Classification of Malware using static and dynamic features

HarvardX Data Science Capstone Project

Raphael Kummer (GitHub)

2023-03-19

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 whereas staDynVxHeaven2698Lab 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

We are able to train different models and with the best to be able to correctly identify between benign programs and Malware (static or dynamic) with $> 95\%$ accuracy. This random forest model also maintains good F1 scores. For something usable into production a lot more data needs to be analyzed, maybe more features to be identified, try some grouping of similar features.

Analysis

Disclaimer: Not all R code will be shown in this report. For the full code listing, 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, staDynVxHeaven2698Lab.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

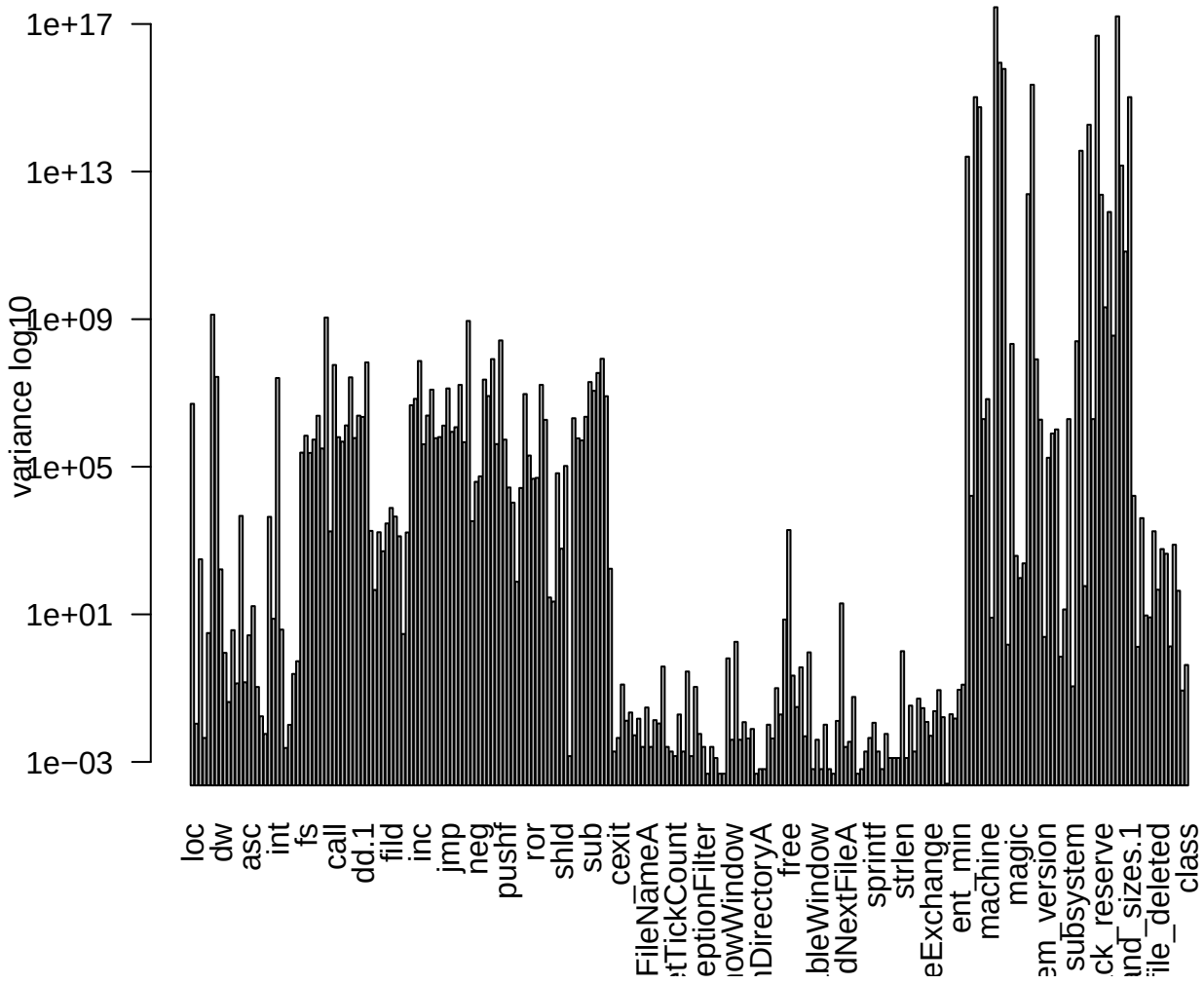
Check feature variance for features without any relevant information, find all these features where the variation is 0 and remove them from the dataset.

```
# 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)

# Exclude any variables with 0 variance
nonzero_variances <- variances[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:



When inspecting the variable names it is clear, that variables with the same purpose are grouped together.

Lets unpack the different parts of this plot further. For this extract all the column names of the nonzero variances.

```
loc, UINT, LONG, BOOL, WORD, dd, db, dw, ptr, DATA, byte, word, char, arg, asc,
align, WinMain, cookie, off, dll, HANDLE, int, entry, .rdata:, .data:, .text:,
proc
```

In the first section there are datatypes (UINT, LONG, BOOL, WORD) and some static file characteristics (handle,dll,.data,.text). Then follows a big block of ASM operations (add, mov, or, mul, nop, …)

