# Project report: Malware detection based on system events trace

Advisor: Ehud Gudes
Author: Alex Korthny, t.z. 308911635

**8/11/2020**

## Contents

## 1. Introduction

The document describes a proposal for malware detection solution based on Machine learning (ML) approach applied to system event traces collected during execution of the malware program.

## 2. Background

There are many published works related to the usage of called API traces to train an ML model and use it for malware detection. Usually such solutions use dedicated VM (Cuckoo, for example) to collect called API traces, extract features from it and use it for model training and/or malware detection.[1] Other articles deal with API call traces for malware detection too [1,2,3,4]

Such approach showed in the past, very good level of accuracy, but , at the same time, it's hardtop use in the real world because it's not applicable to execute day to day real world programs in isolated VM environment or apply other invasive methods (detours, API monitor) to perform called API traces collection. Also, the API traces are susceptible to code morphing techniques that can mislead ML based detection solutions.

The current project proposes an alternative approach to get insight into the program activities and collect information that can be used to train an ML model and use it for malware detection.

## 3. Proposed project

The project proposes to collect per program event traces for ML model training and malware detection.

While called API trace show one specific path of the executed program control flows, matching event trace shows events that happened in the monitored system as the result of a given program execution (example: file opened, registry entry read, image loaded, etc.).Each such event may represent multiple API calls thus they are at a higher level of abstraction.

In terms of variety, called API trace have greater degree of it, while, event trace shows greater resilience vs. code morphing because it's output does not depend on the order or the nature of called API (as far those API working toward the same goal) and not reacting to an empty or failed API calls.

Also, contrary to called API traces, event traces are created and maintained by the OS itself (via Event Trace for Windows (ETW) subscription model) and do not require a Virtual Machine (VM) or any other costly invasive method to collect data.

This difference makes event traces a good candidate to be used as a source for training and malware detection with help of ML. The project selects most accurate ML method for classification from 5 algorithms that are briefly described in Appendix  C.

## 4. Methodology and performed work

### 4.1. Experimentation

#### 4.1.1. Approach

Some experimentation of the proposed approach has already been done. Experiments were performed on a VM (Hyper-V based) running windows 8.1 x86 OS.

Dedicated software was used to

- Execute selected program (with command line arguments, if required)
- Collect and store system events (ETW) traces during execution
- Extract and aggregate features from saved traces

ETW traces were collected for the next types of events:

- Process – process related events, i.e. process creation\termination, etc.
- Thread – thread related events (thread creation, termination)
- Win32 subsystem – win32 subsystem events
- Registry – registry events (registry keys and values accesses)
- Networking – networking events (connection events, inbound\outbound data events, etc)
- WMI – Windows Management Instrumentation component events
- Powershell – powershell application related events
- File – file events (example: open, create, read, write, etc.)
- Image – executable file (image) events, example: load of image
- ALPC – Asynchronous light weighted process communication subsystem events (used as backbone of remote procedure calls, for example)
- Memory – memory usage events (heap allocations, etc.)

#### 4.1.2. Extracted features

Collected traces contain the valuable information in binary format. To apply machine learning (ML) methods to the collected data, data need to be filtered and converted to format that can be used with ML methods.

The procedure of data filtering and conversion is often called "features extraction". In our case it works as follows:

- Events associated with the reviewed process are filtered out
- Per each filtered event, source of the event (ETW channel that produced the event, like "process", "memory", etc.), event name, recorded followed by event specific id number and event specific sub id are taken.
- Combination of channel name, event name, event id and sub id (secondary event identifier field used to identify event subclass) is used to create feature type (in terms of ML) that will be recorded. Example: "fileio_operationend_0_3":
    - "Fileio" marks ETW source
    - "Operationend" stands for event name
    - "0" stands for event id

- "3" stands for event sub id

### 4.1.3. Collected events distribution

The collected data contains 207 different types of events.

The table that shows distribution of collected system events, sorted by weight of event (event frequency divided by the number of processes where the event was detected) is presented in Appendix A of the document.

Figure 1 shows ETW events distribution (across all executed processes):

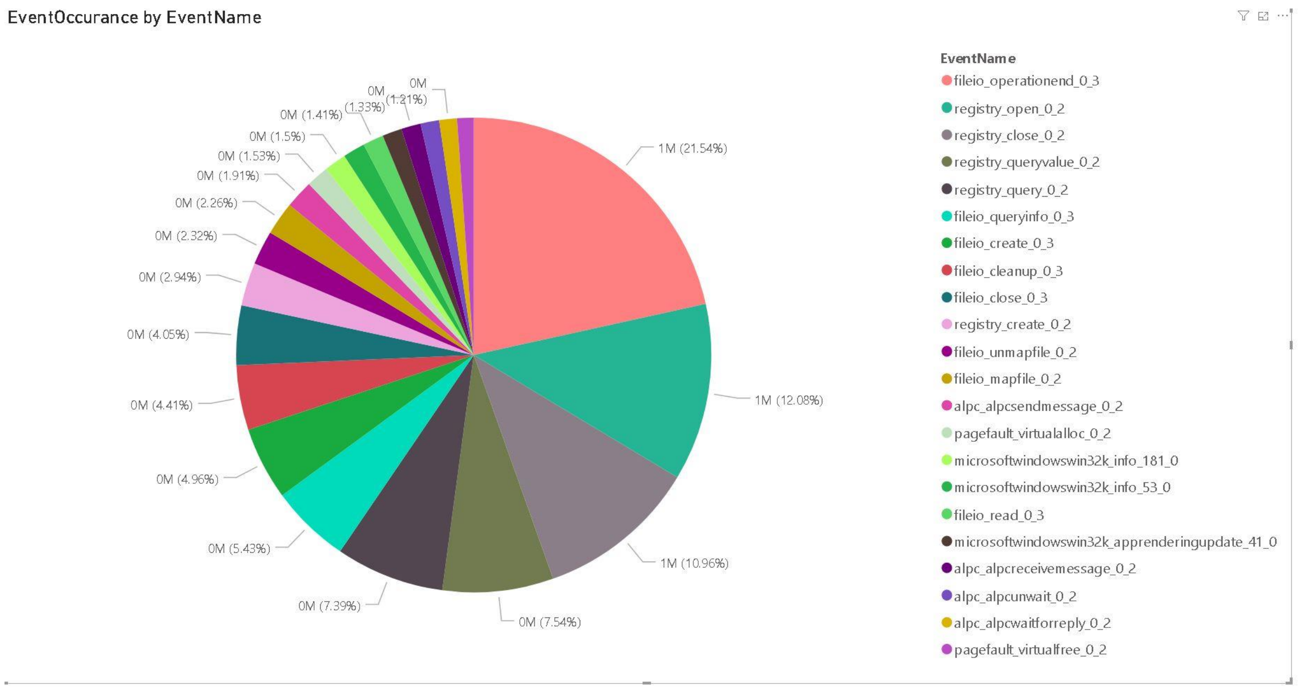

Figure 1: Events occurrence general distribution

Figure 2 shows events distribution map when event appearance per process is taken into account:

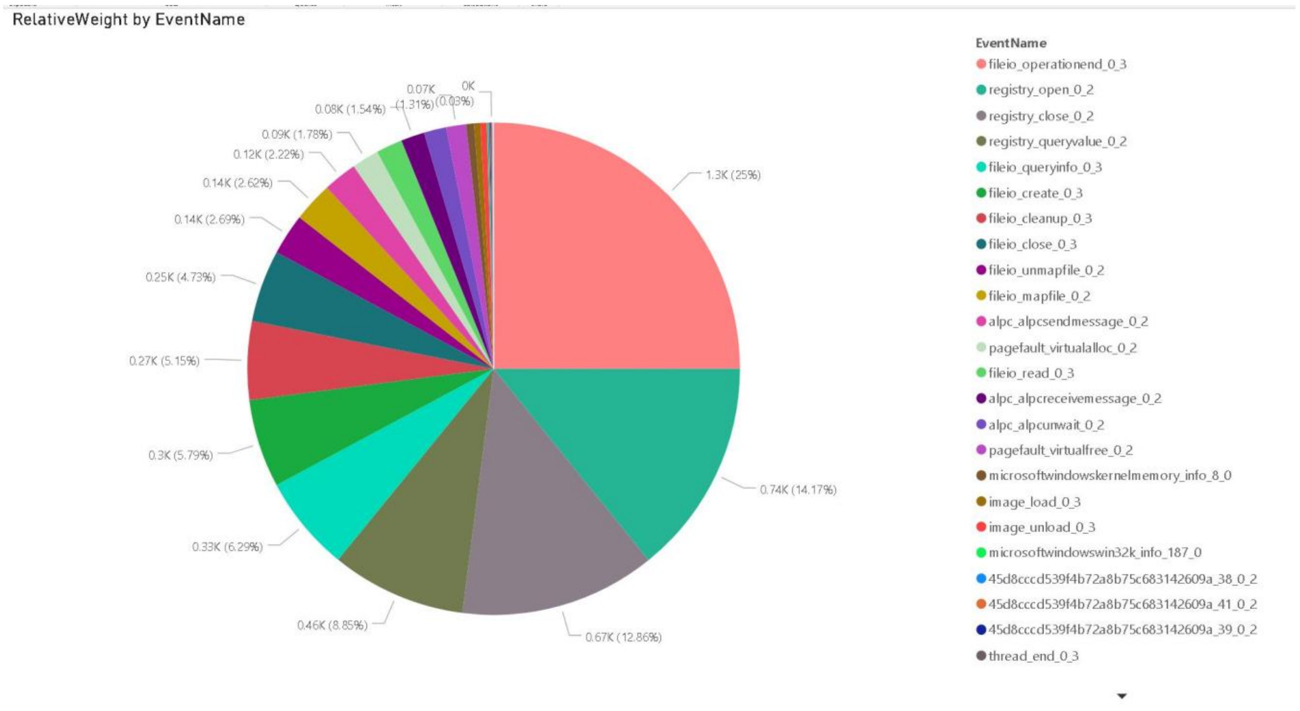

Figure 2: Events occurrence per process distribution

Comparison of the events distribution diagrams shows difference between overall events distribution (Figure 1) and events distribution per process (Figure 2) – which is a promising indication for our goal: it shows that processes indeed can be distinguished one from another based on events patterns.

The full list of events is presented in Appendix A.

#### 4.1.4. Helper software

To perform experimentation a helper program was developed. The program does the following:

- Start ETW events recording
- Execute given executable (including supplied command line arguments, if present)
- Wait for preconfigured period (2 minutes as a default)
- Terminate executed program (if not terminated by itself)
- Stop ETW session and writes to the storage the PID of the executed program (to be used in filters)

To extract features from the collected ETW traces (stored in Event Trace Log (ETL) format), two utilities were used.

- tracerpt utility was used to aggregate ETL traces and convert them to CSV format.
- Dedicated application was used to parse given collection of CSV files
  - Collected data was filtered, events not related to the experimentation scope are dropped
  - Features were extracted from the remaining data records
  - Output was produced in a form that can be used to apply ML algorithms (space separated format with each feature represented by a unique name)

- Summary report was produced

### 4.1.5. Benign programs

The table that shows benign and malicious programs that are used during experimentation (so far) is presented in Appendix B, the table contains only first 100 benign and 100 malware programs, as an illustration.

The experimentation was performed on well-known Windows applications, web browsers and generally avail utilities. In total 291 benign programs were executed.

