ENDGAME.

It's a Bake-off!:
Navigating The Evolving World of Machine Learning Models

SERIES ONE, VOLUME ONE

HYRUM ANDERSON + BOBBY FILAR +
PHIL ROTH + JONATHAN WOODRIDGE

In our previous blog, we reviewed some of the core fundamentals in machine learning with respect to malware classification. We provided several criteria for properly evaluating a machine learning model to facilitate a more thorough understanding of its true performance. In this piece, we dig deeper into the operationalization of machine learning, covering the basics of feature engineering and a few commonly used types of machine learning models. We conclude with a head-to-head comparison of several models to illustrate the various strengths and performance tradeoffs of these varied machine learning implementations. When operationalizing machine learning models, organizations should seek the solution that best balances their unique requirements, including query time, efficacy performance, and overall system performance impact.

Feature Engineering

Feature engineering is a major component of most machine learning tasks, and entails taking some raw piece of data, such as malware, and turning it into a meaningful vector of features. Features are a numeric representation of the raw data that can be used natively by machine learning models. For this post, our feature vector is a vector of floats generated from a static Portable Executable (PE) file. We follow the feature extraction techniques outlined in Saxe and Berlin's malware detection piece, and use these features for all classifiers. The features include:

- **Byte-level histogram**
- **Entropy measured in a sliding window across the PE file**
- **Information parsed from the PE file, including imports, exports, section names, section entropy, resources, etc.**

Similar to previous research, these features are condensed into a vector of fixed length of 1,562 floats. Turning a seemingly infinite list of imports, exports, etc. into a fixed length vector of floats may seem like an impossible task, but feature hashing makes this possible. Think of the vector of floats as a hash table, wherein a hashing function maps one or more features to a single feature that accumulates the values that correspond to those features. This technique is quite common and is surprisingly effective.

Models

Now that we have defined our features, it is time to select the models. There are too many machine learning models for a complete review, so we focus on seven common models: naive Bayes, logistic regression, support vector machine, random forest classifier, gradient-boosted decision tree, k-nearest neighbor classifier, and a deep learning model. In our descriptions below, we explain how each model performs as they pertain to malware classification, although clearly they support many other use cases.

Naive Bayes

Naive Bayes is one of the most elementary machine learning models. Naive Bayes crudely models the features of each class as having been generated randomly from some user-specified statistical distribution, like the Gaussian distribution. Its name comes from its reliance on Bayes theorem with a naive assumption that all features are independent of one another. Independence of features implies that the occurrence of one feature (e.g., entropy of the .text section) has no bearing on the value or occurrence of another feature (e.g., an import of FtpGetFile fromWinINet.dll). It is termed naive since this assumption rarely holds true for all features in most applications (e.g., an import of FtpGetFile is correlated with an import of InternetOpen). Despite its simplicity, naive Bayes works surprisingly well in some real world problems such as spam filtering.

Logistic Regression

Logistic regression (LR) is a model that learns a mapping from a vector of feature values to a number between 0 (benign) and 1 (malicious). LR learns one coefficient for each element of a sample's feature vector, multiplying the value of each element by the corresponding coefficient, and summing up those products across all elements. LR then compresses that number to a value between 0 and 1 using a logistic function, and approximates from the features the target label (0=benign, 1=malicious) provided in the training set.

Support Vector Machine

A support vector machine (SVM) is a so-called max-margin classifier, of which we only consider a linear SVM. Linear models seek a straight line that bisects malicious from benign samples in our feature space. In turn, a linear SVM aims to find the fattest line/slab that separates malicious from benign. That is, of all slabs that could separate malicious features from benign features, SVM finds the one that occupies the most empty space between the malicious and benign classes. This "fatness" objective provides a measure of robustness against new malware samples that may begin to encroach on the margin between malicious and benign samples.

Random Forest Classifier

A random forest classifier is a model that learns to partition the feature space using a committee of decision trees. Each decision tree is trained to essentially play a game of "20 questions" to determine whether a sample's features represent a malicious or benign file. The number of questions and which questions to ask are learned by an algorithm. To provide good generalization without overfitting, each decision tree is trained only on a subset of the data and questions can be asked only from a subset of features (hence, "random"). Tens or even hundreds of decision trees of this type (hence, "forest") are bagged together in a single model, where the final model's output is a majority vote of each randomized decision tree in the forest.

