, tasks, and deliverables that can be assigned to students or team members. The overall aim is to create a reproducible pipeline that uses explainable AI (XAI) on a two-branch CNN model to identify DNA regions important for bacteriophage-bacteria interactions and interpret them biologically.

Background: We have a dataset of bacterial and phage DNA sequences (one-hot encoded) with labels indicating if the pair interacts (yes/no). The model (`model.py`) is a two-branch 1D CNN that processes a bacterium genome (padded to ~7,000,000 bp) and a phage genome (padded to ~200,000 bp) and then concatenates their feature representations to predict interaction. We want to use XAI techniques (Grad-CAM, Integrated Gradients, DeepLIFT, etc.) to find which parts of each sequence contribute most to the prediction. By extracting those important regions and aligning them via BLAST to known sequences, we hope to discover genes or motifs (e.g. phage tail fibers, bacterial surface receptors) that drive the interaction. The tracks below ensure the project is **reproducible** (Track A), uses multiple **attribution methods** (Track B), **extracts meaningful regions** from attribution signals (Track C), and produces **biologically informative reports** (Track D). Cross-cutting tasks address testing and cleanup.

Context Artifacts

- **Model Definition** – `model.py`: Implements the two-branch 1D CNN (separate CNN for bacteria and phage sequences, merged for prediction).
- **Data Utilities** – `utils.py`: Utilities for one-hot encoding DNA sequences (including IUPAC ambiguity codes), caching datasets to numpy arrays, dataset loading, and some attribution helper functions.
- **Grad-CAM & BLAST Script** – `gradcam_to_blast.py`: CLI script that runs Grad-CAM on the model and prepares sequences for BLAST (likely identifies top scoring windows and calls BLAST).
- **BLAST to GenBank Script** – `blast_to_genebank.py`: Retrieves GenBank entries for BLAST hits (to extract gene annotations from accession numbers).
- **BLAST to BED Script** – `blast_to_bedd.py`: Converts BLAST results into BED/BEDDNA format and downloads sequence data (possibly for viewing in genome browsers or further analysis).
- **Top-Window Visualization** – `visualisation.py`: Contains code for selecting the top attribution windows and visualizing them (e.g., plotting attribution scores along the genome).
- **Pipeline Walkthrough Notebook** – `pipeline_walkthrough.ipynb`: A guided Jupyter notebook demonstrating the current process on a single example (from loading data and model, to computing Grad-CAM attributions, BLASTing regions, and interpreting results). This is a good starting point to understand the workflow.

These artifacts provide a starting point. The tasks below will evolve this code into a unified, reproducible pipeline with enhanced XAI methods and biological interpretation.

Track A — Engineering and Reproducibility

Goals: Streamline the entire analysis into a reproducible, one-command pipeline, and provide a containerized environment to ensure consistent results across different systems. By the end of this track, one should be able to go from raw CSV data to a final report with a single command. All dependencies (PyTorch, Captum, Biopython, BLAST+, etc.) should be encapsulated in an environment, and the workflow should be easy to run on any machine (or Docker) with minimal setup.

Tasks:

- **Pipeline and Workflow:**
- [ ] **Config File:** Create a `config.yaml` that centralizes all important parameters (dataset file paths, model checkpoint path, thresholds like attribution cutoff, BLAST options, output directory, etc.). This makes runs configurable without code changes. Include different profiles if needed (e.g., one config for public dataset vs. another for a private dataset).
- [ ] **Workflow Automation:** Implement a Snakemake workflow (`Snakefile`) or a Makefile that defines rules/stages for each step of the pipeline: `cache` (data encoding & caching), `predict` (model predictions), `attribution` (compute attributions), `top_regions` (select high-importance regions), `blast` (align regions via BLAST), `genebank` (fetch annotations for hits), `bed` (generate BED files/tracks), and `report` (final report assembly). The workflow should use the config file for parameters and produce outputs in a structured `results/` folder. Each rule should have clear input/output definitions so that `snakemake --cores 4` can run everything end-to-end.

- [ ] **Stable Run Naming:** Avoid using timestamps or random IDs for output folders. Instead, use human-readable, deterministic names for each run (for example, include the dataset name, model version, date, etc., or use `results/{branch}/{index}` as placeholders in the config). This makes it easier to track results. For example, a run might output to `results/baseline_model/run1/...` consistently, which aids reproducibility and comparison.

- **Unified CLI Interface:**

- [ ] **Single Entrypoint:** Create a unified command-line interface `cli.py` that serves as an entry point to run each stage of the pipeline. Use subcommands for each step (e.g., `cli.py cache`, `cli.py predict`, `cli.py explain`, `cli.py blast`, `cli.py annotate`, `cli.py bed`, `cli.py report`). Each subcommand will internally call the appropriate functions (many of which might already exist in the current scripts). This provides an easy manual alternative to Snakemake for running individual steps or the whole pipeline. For example, a user could run `python cli.py cache --config config.yaml` to cache data, or `python cli.py predict` to get model outputs.

- [ ] **Refactor Utilities into Package:** Organize the code into a Python package (e.g., a `src/` directory or similar) so that functions can be easily imported by the CLI and Snakemake. For instance, common functionality from `utils.py`, `gradcam_to_blast.py`, etc., can be moved into modules like `src/data.py`, `src/attribution.py`, `src/blast.py`, etc. **Important:** while refactoring, keep the function names and behaviors the same to avoid introducing bugs. The idea is to improve code organization (so multiple interfaces can use the same core logic) without changing outcomes. Document any changes and ensure all paths (file locations) are updated accordingly.

- **Environment and Containerization:**

- [ ] **Environment Specification:** Create an `environment.yml` (for Conda) or `requirements.txt` (for pip) listing all dependencies with exact version numbers. This includes PyTorch (ensure the version is compatible with the code and available on the target platforms), Captum (for interpretability methods), Biopython (for GenBank parsing), BLAST+ command-line tools, pandas, NumPy, Snakemake, etc. Pinning versions will ensure that results (like model predictions and attributions) remain consistent [1] [2].

- [ ] **Docker Container:** Write a `Dockerfile` that sets up a container with all the required software. For example, use an official Python base image, install Conda or pip packages from the environment file, and install BLAST+ (from NCBI) in the image. Configure the Dockerfile to also copy the code into the image. Create a `docker-compose.yml` (if needed) to simplify running the container, mounting the `data/` directory (with input CSVs and sequences) and `results/` directory (for outputs) from the host into the container. The goal is that someone can run `docker build` and `docker run ...` to execute the pipeline in an isolated environment, which is crucial for reproducibility and eliminates the "it works on my machine" problem.

