Team 9 Defense Report

CSCE-439-500, November 2025

Dakota Pound, Graham Dungan, Hexin Hu, Mihir Santosh Sanjay, and Trenton Gray
Computer Science and Engineering
Texas A&M University
College Station, Texas

***Abstract*—This paper describes Team 9's development of a feature-based, LightGBM machine learning model using dynamic ensemble selection to detect malware.**

I. Source Code

- **Github - TSDES Defender**: This is the source code for our most recent model, the TSDES defender.
- **Github - Golden Scar**: This was the main development suite for creating TSDES Defender.
- **Github - Tilted Towers**: This is the frontend website for the Tomato Town automatic malware collector.
- **Github - Tomato Town**: This was a script for automatically collecting malicious files from various sources on the internet (MalwareBazaar, MalShare).

II. Introduction

This paper discusses Team 9's approach to creating an ML antivirus for the class CSCE-439-500. The report contains four sections,

1) **Methodology** - This section describes the 'why'. It briefly describes our team's rationale for our decisions while making our model (i.e. why we chose DES, why we created tiered featurization).
2) **Training Process** - This section describes the 'how'. It explains how we implemented and tested the features presented in **Methodology.**
3) **Implementation Details** - This section describes how our team implemented tiered featurization and speculative prediction alongside our model.
4) **Extras** - This section contains extra work we did while developing our model.
5) **Conclusion** - This section concludes the paper and gives Team 9's final thoughts on defense.

The model we discuss in this paper is the Tiered Speculative Dynamic Ensemble Selection (TSDES) model. This model contains five sub-models, `T0`, `T1`, `T2`, `GENERALIST`, `DES`. If this model cannot featurize a binary in time, it will send the features it has collected to one of the 'tier' models, `T0`, `T1`, `T2`. If the model featurizes an entire binary, it will send the features to the `GENERALIST` model and `DES` model. Both models make a prediction, but only `DES`'s prediction is used. Both models are queried because if `DES` (a slower model, by virtue of it being an ensemble), cannot finish in time, the faster `GENERALIST` model will be ready to provide its prediction instead. The following report will describe how we created this model and system.

III. Methodology

This section entails an explanation of our model design choices and rationale. While we will discuss reasoning and evidence as to why we chose certain models, frameworks, etc., a more in-depth technical explanation of the concepts mentioned is available in Section IV. Our team began this project with having very little, if at all, no experience with Machine Learning. As such, many of our initial decisions were made spuriously with the intent of discovering if this choice was good or poor later down the line.

A. Model Selection

Our first question to ask as a team was 'What dataset should we use'. Our team wanted something modern and large, and we were naturally drawn to feature-based models by virtue of us being beginners. Feature-based seemed like it had a lower barrier to entry (as one can control the features, and it is ML instead of DL) and would be most conducive towards us learning about ML. We initially looked at `EMBER2024` [1], however, we wanted something more modern and less likely to be chosen by other teams. We soon found `EMBER2024` [2] and fulfilled our goals for having a large, modern, feature-based dataset. It should be noted that `EMBER2024` came with tools for training a LightGBM model, so we also used this as one of our earlier models. We verified LightGBM was the correct choice in subsection III-B. We had solved the question of 'what dataset should we use', and now the question became 'how should we use this dataset'.

B. Unsupervised Learning

Our team knew that training a LightGBM model on the entirety of `EMBER2024` (over 4.68 million samples) would be both computationally infeasible and would not fulfill the ≤ 1 GB RAM requirement for the competition. However, since we are new to both Machine Learning and Cybersecurity, our team was not familiar with picking representative samples from datasets nor what makes a sample representative. After some research, we found the paper '*A comprehensive investigation of clustering algorithms for User and Entity Behavior Analytics*' [3] to be immensely helpful for giving us a direction. This paper proposed DBSCAN and HDBSCAN as means of unsupervised learning- i.e., we could use these scanning methods to not only find representative samples, but also find relationships among our data (this paper also touches on K-means algorithms which was used extensively as a tool for both analysis and training).

To make scanning computationally feasible over EMBER2024's 2568 features, we had to find a way to reduce the number of samples. We first decided to perform a hierarchical sampling- weekly, by class (malware/benign), and by malware family. However, we still needed a way to reduce the dimensionality of the samples as well. Because of this, we initially reduced the samples with PCA. However, we decided to use UMAP because it is non-linear and better maps "intricate patterns and clusters"[4], which pairs well with our objectives of performing DBSCAN/HDBSCAN.

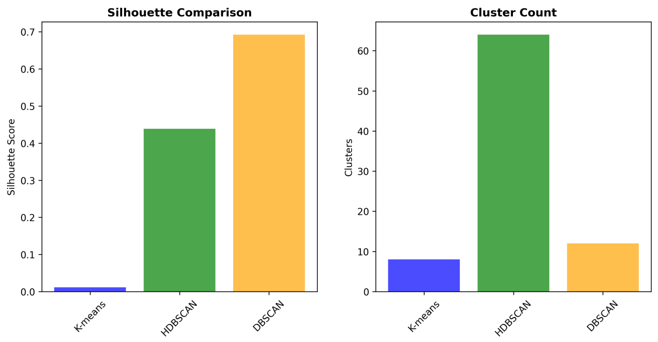


Figure 1: Silhouette scores and cluster counts for K-means, DBSCAN, and HBSCAN.

Our initial scans with DBSCAN and HSBSCAN yielded mixed results as shown in Figure 1- DBSCAN provided a few very tight clusters and a lot of unexplained noise, while HDBSCAN provided a great number of comparatively less tight clusters and less noise.

Our team had to decide how we would pick samples from these scans, so we decided to train multiple models. Each model would be trained on a combination of samples- some from HDBSCAN, some from DBSCAN, with variations in noise.

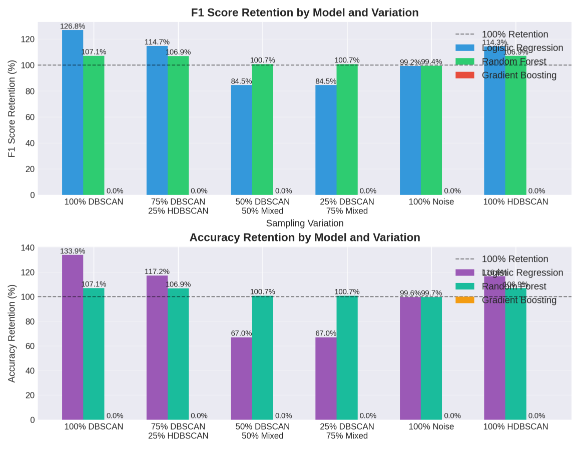


Figure 2: Different model performance for different DB-SCAN/HDBSCAN sampling rates.

As seen in Figure 2, our team tested a Logistic Regression and a Random Forest alongside a Gradient Boosting model

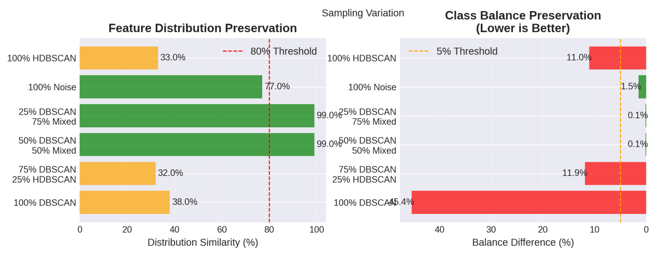


Figure 3: Model retention from sampling from different scans.

