FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Selection of Representative Samples from Datasets for Malware Detection
Student:	Bc. Lukáš Děd
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

Training datasets for malware detection usually consist of millions of samples. With the growth of sample amounts, classification algorithms became more and more expensive. This thesis aims to compare publicly available datasets for malware detection with respect to the reduction rate and classification accuracy.

Detailed instructions:
1) Find and preprocess at least two publicly available datasets (e.g., EMBER, SOREL-20M).
2) Describe and apply at least five instance selection algorithms to the datasets from 1) using the techniques for dealing with large datasets (such as stratification).
3) Compare the instance selection algorithms from 2) using the datasets from 1) in terms of reduction rate and classification accuracy.
4) Try to propose a new instance selection algorithm or modify some existing one, implement it and compare its results with the instance selection algorithms from 2).

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 22 December 2022 in Prague.

Master's thesis

SELECTION OF REPRESENTATIVE SAMPLES FROM DATASETS FOR MALWARE DETECTION

Bc. Lukáš Děd

Faculty of Information Technology
Department of Information Security
Supervisor: Mgr. Martin Jureček, Ph.D.
February 15, 2024

Contents

List of Figures

List of Tables

List of Algorithms

This way, I would like to express my gratitude to Mgr. Martin Jureček, Ph.D., for his expert guidance without which this thesis could not have been created. I would also like to thank my family and friends for their support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on February 15, 2024 .....

Abstract

This thesis focuses on the selection of representative instances for the training set in malware detection. Experiments were conducted on two publicly available datasets containing metadata of Windows PE files, namely the EMBER and SOREL-20M datasets. The theoretical part describes data preprocessing methods, instance selection algorithms, and classification algorithms used in the practical part of this thesis. It also includes a description of the structure of PE files. The practical part outlines the process of preprocessing datasets and main experiments related to the comparison of state-of-the-art instance selection algorithms. As part of the thesis, modifications to the parallel instance selection algorithm PIF were proposed and implemented, and these were also experimentally evaluated and compared with the results of state-of-the-art instance selection algorithms.

Keywords instance selection, PIF, DROP3, MSS, CNN, ICF, AllKNN, RENN, ENN, KNN, machine learning, artificial intelligence, classification, malware, PE files, Windows

Abstrakt

Tato závěrečná se zabývá výběrem reprezentativních instancí trénovací množiny pro detekci malware. Experimenty byly provedeny na dvou veřejně dostupných datasetech, obsahujících metadata Windows PE souborů. Jedná se o datasety EMBER a SOREL-20M. V teoretické části jsou popsány metody předzpracování dat, instance selection algoritmy a klasifikační algoritmy, použité v praktické části této thesis, a také struktura PE souboru. Praktická část popisuje průběh předzpracování datasetů a hlavní experimenty související s porovnáním state-of-the-art instance selection algoritmů. V rámci závěrečné práce byly navrženy a implementovány modifikace paralelního instance selection algoritmu PIF, které byly rovněž experimentálně vyhodnoceny a porovnány s výsledky state-of-the-art instance selection algoritmů.

Klíčová slova výběr instancí, PIF, DROP3, MSS, CNN, ICF, AllKNN, RENN, ENN, KNN, strojové učení, umělá inteligence, klasifikace, malware, PE soubory, Windows

Introduction

The world of information technology is developing rapidly, especially in recent years. Business environments are moving into the digital world, resulting in an increase in the number of devices connected to the Internet. According to [1], by 2022, 90% of all data was generated in the years 2019-2022. Just like in the real world, in the digital world are also criminal entities trying to achieve their goals through illegal means. The number of these entities in the digital environment is also increasing.

Attackers use many methods and technologies to achieve their goals. Malicious software, abbreviated as **malware**, has long been one of the biggest threats. Malware is software that aims to cause damage to a computer system or an entire network and thus to the owner of those assets [2]. Examples include trojans, worms, or today's increasingly common ransomware, whose goal is to encrypt data in the computer systems of the attacker's targets. According to [3], approximately 560,000 new malware samples are detected every day.

One of the possible methods for malware detection is based on the signatures of executable files. Antivirus programs, relying on signature-based methods, operate by comparing a file against a signature database created from previously obtained malware samples. This detection method achieves good results for already-known versions of malware, emphasizing the importance of working with an up-to-date signature database. However, the effectiveness of this method diminishes for new versions of malware [4]. A potential solution to this problem is the use of machine learning algorithms.

Malware detection using machine learning (ML) algorithms is currently a popular method employed by many antivirus programs. ML algorithms classify files based on their properties (referred to as **features**). Examples of features include file size or the number of imported libraries. An essential phase in ML algorithms is the learning process, which takes place using acquired samples. The learning process involves setting the hyperparameters of the ML algorithm to optimize its performance within the addressed issue. The resulting configuration is then used for the classification of new samples.

In addition to properly tuning the hyperparameters of ML algorithms, their performance can be enhanced by selecting representative samples on which the algorithm is trained. The datasets used for model training commonly contain noise and redundant data, which can have a negative impact on the overall performance of the resulting model. Instance selection algorithms are employed to address this problem [5]. Reducing the size of the dataset also results in shorter training times for ML models and lower mem-

ory requirements. Given the time and memory complexity of some ML algorithms, this is another reason for the application of instance selection algorithms. The task of instance selection algorithms is to reduce the size of the data, ensuring that there is no significant loss in the classification model's performance or, conversely, to achieve improvement. This is accomplished by removing redundant and noisy samples from the dataset.

The main theme of this thesis is the selection of representative samples for malware detection using instance selection algorithms. Part of this work is the experimental evaluation of some state-of-the-art algorithms and experiments with modified versions of existing algorithms. The comparison of instance selection algorithms is performed using two publicly available datasets containing metadata of Windows Portable Executable (PE) files. These are the EMBER [6] and SOREL-20M [7] datasets. Both datasets were preprocessed before experimenting with instance selection algorithms. Another contribution is the experimentation with proposed modifications to the Parallel Instance Filtering algorithm, which is also part of the comparison of instance selection algorithms.

This thesis is divided into seven chapters. In the theoretical part, covered in the first four chapters, the algorithms used and the structure of Windows PE files are described. The practical part of this thesis is outlined in the last three chapters. A brief description of all seven chapters follows:

- **Chapter 1** - This chapter provides an overview of all methods applied to the datasets before experimenting with instance selection algorithms.
- **Chapter 2** - This chapter contains a detailed description of several state-of-the-art instance selection algorithms, including pseudocode descriptions.
- **Chapter 3** - Classification algorithms and metrics used in the experimental evaluation of instance selection algorithms are described in Chapter 3.
- **Chapter 4** - The fourth chapter describes the structure of Windows PE files, including headers and individual sections.
- **Chapter 5** - In this chapter, experiments related to the application of methods described in Chapter 1 are documented. It includes descriptions of individual changes during the preprocessing of both datasets.
- **Chapter 6** - The sixth chapter provides a description of proposed modifications to the Parallel Instance Filtering instance selection algorithm.
- **Chapter 7** - In this chapter, all experiments related to instance selection algorithms outlined in Chapters 2 and 6 are described. It includes the experimental evaluation and comparison of these algorithms in terms of computational time, achieved classification accuracy (or F1 score), and the level of dataset reduction.

Chapter 1

Data preprocessing

This chapter describes some of the commonly used data preprocessing methods intended for classification, except for the description of instance selection algorithms described in a separate chapter. The first section is devoted to data cleaning. The second part deals with converting categorical data types into numerical data types. The third section describes methods for scaling data. Section four describes a method used for reducing the number of features.

1.1 Data cleaning

Data cleaning is an important part of data preprocessing. Training models on unclean data often leads to a decrease in the performance of these models. In real-world scenarios, data is often incomplete (missing values), contains typos, unrealistic values (e.g., a person's height of 18 meters instead of 1.8 meters), or noisy data. Instance selection algorithms (described in Chapter 2) are used to remove noise from data and select representative instances [8]. Data also often contains various inconsistencies (e.g., Jan Novak vs J. Novak, Praha vs. Prague) and it is necessary to ensure that values with the same meaning are represented consistently. To improve the performance of machine learning algorithms, it is necessary to remove these errors [9]. Some algorithms cannot even work with missing values in the data (e.g., K Nearest Neighbors). The following sections describe data cleaning methods that deal with completing missing values, removing redundant information from the data, and detecting and removing outliers.

1.1.1 Missing values

Data are represented as matrices of values, where the rows are formed by individual cases (instances) and the columns of the matrix are formed by attribute values (features). Some instances may have missing values for certain features. In order for machine learning algorithms to function properly, it is necessary to find and replace missing values. Finding missing values can sometimes be a problem, as these missing values can be represented in many ways (NaN, N/A, None, ?, ??, Empty, etc.) and it is necessary to transform them into a correct representation depending on the tool used for imputation. There are

several imputation methods based on different approaches. Choosing the appropriate method can have a significant impact on overall classification/prediction accuracy and it is therefore important not to underestimate this step. Before choosing a method, it is important to understand why data is missing and how these missing data are represented. There are three basic groups of missing values [10]:

- **MCAR - missing completely at random** - The absence of a feature value does not depend on the data (the value is not missing due to a systematic error), for example, due to a connection failure when collecting data from sensors over the internet.
- **MAR - missing at random** - The occurrence of a missing value may depend on the value of another feature, for example, sensor failure depending on wind speed.
- **MNAR - missing not at random** - It is known under which conditions the missing values will occur.

In the following sections, some of the commonly used methods for imputing missing values are described. Section 1.1.1.1 deals with replacement by a constant value. Section (1.1.1.2) outlines imputation using statistical properties of the features.

1.1.1.1 Impute constant value

One of the options is to replace a missing value with a constant value. According to [11, p. 105], it is often the case that the missing value is replaced with a constant value that is a valid value from the domain of the corresponding column. This is not an appropriate solution to the missing value problem. The selected constant should be a value outside of the domain of the feature. Common values are, for example, -1 for natural numbers or 'UNKNOWN' for categorical data. This approach is used when dealing with missing values of the type MCAR. This method is straightforward and easy to implement. However, imputing a constant value can also introduce bias into the dataset, especially if the missing values are not of the MCAR type.

1.1.1.2 Imputation using basic statistical properties

Imputation of missing values with a constant whose value is determined in a more sophisticated way. This technique based on descriptive statistics is used for filling in missing values in a dataset using the statistical properties of the data. These properties are calculated from the other (non-missing) values of a given feature. The most commonly used metrics are mean, median, and mode [12].

When using the **mean** as the metric for filling in missing values, the arithmetic mean (1.1) is calculated from the available data (values of the feature), and this value is used to fill in the places containing missing values. This technique is fast, simple, and easy to implement, but it has several disadvantages. Mean imputation is sensitive to outliers, which can significantly increase/decrease the calculated value. This imputation method can also introduce bias into the data, especially if the missing values are not of the MCAR type. If the number of missing values is high, it can lead to a reduction in variance, as a large number of identical values are added. A large number of identical values can create a new peak in the distribution and thus distort it (if it is not a normal distribution) [13]. This is a numerical method and therefore not suitable for categorical columns.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1.1}$$

where:

- $\bar{x}$ is the sample mean,
- x_i is the i-th value,
- N is the number of values

In **median** imputation, missing values are replaced by the median. To perform median imputation, the first step is to sort the values in the dataset. Then, the median is calculated based on the number of values in the dataset. For columns with an odd number of values, the median is the middle value of the sorted array. For columns with an even number of values, the median is the average of the two middle values (of sorted values). The median method is more robust to outliers than the mean method and is more suitable for skewed distributions, other properties of these techniques are almost the same.

Another simple and straightforward technique that is simple to use is **mode** imputation. This method is suitable for categorical data. Missing values in a column are replaced with the value that has the highest frequency in that column.

1.1.2 Basic methods for removing redundancy

In the field of data preprocessing, there are several types of redundancy. It can be, for example, redundant information that does not contribute to the appropriate solution of the given problem or multiple occurrences of the same or very similar data. The following sections describe the solution to redundancy by removing duplicate data (1.1.2.2) and features containing constant values (1.1.2.1) [14]. To remove redundancy caused by the presence of similar instances, instance selection algorithms are used, which aim to select a representative subset of data. These algorithms are described in more detail in Chapter 2. To remove redundant columns (e.g., features with high correlation), feature selection algorithms are used. Another way to select representative features (and thus remove redundant columns) is to use feature extraction techniques discussed in Section 1.4.

1.1.2.1 Removal of features with constant value

One type of redundancy is features containing a constant value, where the value of one or more features is the same for every instance. These columns do not provide any information and, therefore, do not contribute to the model's performance. Since these features are not useful and increase the computational complexity of the model, it is advisable to remove them. The variance of a feature containing only a single value is zero [15]. Based on this fact, these columns can be identified and subsequently removed. Another way to detect constant value features is to calculate the number of unique values for all columns and then remove those with only one value. Figure 1.1 shows an example of a data matrix containing a column with a constant value.