- **Reliability and Testing:**

- [ ] **Logging:** Add logging throughout the pipeline to capture the progress and any issues. For each run, maintain log files (e.g., `pipeline.log` or separate logs per stage) in the output directory. Also keep a top-level `logs/` folder for aggregated or latest logs. Use Python's built-in `logging` module to log both to console and to file. The logs should record key events, parameter values, and any warnings or errors. This will greatly help in debugging and verifying that each step ran as expected.
- [ ] **Input Validation:** Implement checks for input data integrity. For example, if CSV files are used as input, verify they contain required columns (e.g., IDs, sequences, labels) with correct data types. One approach is to define a simple schema (column names and expected types/format) and validate each input file against it, providing a clear error message if something is wrong. This prevents pipeline crashes later due to unexpected input format.
- [ ] **Cache Integrity:** Since sequences are large and one-hot encoding is heavy, we cache arrays (probably as `.npy` files). Ensure these cached files have the expected shapes and dtype. For example, a bacterial sequence should encode to shape (7000000, 4) if padded to 7 million with 4 channels (A,C,G,T), and phage to (200000, 4). Implement a quick check after caching (and before using the cache in predictions) to confirm dimensions match the config thresholds and that non-`N` (or non-`Z`) bases are encoded properly. If any file is corrupted or incomplete, the pipeline should either regenerate it or error out with a clear message.

- [ ] **Consistency Check:** To ensure the model behaves the same across different machines or environments, include a small parity test. For example, after setting up the environment, load the `model_v1.pth` checkpoint and run a prediction on a few known samples (you can hard-code a couple of small example sequences or take from the dataset). Verify that the outputs (predicted interaction probabilities or logits) match expected values (within a tolerance if using floating point) [2]. This test can be part of the pipeline (e.g., a Snakemake rule or a CLI subcommand `cli.py test`) and is important for ensuring that the PyTorch model and dependencies are all working consistently (especially if GPU vs CPU could cause slight differences, use CPU for the parity test for exact reproducibility).

- **Documentation:**

- [ ] **Expand README:** Update `README.md` to include detailed setup and usage instructions. This should cover how to install dependencies (or build the Docker image), how to run the pipeline with Snakemake or the CLI, and what the expected inputs and outputs are. Include a flowchart or step-by-step diagram of the pipeline stages for clarity (you can describe it in text or include a simple diagram image if available). Also add troubleshooting tips for common issues (e.g., BLAST database not found, out-of-memory errors, etc.).
- [ ] **Contributing Guide:** Add a `CONTRIBUTING.md` (especially if multiple students or collaborators will work on this project). In it, outline the coding style (e.g., PEP8 for Python, how to name functions), how to add new steps or modules, and how to run tests. This will help maintain consistency as the code is extended.
- [ ] **Inline Documentation:** While not a separate deliverable, as you refactor and write new code, ensure to add docstrings to functions and in-code comments where necessary. This is particularly helpful for complex parts like the attribution combination logic or peak finding algorithm, so future readers (or the student themselves later on) can understand the rationale.

Deliverables (Track A):

- A `workflow/` directory containing the `Snakefile` and any config profiles, plus the top-level `config.yaml` describing the pipeline configuration.
- The unified `cli.py` script that can run the pipeline steps.
- `environment.yml` (Conda environment file) and a `Dockerfile` (plus optional `docker-compose.yml`) that encapsulate the runtime environment.
- Updated `README.md` and new `CONTRIBUTING.md` documenting how to use and develop the project.
- Logging output and basic tests demonstrating that running `snakemake --cores 4` or the docker container yields a full analysis from scratch.

Acceptance Criteria:

- Running `snakemake --cores 4` (or an equivalent one-command execution via CLI) on a fresh machine takes raw input (CSV sequences) and produces a complete results directory **without any manual intervention or errors**. This includes all intermediate files (cached data, predictions, attribution scores, region files, BLAST results, annotations) and a final report. For example, a user should be able to clone the repo, adjust the config with input file paths, and execute one command to get results.
- The Docker container, when run with appropriate volume mounts, can execute the same end-to-end pipeline successfully. For example, `docker build` followed by `docker run -v /path/to/data:/data -v /path/to/results:/results xai-pipeline:latest snakemake --cores 4` (or an entrypoint that automatically runs the pipeline) should work. All dependencies like BLAST+ executables should function inside the container.
- Reproducibility tests pass: e.g., a known small dataset processed through the pipeline yields identical outputs (same predictions, same selected regions, etc.) on different machines or reruns (given the same random seeds). Any nondeterministic components should be controlled (fixed seeds for randomness, etc.).
- Documentation is clear enough that a new user (like a student joining the project) can follow the README to set up and run the pipeline. The README should also explain the purpose of each track/stage so the context is understood.

Track B — XAI Methods and Comparisons

Goals: Extend the explainability of the model by implementing multiple attribution methods, and build a unified interface to compute and compare these attributions. We want to go beyond Grad-CAM and have methods like Saliency maps, Integrated Gradients, DeepLIFT, etc., available for our dual-input model. Once implemented, we will compare these methods both qualitatively (visual inspection of highlighted regions) and quantitatively (stability of attributions, runtime performance, and agreement between methods on what regions are important). The goal is to determine which methods are most reliable and informative for our application and possibly recommend a default method for routine analysis.