(not depicted within this figure) to determine which models best responded to variations in samples of scans. While 100% DBSCAN, 72% DBSCAN, and 25% HDBSCAN, and 100% HDBSCAN performed well in terms of accuracy and F1 score, they failed to accurately represent the distribution of EMBER2024's features and class balance, as seen in Figure 3. As such, our team went forward with a two-tier sample of 50% HDBSCAN and 50% DBSCAN. We will refer to the subsample of the EMBER2024 dataset using DBSCAN/HDBSCAN as the 'culled' dataset. We will go into more details of the specifics of sampling in Section IV.

C. Training and Hyperparameter Selection

While not directly present in Figure 2, our initial training with LightGBM demonstrated that it, on average, was more accurate with the EMBER2024 dataset than a normal random forest or logistic regression. As such, we decided to continue forward with the LightGBM model.

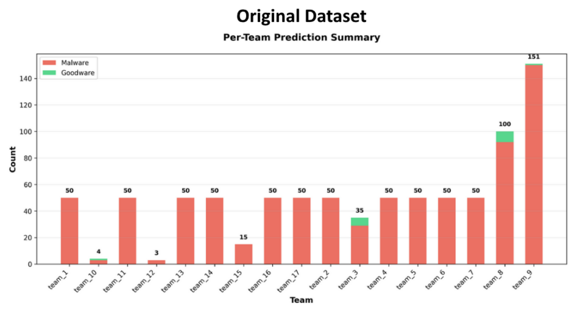
Continuing with our mindset of being novices- we did not know the best way to tune a machine learning model. As such, some cursory research [5] led us to pursue hyperparameter selection for our model. These results, while brief, will be discussed in subsection IV-D.

To ensure that our new culled dataset was still representative, we painstakingly trained a LightGBM model on the full EMBER2024 dataset. We then used the models trained on the full and culled datasets to make predictions about peer binaries, provided via the class. As seen in Figure 4, the TPR fell about 3% from the full to the culled model, with team 5 and team 8 largely to blame. Ultimately, it is seen that our model did not experience too much loss from the culling.

D. Experts and Routing

In seeing how a change in sampling drastically affected the performance of two specific teams' binaries (team 5 and team 8), we became interested in computing the drift between other team's samples and our model's data. As such, we began investigating how certain teams were thwarting our model by computing the KS distance between our data and their features, as seen in Figure-5,

Eventually, this led to a couple of conversations with Professor Botacin, where he recommended looking for a way to estimate 'similarity' between files. When we showed him how we computed drift for all teams, and noted how Team 5 had suspiciously non-drifting features except for one, Professor



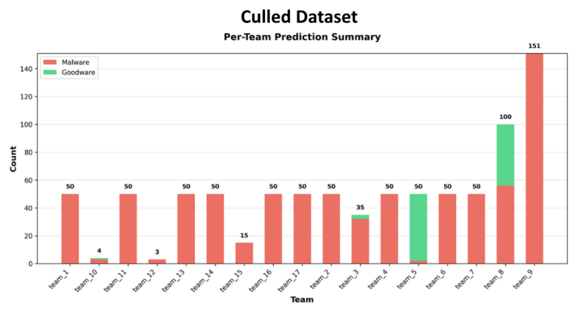


Figure 4: Performance of the full and culled LightGBM models on peer binaries.

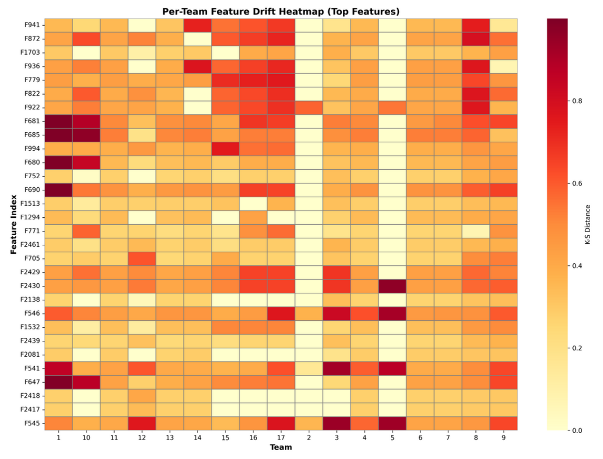


Figure 5: Drifting features between the culled LightGBM model dataset and Team 5 binaries.

Botacin recommended we simply remove the features that are convincing our model that the file is a goodware. This simple suggestion would lead us towards an export-oriented approach for the remainder of crafting out model.

He remarked that our model's accuracy wouldn't change much in its current state, and instead, the only way we could thwart evasive samples is to train specific models (experts) to counteract them. After some independent exploration, our team came upon k-Nearest Oracle Union (KNORA-U) as a form of dynamic ensemble selection (DES). Ultimately, in conjunction with our generalist model (what was recently our culled model, now slightly tweaked), we could greate a swath of other models that collectively vote on a file.

Our team came up with the following experts, as seen in Table-I. Tier models/experts were not included in this list, as they are not a part of DES. Note that we originally had 33 experts, but a few were removed due to introducing too much noise.

Table I: Experts by ID

ID	Expert (short name)
C-NOISE	Outlier/Noise specialist
F-HDR	Header-only
F-HIST	Byte/byte-entropy histograms
F-IMP	Imports-only
F-SECT	Sections-only
F-STR	Strings-only
P-PROXY	Proxy-domain cluster expert
P-PROXY-REC	Proxy high-recall
P-ROB-HS	Proxy robust (Header+Sections)
P-ROB-SECT	Proxy robust (Sections-only)
R-HIENT	High-entropy/packer suspect
R-HIREC	High-recall specialist
R-LOWFPR	Low-FPR specialist
R-NORME	Normal-entropy
R-PE32	PE32 specialist
R-PE32+	PE32+ specialist
R-SIGNED	Signed-binary specialist
R-UNSGN	Unsigned-binary specialist

While mostly self-explanatory, we created these experts by first forming groups;

1) **Cluster Group**: Groups formed out of the clusters from the previous scanning step.
2) **Feature Group**: Experts solely focusing on certain features of a file. As mentioned earlier, Profession Botacin recommended we 'drop' features that are misguiding our model.
3) **Peer Group**: Experts trained on 'proxy domains' of `EMBER2024` data via K-Nearest Neighbors.
4) **Rule Group**: Experts that focused on rules such as high entropy, signatures, etc.

With these experts, we began fine-tuning them (adjusting parameters, sample sizes, or completely changing them) until we thought we had a representative swath to choose from.

Our team realized that we needed to be conservative with sample training sizes for the router, as the KNN index would grow rapidly (initially around 400MB) for larger sample sizes. In order to follow challenge expectations, we had to greatly decrease our dsel holdout (more mentioned in the training process) to prevent this, thus costing us some accuracy with our router.

We eventually found that the router was not providing the boost in accuracy we wanted for how slow and large it was. As such, we sought optimization methods for increasing its speed. We applied the following techniques (more details in the training process)

- FIRE-DES [6] for pruning of experts with a confidence threshold.
- Distance-Weighted Voting such that closer neighbors contribute more to competence scores.

- Overall Local Accuracy (OLA) computing a continuous competence score based on an expert's probability outputs.

Among some other, less notable logical optimizations Since we introduced a number of new techniques for optimizations, we performed a parameter sweep (grid-based, informed by previous metrics of running the predictor) to find the best settings for the router. One of the most subtle yet most powerful optimizations we did was 'boosting', where we found if we boost the GENERALIST, P-PROXY-REC and R-NORME models, we saw a 4% increase in accuracy for peer binaries.

E. Tiered Featurization and Speculative Prediction

Despite our optimizations to our experts and our routers, some binaries continued to thwart us not due to the limitations of our experts' training, but rather the technical capacities of our binary parsers, featurizers, and predictors.

