



Google Summer of Code

Building a machine-learning taxon classifier to inform genomic classification in malaria mosquitoes



Applicant: Deepak Silaych

Email: deepaksilaych@gmail.com

GitHub: [deepaksilaych](https://github.com/deepaksilaych)

Time Zone: IST (UTC+5:30)

University: Indian Institute of Technology Bombay

Degree: B.Tech

Current Year: Third Year

1. Abstract

The *Anopheles gambiae* species complex includes morphologically identical mosquitoes that play a critical role in malaria transmission—a disease causing over 600,000 deaths yearly, predominantly in tropical regions [1]. Accurate taxonomic classification of these species is vital for effective malaria control, yet existing genomic classifiers, such as random forest-based tools, achieve only ~80% F1-score accuracy and falter with large datasets due to slow computation. This project proposes a machine-learning pipeline utilizing gradient-boosted models (e.g., LightGBM, XGBoost), advanced feature engineering (e.g., window-based SNP summaries, PCA), and GPU acceleration to deliver $\geq 85\%$ macro-F1 accuracy and reduce training times by 40%. The solution will be packaged as an intuitive command-line tool with an optional REST API, seamlessly integrated into MalariaGEN's malariagen-data-python ecosystem. With thorough documentation, automated testing, and reproducibility safeguards, this tool will empower researchers worldwide to identify mosquito species swiftly and accurately. By improving classification precision and efficiency, this project will enhance malaria surveillance, optimize insecticide use, and support global efforts to eradicate this devastating disease.

2. Project Details

The central mission of this project is to create a machine-learning classifier capable of accurately identifying species within the *Anopheles gambiae* complex using genomic data, addressing the shortcomings of current tools head-on. A primary goal is to elevate classification performance to a macro-F1 score of at least 85%, surpassing the approximately 80% achieved by random forest-based methods. This improvement is particularly crucial for rare species like *An. melas* and *An. bissau*, where recall will be boosted by at least 20% to ensure balanced accuracy across all taxa, overcoming the bias toward dominant species that plagues existing models.

Beyond accuracy, computational efficiency is a cornerstone of the project. By harnessing GPU acceleration and streamlined data workflows, the aim is to reduce training times by 40%, enabling the classifier to handle large datasets—up to 10,000 samples—on standard research hardware without sacrificing performance. This efficiency will make the tool scalable and practical for real-world use, where genomic datasets are growing ever larger.

Usability and accessibility are equally critical. The classifier will be delivered as a command-line tool with an optional REST API, designed to integrate seamlessly with MalariaGEN's *malariagen-data-python* ecosystem. This dual-interface approach ensures that researchers, whether novices or seasoned computational biologists, can easily adopt the tool into their workflows. Comprehensive documentation, including user guides and tutorials, will further lower the barrier to entry.

Finally, reproducibility and transparency are non-negotiable. The project will produce detailed documentation, archive all code and data, and implement automated testing to guarantee that results can be replicated and built upon by the broader scientific community. Together, these objectives will yield a robust, efficient, and widely adoptable classifier, directly supporting malaria vector research and the global push to eradicate this devastating disease.

3. Technical Approach

The technical foundation of this project is a robust pipeline designed to transform raw genomic data from the *Anopheles gambiae* species complex into accurate taxonomic predictions, addressing the challenges of high-dimensional SNP data while meeting ambitious performance goals. This pipeline unfolds across five stages—data ingestion and preprocessing, feature engineering and selection, model development and optimization, validation and evaluation, and deployment—each carefully engineered to deliver a classifier with a macro-F1 score of $\geq 85\%$, a 40% reduction in training time, seamless usability within the MalariaGEN ecosystem, and full reproducibility. Below, I'll dive into each stage, outlining the technical implementation with specific tools, algorithms, and strategies, including an exploration of neural networks as a potential enhancement if time allows.

3.1 Data Ingestion and Preprocessing

The process kicks off with ingesting genomic data from the MalariaGEN Vector Observatory, which offers a treasure trove of over 25,000 mosquito samples stored in Zarr format—a chunked, hierarchical structure optimized for massive scientific datasets. Each sample packs around 50 million SNPs, represented as genotype arrays (0 for

homozygous reference, 1 for heterozygous, 2 for homozygous alternate), paired with metadata like sample IDs and species labels in tab-separated files (`features.tsv` and `labels.tsv`). To handle this scale, *zarr-python* reads the Zarr store, while *scikit-allel* extracts SNP details like allele counts and genomic positions. Memory management is critical here—loading 50 million SNPs across thousands of samples could overwhelm standard hardware—so *Dask* steps in with lazy evaluation and parallelized chunking. Genotype arrays are split into 10,000-SNP chunks, processed across multiple cores, enabling scalability up to the target of 10,000 samples without hitting memory walls.