ID	x_1	x_2	· · ·	x_j	· · ·
0	0.25	0.48	· · ·	1	· · ·
1	0.21	0.15	· · ·	1	· · ·
2	0.17	0.33	· · ·	1	· · ·
⋮					
n-1	0.45	0.51	· · ·	1	· · ·
n	0.32	0.74	· · ·	1	· · ·

Figure 1.1 Example of a column with a constant value

1.1.2.2 Removal of duplicate rows

We refer to two or more identical rows in a dataset as duplicate rows. These are instances whose feature values (feature vectors) are identical. If such values occur in the data, it is necessary to keep only a single occurrence and remove the remaining rows (only in some cases). After splitting the dataset into training and test sets, identical instances may appear in both subsets and thus bias the results of the model, which could affect its overall reliability and accuracy. On the other hand, if we need to assign weights to individual instances, this is one of the options. The greater the occurrence of a given instance in the dataset, the greater its weight. Removing duplicate rows can also lead to the loss of information, for example, if identical rows represent different time moments. Therefore, it is necessary to consider each case separately to determine whether applying this method is appropriate. Figure 1.2 shows an example of duplicate rows .

x_1	x_2	x_3	x_4	x_5
0.25	0.48	0.67	0.37	0.48
0.17	0.33	0.07	0.72	0.49
0.17	0.33	0.07	0.72	0.49
0.45	0.51	0.78	0.94	0.26

Figure 1.2 Example of duplicate rows

1.1.3 Outlier handling

An outlier is an extreme value that significantly differs from the rest of the population, and its occurrence in a data set is unlikely. Outliers can be included in the data naturally or artificially [16]. An example of a natural outlier is salaries in a factory. Managers, who are significantly fewer in number, usually have significantly higher salaries than the rest of the factory. These salaries can appear as outliers in the data. An artificially created outlier can arise, for example, from a sensor incorrectly measuring a value. The occurrence of outliers in data can have a significant impact on some machine learning algorithms and data preprocessing methods that are sensitive to these deviations (such as min-max normalization described in 1.3.1). The search for outliers should involve a domain expert who has an overview of the problem

at hand and can decide whether a particular value is an outlier or not. The detection of outliers can be divided into four basic categories (some methods may fall into more than one of these categories.) [17]:

- **Statistical methods** - Statistical data processing techniques that assume significant difference of outliers from normal values are used to detect outliers. Examples include the Standard Deviation method, Z-score method, and Interquartile Range Method.
- **Distance-based methods** - Distance-based methods detect outliers based on information about the distances between individual data points. Examples include K Nearest Neighbors or Local Outlier Factor (LOF).
- **Density-based methods** - The search for outliers is based on the assumption that these values have a significantly different density than normal data. Examples include Density-Based Spatial Clustering of Applications with Noise or LOF.
- **Deviation-based methods** - Deviation-based methods detect outliers based on the deviation from the central tendency of the data. Examples include Standard Deviation method, Z-score method, and Interquantile Range method.

There are many ways to deal with detected outlier values. Below are several commonly used methods:

- **Removal of outliers** - The method involves removing instances containing outlier values.
- **Replacement of outliers with boundary values** - Another option is to replace outlier values with the upper or lower bound of the range of normal values. If the outlier value is lower than the values in the range, this deviation is replaced with the lower bound value, and if the outlier value is higher than the upper bound, it is replaced with this value.
- **Transformation** - Outliers can also be dealt with by transforming the data. Data transformation refers to the application of a mathematical function (e.g., logarithm) in order to obtain ”more normally” distributed data.
- **Imputation** - Replacing an outlier value with another (normal) value (e.g., mean, median).

  In Section 1.1.3.1, the Interquartile Range method, which was used in this thesis, is described.

1.1.3.1 Interquartile Range method

Interquartile Range (IQR) outlier detection is another statistical-based method. Detecting outliers involves finding patterns that do not conform to the normal distribution of the processed data. Outliers are data points whose values are either below the lower threshold or above the upper threshold [18]. As the name suggests, the upper and lower threshold is calculated based on the interquartile range. This value is calculated as the difference between the third and first quartile (1.2).

$$IQR = Q3 - Q1 \tag{1.2}$$

where:

- IQR is the value of interquantile range,
- $Q3$ is the 75th percentile,
- $Q1$ is the 25th percentile

Subsequently, using the IQR, the thresholds are calculated. The upper threshold is obtained by adding a multiple of the IQR to the 75th percentile, while the lower threshold is obtained by subtracting a multiple of the IQR from the 25th percentile. The value of 1.5 is often chosen as the multiplier [19]. If the goal is to identify extreme outliers, a multiplier of 3 (or more) is used. Values that do not fall within the calculated thresholds are marked as outliers. The formulas for calculating the upper and lower thresholds are shown in equations (1.3) and (1.4).

$$lowerbound = Q1 - k * IQR \tag{1.3}$$

$$upperbound = Q3 + k * IQR \tag{1.4}$$

where (for both equations):

- k is the multiple that determines how many values will be marked as outliers

The steps described above are summarized in Algorithm 1.

Algorithm 1 IQR outlier detection

1: sort the dataset in ascending order
2: calculate Q1 - the value that separates the first 25% of values from the remaining 75%
3: calculate Q3 - the value that separates the first 75% of values from the remaining 25%
4: calculate the IQR
5: calculate the upper and lower thresholds
6: mark the outliers (values that do not fall within the calculated range)

1.2 Convert category features to numerical

In real-world data, categorical variables often appear. These variables can be divided into two basic categories: nominal and ordinal. Nominal features contain category values that do not depend on the order. These include variables such as gender (male/female), color of an object (blue, red, yellow, etc.), or religion (Christian, Muslim, etc.). The values of ordinal variables have a natural order in the real world. An example of an ordinal variable is food rating (excellent, average, poor). Many machine learning algorithms used for classification (e.g., K nearest neighbors classifier, neural networks) require only

numeric inputs to function properly. To use these algorithms without losing information from categorical variables, it is necessary to replace their values with a numeric representation. This work presents two basic methods for converting categorical variables to numerical ones. The first method is encoding (see 1.2.1), and the second method is feature hashing (see 1.2.2), which is suitable for variables with different lengths. In most cases, both methods result in an increase in the number of variables.

1.2.1 Encoding

As mentioned above, based on their functioning principles, some machine learning algorithms require only numerical values as inputs. One way to convert non-numerical categories to numerical is through encoding. There are many encoding methods available, and the selection of a suitable method depends on the categorical variable that we want to encode. Nominal features are often encoded using the one-hot encoding method (see 1.2.1.1).For ordinal features, we can use the Ordinal encoding method. Label encoding is often used for the target variable (see 1.2.1.2). These methods are described in more detail in the following sections.

1.2.1.1 One-hot encoding

One-hot encoding is used to encode nominal features (i.e., variables that have no inherent order) [20]. The transformation is performed by replacing the original column with a binary vector of length k, where k is the number of unique values of the categorical variable (i.e., one binary feature is created for each category). The variable for the category present in the instance takes on a value of one, while the remaining variables take on a value of zero. The newly created variables are often referred to as dummy variables [21]. This method solves the problem of missing values. If a value is missing in the data, all dummy variables will be set to zero. An example of using one-hot encoding is shown in Figure 1.3.

color = {red, green, blue}

color
red
blue
red
green

⟹

color_r	color_g	color_b
1	0	0
0	0	1
1	0	0
0	1	0

Figure 1.3 One-hot encoding example

1.2.1.2 Label Encoding

Label encoding is used to encode the target variable [22]. Unlike ordinal encoding, the order of individual values does not matter. Each label is assigned an integer value. The principle of operation is the same as the ordinal encoding method, but in this case, the order of individual values is not considered important. The example of label encoding is shown in Figure 1.4.

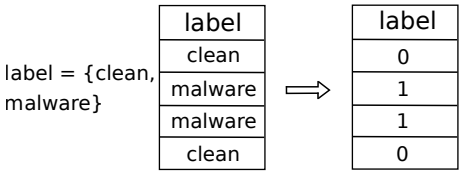

Figure 1.4 Label encoding example

1.2.2 Feature hashing

If a categorical variable has a large number of unique values, one-hot encoding may be impractical, as creating a dummy variable for each unique value results in an increase in the dimensionality of the data. With increasing dimensionality, computational complexity and storage requirements increase as well. One possible solution is to use one-hot encoding only for the k most frequent values, and merge the remaining values into a single category. In these cases, feature hashing, also known as the hashing trick, can be used as well [23]. This method typically creates a smaller number of dummy variables than the number of unique values of the original categorical variable. The size of the input value domain is therefore usually larger than the size of the output value domain. As a result, collisions may occur, meaning that several input values can be transformed into the same output value. The input to the algorithm is the number of output variables and the hash function. The hashing trick can also be used to create a feature vector from documents and generally from variable-length information (e.g., the number of sections in the header of a PE file may vary).

1.3 Feature scaling

Feature scaling is one of the most important techniques used in data preprocessing that can significantly affect the performance of machine learning algorithms. Some machine learning algorithms (Support Vector Machine, K nearest neighbors, Principal Component Analysis, etc.) are sensitive to this problem [24]. The domains of values of individual features differ commonly in real datasets. This fact leads to some features containing larger values and thus significantly influencing the output value of the algorithm. Features containing small values are suppressed and do not significantly affect the output of the algorithm, even though they may have significant informative value for solving the given task. Feature scaling methods address this problem by transforming data into a chosen range that is identical for all features. The following sections describe some commonly used algorithms for feature scaling. Normalization is described in section 1.3.1, standardization is described in section 1.3.2, and robust scaling is summarized in section 1.3.3.

1.3.1 Min-max normalization

One of the most commonly used methods for scaling features is normalization [24], also known as min-max scaling. This method transforms features so that the original distribution of the data is preserved.

The result of the transformation is values ranging from 0 to 1. As the minimum value, maximum value, and mean are included in the calculation, this method is sensitive to outliers. The formula for calculating the transformed values includes the equation (1.5).

$$y_i = \frac{x_i - min}{max - min} \tag{1.5}$$

where:

- y_i is the scaled i-th value
- x_i is the i-th value to be scaled
- min is the minimum value of the given feature across the entire dataset (or training set).
- max is the maximum value of the given feature across the entire dataset (or training set).

1.3.2 Standardization (z-score normalization)

Another technique for dealing with features with different value ranges is standardization [25]. This method can be divided into two parts. First, the dataset is centered by subtracting the mean value of each column from each value of that column. This operation is called centering. Centered columns have a mean value of zero. After centering the data, the feature is scaled by dividing it by its standard deviation. Features that have been scaled have a standard deviation of one. Due to the operations used, standardization is sometimes referred to as center scaling or z-score normalization. Standardization assumes a normal distribution of the scaled data. If the data does not have a normal distribution, the results may not be satisfactory. Due to the use of the mean value in the calculation of the transformed values, this method is also sensitive to outliers, but less so than min-max normalization. The formula for calculating the standardized value includes equation (1.6).

$$y_i = \frac{x_i - \bar{x}}{\sigma} \tag{1.6}$$

where:

- $\bar{x}$ is the mean calculated over all values of the given feature
- σ is the standard deviation of the feature

1.3.3 Robust scaling

Robust scaling [26] is a data scaling method designed to be robust against outliers, which can skew the shape of the probability distribution and thus affect statistical characteristics such as mean and standard deviation, minimum and maximum values. The technique of robust scaling uses characteristics whose sensitivity to outliers is low during calculation. Instead of subtracting the mean/min from the original value, the median (50th percentile or 2nd quantile) is used and the result is then divided by the difference between the third quartile (75th percentile) and the first quartile (25th percentile), i.e., the interquartile

range, instead of standard deviation/range of max-min. The formula for calculating the value transformed using robust scaling includes equation (1.7). Scaled features have a mean and median of zero and a standard deviation of one.

$$y_i = \frac{x_i - q_2}{q_3 - q_1} \tag{1.7}$$

where:

- q_2 is the median (50th percentile) of all values of the given feature
- q_3 is the 3rd quartile (75th percentile) of all values of the given feature.
- q_1 is the 1st quartile (25th percentile) of all values of the given feature.

1.4 Feature extraction

One option for reducing the number of features and mitigating negative impacts (such as computational complexity or overfitting) in models trained on high-dimensional data is to use feature extraction techniques. The goal of feature extraction is to transform the original set of features into another set of features while preserving as much available information as possible. There are several methods that can be divided into two main groups: supervised and unsupervised [27]. The unsupervised Principal Component Analysis method, which was used for feature extraction in this thesis, is described in Section 1.4.1 .