The EMBER2024 THREMBER Python library comes with a feature extractor. However, the THREMBER featurizer has a subtle bug that will time out any time it encounters a file with an Authenticode signature. Viewing featurize.py line 958 (line 5 in the following segment) attempts to iterate through an object called CERTS. The code can be seen below.

```
# ...
# Check if cert is self-signed
certs = signed_data.certificates
if len(certs) > raw_obj["chain_max_depth"]:
    raw_obj["chain_max_depth"] = len(certs)
    for cert in certs[:-1]:
        if cert.issuer == cert.subject:
            raw_obj["self_signed"] = 1
# ...
```

CERTS is an object that stores certificates; however, it is not naturally iterable. So, this script will error and fail to collect signatures whenever Authenticode signatures are involved. We saw a measurable decrease in errors after fixing this bug.

Another optimization we created was making our own copy of the EMBER2024 binary parser. Some teams, like Team 5, would stuff their data directories full of thousands of imports and other information. To avoid this slowing down parsing and featurization, we ended up creating something called 'tiered parsing and featurization'. This will be explained in the section on implementation details, but to put it briefly- our team breaks binaries into 'tiers'. Tiers are groups of features from easiest to some of the hardest/longest. A featurizer will attempt to parse each tier at a time. However, if they can't finish all tiers, the tiers of their previous features will be evaluated with an expert built for those features. This optimization allows us to pick and choose which sections we want to parse and how we parse them.

This notion of being productive about featurizing extended to our prediction step as well. DES is slow and large, by convention. Our GENERALIST model (an improved version of the culled model from earlier) is much faster and smaller, but less accurate. Our team found it productive to do a 'speculative prediction'- the DES model and the GENERALIST model will attempt to predict a file at the same time. The GENERALIST will win most of the time (by virtue of it being smaller). But, instead of instantly returning the outcome of the GENERALIST model, the script will continue trying to predict with the DES model. If the script runs out of time before DES finishes, it will return the already computed GENERALIST prediction. If DES finishes before the script runs out of time, then it will return the DES prediction. This speculative prediction ensures that a prediction is always made; however, more accurate predictions are made when DES has long enough to run. Once again, more information on these improvements is provided in the implementation details section.

Overall, these two improvements are what made it possible for our model to maintain its performance (with some losses) within the 5-second time window of this challenge and were the final components of the TSDES defender model.

F. Methodology Conclusion

Our methodology has saw us through the beginnings of understanding machine learning to optimizing the technologies that we use to parse, featurize, and predict files. We sought pre-sampling for unsupervised learning so we could pick a computationally feasible subset of EMBER2024 to train a model on. Once we started experimenting with drift and started recognizing how our model was being thwarted, we looked towards experts (DES and KNORA-U) to make our model more robust. Once we realized the limitations of DES, we looked towards our own parsers, featurizers, and prediction schemas to make progressive optimizations that allowed our model to sit within the 5-second threshold of this challenge. We will now continue to the next Section IV where we will discuss the technical implications of the training process and the creation of our final model TSDES.

IV. Training Process

The following section describes the technical process of our model. Explanations for why we chose these things can be found in the previous section, III .

A. Pre-Sampling

We began with a two-pass pre-sample from EMBER2024. The objective of this was to downsample the 48GB EMBER2024 dataset to a 500k pre-sample. Our initial pre-sampling scripts employ a streaming approach to avoid using too much RAM, iterating EMBER2024's .JSONL files in chunks of 10,000. During this extraction, we also collapse NONE, empty, or list values into strings.

Once we have collected the raw EMBER2024 data, we stratified the samples into week $\times$ class (i.e. benign, malware) $\times$ family (malware family, provided by EMBER2024) stratum. Inside each of these strata, we performed a Neyman allocation by first splitting the strata by class (attempting to maintain the 50/50 goodware/malware split), resulting in a week $\times$ class bin, and then using Shannon entropy as a proxy for variability inside the bin. Neyman's rule wants variability $N_h \times \sigma_h$, so

Shannon's entropy serves as a measure for how 'mixed' each stratum is.

After counting the families in each (week, class) bin, we built a per-bin mapping of the top $k = 50$ families and assigned any other families to OTHER. This ensures that any extremely rare families (i.e., smaller families) do not break our stratification.

Once these bins were created, we calculated a 'floor' (the minimum number of samples allocated for every family) to ensure that every malware family is covered via min_per_family $= \max(5, \lfloor n_h/(2 \cdot \text{num\_families}) \rfloor)$, where n_h is the bin's budget.

We then performed a second pass, this time performing a weighted reservoir sampling inside of each (week $\times$ class $\times$ family) bin. This time, for each substratum, we kept a reservoir whose size is the same as the substratum's quota. We then put samples into the reservoir, giving each a weight dependent on:

- A higher weight for rarer families in the (week $\times$ class), family_freq $= c_f/N_h$, where c_f is the count of samples in a family f and N_h is the number of samples in the bin.
- A higher weight to encourage diversity based on the hash of the bin's key, $1/(1 + \text{hash\_val})$.

This resulted in a weight function $w = \frac{1}{\sqrt{\text{family\_freq}+10^{-6}}} \times \frac{1}{1+\text{hash\_val}}$. Using this weight function, we sampled from our bins.

This concluded the pre-sample, resulting in a split of 60% test, 20% train, and 20% DSEL holdout (for later routing). We chose a sample size of 500k because we wanted to maintain around 100k samples for the DSEL holdout, to keep things computationally feasible, and we will be sampling again after the DBSCAN and HDBSCAN scanning. This pre-sampling resulted in a 300k / 100k / 100k split.

B. Scanning

In alignment with our goals of unsupervised learning, we first loaded our train pool (100k) to keep things computationally light (instead of doing the test and DSEL holdouts as well). From the features in the train pool, we compressed the 2568 EMBER2024 features to 50 dimensions using UMAP. After manual parameterization, we found that N_NEIGHBORS=15 and MIN_DIST=0.1 worked best using a Euclidean metric and a fixed seed of 42 (don't panic!) for reproducibility. Dimensionality reduction drastically improved our DBSCAN/HDBSCAN times, especially when we wanted to increase sample sizes for better representation.

We then performed DBSCAN and HDBSCAN on our samples.

- **DBSCAN**: For DBSCAN we automatically picked the sensitivity from the 85th percentile of the k-nearest neighbor curve ($k = 16$). Much of the remainder of DBSCAN's parameters and behavior were promoted from the paper 'A comprehensive investigation of clustering algorithms for User and Entity Behavior Analytics' [3].
- **HDBSCAN**: We ran a hierarchical DBSCAN to handle varying distances and give a membership probability per train pool.

Once scanning finished, instead of running scans on the test or DSEL holdouts, we instead performed UMAP (to reduce dimensions to be similar to the train set) and then used k-NN voting to assign DBSCAN and HDBSCAN labels to these samples. This prevents the number of times we have to run scanning and ensures a similar distribution of samples within clusters across our holdouts.

This concludes our scanning, as we then had the same train / test / DSEL split, now with DBSCAN and HDBSCAN labels for sampling during training.

C. Sampling from Scans

Now that we had our train / test / DSEL split with DBSCAN and HDBSCAN labels, we could perform a final sample for training. As mentioned in the methodology, we wanted to perform a two-tier sampling of these scans. Since HDBSCAN resulted in broader, larger clusters, while DBSCAN resulted in smaller, tighter clusters, we effectively used DBSCAN as seeds for selection inside of HDBSCAN clusters. We first made buckets from clusters. In this sampling, we would treat HDBSCAN as the main clusters, and DBSCAN as subclusters of HDBSCAN. We then separated noise from the clusters and reserved 10% of the total training budget for noise and outliers. As before, we stratified by week, class, and family so that everything isn't from one period or class.