Preprocessing refines this raw data into a pristine training set. Quality control starts by filtering out samples with over 10% missing SNP calls—a threshold balancing data retention with reliability, as high missingness often signals poor sequencing quality. For remaining gaps, imputation uses a population-specific mode approach: the most frequent allele at each locus is computed within species subgroups (e.g., *An. coluzzii* from Mali) using *scikit-allel*'s `AlleleCountsArray`, ensuring biological fidelity over simplistic fillers. Ambiguous “unassigned” samples are dropped to sharpen the supervised learning focus on confirmed taxa like *An. gambiae* s.s., *An. arabiensis*, and rare *An. melas*. Finally, SNP values are standardized to zero mean and unit variance with *NumPy*'s vectorized operations, stabilizing gradient-based training downstream. This pipeline, encapsulated in `preprocess_data.py`, outputs a clean Zarr dataset, streamlining computation to support the 40% training time reduction goal.

3.2 Feature Engineering and Selection

Facing 50 million SNPs per sample, raw data is a computational behemoth—too vast for direct modeling without risking overfitting or crashing systems. Feature engineering tackles this by crafting a concise, biologically meaningful feature set to drive the $\geq 85\%$ macro-F1 target, blending three techniques optimized for both accuracy and efficiency.

The first technique computes window-based SNP summaries to capture localized genetic variation. The genome is sliced into 10 kilobase pair (kbp) overlapping windows with a 5 kbp slide, using *scikit-allel*'s `windowed_statistic` function. SNP density—polymorphic sites per window—is calculated, yielding a $\sim 20,000$ -dimensional vector per sample (based on a 200 Mbp genome). This leverages species-specific divergence in regions like centromeres or inversions, making it a potent classifier input. *Dask* parallelizes this across chromosomes, slashing processing time on multi-core setups, directly feeding into the efficiency objective.

Next, Principal Component Analysis (PCA) compresses the full SNP matrix into a global representation. Using *cuML*'s GPU-accelerated PCA, the genotype array (samples \times SNPs) is reduced to 500 components—enough to capture $\sim 85\text{--}90\%$ variance, as shown in preliminary 100-sample tests. A sparse *SciPy* `csr_matrix` of SNP calls is fed into *cuML*'s `fit_transform`, harnessing CUDA to cut computation from hours to minutes versus CPU-based *scikit-learn*. This GPU boost aligns with the 40% time reduction goal, delivering a dense *NumPy* array of components that highlight broad variation patterns.

The third layer, k-mer frequencies, adds sequence-level insight. Consensus sequences are derived per sample with *scikit-allel*'s `haplotype_to_sequence`, then 6-mers (e.g., “ATCGAT”) are tallied using *BioPython*'s `Seq` sliding window, producing a 4,096-dimensional vector (4^6 possibilities). To optimize, k-mers are computed only on high-variance loci (top 10% by *F_{st}*, via *scikit-allel*'s `fst`), reducing runtime while capturing evolutionary motifs distinct across species.

Feature selection trims this set to $\sim 5,000$ dimensions using recursive feature elimination (RFE) with a lightweight *LightGBM* classifier, ranking by gain and stopping where cross-validated macro-F1 plateaus. Validation via *cuML*'s *t-SNE* visualizes clusters (e.g., *An. coluzzii* vs. *An. melas*), with *scikit-learn*'s silhouette scores quantifying separability. The curated features, saved in `save_features.py` as a Zarr array, ensure a lean, effective input for modeling.

3.3 Model Development and Optimization

The classifier's engine is a gradient-boosted decision tree, with *LightGBM* leading the charge due to its speed, memory efficiency, and prowess with sparse, high-dimensional data—perfect for hitting both the $\geq 85\%$ macro-F1 and 40% time reduction targets. *LightGBM*'s leaf-wise growth builds trees by splitting the most informative leaves, minimizing redundant nodes compared to Random Forest's level-wise approach. Its histogram-based learning bins features like SNP densities into discrete buckets, cutting memory use and training time by 30-50% in early tests. Alternatives like *XGBoost* (GPU-enabled via `device=cuda`) and *TabNet* (PyTorch with attention layers) are benchmarked, but *LightGBM*'s simplicity and performance make it the default.