1.4.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) [28] is a linear method for dimensionality reduction of a dataset. The PCA algorithm was invented by mathematician Karl Pearson in 1901 [29]. The method does not require any information about the target variable during computation, making it an unsupervised method. As PCA involves an orthogonal transformation, the result is a set of linearly uncorrelated features that capture the maximum variance [30]. These transformed features are linear combinations of the original features and are called principal components. This means that each principal component is a linear combination of the original features. The first principal component contains the maximum possible variance among all linear combinations, the second principal component contains the maximum variance orthogonal to the first principal component, and so on. In 2, the procedure of the PCA algorithm is described [31].

Algorithm 2 PCA

1: standardize all features ▷ PCA requires zero mean and unit variance for all features
2: **for each** random variable (feature) X_i **do**
3: compute the variance of X_i (1.8)

$$var(X_i) = \frac{\sum_{y=1}^{n}(x_{iy} - \bar{x_i})^2}{n-1} \tag{1.8}$$

where:
$var(X_i)$ is the variance of the i-th random variable
x_{iy} is the value of the y-th element of the i-th random variable
$\bar{x_i}$ is the mean value of the i-th random variable
n is the number of values of the i-th random variable

4: compute the (pairwise) covariance between X_i and other random variables (1.9):

$$cov(X_i, X_j) = \frac{\sum_{y=1}^{n}(x_{iy} - \bar{x_i})(x_{jy} - \bar{x_j})}{n-1} \tag{1.9}$$

where:
$cov(X_i, X_j)$ is the covariance between the i-th and j-th random variables, where i $\neq$ j
x_{jy} is the value of the y-th element of the j-th random variable
$\bar{x_j}$ is the mean value of the j-th random variable
n is the number of instances

5: **end for**
6: based on the computed variances and covariances, create a covariance matrix
7: compute the eigenvectors and eigenvalues of the covariance matrix (1.10)

$$Cv = \lambda v \tag{1.10}$$

where:
C is the covariance matrix
v is the eigenvector of the covariance matrix
λ is the eigenvalue of the covariance matrix

8: sort the eigenvectors in descending order based on the magnitude of eigenvalues
9: select the top k principal components
10: transform the data using the transformation matrix formed by the k selected eigenvectors (1.11)

$$X_{new} = W^T X \tag{1.11}$$

where:
X_{new} is the resulting feature matrix after transformation into the new subspace
X is the original feature matrix
W is the transformation matrix formed by the top k eigenvectors arranged as columns

Chapter 2

Instance selection algorithms

Another possibility to reduce the size of data is to apply instance selection (IS) algorithms. The task of these algorithms is to decrease the number of instances while preserving or even improving the classification/prediction ability [32]. This is achieved by removing noise in the data or eliminating irrelevant and redundant instances. IS algorithms are applied to the training dataset (see 3), and the output is a subset of this set that is sufficiently representative for solving the given problem. Similarly to feature selection, by reducing the number of instances, computational and memory requirements, as well as storage demands, are decreased. In addition to the metrics described in Section 3.2, the suitability of an IS algorithm is compared based on the so-called reduction rate, which describes the extent to which the original dataset has been reduced. Usually, it is necessary to find a compromise between the reduction rate and the resulting accuracy of the model. There are many instance selection algorithms, which can be divided into three basic types based on the way instances are selected [33]:

- **Condensation algorithms** - The principle of these methods is to retain instances that are close to the decision boundary (known as 'border points'), while removing distant instances (referred to as 'internal points') from the training set. Condensation techniques are based on the assumption that internal points have less influence on the formation of the decision boundary (due to their distance from it) and can therefore be eliminated. These algorithms typically achieve a good reduction rate but are prone to overfitting, resulting in a loss of generalization ability on unseen data. Examples of these algorithms include Condensed Nearest Neighbor and its modifications, Modified Selective Algorithm, Max Nearest Centroid Neighbor, Reduced Nearest Neighbor, and Minimal Consistent Set.
- **Edition algorithms** - Contrary to condensation algorithms, these techniques focus on removing some border points while retaining internal points in the training set. Edition algorithms remove instances near the decision boundary whose labels differ from the labels of their nearest neighbors. This results in noise removal and smoothing of the decision boundary. These algorithms are less prone to overfitting but achieve poorer results in terms of data reduction. Examples of Edition algorithms include Edited Nearest Neighbor, AllKNN, Multiedit, and Relative Neighborhood Graph Edition .

- **Hybrid algorithms** - Hybrid algorithms are a combination of the previous two methods. Reduction of the number of instances is achieved by removing both internal and border points. Examples of hybrid algorithms include Instance-Based Learning Algorithm, Decremental Reduction Optimization Procedure or Iterative Case Filtering.

Another possible way to divide instance selection algorithms is according to the direction in which the training set is searched [34]. The following is a description of the individual options:

- **Inceremental** - As the name suggests, incremental algorithms start with an empty subset T_{new}, the size of which gradually increases as the instances are traversed. These are order-dependent techniques, i.e., it depends on the order in which the instances are traversed. The algorithms go through all the instances that are subsequently added to the reduced subset T_{new} if they meet a certain condition. The advantage of incremental algorithms is that newly acquired instances can be added to the subset T_{new} additionally, which makes them a suitable option for online learning and data stream processing. Examples of incremental algorithms include Condensed Nearest Neighbor and its modifications or Instance-Based Learning Algorithm.

- **Decremental** - These techniques start with a reduced subset of the same size as the original training set ($T_{new} = T$). Then, all instances that satisfy a certain condition are sequentially examined, and if the condition is met, they are removed from the subset T_{new}. If the condition is met, the instance is removed immediately after it has been tested. Unlike incremental algorithms, decremental algorithms are typically more computationally demanding and require access to all available data for computation. Examples of decremental algorithms include Reduced Nearest Neighbor, Modified Selective Subset, Edited Nearest Neighbor, and Parallel Instance Filtering.

- **Batch** - Like Decremental algorithms, Batch methods start with a reduced set T_{new} that is identical to the original training set T. The difference is that if one of the instances fulfills the condition while going through the instances, it is not removed immediately but only marked as a candidate for removal. Deletion of the marked data occurs only at once after passing all (or a selected number of) instances. Examples of batch algorithms include Max Nearest Centroid Neighbor, AllKNN or Iterative Case Filtering.

- **Mixed** - Mixed algorithms operate on a pre-selected reduced subset T_{new}, which is either randomly chosen or obtained through incremental/decremental techniques. Based on a chosen criterion, these algorithms iteratively remove or add instances. A special case of mixed algorithms is called **fixed** algorithms, where the number of instances to be added and removed from T_{new} is predetermined and fixed (the number of additions and removals is always the same).

The methods described above are summarized in the image 2.1, which contains a taxonomy of instance selection algorithms. The following sections describe some of the condensation (section 2.1), edition (section 2.2) and hybrid (section 2.3) algorithms. In all cases, these are wrapper methods.

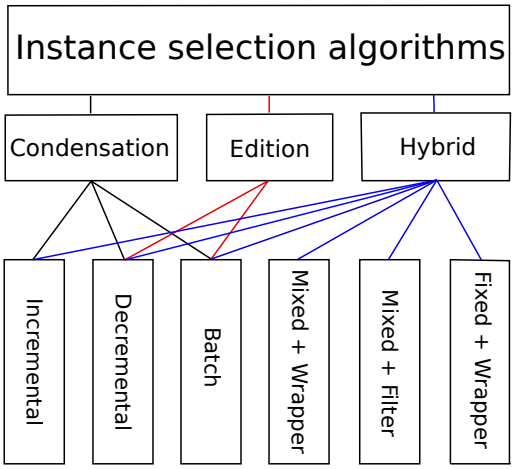

Figure 2.1 Taxonomy of instance selection algorithms

2.1 Condensation algorithms

Condensation algorithms reduce the size of the dataset by removing internal points, i.e., points far from the decision boundary (see description at the beginning of this chapter). The following sections describe some of the condensation algorithms. Section 2.1.1 describes the incremental algorithm Condensed Nearest Neighbor and the decremental algorithm Modified Selective Subset is summarized in section 2.1.2.

2.1.1 Condensed Nearest Neighbors (CNN)

Condensed Nearest Neighbor [35] was created in 1968 by Hart and is considered one of the first instance selection algorithms. It is an incremental algorithm, i.e., incrementally creates a reduced subset T_{new} by successively traversing instances of the original training set T.

At the beginning, a random instance from the set T is selected and moved to an empty set T_{new}. Subsequently, the remaining instances are classified using a 1-NN classifier trained on the set T_{new}. If an instance is classified incorrectly, it is moved to the set T_{new}. The classification of each instance is always performed using the most up-to-date version of T_{new}. This iteration over all instances in T is repeated until either no instances are moved to T_{new} during a complete cycle or the set T becomes empty.

The output of the CNN algorithm is not a minimal consistent set but only a consistent set. A minimal consistent set is one in which all instances from the original set are correctly classified using a 1-NN classifier, and this set cannot be further reduced without violating this condition [36]. In other words, a subset T_{new} ($T_{new} \subseteq T$) is consistent if, for all instances x from the original set T, their nearest neighbor from the set T_{new} has the same class as x. The entire procedure is summarized in pseudocode 3 .

Algorithm 3 CNN

Let:
T be the original dataset
T_{new} be the reduced consistent subset from the original dataset T
$T_{new} \leftarrow \emptyset$
randomly select an initial instance x_{start} from T and move it to T_{new}, i.e., $T \leftarrow T \setminus \{x_{start}\}$ and $T_{new} \leftarrow T_{new} \cup \{x_{start}\}$
$moved \leftarrow false$
for each $x \in T$ **do**
    classify x using a 1-NN classifier with T_{new} as the training set
    if x is misclassified **then**
        move x from T to T_{new}, i.e., $T \leftarrow T \setminus \{x\}$ and $T_{new} \leftarrow T_{new} \cup \{x\}$
        $moved \leftarrow true$
    end if
end for
if $moved$ AND $T \neq \emptyset$ **then**
    go back to the step 3
end if
return T_{new}

There are many modifications of the CNN algorithm [37]. Examples include Tomek Condensed Nearest Neighbor, Modified Condensed Nearest Neighbor, or Generalized Condensed Nearest Neighbor.

2.1.2 Modified Selective Subset (MSS)

One of the representatives of decremental algorithms, which also falls into the category of condensation algorithms, is Modified Selective Subset [38]. Decremental algorithms start working with the set $T_{new} = T$ and while traversing individual instances, they remove from T_{new} those instances that meet the selected criterion (a more detailed description of decremental algorithms can be found at the beginning of this chapter). The Modified Selective Subset algorithm is based on the so-called *selective subset*. The set $T_{new} \subseteq T$ is a selective subset if:

- Consistent (see Section 2.1.1)
- For all instances x from the original dataset T, it holds that the distance between their nearest neighbor in the selective subset T_{new}, which belongs to the same class as x, is smaller than the distance to the nearest enemy of x in the original set T. The nearest enemy refers to the closest neighbor that belongs to a different class.

As the name suggests, the output of the described algorithm is a modified selective subset. Modified selective subset can be defined based on the following terms:

- **Related neighbor** - An element x_j is a related neighbor of element x_i belonging to the same class if the distance between x_j and x_i is smaller than the distance between x_i and its nearest enemy.
- **Relative neighborhood of element** x_i - The relative neighborhood RN_i of element x_i refers to the set of all its related neighbors.

- **Modified Selective Subset** - A subset MSS of the original dataset T is called a modified selective subset if, for all elements x from T, MSS contains the furthest relative neighbor of x from its relative neighborhood.

The proposed algorithm processes elements of each class separately. Firstly, it sorts all elements of a given class in ascending order based on their distances to their nearest enemies. Then, each element x is compared to other elements of the same class that have a higher index in the sorted array (including x itself). If the compared element y is part of the set S (where initially, S contains all elements belonging to the currently processed class) and the distance between x and y is smaller than the distance between y and its nearest enemy, the element y is removed from the set S. If there was at least one change in the set S during this traversal, the element x is added to the set MSS (where initially, $MSS = \emptyset$). The process continues until the set S is emptied or all elements x have been processed. This procedure is performed for all classes and is summarized in pseudocode 4.