Tasks:

- **Implement Multiple Attribution Methods (using Captum or similar):** Implement at least five different attribution techniques to explain the model's predictions. Use the Captum library (which integrates with PyTorch) for many of these methods, as it provides out-of-the-box implementations [2]. The methods to implement include:
- [ ] **Saliency Maps & Guided Backpropagation:** Compute raw gradient saliency for each input (bacterium and phage sequence) with respect to the predicted interaction score. Saliency is simply the absolute value of the gradient of the output with respect to each input feature [1]. Additionally, implement **Guided Backpropagation**, which is similar to saliency but backpropagates only positive gradients through ReLU activations (to suppress noise) [3]. These methods highlight which nucleotide positions, if changed, would most affect the prediction.
- [ ] **Integrated Gradients (IG):** Implement Integrated Gradients for the inputs. This involves interpolating from a baseline sequence to the actual sequence and accumulating gradients [4]. Choose a sensible baseline for DNA sequences (e.g., an all-neutral baseline like all "N" or a random sequence). To make IG robust, consider averaging over multiple baselines or using a zero baseline for one-hot (which represents absence of any nucleotide). Also try Captum's **LayerIntegratedGradients** to attribute output back to neurons in an internal layer (e.g., the final convolution layer), which can sometimes give sharper attributions for deep networks by focusing on specific layers.
- [ ] **DeepLIFT and DeepLIFT-Shap:** Implement **DeepLIFT** (Deep Learning Important FeaTures) which compares the activation of each neuron to a reference activation and attributes differences in output to differences in input [4]. Use Captum's DeepLIFT for a quick implementation. Then implement **DeepLIFT-Shap**, which is an extension of DeepLIFT that approximates Shapley values by considering multiple reference baselines (it averages attributions over several reference inputs) – this provides a more robust attribution by simulating "leave-one-out" for groups of features, akin to SHAP values.
- [ ] **Input * Gradient and Occlusion:** Implement **Input × Gradient**, which is simply the element-wise product of the input and its gradient. This can be seen as scaling the saliency by the input's value; for one-hot encoded sequences, this has the effect of keeping only the gradient corresponding to the present base (since other bases have value 0). Also implement **Occlusion** (a form of ablation): slide a window (e.g., 50 bp or 100 bp wide) along the sequence and replace that window with a baseline (e.g., all Ns or zeros) to see how the prediction score drops [5]. This gives a coarse attribution by measuring the importance of contiguous regions (which is useful for very long sequences where fine-grained gradients might be noisy).

- [ ] **Noise Tunnel (SmoothGrad/VarGrad):** For any gradient-based method (Saliency, IG, etc.), integrate Captum's **NoiseTunnel** to improve stability [6]. NoiseTunnel adds random noise to the input multiple times and averages the attributions (SmoothGrad) or computes the variance

(VarGrad), to reduce noise and highlight consistent signals. Implement options to run Saliency or IG under a NoiseTunnel with a fixed random seed for reproducibility.

- **Handle Dual-Input Attributions:**
  Our model has two input sequences (bacteria and phage). Ensure that the attribution methods handle multi-input properly and produce understandable outputs for each branch.

- [ ] **Standardize Attribution Outputs:** Design a format (e.g., a dictionary or tuple) to return attributions separately for the bacterium and phage sequences. After computing attributions with Captum (which will return a tuple of tensors for multi-input), post-process each to get a 1D importance array per sequence. For a one-hot encoded sequence, you'll need to **combine attributions across the nucleotide channels** at each position. For example, if A, C, G, T are one-hot channels, and a certain position in the sequence is an 'A', you might take the attribution for 'A' channel at that position (since other channels are zero for that position). Alternatively, sum the absolute attributions of all channels at that position to get an importance score regardless of which nucleotide is present [7]. Reuse or refine the existing `combine_attributions` helper to do this consistently for all methods.

- [ ] **Align attributions to true sequence length:** Ensure that the attribution arrays correspond to the actual length of each sequence (before padding). If sequences are padded with a special symbol (e.g., 'Z' which maps to [0,0,0,0]), those positions should be ignored or masked out in the attribution results. Implement a step to trim or zero-out attributions beyond the real sequence length of each sample. This way, downstream region extraction (Track C) won't accidentally pick up high attribution scores in padded regions (which could happen due to edge effects in the model). Each attribution method's output should clearly associate with actual nucleotide positions (e.g., indices 1..N of the genome).

- **Evaluation and Comparison of Methods:**
  Develop tools to systematically compare these attribution methods to understand their performance and agreement.

- [ ] **Benchmark Notebook:** Create a notebook or script to benchmark the runtime and memory usage of each method on a fixed subset of data (e.g., 10 bacterium-phage pairs). Record how long each method takes and how much GPU/CPU memory is used, since methods like Occlusion can be significantly slower than Saliency. This will inform what methods are practical for larger datasets.
- [ ] **Qualitative Comparison:** For a few example pairs, generate plots of the attribution scores along the sequences for different methods. You can overlay these or put them side by side. Visually inspect if certain peaks/regions appear in multiple methods' attributions. This will require normalizing or scaling the attributions appropriately (some methods like IG give cumulative sums that might need normalization). Include these plots in the notebook for discussion.
- [ ] **Quantitative Metrics:** Compute metrics to compare attribution outputs: e.g., Spearman rank correlation between the attribution score arrays from two methods (this tells if they rank positions similarly), or Jaccard index of the top-k important regions from each method (for instance, do IG and DeepLIFT highlight overlapping regions or completely different ones?). Another metric is to check **stability**: run the same method multiple times (especially ones with randomness like NoiseTunnel or random baseline) and measure how consistent the results are (e.g., overlap of top 1% positions between runs). Implement functions to compute these metrics.

- [ ] **Reproducibility of Methods:** Where methods have stochastic elements (like NoiseTunnel or random initializations in IG baselines), ensure to set random seeds and document them so results are reproducible. As an extra test, you could repeat an attribution computation with the same seed and confirm that the outputs are identical. This gives confidence that comparisons are fair (differences are due to method, not random variation).

- **User Experience Enhancements:**

- [ ] **Notebook Integration:** Extend the existing `pipeline_walkthrough.ipynb` (or create a new notebook) to allow easy switching between attribution methods. For example, have a dropdown or a variable at the top where one can select the method (Saliency, IG, DeepLIFT, etc.), and then the notebook will compute and plot that method's attribution for a given example. This will help the student or user interactively explore different explainability techniques on the same example and see the differences in real-time. Include visualizations such as plotting attributions on the nucleotide sequence or heatmaps if appropriate.
- [ ] **API Design:** In the code (perhaps a new module `src/xai.py`), design a clean API function like `explain(model, bacteria_seq, phage_seq, method="IG", target_class=1, **kwargs)` that returns the attributions for the chosen method. This function can handle the selection of method and any special parameters needed (e.g., number of IG steps, NoiseTunnel iterations, occlusion window size). Having a single function or class for this will simplify how the rest of the pipeline calls attribution methods (for example, the pipeline can loop over a list of methods to produce a comparative analysis).

