Report on Malware classification

HarvardX Data Science Capstone Project

Raphael Kummer

2023-03-14

Contents

Malware

Introduction

Malicious software (Malware) is a software used to harm, damage, access or alter computer systems or networks. There are several types of Malware, including viruses, worms, trojan horses, ransomware and spyware. Most malware access computer systems by internet and emails. They can hide in legitimate applications or are hidden in the system. Static analyses can be effective, but also easily evaded through obfuscation and altering the malware so much, that it is no longer recognized. Therefore additional dynamic features are used to classify malwares.

Dataset

The dataset *Malware static and dynamic features VxHeaven and Virus Total Data Set* [2] from the UCI Machine Learning Repository contains three csv.

- staDynBenignLab.csv: 1086 features extracted from 595 files on MS Windows 7 and 8, obtained Program Files directory.
- staDynVxHeaven2698Lab.csv: 1087 features extracted from 2698 files of VxHeaven dataset.
- staDynVt2955Lab.csv: 1087 features extracted from 2955 provided by Virus Total in 2018.

The staDynBenignLab has all the features of typical non-threatening programs wheare as staDynVx-Heaven2698Lab and staDynVt2955Lab are extracted features of malware programs from the databases of VxHeaven and Virus Total. The dataset provides static features like ASM, compiler version, operating system version, Hex dump and PE header [3] (portable executable) and dynamic features extracted from a cuckoo sandbox [4]. The dataset only contains windows specific programs and malware. The dataset is downloaded from the UCI Machine Learning Repository Dataset and the containing csv files are extracted. The data in each of the csv must be labeled first, describing the class of program identified (e.g. class 0 for benign, class 1 for static malware features and class 2 for dynamic malware features).

Goal

The dataset is used to explore and gain insight on how normal programs and malware differ in static and dynamic structure. Several machine learning algorithms to classify an unseen program as benign or malware (static or dynamic) are developed and compared. Overall, the goal of this project is to improve our understanding of how malware differs from normal programs, and to develop effective techniques for automatically identifying and classifying malware based on its static and dynamic features.

Summary

Analysis

Disclaimer: Not all R code will be shown in this report. For the full code view the R-File on github.

Download

The zipped dataset is downloaded from the UCI archive and unzipped.

This will extract the following files: staDynBenignLab.csv, staDynVt2955Lab.csv, staDynVx-Heaven2698Lab.csv

Where

- *staDynBenignLab.csv*has 1087 features extracted from 595 files on MS Windows 7 and 8, obtained Program Files directory
- *staDynVxHeaven2698Lab.csv* has 1087 features extracted from 2698 files of VxHeaven dataset.
- *staDynVt2955Lab.csv* has 1088 features extracted from 2955 provided by Virus Total in 2018.

Preprocessing

Add classes to each dataset, depending on the origin of their data:

- 0: Normal *benign* programs
- 1: Malware with static features extracted from the *VxHeaven (vxheaven)* dataset
- 2: Malware with dynamic features extracted from the *Virus Total (vt)* dataset

Cleanup

The datasets have

- staDynBenignLab: 0,
- staDynVxHeaven2698Lab: 0 and
- staDynVt2955Lab: 0

missing values.

Check for features that not present in all three datasets, remove unique columns and combine all in one large dataset.

These columns are not present in all datasets:

. . . 1, filename, ___vbaVarIndexLoad, SafeArrayPtrOfIndex

So they are removed and the resulting dataset we now work with has 1086 features for each of the 595 different programs.

Variance

Check feature variance for features without any relevant information, find all these features where the variation is 0 and remove them from the dataset.