Algorithm 4 MSS

Let:
T be the original dataset
n be the number of classes in the dataset
S is an array of length n containing subsets of elements from the dataset T, divided according to the classes to which the elements belong
MSS be the modified selective subset
$dist(y_j, ne_{y_j})$ be the distance of the j-th element y to its nearest enemy ne_{y_j}
$dist(x_i, y_j)$ be the distance between the i-th element x and j-th element y

```
1:  MSS = ∅
2:  for each S_n ∈ S do
3:      for each x_i ∈ S_n do
4:          add ← false
5:          for each y_j ∈ S_n, where j ≥ i do
6:              if dist(x_i, y_j) < dist(y_j, ne_{y_j}) then
7:                  S_n ← S_n \ {y_j}
8:                  add ← true
9:              end if
10:         end for
11:         if add then
12:             MSS ← MSS ∪ {x_i}
13:         end if
14:         if S_n = ∅ then
15:             continue to the next S_n
16:         end if
17:     end for
18: end for
19: return MSS
```

2.2 Edition algorithms

The task of edition algorithms is to reduce data by removing noise. Using these techniques, ”problematic” border points are removed, which results in a smoothing of the decision boundary. A more detailed

description of the working principle of edition algorithms is at the beginning of this chapter. According to [39, p. 10], due to the low rate of data reduction, edition algorithms are not used independently and in practice are part of more complex (hybrid) algorithms. The decremental Edited Nearest Neighbor algorithm is described in 2.2.1. The AllKNN batch algorithm, which is a modification of the Edited Nearest Neighbor algorithm, is described in section 2.2.2.

2.2.1 Edited Nearest Neighbors (ENN)

Edited Nearest Neighbor [40] is a representative of decremental algorithms. The output of the decremental algorithms is the set T_{new}, which at the beginning is identical to the original set T and its size is reduced by successive traversal. During the process of reducing the set T, classification using the K Nearest Neighbor classifier (see 3.1) is used. The appropriate value of the parameter K is usually chosen based on experiments. Before applying ENN, we set $T_{new} = T$. The Edited Nearest Neighbor algorithm first finds for all elements x from the original set T their K nearest neighbors (without elements x) according to the selected distance metric (see 3.1). Using the found K nearest neighbors, classification is performed for all elements x. Elements x whose actual class does not match the classified class are removed from T_{new}. The procedure described above is summarized in pseudocode 5.

Algorithm 5 ENN

Let:
T be the original dataset
K be the parameter of the KNN algorithm indicating the number of nearest neighbors
n be the number of elements in the dataset T
$target_{KNN_i}$ be the class assigned to element x_i by the KNN algorithm
$target_{R_i}$ be the true class of element x_i

1: **for each** $x_i \in T$, where $i = 1,...,n$ **do**
2: calculate the distances between x_i and the other elements in T
3: sort the elements in ascending order based on their distances to x_i
4: obtain $target_{KNN_i}$ by classifying x_i according to the majority class of its K nearest neighbors
5: **end for**
6: **for each** $x_i \in T$, where $i = 1,...,n$ **do**
7: **if** $target_{KNN_i} \neq target_{R_i}$ **then**
8: $T = T \setminus \{x_i\}$
9: **end if**
10: **end for**
11: **return** T

A modified version of the ENN algorithm is Repeated Edited Nearest Neighbor (RENN). This algorithm further smooths the decision boundary by repeatedly applying the ENN algorithm until all remaining elements have the same majority class of k nearest neighbors with their class. The individual steps of the RENN algorithm are summarized in pseudocode 6.

Algorithm 6 RENN

Let:
T be the original dataset
K be the parameter of the KNN algorithm indicating the number of nearest neighbors
n be the number of elements in dataset T
$target_{KNN_i}$ be the class assigned to element x_i by the KNN algorithm
$target_{R_i}$ be the true class of element x_i

1: **for each** $x_i \in T$, where $i = 1, ..., n$ **do**
2: calculate the distances between x_i and the other elements in T
3: sort the elements in ascending order based on their distances to x_i
4: obtain $target_{KNN_i}$ by classifying x_i according to the majority class of its K nearest neighbors
5: **end for**
6: $missclassified \leftarrow false$
7: **for each** $x_i \in T$, where $i = 1, ..., n$ **do**
8: **if** $target_{KNN_i} \neq target_{R_i}$ **then**
9: $T = T \setminus \{x_i\}$
10: $misclassified \leftarrow true$
11: **end if**
12: **end for**
13: **if** $misclassified$ **then**
14: go back to step 1
15: **end if**
16: **return** T

2.2.2 AllKNN

AllKNN [41] is a modification of the Edited Nearest Neighbors algorithm proposed by Tomek. The operating principle of the AllKNN method consists in repeatedly applying the ENN algorithm, each time for a different number of nearest neighbors. These are values from 1 to K, where K is an optional parameter. This is a batch method, i.e., misclassified instances are during traversal only flagged and they are removed at once at the end of the algorithm. The entire procedure of the AllKNN algorithm is included in pseudocode 7.

Algorithm 7 AllKNN

Let:
T be the original dataset
K be the parameter of the KNN algorithm, with values ranging from 1 to K
n be the number of elements in dataset T
$target_{KNN_i}$ be the class assigned to element x_i by the KNN algorithm
$target_{R_i}$ be the true class of element x_i
$flags$ be an array of flags, where the index of the flag corresponds to the index of the instance

1: set all values in $flags$ to 0

AllKNN Algorithm (continued)

```
2:  for each nn ∈ {1...K} do
3:      for each x_i ∈ T, where i = 1,...,n do
4:          calculate the distances between x_i and the other elements in T
5:          sort the elements in ascending order based on their distances to x_i
6:          obtain target_{KNN_i} by classifying x_i according to the majority class of its nn nearest neighbors
7:      end for
8:      for each x_i ∈ T, where i = 1,...,n do
9:          if target_{KNN_i} ≠ target_{R_i} then
10:             set flags[i] = 1
11:         end if
12:     end for
13: end for
14: for each x_i ∈ T, where i = 1,...,n do
15:     if flags[i] = 1 then
16:         T ← T \ {x_i}
17:     end if
18: end for
19: return T
```

2.3 Hybrid algorithms

Hybrid algorithms are combinations of multiple methods, making them more complex. By applying hybrid algorithms, usually both inappropriate border points and internal points that do not have a significant impact on decision border can be removed. The goal is to retain only representative border points in the reduced set, which are close to the decision border and have the biggest influence on its formation. The following subsections describe some hybrid algorithms. Subsection 2.3.1 describes the decremental algorithm Decremental Reduction Optimization Procedure version 3. The fully parallel algorithm Parallel Instance Filtering is described in subsection 2.3.2, and subsection 2.3.3 summarizes the Iterative Case Filtering algorithm.

2.3.1 Decremental Reduction Optimization Procedure 3

The Decremental Reduction Optimization Procedure (DROP) [39] family is a group of five decremental methods designed for instance selection. According to [42, p. 140], they are among the best wrapper algorithms in terms of achieved accuracy and data reduction. The DROP3 version achieves the best results among these DROP methods. The third version of the DROP algorithm combines the methods of ENN and DROP2.

First, the instance set is reduced using the ENN algorithm. Then, the instances are sorted in descending order based on their distances to their nearest enemies. This step ensures that internal points, which are farther from the decision boundary and thus have a greater distance to their nearest enemies, are removed first. For each instance, a list of $K+1$ nearest neighbors (where K is the input parameter of the algorithm) and a list of so-called associates are created. Associates of an instance x are considered those instances that have x among their K nearest neighbors. Subsequently, instances x are removed if

the number of associates correctly classified without x as a neighbor (using the $(K+1)$-th neighbor instead) is greater than or equal to the number of associates correctly classified when instance x is taken into account. If an instance x is removed from the dataset, the list of nearest neighbors of all associates of x must be updated, replacing x with another nearest neighbor nn, ensuring that each remaining instance still has $K+1$ nearest neighbors. After finding a new neighbor nn for element a, a is added to the list of associates of nn. This is done for all a that were affected by the removal of x. Once all instances have been processed, the algorithm returns the reduced dataset. The individual steps of the described procedure are summarized in pseudocode 8.

Algorithm 8 DROP3

Let:
T be the original dataset
T_{new} be the reduced dataset
K_{ENN} be the input parameter of the ENN algorithm
K be the number of neighbors used for classifying associates
n_{with} be the number of associates of x correctly classified with the help of x
$n_{without}$ be the number of associates of x correctly classified without the help of x (using the $(K+1)$-th neighbor instead)
A_x be the list of associates of element x

$T_{new} \leftarrow T$
$T_{new} \leftarrow ENN(T_{new}, K_{ENN})$
sort the elements of T_{new} in descending order based on their distances to their nearest enemies
for each $x \in T_{new}$ **do**
    find the $K+1$ nearest neighbors of x in T_{new}
    add x to the associates lists of its K nearest neighbors
end for
for each $x \in T_{new}$ **do**
    if $n_{without} \geq n_{with}$ **then**
        $T_{new} \leftarrow T_{new} \setminus \{x\}$
        for each $a \in A_x$ **do**
            remove x from the nearest neighbors list of a
            find a new nearest neighbor nn_{new} for element a
            add a to the associates list of nn_{new}
        end for
    end if
end for
return T_{new}

2.3.2 Parallel Instance Filtering

Another representative of hybrid algorithms is Parallel Instance Filtering (PIF) [32]. This is a decremental method. The PIF algorithm can be divided into three main parts, each of which can be parallelized, allowing it to be used (unlike some other IS algorithms) on datasets with a large number of instances.

Firstly, noise is filtered in the data by applying Wilson editing , which means removing elements misclassified using K nearest neighbors algorithm. Subsequently, the dataset is divided into disjoint subsets. Elements are assigned to these subsets according to their closest enemies, i.e., in each subset,

there are only those elements that have a common nearest enemy. If an element has multiple nearest enemies, only one of them is randomly selected. After dividing the dataset into disjoint subsets, a filter rule is applied to each subset that is greater than the value of the chosen parameter m. This rule removes an element y if it finds an element x different from y such that the distance between y and the nearest enemy ne (ne is the same for all elements of the given subset) is greater than or equal to the maximum of the distances between x and y and between x and ne. The procedure of the PIF algorithm described above is summarized in pseudocode 9.

Algorithm 9 PIF

Let:
T be the original dataset
T_{new} be the reduced dataset
NE be the set of elements that are the nearest enemies for at least one of the other elements in T_{new}
K be the parameter for the Wilson editing algorithm
m be the parameter indicating the minimum subset size
$d(x,y)$ be the distance (e.g., Euclidean distance) between elements x and y

$T_{new} \leftarrow T$
$T_{new} \leftarrow WilsonEditing(T_{new}, K)$
for each $x \in T_{new}$ **do**
    find the nearest enemy ne_x
    add x to the subset S_{ne_x}
end for
for each $ne \in NE$ **do**
    if $|S_{ne}| \geq m$ **then**
        for each $y \in S_{ne}$ **do**
            for each $x \in S_{ne}$ where $x \neq y$ **do**
                if $d(y,ne) \geq \max\{d(x,y), d(x,ne)\}$ **then**
                $T_{new} \leftarrow T_{new} \setminus \{y\}$
                continue to the next y
                end if
            end for
        end for
    end if
end for
return T_{new}

2.3.3 Iterative Case Filtering

The next state-of-the-art hybrid algorithm is Iterative Case Filtering (ICF) [43]. It is a batch method, meaning that instances satisfying the reduction condition are only flagged during the traversal, and their removal occurs all at once.

The ICF algorithm uses the terms $LocalSet(x)$, $Coverage(x)$, and $Reachable(x)$. The following is an explanation of these terms:

- **LocalSet(x)** - It is the set of all nearest neighbors of element x that belong to the same class as x and have a smaller distance to x than its nearest enemy. This set is also referred to as the Relative Neighborhood of x in the MSS algorithm (see 2.1.2).
- **Coverage(x)** - It is the set of all elements y for which element x belongs to their LocalSet(y). A similar concept is used in the DROP3 algorithm (see 2.3.1), which refers to such a set as associates(x).
- **Reachable(x)** - It is the set of all elements y that belong to the LocalSet of x.

The execution of the algorithm can be divided into two parts. At the beginning, the dataset T_{new} (in the beginning $T_{new} = T$) is denoised using Wilson editing (as in the PIF and DROP3 algorithms). Wilson's editing is described in section 2.2.1. In the second part, the sets $Reachable(x)$ and $Coverage(x)$ are created for all elements x from T_{new}. Subsequently, those elements x whose $Reachable(x)$ set is larger than the $Coverage(x)$ set are flagged. After traversing the entire set T_{new}, all marked elements are removed at once. The second part is repeated until at least one element meets the condition for removal. After completion, the reduced dataset T_{new} is returned. Pseudocode 10 summarizes the steps of the ICF algorithm.