Deliverables (Track B):

- A new module (e.g., `src/xai.py`) that contains implementations for the attribution methods and a unified interface to invoke them. The code should be well-documented, explaining how each method works at a high level (with references if helpful).
- A comparison report or notebook (Jupyter Notebook) that benchmarks and visualizes the methods. Expect to include charts of attribution scores and a summary table of metrics (e.g., a table showing pairwise Spearman correlations between methods, time per sample, etc.).
- An update to the pipeline or CLI to allow choosing an attribution method (for example, via config or CLI argument). By default, the pipeline might run a recommended method (or a couple of methods for comparison) as decided from the evaluation.

Acceptance Criteria:

- At least five attribution methods are implemented, tested, and produce reasonable output for both branches of the model (bacteria and phage inputs). Specifically, methods should run without errors and output attribution arrays of the correct shape for each input.
- The outputs of different methods can be compared. For example, if one method highlights a region around gene X in the bacterium as important, at least some other methods should also assign high importance to that region (unless a method is known to behave very differently). If one method gives completely random or uncorrelated results, there should be an investigation or explanation (maybe the method is not suitable for this model or requires parameter tuning).
- The evaluation notebook demonstrates that you have looked at at least a few examples and compared methods. There should be a clear summary of findings — for instance, you might conclude "Integrated Gradients and DeepLIFT tend to agree on top regions (Spearman ρ ~ 0.8), while Saliency is noisier and highlights broader areas. Occlusion finds similar regions as IG but at a coarser resolution and much slower. We recommend using Integrated Gradients with SmoothGrad as a default for its balance of precision and stability."

- A brief report (could be in the notebook or a separate markdown) is provided, recommending two default attribution methods (with parameters) for general use in the project, based on the comparisons. For example, you might choose Integrated Gradients (with 50 steps and baseline of all Ns) and Occlusion (with window size 100bp) as the two methods to include in final analysis, giving reasoning for their choice (e.g., IG for fine-grained insight, Occlusion for validation). This report ensures the knowledge from comparison is not lost.

Track C — Signal Processing and Region Extraction

Goals: Enhance how we extract important genomic regions from the raw attribution scores. Initially, a simple fixed-size sliding window approach was used (e.g., take the top scoring window of fixed length). This track aims to develop a more adaptive, signal processing-informed approach to identify peaks of importance in the attribution signals. By doing so, we hope to capture meaningful biological regions (like genes or regulatory elements) of varying lengths, rather than arbitrary fixed-size windows. The end result should be a procedure that, given an attribution score array for a sequence, returns a set of significant regions (with start/end positions on the genome) that likely correspond to biologically relevant features, with reduced false positives and more consistency.

Tasks:

- **Dynamic Peak Detection:**
- [ ] **Prominence-Based Peaks:** Implement a peak-finding algorithm on the attribution signal for each sequence. Instead of pre-defining window size, identify local maxima (peaks) in the 1D attribution score array. Use a prominence-based approach to define peaks: a peak's *prominence* is a measure of how much it stands out from the surrounding baseline [8]. You can use functions like SciPy's `signal.find_peaks` which supports finding peaks by prominence and other criteria. Determine suitable parameters (prominence threshold, minimum height, etc.) so that you catch significant attribution spikes while ignoring noise.
- [ ] **Adaptive Peak Width:** For each peak, estimate its width (for example, how far left and right the attribution stays high before dropping off). SciPy's `peak_widths` can measure the width of peaks at half-prominence or other relative heights [8]. Use this to allow regions of different sizes: a sharp peak might indicate a small motif, whereas a broad plateau might indicate a larger gene or region of importance.
- [ ] **Merge Close Peaks:** If two peaks are very close to each other (closer than a certain minimum gap), consider merging them into one region, especially if the valley between them isn't deep. This prevents selecting two nearly overlapping regions separately. Implement a rule: e.g., if peaks are within X bases or if the dip between them is less than Y% of their height, merge them. This can be done by scanning the peak list or even by adjusting the `find_peaks` parameters for minimum distance.
- [ ] **Multi-scale Detection:** To ensure robustness, run the peak detection at multiple smoothing levels or window sizes. For instance, you could smooth the attribution signal with different window lengths (or use a wavelet transform for multi-scale peak detection). If a region appears as a peak in several scales, it's likely a true signal. Practically, you might take the intersection of peaks found at (say) raw signal vs a slightly smoothed signal. Alternatively, run `find_peaks` with a couple of different parameter sets (one catching broader peaks, one catching finer peaks) and combine the results. The goal is to avoid missing a peak that just because it was narrow or broad under one setting.

- [ ] **Parameter Tuning:** The above methods will have parameters (prominence threshold, smoothing window, etc.). Use a small set of development examples to tune these. For example, manually verify on known cases (maybe the walkthrough example or others where we suspect certain genes) that the peak detection finds those regions. Adjust parameters to get a reasonable number of peaks (not too many false positives, not too few missing ones).

- **Thresholding and Region Selection:**

- [ ] **Threshold Strategies:** Implement different ways to threshold or select top regions from the attribution signal:
    - **Absolute Threshold:** e.g., select all positions above a certain attribution score (like >0.9 of max). This might be tricky since attributions aren't probabilities, but if normalized, >0.9 means very high importance.
    - **Percentile Threshold:** e.g., take the top 1% highest attribution values in the sequence as candidate regions, or consider positions above the 99th percentile of attribution. This is relative to each sequence's distribution, ensuring you pick the extreme attributions for that sample.
    - **Relative to Local Baseline:** For each peak region, compare its average score to the local surrounding region's average. You could compute a local baseline by taking a window around the peak and excluding the peak itself. If the peak's value is, say, 2-fold above the local baseline, mark it as significant. This helps adapt to cases where an entire sequence might have high attributions (perhaps the model is very confident and highlights a broad region) – a local measure would still identify the highest contrast sub-regions.
    Experiment with these approaches or even combine them (e.g., first identify peaks, then apply a threshold to filter out small ones).

- [ ] **Top-K Diverse Regions:** Often we may want a fixed number of regions per sample for downstream analysis (like BLASTing the top 3 regions). Implement a selection of the top K peaks by importance, but ensure diversity. For example, after picking the highest peak, you might skip any other peak that overlaps significantly with it or is very close, to ensure you cover different parts of the genome. This is akin to non-maximum suppression used in object detection. Define a "diversity constraint," such as requiring selected regions to be a minimum distance apart or on different sequence segments if possible. This will ensure we don't pick, say, five peaks that are all within one gene while another gene with slightly lower scores is ignored.