Figure 3 illustrates the process of features extraction for benign applications:

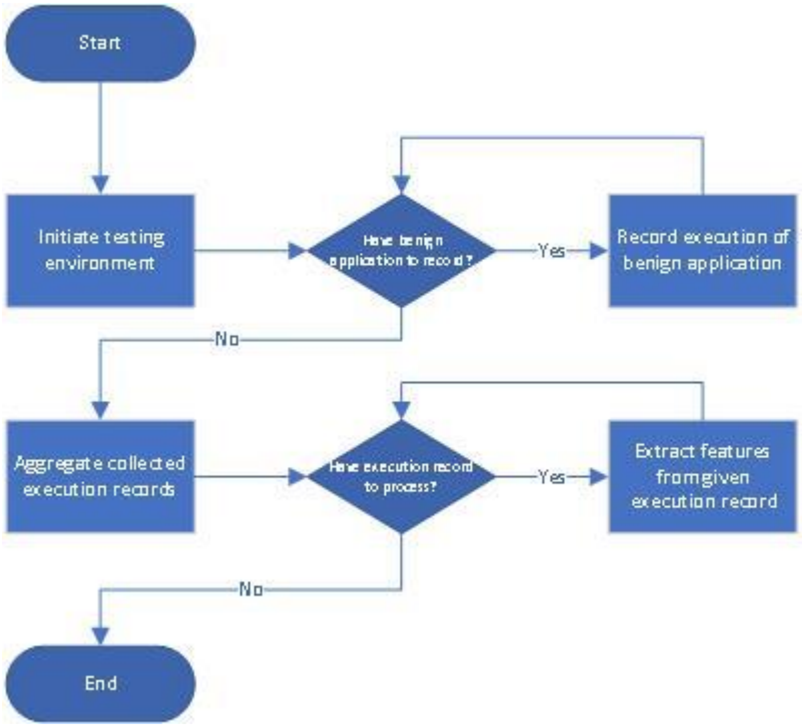
Start
Initiate testing environment
Have benign application to record?
Yes
Record execution of benign application
No
Aggregate collected execution records
Have execution record to process?
Yes
Extract features from given execution record
No
End

Figure 3: Features extraction for benign case

### 4.1.6. Malware

Malware programs were retrieved from multiple sources, including

- theZoo GitHub repository
- VxHeaven repository archive

Contrary to benign applications, safe execution of malware samples requires additional measures

- Isolated execution environment (no network connectivity)
- Disable AV software
- Extraction of execution recording after each malware sample execution (to avoid data loss)
- Reset of execution environment (testing VM) after each malware sample execution (to avoid impact of one malware sample on another)

- Additional virtual hard disk volume was used to prepare each malware sample for execution and extraction of the results. The following steps were used to execute the malicious programs:
    - Prior to malware sample execution the volume was mounted on the master PC (PC that hosted the test VM)
    - One-time preparation (done once per all tests): encrypted storage with malware samples created on the mounted volume. Encryption was needed to avoid accidental execution of malware sample and prevent sample removal by AV software
    - Execution script with path to the executed sample of malware copied to the volume
    - In context of the script selected sample of malware being decrypted and executed by the trace collection helper program mentioned above. Results of the execution preserved at the volume.
    - Prior to the execution of the testing VM, the virtual volume with the execution script is detached from the master PC
    - Execution time of VM instance with malware sample execution script set to 5 minutes, it is enough to load the VM, execute the script (set to be limited by 2 minutes) and preserve execution results on the virtual volume
    - After 5 minutes of execution, the test VM being forcefully shut down by the master PC
    - The virtual volume attached to the master PC and the execution results (trace logs) were copied over to the master PC
    - The testing VM was reset to the initial state (to ensure that executed malware won't leave any trace on it) and ready for tests with next malware sample

Malware samples from theZoo collection were represented by single example per each malware family, overall, about 55 samples. Malware examples from VxHeaven collection belonged to the following malware families:

- Backdoor-Win32-Poison
- Backdoor-Win32-SdBot
- Trojan-Downloader-Win32-Banload
- Trojan-Downloader-Win32-FraudLoad
- Trojan-Downloader-Win32-Small
- Trojan-Downloader-Win32-VB
- Trojan-PSW-Win32-LdPinch
- Trojan-Win32-Delf

From each family there were about 100 different samples (with different hash values) taken for experimentation.

Figure 4 illustrates the process of features extraction for malware samples:

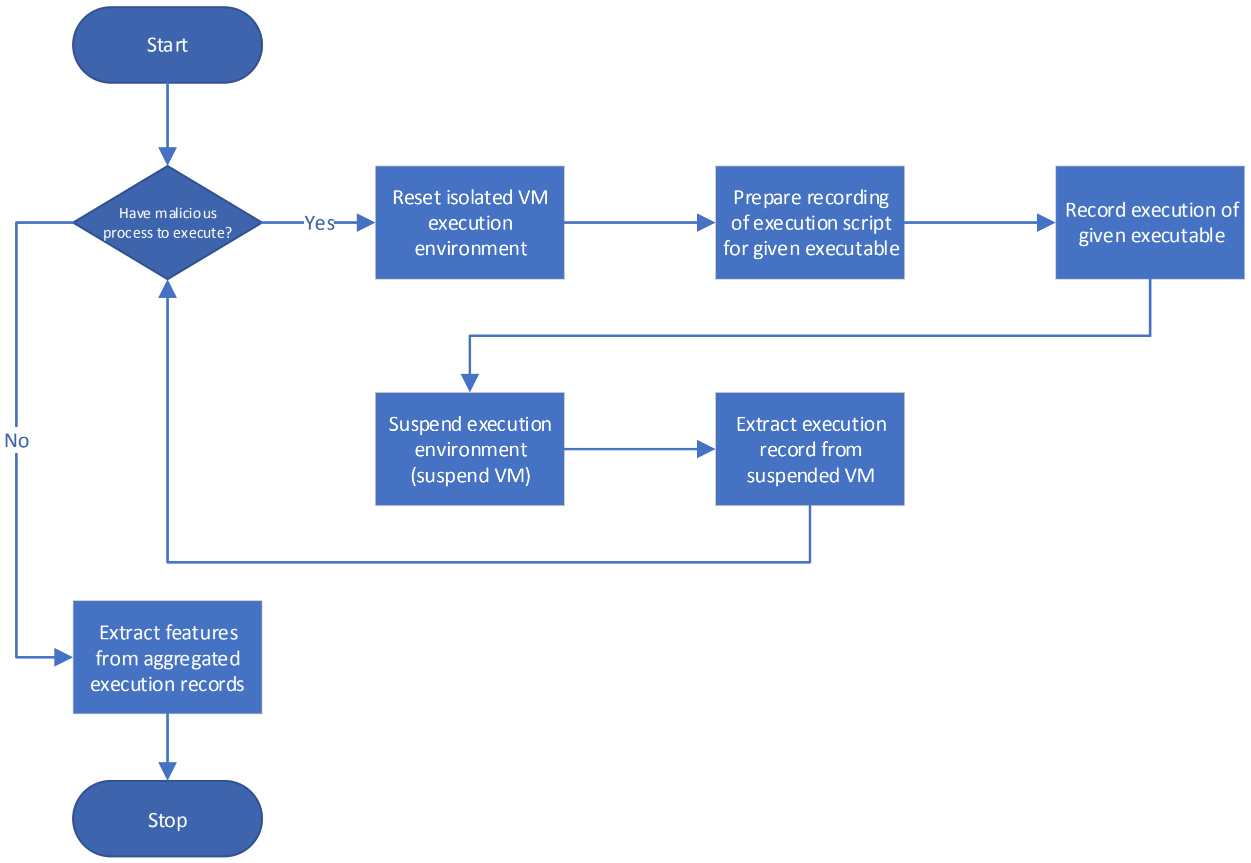
Start
Have malicious process to execute?
Yes
No
Reset isolated VM execution environment
Prepare recording of execution script for given executable
Record execution of given executable
Suspend execution environment (suspend VM)
Extract execution record from suspended VM
Extract features from aggregated execution records
Stop

Figure 4: Features extraction for malware case

Results were collected from more than 800 malware samples, after removal of duplicates (removal of execution traces that are similar), classification was performed on 587 unique malware execution traces.

Note: at this point we already can state that usage of ETW for event tracing showed good results vs. non-essential malware modifications: more than 100 malware samples (13%) were detected as duplicates on another malware samples despite different hash values of executables.

## 5. Features extraction and classification methodology

Features extraction and classification of the collected features was performed based on similar API call tracing methods that were adopted to achieve above 90% accuracy of detection.

Specifically, the method suggested by article [4] was used for features extraction and classification of malware samples vs. benign programs.

Figure 5 describes the control flow:

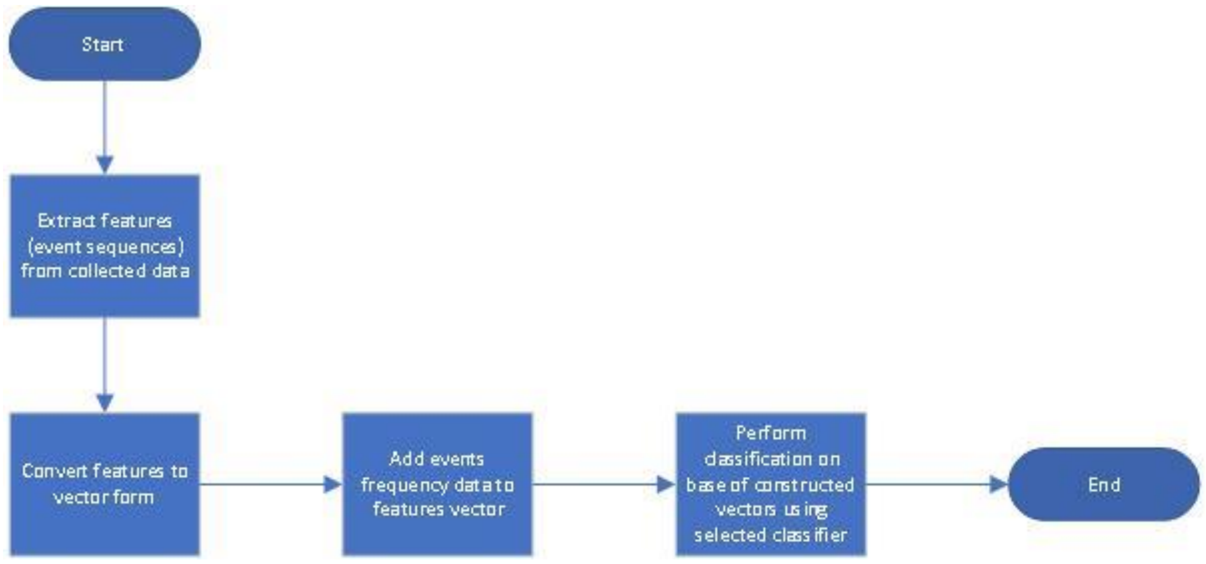
Start
Extract features (event sequences) from collected data
Convert features to vector form
Add events frequency data to features vector
Perform classification on base of constructed vectors using selected classifier
End

Figure 5: Features extraction and Classification procedure

The detailed description of the features extraction methodology is presented in the next section.

The detailed description of the classification methodology is presented in Appendix C.

## 6. Features extraction details

The features extraction methodology uses the following definitions based on referred article [4].

$D = \{d_1,d_2,\dots,d_n\}$
$V = \{w_1,w_2,\dots,w_n\}$
$f_d(w)$= frequency of the word $w \in V$ in $d \in D$
$\vec{t}_d=(f_d(w_1), f_d(w_2),\dots, f_d(w_n))$

D is the corpus and d represent a document.
V is the vocabulary and w represent each word that appears in the corpora

The algorithm uses the “bag of words model” described in article [4]: in context of the model we take in account presence of a certain word in a document, the exact location of the word not important.
The article author substituted OS API traces for words, we use a similar approach and substitute extracted ETW events for words.
As a result, for us, set V represents the vocabulary set of all unique ETW events (represented by $w_n$ where n is the index of the ETW event in the vocabulary).
Each document (in our case it’s event trace) is represented by how many times every $w_n$ occurs in the document, as represented by $\vec{t}_d$, where $f_d(w_n)$ represents the frequency of the ETW event of index n in the log d.

By analogy with article [4], we use the same n-gram model as demonstrated in the article.
We also use the same consideration for TF-IDF and TF weighting (cited from the article [4]):

*“Term Frequency (TF) refers to the number of times a certain word occurs in a corpus. Inverse Document*

*Frequency (IDF) refers to the amount of times the word occurs throughout the document D. The TF-IDF weight of a term is computed as following:*

*tf(w,d)=$f_d$(w): frequency of w in document d*
*idf(w,D)=*$log\,\frac{1+|D|}{1+df(d,w)}$

*Where df(d,w) is the number of documents the word w appears in.*

*This is a logarithmically scaled value of the number of documents in the corpus divided by the number of times word w appears throughout the corpus.*

*tfidf(w,d,D)=tf(w,d) x idf(w,D)*

*The tfidf value increases proportionally by the frequency of w in a document, decreases proportionally by the log of the frequency of w in the corpus. The assumption is that a word that is more prevalent throughout the corpus is more likely to be less significant. The resulting vectors comprised of raw tfidf values that represent each document are normalized using the Euclidean norm:*

$v_{norm}=\frac{v}{\sqrt{v_1^2+v_2^2+\cdots+v_n^2}}$
*"*

Next, the gives set of event traces is transformed with help of either tf (CountVectorized) or tfidf (TfidfVectorized) to features vectors. This vector is supplied as input for a classification method (several well-known methods were used to find out best result)

## 7. Classification algorithms overview

See details in Appendix C.

#### 7.1.1. Ridge classification algorithm

This algorithm first converts binary targets to {-1, 1} and then treats the problem as a regression task, optimizing the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients. The predicted class corresponds to the sign of the regressor's prediction. For multiclass classification, the problem is treated as multi-output regression, and the predicted class corresponds to the output with the highest value.

#### 7.1.2. Perceptron classification algorithm

The **Perceptron** is a simple classification algorithm suitable for large scale learning. By default:

- It does not require a learning rate.
- It is not regularized (penalized).
- It updates its model only on mistakes.

#### 7.1.3. Passive-Aggressive classification algorithm

The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are like the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter C.