We then return to the HDBSCAN clusters, where we use a square-root rule: Biggest clusters get more sample allocations, but because of the square root, smaller clusters still get representation as well.

Then, inside of each HBDSCAN cluster, we split its quota proportionally by DBSCAN subcluster size. Effectively, this is where we combine the DBSCAN and HDBSCAN clusters. Within the HDBSCAN/DBSCAN clusters, we compute distances to the bucket centroid in UMAP space, where we sample from 70% of the 'core' (closest to the centroid) and 30% from the 'border' (farthest away). Figure-6 demonstrates a simplified visual for this approach.

We can conclude sampling from scans with a new train / test / DSEL split of approximately 270k samples (96k, 87k, 86k). It should be noted that subsample sizes do matter greatly- if a selected sample from a scan is 99% of the size of the scan, then one is not effectively capitalizing on the properties that the scan found in the data. i.e., a 50% subsample would matter a great deal more because it culls data that is unrepresentative/useful. The new dataset size is effectively 33% / 33% / 33% to maintain the sizes of the test and DSEL splits.

D. Training and Hyperparameter Selection

As mentioned previously, all models created for this project are LightGBM. This was decided initially because EMBER2024 natively uses LightGBM, but we later found through varying DBSCAN/HDBSCAN sampling ratios that LightGBM performed better than a Regression or Random Forest model (see III).

First, we will discuss how we trained our first model using the train / test / DSEL split. This initial model had been initially

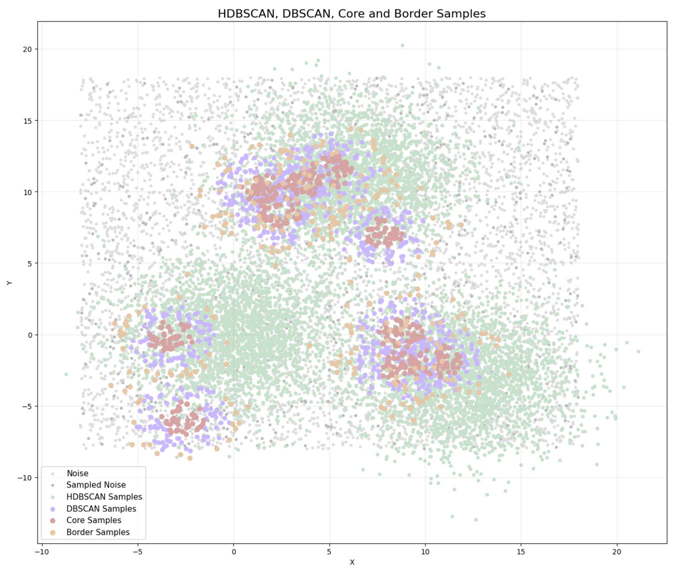


Figure 6: A mock diagram demonstrating sampling from DBSCAN/HDBSCAN.

referred to as the 'culled' model, in reference to the fact that it was culled from the `EMBER2024` dataset, but it will now be referenced as the `GENERALIST` model.

To be in training, we pre-allocated `float32` shape to load the `EMBER2024` feature dimension to avoid repeated allocation. Many of the most complex steps in training were primarily focused on making the training process happen efficiently within 32GB of RAM. After this, we build a `sha256` → `row_index` placement for each raw `EMBER2024` `.jsonl` entry. We early-exit as soon as we've iterated through all `.jsonl`, as we've now collected all of the data we need for training.

The training procedure was very straightforward. We put our data into `X_train` and `Y_train` for training and `X_val` and `Y_val` for testing. We simply call `lgb.train(...)`, collecting elapsed time, best iterations, and validation AUC/binary_error.

For hyperparameter selection, we effectively repeat the training process for a larger search space of parameters. Our team decided to go with random search for hyperparameter selection, as mentioned in subsection III-C, given evidence that random search can be faster than grid search [5]. Table-II highlights these search parameters,

Table II: LightGBM Hyperparameter Search Space

Hyperparameter	**Values used in random search**
num_iterations	{100, 200, 300, 500, 1000}
learning_rate	{0.01, 0.05, 0.1, 0.2}
max_depth	{-1, 3, 5, 7, 10}
num_leaves	{31, 64, 127, 255, 511}
min_data_in_leaf	{20, 50, 100, 200}
bagging_fraction	{0.7, 0.8, 0.9, 1.0}
bagging_freq	{0, 1, 5}
feature_fraction	{0.7, 0.8, 0.9, 1.0}
lambda_l1	{0.0, 0.1, 1.0, 10.0}
lambda_l2	{0.0, 0.1, 1.0, 10.0}
min_gain_to_split	{0.0, 0.1, 0.5, 1.0}

For each sampled config (a random permutation of these values), we ran a K-fold cross-validation and trained with our typical LightGBM model. All permutations were evaluated on max AUC and minimial error. The difference in training parameters for the `HYPER` model (the one selected through random hyperparameter search) can be found in Table-III, compared to the `GENERALIST` model. The `HYPER` saw a marginal increase in performance AUC, as seen in Table-IV during training.

Table III: LightGBM parameters: `HYPER` vs `GENERALIST`

Hyperparameter	`HYPER`	`GENERALIST`
num_iterations	1000	500
learning_rate	0.1	0.1
max_depth	5	-1
num_leaves	255	64
min_data_in_leaf	50	100
bagging_fraction	1.0	0.9
bagging_freq	5	1
feature_fraction	0.7	0.9
lambda_l2	0.0	1.0

Table IV: Performance Comparison: `HYPER` vs `GENERALIST`

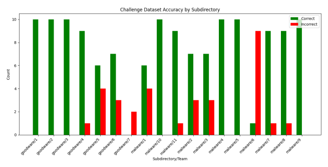
Metric (Validation)	`HYPER`	`GENERALIST`
AUC (final)	0.9983800	0.9984947
AUC (best)	0.9983806	0.9984968
Binary Log-Loss (final)	0.0692616	0.0494727
Binary Log-Loss (best)	0.0692616	0.0461300
Binary Error (final)	0.0158175	0.0156333
Binary Error (best)	0.0155642	0.0155642

More importantly, we then used `HYPER` and `GENERALIST` to evalutate `challenge` and `peer` binries provided through the class. Unfortunately, `HYPER` performed worse than `GENERALIST` by approximately 1%-2% in both sets, as seen in Table-V

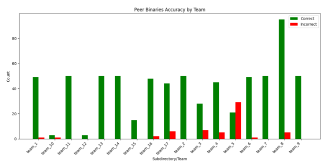
Table V: Accuracy Comparison on Challenge and Peer Sets

Model	**Challenge Accuracy**	**Peer Accuracy**
GENERALIST	81.40%	92.47%
HYPER	80.23%	90.49%

As such, our team continued to use the `GENERALIST` model without random hyperparameter selection. `GENERALIST`'s performance (without applying the 5-second timeout) with the `challenge` and `peer` sets can be seen in Figure-7a and Figure-7b, respectively.



(a) `GENERALIST`'s performance on the `challenge` binary set.



(b) `GENERALIST`'s performance on the `peer` binary set.

Figure 7: `GENERALIST`'s performance across the `challenge` and `peer` binary sets.

E. Expert Training

As mentioned earlier, our team found that we needed to go with an expert-driven approach to increase the strength of our model. The full list of experts and their purposes can be found in subsection III-D. All experts were trained with a LightGBM and settings consistent with the `GENERALIST`.

1) Cluster Experts: While we originally created multiple cluster experts (based off of various clusters found in the DBSCAN/HDBSCAN step), we found that these introduced a lot of noise and were generally unhelpful. We only kept `C-NOISE`, trained on DBSCAN noise samples. Training involved loading the test / train split from subsection IV-C, selecting all samples labeled as -1 (noise). The model was trained, as per usual, on the full features of this subset.