- **Region Post-processing:**

- [ ] **Refine Region Boundaries:** Once initial peak regions are selected, adjust their boundaries to capture full motifs. For each region, extend the boundaries to the nearest local minima around that peak. In practice, move left from the peak until the attribution value rises (i.e., you passed a valley) or perhaps until a certain drop percentage from the peak is observed; do the same to the right. This captures the context around the peak, which might include the entire gene or regulatory element that the model found important. Conversely, if a region is very large and flat, you might contract if needed, but expansion is usually more relevant if the peak picking was conservative.
- [ ] **Deduplicate Overlapping Regions:** Ensure that if two selected regions overlap or one is contained within another, they are merged or one is discarded. It's better to have one consolidated region per locus. As you merge, consider updating the "score" of the merged region (you could take the maximum attribution within it as the score, or perhaps the sum/area which might correlate with importance).

- [ ] **Feature Extraction from Regions:** For each region identified, calculate summary statistics that can be used later (for ranking or filtering). For example, compute the **mean attribution**

score in the region, the **max attribution score**, and the **area under the attribution curve** (which is essentially the sum of attributions in that region). These can serve as features to prioritize regions (a region with high area might be more important overall than one with a sharp single spike). Store these along with the region coordinates.

- **Validation and Visualization:**

- [ ] **Sanity Check Plots:** Create plots overlaying the identified regions on the attribution curves for several examples to ensure the algorithm is picking sensible regions. For each sample, plot the attribution score vs position, and mark the chosen regions (e.g., as colored blocks or vertical spans). This visual validation is crucial to tweak the method. The `visualisation.py` might have some functions to help with plotting attributions; extend or adapt those.
- [ ] **IGV Tracks Export:** For integration with genome browsers or sharing with biologists, output the attribution scores and regions in standard formats. For example, generate a BED file for each sample listing the coordinates of important regions (if the sequences have a reference genome coordinate, use that; if not, just use an index coordinate system for the sequence). Also, create a BedGraph or Wig file that contains a track of attribution scores across the genome. These can be loaded into IGV or UCSC Genome Browser if the sequences are aligned to a reference, or just for visualization. Ensure that the BED/BEDGRAPH uses the correct naming (if sequences are identified by an accession or name, use that as the chromosome name).
- [ ] **Cross-Method Comparison:** Using the results from Track B, compare the regions extracted from different attribution methods. For example, do Integrated Gradients and Saliency yield regions that overlap largely (compute the Jaccard index of sets of region coordinates for the same sample under two methods)? If a region is found by multiple methods, it's more likely to be a true positive. Summarize these findings: you might produce a table of overlap percentages or simply note that "on average, ~70% of the top 5 regions by IG overlapped with those by DeepLIFT" etc. If there are major discrepancies, that could inform which method is more reliable for region finding.

Deliverables (Track C):

- A module/file `src/peaks.py` (or integrated into `attribution.py`) containing functions for peak detection and region extraction (e.g., `find_peaks(attribution_array) -> list of regions`). This should be well-documented, with flexibility to adjust parameters (like prominence, thresholds, etc., possibly via the config file).
- Example output files for a few samples: e.g., `results/sample1/regions.bed`, `results/sample1/attribution.bedgraph`, which demonstrate the format and content of the region extraction.
- A Jupyter notebook showcasing the improvements of this dynamic approach vs the old fixed-window approach. This notebook should include plots of attribution signals with the new regions overlaid, and perhaps a comparison where the old method's fixed windows are shown too, illustrating how the new method picks regions more precisely. Include a brief analysis of any differences (for instance, "the fixed 1kb window captured only part of gene X, whereas the new peak-based method correctly identified the whole gene X region").

Acceptance Criteria:

- The new region extraction method should output a **reasonable number of regions** per sample (for instance, not hundreds of tiny fragments, and not just one giant region unless truly only one area is important). "Reasonable" might be on the order of a few regions for an interacting pair, depending on model behavior. This likely means your parameters are tuned so that noise doesn't produce peaks and real signals do.

- Regions identified should make **biological sense** upon inspection. This is subjective, but for example, if the model is known to look at phage tail fiber genes and bacterial receptor genes, the regions should often coincide with genes (or parts of genes) when checked against annotations. As part of acceptance, take a couple of known examples and verify that the regions correspond to actual genes or regulatory elements (you can cross-reference with GenBank features in Track D once that is implemented).
- The method is **robust across methods and runs**: if you run it on attributions from different XAI methods, it should still find largely overlapping regions (assuming the methods agree reasonably). If one method produces too scattered attributions, the peak finder should ideally ignore those or result in smaller regions, whereas for stable methods the regions will be clear. Also, running the pipeline on two different random seeds (for methods like SmoothGrad) should yield similar regions, indicating stability.
- The pipeline integration is done: i.e., after computing attributions in Track B, the Snakemake or CLI pipeline calls the peak detection to produce region outputs for each sample. These region outputs will feed into Track D (BLAST and annotation). The `top_regions` rule from Track A should now utilize this improved logic instead of a simplistic approach. A check: the final number of BLAST queries (regions) per sample is manageable (if too many regions are output, it might overwhelm BLAST or produce too much data, so likely we want to cap regions per sample to perhaps 5-10 top regions).
- Visually, when the student or user looks at the attribution plot and the highlighted regions, it should align with intuition – peaks are properly captured. If any obviously important-looking peak is missed by the algorithm, that would be a fail condition to address by adjusting parameters.

Track D — Biological Queries and Reporting

Goals: Turn the raw outputs of the model (important sequence regions) into meaningful biological insights. This involves taking the high-importance regions identified in Track C, running BLAST searches to find similar sequences or known genes, and then annotating those hits with biological information (gene names, functions, etc.). Ultimately, we want to produce human-readable reports that summarize what was found for each bacterium-phage pair, as well as an aggregate summary across all samples. The reports should highlight, for example, if certain genes or functions keep appearing in the important regions (which could suggest biological hypotheses, like a phage binding protein or a bacterial defense system being important for interactions).