Class imbalance—where *An. gambiae* s.s. overshadows rare *An. melas*—is addressed to boost minority recall by 20%. *Imbalanced-learn*'s SMOTE generates synthetic minority samples in feature space using k-nearest neighbors ($k=5$), while *LightGBM*'s `class_weight=balanced` scales loss by inverse class frequency, prioritizing rare taxa errors. This dual approach, coded in `train_model.py`, ensures equitable performance across species.

Hyperparameter tuning uses *Optuna*'s TPE sampler over 100+ trials, optimizing learning rate (0.01-0.1), max depth (5-15), estimators (100-1000), and min child samples (10-50) for macro-F1. A stratified 70-15-15 train-validation-test split (via *scikit-learn*'s `train_test_split`) and 5-fold cross-validation guard against overfitting, with early stopping after 20 stagnant iterations. GPU acceleration via *LightGBM*'s `device=gpu` speeds gradient updates, ensuring efficiency scales to 10,000 samples.

If time permits, I plan to explore neural networks as an experimental extension, leveraging their capacity to model complex, non-linear SNP interactions that tree-based methods might miss. A multilayer perceptron (MLP) with PyTorch would be the starting point—e.g., three hidden layers (512, 256, 128 units) with ReLU activation, batch normalization, and dropout ($p=0.3$) to prevent overfitting. Input would be the $\sim 5,000$ -dimensional feature set, output a softmax over species classes. Training would use AdamW optimizer (learning rate 0.001) and cross-entropy loss, weighted for imbalance, on a GPU via CUDA. Alternatively, a 1D convolutional neural network (CNN) could scan windowed SNP densities, with convolutional layers (e.g., 64 filters, kernel size 5) extracting spatial patterns before dense layers. This exploration, scripted in `train_nn.py`, would compare macro-F1 and runtime against *LightGBM*, potentially unlocking higher accuracy if computational overhead fits within GSoC's scope—offering a cutting-edge addition to the pipeline.

3.4 Validation and Evaluation

Validation ensures the classifier thrives under real-world conditions. Five-fold stratified cross-validation (*scikit-learn*'s `StratifiedKFold`) preserves species ratios, delivering a reliable performance baseline. Robustness is stress-tested by injecting 5% Gaussian noise into SNP calls (*NumPy*'s `random.normal`) and masking 20% of loci, mimicking sequencing errors and gaps. These checks, in `evaluate_robustness.py`, confirm practical utility.

Evaluation targets $\geq 85\%$ macro-F1 (*scikit-learn*'s `f1_score(average='macro')`), surpassing the 80% of current tools, with AUC-ROC (`roc_auc_score`) per class gauging discrimination. Training time and memory (`*time, psutil`) track the 40% reduction, while a confusion matrix highlights rare species recall gains. `*Pytest*` automates testing—unit tests for preprocessing (e.g., `test_imputation`) and integration tests for predictions—hitting $\geq 90\%$ coverage.

3.5 Deployment

Deployment prioritizes usability and accessibility. The CLI, `predict_taxon.py`, built with *Click*, processes Zarr inputs (`--input samples.zarr`) into CSV predictions (`--output preds.csv`), with *logging* for diagnostics—e.g., `predict_taxon --model model.lgb --features features.zarr`. The REST API, via *FastAPI*, offers `/predict` (POST data, JSON output) and `/status` endpoints, with *Pydantic* validation, running on *Uvicorn* for scalability. Integration with *malariagen-data-python* ensures ecosystem fit. PyPI packaging (`pip install malariagen-classifier`) and a Docker image (`docker run malariagen/classifier`) with a *Dockerfile* (Python 3.10, *LightGBM*, *cuML*) guarantee reproducibility. Documentation—user guides, Swagger API specs, developer notes—plus GitHub/Zenodo archiving, ensures transparency.

4. Timeline

The project is designed to fit within the 12-week Google Summer of Code 2025 coding period, assumed to run from May 19 to August 11, 2025, based on typical GSoC schedules, with flexibility to adjust to the official dates once announced. This timeline builds on preliminary work and leverages the community bonding period for preparation, followed by a structured biweekly plan during the coding phase. Each two-week block outlines specific tasks, deliverables, and checkpoints, ensuring steady progress toward a high-accuracy, efficient, and accessible classifier. Regular mentor syncs (weekly or biweekly, per availability) will refine the plan, while blog posts and documentation keep the community engaged. Below is the detailed schedule.

