CS 181 – PRACTICAL 3

CASEY GRUN, SAM KIM, RHED SHI

1. Warmup

We attempted to implement multiclass logistic regression and generative classifers with Bayesian class-conditional densities on fruit data consisting of 59 data points sorted into apples, oranges, and lemons. The number of classes is $K = 3$. The data points reflect the height and width of ech fruit, giving us two parameters $\mathbf{x_1}$ and $\mathbf{x_2}$. Our basis function is given by

$$\Phi(\mathbf{x}) = w_0 + w_1 \cdot \mathbf{x_1} + w_2 \cdot \mathbf{x_2} = \mathbf{w}^T\mathbf{x}$$

1.1. **Logistic Regression.**

We first plotted the lines described by the weights in our logistic regression learning algorithm shown below.

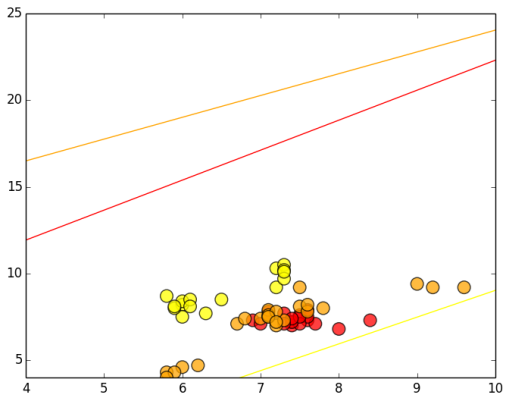

Figure 1. Lines described by the weights of the logistic regression algorithm.

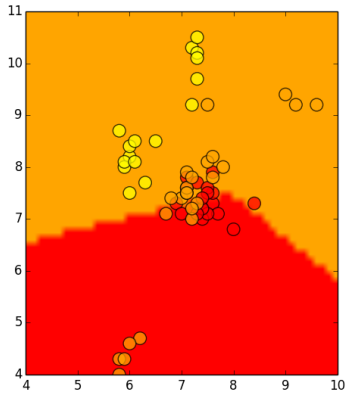

FIGURE 2. Logistic regression with decision boundaries.

Next, we plotted the coloring of the graph using the maximum $p(C_k)$ for each point in the coordinate space. Admittedly, our algorithm fails to capture the lemons due to some bug in the code. We randomized the guesses for our $\mathbf{w}$ vector and ran the optimization using the inverse Hessian and gradient of the error function given by

$$\nabla E(\mathbf{w}) = \Phi^T(\mathbf{y} - \mathbf{t})$$
$$\mathbf{H} = \Phi^T \mathbf{R} \Phi$$

1.2. **Generative Classifier with Bayesian Class-Conditional Densities.** We defined the weights of our generative classifier method with Bayesian class-conditional densities as

$$\mathbf{w}_k = \mathbf{\Sigma}_k^{-1} \mu_k$$
$$w_{k0} = -\frac{1}{2} \mu_k^T \mathbf{\Sigma}_k^{-1} \mu_k + ln(p(C_k))$$

where we have

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} t_n \mathbf{x}_n$$
$$\mathbf{\Sigma}_k = \frac{1}{N_k} \sum_{n \in C_k} (\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T$$

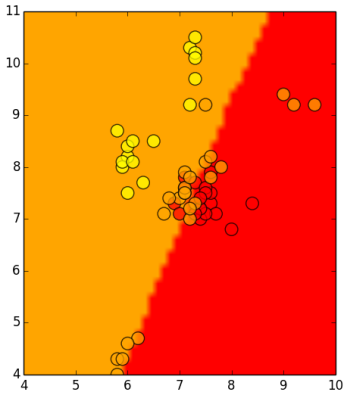

FIGURE 3. Logistic regression with decision boundaries.

Our results are shown here in the figure. Although there only appears to be one decision boundary between the apples and oranges, this boundary shows curved features not seen in the logistic regression because of the individual covariances in our weight calculations.

2. CLASSIFICATION

Our challenge was to classify a set of programs, based on traces of their system calls, as either a type of malware or a non-malicious program. Given an $N \times D$ matrix $\mathbf{X}$ representing the programs, along with a N-dimensional vector $\vec{t}$ of training labels, produce a function

$$y : \mathbf{X}' \mapsto \vec{t'}$$

. That is, a function which could produce labels $\vec{t'}$ for some matrix $\mathbf{X}'$. There were two parts to this challenge: determining the *feature functions* to generate rows of the matrix $\mathbf{X}$ based on the system calls for each malware program, and determining what classification algorithm to use to generate the function y.