#### 7.1.4. Multinomial Naïve Bayes classification algorithm

The multinomial Naive Bayes classification algorithm is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

#### 7.1.5. Linear SVM classification algorithm

*Support Vector Machines* belong to the discriminant model family: they try to find a combination of samples to build a plane maximizing the margin between the two classes. Regularization is set by the C parameter: a small value for C means the margin is calculated using many or all of the observations around the separating line (more regularization); a large value for C means the margin is calculated on observations close to the separating line (less regularization).

All the classifiers above were implemented by the Python machine learning libraries.

## 8. Python implementation details

The following python script (parts of it) was used to perform classification of programs based on the collected data:

```
#Python code sample parts to demonstrate preprocessing and classification of the project

def benchmark(clf, vecdesc, ncolumns, train_labels, test_labels, showTopFeatures=False):

  print('_'*60)

  print("Training: ")

  print(clf)

  t0 = time()

  clf.fit(X_train, train_labels)


  train_time = time() - t0

  t0 = time()

  pred = clf.predict(X_test)

  test_time = time() - t0

  score = metrics.accuracy_score(test_labels, pred)
```

```
    if hasattr(clf, 'coef_'):
        print("dimensionality: %d" % clf.coef_.shape[1])
        print("density: %f" % density(clf.coef_))
        print()
    clf_descr = str(clf).split('(')[0]
    mCount = 0
    for i in test_labels:
        if i != 'Bening':
            mCount+=1
    maliciosTestRate = mCount/len(test_labels)
    predictaments = pred.tolist()
    return vecdesc, clf_descr, ncolumns, score, maliciosTestRate, train_time, test_time
#=================================== End of Utility code


#=================================== Calculation start line
results = [] # future storage of results
# raw data location
dataset_url = os.getcwd() + '//data//alltraces_100p_average_unmapped.csv'


# read the input data set
data = pd.read_csv(dataset_url, delimiter=' ', dtype=str)
y_names = data.pop('FileName')
data_length = len(data.columns)
data.drop_duplicates(inplace=True)
rng = RandomState()  # random seed


# Perform 100 iterations to ensure that results are statistically reliable
```

```
for iteration in range(100):

    # randomly split 70% to train, 30% to test

    train = data.sample(frac=0.7, random_state=rng)

    test = data.loc[~data.index.isin(train.index)]


    # Prepare train data

    X = train.to_numpy()[:, 1:]

    y = train.to_numpy()[:, 0]

    X = X.astype(str)

    X = np.char.lower(X) # normalize by char case

    X_list = X.tolist()

    X = flatten_list_of_lists(X_list) # transform data format (flatten) to form that acceptable by classifier

    y = y.tolist()


    # Prepare test data

    X_t = test.to_numpy()[:, 1:]

    y_t = test.to_numpy()[:, 0]

    X_t = X_t.astype(str)

    X_t = np.char.lower(X_t)

    X_list = X_t.tolist()

    X_t = flatten_list_of_lists(X_list) # transform data format (flatten) to form that acceptable by classifier

    y_t = y_t.tolist()


    # collect results per each supported features extraction method

    for vectorizer in (

        TfidfVectorizer(token_pattern='(?u)\\b\\w+\\b', ngram_range=(nGRAM1, nGRAM2), min_df=1,
use_idf=True,smooth_idf=True),

        CountVectorizer(ngram_range=(nGRAM1, nGRAM2))):
```

```
    vec_descr = str(vectorizer).split('(')[0]

    analyze = vectorizer.build_analyzer()

    X_train = vectorizer.fit_transform(X)

    X_test = vectorizer.transform(X_t)

    clf_LSVC = Pipeline([

      ('feature_selection', SelectFromModel(LinearSVC(penalty="l1", dual=False,

                                                      tol=1e-3))),

      ('classification', LinearSVC(penalty="l2"))])


    # collect results per each supported classification method

    for clf, name in (

            (RidgeClassifier(tol=1e-2, solver="sparse_cg"), "Ridge Classifier"),

            (Perceptron(n_iter_no_change=50), "Perceptron"),

            (PassiveAggressiveClassifier(n_iter_no_change=50), "Passive-Aggressive"),

            (MultinomialNB(alpha=.01),"MultinomialNB"),

            (clf_LSVC,"LSVC")):

        print('=' * 80)

        print(name)

        results.append(benchmark(clf,vec_descr,data_length, y, y_t))


list_header = ["Preprocessing", "Classification", "Trace Size", "Score", "Malicious rate", "Train time",
"Test time"]

# using list comprehension, convert tupple to list

out = [list(ele) for ele in results]


# write to output file

outputPath = os.getcwd() + '//output//calculation_results.csv'

with open(outputPath, 'w') as myfile:
    wr = csv.DictWriter(myfile, fieldnames=list_header)

    wr.writeheader()

    for ele in out:

        wr.writerow({"Preprocessing":ele[0], "Classification":ele[1], "Trace Size":ele[2], "Score":ele[3],
"Malicious rate":ele[4], "Train time":ele[5], "Test time":ele[6]})
```

Figure 6 illustrates the workflow of the script:

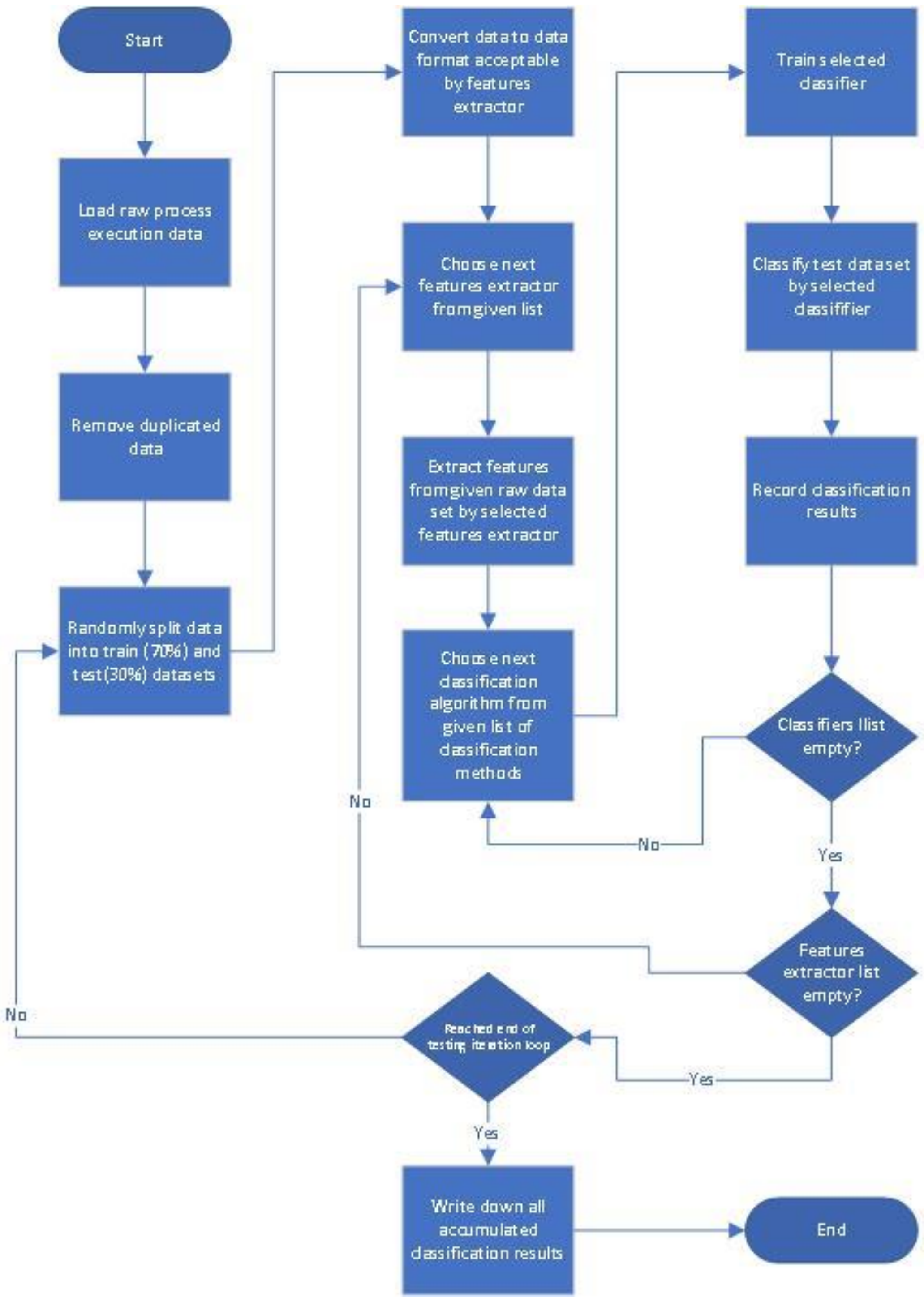
Start
Load raw process execution data
Remove duplicated data
Randomly split data into train (70%) and test (30%) datasets
Convert data to data format acceptable by features extractor
Choose next features extractor from given list
Extract features from given raw data set by selected features extractor
Choose next classification algorithm from given list of classification methods
Train selected classifier
Classify test dataset by selected classifier
Record classification results
Classifiers list empty?
No
Yes
Features extractor list empty?
No
Yes
Reached end of testing iteration loop
No
Yes
Write down all accumulated classification results
End

Figure 6: Python code control flow diagram

Following Python classes were used for collected features preprocessing:

- CountVectorized
- TfidfVectorized

Classification of the collected and preprocessed data was performed with the help of the next Python classification classes:

- Ridge
- Perceptron
- Passive-Aggressive
- MultinomialNB
- Linear SVC

## 9. Experimental evaluation

We use the next set of definitions in our work:

- Accuracy = $\frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$
- Precision = $\frac{TP(True\ Positive)}{TP+FP(False\ Positive)}$
- ROC (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:
  - True positive rate (TPR)
  - False positive rate (FPR)

  TPR = $\frac{TP}{TP+FN}$

  FPR = $\frac{FP}{TP+FN}$

  A ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives.
- AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).
  AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example.

Figure 7 demonstrates the accuracy comparison between different classification methods used in the experiment:

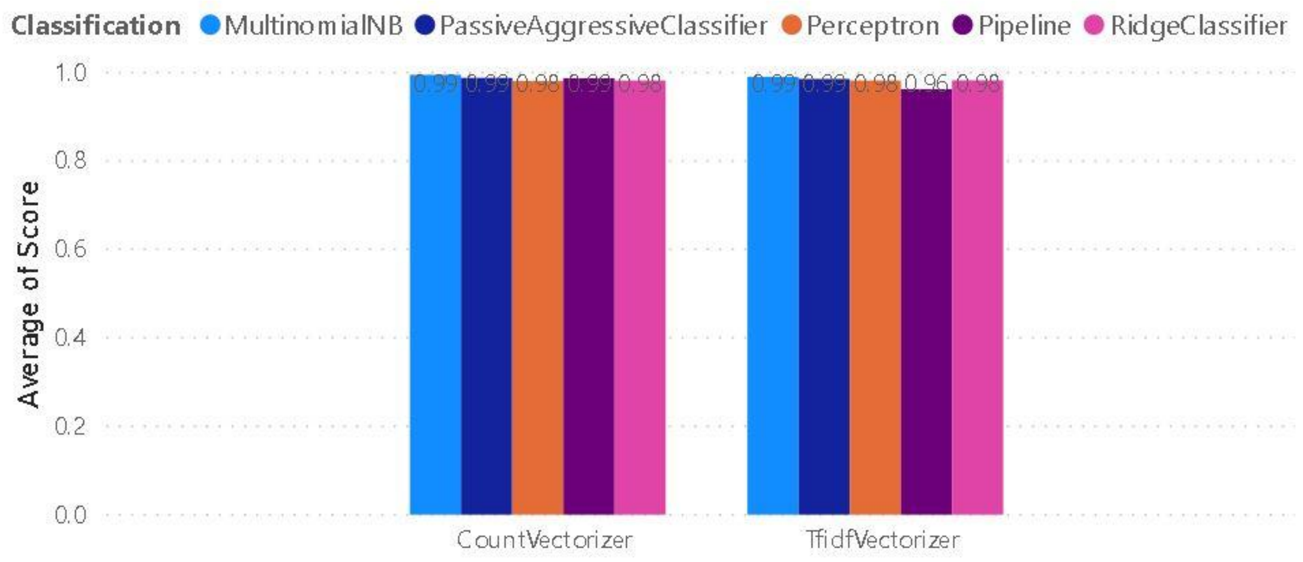

Figure 7: Accuracy of different classification methods

Figure 8 demonstrates the training time comparison between the different methods:

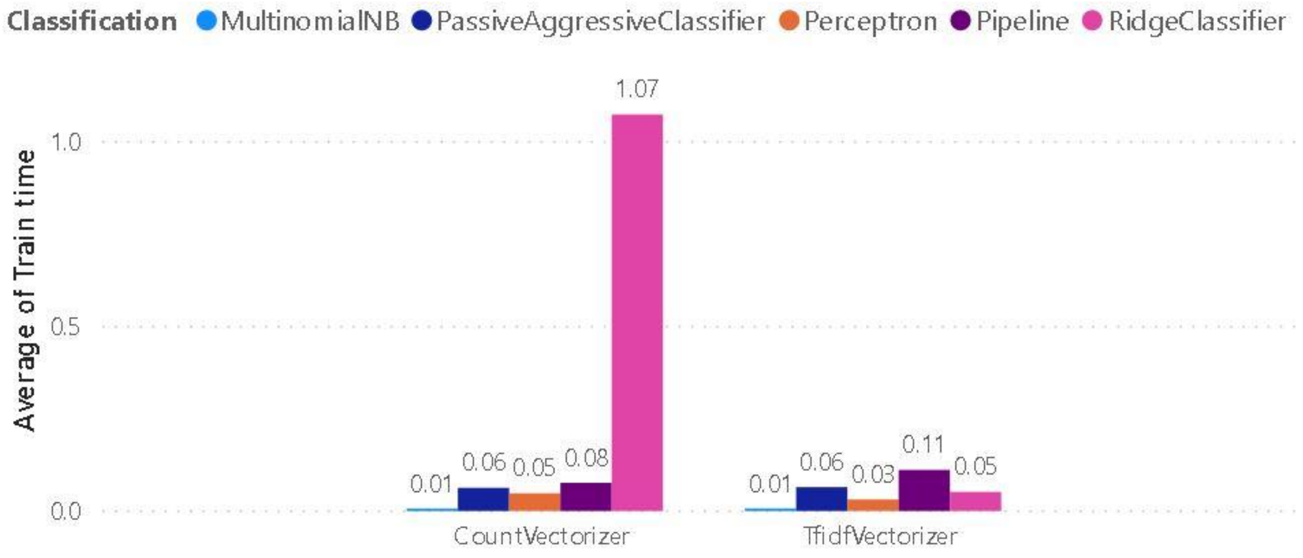

Figure 8: Training time per classification method

Figure 9 demonstrates the testing time comparison between the methods:

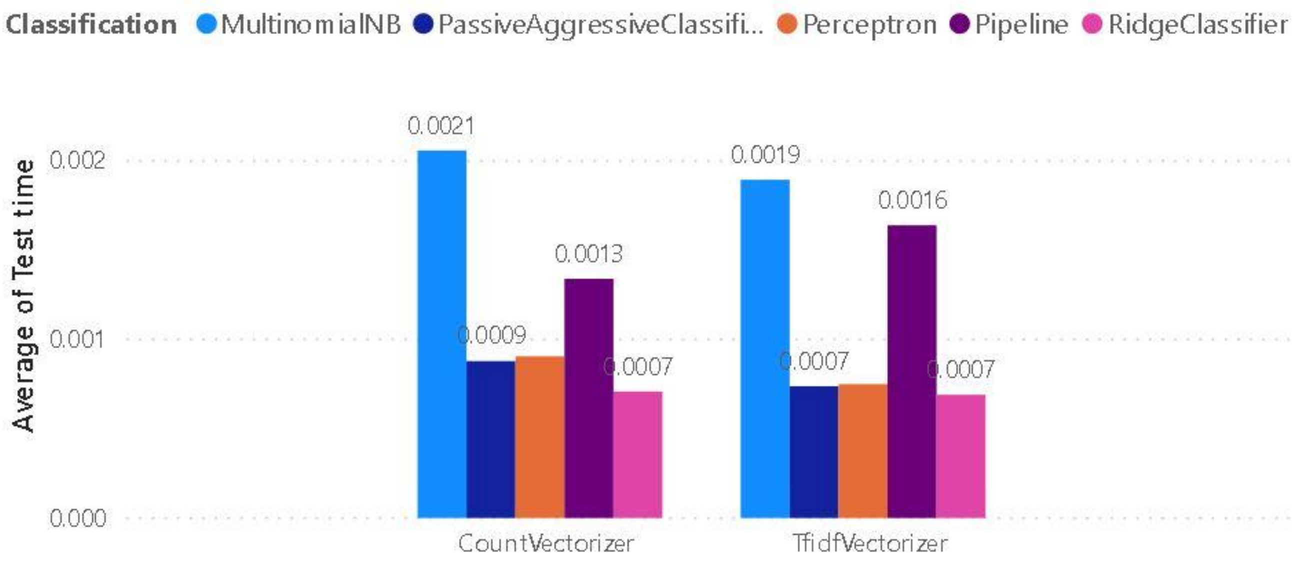

Figure 9: Testing time per classification method

Figure 10 shows ROC curve and AUC value for the PassiveAggressive method case:

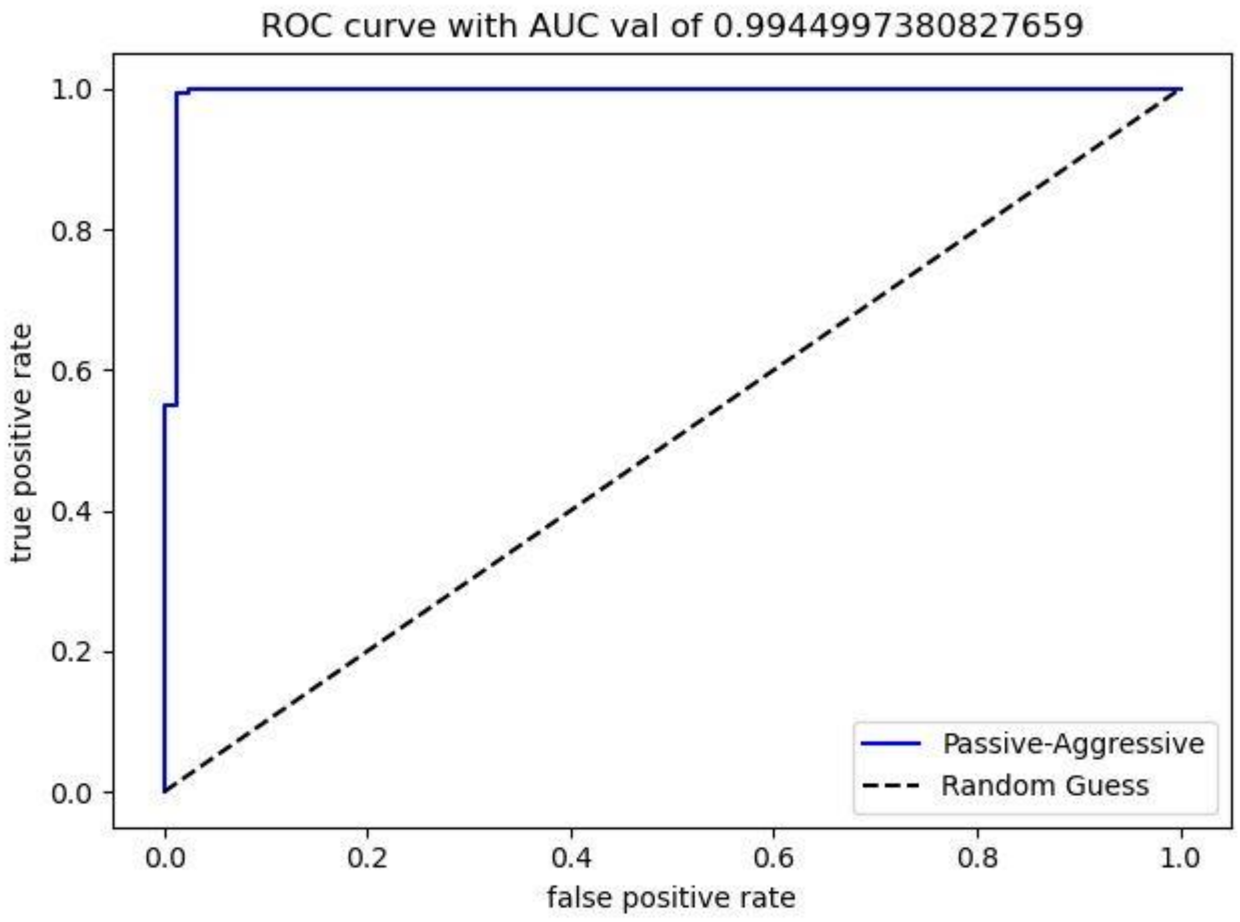

Figure 10: ROC curve example

Figure 11 illustrates the relation between classification accuracy and percentage of malicious code in the training set:

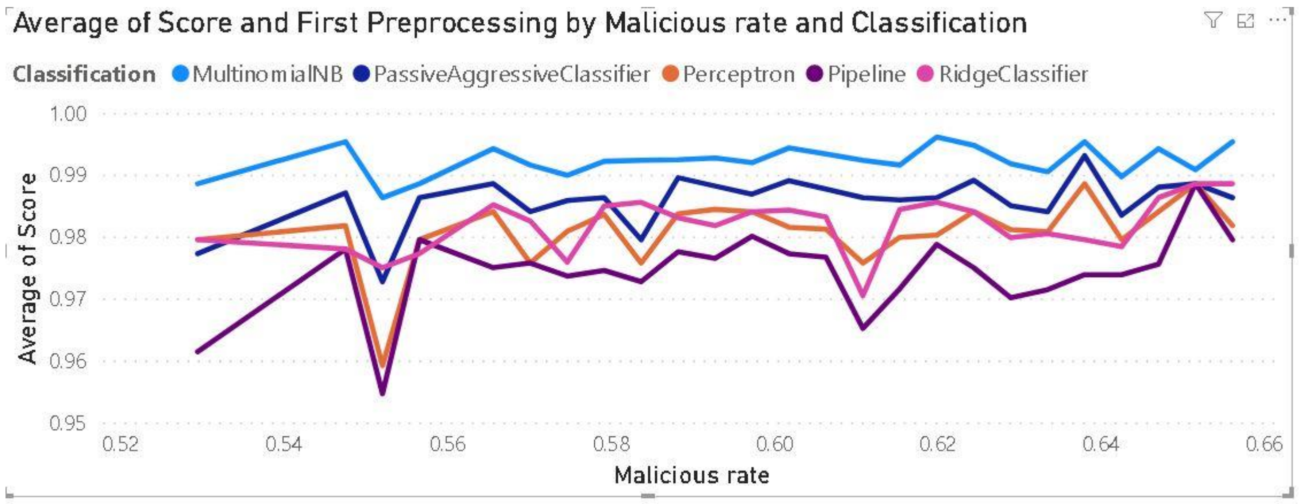

Figure 11: Relation between malicious rate and accuracy

From Figure 11 we can learn that MultinomialNB classification method produces the best results, but taking into account the training and testing times too, we choose PassiveAggressive method with TfidfVEctorized based features preprocessing as the solution that shows the best combination of precision (99%), training (0.06) and testing (0.0007) times.

## 10. Comparison with the related works

### 10.1. Review of related works

- Article [1] collects API traces and uses them as input to create signatures. The work creates several families (classes) of malware and uses created signatures to find out degree of membership of a given sample (by generated signature) to each malware family. On base of the results (calculated degree of membership) a database for malware's behaviors classes (and its membership degrees to each classes) was constructed.

  At signatures generation phase the work uses the term of "Critical API calls" defined as:

  *"Critical API calls contain all API calls that can lead to security infraction, changes to the operating system's behavior or API calls used for communication (modification of the system registry value, Input/Output, API functions for network resources access, etc.)."*

  The signature of program behavior based on API call tracing can be presented as a set of two components: the call frequency and the interaction of the critical API calls. An analysis of the first component allows determining the distribution of the critical API calls by groups concerning their malicious activity and displays the quantitative component of the signature. The second component of the signature implies the mapping nature of the interaction of malware's critical API calls into the vector space and describes their interactions.

The work uses a preset of 26 API categories (Hooking, file and directory, etc.) to map critical API calls at features extraction phase.
Extracted features were used as input to the MultonomialNB classifier.