Gradient-Boosted Decision Tree

Like the random forest classifier, a gradient-boosted decision tree (GBDT) combines the decision of many small decision trees. In this case, however, the number of questions that can be asked about the features are restricted to a relatively small number (perhaps one to ten questions per tree). From this initial tree, the model makes several errors, and in the next round (i.e., next decision tree), the algorithm focuses more attention on correcting errors from the previous round. After tens or hundreds of rounds, a final determination is made by a weighted vote of all the trees.

K-Nearest Neighbors (k-NN)

k-nearest neighbors (k-NN) is another extremely simple model. The idea is that you can classify an object by its closest (or most similar) neighbors. For example, in a database of ten million malware and benign samples, if a file's ten most similar matches in the database are all malware, then the object is probably malware. This classifier is non-parametric (doesn't try to fit the data using some user-prescribed function) and completely lazy (requires no training!). Nevertheless, in straightforward implementations one must search the entire dataset for every query, so memory and speed may be limiting factors for large datasets. A whole branch of research is devoted to approximating nearest neighbor search to improve query speed.

Deep Learning

Deep learning refers to a broad class of models defined by layers of neural networks. Since this class is so broad, we briefly describe only the deep learning classifier in the malware detection paper previously referenced in the feature engineering section. This model consists of 1 input layer, 2 fully-connected hidden layers, and an output classifier layer. Our implementation of the model differs slightly from that paper in an effort to improve model training convergence. It should be noted that, like all models in this blog post, this deep learning model ***does not*** represent any company's next-gen malware product, and is intended only for coarse comparison with other model types. Since deep learning is a fast moving research field and may produce models of varying complexity and performance, it would be unfair and misleading to draw narrow product conclusions about this particular embodiment of deep learning.

Bake-Off Framework

With our features and models selected, it is now time to evaluate the performance of each model and explore whether there is a best-of-breed model for malware classification.

As we previously highlighted in our last post, one must take care when comparing models: "accuracy" alone doesn't tell the whole story. We discussed several performance measures including false positive rate (FPR), false negative rate (FNR), true positive rate (TPR)—which depend on a user-specified threshold on a model's output—and the Receiver Operating Characteristic (ROC) which explicitly shows the tradeoff between FPR and TPR for all possible model score thresholds. In what follows, we'll provide a summary of the model performance by reporting the area under the ROC curve (AUC), which can be thought of as the TPR averaged over all possible thresholds of the model score: it allows performance comparisons without first selecting an arbitrary threshold. However, at the end of the day, a model must make an up/down decision on whether the sample is malicious, so in addition to AUC, we also report FPR and FNR with a threshold set at the "middle" value (0.5 for models that report scores between 0 and 1).

Since these metrics are not the only consideration for a particular task, we also compare model size (in memory) and speed (query rate measured in samples per second using wall clock time). Wall clock time can be misleading due to differences in machine specs, threading, and current process loads. We attempt to mitigate these issues by testing all models on the same machine, under minimal load, and all speed measurements are done on a single thread.

Experimental Setup

We implement our bake-off test in Python leveraging scikit-learn, Keras, and a custom NearestNeighbor algorithm. We train each of these models against the same dataset: a random sample of 2 million unique malicious and benign PE files. Unfortunately, in the real world data doesn't follow a perfect Gaussian distribution, but rather is often quite skewed or imperfect. In this case, the breakdown of malicious to benign is 98/2, highlighting the class imbalance problem mentioned in the previous post. We attempt to maximize the effectiveness of each model during training by performing a grid search to determine ideal parameters for learning. Each machine learning algorithm has a set of hand tunable parameters that determine how the training is conducted. In an exhaustive grid search, the models are trained with multiple combinations of these parameters and the performance at each grid point is analyzed.