Algorithm 10 ICF

Let:
T be the original dataset
T_{new} be the reduced dataset
K be the parameter for Wilson editing algorithm
$Coverage(x)$ see description above
$Reachable(x)$ see description above
removed be the variable indicating whether at least one instance was removed during the traversal of the dataset

1: $T_{new} \leftarrow T$
2: $T_{new} \leftarrow WilsonEditing(T_{new}, K)$
3: **for each** $x \in T_{new}$ **do**
4: compute $Coverage(x)$
5: compute $Reachable(x)$
6: **end for**
7: $removed \leftarrow false$
8: **for each** $x \in T_{new}$ **do**
9: **if** $|Reachable(x)| > |Coverage(x)|$ **then**
10: flag x for removal
11: set $removed \leftarrow true$
12: **end if**
13: **end for**
14: **for each** $x \in T_{new}$ **do**
15: **if** x is flagged **then**
16: $T_{new} \leftarrow T_{new} \setminus \{x\}$
17: **end if**
18: **end for**
19: **if** $removed = true$ **then**
20: go back to step 3
21: **end if**
22: return T_{new}

Chapter 3

Classification algorithms

Classification tasks can be solved using both supervised and unsupervised machine learning algorithms. The goal of classification algorithms is to determine the class to which a classified instance belongs. Classification is used in various fields, such as medicine, security, or the automotive industry.

The dataset is commonly divided before training classification algorithms. Most commonly, it is split into three parts: the **training**, **validation**, and **test** sets. The classification algorithm is trained using the training set. The validation set is used for tuning the hyperparameters of the chosen algorithm, and the test set is used for the final evaluation of the resulting model.

Since, in this thesis, a K nearest neighbors classifier was used for evaluating experiments during data preprocessing and for comparing instance selection algorithms, its description is provided in Section 3.1. Section 3.2 describes the metrics used for the evaluation of the experimental results.

3.1 K Nearest Neighbors (KNN)

K Nearest Neighbors [44] is considered as one of the most straightforward machine learning algorithms used for classification. A new instance is classified with the help of *K* nearest neighbors, where *K* is a chosen parameter. The appropriate value of the *K* parameter is usually determined based on experiments. Firstly, the distances between the classified instance and all instances from the training set are calculated. The equation (3.1) contains the formula for calculating the *Minkowski* distance.

$$d_{Mink}(x,y) = (\sum_{i=1}^{N} |x_i - y_i|^p)^{\frac{1}{p}} \tag{3.1}$$

where:

- N is the dimension of the feature vector
- x_i is the value of the i-th feature of the vector x

- y_i is the value of the i-th feature of the vector y
- p is a parameter

Very commonly used metrics in KNN are the *Euclidean* distance and the *Manhattan* distance [45]. Both distances are special cases of the Minkowski distance, with parameter values of $p = 2$ for the Euclidean distance and $p = 1$ for the Manhattan distance. After the distance calculation, the instances (neighbors) are sorted in ascending order based on their distances. The classified instance is then assigned the majority class among the K nearest neighbors. The described procedure is summarized in pseudocode 11.

Algorithm 11 KNN Algorithm

Let:
T be the training set
K be a parameter indicating the number of nearest neighbors used for classification
x be a classified instance
$y \in T$ be an instance from the training set

1: Calculate the distances between x and all $y \in T$ (3.1)
2: Sort T in ascending order by the distance of instances $y \in T$ to the classified instance x
3: Select K first instances as nearest neighbors
4: Assign the instance x a class according to the majority class of the K nearest neighbors

3.2 Classification metrics

After creating a classification model, its performance is evaluated on a test set. The evaluation of the model is done using one or more metrics commonly used for classification problems. During the binary classification of individual instances, four situations can occur [46]:

- **True Positive (TP)** - This situation occurs when the classifier correctly labels an instance as positive (malware is correctly classified as malware).
- **False Positive (FP)** - This situation occurs when the classifier incorrectly labels an instance as positive (a clean file is incorrectly classified as malware). This is a Type I error.
- **True Negative (TN)** - This situation occurs when the classifier correctly labels an instance as negative (a clean file is correctly classified as clean).
- **False Negative (FN)** - This situation occurs when the classifier incorrectly labels an instance as negative (malware is incorrectly classified as a clean file). This is a Type II error.

Many metrics utilize the information about the situations described above for evaluation. Below are described the metrics used for evaluating the experimental results in this thesis [47]:

- **Accuracy** - It is the ratio of the number of correctly classified instances to the total number of instances in the test set. The equation (3.2) shows the formula for calculating accuracy.

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN} \tag{3.2}$$

- **Precision** - It is the ratio of the number of TP (True Positive) classifications to the total number of instances classified as Positive. The formula for calculating precision is provided in the equation (3.3).

$$Precision = \frac{\#TP}{\#TP + \#FP} \tag{3.3}$$

- **Recall (True Positive Rate - TPR)** - The recall metric is used to evaluate how well the model is able to correctly classify positive instances (in our case, malware). It is the ratio of the number of TP (True Positive) classifications to the total number of instances that are actually positive (malware). Equation (3.4) shows the formula for calculating recall.

$$Recall = \frac{\#TP}{\#TP + \#FN} \tag{3.4}$$

- **F1 Score** - To obtain the F1 score, it is necessary to calculate the harmonic mean of precision and recall. Equation (3.5) shows the formula for calculating the F1 score.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.5}$$

Chapter 4

Structure of Portable Executable file format

Since this thesis works with datasets containing metadata of Portable Executable (PE) files, this chapter describes the structure of this format. The PE format is utilized by executable files for the Windows operating system, and it is based on the Common Object File Format (COFF) files [48]. Examples of files having the PE format include those with the extensions EXE and DLL [49]. Information contained in the PE file header is crucial for the successful loading into the computer's memory and subsequent code execution. The structure of a PE file is depicted in Figure 4.1 [50]. The following sections describe the main parts of the PE file. Section 4.1 provides a description of the DOS header. Section 4.2 details the structure of the DOS Stub. The structure of the NT Headers, which includes the File Header and Optional Header, is described in Section 4.3. Information about the Section table is in Section 4.4, and a description of the Sections in a PE file is in Section 4.5.

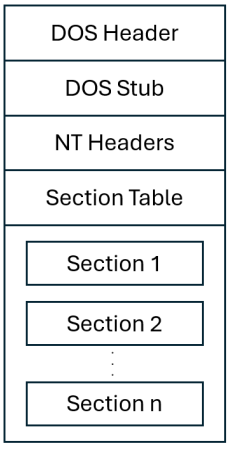

Figure 4.1 The structure of a PE file

4.1 DOS Header

The first structure is the DOS Header, consisting of the initial 64 bytes of the PE file [51]. Nowadays, this structure does not significantly impact the functionality of executable files. However, it remains a part of the file for backward compatibility with MS-DOS executable files [52]. The DOS Header structure contains a total of 19 fields, but from the perspective of Windows executable files, the following two are noteworthy:

- **e_magic** - This is the first field of the DOS Header structure. The ASCII value of this field is **MZ**, which are the initials of one of the MS-DOS developers. This field is often used for the initial identification of a PE file.
- **e_lfanew** - This is the last field of the DOS Header structure, containing the offset to the start of the File Header (or COFF Header), which falls under NT Headers (see description below). From the perspective of Windows executable files, this is the most important field in the DOS header.

Thanks to the ASCII string 'MZ' contained in the e_magic field, this structure is also referred to as the MZ Header.

4.2 DOS Stub

After the DOS Header, the DOS Stub structure follows, which is also part of PE files for backward compatibility with MS-DOS executable files. During the linking process of the executable file, a message is placed here by the linker. This message is printed if the file is executed in an MS-DOS environment. The default displayed value is the message 'This program cannot be run in DOS mode'.

4.3 NT Headers

NT Headers is a structure that consists of three fields [53]. The first of them is the **PE Signature**. This 4-byte field contains the constant ASCII value "PE\0\0," which is used for identifying PE files [54]. The remaining two attributes, which are the File and Optional headers, are described in sections 4.3.1 and 4.3.2. Figure 4.2 contains a schema of the NT Headers structure.

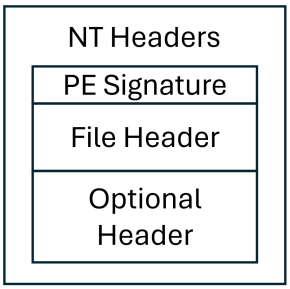

Figure 4.2 The structure of the NT header

4.3.1 File Header

After the PE Signature, the **File Header** structure follows. It is a 20-byte structure that contains this fundamental information about the PE file [55]:

- **Machine** - It is a value that uniquely identifies the target CPU type. Examples include IMAGE_FILE_MACHINE_AMD64 or IMAGE_FILE_MACHINE_I386.
- **NumberOfSections** - The number of sections in the file.
- **TimeDateStamp** - It is the creation date, represented as the number of seconds since January 1, 1970.
- **PointerToSymbolTable** - It is a pointer pointing to the COFF symbol table. Since COFF debugging information is currently deprecated, this value is often set to zero.
- **NumberOfSymbols** - This value determines the number of symbols in the COFF symbol table. Similar to PointerToSymbolTable, this value is often set to zero (for the same reason).
- **SizeOfOptionalHeader** - The size of the Optional Header structure.
- **Characteristics** - These are flags indicating attributes of the PE file. From the perspective of malware detection, this is an important element [56]. Attributes, for example, include information about whether it is a dynamically linked library or whether the architecture of the target processor is 32-bit.

4.3.2 Optional Header

The **Optional Header** is a structure whose size is not fixed. Despite the name suggesting it is optional, executable files must include it. The information contained in this header is crucial for the PE loader, which performs the loading and execution of the executable file based on this information. The content can be divided into three main parts:

- **Standard Fields**
- **Windows-Specific Fields**
- **Data Directories**

More detailed information about the aforementioned parts is provided in Sections 4.3.2.1, 4.3.2.2, and 4.3.2.3.

4.3.2.1 Standard Fields

After the File Header, the Standard Fields follow. The first eight fields of this Optional Header part are identical for both PE32+ and PE32 files. The PE32 format additionally includes the ninth field, **BaseOfData**. The description of all nine fields follows:

- **Magic** - The magic number can have one of three possible values. The value 0x10b is used to denote the 32-bit PE32 format. The 64-bit PE32+ format is represented in the Magic field by the value 0x20b. The last possible value is 0x107, which represents ROM image files.
- **MajorLinkerVersion** - This field contains the major version of the linker.
- **MinorLinkerVersion** - This field contains the minor version of the linker.
- **SizeOfCode** - This field contains the size of the **.text** section or the sum of the sizes of all **.text** sections if the PE file contains multiple ones.
- **SizeOfInitializedData** - This field contains the size of the **.data** section or the sum of the sizes of all **.data** sections if the PE file contains multiple ones.
- **SizeOfUninitializedData** - The SizeOfUninitializedData field contains the size of the **.bss** section or the sum of the sizes of all **.bss** sections if the PE file contains multiple ones.
- **AddressOfEntryPoint** - It is a Relative Virtual Address (RVA) pointing to the entry point of the program. This RVA contains the relative distance from the image base and changes depending on the location in the memory. If the entry point is missing (not mandatory for DLLs), the field must be set to zero.
- **BaseOfCode** - BaseOfCode is an RVA pointing to the beginning of the **.text** section. Its value is obtained after loading the file into the memory.
- **BaseOfData (only for PE32)** - This is an RVA pointing to the beginning of the .data section. The RVA is obtained after loading the file into the memory.

4.3.2.2 Windows-Specific Fields

After the Standard Fields, there are 21 fields that constitute an extension of the original COFF Optional Header. The information contained in these fields is crucial for the loader and linker in executable files running under the Windows operating system. These fields include:

- **ImageBase** - This field represents the preferred address of the first byte of the executable file, i.e., the address to which the file should be loaded. It must be a multiple of 64K. In most cases, this field is ignored by the PE loader, and an address from the free part of the memory is chosen.
- **SectionAlignment** - SectionAlignment determines the alignment of individual sections. The value (in bytes) of this field must be greater than or equal to the value of the FileAlignment field.
- **FileAlignment** - This field represents the alignment of raw data sections in the image file on disk. Similar to SectionAlignment, the value, specified in bytes, should range between 512 and 64K and should be a power of two.
- **MajorOperatingSystemVersion** - This field contains the major version number of the required operating system.