Tasks:

- **BLAST Enhancements:**
- [ ] **Local vs Remote BLAST:** Update the pipeline to allow BLAST searches either via NCBI's remote servers or using a local BLAST+ database. In the `config.yaml`, have an option to choose `blast_mode: remote` or `blast_mode: local`. For local BLAST, the user should provide a path to a pre-downloaded nucleotide database (like `nt` or a custom database of phage/bacteria genomes). Installing BLAST+ in the Docker image (Track A) is important here. Ensure the pipeline can intelligently switch modes: for remote BLAST, use Biopython's `Bio.Blast.NCBIWWW` or direct HTTP calls to NCBI (with proper rate limiting); for local, call `blastn` or `tblastx` via subprocess with the appropriate arguments.

- [ ] **BLAST Options & Filtering:** Expose BLAST parameters via the config (e.g., e-value cutoff, maximum number of hits to return per query, etc.). After BLAST is run for a region, filter the results to retain only high-quality matches. Implement filters such as:
    - **Percent Identity**: e.g., require that the alignment identity is above a threshold (like >70% or >90% for a very strict match). This helps focus on closely related sequences.
    - **Alignment Coverage**: require that the query region is mostly covered by the alignment (for instance, the alignment length >= 80% of the query length, so we ignore tiny partial hits).
    - **E-value**: filter out hits with poor e-values (threshold can be 1e-5 or 1e-10 depending on how stringent we want to be).
    - **Strand**: if a hit comes from the reverse complement strand, note that (it might be relevant if the region is a reverse complement of something like a phage integration site, etc.). Ensure we capture the strand info from BLAST results.
      Combine these filters to produce a cleaned list of top hits for each region. If a region yields no good hits (which can happen for novel sequences), handle that gracefully (maybe mark as "no significant similarity found").

- [ ] **Multi-Hit Regions:** Sometimes a single query region might hit multiple locations in the database (especially if the region contains a common motif or a repetitive element). Decide how to handle this. For example, if a query hits 10 different phage genomes at the same gene, that's actually useful info (the gene is common in many phages). We should capture all relevant hits but perhaps group them by gene/function later. Ensure the BLAST parsing code can handle multiple hits per query region. Possibly, limit to the top 5 hits per query for manageability, but keep the data structure flexible.

- **Annotation and Interpretation:**

- [ ] **GenBank Feature Extraction:** Using the hit accessions from BLAST, retrieve detailed annotations. If using remote BLAST, NCBIWWW might give some info, but often it's useful to fetch the full GenBank record for the top hits. Utilize Biopython's Entrez utilities or `SeqIO` to fetch GenBank entries by accession ID [9] [10]. For each hit, extract key features:
    - **Gene or Protein Name:** Many GenBank CDS features have a `/gene` or `/product` annotation. For example, a hit might correspond to a gene with name "gp37" or product "tail fiber protein". Extract those if present.
    - **Protein ID:** If the hit is a coding sequence, get its protein ID (RefSeq or GenPept ID) which could be used to cross-reference other databases.
    - **Neighboring Genes:** To provide context, also extract features upstream and downstream of the hit (say ±5 kb or ±N genes). Sometimes knowing the neighborhood (e.g., the hit gene is next to a known integrase or in a certain operon) can be insightful. This could be optional but is valuable for interpretation.
      Create a structured summary for each hit: e.g., "Region X (bacteria position 1,200,000-1,250,000) matches *E. coli* gene *rhsA* (a toxin/antitoxin system protein) with 95% identity over 1,000 bp." Include the gene product description if available.
- [ ] **Functional Tagging:** To systematically analyze functions, map gene names or product keywords to broader categories. For instance, if a product is "tail fiber protein", tag it as "Phage tail fiber (host specificity)". If a gene is "cas3", tag as "CRISPR-Cas system (defense)". We might not build a full ontology, but at least create a list of keywords of interest (e.g., receptor, tail, capsid, integrase, restriction enzyme, etc.) and if any hit's description contains those, flag them. Optionally, for a more formal approach, use Gene Ontology (GO): tools like Blast2GO can retrieve GO terms for blast hits [11], but implementing that might be too heavy. Instead, perhaps use a simpler approach with Biopython or an online service to get GO terms for a protein ID, then

summarize GO categories. This step can help in finding common themes (like many hits related to “membrane proteins” or “viral structural proteins”).

- [ ] **Cohort-level Summary:** After processing all samples, aggregate the findings. For example, list the most frequent genes or functions that appeared in the important regions across different bacteria-phage pairs. If, say, multiple interactions highlight a phage tail fiber gene and a bacterial O-antigen biosynthesis gene, these might be key players in interactions. Create a table or list of such recurring features. Also, note whether they appeared in the phage sequence, the bacterial sequence, or both. This summary will be valuable for the report to draw general conclusions.

- **Report Generation:**

- [ ] **Per-Sample Reports:** Develop a template (possibly using Jinja2 or a notebook to HTML conversion) for an individual interaction report. For each bacterium-phage pair (especially those labeled as interacting "yes"), generate a report containing:
    - **Overview:** the names/IDs of the bacterium and phage, and the model’s prediction/confidence.
    - **Attribution Plot:** a figure showing the attribution scores along both genomes, with the top regions highlighted (from Track C). This gives a visual idea of where the model focused.
    - **Region Table:** a table listing the coordinates of the top regions for both the bacterium and phage, their attribution scores (mean/max), and any interesting patterns (like GC content or length).
    - **BLAST Hits:** for each region, list the top BLAST hit(s) after filtering, including gene name, organism, % identity, etc. If a region didn’t match anything, state "no significant hit".
    - **Feature Annotation:** if available, provide a brief description of what the gene does (e.g., "gene X is a known receptor binding protein in phages, which suggests this region may be involved in host recognition"). You can pull this from the GenBank annotation or UniProt if protein ID is available.
    - Aim to format this cleanly, possibly with sections for "Phage-side findings" and "Bacteria-side findings". Consider generating this as an HTML file or PDF for each sample for easy sharing. Using a templating engine (Jinja2) with the data gathered can automate this nicely.