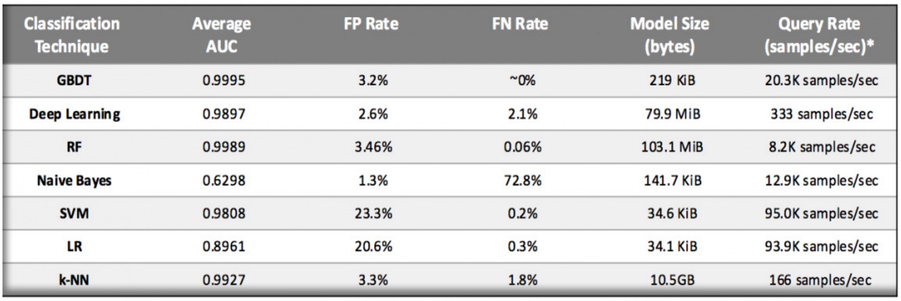
Performance measures were calculated using a training set and a testing set. The models are trained on the training set, then performance measures are calculated on the disjoint testing set. The disjointness is key: if our testing set were to contain items from our training set, then we would be cheating and the results would not represent the performance of our classifier in the real world. However, it is possible to hand craft the training and testing set to achieve better (or worse) performance. To remove any bias, we use a technique called cross-validation. Cross-validation creates several randomly-selected training/test splits and the performance metrics can be aggregated over all sets. The number of training/test set divisions is called folds and we use ten folds in our experiments.

Results

Results for the bake-off and interpretation of each model's performance are given in Table 1. False positive and false negative rates were calculated using a scoring threshold naively chosen halfway between benign and malicious.

** Query time does not include time required to extract features*

*** Update: We used a default threshold for each model (0.5 on a scale of 0 to1) to report FP and FN rates. As with other details in each model, the threshold would be chosen in a real-world scenario to a fixed low FP rate with a corresponding trade-off in FN rate. AUC is an average measure of accuracy before thresholding, and }is the most appropriate metric that should be used here to compare models.*



Classification Technique	Average AUC	FP Rate	FN Rate	Model Size (bytes)	Query Rate (samples/sec)*
GBDT	0.9995	3.2%	~0%	219 KiB	20.3K samples/sec
Deep Learning	0.9897	2.6%	2.1%	79.9 MiB	333 samples/sec
RF	0.9989	3.46%	0.06%	103.1 MiB	8.2K samples/sec
Naive Bayes	0.6298	1.3%	72.8%	141.7 KiB	12.9K samples/sec
SVM	0.9808	23.3%	0.2%	34.6 KiB	95.0K samples/sec
LR	0.8961	20.6%	0.3%	34.1 KiB	93.9K samples/sec
k-NN	0.9927	3.3%	1.8%	10.5GB	166 samples/sec

ENDGAME.

Table 1: AUC, FP/FN rate and Model Metrics for Classifiers

AUC

The top performing models based on the average AUC are the boosted trees models (GBDT and Random Forest), as well as the Deep Learning and k-NN models, with scores hovering around 0.99. These models all had significantly smaller FP/FN rates than the next highest scoring models, SVM and Logistic Regression, which saw FP rates in the 20% range. A model that produces 1 FP for every 5 samples is far too many to be used operationally (imagine the alerts!). Naive Bayes performed poorly in this exercise likely due to a lack of feature independence between the malicious and benign labels.

Model Size

The size of the final model has little to do with an ability to correctly classify samples, but it is extremely important for real-world implementation for classifying malware on a user's machine. The bigger the model is on disk (or in memory), the more resources it consumes. Larger models are infeasible on an endpoint because of the negative impact on the host environment. Avoiding any degradation to the user's experience on their machine is paramount for any successful endpoint solution.The results of the bake-off show three major classes of model sizes by impact of the host system.

As Table 1 depicts, SVM, Logistic Regression, Naive Bayes, and GBDT models consumed less than 1MB, which would be considered small and have a negligible impact on a system. After eliminating the models that had unreasonable average AUC scores, GBDT again outperforms the competition.

Both Deep Learning and Random Forest performed well in the AUC test, but the size of the models on disk could have a negative impact. While 80-100MB is certainly not large, these models look like monsters compared to the tiny GBDT. Moreover, there is not a significant enough increase in AUC to warrant a model of this size. Finally, k-NN weighed in at

10.5GB which is probably not a viable solution for an endpoint model. Remember: k-NN has no parameters, instead it requires maintaining the entire training set in order to perform a nearest neighbor search! With its high AUC average and low FP/FN rates, k-NN represents a potentially viable cloud-based classification solution.