- **MinorOperatingSystemVersion** - This field contains the minor version number of the required operating system.
- **MajorImageVersion** - This field contains the major version number of the image.
- **MinorImageVersion** - This field contains the minor version number of the image.
- **MajorSubsystemVersion** - This field contains the major version number of the subsystem.
- **MinorSubsystemVersion** - This field contains the minor version number of the subsystem.
- **Win32VersionValue** - Reserved field; its value must be zero.
- **SizeOfImage** - This field contains information about the size of the image in bytes after loading into memory. This value includes all file headers and must be a multiple of Section Alignment.
- **SizeOfHeaders** - It represents the sum of the sizes of the DOS Stub, NT Headers, and section headers. If it is not a multiple of FileAlignment, it is rounded to the nearest such value.
- **CheckSum** - Checksum of the file.
- **Subsystem** - Here, the required subsystem for the successful execution of the PE file is specified. Individual subsystems are represented by values between 0 and 16. For example, IMAGE_SUBSYSTEM _WINDOWS_GUI is a constant for the Windows graphical user interface subsystem, represented by the decimal value 2.
- **DllCharacteristics** - This field contains information about the PE file in the form of flags.
- **SizeOfStackReserve** - This field contains information about the size of the stack to reserve.
- **SizeOfStackCommit** - This field contains information about the size of the stack to commit.
- **SizeOfHeapReserve** - This field contains information about the size of heap space to reserve.
- **SizeOfHeapCommit** - This field contains information about the size of heap space to commit.
- **LoaderFlags** - Reserved field; its value must be zero.
- **NumberOfRvaAndSizes** - This field contains information about the number of entries in the Data Directories section.

4.3.2.3 Data Directories

Data Directories, which immediately follow the Windows-Specific Fields, are the last element of the Optional Header. It is a structure containing up to (the count is not fixed) sixteen fields:

- **VirtualAddress** - It contains the RVA pointing to the beginning of the Data Directory entry.
- **Size** - It contains the size of the Data Directory entry.

Currently, two Data Directory entries are reserved, and therefore, the values of their fields must be set to zero, same as all missing Data Directories. The remaining fourteen entries are described below:

- **Export Table** - Contains the RVA and size of the export table. This table is located in the **.edata** section.
- **Import Table** - Contains the RVA and size of the import table. This table is located in the **.idata** section.
- **Resource Table** - Contains the RVA and size of the resource table. This table is located in the **.rsrc** section.
- **Exception Table** - Contains the RVA and size of the exception table. This table is located in the **.pdata** section.
- **Certificate Table** - Contains the RVA and size of the certificate table.
- **Base Relocation Table** - Contains the RVA and size of the base relocation table. This table is located in the **.reloc** section.
- **Debug** - Contains the RVA and size of debug data.
- **Global Ptr** - RVA pointing to the value of the global pointer register. The Size field is set to zero in this case.
- **TLS Table** - Contains the RVA and size of the Thread Local Storage (TLS) table.
- **Load Config Table** - Contains the RVA and size of the load configuration table.
- **Bound Import** - Contains the RVA and size of the bound import table.
- **IAT** - Contains the RVA and size of the import address table.
- **Delay Import Descriptor** - Contains the RVA and size of the delay import descriptor.
- **CLR Runtime Header** - Contains the RVA and size of the CLR runtime header.

4.4 Section Table

The Section Table follows directly in the structure of the PE file after the NT Headers. This must be adhered to because there is no field pointing to this structure. It is a table whose entries are section headers. The structure of each of the headers is as follows:

- **Name** - This 8-byte field contains the name of the section. The name is represented as a UTF-8 string, and if its size is less than 8 bytes, it is padded with null bytes. In the case of executable files, strings longer than 8 bytes are not used.

- **VirtualSize** - It represents the size of the section after loading into the memory. This value may be larger than the value in the SizeOfRawData field. This situation occurs when the section is zero-padded.
- **VirtualAddress** - This field contains the RVA pointing to the first byte of the section. The RVA is obtained after loading the executable file into memory.
- **SizeOfRawData** - For executable files, this is the size of initialized data on disk. This value must be a multiple of the FileAlignment value from the Optional Header and also less than or equal to VirtualSize.
- **PointerToRawData** - For executable files, this value, which is a multiple of the FileAlignment value, is the file pointer to the beginning of the section within the executable file.
- **PointerToRelocations** - This field contains a pointer to the beginning of relocation entries for the section.
- **PointerToLineNumbers** - This is a pointer to the beginning of line-entries records. For executable files, this value is set to zero because these are deprecated records.
- **NumberOfRelocations** - The value of this field indicates the number of relocation entries. PE files do not have relocation entries, so this value is set to zero.
- **NumberOfLinenumbers** - It is a value indicating the number of line-number entries, and for PE files, it is set to zero.
- **Characteristics** - This field of the section header contains flags characterizing the section.

4.5 Sections

The sections are located in the file structure after the section table and contain the actual data of the PE file. Each file has at least two sections, one for code and the other for data [57]. There are many types of sections, and the most common ones are described below.

- **.text** - This section contains executable code that is executed upon launching the PE file.
- **.data** - In this section, initialized data is stored.
- **.rdata** - This section contains data intended for read-only access.
- **.bss** - It holds uninitialized data.
- **.edata** - Export tables are stored in this section.
- **.idata** - The .idata section contains import tables.
- **.reloc** - It contains relocation information for the PE file.

- **.rsrc** - All resources used by the executable file are stored in the .rsrc section. Examples include icons.
- **.tls** - The Thread Local Storage section provides storage for the currently executing threads of the program.

Chapter 5

Preprocessing of Datasets Before Applying IS Algorithms

In this chapter, the data preprocessing methods applied before experiments with instance selection algorithms are described. Section 5.1 specifies the parameters of the computing stations used during preprocessing. An overview of the selected datasets is provided in Section 5.2. Section 5.3 describes the process of preprocessing the EMBER and SOREL-20M datasets, including the applied methods and experimental results.

5.1 Used hardware devices

Experiments and computations were conducted on two computing stations. All experiments with the EMBER dataset took place on the NVIDIA DGX Station. The specifications of the NVIDIA DGX Station are described in Table 5.1.

NVIDIA DGX Station A100 Version 5.4.2	
Processor	AMD EPYC 7742 64-Core Processor 2.25 GHz
Memory	512 GB
Operating system	Ubuntu 20.04.5 LTS

Table 5.1 Specifications of the NVIDIA DGX Station

Due to its size, the SOREL-20M dataset was partially processed on the GPU2 computing station, which has a larger memory. Specifically, this involved parsing the dataset into CSV files and data cleaning. The specifications of the GPU2 computing station are provided in Table 5.2. The remaining preprocessing of the SOREL-20M dataset was performed on the NVIDIA DGX Station.

GPU2 Station	
Processor	2x Intel(R) Xeon(R) Gold 6136, 3.00GHz, 12 cores
Memory	755 GB
Operating system	Ubuntu 20.04.5 LTS

Table 5.2 Specification of parameters for GPU2 computing station

5.2 Information about the Chosen Datasets

For the experiments in this master's thesis, two datasets containing features extracted from malicious and benign PE files were selected. Section 5.2.1 provides an overview of the EMBER dataset, while Section 5.2.2 contains information about the SOREL-20M dataset.

5.2.1 EMBER

EMBER [6] is a dataset designed for static detection of Windows portable executable files. In this thesis, experiments were conducted using version 2 from 2018, which contains one million instances. For the experiments, 800k instances were used from the original dataset, the remaining 200k were unlabeled and thus were not suitable for experiments with supervised algorithms. The dataset containing instances represented as JSON objects was transformed into two CSV files (feature and label separation) using the Python script **ember_json_parsing.py**. Feature hashing (see 5.3) was applied during parsing on parts with variable sizes and some structures. Each JSON record contained the following thirteen main elements, and some of them contained additional substructures:

- **sha256 (string)** - Contains the SHA-256 hash of the file.
- **md5 (string)** - Contains the MD5 hash of the file.
- **avclass (string)** - Contains the malware type.
- **appeared (date)** - Contains the month and year of the first appearance.
- **label (number)** - Contains one of three possible values (malicious = 1, benign = 0, unlabeled = -1).
- **general (JSON)** - Contains general metadata, such as file size, the number of imported functions, or information about whether the file is digitally signed.
- **header (JSON)** - Contains information obtained from File and Optional Headers.
- **imports (JSON)** - For each imported library, a list of imported functions is provided.
- **exports (list)** - Contains a list of exported symbols.
- **section (JSON)** - Contains information obtained from section headers.
- **histogram (list)** - Contains information about the frequency of individual bytes.

- **byteentropy (list)** - A list containing the entropy of individual bytes.
- **strings (JSON)** - Contains information about strings, including a histogram of printable characters.

An example structure of one of the original JSON records is provided in **ember_example_structure.txt** due to its size.

5.2.2 SOREL-20M

The second chosen large-scale dataset is SOREL-20M [7]. SOREL-20M was created from nearly twenty million instances containing pre-extracted features and metadata of Windows Portable Executable files. The data were parsed from two databases: the SQLite3 database **meta.db** and the LDBM **pe_metadata** database, composed of files **lock.mdb** and **data.mdb**. The SQLite3 database contained information about labels, tags, detection counts, and first/last occurrences related to SHA-256 file hashes. These hashes served as keys for accessing the LDBM database, which contained PE metadata extracted using the Python module pefile [58]. Each record consisted of 12 main parts, some of which were composed of additional substructures:

- **DOS_HEADER (JSON)** - The fields of this structure is identical to the fields of the DOS header.
- **NT_HEADERS (JSON)** - Contains the PE signature.
- **FILE_HEADER (JSON)** - Contains information obtained from the File Header.
- **Flags (list)** - A list of executable file attributes corresponding to the Characteristics field in the File Header.
- **OPTIONAL_HEADER (JSON)** - This part contains information obtained from the Optional Header of the PE file.
- **DllCharacteristics (list)** - A list of executable file attributes obtained from the Optional Header.
- **PE Sections (list)** - A list of structures of section headers containing information about individual sections.
- **Directories (list)** - A list of Data Directory entries.
- **Version Information (list)** - Contains general information about the executable file.
- **Imported symbols (list)** - Contains a list of imported libraries and functions.
- **Resource directory (list)** - Contains information about resources used in the PE file, such as icons, bitmaps, or strings.
- **Base relocations (list)** - Contains information about address relocations in memory.

An example of the original JSON record is provided in the file **sorel_example_structure.txt**, due to its size. Parsing was performed using the script **sorel_json_parsing.py**.

5.3 Preprocessing procedure

As feature hashing was used during parsing, experiments with the number of transformed feature bins were conducted first. The impact of the created bins on classification accuracy was evaluated separately for each structure. This means that while experimenting with the number of bins for one structure, the bin counts for other structures were fixed. Only the training and validation sets of the EMBER dataset were used for experiments, and the resulting bin counts were also applied to the SOREL-20M dataset (identical transformed features). Feature hashing was applied to these structures:

- **Characteristics** from File Header
- **DllCharacteristics** from Optional Header
- From the structure **IMAGE_SECTION_HEADER**:
  - List of **Name** fields
  - List of **VirtualSize** fields
  - List of **SizeOfRawData** fields
  - List of **Characteristics** fields
  - List of **Entropy** fields
- List of imported libraries
- List of ordered pairs (imported library, imported function)
- List of exported functions
- List of **Entry** fields (only for EMBER)

Table 5.3 contains the counts of bins experimented with during feature hashing for individual sections.

Section	Bin counts
Pairs library-function	0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1750, 2000
Library names	0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 200, 300, 400, 500, 600, 700
Other	0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120

Table 5.3 Tested numbers of bins for individual structures

For evaluation, a KNN classifier with the parameter K=3 was used. Before evaluation, the following methods were applied to all versions: **imputation of missing values with a constant**, **removal of constant features**, **removal of features with unique values**, **removal of duplicate rows**, **one-hot encoding** (features ”machine,” ”subsystem,” and ”magic”), and **standardization**. More detailed information about each method is provided below.

As part of the experiments, a comparison was also made between a version of the dataset where feature scaling was applied to all features and a version where feature scaling was omitted for features created

using one-hot encoding and feature hashing (hereinafter referred to as the dataset with **"non-dummy" features scaled**). For both EMBER and SOREL-20M, two versions were created for further experimentation. In Table 5.4, selected bin counts for individual structures and dataset versions are provided. The values were selected based on the classification accuracy of the validation set. Complete results have been included in the appendices.

Section	All features scaled		"Non-dummy" features scaled	
	Num of bins	Accuracy	Num of bins	Accuracy
Characteristics	50	0.9521	10	0.9471
DllCharacteristics	40	0.9519	20	0.9471
Name	10	0.9525	0	0.9499
Misc	10	0.9525	0	0.9499
SizeOfRawData	10	0.9525	0	0.9499
Flags	10	0.9525	0	0.9499
Entropy	10	0.9525	0	0.9499
Imported libraries	110	0.9525	50	0.9472
Pairs (library-function)	900	0.9521	1300	0.9476
Exports	0	0.9524	0	0.9482
Entry	10	0.9519	0	0.9472

Table 5.4 Selected bin counts for individual structures

The parsing of both datasets was thus carried out with the above-mentioned counts. Further preprocessing of datasets before applying IS algorithms is described in sections 5.3.1 and 5.3.2.