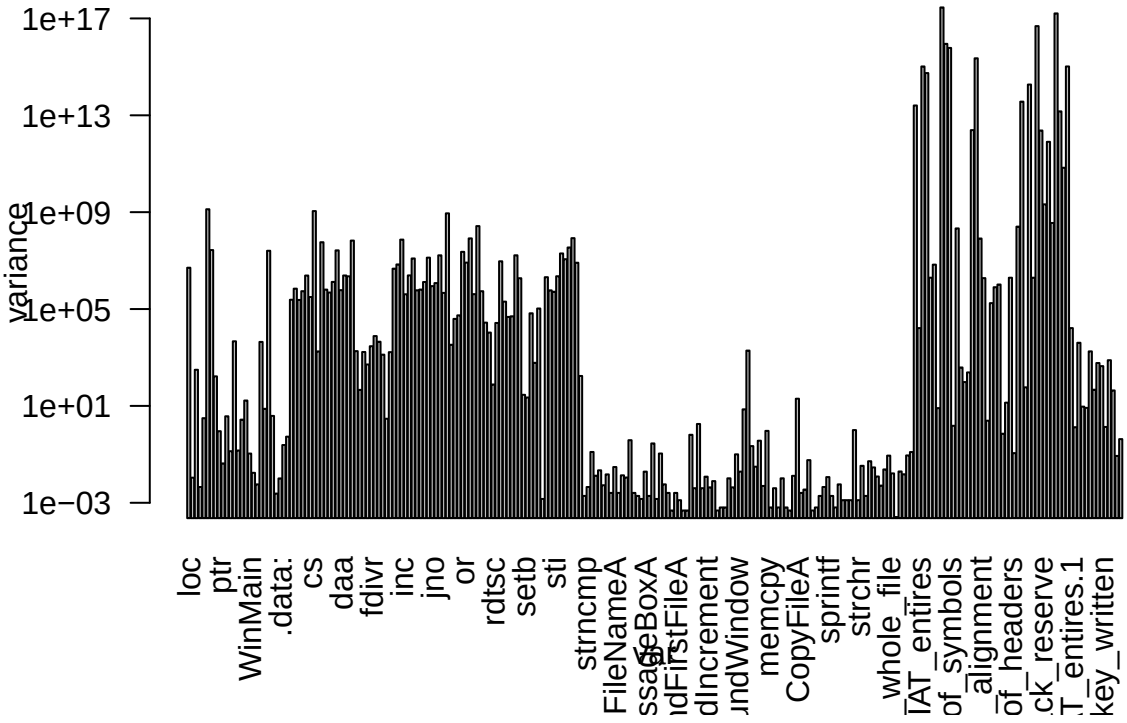
```
# Calculate the variance of each variable in the dataset
variances <- apply(data, 2, var)

# Find which variances are equal to 0
zero_variances <- which(variances == 0)
```

840 found to have a variance of 0. E.g. all variables of "count_file_renamed" are 0 so there is no benefit for the analysis or training of models. These are removed from the dataset and the remaining variances are plotted here:

```
# Exclude any variables with 0 variance
nonzero_variances <- variances[variances > 0]
# only 246 variables remain

# Create a bar plot of the variances
barplot(nonzero_variances, main="Log Variance", las=2, log="y", xlab="var", ylab="variance")
```



Feature Clusters

File/exe charakteristics

Compiler/Linker charakteristics

Imports

Datatypes

ASM and Functions used

GUI and Menu

Events

System

DLL

Inspection

Models

Test several machine learning approaches to classify unseen programs (according to their static and dynamic features) as benign or possibly malicious, differenciate between identified according to static or dynamic features. The dataset is split into two parts, one for training and anoter for testing the model. The proportion of the data allocated to the test set is 30% of the complete data, the other 70% are used for the training set.

The goal is to achieve a high accuracy (>90%) with balanced precision and recall (F1) over all classes.

Guessing

The simplest approach, though not very usefull, would be by simply guessing the classification of a program. It is expected that the accuracy would be 33.3%.

The achieved accuracy is 0.3312, close to the expected 33.3%.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312	0.1502463	0.3640138	0.3978567

Naive Bayes

Naive Bayes is a simple and fast algorithm often used for classifications. The algorithm is based on the Bayes' theorem, that the probability of an event might be related to the event based on knowledge of conditions.