```
es, fs, ds, ss, gs, cs, add, bt, call, cld, cli, cmc, cmp, daa, db.1, dd.1,
dec, faddp, fchs, fdiv, fdivp, fdivr, fild, fistp, fld, fstcw, fxch, imul, in,
inc, int.1, jb, je, jg, jge, jl, jmp, jno, jo, lea, loope, mov, mul, neg, not,
or, out, pop, popf, push, pushf, rcl, rcr, rdtsc, rep, ret, rol, ror, sar, sbb,
scas, setb, setle, shl, shld, shr, sidt, stc, std, sti, stos, sub, test, xchg,
xor, nop
```

also with significant variance compared to other groups, like the function calls to the Windows operating system API.

time, strncmp, _cexit, SetCursorPos, memmove, fwrite, GetModuleHandleA, ExitProcess, WriteFile, GetModuleFileNameA, CloseHandle, RegCloseKey, GetLastError, Sleep, CreateFileA, GetCommandLineA, GetTickCount, MessageBoxA, GetStartupInfoA, RegOpenKeyExA, FindClose, CreateThread, lstrlenA, UnhandledExceptionFilter, WaitForSingleObject, FindFirstFileA, DeleteFileA, TlsSetValue, TlsGetValue, RaiseException, ShowWindow, RegSetValueExA, GetSystemMetrics, InterlockedIncrement, InterlockedDecrement, SendMessageA, SetLastError, GetSystemDirectoryA, ShellExecuteA, CreateMutexA, _initterm, GetForegroundWindow, FindWindowA, exit, free, ExitWindowsEx, malloc, memset, __set_app_type, memcpy, CreateProcessA, EnableWindow, GetFileAttributesA, lstrcpyA, CreateDirectoryA, TlsAlloc, lstrcatA, CopyFileA, FindNextFileA, SetFileAttributesA, SetWindowTextA, TlsFree, WinExec, __p__fmode, __getmainargs, sprintf, __setusermatherr, RegOpenKeyA, strstr, CreateToolhelp32Snapshot, Process32Next, Process32First, strlen, strchr, strcpy, fclose, rand, strcat, fopen, InterlockedCompareExchange, strcmp, ent_whole_file, ent_mean, ent_var, ent_median, ent_max, ent_min, max_min