5.3.1 EMBER

Both created versions of the EMBER dataset were divided into training, validation, and test sets in a ratio of 60:20:20. The sizes of the individual sets are in Table 5.5. In sections 5.3.1.1, 5.3.1.2, 5.3.1.3, 5.3.1.4, 5.3.1.5, 5.3.1.6, 5.3.1.7, 5.3.1.8, 5.3.1.9, and 5.3.1.10, methods of data preprocessing and experimental evaluation are described in the order in which they were applied to the dataset.

Set	All features scaled		"Non-dummy" features scaled	
	Num of instances	Num of features	Num of instances	Num of features
Train	480000	1834	480000	2054
Validation	120000	1834	120000	2054
Test	120000	1834	120000	2054

Table 5.5 Sizes of the created sets of the EMBER dataset

5.3.1.1 Missing values imputation

In the EMBER dataset, missing values were present in the following features:

- All features **VirtualAddress** and **Size** in the Data Directory structures
- Categorical features **Machine** and **Subsystem**

In all cases, a constant value was used to impute missing values. Missing values represented by NaN for the features VirtualAddress and Size were replaced with zeros. For categorical features **Machine** and **Subsystem**, where missing values were represented by the string **"???"**, these values were replaced with the string **"UNKNOWN,"** which, according to [50], is a valid category for both mentioned features.

5.3.1.2 Removal of constant features

Constant features were subsequently removed from both versions of the EMBER dataset, i.e., features with the same value for all instances in the training set. In most cases, these were features created using feature hashing. Table 5.6 contains the original and new counts of features for both versions.

Dataset version	Num of features before	Num of features after
All features scaled	1834	1758
"Non-dummy" features scaled	2054	2042

Table 5.6 Changes after the removal of constant features in the EMBER dataset

5.3.1.3 Removal of features with unique values

Subsequently, the features whose value was unique for each instance of the training set were removed. This condition applied to the **sha256** and **md5** features, which were removed from the dataset. The original and new counts after removing these features are in Table 5.7.

Dataset version	Num of features before	Num of features after
All features scaled	1758	1756
"Non-dummy" features scaled	2042	2040

Table 5.7 Changes after removing unique features from the EMBER dataset

5.3.1.4 Removal of duplicate instances

Duplicate instances were removed from all three sets, i.e., training, validation, and test. In all three sets, the number of duplicate instances was in the order of units. Changes in the number of instances for each set and version of the EMBER dataset are summarized in Table 5.8.

Set	All features scaled		”Non-dummy” features scaled	
	Before	After	Before	After
Train	480000	479952	480000	479952
Validation	120000	159993	120000	159993
Test	120000	159997	120000	159997

Table 5.8 Changes after removing duplicate instances from the EMBER dataset

5.3.1.5 Removal of inconsistencies

In this thesis, the term ’inconsistencies’ refers to two or more instances with the same feature vector but different labels. No inconsistencies were found in the EMBER dataset.

5.3.1.6 Conversion of categorical features to numeric

Since the KNN classifier and the employed IS algorithms work with numerical vectors, the following operations were applied to the remaining categorical features:

- The **appeared** feature was converted from the date format to the datetime/10^9 format.
- One-hot encoding was applied to the features **machine**, **subsystem**, and **magic**.

The changes in the number of features are recorded in Table 5.9.

Dataset version	Num of features before	Num of features after
All features scaled	1756	1778
”Non-dummy” features scaled	2040	2062

Table 5.9 Changes after applying one-hot encoding to the EMBER dataset

5.3.1.7 Outlier handling

The IQR method was used for detecting outliers. Each of the two versions was then divided into three additional sub-versions:

- Version without replacing outliers
- Version with outliers replaced by the median
- Version with outliers replaced by the mean

Thus, six new versions were created from the two original versions of the EMBER dataset (see Figure 5.1). These newly created versions were processed separately, and their evaluation took place after applying feature scaling.

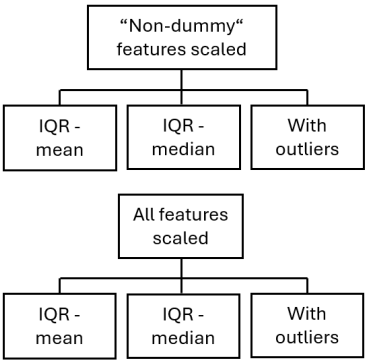

Figure 5.1 Diagram of the distribution of datasets within outlier handling

5.3.1.8 Feature scaling

In this thesis were conducted experiments with the following feature scaling methods:

- Min-max normalization
- Standardization
- Robust scaling

Each of the above-mentioned methods was applied to all versions created so far. Thus, the six original versions of the EMBER dataset were transformed into eighteen new versions. The diagram in Figure 5.2 summarizes the gradual creation of versions of the EMBER dataset during preprocessing.

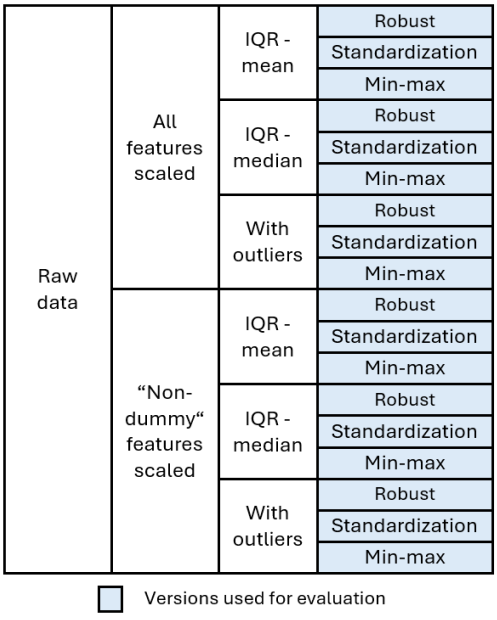

Figure 5.2 EMBER dataset partitioning scheme

5.3.1.9 Evaluation before feature extraction

A KNN classifier with the parameter **K**=3 was used to evaluate all eighteen variants on the validation set of the EMBER dataset. The highest accuracy was achieved when applying min-max normalization to all features without replacing outliers (labeled as **Scaled all** in the following table). Among the datasets without applied feature scaling to dummy features, the best accuracy was achieved by combining standardization and replacing outliers with the mean (labeled as **Not scaled all** in the following table). These two versions were used for further experiments with feature extraction. Information about the accuracy of selected versions is in Table 5.10. Complete results are included in the appendices.

Metric	Scaled all	Not scaled all
Accuracy	0.9554	0.9513
F1 score	0.9555	0.9514

Table 5.10 Information about selected versions of the EMBER dataset before feature extraction

5.3.1.10 Feature extraction

The PCA method was used for feature extraction from both datasets selected in 5.3.1.9. The number of tested features obtained by the PCA method ranged from 1 to 120. The transformed version of the 'Scaled all' dataset with 119 features achieved the highest accuracy on the validation set and was used for the following experiments with IS algorithms. Information about the chosen dataset is in Table 5.11.

Set sizes			Number of features	Accuracy	F1
Train	Validation	Test			
479952	159993	159997	119	0.9544	0.9545

Table 5.11 Information about the EMBER dataset after preprocessing

5.3.2 SOREL-20M

Based on the experiments described at the beginning of Section 5.3, two versions of the SOREL-20M dataset with different numbers of features created using feature hashing were parsed. Like the EMBER dataset, the dataset was divided into training, validation, and test sets in a ratio of 60:20:20. The numerical representation of the created sets is in Table 5.12. Sections 5.3.2.1, 5.3.2.2, 5.3.2.3, 5.3.2.4, 5.3.2.5, 5.3.2.6, 5.3.2.7, 5.3.2.8, 5.3.2.9, and 5.3.2.10 describe the applied preprocessing methods and the results of the experimental evaluation in the order in which they were applied to the SOREL-20M dataset.

Set	All features scaled		”Non-dummy” features scaled	
	Num of instances	Num of features	Num of instances	Num of features
Train	11626773	1242	11626773	1472
Validation	3875591	1242	3875591	1472
Test	3875592	1242	3875592	1472

Table 5.12 The sizes of the created sets of the SOREL-20M dataset

5.3.2.1 Missing values imputation

The SOREL-20M dataset contained missing values for the same features as the EMBER dataset. For this reason, the procedure for completing the missing values was identical to the procedure given in the 5.3.1.1 section.

5.3.2.2 Removal of constant features

In both versions of the SOREL-20M dataset created so far, constant features were found and subsequently removed. In most cases, these were features created using feature hashing. The changes after the removal of constant features are in Table 5.13.

Dataset version	Num of features before	Num of features after
All features scaled	1242	1062
”Non-dummy” features scaled	1472	1405

Table 5.13 Changes after the removal of constant features from the SOREL-20M dataset

5.3.2.3 Removal of features with unique values

Neither version contained features for which each instance in the training set had a unique value.

5.3.2.4 Removal of duplicate instances

The removal of duplicates, as with EMBER, took place for the training, validation, and test sets. The number of deleted instances for the SOREL-20M dataset was in the order of millions. Changes in the number of instances according to individual sets and versions of the SOREL-20M dataset are summarized in table 5.14.

Set	All features scaled		”Non-dummy” features scaled	
	Before	After	Before	After
Train	11626773	6931041	11626773	6438086
Validation	3875591	2390916	3875591	2223164
Test	3875592	2389201	3875592	2222007

Table 5.14 Changes after removing duplicate instances from the SOREL-20M dataset

5.3.2.5 Removal of inconsistencies

In the case of the SOREL-20M dataset, inconsistencies were found. The size of both versions decreased by thousands of instances after removal. The exact changes in the number of instances for individual versions and their sets are recorded in Table 5.15.

Set	All features scaled		”Non-dummy” features scaled	
	Before	After	Before	After
Train	6931041	6926181	6438086	6430468
Validation	2390916	2388994	2223164	2220534
Test	2389201	2387255	2222007	2219577

Table 5.15 Changes after removing inconsistencies from the SOREL-20M dataset

5.3.2.6 Conversion of categorical features to numeric

One-hot encoding was applied to the remaining categorical features of both versions that were not transformed using feature hashing (see the beginning of Section 5.3). These features include **Machine**, **Subsystem**, and **Magic**. Table 5.16 summarizes the changes in the number of features after applying one-hot encoding.

Dataset version	Num of features before	Num of features after
All features scaled	1062	1088
”Non-dummy” features scaled	1405	1431

Table 5.16 Changes after applying one-hot encoding to the SOREL-20M dataset

5.3.2.7 Outlier handling

The transformation of the original two versions of the SOREL-20M dataset into six new versions followed the procedure used for the EMBER dataset. More details about the methods used can be found in Section 5.3.1.7.

5.3.2.8 Feature scaling

The creation of eighteen new versions from the original six versions was carried out based on the same procedure used for the EMBER dataset. A description of the methods used and the newly created versions is provided in Section 5.3.1.8.

5.3.2.9 Evaluation before feature extraction

The eighteen created versions of the SOREL-20M dataset were evaluated using a KNN classifier with the parameter **K**=3. Due to the imbalanced representation of classified classes in the datasets, besides

accuracy, the F1 score was also taken into account. The version with standardized 'non-dummy' features and outliers replaced by the mean (labeled as **Not scaled all** in the following table) achieved the highest accuracy. In the achieved F1 score, this version ranked second. Among the versions with feature scaling applied to all features, the combination of robust scaling with outliers replaced by the arithmetic mean (labeled as **Scaled all** in the following table) achieved the highest accuracy and F1 score (among all versions). Experiments with feature extraction were conducted on both mentioned dataset variants. The achieved F1 scores and accuracies of selected versions are presented in Table 5.17. The complete results of the experiments have been included in the appendices.

Metric	Scaled all	Not scaled all
Accuracy	0.9765	0.9770
F1 score	0.9595	0.9562

Table 5.17 Information about selected versions of the SOREL-20M dataset before feature extraction

5.3.2.10 Feature extraction

The PCA method was also used for feature extraction from the two selected versions of the SOREL-20M dataset (see 5.3.2.9). The extracted features were tested within the range of 1 to 120 during the experiments. The 'Scaled all' version with 106 extracted features has been chosen for experiments with IS algorithms. A description of the selected version is provided in Table 5.11.

Set sizes			Number of features	Accuracy	F1
Train	Validation	Test			
6926181	2388994	2387255	106	0.9761	0.9587

Table 5.18 Information about the SOREL-20M dataset after preprocessing

Chapter 6

Proposed modifications of the PIF algorithm

This chapter describes proposed modifications of the PIF algorithm, which were applied and experimentally evaluated alongside other state-of-the-art IS algorithms. Combining two proposed modifications resulted in a total of five modified versions, described below. Section 6.1 contains a description of versions of the PIF algorithm with replaced editing algorithms. The principle of applying repeated subset filtration in the PIF algorithm is described in Section 6.2, and Section 6.3 contains combinations of modifications proposed in the previous sections.