$P(A|B) = P(B|A) * P(A)/P(B)$

Naive means that the features are mutually independent, so a the probability of one feature does not affect the probability of another feature. There are several Naive Bayes variants:

- Gaussian Naive Bayes where the features are normally distributed
- Bernoulli Naive Bayes where the features are binary
- Multinomial Naive Bayes where frequency of occurences of features is calculated

Once the likelihood and prior probabilities have been calculated, Naive Bayes uses Bayes' theorem to compute the probability of each class label for a given set of feature values. The class label with the highest probability is then selected as the predicted label for these set of input data. The algorithm can be effective for classifications when lots of features are involved, but performance may suffer when features are not independent or unbalanced distributed.

```
# Train the Naive Bayes model
nb_model <- naiveBayes(class ~ ., data=train_set)
nb_pred <- predict(nb_model, newdata=test_set)

# creater confusion matrix
nb_cm <- confusionMatrix(table(factor(test_set$class), nb_pred))

# Get overall accuracy and F1 scores
nb_acc <- nb_cm$overall['Accuracy']
nb_f1_c0 <- nb_cm$byClass[, 'F1'][1]
nb_f1_c1 <- nb_cm$byClass[, 'F1'][2]
nb_f1_c2 <- nb_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="Naive Bayes", Accuracy=nb_acc, "F1 Class0"=nb_f1_c0, "F1 Class1"=nb_f1_c1, "F
```

The Naive Bayes is not a route to follow further, we get a very low overall accuracy 0.2112, lower than guessing.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312	0.1502463	0.3640138	0.3978567
Naive Bayes	0.2112	0.1910112	0.0142857	0.3893805

SVM

Support Vector Machine (SVM) is a supervised machine learning algorithm used for regression or classification tasks. The algorithm tries to fit hyperplanes that separates the data into different classes. In the training finding the hyperplane with the largest margin (distance between the hyperplane and the closest data points from each class).
In 2-D the hyperplane is basically a line, that separates the classes. Non-binary classification can be done by transforming the input features in higher dimensional space where it can be separated by hyperplanes.

SVM are powerfull if the data is clearly separated, but must be used carefully with noisy data or imbalanced classes.

```
# train SVM model
svm_model <- svm(class ~ ., data=train_set)
svm_pred <- factor(round(predict(svm_model, newdata=test_set)), levels=c(0,1,2))

# creater confusion matrix
svm_cm <- confusionMatrix(table(factor(test_set$class), svm_pred))

# Get overall accuracy and F1 scores
svm_acc <- svm_cm$overall['Accuracy']
svm_f1_c0 <- svm_cm$byClass[, 'F1'][1]
svm_f1_c1 <- svm_cm$byClass[, 'F1'][2]
svm_f1_c2 <- svm_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="SVM", Accuracy=svm_acc, "F1 Class0"=svm_f1_c0, "F1 Class1"=svm_f1_c1, "F1 Cla
```

With 0.522157 we are now better than guessing, but the F1 scores are still a bit low.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.331200	0.1502463	0.3640138	0.3978567
Naive Bayes	0.211200	0.1910112	0.0142857	0.3893805
SVM	0.522157	NA	0.6452636	0.3126787

KNN

K-Nearest Neighbors (KNN) is a supervised machine learning algorithm used for regression or classification tasks. It uses the proximity to grouping to make predictions of an individual point, assuming similar points can be found close to another.

Small k values can lead to overfitting and the model becomes to sensitive to noise and outliers, whereas a large k may result in underfitting and the model does not capture patterns in the data.