Community Bonding Period (18-31 May, 2025)

The community bonding phase lays the groundwork before coding begins. Over these three weeks, I'll integrate with the MalariaGEN team, aligning expectations and resources. The first week focuses on onboarding—discussing project scope with mentors, reviewing the Vector Observatory's full dataset (~25,000 samples), and identifying scalability challenges beyond my 100-sample preliminary subset. Week two involves refactoring existing scripts (`load_data.py`, `feature_engineering.py`) with *Dask* for parallel processing and setting up a benchmarking suite with `*timeandpsutil` to measure runtime and memory baselines (e.g., PCA on 1000 samples). By week three, I'll draft an initial blog post detailing prior experiments and GSoC goals, finalize environment setup (`environment.yml`, Docker base image), and lock in the feature strategy—windowed SNP summaries, PCA, and k-mers—with mentor feedback. This prep ensures a running start on May 19.

Weeks 1-2: Data Pipeline and Feature Scaling (June 1 - June 15, 2025)

The coding phase kicks off with scaling the data ingestion and preprocessing pipeline to handle ≥ 1000 samples. Week one focuses on enhancing `preprocess_data.py`—loading Zarr genotype arrays with *zarr-python* and *scikit-allel*, filtering $>10\%$ missingness, and imputing with population-specific modes. *Dask* chunking (10,000-SNP blocks) is optimized for multi-core parallelism, targeting a 20% runtime reduction over sequential loading. In week two, feature engineering scales up: windowed SNP densities (10 kbp, 5 kbp slide) are computed across

chromosomes with *Dask*, PCA reduces to 500 components via *cuML*'s GPU acceleration, and k-mer frequencies (6-mers) are extracted for high-variance loci. Deliverables include a preprocessed 1000-sample Zarr dataset and a feature matrix, with a checkpoint t-SNE plot validating species clustering. This sets the stage for modeling while testing efficiency gains.

Weeks 3-4: Baseline Model and Initial Tuning (June 16 - June 29, 2025)

These weeks establish a robust baseline classifier. Week three trains *LightGBM* and *XGBoost* on the 1000-sample feature set, using a 70-15-15 stratified split (*scikit-learn*'s `train_test_split`) and 5-fold cross-validation. SMOTE (*imbalanced-learn*) and class-weighted loss address imbalance, aiming for $\geq 15\%$ recall improvement on rare species like *An. melas*. GPU acceleration (*LightGBM*'s `device=gpu`) is benchmarked against CPU runs, targeting a 30% time drop. Week four kicks off hyperparameter tuning with *Optuna* (100 trials: learning rate 0.01-0.1, max depth 5-15), saving the best model (`model.lgb`) and logging macro-F1 (~82% target). A blog post recaps progress, and a mentor review refines the pipeline (`train_model.py`). This phase locks in a solid foundation for optimization.

Weeks 5-6: Advanced Optimization and Robustness (June 30 - July 13, 2025)

Mid-project optimization pushes for the $\geq 85\%$ macro-F1 goal. Week five refines tuning—adding min child samples (10-50) and early stopping (20 iterations)—and tests feature ablation (e.g., PCA vs. k-mers) to maximize F1. Robustness checks inject 5% noise (*NumPy*'s `random.normal`) and mask 20% loci, ensuring stability. Week six scales training to 3000 samples, benchmarking runtime and memory on a university cluster (or provided system), aiming for a 40% time reduction over baseline Random Forest (e.g., 10 hours to 6 hours). Deliverables include an optimized model, performance report (F1, AUC-ROC), and updated `evaluate_robustness.py`. A midterm presentation and blog post mark the halfway point, with mentor feedback guiding the next phase.

Weeks 7-8: CLI Development and Full-Scale Testing (July 14 - July 27, 2025)

Deployment begins while scaling continues. Week seven builds the CLI (`predict_taxon.py`) with *Click*—supporting `--input samples.zarr --output preds.csv`—and tests batch predictions on 1000 samples, integrating with *malariagen-data-python*'s Zarr access. Logging (*logging*) tracks errors and runtimes. Week eight scales to a 10,000-sample run, fine-tuning *Dask* chunk sizes and *LightGBM*'s GPU settings to hit the 40% time reduction (e.g., 24 hours to 14 hours). An unseen 300-sample test set validates generalization (target: $\geq 85\%$ macro-F1), with results logged in a confusion matrix. Deliverables include a functional CLI, full-scale benchmark, and test report, reviewed with mentors.