In comparison with this work we can note the following:

- The work [1] maps collected API into one of 26 hardcoded categories which limits API variety appearing in logs to subset of APIs that are enlisted in those 26 categories only
- The critical API call term allows to reduce part of non-important API calls from the trace log, still , in our opinion, traces based on ETW events are more reliable in this case, because non-important API are not expected to produce meaningful ETW events

- Article [2] uses the Detours library to collect API call traces. Like the work [1], collected API calls being categorized to one out of 26 API categories. The work uses both Critical API calls (like in [1]) and *Longest Common Subsequence* (LCS) is defined as follows:
  In the formula, $Xi$ and $Yi$ represent the $i$th character of sequences $X$ and $Y$, respectively. For example, the LCSs of ABCD and ACB are AB and AC:

$$\mathrm{LCS}(X_i,Y_j)=\begin{cases} 0 \; if \; i=0 \; or \; j=0, \\ LCS(X_{i-1},Y_{j-1}) + common\ character \; if \; x_i=y_j, \\ longest(LCS(X_i,Y_{j-1})LCS(X_{i-1},Y_j)) if \; x_i \neq y_j \end{cases}$$

  Since the LCS shows the longest malware API call sequence pattern, it can be treated as a malware's signature.
  Authors of the work stated that extraction of LCS is an NP-hard problem, authors also used DNA sequence alignment algorithms to compare extracted signatures with database of known threats, which also was resource and time-consuming task.
  As a result, we can state that, in comparison with our project, the work has same limitation as article [1] and also have high cost of implementation due to the complex algorithms being used.

- Article [3] also uses API call traces as input. Authors of the work extract features from collected traces in the form of patterns:
  **Definition 1 (Pattern Instance)**:
  Given a pattern $P\langle e_1, e_2, \ldots, e_n \rangle$, a consecutive series of events SB ($sb_1, sb_2,\ldots, sb_n$) in an API call sequence S in API call database (APIDB) is an instance of P if it is of the following Quantified regular expression (QRE):
  $e_1$; [-$e_1$, …, $e_n$]*; …; [-$e_1$, …, $e_n$]*; $e_n$

  For each malware, the authors extract a feature vector from its behavior. Assume MB and Fv are consecutive representatives of a malware behavior and feature vector for each malware. Fs is a feature set that is extracted from the previous part.
  Then:

$$\mathrm{Fv}_i=\begin{Bmatrix} total \; Fsi \; in \; Mb, if \; Mb \; contains \; Fsi \\ 0, otherwise \end{Bmatrix}$$

  Additionally, the authors select features that are more discriminative between malware and benign samples with help of statistical measure – the Fisher score that indicates the discriminative power of the features.

Resulting features vectors were used with classifier method of RandomForest to get classification results.
From our point of view, this is interesting work that invested lot of efforts to raise the quality of the extracted features.
Since the work based on API call traces, our comparison with the work will point to the same conclusion as in case of article [1] and [2].

- Article [4] proposes to collect API traces and treat collected data as "bag of words", sending it to either tf (CountVectorized) or tfidf (TfidfVectorized) features extractor. Resulting features were used with several classifiers in order to find a classifier that produces the best accuracy result.
  Our project is quite similar to this article: it uses the same approach to extract features and finds the best classifier method. The major difference comes in the type of the input: we use ETW data instead of API call traces.

## 10.2. Results comparison

Table 1 summarizes the comparison between the results described in the referenced works and the results received from our project:

| Work | Accuracy | Our results |
|---|---|---|
| **Dynamic Signature-based Malware Detection Technique Based on API Call Tracing [1]** | 96.56% | 99% |
| **A Novel Approach to Detect Malware Based on API Call Sequence Analysis [2]** | 100% | 99% |
| **Malware detection by behavioral sequential patterns [3]** | 98.4% | 99% |
| **NtMalDetect: A Machine Learning Approach to Malware Detection Using Native API System Calls [4]** | 96% | 99% |

Table 1: Comparison table with referenced works

Note: this comparison should be taken carefully because each of the articles uses different data sets for testing.

## 11. Summary and conclusions

Based on the achieved results we can conclude that usage of Windows Event logs as source of features for malware classification looks like a promising path for further investigation.

Specifically, in future work next items should be addressed:

- Use up to date set of malware samples
- Use up to date Windows OS version
- Increase the number of benign and malware samples to 10000 at least

Additional paths of research that worth to note:

- Verify that amount of collected traces per process can be reduced significantly (currently it's on the average equal to 6558) to ~100 without loss of accuracy
- Adaptation of benign/malware features collection tool chain to work with next cases
    - Scripts
    - Execution scenarios that spawn across several processes (example: malware dropper download payload and executes it or scenario of process hollowing)

Known limitation of the suggested method:

- The method relies on OS specific (Windows) built in tracing facility, making the method applicable for Windows OS only

## 12. References

1. Oleg Savenko, Andrii Nicheporuk , Ivan Hurman and Sergii Lysenko, "Dynamic Signature-based Malware Detection Technique Based on API Call Tracing" Proceeding ICTERI Workshops 2019: 633-643

2. Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. "A Novel Approach to Detect Malware Based on API Call Sequence Analysis", International Journal of Distributed Sensor Networks Volume 2015, Article ID 659101

3. Ashkan Sami, Mansour Ahmadi "Malware detection by behavioural sequential patterns." Computer Fraud & Security · August 2013

4.Chan Woo Kim. "NtMalDetect: A Machine Learning Approach to Malware Detection Using Native API System Calls". arXiv:1802.05412 (2018)

5. Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. "Multinomial Naive Bayes for Text Categorization Revisited" G.I. Webb and Xinghuo Yu (Eds.): AI 2004, LNAI 3339, pp. 488–499, 2004.

6. Thorsten Joachim. "Text Categorization with Support Vector Machines: Learning with Many Relevant Features" Informatik LS8, Baroper Str. 301 44221 Dortmund, Germany, 1998

7. Vapnik V. The Nature of Statistical Learning Theory. Springer,New York, 1995

8. Vapnik V. Estimation of Dependencies Based on Empirical Data.Springer Series in Statistics. Springer-Verlag, 1982

## 13. Appendix A – list of events

| EventName | EventId | EventOccurance | EventPerProcessOccurance |
|---|---|---|---|
| image_load_0_3 | 2 | 19196 | 878 |
| microsoftwindowswin32k_stop_85_0 | 63 | 619 | 619 |
| microsoftjscript_methodunload_10_0 | 172 | 58 | 2 |
| microsoftwindowswin32k_info_49_0 | 39 | 196 | 61 |
| fileio_delete_0_3 | 73 | 164 | 56 |
| fileio_mapfile_0_2 | 1 | 119985 | 878 |
| registry_create_0_2 | 28 | 156055 | 259 |
| fileio_fscontrol_0_3 | 8 | 1745 | 137 |
| fileio_cleanup_0_3 | 7 | 234473 | 873 |
| fileio_close_0_3 | 10 | 215236 | 873 |
| fileio_read_0_3 | 6 | 74996 | 832 |
| microsoftwindowswin32k_info_53_0 | 40 | 79578 | 59 |
| fileio_create_0_3 | 3 | 263644 | 873 |
| 45d8cccd539f4b72a8b75c683142609a_38_0_2 | 13 | 2741 | 868 |
| fileio_operationend_0_3 | 4 | 1144230 | 878 |
| microsoftwindowstcpip_info_1229_0 | 136 | 636 | 2 |
| image_unload_0_3 | 64 | 18914 | 871 |
| microsoftwindowstcpip_info_1324_0 | 98 | 3 | 1 |
| registry_open_0_2 | 18 | 641461 | 868 |
| fileio_queryinfo_0_3 | 5 | 288141 | 878 |
| microsoftjscript_stop_130_0 | 196 | 24 | 2 |
| fileio_setinfo_0_3 | 9 | 2027 | 278 |
| microsoftwindowswin32k_start_60_0 | 50 | 13923 | 60 |
| registry_enumeratevaluekey_0_2 | 26 | 16257 | 573 |
| pagefault_virtualalloc_0_2 | 11 | 81451 | 878 |
| microsoftwindowswin32k_stop_70_0 | 51 | 13923 | 60 |
| microsoftwindowspowershell_stop_40962_1 | 156 | 1 | 1 |
| pagefault_virtualfree_0_2 | 12 | 59754 | 878 |
| microsoftwindowswin32k_info_181_0 | 46 | 79714 | 57 |
| microsoftwindowswin32k_start_86_0 | 25 | 879 | 625 |
| alpc_alpcsendmessage_0_2 | 14 | 101686 | 878 |
| thread_end_0_3 | 57 | 1998 | 870 |
| alpc_alpcunwait_0_2 | 15 | 67208 | 878 |

| | | | |
|---|---|---|---|
| registry_queryvalue_0_2 | 19 | 400405 | 868 |
| microsoftjscript_start_78_0 | 197 | 24 | 2 |
| microsoftwindowstcpip_info_1169_0 | 84 | 32 | 5 |
| alpc_alpcreceivemessage_0_2 | 16 | 70606 | 878 |
| 45d8cccd539f4b72a8b75c683142609a_39_0_2 | 17 | 2453 | 868 |
| registry_close_0_2 | 20 | 581996 | 868 |
| microsoftwindowswin32k_stop_91_0 | 151 | 62 | 3 |
| microsoftwindowstcpip_info_1223_0 | 135 | 1 | 1 |
| microsoftwindowskernelmemory_info_8_0 | 21 | 20248 | 755 |
| fileio_unmapfile_0_2 | 22 | 123365 | 878 |
| microsoftwindowswin32k_info_2_0 | 42 | 117 | 58 |
| microsoftwindowswin32k_start_84_0 | 23 | 625 | 625 |
| microsoftwindowswin32k_info_169_0 | 55 | 12058 | 157 |
| microsoftwindowswin32k_info_59_0 | 35 | 8112 | 144 |
| microsoftwindowswin32k_info_187_0 | 24 | 2129 | 625 |
| fileio_dirnotify_0_3 | 83 | 5 | 3 |
| registry_kcbcreate_0_2 | 27 | 15831 | 369 |
| microsoftwindowswin32k_info_203_0 | 31 | 15152 | 112 |
| microsoftwindowswin32k_info_1_0 | 29 | 12103 | 112 |
| alpc_alpcwaitforreply_0_2 | 30 | 64099 | 368 |
| microsoftwindowswin32k_info_52_0 | 32 | 9518 | 149 |
| registry_query_0_2 | 33 | 392547 | 304 |
| microsoftwindowswin32k_info_28_0 | 61 | 122 | 61 |
| registry_enumeratekey_0_2 | 34 | 28398 | 198 |
| microsoftwindowswin32k_info_65_0 | 36 | 11971 | 122 |
| microsoftwindowswin32k_info_27_0 | 37 | 122 | 61 |
| microsoftwindowswin32k_info_29_0 | 38 | 196 | 61 |
| microsoftjscript_sourceunload_42_0 | 173 | 30 | 2 |
| microsoftwindowswin32k_info_26_0 | 41 | 117 | 58 |
| microsoftjscript_stop_66_0 | 170 | 250 | 2 |
| microsoftwindowswin32k_info_189_0 | 43 | 929 | 87 |
| microsoftwindowswin32k_info_31_0 | 44 | 74 | 61 |
| microsoftwindowswin32k_apprenderingupdate_41_0 | 45 | 70630 | 54 |
| fileio_direnum_0_3 | 69 | 41116 | 286 |
| microsoftwindowswin32k_apprenderingtightupdate_42_0 | 47 | 9067 | 52 |
| microsoftwindowswin32k_stop_77_0 | 48 | 19378 | 178 |
| microsoftjscript_stop_132_0 | 190 | 18 | 2 |
| microsoftwindowswin32k_start_62_0 | 49 | 13801 | 58 |
| microsoftwindowswin32k_info_50_0 | 59 | 196 | 61 |
| microsoftwindowswin32k_stop_83_0 | 52 | 13801 | 58 |
| microsoftwindowswin32k_info_61_0 | 53 | 8647 | 125 |
| microsoftwindowswin32k_start_76_0 | 54 | 19339 | 168 |
| microsoftwindowswin32k_info_168_0 | 56 | 13275 | 64 |