The KNN algorithm will perform best with a small number of features. Therefore we don't expect KNN to perform very well.

```
knn_model <- knn(train_set[, -ncol(train_set)], test_set[, -ncol(test_set)], train_set$class, k = 3)

# creater confusion matrix
knn_cm <- confusionMatrix(table(factor(test_set$class), knn_model))

# Get overall accuracy and F1 scores
knn_acc <- knn_cm$overall['Accuracy']
```

```
knn_f1_c0 <- knn_cm$byClass[, 'F1'][1]
knn_f1_c1 <- knn_cm$byClass[, 'F1'][2]
knn_f1_c2 <- knn_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="KNN", Accuracy=knn_acc, "F1 Class0"=knn_f1_c0, "F1 Class1"=knn_f1_c1, "F1 Cla
```

Accuracy is now very good 0.9162667, and also the F1 scores around 0.9.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312000	0.1502463	0.3640138	0.3978567
Naive Bayes	0.2112000	0.1910112	0.0142857	0.3893805
SVM	0.5221570	NA	0.6452636	0.3126787
KNN	0.9162667	0.8664688	0.9292308	0.9138702

Decision Trees

Decision Trees can be used for regression and classification models. Each node in the tree represents a decision on feature which are followed until a leaf node is reached, which represents a class.

Decision Trees are a popular and intuitive machine learning algorithm used for both classification and regression tasks. They are based on a tree-like model of decisions and their possible consequences, where each internal node represents a test on a feature, each branch represents the outcome of the test, and each leaf node represents a class label or a predicted value.

The main idea behind Decision Trees is to recursively split the data into subsets that are as homogeneous as possible with respect to the class label or the target variable. The splitting criterion used to select the best feature and split point depends on the type of data and the task. For classification tasks, common splitting criteria include Gini impurity, entropy, and classification error. For regression tasks, common splitting criteria include mean squared error and mean absolute error.

During the training phase, the Decision Tree algorithm searches the feature space for the best splits that maximize the homogeneity of the resulting subsets. The process continues until a stopping criterion is met, such as a maximum tree depth, a minimum number of samples per leaf, or a maximum impurity reduction.

Once the Decision Tree has been constructed, it can be used to make predictions on new data by following the decision path from the root node to a leaf node that corresponds to the predicted class label or target value.

Decision Trees have several advantages, such as being easy to interpret and visualize, and being able to handle both categorical and numerical features. Additionally, Decision Trees can capture nonlinear and interaction effects between the features, making them suitable for complex and nonlinear datasets.

However, Decision Trees can also suffer from several limitations, such as being prone to overfitting if the tree is too complex, being sensitive to the choice of splitting criterion and the ordering of the features, and being unstable if the training data is noisy or contains outliers. To address these issues, ensemble methods such as Random Forest and Gradient Boosted Trees are often used to improve the performance of Decision Trees.

```
dt_model <- rpart(class ~ ., data=train_set)
dt_pred <- round(predict(dt_model, newdata=test_set))
dt_acc <- sum(dt_pred == test_set$class) / nrow(test_set)

# create confusion matrix
dt_cm <- confusionMatrix(table(factor(test_set$class), dt_pred))

# Get overall accuracy and F1 scores
dt_acc <- dt_cm$overall['Accuracy']
dt_f1_c0 <- dt_cm$byClass[, 'F1'][1]
dt_f1_c1 <- dt_cm$byClass[, 'F1'][2]
```

```
dt_f1_c2 <- dt_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="Decision Tree", Accuracy=dt_acc, "F1 Class0"=dt_f1_c0, "F1 Class1"=dt_f1_c1,
```

Even better accuracy 0.9237333 than with KNN, and the F1 scores are overall better and all > 0.916.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312000	0.1502463	0.3640138	0.3978567
Naive Bayes	0.2112000	0.1910112	0.0142857	0.3893805
SVM	0.5221570	NA	0.6452636	0.3126787
KNN	0.9162667	0.8664688	0.9292308	0.9138702
Decision Tree	0.9237333	1.0000000	0.9155346	0.9163253

Random Forest

Random Forest is an ensemble learning algorithm that is commonly used for classification and regression tasks. The main idea behind Random Forest is to combine multiple decision trees into a single model by randomly selecting subsets of the data and features for each tree and then aggregating their predictions.