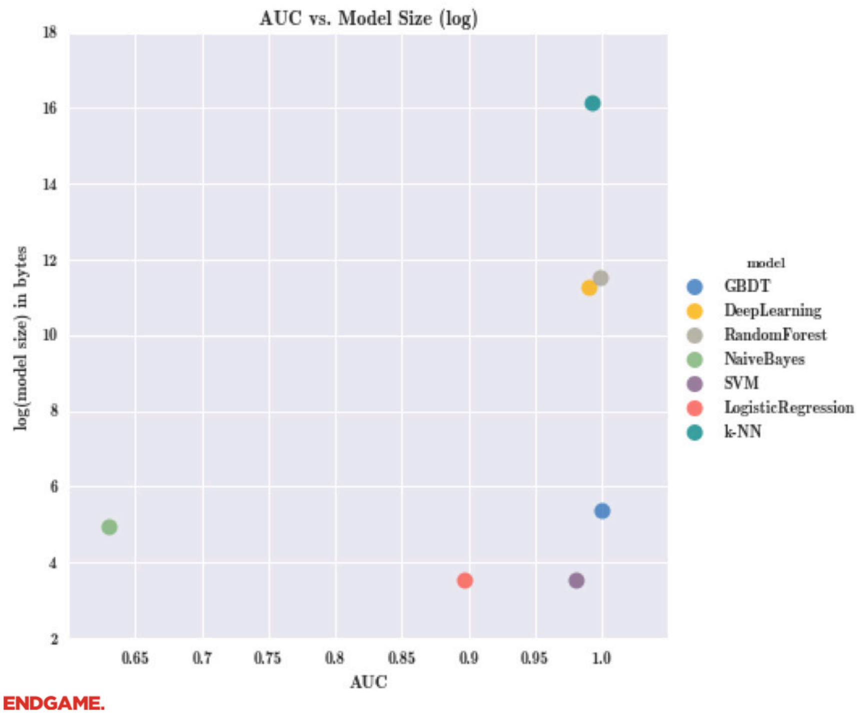


Figure 1 – Model AUC vs. Model Size on Disk

Query Time

An endpoint malware classifier must make snap decisions on whether a newly observed binary is malicious or benign. Failure to do so could result in the execution of a binary that wreaks havoc on an individual machine or an enterprise network.

The ability for a model to quickly classify multiple binaries could be a major differentiator in our search for an ideal model. Figure 3 shows how many queries can be made against each model per second. This time does not include feature calculation, since that time is equivalent for each model.