| | | | |
|---|---|---|---|
| microsoftwindowstcpip_info_1039_0 | 145 | 1 | 1 |
| microsoftwindowswin32k_info_45_0 | 58 | 870 | 619 |
| microsoftwindowswin32k_info_30_0 | 60 | 122 | 61 |
| microsoftwindowswin32k_stop_87_0 | 62 | 870 | 619 |
| process_end_0_4 | 65 | 869 | 869 |
| 45d8cccd539f4b72a8b75c683142609a_41_0_2 | 66 | 2715 | 861 |
| microsoftjscript_stop_91_0 | 176 | 18 | 2 |
| registry_deletevalue_0_2 | 67 | 243 | 61 |
| microsoftjscript_allocatefunction_110_0 | 171 | 240 | 2 |
| microsoftwindowstcpip_info_1194_0 | 125 | 1 | 1 |
| registry_setvalue_0_2 | 68 | 2137 | 200 |
| registry_querysecurity_0_2 | 70 | 386 | 19 |
| registry_kcbdelete_0_2 | 71 | 1113 | 4 |
| microsoftjscript_allocateobject_108_0 | 205 | 2 | 2 |
| registry_delete_0_2 | 100 | 3 | 3 |
| fileio_write_0_3 | 72 | 25199 | 296 |
| fileio_dletepath_0_3 | 74 | 164 | 56 |
| fileio_rename_0_3 | 75 | 87 | 8 |
| fileio_renamepath_0_3 | 76 | 87 | 8 |
| process_terminate_0_2 | 77 | 764 | 753 |
| thread_start_0_3 | 78 | 1272 | 342 |
| microsoftjscript_info_103_0 | 203 | 18 | 2 |
| microsoftwindowstcpip_info_1331_0 | 144 | 637 | 2 |
| microsoftwindowswin32k_info_64_0 | 79 | 8635 | 51 |
| microsoftwindowswin32k_start_92_0 | 80 | 8592 | 33 |
| microsoftjscript_stop_75_0 | 192 | 24 | 2 |
| microsoftwindowswin32k_stop_93_0 | 81 | 8592 | 33 |
| microsoftwindowskernelmemory_info_1_1 | 82 | 69 | 16 |
| microsoftwindowstcpip_info_1170_0 | 85 | 28 | 5 |
| registry_setsecurity_0_2 | 86 | 627 | 6 |
| microsoftwindowswin32k_start_205_0 | 87 | 1 | 1 |
| microsoftwindowswin32k_info_88_0 | 88 | 164 | 12 |
| microsoftwindowswin32k_info_63_0 | 89 | 273 | 9 |
| microsoftwindowswin32k_info_33_0 | 90 | 6 | 2 |
| microsoftwindowswin32k_info_225_0 | 91 | 147 | 5 |
| microsoftwindowspowershell_start_40961_1 | 153 | 1 | 1 |
| microsoftwindowswin32k_info_204_0 | 92 | 3 | 3 |
| microsoftwindowswin32k_info_6_0 | 93 | 1 | 1 |
| microsoftwindowswin32k_oldtonewrendering_40_0 | 94 | 8 | 3 |
| microsoftwindowswin32k_info_151_0 | 95 | 7 | 2 |
| pagefault_memresetinfo_0_2 | 96 | 530 | 10 |
| process_start_0_4 | 97 | 314 | 189 |
| systemconfig_codeintegrity_0_2 | 116 | 1 | 1 |

| | | | |
|---|---|---|---|
| fileio_flush_0_3 | 99 | 73 | 14 |
| microsoftwindowstcpip_info_1127_0 | 101 | 16 | 3 |
| thread_dcend_0_3 | 102 | 18 | 6 |
| microsoftwindowswin32k_info_35_0 | 103 | 4 | 4 |
| microsoftwindowswin32k_info_38_0 | 104 | 9 | 7 |
| systemconfig_cpu_0_3 | 105 | 1 | 1 |
| systemconfig_logdisk_0_2 | 117 | 2 | 1 |
| systemconfig_dpi_0_2 | 106 | 1 | 1 |
| systemconfig_nic_0_2 | 107 | 3 | 1 |
| systemconfig_phydisk_0_2 | 108 | 2 | 1 |
| systemconfig_idechannel_0_2 | 109 | 3 | 1 |
| systemconfig_opticaldisk_0_2 | 110 | 1 | 1 |
| systemconfig_pnp_0_5 | 111 | 90 | 1 |
| systemconfig_irq_0_3 | 112 | 13 | 1 |
| systemconfig_services_0_3 | 113 | 171 | 1 |
| image_dcend_0_3 | 118 | 139 | 1 |
| systemconfig_power_0_2 | 114 | 1 | 1 |
| systemconfig_platform_0_2 | 115 | 1 | 1 |
| registry_flush_0_2 | 119 | 17 | 2 |
| microsoftwindowswin32k_info_3_0 | 120 | 1 | 1 |
| microsoftwindowstcpip_info_1013_0 | 121 | 167 | 137 |
| microsoftwindowstcpip_info_1001_0 | 122 | 167 | 137 |
| microsoftwindowstcpip_info_1191_0 | 123 | 33 | 14 |
| microsoftwindowstcpip_info_1008_0 | 124 | 33 | 14 |
| microsoftwindowstcpip_info_1123_0 | 126 | 1 | 1 |
| microsoftwindowstcpip_info_1009_0 | 127 | 167 | 137 |
| microsoftwindowstcpip_info_1051_0 | 128 | 27 | 9 |
| microsoftwindowstcpip_info_1192_0 | 129 | 1 | 1 |
| microsoftjscript_stop_87_0 | 202 | 14 | 2 |
| microsoftwindowstcpip_info_1002_0 | 130 | 1 | 1 |
| microsoftwindowstcpip_info_1003_0 | 131 | 12 | 9 |
| microsoftwindowstcpip_info_1004_0 | 132 | 1 | 1 |
| microsoftwindowstcpip_info_1031_0 | 133 | 1 | 1 |
| microsoftwindowswin32k_info_67_0 | 152 | 1 | 1 |
| microsoftwindowstcpip_info_1332_0 | 134 | 2 | 1 |
| microsoftwindowstcpip_info_1074_0 | 137 | 1286 | 2 |
| microsoftwindowswin32k_start_89_0 | 150 | 62 | 3 |
| microsoftwindowstcpip_info_1033_0 | 138 | 1 | 1 |
| microsoftwindowstcpip_info_1105_0 | 139 | 3 | 1 |
| microsoftwindowstcpip_info_1104_0 | 140 | 1 | 1 |
| microsoftwindowstcpip_info_1193_0 | 141 | 45 | 14 |
| microsoftwindowstcpip_info_1159_0 | 142 | 637 | 2 |
| microsoftwindowstcpip_info_1157_0 | 143 | 635 | 2 |

| | | | |
|---|---|---|---|
| microsoftwindowstcpip_info_1044_0 | 146 | 13 | 9 |
| microsoftwindowstcpip_info_1184_0 | 147 | 1 | 1 |
| microsoftwindowstcpip_info_1040_0 | 148 | 1 | 1 |
| microsoftwindowstcpip_info_1038_0 | 149 | 2 | 1 |
| microsoftwindowspowershell_tobeusedwhenoperationisjust executingamethod_7939_1 | 154 | 6 | 1 |
| microsoftwindowspowershell_tobeusedwhenoperationisjust executingamethod_7938_1 | 155 | 2 | 1 |
| microsoftwindowspowershell_tobeusedwhenoperationisjust executingamethod_7937_1 | 157 | 14 | 1 |
| microsoftwindowstcpip_info_1100_0 | 158 | 2 | 1 |
| microsoftjscript_usedpagesize_105_0 | 164 | 1408 | 2 |
| microsoftwindowstcpip_info_1156_0 | 159 | 1 | 1 |
| microsoftwindowstcpip_info_1158_0 | 160 | 1 | 1 |
| registry_querymultiplevalue_0_2 | 161 | 514 | 257 |
| microsoftwindowstcpip_info_1021_0 | 162 | 11 | 8 |
| microsoftwindowstcpip_info_1034_0 | 163 | 11 | 8 |
| microsoftjscript_scriptcontextload_11_0 | 165 | 10 | 2 |
| microsoftjscript_sourceload_41_0 | 166 | 30 | 2 |
| microsoftjscript_start_65_0 | 167 | 280 | 2 |
| microsoftjscript_info_67_0 | 168 | 270 | 2 |
| microsoftjscript_methodload_9_0 | 169 | 58 | 2 |
| microsoftjscript_scriptcontextunload_12_0 | 174 | 10 | 2 |
| microsoftjscript_start_90_0 | 175 | 18 | 2 |
| microsoftjscript_start_68_0 | 177 | 24 | 2 |
| microsoftjscript_stop_69_0 | 178 | 24 | 2 |
| microsoftjscript_start_70_0 | 179 | 24 | 2 |
| microsoftjscript_stop_71_0 | 180 | 24 | 2 |
| microsoftjscript_start_72_0 | 181 | 24 | 2 |
| microsoftjscript_stop_73_0 | 182 | 24 | 2 |
| microsoftjscript_start_88_0 | 183 | 18 | 2 |
| microsoftjscript_start_153_0 | 184 | 18 | 2 |
| microsoftjscript_stop_89_0 | 185 | 18 | 2 |
| microsoftjscript_start_143_0 | 186 | 18 | 2 |
| microsoftjscript_stop_144_0 | 187 | 18 | 2 |
| microsoftjscript_stop_154_0 | 188 | 18 | 2 |
| microsoftjscript_start_131_0 | 189 | 18 | 2 |
| microsoftjscript_start_74_0 | 191 | 24 | 2 |
| microsoftjscript_start_139_0 | 193 | 24 | 2 |
| microsoftjscript_stop_140_0 | 194 | 24 | 2 |
| microsoftjscript_start_129_0 | 195 | 24 | 2 |
| microsoftjscript_freememory_106_0 | 198 | 66 | 2 |
| microsoftjscript_freememoryblock_107_0 | 199 | 116 | 2 |

| | | | |
|---|---|---|---|
| microsoftjscript_stop_79_0 | 200 | 24 | 2 |
| microsoftjscript_start_86_0 | 201 | 14 | 2 |
| microsoftjscript_start_100_0 | 204 | 2 | 2 |
| microsoftjscript_stop_104_0 | 206 | 2 | 2 |

## 14. Appendix B – executed programs

Following table includes examples of 100 benign and 100 malicious programs that were exeuted:

| Executable name | Type |
|---|---|
| accessenum | Benign |
| aitagent | Benign |
| aitstatic | Benign |
| alg | Benign |
| append | Benign |
| appidcertstorecheck | Benign |
| appidpolicyconverter | Benign |
| ARP | Benign |
| at | Benign |
| AtBroker | Benign |
| attrib | Benign |
| audiodg | Benign |
| auditpol | Benign |
| AuthHost | Benign |
| autoruns | Benign |
| autorunsc | Benign |
| AutoWorkplace | Benign |
| AxInstUI | Benign |
| baaupdate | Benign |
| backgroundTaskHost | Benign |
| BackgroundTransferHost | Benign |
| bcdboot | Benign |
| bcdedit | Benign |
| bdechangepin | Benign |
| BdeHdCfg | Benign |
| BdeUISrv | Benign |
| bdeunlock | Benign |
| BitLockerDeviceEncryption | Benign |
| BitLockerWizard | Benign |
| BitLockerWizardElev | Benign |
| bitsadmin | Benign |
| bootcfg | Benign |
| bootim | Benign |

| | |
|---|---|
| bootsect | Benign |
| bridgeunattend | Benign |
| bthudtask | Benign |
| BulkOperationHost | Benign |
| ByteCodeGenerator | Benign |
| cacls | Benign |
| calc | Benign |
| CameraSettingsUIHost | Benign |
| CertEnrollCtrl | Benign |
| certreq | Benign |
| certutil | Benign |
| change | Benign |
| changepk | Benign |
| charmap | Benign |
| CheckNetIsolation | Benign |
| chglogon | Benign |
| chgport | Benign |
| chgusr | Benign |
| chkdsk | Benign |
| chkntfs | Benign |
| choice | Benign |
| chrome | Benign |
| cipher | Benign |
| cleanmgr | Benign |
| cliconfg | Benign |
| clip | Benign |
| clockres | Benign |
| CloudNotifications | Benign |
| CloudStorageWizard | Benign |
| cmd | Benign |
| cmdkey | Benign |
| cmdl32 | Benign |
| cmmon32 | Benign |
| cmstp | Benign |
| cofire | Benign |
| colorcpl | Benign |
| comp | Benign |
| compact | Benign |
| CompMgmtLauncher | Benign |
| ComputerDefaults | Benign |
| conhost | Benign |
| consent | Benign |
| control | Benign |
| convert | Benign |

| | |
|---|---|
| coreinfo | Benign |
| CredentialUIBroker | Benign |
| credwiz | Benign |
| cscript | Benign |
| ctfmon | Benign |
| cttune | Benign |
| cttunesvr | Benign |
| dasHost | Benign |
| dccw | Benign |
| dcomcnfg | Benign |
| ddodiag | Benign |
| debug | Benign |
| Defrag | Benign |
| DeviceEject | Benign |
| DevicePairingWizard | Benign |
| DeviceProperties | Benign |
| DFDWiz | Benign |
| dfp | Benign |
| dfrgui | Benign |
| dialer | Benign |
| diskext | Benign |
| diskmon | Benign |
| diskpart | Benign |
| | |
| 1002 | Malware |
| 1003 | Malware |
| 131 | Malware |
| 21 | Malware |
| 3_4 | Malware |
| 5a765351046fea1490d20f25 | Malware |
| 798_abroad | Malware |
| aapt | Malware |
| Backdoor-Win32-Poison-a | Malware |
| Backdoor-Win32-Poison-ac | Malware |
| Backdoor-Win32-Poison-adi | Malware |
| Backdoor-Win32-Poison-aec | Malware |
| Backdoor-Win32-Poison-ahm | Malware |
| Backdoor-Win32-Poison-apv | Malware |
| Backdoor-Win32-Poison-atk | Malware |
| Backdoor-Win32-Poison-ats | Malware |
| Backdoor-Win32-Poison-att | Malware |
| Backdoor-Win32-Poison-atz | Malware |
| Backdoor-Win32-Poison-auc | Malware |
| Backdoor-Win32-Poison-avd | Malware |