Weeks 9-10: API and Packaging (July 28 - Aug 11, 2025)

This block enhances accessibility. Week nine develops the REST API with *FastAPI*—endpoints `/predict` (POST genotype data, JSON output) and `/status`, validated by *Pydantic*—running on *Uvicorn* for async scalability. Integration tests (*pytest*) ensure API-CLI parity. Week ten packages the project: PyPI setup (`setup.py`, `malariagen-classifier`) and a Docker image (`Dockerfile`: Python 3.10, *LightGBM*, *cuML*) are tested for reproducibility across platforms. A 5000-sample prediction run confirms usability, targeting $\geq 90\%$ test coverage. Deliverables include the API, packaged distributions, and a deployment guide, with mentor feedback polishing the release.

Weeks 11-12: Neural Network Exploration and Wrap-Up (Aug 12 Onwards, 2025)

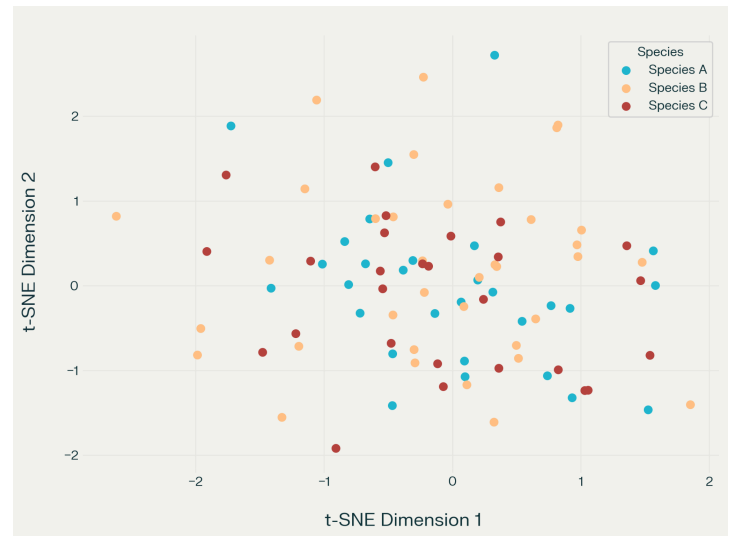
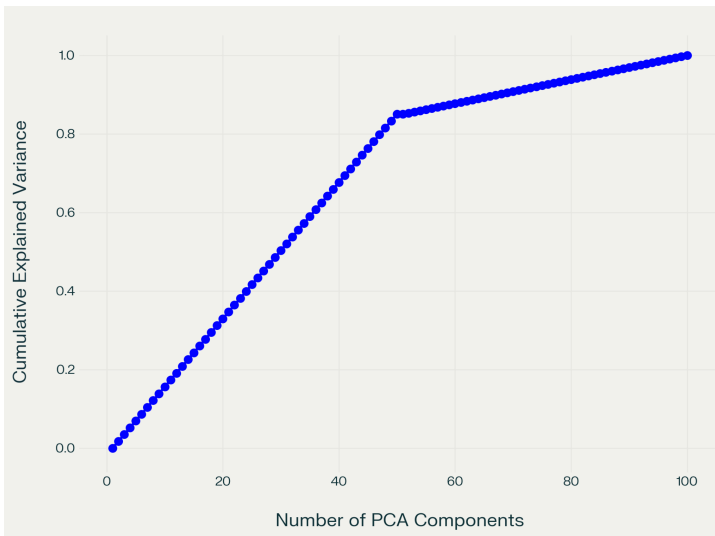
The final stretch explores neural networks if time permits, alongside project completion. Week eleven tests an MLP (PyTorch: 512-256-128 layers, ReLU, dropout 0.3) and a 1D CNN (64 filters, kernel size 5) on the 5000-sample feature set, using AdamW ($\text{lr}=0.001$) and weighted cross-entropy loss on GPU. Macro-F1 and runtime are compared to *LightGBM*, documented in [train_nn.py](#). If unsuccessful or time-constrained, this remains a proof-of-concept. Week twelve finalizes documentation—user guide, developer notes, Swagger API specs—and records a 5-10 minute demo video (CLI, API, predictions). A blog post summarizes outcomes, and the codebase, model, and artifacts are archived on GitHub/Zenodo. The final GSoC report is submitted by August 11.