To build a Random Forest model, the algorithm first creates a set of decision trees using a bootstrapped sample of the training data. For each tree, a random subset of the features is selected at each split to reduce correlation among the trees and to promote feature diversity.

During the training phase, each decision tree is grown using a greedy algorithm that selects the best feature and split point to minimize the impurity of the resulting node. The impurity measure used for classification tasks is typically Gini impurity or entropy, while for regression tasks it is typically mean squared error.

Once all the trees have been grown, the Random Forest algorithm aggregates their predictions by either taking the majority vote (for classification tasks) or the average (for regression tasks) of the predictions across all trees.

One of the main advantages of Random Forest is that it is a highly accurate and robust algorithm that can handle noisy and high-dimensional data. Additionally, Random Forest can provide important information on the relative importance of the different features in the data, making it useful for feature selection and interpretation.

However, Random Forest can be computationally expensive and may require tuning of several hyperparameters, such as the number of trees, the maximum depth of each tree, and the size of the feature subsets. Additionally, Random Forest may not perform well on datasets with highly correlated features or unbalanced class distributions.

```
rf_model <- ranger(class ~ ., data=train_set, num.trees=480)
rf_pred <- factor(round(predict(rf_model, data = test_set)$predictions), levels = c(0, 1, 2))
rf_acc <- sum(rf_pred == test_set$class) / nrow(test_set)

# Calculate F1 score
rf_cm <- confusionMatrix(table(factor(test_set$class), rf_pred))

# Get overall accuracy and F1 scores
rf_acc <- rf_cm$overall['Accuracy']
rf_f1_c0 <- rf_cm$byClass[, 'F1'][1]
rf_f1_c1 <- rf_cm$byClass[, 'F1'][2]
rf_f1_c2 <- rf_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="Random Forest", Accuracy=rf_acc, "F1 Class0"=rf_f1_c0, "F1 Class1"=rf_f1_c1,
```

Increasing the number of trees (to about 500) increases the model performance slightly but too high and the accuracy and F1 scores will come down again. While accuracy has gone up to 0.9541333, the average F1 score has come down slightly but still > 0.95.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312000	0.1502463	0.3640138	0.3978567
Naive Bayes	0.2112000	0.1910112	0.0142857	0.3893805
SVM	0.5221570	NA	0.6452636	0.3126787
KNN	0.9162667	0.8664688	0.9292308	0.9138702
Decision Tree	0.9237333	1.0000000	0.9155346	0.9163253
Random Forest	0.9541333	0.9422492	0.9480676	0.9620397

Gradient Boosting

Gradient Boosting is a powerful machine learning algorithm that is widely used for both regression and classification tasks. It is an ensemble learning method that combines multiple weak learners (typically decision trees) to create a strong predictive model.

The main idea behind Gradient Boosting is to iteratively train new weak learners to correct the errors made by the previous ones. In each iteration, the algorithm calculates the negative gradient of the loss function with respect to the predicted values, and then fits a new weak learner to the residual errors. The new learner is added to the ensemble, and its predictions are combined with the predictions of the previous learners using a weighted sum.

The key to the success of Gradient Boosting is the use of a gradient-based optimization algorithm to update the weights of the weak learners. By minimizing the loss function in the direction of the negative gradient, the algorithm is able to find a series of weak learners that together can approximate the true underlying function.

There are several hyperparameters that can be tuned to control the performance and complexity of the Gradient Boosting model, such as the learning rate, which determines the step size in the gradient descent algorithm, the number of weak learners, and the maximum depth of the decision trees.

Gradient Boosting has several advantages over other machine learning algorithms, such as its ability to handle complex and nonlinear relationships between the features and the target variable, its robustness to noise and outliers in the data, and its ability to handle missing values and categorical features.