2) Feature Experts: We used five feature experts, their names and specialties can be found in Table-VI For each of these experts, we created feature masks using `feature_utils.apply_feature_mask` which slices the relevant feature columns for each expert. Once again, training was the same as the `GENERALIST` model.

Table VI: Description of Feature Experts

Expert	Description
F-HDR	Operates only on PE header attributes.
F-SECT	Uses structural and semantic section features (entropy, size, counts).
F-IMP	Focuses solely on import table hashing and API-related features.
F-STR	Uses string-related statistical features.
F-HIST	Relies on byte-level histograms and distribution-based signals.

3) Rule Experts: We used six feature experts which specialize on semantic subsets of the data (file types, feature groupings). The descriptions of these experts can be found in Table-VII.

Table VII: Description of Rule Experts

Expert	Description
Architecture-specific experts:	
R-PE32	Trained only on 32-bit samples.
R-PE32+	Trained only on 64-bit samples.
Signing-focused experts:	
R-SIGNED	Operates on cryptographically signed binaries; uses `scale_pos_weight = 0.5` to discourage false positives.
R-UNSGN	Operates on unsigned binaries.
Obfuscation-based experts:	
R-HIENT	Trained on high-entropy (packed/obfuscated) samples.
R-NORME	Trained on low-entropy samples, also using `scale_pos_weight = 0.5`.
Operating-point expert:	
R-LOWFPR	Trained on the full dataset but with `scale_pos_weight = 0.3`, heavily biasing the model toward low false-positive operation.

Since rule experts were slightly different than most experts, they had a different training procedure. For each rule, a filter was applied to `EMBER2024` samples to extract that satisfying condition (is it signed, is it unsigned, etc.). This also invovled creating ad-hoc subsets of the larger dataset, as not all files naturally fulfilled the requiresments of these rule-based experts. We then combined default LightGBM parameters with any rule specific overides (`scale_pos_weight = 0.3`) and trained as per usual.

4) Peer Experts: Peer experts are experts that specialize on samples provided via the `peer` sample set. It is important to emphasize that peer experts **were not** trained on raw `peer` binaries. Alternatively, the peer experts were trained on a 'proxy domain'. To collect samples for the peer experts, our algorithm loads peer features separately and uses KNN to identify the nearest training samples (50 neighbors, Euclidean). This ensured that we were not directly training on peer samples while still having a larger and similar corpora of training data. Descriptions of the experts that we created can be found in Table-VIII. Much like the cluster experts, a feature mask was created for the respective feature-masked peer experts and was trained on the proxy domain using the same LightGBM settings as per usual.

Table VIII: Description of Peer Experts

Expert	Description
P-PROXY	Full feature set; serves as the baseline peer-oriented classifier.
P-ROB-HS	Uses header and section features only; improves robustness against feature removal affecting imports and strings.
P-ROB-SECT	Relies exclusively on section-level features; maximizes resilience under heavy feature reduction.
P-PROXY-REC	Full feature set but trained with `scale_pos_weight = 2.0` to bias toward higher malware recall.

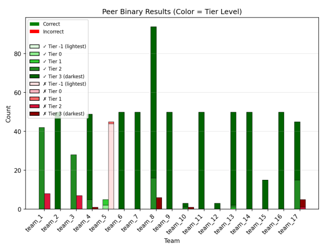
5) Tier Experts: As we will discuss in subsection V-A, files have been broken into four tiers for featurization. If the file can't be featurized in time, a tier expert/model will evaluate the features collected from that tier. There are three tier experts, `T0`, `T1`, and `T2`, however, this section will only discuss `T0` and `T1`. `T2` was created with the intention of thwarting `challenge` and `peer` samples, much like how the peer experts behave. An explanation of `T2` can be found in the next section, IV-E6. The descriptions of `T0` and `1` can be found in Table-IX.

Table IX: Description of Tier Experts

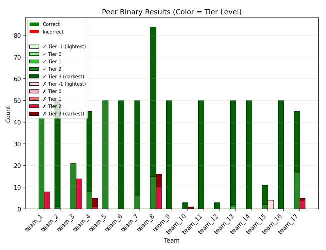
Tier	Description
T0 (Metadata-Only)	Uses ∼1,445 lightweight metadata features extracted from basic PE analysis. Extremely fast extraction, intended for early, low-cost triage.
T1 (Histogram Tier)	Extends T0 with byte-level histograms and entropy distributions, totaling ∼1,957 features. Designed to capture coarse structural and content distribution patterns.

Tier experts were trained much like the feature experts- the feature masks were just the tiers that they represented. Training and parameters for training are the same as per usual.

6) Peer Tier Experts: The purpose of creating peer tier experts was because Team 5 performed import stuffing- such large volumes of data resulted in the parser taking too long. Given our timeout of 2.95 seconds set for the tier featurizer, Team 5 samples and samples like Team 5 would always be evaluated by the `T2` model. The word 'peer' comes from the fact that we are using proxy domains to specifically target malware that has thwarted previous versions of our malware. As such, `T2` was trained much like the peer samples were- using KNN to create a proxy domain. The current `T2` used in `TSDES` was trained on 50% random data from the train holdout, and 50% from a proxy-domain containing files from Team 5 and `challenge` goodware 1 and goodware 2. Figure-8a and Figure-8b demonstrate the impact of `T2` before and after its peer training on the `peer` set.



(a) `T2`'s performance before peer training.



(b) `T2`'s performance after peer training.

Figure 8: Performance of `T2` before and after peer training.

F. DES with KNORA-U

Given our trained experts, we want a way to allow them to vote on malware based on their competency with the file. This is what we call our router, and is how we allow this voting to occur. This section will describe how we created our router with KNORA-U and how we used it to make predictions.

1) Building a Router: Our router first loads our experts. Some experts require feature masks which are specified in metadata associated with their files. Then, the DSEL set (the holdout that we have not used for any training or testing until this point, created in subsection IV-C) is loaded. The DSEL set is split into a competence set (80%, think of this as a training set) and a calibration set (20%). The function `build_competence_map(...)` takes in the experts, DSEL features, and the competence subset. This function then returns compentence features and labels. Then, for each expert, the DES builder applies a feature mask (if necessary), uses the expert to predict on all of the samples in the DSEL holdout. The program then saves the model's predictions and their probabilities which will later be used to continuous OLA competence.

Next, the router builder computes a KNN region-of-competency index for each expert. This is accomplished by performing KNN on the DSEL features to determine if a given sample is 'locally similar' to a certain neighborhood in DSEL.

The router then saves its state to be used in prediction. It should be noted that, due to the size of the KNN global map, the DSEL sample size had to be decreased from 86K to 9K to fit within the 1 GB constraint of the competition.

2) Using a Router: Making predictions with the router is performed in two stages.

1) **Stage 1 - Region of Competence via KNN**: When a sample is provided, the router uses the sample to query the KNN index to define a local region of competence for DES. This is used for competence estimation in stage 2 and FIRE-DES pruning in stage 3.
2) **Stage 2 - Continuous OLA Competence + Weighted Voting**: Stage 2 computes continuous OLA competence for earch expert. Effectively, experts that assign higher probabilities to the correct class in the given neighborhood get higher OLA. The router then applies a competence threshold, inspired by FIRE-DES [6] style pruning. Effectively, if several experts are below a minimum, the router simply returns to the `GENERALIST` prediction or returns a neutral probability for the expert. Finally, the last part of stage 2 performs a manual emphasis schema. For each competent expert, it weights their scores. If boosts were provided (boosts are effectively multipliers for an expert's competence, provided by the user in `des-config.yaml`) their weight is multiplied. With these weights, it performs a softmax sharpening to emphasize the most knowledgeable experts and to de-emphasize lesser knowledgeable experts.