These show a low variance. Some of the highest variances are found in the last section.

filesize, number_of_IAT_entires, number_of_rva_and_sizes, size_code, SizeOfHeaders, machine, number_of_sections.1, compile_date, pointer_to_symbol_table, number_of_symbols, size_of_optional_header, characteristics, magic, major_linker_version, minor_linker_version, size_init_data, size_uninit_data, section_alignment, file_alignment, major_operating_system_versio minor_operating_system_version, major_image_version, minor_image_version, major_subsystem_version, minor_subsystem_version, size_of_headers, subsystem, dll_characteristics, loader_flags, number_of_imports.1, AddressOfEntryPoint, SizeOfHeaders.1, CheckSum, size_of_stack_reserve, size_of_stack_commit, size_of_heap_reserve, size_of_heap_commit, image_base, Size_image, BaseOfCode, number_of_rva_and_sizes.1, number_of_IAT_entires.1, count_mutex, files_operations, count_file_read, count_file_written, count_file_exists, count_file_deleted, count_file_copied, count_regkey_written, count_regkey_deleted, count_file_opened, count_dll_loaded, label

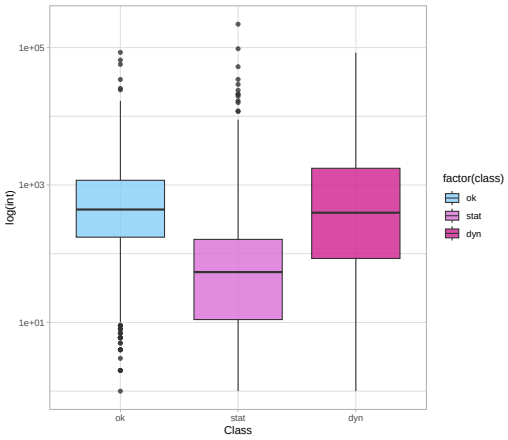
These describe program features like the operating systems characteristics, compiler used and how the program was set up (like heap and stack usage, memory allocation, headers and dlls, ...). Overall the ASM functions and mixed section where a mix of OS characteristics, compiler features and program usage are listed, show the most variance.