However, Gradient Boosting can also be computationally expensive and prone to overfitting if the number of weak learners or the complexity of the individual trees is too high. To address these issues, techniques such as early stopping, regularization, and subsampling can be used to improve the performance and stability of the algorithm.

```
gb_model <- xgboost(data = as.matrix(train_set[, -1]),
                    label = train_set$class,
                    nrounds = 3,
                    objective = "multi:softmax",
                    num_class = 3,
                    eval_metric = "merror",
                    verbose = 0)

# convert predictions to integer class labels
gb_pred <- factor(predict(gb_model, as.matrix(test_set[, -1])), levels=c(0,1,2))

# Confusion matrix
gb_cm <- confusionMatrix(table(factor(test_set$class), gb_pred))

# Get overall accuracy and F1 scores
```

```r
gb_acc <- gb_cm$overall['Accuracy']
gb_f1_c0 <- gb_cm$byClass[, 'F1'][1]
gb_f1_c1 <- gb_cm$byClass[, 'F1'][2]
gb_f1_c2 <- gb_cm$byClass[, 'F1'][3]

ml_results <- ml_results %>%
  bind_rows(tibble(Model="Gradient Boost", Accuracy=gb_acc, "F1 Class0"=gb_f1_c0, "F1 Class1"=gb_f1_c1,
```

Well we got the "ultimate" accuracy of 1 and all the F1 scores also at the maximum.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3312000	0.1502463	0.3640138	0.3978567
Naive Bayes	0.2112000	0.1910112	0.0142857	0.3893805
SVM	0.5221570	NA	0.6452636	0.3126787
KNN	0.9162667	0.8664688	0.9292308	0.9138702
Decision Tree	0.9237333	1.0000000	0.9155346	0.9163253
Random Forest	0.9541333	0.9422492	0.9480676	0.9620397
Gradient Boost	1.0000000	1.0000000	1.0000000	1.0000000

This is extremely unlikely to happen in practice, so there might be some problems:

- Dataset is too small
- Dataset is too simple
- Data leakage (training data is used in verification)

At the moment the Gradient Boosting results must be excluded until the underlying problem is identified and fixed.

Conclusion

While some models perform not or only slightly better than guessing, like Naive Bayes and SVM. Others achieve very good performance while stay balanced, like KNN, Decision Tree and Random Forest. For some of the models some additional tuning of parameters might be beneficial.

Future Improvements

Bigger datasets like the one from Microsoft used for the (Ronen et al. 2018) "Microsoft Malware Classification Challenge" with more than 20'000 malware samples. The malwares are also finer classified into 9 different families and types (e.g. Worm, Adware, Backdoor, Trojan, TrojanDownloader, ...). Since the Microsoft dataset includes the file contents in hex format, lots of work would have been allocated to preprocessing the files. Also the dataset is quite popular in the cybersecurity research community with over 50 research papers and thesis works citing the dataset, this would violate the Capstone rules.

The models are not very refined and some tuning might increase the Accuracy (or F1 Score for the matter).

Extracting other features from the original hex-files could provide more unique features. For this a more in-depth look at the dissassembly of these malwares is required. Also some feature combination or "pools" could be interessting, like the GUI or Menu relations or Events the program is using/triggering.

Some model approaches were not tested or lead to faulty results:

- Neuronal Network
- Gradient Descent
- Gradient Boosting

System

Hardware

All above computations are done with an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz CPU with 4 and 7.49 GB of RAM.