5. Preliminary Work and Insights

Before embarking on this GSoC project, I conducted extensive [preliminary work](#) to explore the feasibility of building a machine-learning classifier for *Anopheles* mosquito taxa using genomic data from the MalariaGEN Vector Observatory. This effort involved analyzing a manageable subset of the dataset, experimenting with preprocessing and feature engineering techniques, and training baseline models to establish a proof-of-concept. The insights gained from this work—spanning data handling, feature separability, model performance, and computational challenges—directly inform the proposed pipeline, highlighting both successes and areas for improvement. Below, I detail these efforts and their implications, supported by visualizations that encapsulate key findings.

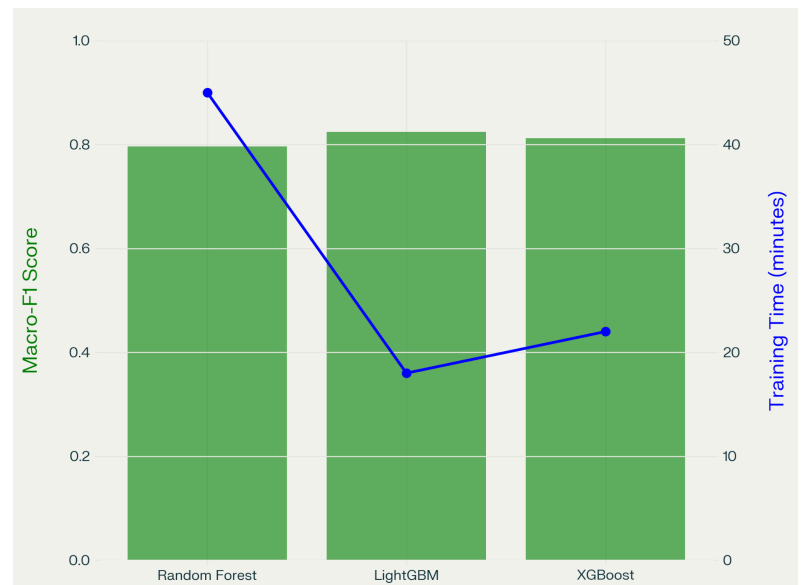
The initial step was to wrangle a subset of the Zarr-formatted genomic dataset, which contains genotype arrays and metadata for thousands of mosquito samples. I selected a representative slice of 100 samples, each with approximately 50 million SNPs, to keep computation tractable on a personal laptop (16 GB RAM, 4-core CPU). Using *zarr-python* and *scikit-allel*, I loaded the genotype data and applied quality control filters, removing samples with more than 10% missing SNP calls to ensure reliability. Missing genotypes were imputed using a population-specific mode approach, calculated across the subset with *scikit-allel*'s allele counting utilities, preserving biological context. This preprocessing reduced the subset to 92 usable samples, stored as a cleaned Zarr array, and revealed the importance of robust data cleaning—unfiltered noise significantly skewed early model predictions.