Feature Clusters

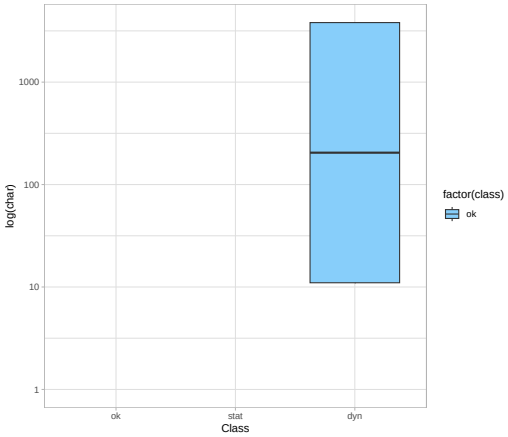
Cluster some features that are most likely to be compared against each other like data types (e.g. uint,int,char,word,byte,int…) or File specific characteristics or assembly functions. Some I could not figure out the meaning behind it (like ent_whole_file or ent_var) so these are left out.

Datatypes

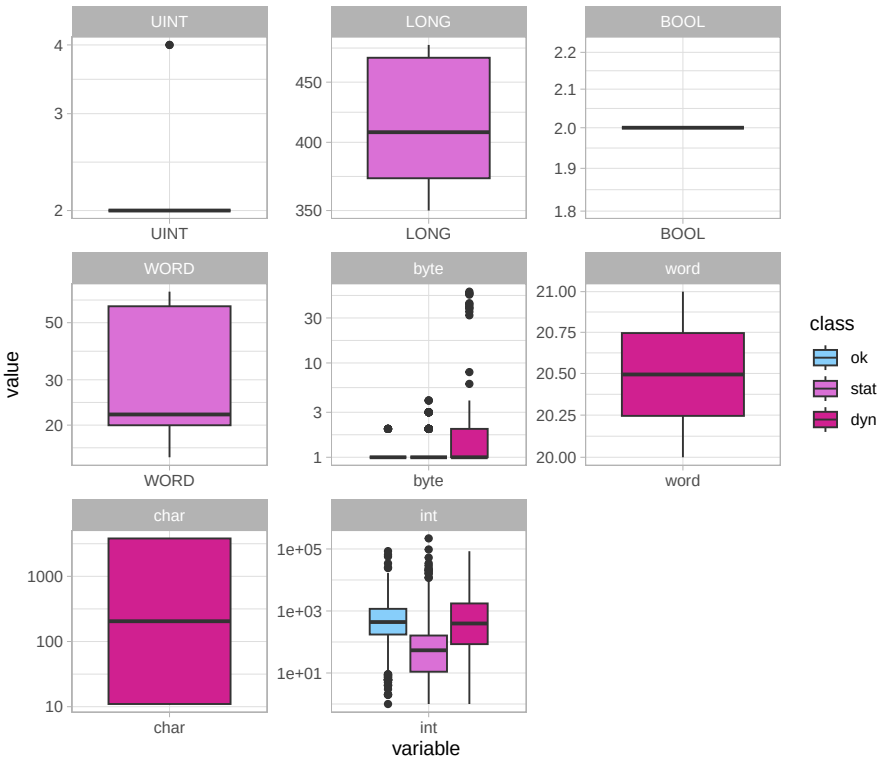
C is a strongly typed language, which means that every variable must have a specific data type assigned to it. The size of these datatypes can be different depending on the platform and/or compiler. Commonly used datatypes are BOOL (true, false), integers (like UINT, int, LONG, …) and natural numbers (float, double). There are several datatype features recorded in the dataset, lets make a group of datatypes.



The static malware has generally lower number of datatypes “int” than the other classes. But benign and dynamic Malware are quite similar, although bening is a bit denser packet around the mean. There are some types that have data only in one class, e.g. char. Some datatypes would probably make sense to condense because they are the same (word and WORD). First lets show the usage of “int” in all three classes.



Here a plot for all datatypes. Only byte and int have variables in all classes. WORD and word are probably the same and can be compared but there is no word for the benign class.