The prediction is then formed by averaging the weights from the softmax of all competent experts.

The router has lots of moving parts and, as such, a manual grid parameter sweep was performed in order to find the best parameters. Other than the parameter sweep, we did slight tweaking (adding and removing experts, increasing and decreasing boosting) for some of the parameters ourselves.

V. Implementation Details

A. Tier-Based Routing

As mentioned in subsection IV-E6, some teams manipulated sections of binaries that are computationally hard to parse. While the files have a 5 MB limit, this does not mean that the sections of this file would be any easier to parse. Because of this, our team had to devise a way to make predictions on files even if they were too hard to parse and make a prediction on within 5 seconds.

Our team decided on Tiered Featurization: we will break a binary's features into 'tiers' and parse them one at a time. Table X outlines the breakdown of all tiers, including their features. The rationale behind tiered featurization is that there will be files too large to fully parse but we still want to be able to make inferences on these files. As such, we will train tier experts (`T0`,`T1`,`T2`) on the tiers of a binary.

Table X: Tier Overview Without Timing

Tier Name		Features	Rationale
T-1	Timeout	None	T1 could not be parsed in time. Returns a 0.
T0	Ultra-Fast Metadata	PE headers, imports, exports, data directories, authenticode (1,445 features)	Instant structural metadata extracted directly from PE headers.
T1	Histogram Features	Byte histogram, entropy histogram (512 features)	Fast statistical characterization of raw byte content.
T2	File Structure	General file info, section analysis (60 features)	Deeper structural inspection of PE layout and section properties.
T3	String Analysis	String extraction and analysis (104 features)	Most expensive tier; captures semantic content and complex string patterns.

Logistically, tiered featurization starts as a single process. The `main` process creates another process called the `featurizer`, which has access to the file. The `main` process will start a timer. The `featurizer` will parse the binary in the aforementioned tiers, and will give the `main` process the features as it completes them. If the `main` process runs out of time, it will stop the `featurizer`. The `main` process will then use whatever fully completed tiers it recently received to make predictions with. The following outcomes are as follows:

1) T-1: If the `featurizer` could not featurize T0, it will return a 0 for prediction.
2) T-0: Use model T0 to make a prediction.
3) T-1: Use model T1 to make a prediction.
4) T-2: Use model T3 to make a prediction.
5) T-3: Use the GENERALIST and DES models (best outcome).

As mentioned before, the tiered featurizer has a timeout capability. The timeout has been hard-coded into the tiered feautrizer to be 2.95. We chose this number because, on average, `peer` samples took approximately 2.24 seconds to featurize and 3.016052 seconds with 95% confidence. Distributions of time for featurization and prediction can be found in Figure-10.

We settled for 2.95 as an upper-bound estimate. Any time that is not used by the tier featurizer is donated to the prediction step. For example, if it only takes 1 second to featurize, then the remaining $2.95 - 1 = 1.95$ seconds will be given to the predictor. More on the predictor can be found in the following subsection, subsection V-B. As a technical note, the `featurizer` processes are kept alive in a pool. This ensures that when the Flask application in our Docker container starts, these processes are kept alive across predictions to minimize startup time.

To become more robust against teams that always landed in certain tiers, like Team 5 in Tier 2, we performed peer training on T2 with proxy domains for such teams. Such training did cause some compromises to be made in terms of accuracy at this stage; however, our team has justified this through the following hypothesis: Files that end up being evaluated in certain tiers are the kinds of files that we should train those tiers on. So, even if we run the risk of overfitting peer-trained tier experts, we are slightly less likely to overfit, given that the tier expert fills a very niche role.

We have included a logical flow diagram to outline the behavior of the tiered featurizer, which can be seen in Figure-9.

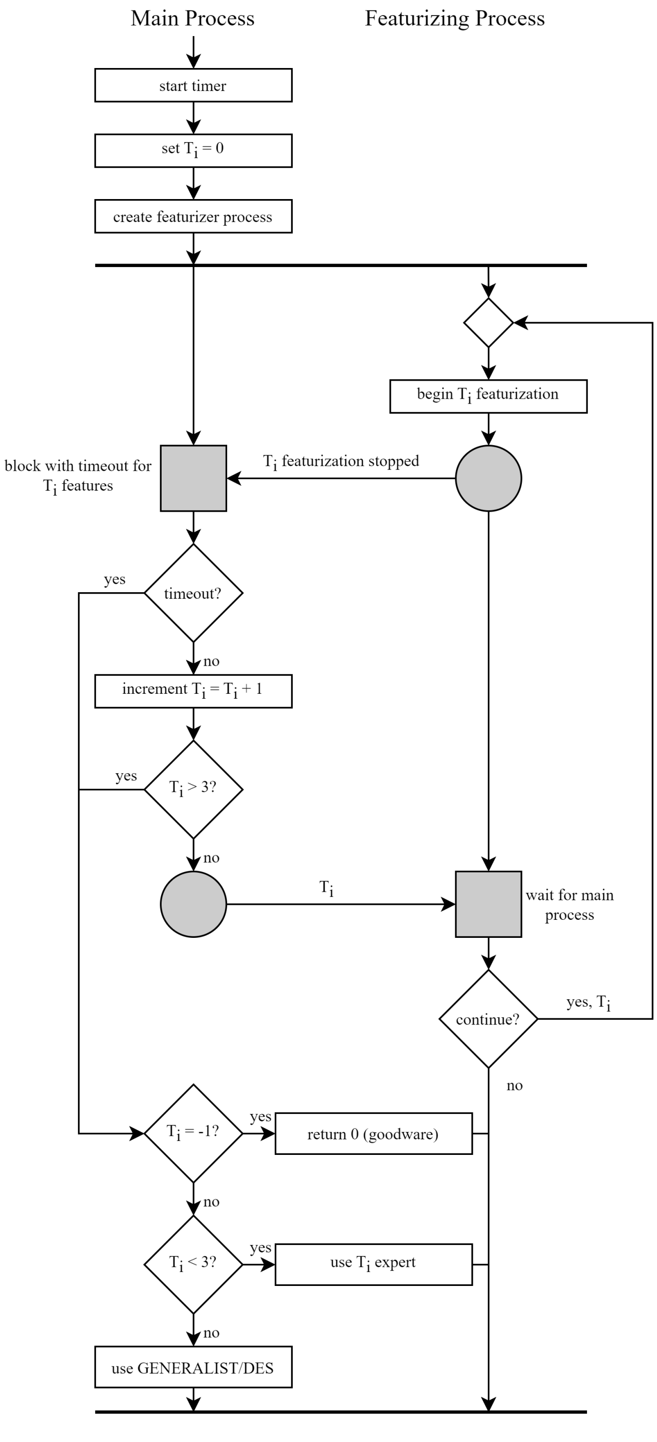


Figure 9: A logical diagram of the tiered featurizer.

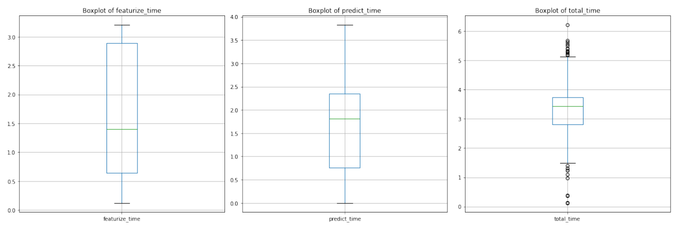


Figure 10: Three boxplots showing the spread of featurization, prediction, and total time.