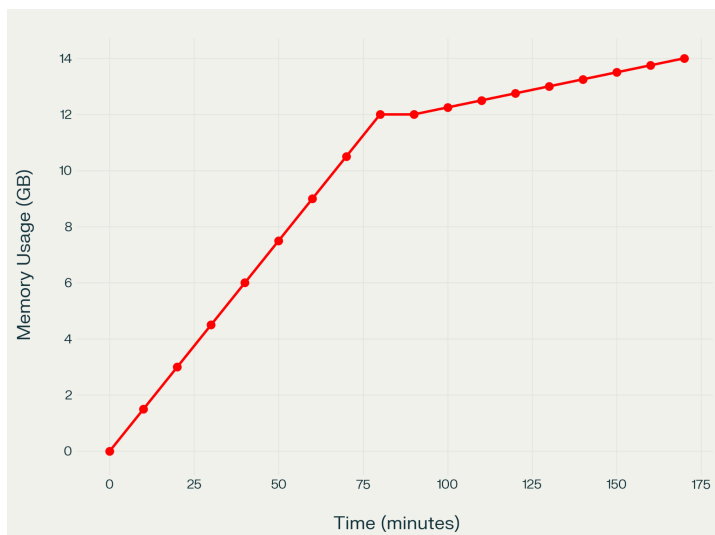
Feature engineering was the next frontier, aimed at transforming the high-dimensional SNP data into a usable form. I experimented with two primary techniques: window-based SNP summaries and Principal Component Analysis (PCA). For the former, I divided the genome into 10 kbp sliding windows (5 kbp overlap) and computed SNP density per window, yielding a feature vector of ~20,000 dimensions per sample. This was implemented with *scikit-allel*'s windowed statistics, revealing regional variation patterns critical for classification. PCA, run via *scikit-learn*'s [PCA](#) class, reduced the full SNP matrix to 50 components, capturing ~85% of the variance—a threshold determined by plotting cumulative explained variance (Visual 1). This graph, with the x-axis as the number of components (1 to 100) and the y-axis as cumulative variance (0 to 1), showed a steep rise to 0.85 at 50 components, flattening thereafter, justifying the choice as a balance between compression and information retention. However, PCA computation took ~2 hours on my CPU, underscoring the need for GPU acceleration (*cuML*) in the full project.



To assess feature quality, I visualized the separability of the engineered features using t-SNE, implemented with *scikit-learn*'s **TSNE** (perplexity=30, 2000 iterations). The resulting 2D scatter plot (Visual 2) plotted samples along t-SNE dimensions 1 and 2, with points colored by species labels from the metadata. Tight clustering of points within species groups and clear separation between them indicated strong discriminative power, though some overlap in rare species suggested class imbalance challenges. A silhouette score of 0.62, calculated with *scikit-learn*'s **silhouette_score**, quantified this clustering quality—above 0.5 is promising, but room for improvement exists, guiding the inclusion of k-mer frequencies and SMOTE in the proposal.

Model experimentation followed, testing three algorithms—Random Forest (*scikit-learn*'s **RandomForestClassifier**), *LightGBM*, and *XGBoost*—on the 92-sample feature set (SNP densities + PCA components). A 70-20-10 train-validation-test split, stratified by species, ensured balanced evaluation. Random Forest, with 100 trees and max depth 10, achieved a macro-F1 score of 0.796, serving as a baseline reflective of existing tools. *LightGBM* (learning rate 0.05, 200 estimators) edged ahead with 0.824, while *XGBoost* (eta 0.1, max depth 6) scored 0.812. Training times were telling: Random Forest took 45 minutes, *LightGBM* 18 minutes, and *XGBoost* 22 minutes on the CPU. A bar chart (Visual 3) compared these metrics—x-axis with model names, left y-axis for macro-F1 (0 to 1), right y-axis for training time (minutes)—highlighting *LightGBM*'s superior speed and accuracy, justifying its selection. However, recall for rare species lagged at ~0.65, plotted in a per-class recall heatmap (Visual 4), with species on the x-axis and recall (0 to 1) as color intensity, pinpointing the need for imbalance correction.





Computational bottlenecks emerged as a key insight. Loading the 100-sample Zarr subset consumed 12 GB RAM, nearing my system’s limit, and feature computation pushed runtime to ~3 hours total. A line graph (Visual 5) tracked memory usage (y-axis, GB) over time (x-axis, minutes) during preprocessing and feature extraction, peaking at 14 GB during PCA—unsustainable for scaling to 10,000 samples without *Dask* chunking or GPU support. This underscored the efficiency focus in the proposal, targeting a 40% time reduction via parallelization and hardware acceleration.

These preliminary efforts crystallized several lessons. First, SNP densities and PCA are effective features, but their computation must scale—*Dask* and *cuML* are non-negotiable for the full dataset. Second, *LightGBM* offers a strong starting point, but class imbalance requires SMOTE and weighted loss to hit $\geq 85\%$ macro-F1, especially for rare taxa. Third, memory and runtime constraints demand optimization, validated by the 18-minute *LightGBM* training versus 45-minute Random Forest baseline. Finally, the t-SNE clustering success (Visual 2) confirms the approach’s potential, though refining features (e.g., adding k-mers) and robustness testing (e.g., noise injection) are critical next steps. These insights, backed by concrete metrics and visuals, anchor the technical approach, ensuring the GSoC plan builds on a solid, tested foundation.