| | |
|---|---|
| Backdoor-Win32-Poison-avh | Malware |
| Backdoor-Win32-Poison-ax | Malware |
| Backdoor-Win32-Poison-azx | Malware |
| Backdoor-Win32-Poison-bad | Malware |
| Backdoor-Win32-Poison-bcr | Malware |
| Backdoor-Win32-Poison-bd | Malware |
| Backdoor-Win32-Poison-bep | Malware |
| Backdoor-Win32-Poison-bet | Malware |
| Backdoor-Win32-Poison-bex | Malware |
| Backdoor-Win32-Poison-bey | Malware |
| Backdoor-Win32-Poison-bfa | Malware |
| Backdoor-Win32-Poison-bfk | Malware |
| Backdoor-Win32-Poison-bfm | Malware |
| Backdoor-Win32-Poison-bfn | Malware |
| Backdoor-Win32-Poison-bfp | Malware |
| Backdoor-Win32-Poison-bfq | Malware |
| Backdoor-Win32-Poison-bfw | Malware |
| Backdoor-Win32-Poison-bfz | Malware |
| Backdoor-Win32-Poison-bgv | Malware |
| Backdoor-Win32-Poison-bgw | Malware |
| Backdoor-Win32-Poison-bhf | Malware |
| Backdoor-Win32-Poison-bhg | Malware |
| Backdoor-Win32-Poison-bhm | Malware |
| Backdoor-Win32-Poison-bhs | Malware |
| Backdoor-Win32-Poison-bia | Malware |
| Backdoor-Win32-Poison-bib | Malware |
| Backdoor-Win32-Poison-big | Malware |
| Backdoor-Win32-Poison-bkd | Malware |
| Backdoor-Win32-Poison-bke | Malware |
| Backdoor-Win32-Poison-bkj | Malware |
| Backdoor-Win32-Poison-bkr | Malware |
| Backdoor-Win32-Poison-bky | Malware |
| Backdoor-Win32-Poison-blk | Malware |
| Backdoor-Win32-Poison-bls | Malware |
| Backdoor-Win32-Poison-blu | Malware |
| Backdoor-Win32-Poison-blz | Malware |
| Backdoor-Win32-Poison-bmc | Malware |
| Backdoor-Win32-Poison-bmd | Malware |
| Backdoor-Win32-Poison-bmf | Malware |
| Backdoor-Win32-Poison-bmh | Malware |
| Backdoor-Win32-Poison-bmv | Malware |
| Backdoor-Win32-Poison-bmw | Malware |
| Backdoor-Win32-Poison-bmx | Malware |
| Backdoor-Win32-Poison-bnb | Malware |

| | |
|---|---|
| Backdoor-Win32-Poison-bnc | Malware |
| Backdoor-Win32-Poison-bng | Malware |
| Backdoor-Win32-Poison-bnk | Malware |
| Backdoor-Win32-SdBot-02 | Malware |
| Backdoor-Win32-SdBot-04-a | Malware |
| Backdoor-Win32-SdBot-04-c | Malware |
| Backdoor-Win32-SdBot-04-d | Malware |
| Backdoor-Win32-SdBot-04-f | Malware |
| Backdoor-Win32-SdBot-04-g | Malware |
| Backdoor-Win32-SdBot-05-aa | Malware |
| Backdoor-Win32-SdBot-05-ar | Malware |
| Backdoor-Win32-SdBot-05-d | Malware |
| Backdoor-Win32-SdBot-05-e | Malware |
| Backdoor-Win32-SdBot-05-f | Malware |
| Backdoor-Win32-SdBot-05-g | Malware |
| Backdoor-Win32-SdBot-05-m | Malware |
| Backdoor-Win32-SdBot-05-n | Malware |
| Backdoor-Win32-SdBot-05-o | Malware |
| Backdoor-Win32-SdBot-05-p | Malware |
| Backdoor-Win32-SdBot-05-q | Malware |
| Backdoor-Win32-SdBot-05-s | Malware |
| Backdoor-Win32-SdBot-05-v | Malware |
| Backdoor-Win32-SdBot-05-w | Malware |
| Backdoor-Win32-SdBot-05-z | Malware |
| Backdoor-Win32-SdBot-12 | Malware |
| Backdoor-Win32-SdBot-a | Malware |
| Backdoor-Win32-SdBot-aa | Malware |
| Backdoor-Win32-SdBot-aac | Malware |
| Backdoor-Win32-SdBot-aaj | Malware |
| Backdoor-Win32-SdBot-aap | Malware |
| Backdoor-Win32-SdBot-aav | Malware |
| Backdoor-Win32-SdBot-aay | Malware |
| Backdoor-Win32-SdBot-abc | Malware |
| Backdoor-Win32-SdBot-abe | Malware |
| Backdoor-Win32-SdBot-abf | Malware |
| Backdoor-Win32-SdBot-abh | Malware |
| Backdoor-Win32-SdBot-abo | Malware |

## 15. Appendix C – classification algorithms

### 15.1. Ridge classification algorithm

The Ridge classifier first converts binary targets to {-1, 1} and then treats the problem as a regression task. The predicted class corresponds to the sign of the regressor's prediction. For multiclass classification, the problem is treated as multi-output regression, and the predicted class corresponds to the output with the highest value.

In mathematical notation if $\hat{y}$ is the predicted value, then:

$\hat{y}(w,x)=w_0+w_1x_1+...+w_px_p$

Across the module, we designate the vector $w=(w_1,...,w_p)$ as coefficients and w0 as intercept.

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients.The ridge coefficients minimize a penalized residual sum of squares:

$$\min_{w} ||Xw-y||_2^2+\alpha||w||_2^2$$

The complexity parameter $\alpha \geq 0$ controls the amount of shrinkage(Figure 12): the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

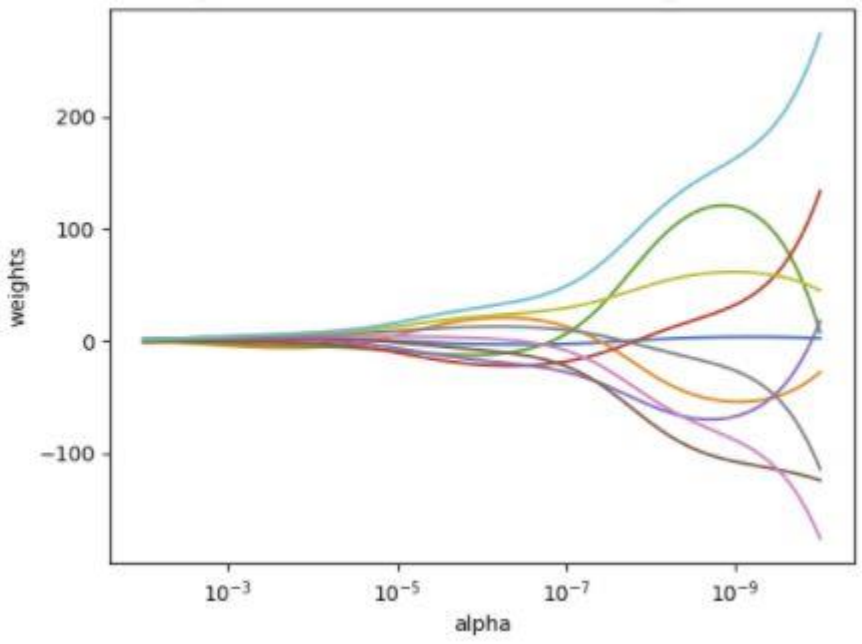

Figure 12: Ridge coefficients as function

### 15.2. Perceptron classification algorithm

The perceptron is an algorithm for learning a binary classifier called a threshold function: a function that maps its input **x** (a real-valued vector) to an output value **f(x)** (a single binary value):

$$f(x)=\begin{cases} 1 \text{ if } w\cdot\mathbf{x}+\mathbf{b}>\mathbf{0}, \\ 0 \text{ otherwise} \end{cases}$$

where **w** is a vector of real-valued weights, **w▪x** is the dot product $\sum_{i=1}^{m} w_i x_i$, where *m* is the number of inputs to the perceptron, and *b* is the *bias*. The bias shifts the decision boundary away from the origin and does not depend on any input value.

The value of f(x) (0 or 1) is used to classify **x** as either a positive or a negative instance, in the case of a binary classification problem. If *b* is negative, then the weighted combination of inputs must produce a positive value greater than $|b|$ in order to push the classifier neuron over the 0 threshold. Spatially, the bias alters the position (though not the orientation) of the decision boundary. The perceptron learning algorithm does not terminate if the learning set is not linearly separable. If the vectors are not linearly separable learning will never reach a point where all vectors are classified properly. The most famous example of the perceptron's inability to solve problems with linearly not separable vectors is the Boolean exclusive-or problem.

In the context of neural networks, a perceptron is an artificial neuron using the Heaviside step function as the activation function. The perceptron algorithm is also termed the **single-layer perceptron**, to distinguish it from a multilayer perceptron, which is a misnomer for a more complicated neural network. As a linear classifier, the single-layer perceptron is the simplest feedforward neural network.

## 15.3. Passive aggressive classification algorithm

Let's suppose that we have a dataset:

$$\begin{cases} X = \{\bar{x}_0, \bar{x}_1, \dots, \bar{x}_t, \dots\} \, where \; \bar{x}_i \in \mathbb{R}^n \\ Y = \{y_0, y_1, \dots, y_t, \dots\} \, where \; y_i \in \{-1, +1\} \end{cases}$$

The index t has been chosen to mark the temporal dimension. In this case, in fact, the samples can continue to arrive for an indefinite time. Of course, if they are drawn from same data generating distribution, the algorithm will keep learning (probably without large parameter modifications), but if they are drawn from a completely different distribution, the weights will slowly *forget* the previous one and learn the new distribution. For simplicity, we also assume we are working with a binary classification based on bipolar labels.

Given a weight vector w, the prediction is simply obtained as:

$$\tilde{y}_t = sign(\bar{w}^T \cdot \bar{x}_t)$$

All these algorithms are based on the Hinge loss function (the same one used by SVM):

$$L(\bar{\theta}) = max\big(0, 1 - y \cdot f(\bar{x}_t; \bar{\theta})\big)$$

The value of L is bounded between 0 (meaning perfect match) and K depending on f(x(t),θ) with K>0 (completely wrong prediction). A Passive-Aggressive algorithm works generically with this update rule:

$$\begin{cases} \bar{w}_{t+1} = argmin_{\bar{w}} \frac{1}{2}\|\bar{w} - \bar{w}_t\|^2 + C\xi^2 \\ L(\bar{w}; \bar{x}_t, y_t) \leq \xi \end{cases}$$

To understand this rule, let us assume the slack variable ξ=0 (and L constrained to be 0). If a sample x(t) is presented, the classifier uses the current weight vector to determine the sign. If the sign is correct, the loss function is 0 and the argmin is w(t). This means that the algorithm is **passive** when a correct classification occurs. Let's now assume that a misclassification occurred:

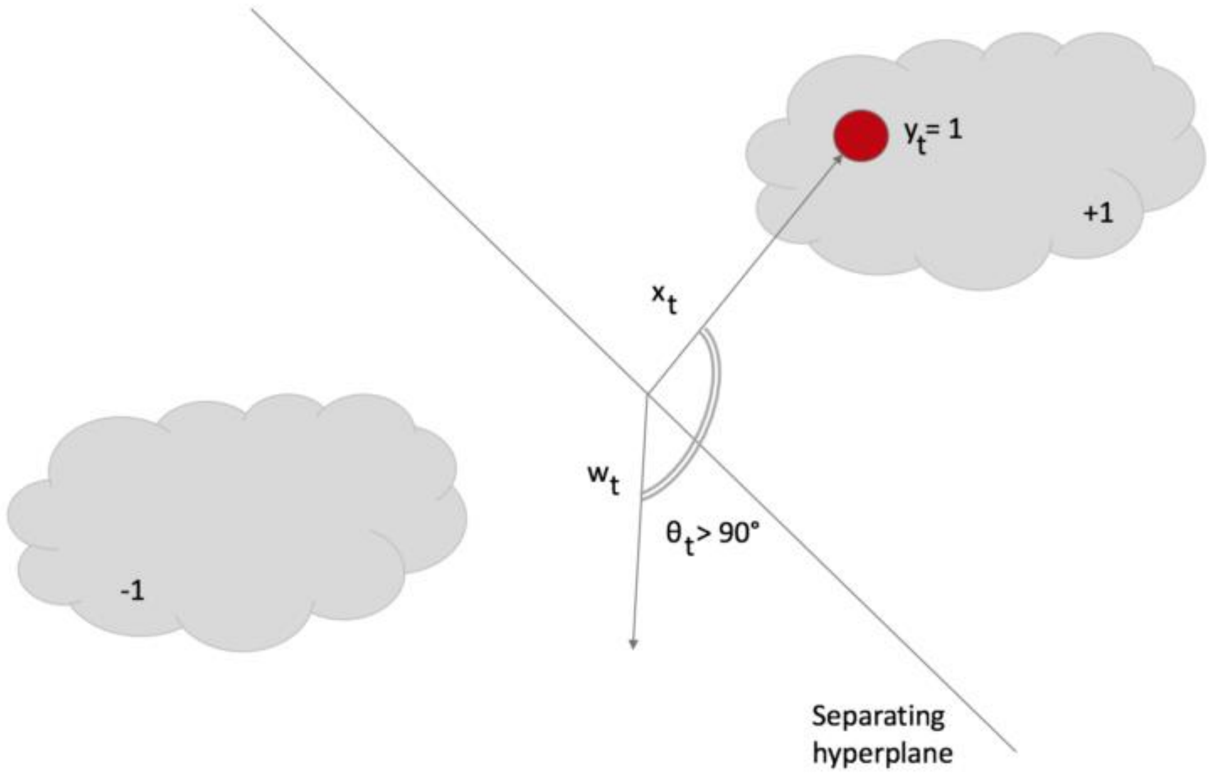

Figure 13

The angle θ > 90°, therefore, the dot product is negative, and the sample is classified as -1, however, its label is +1. In this case, the update rule becomes very **aggressive**, because it looks for a new w which must be as close as possible as the previous (otherwise the existing knowledge is immediately lost), but it must satisfy L=0 (in other words, the classification must be correct).