We developed a cross-validation procedure to test our predictions on folds of the training data before submitting our predictions to Kaggle on the test data; we used 10 folds. A persistent problem we observed was that predicted scores in cross-validation were 0.1–0.2 higher than the scores we recieved on the public leaderboard on Kaggle. We suspect this was because of the heterogeneous training data

2.1. Feature Functions.

We evaluated a number of feature functions:

System call counts: Our first feature simply summed the number of system calls across all threads in all processes for a particular program. This method achieved ???? on the public leaderboard.

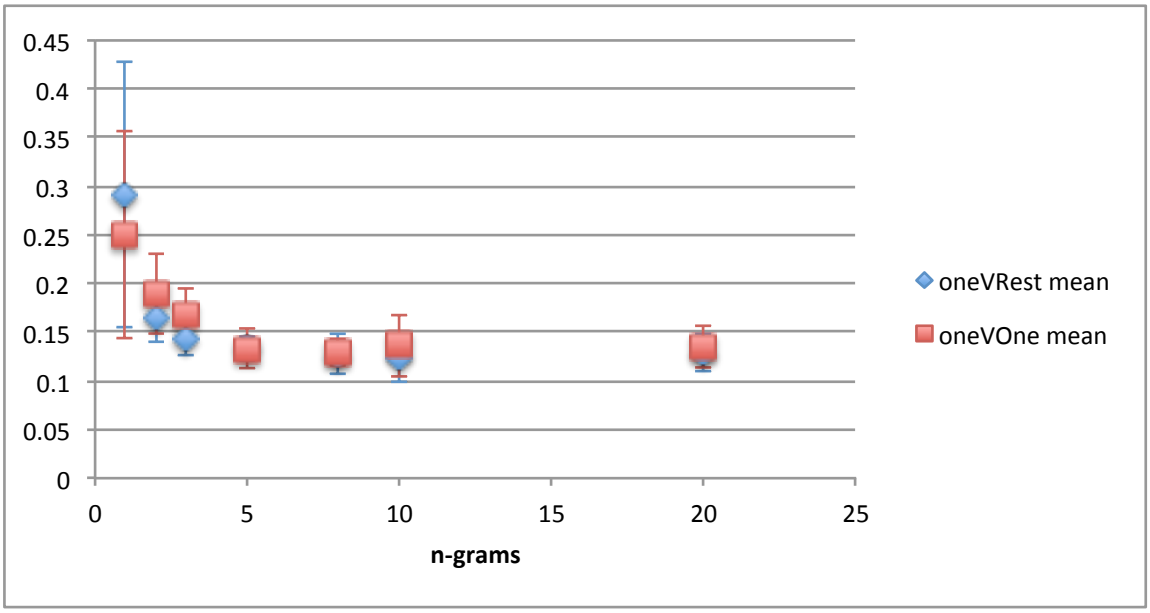

FIGURE 4. Tuning the length n of sequences of system calls. Y-axis is error.

n-grams of system calls: Our most substantial feature examined counts of sequences of system calls. Each unique sequence of n system calls was represented as a feature that counted how many times this sequence appeared in any given malware. We used cross-validation to try and optimize the choice of n—the length of these sequences. Too small an n would result in uninformative sequences, while too large an n would not generalize well. We found the optimum n to be approximately 8.

DLLs loaded: We also created features counting how many times each unique DLL was loaded with the `load_dll` system call. This feature proved to be reasonably informative, and achieved our best score on Kaggle (in combination with the 8-grams of system calls).

Registry keys: Finally, we employed a similar method to count the number of times each unique registry key was read by a program. This feature was less informative than the DLLs or 8-grams, so it was not included in our final predictions.

Additionally, we attemped classification on a reduced-dimensionality feature set, generated by performing a truncated singular value decomposition (TSVD) on the full feature set. This method did not perform particularly well, so it was not pursued.

Our final/best score was achieved by combining the DLLs, 8-grams, and system call counts features.

2.2. **Classification Methods.** We evaluated a number of classification learning algorithms, as implemented by the scikit-learn Python library. These include:

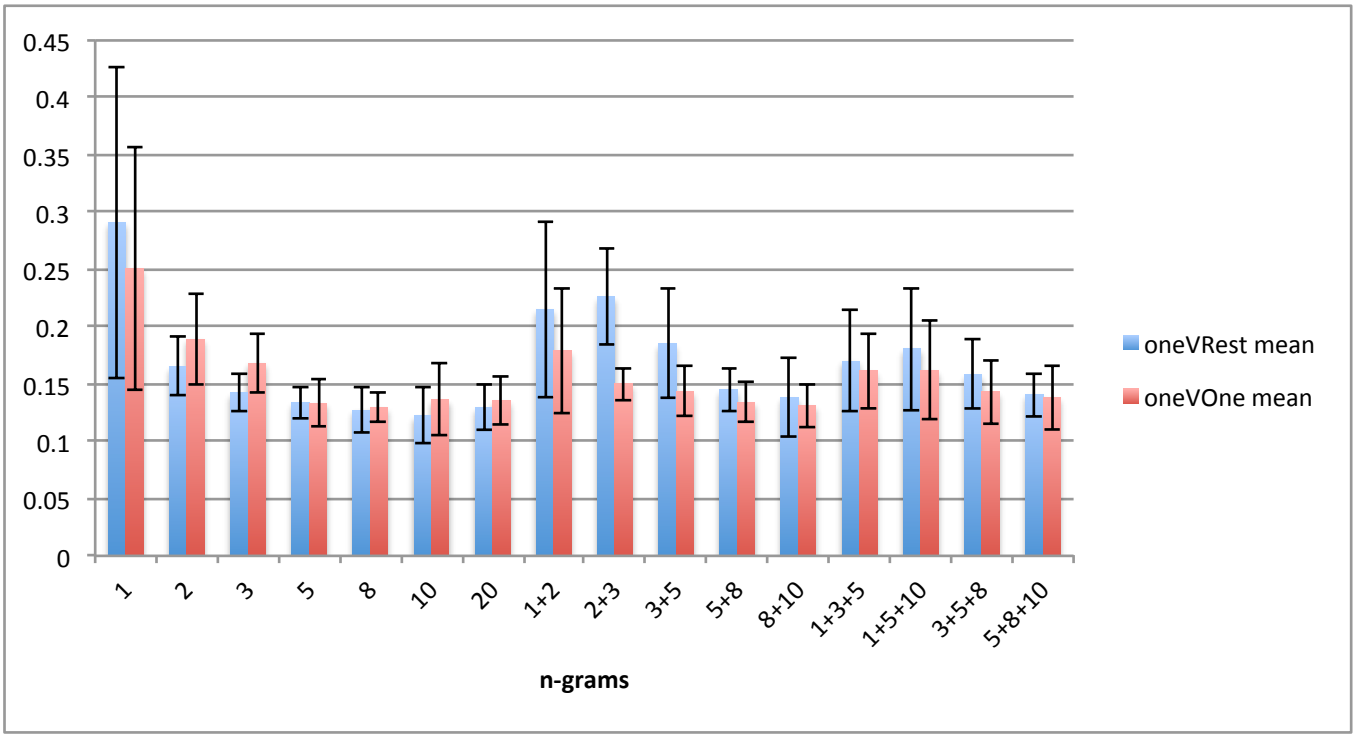

FIGURE 5. Considering combinations of n-grams of system calls. Y-axis is error.

One-vs-rest: As discussed in the textbook, this algorithm fits one classifier per class to classify if a point is in a class C_k or not. We used linear discriminant functions.

One-Vs-One: This algorithm introduces a discriminant function for every pair of classes, where each point is classified according to a majority of decisions among the discriminant functions. We used linear discriminant functions.

Logistic regression: This algorithm minimizes an error based on the training set to optimize the parameters for the likelihood function.

Decision trees: This algorithm goes down a tree of binary nodes in which the inputs are used to determine which path to go down, which ultimately assign probabilities to the target variable.

Forests of randomized trees: Related to the decision trees, this algorithm uses a set of trees, where each tree is built using random decisions. Whereas a decision tree will use the best split among all the features for each node, the random tree will pick the best split from a random subset of features. The trees are built independently and then their results are averaged to make predictions. This results in the bias increasing (due to not picking the best split) but the variance decreasing (due to averaging) which offsets the increase in bias.