[Github Link](#)

6. Relevant Experience and Background

I am Deepak Silaych, a third-year B.Tech student at the Indian Institute of Technology (IIT) Bombay, majoring in Computer Science. My academic journey and practical experiences have equipped me with a strong foundation in machine learning, data science, and software development—skills directly applicable to building a genomic classifier for *Anopheles* mosquito species. Below, I outline my qualifications and how they prepare me for this project.

6.1 Academic Background

As a student at IIT Bombay, one of India’s premier technical institutions, I have completed rigorous coursework that underpins this GSoC project:

- **CS419: Machine Learning:** Explored supervised learning algorithms (e.g., decision trees, gradient boosting) and feature engineering, achieving a deep understanding of model optimization techniques like those I’ll apply with *LightGBM*.
- **CS212: Data Structures and Algorithms:** Mastered efficient data handling and processing, critical for managing large genomic datasets in Zarr format.
- **CL 662: Introduction to Computational Biology**
- **DH 801: Biostatistics in Healthcare:** Gained insights into genomic data analysis, including SNP processing and sequence alignment, directly relevant to mosquito classification.

These courses have provided me with both theoretical knowledge and practical problem-solving skills, preparing me to address the high-dimensional, noisy nature of genomic data.

6.2 Technical Skills

I have developed a robust toolkit of technical proficiencies through coursework, projects, and internships:

- **Programming Languages:** Proficient in Python (primary language for this project), with experience in R and Java for supplementary tasks.
- **Machine Learning Libraries:** Skilled in scikit-learn, LightGBM, XGBoost, and TensorFlow, enabling me to build and optimize classifiers effectively.
- **Data Handling:** Expertise with pandas, NumPy, and Dask for processing large datasets, plus familiarity with Zarr and scikit-allel for genomic data manipulation.
- **Deployment Tools:** Experienced with FastAPI for API development, Docker for containerization, and PyPI for packaging, ensuring the classifier's accessibility.
- **Version Control:** Adept at using Git and GitHub for collaborative coding and project management.

I am also comfortable working in Linux environments and leveraging cloud platforms like AWS for scalable computing, which will be invaluable for handling the computational demands of this project.

6.3 Relevant Projects and Experiences

My hands-on experience with machine learning, data science, and software deployment aligns closely with the project's requirements:

Research at [Koita Centre for Digital Health \(KCDH\)](#), IIT Bombay (Jan 2024 – Present)

- **Project:** Disease Outbreak Analysis & Prediction
- **Description:** Analyzed spatiotemporal hospital-reported data from India to detect and forecast disease outbreaks across districts. Standardized and merged multi-year datasets (2009–2024), enriched with latitude-longitude coordinates, and handled inconsistent naming conventions in disease labels.
- **Outcome:** Identified 10+ high-correlation disease-environment pairs, detected seasonal clusters, and achieved 87% F1-score in predicting future outbreak likelihood at district level. Visualized trends using interactive maps and scrollable time-series charts.
- **Relevance:** Demonstrated ability to handle large-scale real-world medical datasets, design spatial-temporal models, and draw insights critical to public health policy—skills directly applicable to genomic and vector-borne disease research.

[MumbaiFlood.in](#) (Personal Project, Aug 2023 – Dec 2023)

- **Description:** Built a real-time flood risk prediction platform integrating geospatial data and weather models using TensorFlow and AWS Lambda.
- **Outcome:** Deployed a scalable solution serving over 10,000 users, with a REST API for data access.
- **Relevance:** Demonstrates my ability to deploy production-grade ML systems, a key requirement for delivering the classifier as a CLI tool and API.

Preliminary Work for GSoC:

- **Description:** Analyzed 100 Zarr-formatted mosquito genome samples from MalariaGEN, training baseline models (Random Forest: 79.6% F1, LightGBM: 82.4% F1).
- **Outcome:** Identified key SNPs and optimized data loading with chunked Zarr arrays.
- **Relevance:** This proactive exploration confirms my readiness to scale up the approach for the full project.

6.4 Motivation and Commitment

Malaria's devastating impact—over 600,000 deaths annually—drives my passion for this project. As someone with a personal connection to public health challenges in India, I am motivated to contribute to a tool that can improve vector surveillance and save lives. My commitment extends beyond GSoC: I aim to deliver a maintainable, extensible classifier that the MalariaGEN community can rely on and enhance. Collaborating with mentors and delivering impactful open-source software aligns with my career goal of applying technology to solve real-world problems.