- [ ] **Aggregate Report:** Create a high-level report that looks at all the samples together. Contents might include:

    - **Summary Statistics:** e.g., how many interactions had at least one identifiable gene hit? What proportion of phage important regions corresponded to known tail proteins? etc.
    - **Frequent Hits Table:** a table of the most common genes or functions that were found in Track D across samples, along with counts (e.g., "Tail fiber protein – found in 5 phage regions", "Type IV secretion system protein – found in 3 bacterial regions").
    - **Examples:** highlight one or two interaction pairs as case studies, possibly with an IGV snapshot or a diagram. For instance, show an IGV screenshot of a bacterial genome region with a prophage or receptor gene that was identified. (This might require pre-loading sequences into IGV manually and screenshotting, unless we automate a static image. If automation is complex, this can be a manual figure for the final report.)
    - **Method Comparison:** briefly include if different attribution methods (Track B) led to different biological findings. If, say, IG and DeepLIFT both pointed to the same gene, that increases confidence. If they differed, mention that and which method seemed more plausible.

    - **Conclusions:** a short text section summarizing what was learned: e.g., "The XAI analysis frequently identified phage tail fiber genes and corresponding bacterial surface polysaccharide synthesis genes, suggesting these are key to determining host range. This matches known biology that phages bind to bacterial surface receptors. We also discovered a few hypothetical proteins of unknown function that were flagged, which could be novel interaction factors."
      The aggregate report can be an extended Markdown or PDF document combining text, tables, and figures.

- [ ] **Export Data:** Ensure all the results are also saved in machine-readable formats in case further analysis is needed. For example, output a TSV or CSV file listing every region and its top BLAST hit and annotation. Also, have combined BED files for all regions found (one for phage side, one for bacteria side perhaps) so one can quickly view all important regions across the dataset. This will complement the human-readable reports.

Deliverables (Track D):

- An automated report generation system (could be a Jupyter Notebook that collects everything or a Python script using templates) that produces the per-sample HTML/PDF reports and the aggregate report. The code/template for this should be included in a `report/` directory. If using Jinja2 templates, include those files (e.g., `report/template_sample.html` and `report/template_summary.html`).
- The `blast_features_summary.tsv` (or similar) file containing all identified features from BLAST annotated with gene names, products, etc., in a structured format (columns like sample_id, region_coordinates, hit_accession, gene_name, product, %identity, e-value, etc.). This will be a key output for researchers to filter or query later.
- Example output reports for a couple of interactions (to show that the template populates correctly with real data).
- Updated CLI command `cli.py report` (or Snakemake rule `report`) that generates the above reports after all previous steps have run. Ensure this step depends on all necessary previous results (attributions, BLAST results, annotations, etc.) so that running the pipeline end-to-end includes report generation.

Acceptance Criteria:

- Running the pipeline end-to-end produces a **complete set of reports** without manual intervention. Each sample with an interaction has its own report file, and there is a global summary report. These should be found in the results directory (e.g., `results/reports/sample_<id>.html` and `results/reports/summary.html`).
- The content of the reports is accurate and comprehensible. Key points:
- Gene names and functions in the reports should be real and correctly match the BLAST hits (no placeholder or missing info for the majority of cases).
- The per-sample report should clearly pinpoint which regions were important and what they likely correspond to. A reader (with some biology background) should be able to follow the logic from model -> important region -> BLAST hit -> gene function.
- The aggregate report should highlight meaningful trends (if the pipeline found nothing but random hits, that might indicate an issue with attribution or region selection; ideally, patterns emerge such as common phage tail genes or common bacterial receptor genes).
- The BLAST process should not overwhelm with data. That is, after filtering, each region ideally has a handful of top hits listed, not pages of results. The filtering thresholds should be tuned such that we get the most relevant hits. If a region matched a repetitive element with hundreds

of hits, we either limit those or summarize them (e.g., "100 hits to transposases across many genomes, showing this region might be a common mobile element").
- The pipeline is flexible to use either remote or local BLAST. If internet is available and `blast_mode: remote`, it should fetch results from NCBI reliably (maybe with a delay between queries to respect rate limits). If `blast_mode: local`, assuming the user has the BLAST database downloaded and path set, the pipeline should use it and be faster. Test both modes with a small example to ensure they produce equivalent outputs.
- The entire reporting process should take into account runtime – e.g., if there are 100 regions to BLAST, make sure to either batch queries or handle sequentially with progress logging, and document how long it might take. (This may be more of a note, but a user should know if they need to download a large database or wait for an hour while BLAST runs on many queries.)

Cross-Cutting Tasks and Cleanup

These tasks are general improvements and cleanup to be done alongside the above tracks:

- [ ] **Archive Legacy Code:** Move any old or exploratory notebooks (except the main `pipeline_walkthrough.ipynb`) into a `notebooks/legacy/` folder. This keeps the repository clean and directs new users to the updated pipeline instead of outdated scripts. Clearly mark them as deprecated if they are not to be used with the new pipeline.
- [ ] **Repository Hygiene:** Remove unnecessary files like `.DS_Store` (macOS artifacts) from the repo. Ensure the `.gitignore` is up to date (it already should ignore `__pycache__/`, data files, results, etc., but double-check). This prevents clutter and accidentally committing large or private files.
- [ ] **Unit Tests:** Write minimal unit tests for critical utility functions and logic. This could include:
- One-hot encoding/decoding: test that a sequence with all nucleotides including ambiguous ones correctly one-hot encodes and then decodes back to the original [10] (where decoding is applicable).
- Data caching: test that saving and loading from cache (maybe with a small dummy sequence) preserves the content and shape.
- Dataset batching: if there's a custom Dataset class, test that indexing and length work as expected.
- Attribution combination: given a fake attribution tensor for a one-hot sequence (e.g., high score on 'A' channel at a position), ensure the combine_attributions function returns the correct 1D array highlighting that position.
- Peak finding: construct a synthetic attribution array with known peaks and valleys, and verify that your peak detection returns the expected regions.
  Use a testing framework like `pytest` for simplicity, and perhaps include a GitHub Actions CI configuration to run tests on each commit (this might be optional but is good practice). Even a small test suite will catch obvious bugs as the code evolves.
- [ ] **Sample Dataset for CI/Dev:** Create a very small example dataset that can run through the entire pipeline quickly (for continuous integration or just quick iteration). For instance, a few very short sequences (maybe 1000 bp bacteria, 200 bp phage) with a dummy model or the real model truncated. This could be synthetic or a downsample of real data. The point is to have something that runs in, say, a minute rather than hours, so that when making changes, one can verify the pipeline still works end-to-end. Store this mini dataset in `data/sample/` and allow the config to point to it. Also, you might include a flag in the pipeline to use a "fast mode" (like fewer IG steps, smaller occlusion window) for testing. Ensure the sample data is small enough to include in the repository or easily downloadable.