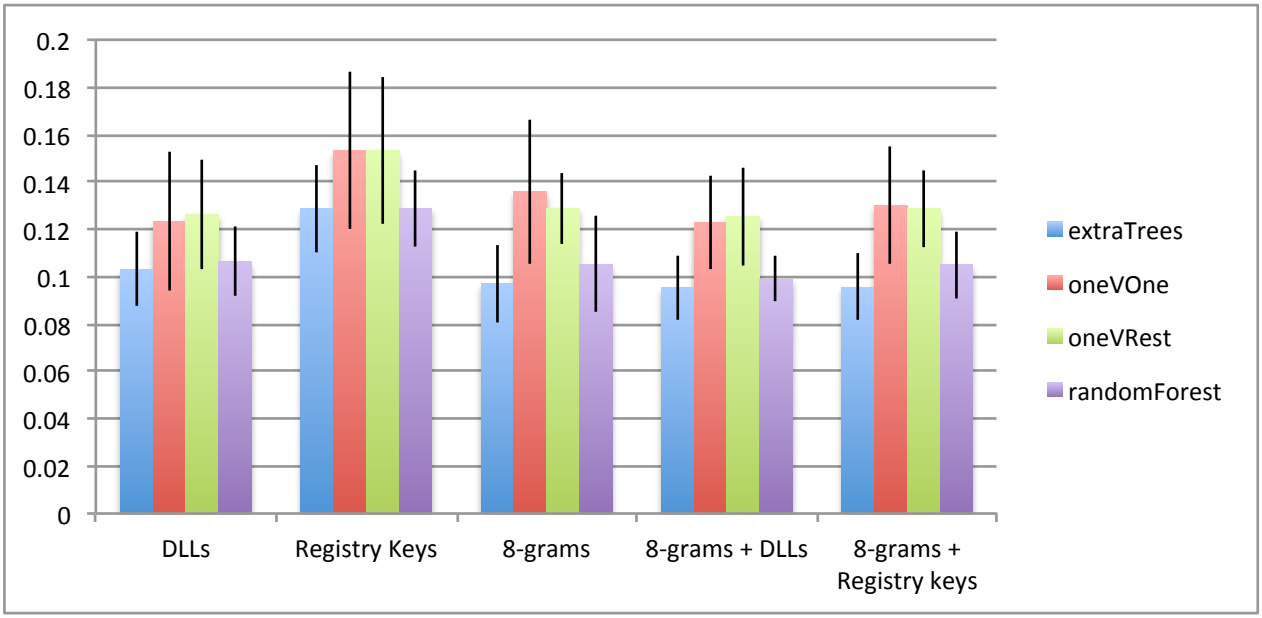

FIGURE 6. Comparison of best-performing features. Y-axis is error.

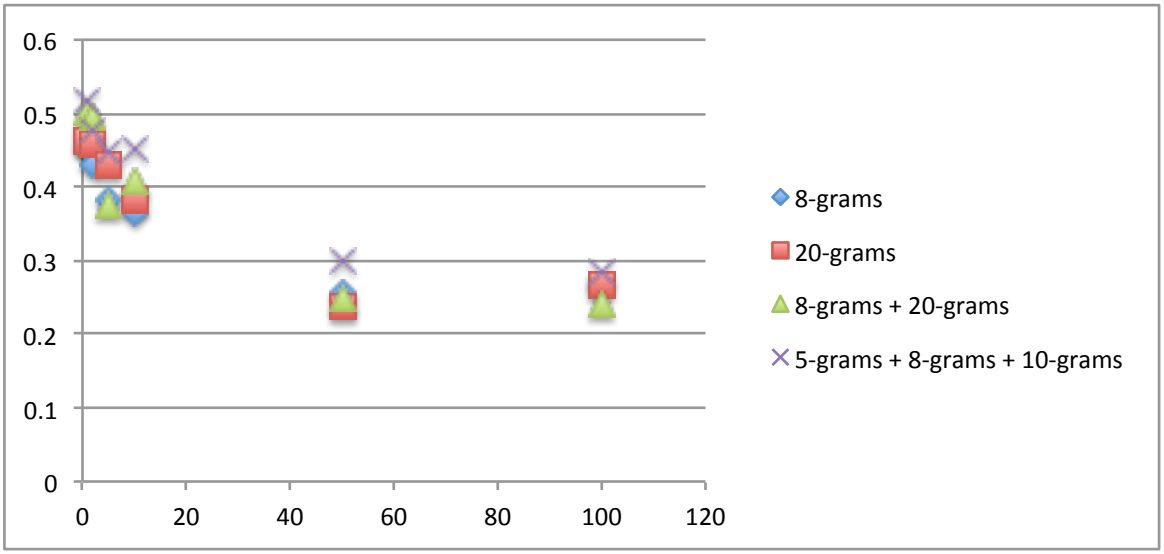

FIGURE 7. Comparison of a truncated singular value decomposition (TSVD) on various combinations of n-grams. The TSVD performed considerably worse than the full feature set in cross-validation, so it was not pursued.

Forests of extremely randomized trees: This is similar to the forests of randomized trees algorithm, except that splits are computed slightly differently in building the random trees. In addition to using a random subset of features for each node, instead of picking the best feature, thresholds are drawn randomly for each feature and these thresholds are picked as the splitting rules. This results in an even higher bias but lower variance.

We evaluated these learning algorithms based on the extracted features of system call counts and system call bigrams, and using cross-validation methods to withhold 10% of the data over 10 trials and averaging the percent errors over the predictions on the withheld data. Standard deviations over the 10 trials were calculated and the average running time for a single learning and prediction run are also noted. The results are summarized in the table below:

Classification algorithm	% error	Standard deviation	Running time (s)
One-vs-rest (linear)	12.7	2.4	7-8
One-vs-one (linear)	12.9	1.4	7-9
Logistic regression	11.5	1.5	600-700
Decision tree	10.9	1.3	50-60
Random trees	10.4	1.2	8-10
Extremely random trees	9.7	1.6	16-20

Additionally, on Kaggle, the one-vs-rest, decision tree, and extremely random trees classification algorithms using the system call counts and system call bigrams reached accuracies of 68.5%, 79.1%, and 81.0%, respectively.

3. Conclusion

References