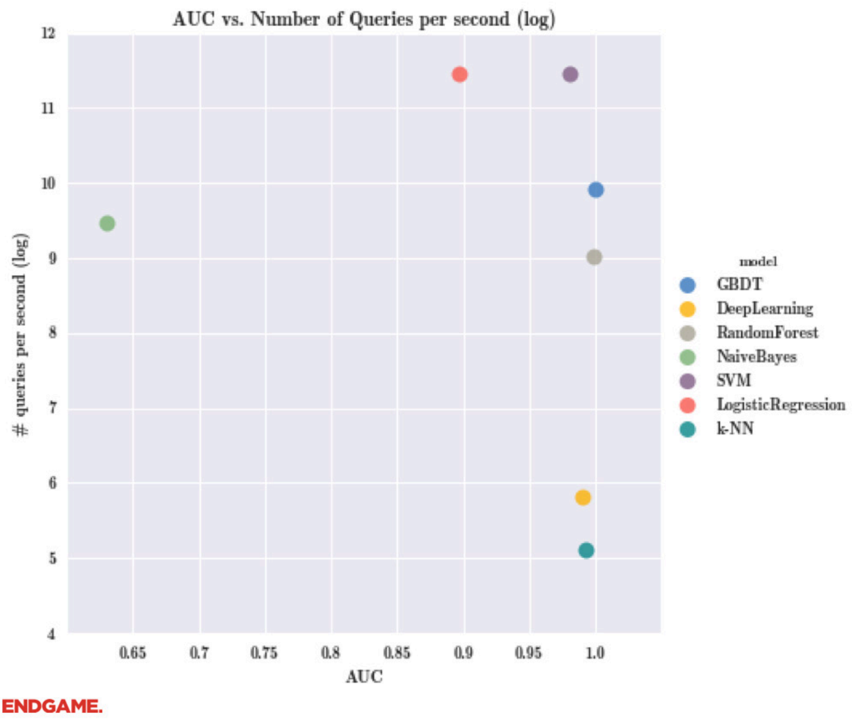


Figure 2 – Model AUC vs. Query per second

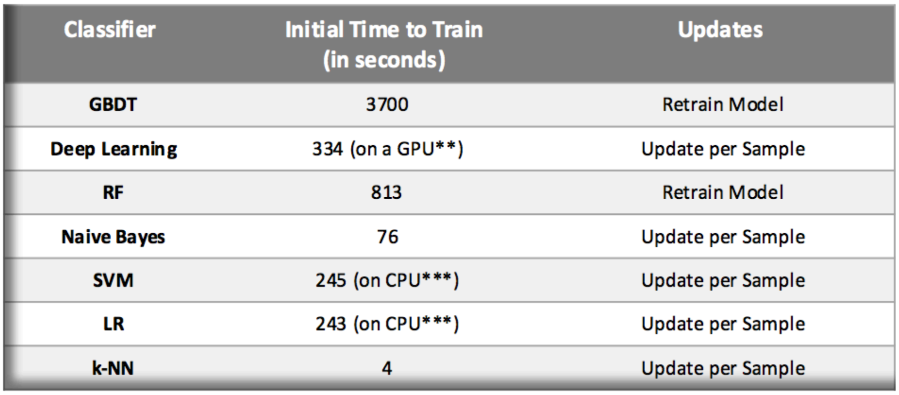
SVM and Logistic Regression outperformed the rest of the competition in this category. Both models only require a couple of addition and multiplication operations per feature at query time. GBDT and Random Forest models were in the next tier for speed. The query time of these models is dependent on the size and quantity of the decision trees they've trained. The number and size of those trees can be controlled during training and there is a tradeoff between speed and classification accuracy here. For our bake-off, both models were tuned for best classification accuracy, so it is nice to see those models still performing quickly. For each query, Deep Learning involves a series of multiple matrix multiplications for each query, and k-NN involves searching through the entire training set for each query, so it is not surprising to see that these models are slower.

Other Considerations

AUC, model size, and query time are all important metrics for evaluating machine learning models, but there are other areas that warrant consideration, too. For instance, training time is an important factor, particularly when your dataset is constantly evolving. Malware is a great example of this problem as new families and attack vectors are constantly developed. The ability to quickly retrain and evaluate models is crucial to keeping pace with the latest malicious techniques. Table 2 details each model's initial training time and how the model can be updated.

*** We show GPU instead of CPU time due to default implementation and customary training method. The CPU time would consist of nearly 16 hours (57K seconds!)*

**** SVM and LR models could also be conveniently trained using a GPU. However, default implementations of publicly available packages leverage CPU.*



Classifier	Initial Time to Train (in seconds)	Updates
GBDT	3700	Retrain Model
Deep Learning	334 (on a GPU**)	Update per Sample
RF	813	Retrain Model
Naive Bayes	76	Update per Sample
SVM	245 (on CPU***)	Update per Sample
LR	243 (on CPU***)	Update per Sample
k-NN	4	Update per Sample

ENDGAME.

Table 2 – Model Training Time and Update Method

Final Thoughts

Machine learning has garnered a lot of hype in the security industry. However, machine learning models used in products may differ in architecture, implementation, training set quality, quantity and labeling (including the use of unlabeled samples!), feature sets, and thresholds. None of the models analyzed represent any company's next-gen malware product, and are intended only for coarse comparison with other model types. That said, one may coarsely compare results to get a sense of suitability to a particular application (e.g., endpoint vs. cloud-based malware detection).

As the bake-off demonstrated, each model has strengths and weaknesses. GBDTs are an especially convenient model for real-time endpoint-based malware classification due to a good combination of high AUC, small size, and fast query speed. However, GBDTs require a complete retraining to update a model, which is the most time consuming training process of all tested algorithms. Both Random Forest and k-NN models performed nearly as well as GBDT (avg. AUC), but were significantly larger (in size on disk) models. However, despite its size, k-NN requires no training at all!

There is no magical formula for choosing the machine learning model that's right for your situation. Some of the considerations that we've discussed, like query time or training time, might not matter in other application areas. In some situations, you're trying to provide the best experience to the user of your model and so convenient metrics for optimization like AUC may not exist. When computing resources are so cheap and developer and data scientist time so expensive, it might be best to go with the model with which you have the most experience and best meets your operational requirements. The next step to building a great classifier involves handling corner cases, continually cleaning, managing, and improving your collected dataset, and verifying your results at every step of the way. In the next and final post in this three-part series, we'll consider some of the finer details of implementing a lightweight endpoint malware classifier using GBDTs.

ENDGAME.