By addressing these cross-cutting items, we ensure the project remains maintainable and high-quality as we implement Tracks A–D.

Suggested Order of Work (Milestones)

While the tracks are listed separately, they interconnect. A suggested sequence to tackle them is:

1. **Pipeline Skeleton & Config (A1):** Start with Track A basics – get the config file, Snakemake workflow, and simple CLI in place. Initially, you can still call existing scripts (like gradcam_to_blast) from Snakemake rules to mimic the pipeline, then gradually refactor. Aim to have a pipeline that runs through the existing process (Grad-CAM on one sample to BLAST) in an automated way. Also set up the environment file.
2. **Implement Core XAI Methods (B1):** While pipeline is being set up, implement a few attribution methods (say Saliency, Integrated Gradients, DeepLIFT) and integrate one of them (e.g., IG) into the pipeline as a proof of concept. At this stage, focus on getting at least one method fully working end-to-end (so you can replace Grad-CAM with IG, for example).
3. **Dynamic Region Extraction (C1):** Replace or augment the fixed-window selection with the new peak finding approach. Test this on outputs from the method implemented in step 2. Iterate until the regions output look reasonable. This is important to do before running tons of BLAST queries, to avoid wasting time on bad regions.
4. **Basic Annotation & Reporting (D1):** Once you have some regions, implement the BLAST retrieval and a basic version of the annotation (maybe just gene name extraction) and generate a simple per-sample report. This doesn't have to be fancy initially – even a CSV output of regions and hit gene names is fine. The key is to ensure the pipeline can connect the dots from attribution -> region -> BLAST -> annotation. Try this on a few samples to see if the results make sense, and adjust thresholds as needed.
5. **Expand XAI & Finalize (B2/D2/A2):** Go back to Track B to add the remaining methods and complete the comparison notebook. With those insights, finalize which method(s) to use by default. Enhance the annotation step with more details or functional tagging now that you trust the regions. Add the Dockerization and make sure everything runs in the container. Finalize documentation and polish the report templates. Add any additional tests needed. In this phase, everything comes together for the final deliverable.

This order ensures that early on you have a working pipeline (even if using simpler methods), and you iteratively replace components with better versions. By the end, all tracks should converge into a coherent workflow.

Notes for Students

- **Understand the Pipeline First:** Use the `pipeline_walkthrough.ipynb` notebook to run through one example manually. This will give you an intuition for what each step is doing (e.g., how Grad-CAM was used, what the BLAST outputs look like). It's much easier to build or refactor code when you have seen the intermediate results and understand the goal.
- **Work Incrementally:** It's tempting to write a lot of code at once, but it's better to implement one piece at a time and test it. For example, after writing the caching step, run it on a small input and verify the output. Then move to the next step. The Snakemake rules can be tested individually with `snakemake -n -p <rule_name>` to see if inputs/outputs are lined up.

- **Determinism for Fair Comparison:** When comparing XAI methods (Track B), ensure that any randomness is controlled. Set random seeds for PyTorch, NumPy, etc., especially when using NoiseTunnel or random baselines. Document these in the output (for instance, if you generate a plot with SmoothGrad, note in the caption that 50 noisy samples were used with σ=0.1). This will allow you (and others) to reproduce your analysis exactly and trust the comparisons.
- **Coding Style:** Keep the code Pythonic and clean. Use functions to avoid duplication (for example, a lot of attribution methods share a pattern — you can have a helper that takes a method name and does common pre/post processing). Use list comprehensions and library functions where appropriate, but also prioritize clarity (a few extra lines of comments or straightforward logic is better than an overly clever one-liner when it comes to others reading your code).
- **Biology Context:** If you are not from a biology background, take some time to read about bacteriophage biology and terms you encounter (e.g., tail fiber, integrase, receptor, etc.). This will help in understanding the significance of the genes you find. For instance, if your pipeline flags a gene called "tolC" in bacteria, knowing that TolC is part of an efflux pump and could be a phage receptor might be insightful. Leverage online resources or ask a mentor about any unfamiliar terms.
- **Don't Reinvent the Wheel:** Many components here (like peak finding, BLAST parsing, etc.) have existing solutions. It's good to use libraries (Captum, SciPy, Biopython) rather than write from scratch, as long as you understand what they do. When in doubt, search for examples or documentation – for instance, Captum has tutorials on multi-input models, Biopython has examples of GenBank parsing, and Snakemake has plenty of workflow examples in bioinformatics. Learning to read documentation and adapt examples is a key skill.
- **Time Management:** Some tasks (like running BLAST on many regions, or comparing many XAI methods) can be time-consuming. Use small subsets for development. You can, for example, test your attribution methods on one or two samples rather than all 100, to iterate faster. Similarly, use the small sample dataset for quick test runs of the pipeline. Only scale up once things are working.
- **Documentation and Communication:** As you complete parts of the project, keep notes on what you did and why. This will help in writing the final report and also in discussions with your mentor. If something didn't work and you changed approach, note that too. Sometimes a "failed" method (e.g., if Saliency was too noisy to be useful) is worth mentioning, along with the reason, as it shows understanding of the problem.
- **Learning Outcome:** The goal of this project is not just to get results, but for you to learn about XAI, reproducible ML pipelines, and some bioinformatics. Don't hesitate to dive deeper into any of these areas if you find them interesting – for example, you might read the original Integrated Gradients paper [4] or a tutorial on Snakemake. This broader understanding will make the implementation easier and more rewarding. Good luck, and have fun exploring!

---

[1] [2] [3] [5] [6] [7] Captum · Model Interpretability for PyTorch
https://captum.ai/api/saliency.html

[4] [PDF] Explainability of Deep Learning for Genomic sequences - IS MUNI
https://is.muni.cz/th/lzelx/Explainability_of_Deep_Learning_for_Genomic_sequences_orig.pdf

[8] peak_prominences — SciPy v1.16.2 Manual
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.peak_prominences.html

[9] 03 - Parsing GenBank files - widdowquinn.github.io
https://widdowquinn.github.io/2018-03-06-ibioic/01-introduction/03-parsing.html

10 Dealing with GenBank files in Biopython - University of Warwick
https://warwick.ac.uk/fac/sci/moac/people/students/peter_cock/python/genbank/

11 Blast2GO: a universal tool for annotation, visualization and analysis ...
https://academic.oup.com/bioinformatics/article/21/18/3674/202517