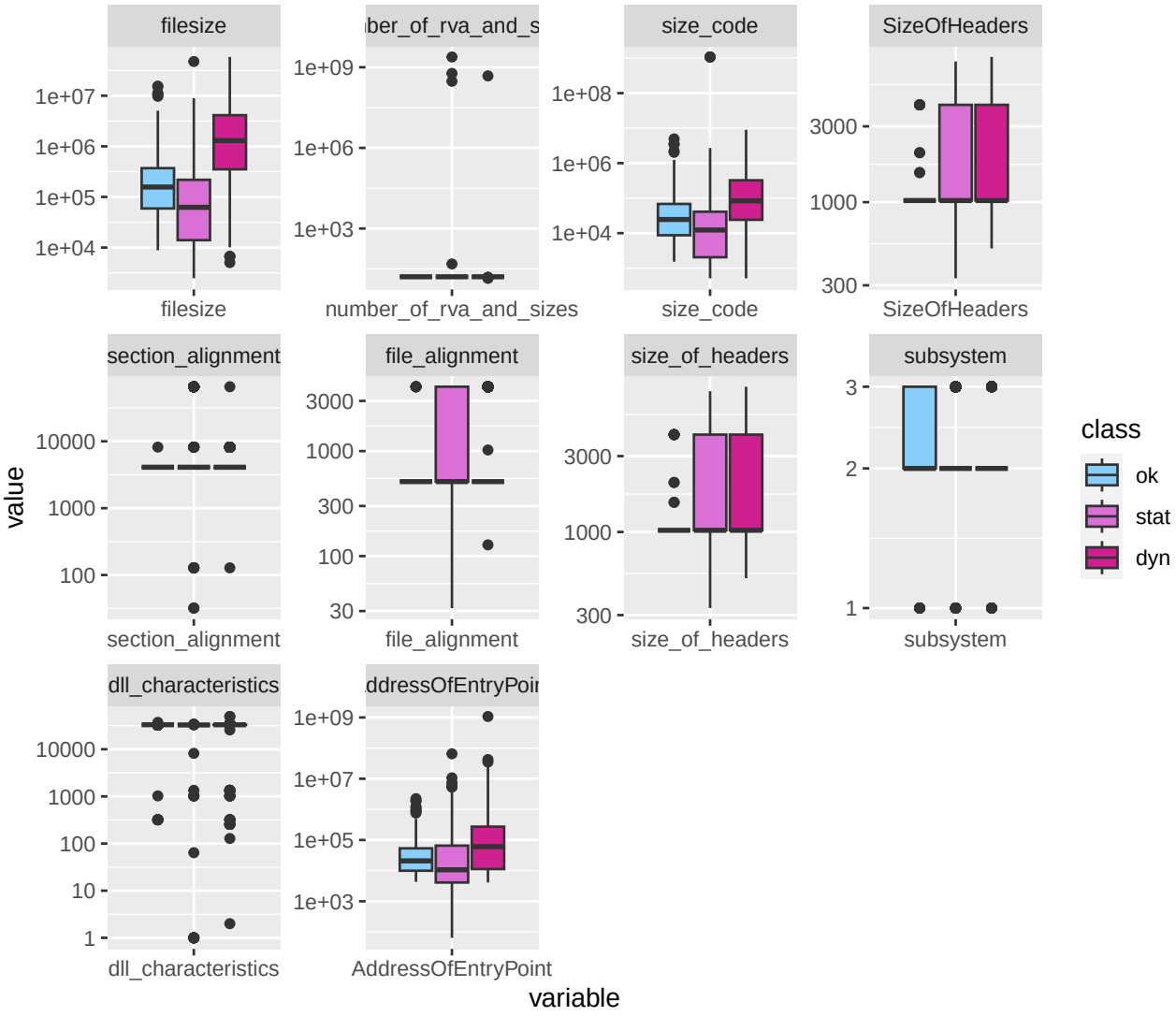


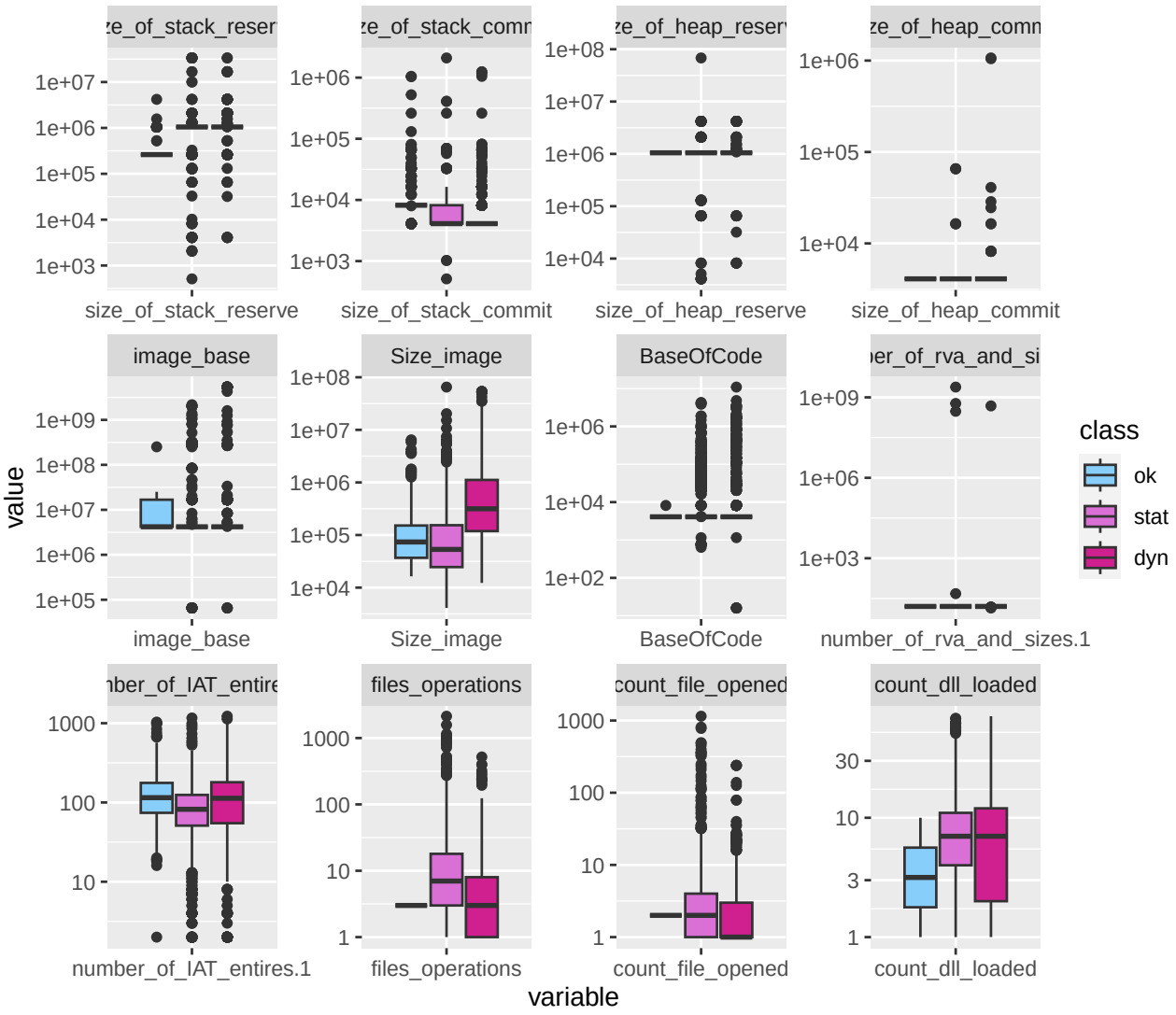
In a scatterplot Matrix it shows that some features are heavy correlated, like LONG strongly correlates with BOOL and WORD but not byte, word, char and int. But overall the data on datatypes is just too small for any good analysis.

As is, the datatypes are a bit limited in options to analyze. Some datatypes need to be assessed further and maybe combined together.

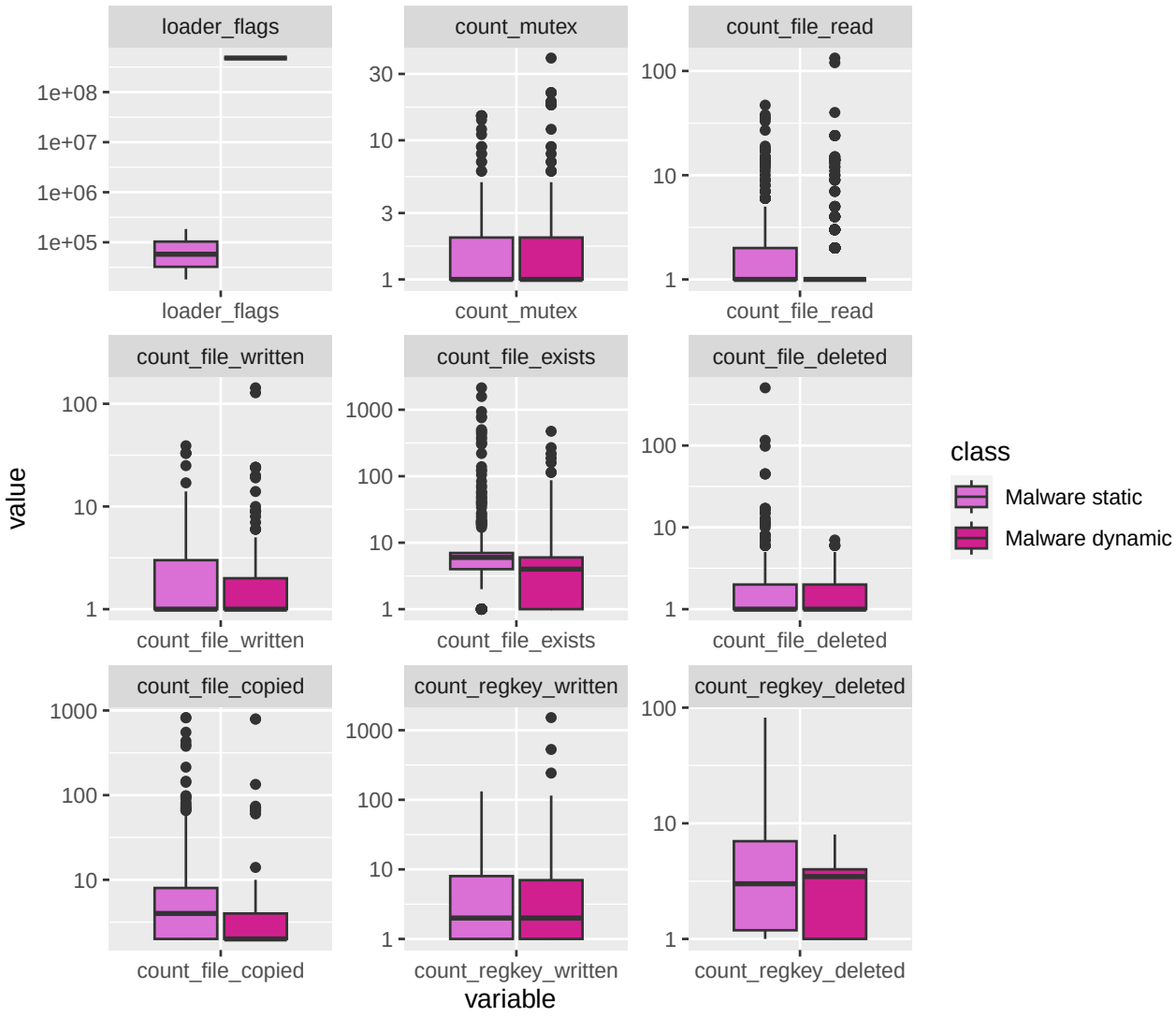
File and executable

File characteristics like size of number of sections or runtime fingerprints like loaded dll's or stack/heap sizes.

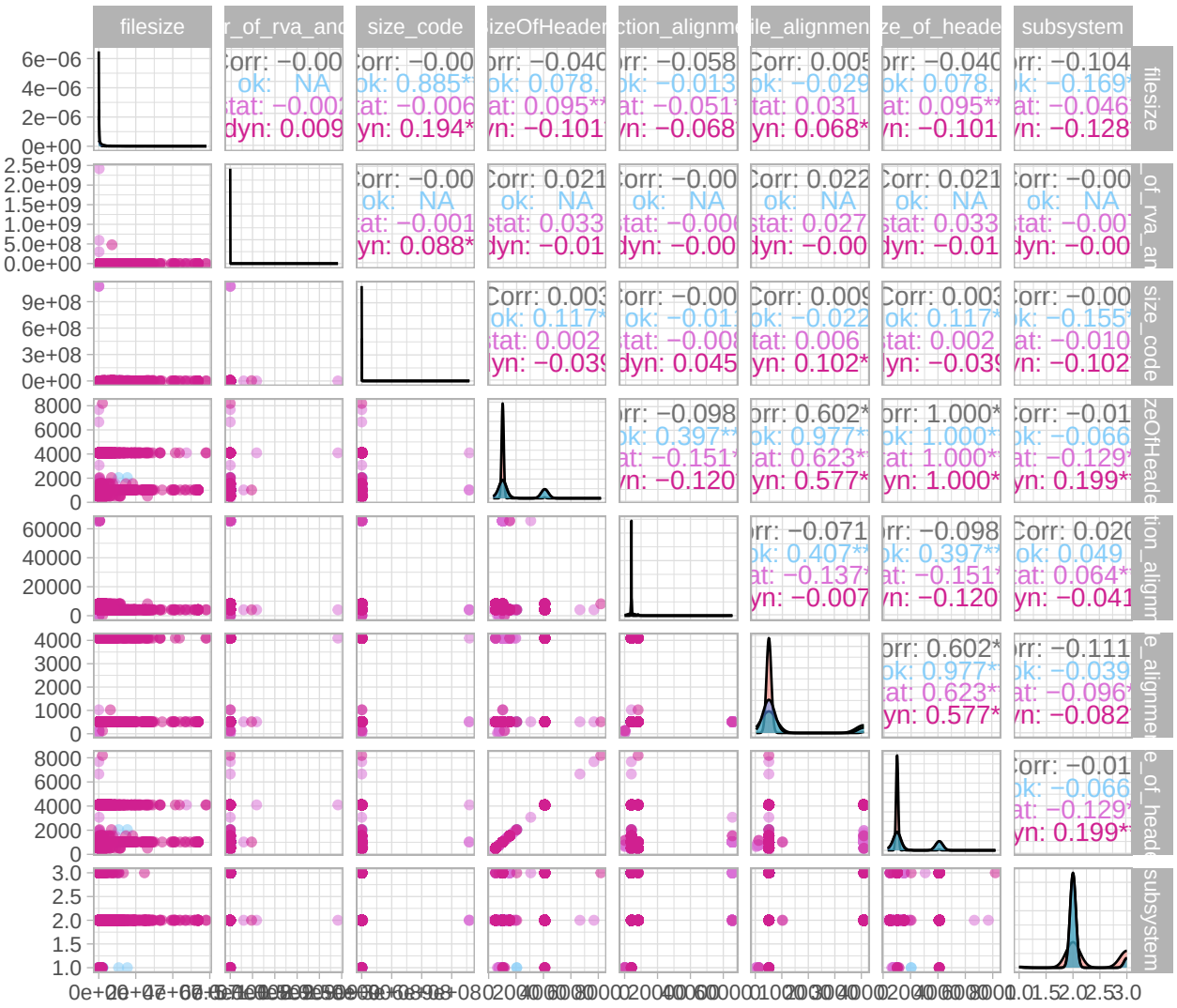




Some do not have characteristic data for all three classes, eg. .rdata, .data, loader_flags, count_mutex, count_file_read, count_file_written, count_file_exists, count_file_deleted, count_file_copied, count_regkey_written and count_regkey_deleted. These are exclusively available for static and dynamic malware. Wile most are quite similar in their distribution, loader_flags is very different between the two classes.



There are lots of features that correlate. There is lots of correlation with filesize, because number of sectors, code size, header size etc. will correlate with overall filesize. But then number_of_rva_and_sizes don't has correlation with anything other.



Then in the second half of features there is not a lot of correlation to be seen.

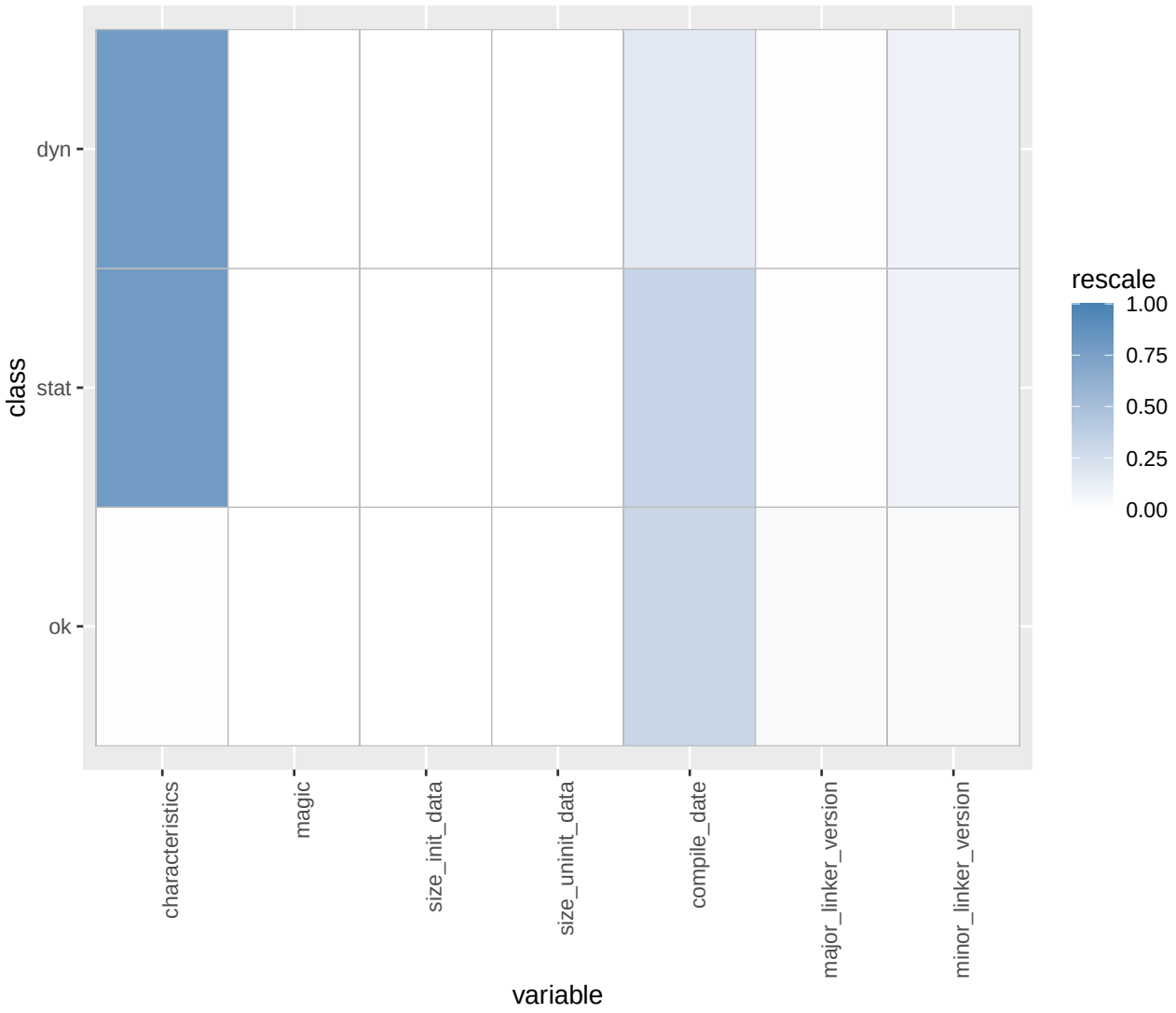

of_stack_res
_of_stack_co
_of_heap_res