Software

This report is compiled using R markdown with RStudio.

```
sessionInfo()
```

```
## R version 4.2.2 (2022-10-31)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Arch Linux
##
## Matrix products: default
## BLAS:   /usr/lib/libblas.so.3.11.0
## LAPACK: /usr/lib/liblapack.so.3.11.0
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] GGally_2.1.2      glmnet_4.1-6      Matrix_1.5-1      nnet_7.3-18
##  [5] xgboost_1.7.3.1   rpart_4.1.19      ranger_0.14.1     class_7.3-20
##  [9] e1071_1.7-12      httr_1.4.5        caret_6.0-93      lattice_0.20-45
## [13] forcats_0.5.2     dplyr_1.0.10      purrr_0.3.5       readr_2.1.3
## [17] tidyr_1.2.1       tibble_3.2.0      tidyverse_1.3.2   kableExtra_1.3.4
## [21] stringr_1.4.1     scales_1.2.1      lubridate_1.9.0   timechange_0.1.1
## [25] rmarkdown_2.18    benchmarkme_1.0.8 ggplot2_3.4.0     pacman_0.5.1
##
## loaded via a namespace (and not attached):
##  [1] googledrive_2.0.0     colorspace_2.0-3      ellipsis_0.3.2
##  [4] fs_1.5.2              rstudioapi_0.14       proxy_0.4-27
##  [7] listenv_0.8.0         bit64_4.0.5           prodlim_2019.11.13
## [10] fansi_1.0.4           xml2_1.3.3            codetools_0.2-18
## [13] splines_4.2.2         doParallel_1.0.17     knitr_1.41
## [16] jsonlite_1.8.4        pROC_1.18.0           broom_1.0.1
## [19] dbplyr_2.2.1          compiler_4.2.2        backports_1.4.1
## [22] assertthat_0.2.1      fastmap_1.1.0         gargle_1.2.1
## [25] cli_3.6.0             htmltools_0.5.4       tools_4.2.2
## [28] gtable_0.3.1          glue_1.6.2            reshape2_1.4.4
## [31] Rcpp_1.0.9            cellranger_1.1.0      vctrs_0.5.2
## [34] svglite_2.1.1         nlme_3.1-160          iterators_1.0.14
```

```
## [37] timeDate_4021.106     gower_1.0.0           xfun_0.35
## [40] globals_0.16.2        rvest_1.0.3           lifecycle_1.0.3
## [43] googlesheets4_1.0.1   future_1.29.0         MASS_7.3-58.1
## [46] ipred_0.9-13          vroom_1.6.0           hms_1.1.2
## [49] RColorBrewer_1.1-3    yaml_2.3.6            reshape_0.8.9
## [52] stringi_1.7.8         highr_0.9             foreach_1.5.2
## [55] hardhat_1.2.0         lava_1.7.0            shape_1.4.6
## [58] benchmarkmeData_1.0.4 rlang_1.0.6           pkgconfig_2.0.3
## [61] systemfonts_1.0.4     archive_1.1.5         evaluate_0.18
## [64] recipes_1.0.3         bit_4.0.5             tidyselect_1.2.0
## [67] parallelly_1.32.1     plyr_1.8.8            magrittr_2.0.3
## [70] R6_2.5.1              generics_0.1.3        DBI_1.1.3
## [73] pillar_1.8.1          haven_2.5.1           withr_2.5.0
## [76] survival_3.4-0        future.apply_1.10.0   modelr_0.1.10
## [79] crayon_1.5.2          utf8_1.2.3            tzdb_0.3.0
## [82] grid_4.2.2            readxl_1.4.1          data.table_1.14.6
## [85] ModelMetrics_1.2.2.2  reprex_2.0.2          digest_0.6.30
## [88] webshot_0.5.4         stats4_4.2.2          munsell_0.5.0
## [91] viridisLite_0.4.1
```

Resources

[1] Rafael Irizarry. 2018. Introduction to Data Science.https://rafalab.dfci.harvard.edu/dsbook/

[2] Malware static and dynamic features VxHeaven and Virus Total Data Set https://archive.ics.uci.edu/ml /datasets/Malware+static+and+dynamic+features+VxHeaven+and+Virus+Total#

[3] PE Format https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[4] cuckoo sandbox https://cuckoosandbox.org/

Ronen, Royi, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. 2018. "Microsoft Malware Classification Challenge." arXiv. https://doi.org/10.48550/ARXIV.1802.10135.