B. *Speculative Prediction*

We found that, even if Tier 3 was being achieved, it was taking a long time for our DES model to make predictions on its own. Our team decided that we needed to find a faster way to make somewhat accurate predictions. Following the idea of speculative execution, our team decided that the predictor should attempt to make two predictions at the same time- one with the GENERALIST and the other with the DES model. Generally, the GENERALIST takes a lot less time to predict the DES model by virtue of it being smaller and not an ensemble model. Since the GENERALIST will always finish early, we can store the GENERALIST prediction and wait for the DES prediction in the meantime. If the DES prediction does not finish in the allocated time, then it will return the GENERALIST's prediction. If the DES model does finish in time, then we discard the GENERALIST's prediction and use the DES prediction instead. If neither the GENERALIST or DES finish in time, the model returns a 0 for goodware. A diagram of this process can be seen in Figure-11.

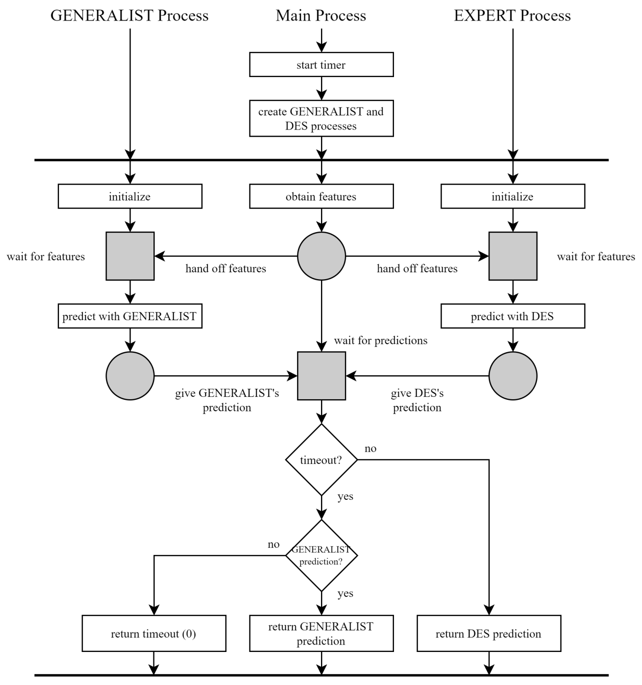


Figure 11: A logical diagram of the speculative predictor.

As mentioned in the previous subsection IV-E5, the speculative predictor and tiered featurizer share time. The predictor itself has a timeout of 1.95s, determined by the math performed in the previous decision for the time allocated to the featurizer, and with a small buffer for I/O operations. Adding the times of the tiered featurizer and the speculative predictor, $2.95 + 1.95 = 4.9$ seconds, which is a generous buffer for any I/O operations that could occur in between. It should be noted that more experimentation can be done to justify this buffer time. Following, our team was more confident in allocating a smaller prediction time because prediction times do not correlate with featurization times, as seen in Figure-12 with an $R^2 = 0.052$. This makes sense, as featurization generates a feature vector of fixed size, which may or may not be related to the size of the file being featurized. As mentioned in the tiered featurizer, any time that is 'saved' when featurizing (i.e., the 2.95-second timeout was not reached), it will 'donate' the extra time to the predictor.

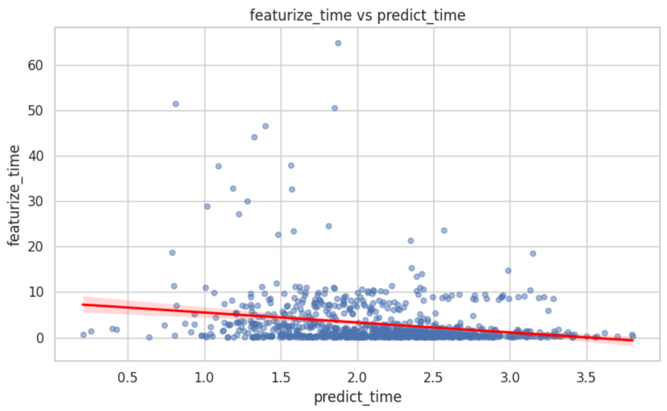


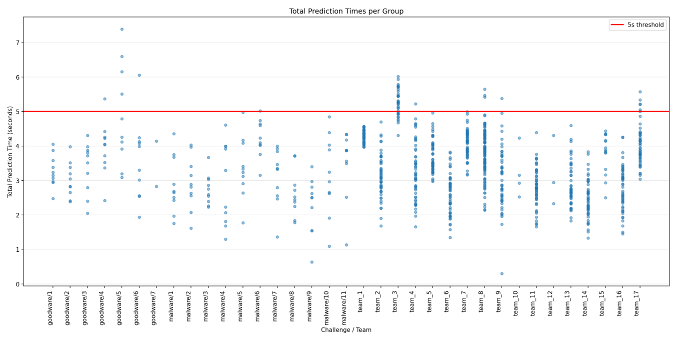
Figure 12: A graph of featurization time vs prediction time with a linear regression.

The creation of the speculative predictor (and the subsequent pooling of the predictor) drastically reduced and normalized the time it took to predict files. All binaries went from mostly meeting the 5-second deadline, as seen in Figure-13a, to falling well below it, as seen in Figure-13b.

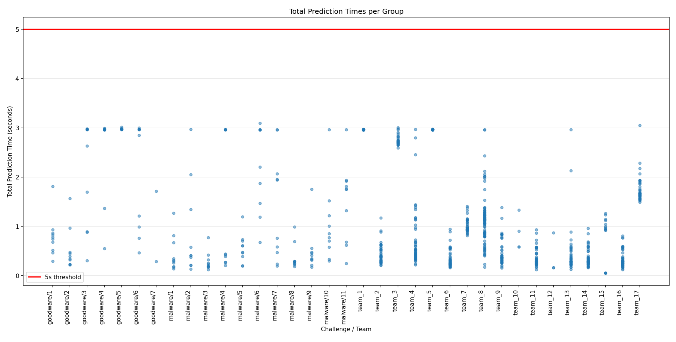
C. *Binary Analysis Pipeline*

To offer a brief overview of the analysis pipeline for binaries, we will provide a short explanation and diagram, provided in Figure-14.

1) The Flask app starts up. During this time, it initializes the DES process, GENERALIST process, and timed featurizer. It is crucial that the DES model loads the KNN competency regions independently, as this is nearly 300MB and should not be loaded by any other process. The other processes themselves are 200MB, totalling to approximately 900MB total, with buffer for the `challenge` binaries (even though those are larger than this challenge permits) and `peer` binaries (even though they are smaller than 5MB).
2) The Flask app receives a binary and featurizes it with tiered featurization.
3) If the tiered featurizer was not able to featurize the entire file, it will send it to its respective tier model. If it wasn't



(a) Prediction time for binaries before the speculative predictor.



(b) `T2`'s performance after peer training.

Figure 13: Prediction time for binaries before and after the speculative predictor.

able to finish Tier 0, it will fall back to the Tier -1 case and will return 0 to indicate timeout/goodware.

4) If the tiered featurizer featurizes the entire file, it will send it to the speculative predictor.
5) The speculative predictor will give the features to the `GENERALIST` and `DES` processes.
6) The speculative predictor will return the `DES` prediction if it finishes in time.
7) If `DES` doesn't finish in time, it will see if it can return the `GENERALIST` prediction.
8) If the `GENERALIST` predictor didn't finish in time, it will return 0 to indicate timeout/goodware.

VI. Extras

This section labels any extra projects or tasks not directly linked to the creation of the `TSDES` model.