_of_heap_co
image_base
Size_image
BaseOfCode
_of_rva_and_
of_stack_re
of_stack_co
of_heap_re
of_heap_co
image_base
Size_image
BaseOfCod
of_rva_and

Compiler/Linker

Compilers are used to convert the code a programmer writes to a format that computers can understand. There are several compilers (most notably gcc, vc for C/C++) and they all leave a mark. You can extract e.g. the compiler, version and date of compilation from a program header. Below a boxplot of several compiler and linker characteristics and versions.

Most interessting are probably characteristics, compile_data and the linker version.

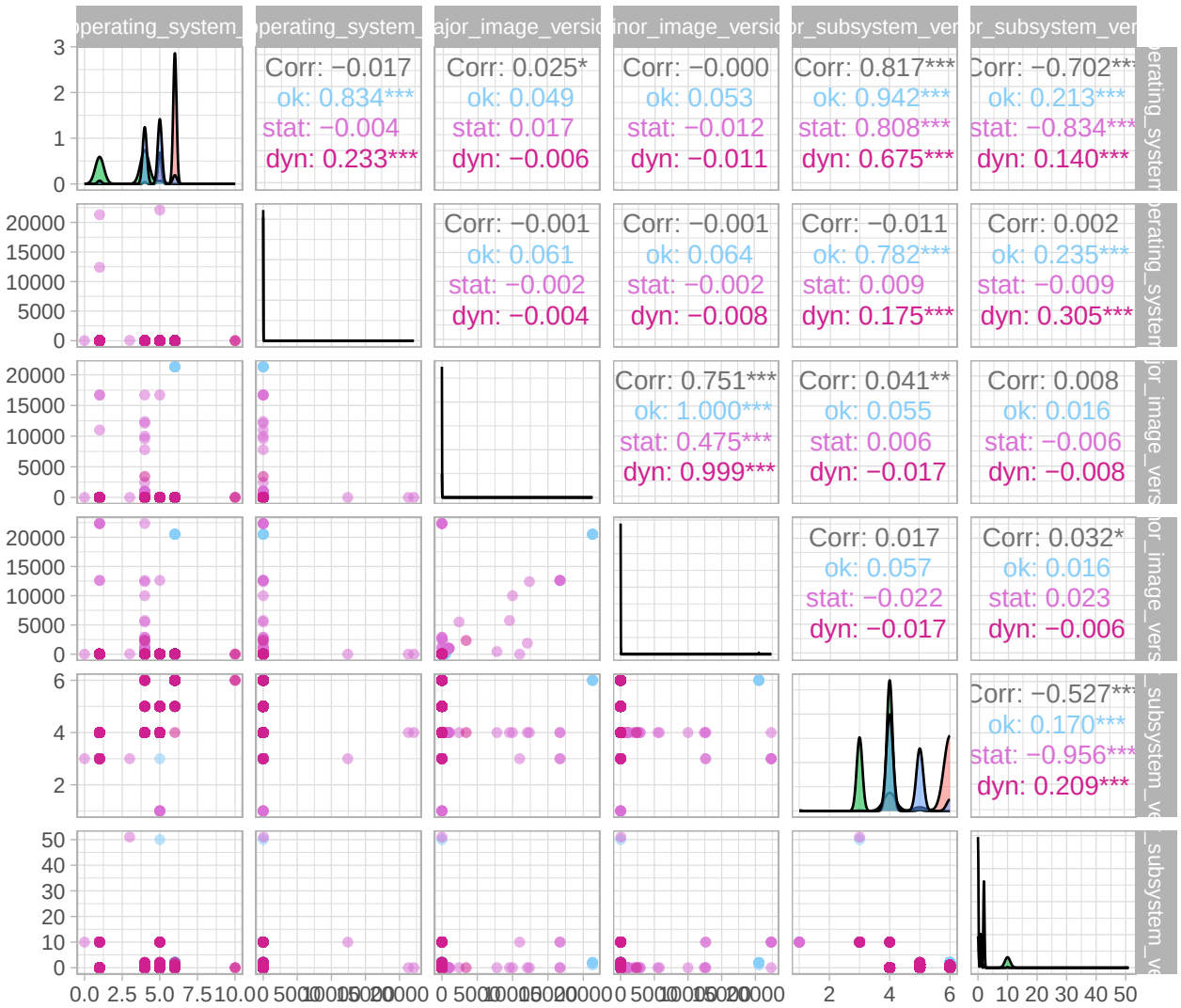
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



I think this category is very suspective, because of the limited data. A snapshot of the benign programs, most probably from Microsoft, are only taken once. A more diverse approach where such benign programs are characterized from several devices over a course of several months/years with different update states might give a more realistic picture.

OS Version

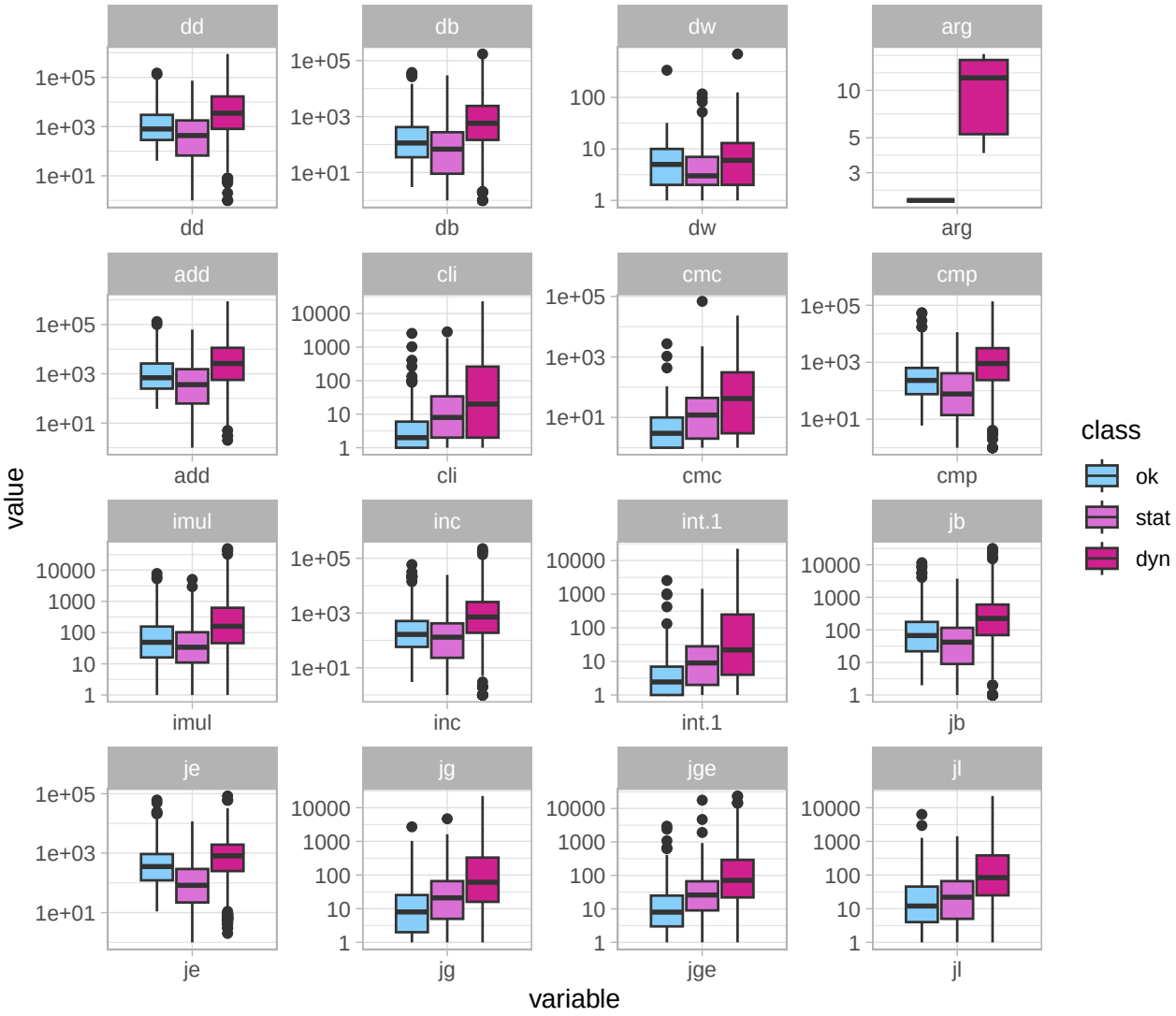
There are also some features linked to the operating system, specially the OS and image versioning.

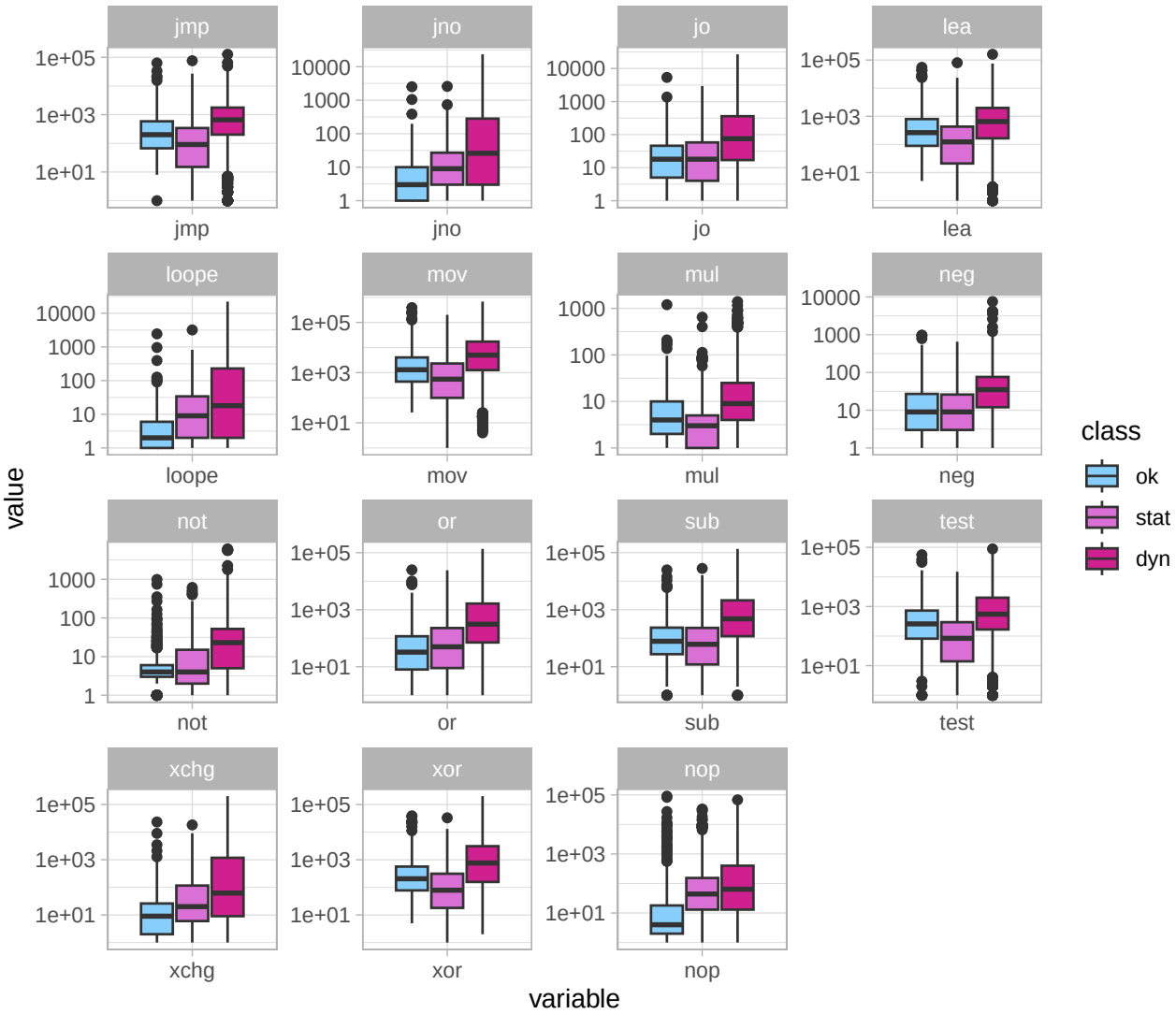


As expected with limited data, and lots probably taken around the same time with the same OS, most version correlate heavily.

ASM/Functions

There are way to many assembly functions listed for plotting. so some are selected.



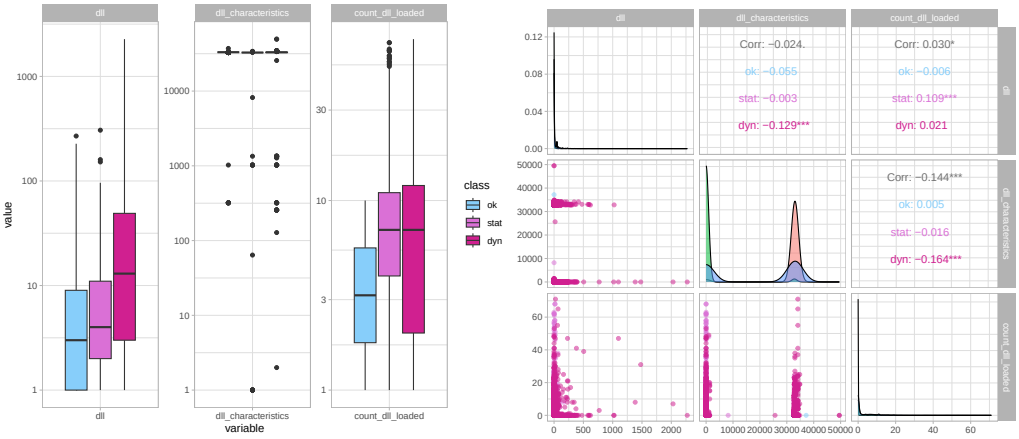


Most of the time benign applications use less ASM operations, probably due to the fact, that malware needs to add some harmfull code on top of an seemingly legitimate application. In this feature groupe we probably see the most significant difference between the three classes.

Only the first subset of matrix plots is included. Most of the ASM operations correlate with another, if only for *arg* in this case.

DLL

Dynamic Link Library (DLL) contains code and data that can be used by several programs at the same time. Common tasks and algorithms can be provided by the OS or programs to give other users defined and tested functions, like kernel32.dll that contains various system functions and resources (IO, memory management, error handling, …).



Malware generally loads more dll's than legit programs. Maybe due to the fact, that in hiding in a legitimate program it must provide the functionality of the application and some harmful code.

Models

We build and 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 20% of the complete data, the other 80% 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 useful, would be by simply guessing the classification of a program. It is expected that the accuracy would be 33.3%.

The achieved accuracy is 0.3208, close to the expected 33.3%.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208	0.1393597	0.3554604	0.3826087

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)

# Make predictions on test set
nb_pred <- predict(nb_model, newdata=test_set)
```