The introduction of the slack variable allows to have soft-margins (like in SVM) and a degree of tolerance controlled by the parameter C. In particular, the loss function must be L <= ξ, allowing a larger error. Higher C values yield stronger aggressiveness (with a consequent higher risk of destabilization in presence of noise), while lower values allow a better adaptation. In fact, this kind of algorithms, when working online, must cope with the presence of noisy samples (with wrong labels). A good robustness is necessary, otherwise, too rapid changes produce consequent higher misclassification rates.

After solving both update conditions, we get the closed-form update rule:

$$\bar{w}_{t+1} = \bar{w}_t + \frac{\max\left(0,1 - y_t(\bar{w}^T \cdot \bar{x}_t)\right)}{\|x_t\|^2 + \frac{1}{2C}} y_t \bar{x}_t$$

This rule confirms our expectations: the weight vector is updated with a factor whose sign is determined by y(t) and whose magnitude is proportional to the error. Note that if there is no misclassification the nominator becomes 0, so w(t+1) = w(t), while, in case of misclassification, w will rotate towards x(t) and stops with a loss L <= ξ. In the next figure, the effect has been marked to show the rotation, however, it is normally as smallest as possible:

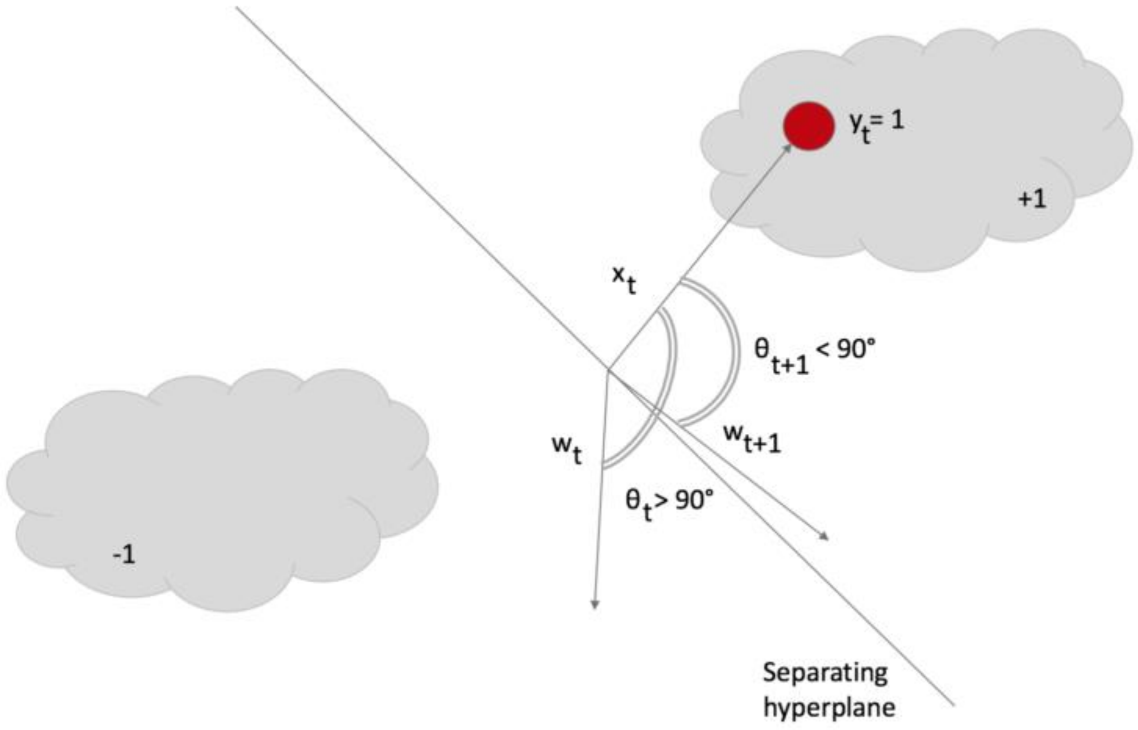

Figure 14

After the rotation, θ < 90° and the dot product becomes negative, so the sample is correctly classified as +1.

## 15.4. Multinomial Naïve Bayes classification algorithm

Based on article [5]:

Let the set of classes be denoted by C. Let N be the size of our vocabulary. Then MNB assigns a test document $t_i$ to the class that has the highest probability $\Pr(c|t_i)$, which, using Bayes' rule, is given by:

$$\Pr(c|t_i) = \frac{\Pr(c)\Pr(t_i|c)}{\Pr(t_i)}, \ c \in C \qquad (1)$$

The class prior Pr(c) can be estimated by dividing the number of documents belonging to class c by the total number of documents. $\Pr(t_i|c)$ is the probability of obtaining a document like $t_i$ in class c and is calculated as:

$$\Pr(t_i|c)=\left(\sum_n f_{ni}\right)!\prod_n \frac{\Pr(w_n|c)^{f_{ni}}}{f_{ni}!}, \quad (2)$$

where $f_{ni}$ is the count of word n in our test document $t_i$ and $\Pr(w_n|c)$ the probability of word n given class c. The latter probability is estimated from the training documents as:

$$\widehat{Pr}(w_n|c) = \frac{1+F_{nc}}{N+\sum_{x=1}^{N} F_{xc}}, \quad (3)$$

where $F_{xc}$ is the count of word x in all the training documents belonging to class c, and the Laplace estimator is used to prime each word's count with one to avoid the zero-frequency problem. The normalization factor $\Pr(t_i)$ in Equation 1 can be computed using

$$\Pr(t_i)=\sum_{k=1}^{|C|} \Pr(k)\Pr(t_i|k). \quad (4)$$

Note that that the computationally expensive terms $\left(\sum_n f_{ni}\right)!$ and $\prod_n f_{ni}!$ in Equation 2 can be deleted without any change in the results, because neither depends on the class c, and Equation 2 can be written as:

$$\Pr(t_i|c) = \alpha \prod_n \Pr(w_n|c)^{f_{ni}}, \quad (5)$$

where α is a constant that drops out because of the normalization step.

## 15.5. Linear SVM classification algorithm

Based on article [6]: Support vector machines (SVM) are based on the Structural Risk Minimization principle [7] from computational learning theory. The idea of structural risk minimization is to find a hypothesis h for which we can guarantee the lowest true error. The true error of h is the probability that h will make an error on an unseen and randomly selected test example. The following upper bound connects the true error of a hypothesis h with the error of h on the training set and the complexity of h [7].

$$P(error(h)) \le train\_error(h)+2\sqrt{\frac{d\left(ln\frac{2n}{d}+1\right)-ln\frac{n}{4}}{n}} \quad (6)$$

The bound holds with probability at least 1-n. n denotes the number of training examples and d is the VC-Dimension (VCdim) [7], which is a property of the hypothesis space and indicates its expressiveness. Equation (6) reflects the well-known trade-off between the complexity of the hypothesis space and the training error. A simple hypothesis space (small VCdim) will probably not contain good approximating functions and will lead to a high training (and true) error. On the other hand, a too rich hypothesis space (high VCdim) will lead to a small training error, but the second term in the right-hand side of (7) will be large. This situation is commonly called "overfitting". We can conclude that it is crucial to pick the hypothesis space with the "right" complexity.
In Structural Risk Minimization this is done by defining a structure of hypothesis spaces Hi, so that their respective VC-Dimension di increases.

$$H_1 \subset H_2 \subset H_3 \subset \ldots \subset H_i \subset \ldots \text{ and } \forall i: d_i \le d_{i+1} \quad (7)$$

The goal is to find the index $i^*$ for which (6) is minimum. How can we build this structure of increasing VCdim? In the following we will learn linear threshold functions of the type:

$$h(\tilde{d}) = \text{sign}\{\tilde{w} \cdot \tilde{d} + b\} = \begin{cases} +1, if\ \tilde{w} \cdot \tilde{d} + b > 0 \\ -1, else \end{cases} \quad (8)$$

Instead of building the structure based on the number of features using a feature selection strategy, Support vector machines uses a refined structure which acknowledges the fact that most features in text categorization are relevant.

**Lemma 1**. *[8] Consider hyperplanes $h(\tilde{d}) = \text{sign}\{\tilde{w} \cdot \tilde{d} + b\}$ as hypotheses. If all example vectors $\tilde{d}_i$ are contained in a ball of radius R and it is required that for all examples $\tilde{d}_i$*

$$|\tilde{w} \cdot \tilde{d}_i + b| \geq 1, \text{ with } ||\tilde{w}|| = A \quad (9)$$

*then this set of hyperplane has a VCdim d bounded by*

$$d \leq \min([R^2A^2], n)+1 \quad (10)$$

Please note that the VCdim of these hyperplanes does not necessarily depend on the number of features! Instead the VCdim depends on the Euclidean length $||\tilde{w}||$ of the weight vector $\tilde{w}$. This means that we can generalize well in high dimensional spaces, if our hypothesis has a small weight vector.
In their basic form support vector machines find the hyperplane that separates the training data, and which has the shortest weight vector. This hyperplane separates positive and negative training examples with maximum margin. Finding this hyperplane can be translated into the following optimization problem:

$$\text{Minimize: } ||\tilde{w}|| \quad (11)$$
$$\text{so that: } \forall i\text{: } y_i[\tilde{w} \cdot \tilde{d}_i + b] \geq 1 \quad (12)$$

$y_i$ equals +1 (-1), if document $d_i$ is in class + (-). The constraints (12) require that all training examples are classified correctly. We can use the lemma from above to draw conclusions about the VCdim of the structure element that the separating hyperplane comes from. A bound similar to (10) [Shawe-Taylor et al., 1996] gives us a bound on the true error of this hyperplane on our classification task.
Since the optimization problem from above is difficult to handle numerically, Lagrange multipliers are used to translate the problem into an equivalent quadratic optimization problem [7].

$$\text{Minimize: } -\sum_{i=1}^{n} \alpha_i + \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \tilde{d}_i \cdot \tilde{d}_j \quad (13)$$
$$\text{So that: } \sum_{i=1}^{n} \alpha_i y_i = 0 \text{ and } \forall i\text{: } \alpha_i \geq 0 \quad (14)$$

For this kind of optimization problem efficient algorithms exist, which are guaranteed to find the global optimum . The result of the optimization process is a set of coefficients $\alpha_i^*$ for which (13) is minimum. These coefficients can be used to construct the hyperplane fulfilling (11) and (12).

$$\tilde{w} \cdot \tilde{d} = (\sum_{i=1}^{n} \alpha_{*i} y_i \tilde{d}_i) \cdot \tilde{d} = \sum_{i=1}^{n} \alpha_{*i} y_i (\tilde{d}_i \cdot \tilde{d}) \text{ and } b = \frac{1}{2}(\tilde{w} \cdot \tilde{d}_+ + \tilde{w} \cdot \tilde{d}_-) \quad (15)$$

Equation (15) shows that the resulting weight vector of the hyperplane is constructed as a linear combination of the training examples. Only those examples contribute for which the coefficient $\alpha_i$ is greater than zero. Those vectors are called Support Vectors. They are those training examples which have minimum distance to the hyperplane. To calculate b, two arbitrary support vectors $\tilde{d}_+$ and $\tilde{d}_-$ (one from the class + and one from -) can be use.