A. *Tomato Town and Tilted Towers*

Tomato Town and Tilted Towers were scripts for automatically collecting malware throughout the semester. Tomato Town queried malware distribution websites (MalwareBazaar, MalShare) for malware, downloaded them, and featurized them (using the `EMBER2024` featurizer) while respecting API limits. Tilted Towers is a front-end visualization of the data as well as offering a small suite to use the tools. Our team, unfortunately, did not have much time to capitalize on this venture. However, as of November 14, 2025, we were able to passively collect

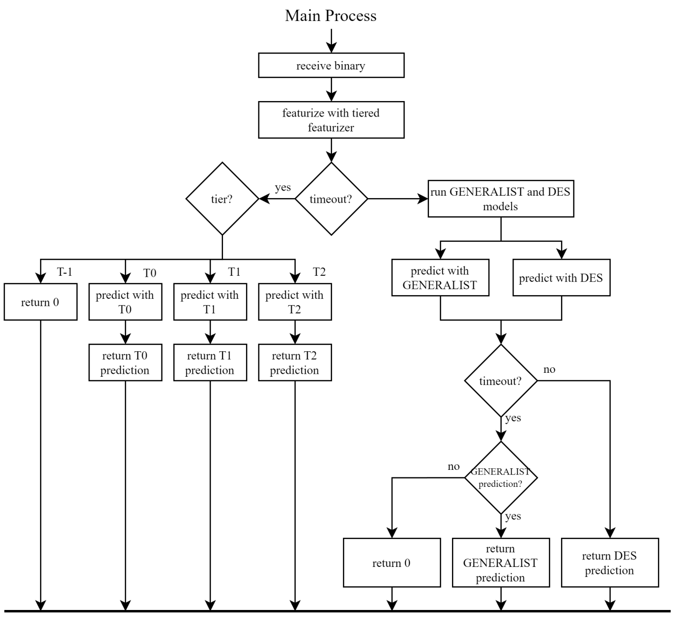
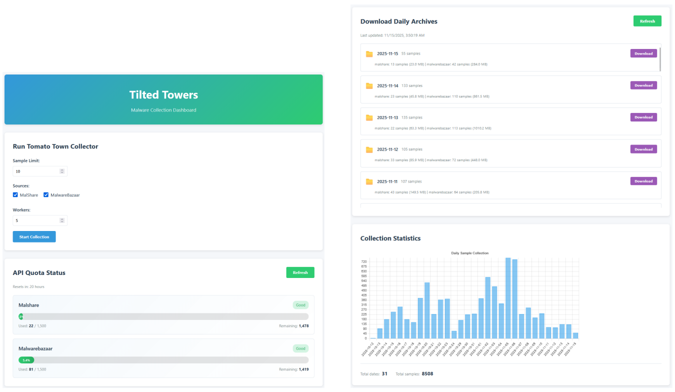


Figure 14: A diagram demonstrating the TSDES binary analysis pipeline.

8,505 PE samples. Screenshots of the websites can be seen in Figure-15.


Tilted Towers
Malware Collection Dashboard

Run Tomato Town Collector
Sample Limit: 10
Sources: ☑ MalShare ☑ MalwareBazaar
Workers: 5
Start Collection

API Quota Status
Refresh
Resets in: 20 hours
Malshare Good
Used: 22 / 1,500 Remaining: 1,478
Malwarebazaar Good
5.4%
Used: 81 / 1,500 Remaining: 1,419

Download Daily Archives
Refresh
Last updated: 11/15/2025, 3:50:19 AM
2025-11-15 55 samples Download
malshare: 13 samples (23.0 MB) | malwarebazaar: 42 samples (284.0 MB)
2025-11-14 133 samples Download
malshare: 23 samples (45.8 MB) | malwarebazaar: 110 samples (861.5 MB)
2025-11-13 135 samples Download
malshare: 22 samples (63.3 MB) | malwarebazaar: 113 samples (1010.2 MB)
2025-11-12 105 samples Download
malshare: 33 samples (85.9 MB) | malwarebazaar: 72 samples (448.0 MB)
2025-11-11 107 samples Download
malshare: 43 samples (149.5 MB) | malwarebazaar: 64 samples (205.8 MB)

Collection Statistics

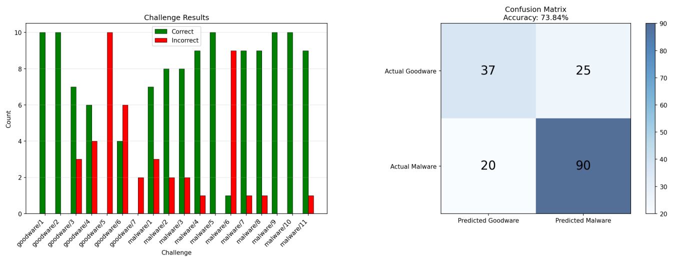
Total dates: 31 Total samples: 8508

Figure 15: Screenshots of the Tilted Towers website using the Tomato Town tool.

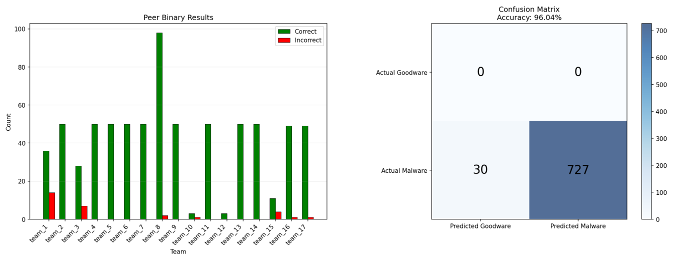
VII. Conclusion

Team 9[1] has created a feature-based malware detection system using DES, tiered featurization, and speculative prediction. Many of our members have started from knowing very little, if anything, about machine learning. From Professor Botacin's instruction to our own independent exploration and experimentation of existing technologies, we have made great strides in the direction of improving as engineers. The results of the `TSDES` model can be found in Figure-16 below. `TSDES` scored 73.84% on `challenge` binaries, and 96.04% accuracy on `peer` binaries.

[1]Contributions can be distributed evenly among all Team 9 members.



(a) TSDES performance on `challenge` binaries.



(b) TSDES performance on `peer` binaries.

Figure 16: Prediction results for the TSDES model on `challenge` and `peer` binaries.

References

[1] Hyrum S. Anderson and Phil Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models". In: *arXiv preprint arXiv:1804.04637* (2018).

[2] Robert J. Joyce, Gideon Miller, Phil Roth, Richard Zak, Elliott Zaresky-Williams, Hyrum Anderson, Edward Raff, and James Holt. "EMBER2024 – A Benchmark Dataset for Holistic Evaluation of Malware Classifiers". In: *arXiv preprint arXiv:2506.05074* (2025).

[3] Pierpaolo Artioli, Antonio Maci, and Alessio Magrì. "A comprehensive investigation of clustering algorithms for User and Entity Behavior Analytics". In: *Frontiers in Big Data* (2024).

[4] Ahmad al-Qerem and Shadi Nashwan. "Exploring Cybersecurity Data Patterns: A Comparative Analysis of Dimensionality Reduction Techniques". In: *Proceedings of the 2024 9th International Conference on Information Systems Engineering (ICISE '24)* (2024).

[5] *Practical hyperparameter optimization: Random vs Grid Search*. https://stats.stackexchange.com/questions/160479/practical-hyperparameter-optimization-random-vs-grid-search. Accessed: 2025-02-14.

[6] Rafael M.O. Cruz, Dayvid V.R. Oliveira, George D.C. Cavalcanti, and Robert Sabourin. "FIRE-DES++: Enhanced online pruning of base classifiers for dynamic ensemble selection". In: *Pattern Recognition* 85 (2019), pp. 149–160. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2018.07.037. URL: https://www.sciencedirect.com/science/article/pii/S0031320318302760.