The Naive Bayes is not a route to follow further, we get a very low overall accuracy 0.196, lower than guessing.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960	0.1825796	0.0314136	0.3465211

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)

# Make predictions on test set
svm_pred <- factor(round(predict(svm_model, newdata=test_set)), levels=c(0,1,2))
```

With 0.5384615 we are now better than guessing, but the F1 scores are still a bit low.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208000	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960000	0.1825796	0.0314136	0.3465211
SVM	0.5384615	NA	0.6550898	0.3501401

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 have high hopes for this approach.

```
# Train a KNN model and make predictions on test set
knn_model <- knn(train_set[, -ncol(train_set)], test_set[, -ncol(test_set)], train_set$class, k = 3)
```

Accuracy is now very good 0.9272, and also the F1 scores around 0.9.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208000	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960000	0.1825796	0.0314136	0.3465211
SVM	0.5384615	NA	0.6550898	0.3501401
KNN	0.9272000	0.8867925	0.9361314	0.9261745

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. These trees are easy to visualize and interpret. But if a tree is too complex, it is prone to overfitting and must be used carefully with noisy data. For improved performance other variants might be considered like Random Forest and Gradient Boosted Trees.

```
# Train a Decision Tree model
dt_model <- rpart(class ~ ., data=train_set)

# Make predictions on test set
dt_pred <- round(predict(dt_model, newdata=test_set))
```

Even better accuracy 0.9216 than with KNN, and the F1 scores are overall better and all > 0.916.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208000	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960000	0.1825796	0.0314136	0.3465211
SVM	0.5384615	NA	0.6550898	0.3501401
KNN	0.9272000	0.8867925	0.9361314	0.9261745
Decision Tree	0.9216000	1.0000000	0.9143357	0.9134276

Random Forest

Random Forest combine multiple Decision Trees into a single model by randomly selecting a subset of data and features for each tree and combining the predictions (majority voting).

Random Forest corrects the habit of Decision Trees to ovefitting and are more robust with noisy data. They generally perform better than Decision Trees but with lower accuracy than Gradient Boosted Trees. But the additional complexity comes at the cost of more computational power required. They also don't perform well where the feature correlation is high or unbalanced.

```
# Train a Random Forest model
rf_model <- ranger(class ~ ., data=train_set, num.trees=480)

# Make predictions on test set
rf_pred <- factor(round(predict(rf_model, data = test_set)$predictions), levels = c(0, 1, 2))
```

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 0.9616, the average F1 score has come down slightly but still > 0.95.

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Guessing	0.3208000	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960000	0.1825796	0.0314136	0.3465211
SVM	0.5384615	NA	0.6550898	0.3501401
KNN	0.9272000	0.8867925	0.9361314	0.9261745
Decision Tree	0.9216000	1.0000000	0.9143357	0.9134276
Random Forest	0.9616000	0.9483568	0.9571429	0.9682948

Gradient Boosting

Gradient Boosting iteratively new weak learners (Decision Trees) to correct errors made by previous ones. Each iteration the negative gradient of the loss function is calculated. A new weak learner (Decision Tree) is fitted.
Several tuning parameters are available like learning rate (determines the step size), number of weak learners and depth of the Decision Trees. Gradient Boosting can be computationally expensive and prone to overfitting if the number of weak learners or the complexity of the individual trees is too high. Early stopping, regularization, and subsampling can be used to improve the performance and stability of the algorithm.

```
# Train a Gradient Boosting model
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)

# Make predictions on test set
# convert predictions to integer class labels
```

```
gb_pred <- factor(predict(gb_model, as.matrix(test_set[, -1])), levels=c(0,1,2))
```

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	0.3208000	0.1393597	0.3554604	0.3826087
Naive Bayes	0.1960000	0.1825796	0.0314136	0.3465211
SVM	0.5384615	NA	0.6550898	0.3501401
KNN	0.9272000	0.8867925	0.9361314	0.9261745
Decision Tree	0.9216000	1.0000000	0.9143357	0.9134276
Random Forest	0.9616000	0.9483568	0.9571429	0.9682948
Gradient Boost	1.0000000	1.0000000	1.0000000	1.0000000

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

- Overfitting
- 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

In conclusion, the use of machine learning models has proved to be an effective tool for identifying and distinguishing between benign programs and malware. The performance of the models varies, with some achieving better accuracy than others, and some requiring additional tuning of parameters to improve their performance. However, overall, the models that have been tested show promising results in terms of their ability to accurately identify malware and benign programs.

Furthermore, the use of machine learning in cybersecurity has become increasingly important in recent years, as the number of cyber threats and attacks continues to rise. As technology advances, the methods used by cybercriminals to carry out attacks become more sophisticated and difficult to detect. Therefore, the use of machine learning models, along with other security measures, is essential in preventing and mitigating the impact of cyber attacks.

Overall, the results obtained from the use of machine learning models in identifying malware and benign programs are encouraging. Further research and development in this area will undoubtedly lead to even more effective tools for combating cyber threats, and the use of machine learning will continue to play a crucial role in cybersecurity.

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. We are able to correctly identify between benign programs and Malware (static or dynamic) with $> 95\%$ accuracy and also good F1 scores:

Model	Accuracy	F1 Class0	F1 Class1	F1 Class2
Random Forest	0.9616	0.9483568	0.9571429	0.9682948

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).

The goal might be to only identify two classes, benign and Malware. For now it was interesting to see if we can classify the Malware even further into dynamic and static, but in real world applications, this might not be of interests.

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 interesting, like the GUI or Menu relations or Events the program is using/triggering. Also more specific information about what dll's are loaded or function used how much, like encryption used and how often these dll's are called.

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.

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.