6.1 Replacement of the editing algorithm

The first proposal was to replace the original Wilson editing with other editing algorithms. Specifically, the algorithms RENN (see 2.2.1) and AllKNN (see 2.2.2) were considered. The version using the RENN algorithm is further referred to as **PIF-RENN**, and the designation **PIF-AllKNN** is used for the modification of the PIF algorithm in which the AllKNN algorithm is used for editing. Both mentioned algorithms more thoroughly reduce 'noisy' border points, resulting in a 'smoother' decision boundary. The aim is to assess the impact of this fact on the subsequent filtration of disjoint subsets and the overall performance of the PIF algorithm. Pseudocode 12 contains the algorithms described above. Changes compared to the original version of the PIF algorithm are highlighted in red.

Algorithm 12 PIF-AllKNN/PIF-RENN

Let:
T be the original dataset
T_{new} be the reduced dataset
NE be the set of elements that are the nearest enemies for at least one of the other elements in T_{new}

PIF-AllKNN/PIF-RENN (continued)

K be the parameter for the AllKNN/RENN algorithms
m be the parameter indicating the minimum subset size
$d(x,y)$ be the distance (e.g., Euclidean distance) between elements x and y

$T_{new} \leftarrow T$
$T_{new} \leftarrow AllKNN(T_{new},K)/RENN(T_{new},K)$ ▷ Select AllKNN or RENN
for each $x \in T_{new}$ **do**
    find the nearest enemy ne_x
    add x to the subset S_{ne_x}
end for
for each $ne \in NE$ **do**
    if $|S_{ne}| \geq m$ **then**
        for each $y \in S_{ne}$ **do**
            for each $x \in S_{ne}$ where $x \neq y$ **do**
                if $d(y,ne) \geq \max\{d(x,y),d(x,ne)\}$ **then**
                $T_{new} \leftarrow T_{new} \setminus \{y\}$
                continue to the next y
                end if
            end for
        end for
    end if
end for
return T_{new}

6.2 Repeated PIF

This modification involves repeatedly applying the filtration rule to updated disjoint subsets containing elements with the same nearest enemy. For this reason, the proposed algorithm is further referred to as **Repeated PIF (RPIF)**. Another iteration of the algorithm occurs if, during the previous filtration, at least one element y was removed. Another option is to use a parameter specifying the maximum number of iterations. The RPIF algorithm is summarized in pseudocode 13. The parts highlighted in red indicate changes compared to the PIF algorithm.

Algorithm 13 RPIF

Let:
T be the original dataset
T_{new} be the reduced dataset
NE be the set of elements that are the nearest enemies for at least one of the other elements in T_{new}
K be the parameter for the Wilson Editing algorithm
m be the parameter indicating the minimum subset size
$d(x,y)$ be the distance (e.g., Euclidean distance) between elements x and y
max_iter be the parameter specifying the maximum number of iterations (OPTIONAL)

$T_{new} \leftarrow T$
$T_{new} \leftarrow WilsonEditing(T_{new},K)$
$progress \leftarrow true$
$iter \leftarrow 0$

RPIF (continued)

```
5:  while progress do
6:      iter = iter + 1
7:      if max_iter is set AND iter > max_iter then
8:          go to step 29
9:      end if
10:     progress ← false
11:     for each x ∈ T_new do
12:         find the nearest enemy ne_x
13:         add x to the subset S_ne_x
14:     end for
15:     for each ne ∈ NE do
16:         if |S_ne| ≥ m then
17:             for each y ∈ S_ne do
18:                 for each x ∈ S_ne where x ≠ y do
19:                     if d(y,ne) ≥ max{d(x,y), d(x,ne)} then
20:                         T_new ← T_new \ {y}
21:                         progress ← true
22:                         continue to the next y
23:                     end if
24:                 end for
25:             end for
26:         end if
27:     end for
28: end while
29: return T_new
```

6.3 RPIF with edition algorithm changed

Further experiments were conducted with two modified versions of the PIF algorithm, combining adjustments mentioned in sections 6.1 and 6.2. The version of the RPIF algorithm with the AllKNN editing algorithm is further referred to as **RPIF-AllKNN**, and the designation **RPIF-RENN** is used for the RPIF algorithm that performs editing using the RENN algorithm. The pseudocode for the described modifications is presented in section 14.

Algorithm 14 RPIF-AllKNN/RPIF-RENN

Let:
T be the original dataset
T_{new} be the reduced dataset
NE be the set of elements that are the nearest enemies for at least one of the other elements in T_{new}
K be the parameter for the AllKNN/RENN algorithms
m be the parameter indicating the minimum subset size
$d(x,y)$ be the distance (e.g., Euclidean distance) between elements x and y

RPIF-AllKNN/RPIF-RENN (continued)

$T_{new} \leftarrow T$
$T_{new} \leftarrow AllKNN(T_{new}, K) / RENN(T_{new}, K)$ ▷ Select AllKNN or RENN
$progress \leftarrow true$
while $progress$ **do**
    $progress \leftarrow false$
    for each $x \in T_{new}$ **do**
        find the nearest enemy ne_x
        add x to the subset S_{ne_x}
    end for
    for each $ne \in NE$ **do**
        if $|S_{ne}| \geq m$ **then**
            for each $y \in S_{ne}$ **do**
                for each $x \in S_{ne}$ where $x \neq y$ **do**
                    if $d(y, ne) \geq \max\{d(x, y), d(x, ne)\}$ **then**
                        $T_{new} \leftarrow T_{new} \setminus \{y\}$
                        $progress \leftarrow true$
                        continue to the next y
                    end if
                end for
            end for
        end if
    end for
end while
return T_{new}

Chapter 7

Experiments with instance selection algorithms

This chapter presents the main part of this thesis, focusing on the experimental evaluation and mutual comparison of instance selection algorithms described in Chapters 2 and 6. The chosen classification algorithm is KNN with a parameter K=3. The main metric used for comparing IS algorithms was the ratio between the achieved accuracy when using the reduced training set and the level of reduction of this set (7.1).

$$M_{AccSize} = \frac{Acc_{red}}{Size_{red}} \tag{7.1}$$

where:

- Acc_{red} is the achieved accuracy on the test set when classifying using the reduced training set, expressed as a percentage
- $Size_{red}$ is the size of the reduced training set compared to the original training set, expressed as a percentage

Additionally, classification accuracy, reduction level, and runtime of IS algorithms were considered in the comparison. For all IS algorithms, a custom implementation was developed. The main programming language used was Python, but computationally intensive and parallelizable parts of the algorithms were implemented in the C programming language for speed. Section 7.1 describes the process of tuning the parameters of instance selection algorithms. The main experiments, the comparison of IS algorithms, and the discussion of the obtained results are presented in Section 7.2.

7.1 Tuning parameters of instance selection algorithms

This section describes the tuning of parameters for IS algorithms conducted before their mutual comparison. Parameter tuning was performed separately for both datasets, meaning that the parameters of IS

algorithms were set based on independent evaluations on each dataset. The size of the training set subset for experiments was 75,000 for both the SOREL-20M and EMBER datasets. The parameter labels correspond to the labeling used in the pseudocodes in Chapters 2 and 6. Algorithms with multiple parameters were tested with all possible combinations of the specified parameter values. Table 7.1 contains the parameter values for each algorithm that were experimented with during the tuning process. The parameter selection was based on $M_{AccSize}$, considering only parameter combinations where the accuracy did not decrease by more than p percent compared to the original accuracy. This condition was created because the metric $M_{AccSize}$ does not include a penalty for accuracy loss. Algorithms with 'significantly' lower accuracy could therefore surpass algorithms with high accuracy if they achieved a greater level of dataset reduction. In this work, $p = 5\%$ was chosen. If no parameter combination meeting this condition was found, the combination with the smallest accuracy reduction was selected.

Algorithms	Parameter	Parameter values
ENN, RENN, AllKNN, CNN, ICF	K	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29
PIF/RPIF - all versions	K	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29
	m	2, 3, 4, 5, 6, 7, 8, 9, 10
	max_iter	2, 3, 4, all
DROP3	K_{ENN}	1, 3, 5, 7, 9, 11, 13, 15, 17, 19
	K	1, 3, 5, 7, 9, 11, 13, 15, 17, 19

Table 7.1 Tested parameter values of instance selection algorithms

The results of tuning the parameters of IS algorithms on the EMBER and SOREL-20M datasets are presented in Tables 7.2 and 7.3. Both tables include classification results in the first row when using the KNN classifier with the unreduced training set. The selected parameters are in the **Params** column. The parameter names correspond to the notation in the pseudocodes of IS algorithms, as described in Chapters 2 and 6.

Algorithm	**Params**	$M_{AccSize}$	**Size (%)**	**Acc (%)**	**F1 (%)**
KNN	-	-	100	92.67	92.69
ENN	K=27	1.01	89.69	90.28	90.32
RENN	K=27	1.02	86.15	88.27	88.34
AllKNN	K=29	1.09	82.14	89.65	89.70
CNN	K=3	5.22	17.54	**91.55**	**91.59**
ICF	K=7	5.07	17.45	88.56	88.66
DROP3	K_{ENN}=17, K=19	8.73	10.10	88.17	88.07
PIF	K=29, m=2	8.67	10.31	89.34	89.38
PIF-RENN	K=13, m=2	8.79	10.02	88.09	88.07
PIF-AllKNN	K=17, m=2	10.52	8.42	88.65	88.56
RPIF	K=25, m=2, max_iter=all	12.55	7.04	88.30	88.33
RPIF-RENN	K=13, m=2, max_iter=2	13.33	6.60	88.05	88.04
RPIF-AllKNN	K=17, m=2, max_iter=2	**16.31**	**5.41**	88.27	88.22

Table 7.2 Selected parameters of IS algorithms for experiments with the EMBER dataset

Further, the tables include metric values (7.1), sizes of reduced sets, accuracies, and F1 scores. In the columns $M_{AccSize}$, *Size*, *Acc*, and $F1$, the best achieved values across IS algorithms are highlighted in bold (the first row is not considered). If none of the combinations of parameter values for a specific algorithm resulted in an accuracy loss of less than five percent, the metric value in the corresponding row is highlighted in red. This case occurred only for the SOREL-20M dataset during the tuning of parameters for the ICF algorithm. In the case of both datasets, the CNN algorithm achieved the highest accuracy and F1 score. In terms of $M_{AccSize}$ and the level of reduction, for both the EMBER and SOREL-20M datasets, the RPIF-AllKNN algorithm was the best.

Algorithm	Params	$M_{AccSize}$	Size (%)	Acc (%)	F1 (%)
KNN	-	-	100	92.84	87.43
ENN	K=29	1.00	90.3880	90.71	82.81
RENN	K=29	1.01	88.1360	89.44	80.24
AllKNN	K=29	1.08	83.5973	90.35	82.12
CNN	K=3	4.89	18.6000	**90.97**	**84.69**
ICF	K=5	7.81	11.19	87.36	78.91
DROP3	$K_{ENN} = 13$, K=13	11.51	7.67	88.24	79.34
PIF	K=29, m=2	6.35	14.22	90.33	82.16
PIF-RENN	K=29, m=2	7.19	12.39	89.14	79.87
PIF-AllKNN	K=29, m=2	7.47	12.05	90.03	81.59
RPIF	K=29, m=2, max_iter = all	10.69	8.29	88.63	79.81
RPIF-RENN	K=19, m=2, max_iter = all	12.11	7.28	88.24	79.18
RPIF-AllKNN	K=29, m=2, max_iter = 4	**13.82**	**6.38**	88.23	79.21

Table 7.3 Selected parameters of IS algorithms for experiments with the SOREL-20M dataset

7.2 Comparison of IS algorithms

The content of this section involves the comparison of the listed instance selection algorithms. In Subsection 7.2.1, the comparison of IS algorithms when applied to the EMBER dataset is presented, and the results of IS algorithms for the SOREL-20M dataset are described in Subsection 7.2.2.

7.2.1 EMBER

In the case of EMBER, IS algorithms were applied to the following set sizes: 1k, 2k, 5k, 10k, 20k, 30k, 40k, 50k, 60k, 75k, 100k, 200k, 300k, 479,952. The parameter k in these values denotes multiples of a thousand. The DROP3 algorithm, due to computational complexity, was applied only up to a size of 300k. The remaining algorithms were used to reduce the entire training set. This subsection includes a total of four tables. Each table contains values for one of the following metrics: the $M_{AccSize}$ metric, classification accuracy, the size of reduced sets, and computational time. The content of each table is visualized in separate graphs. Values marked in red represent sets where the reduction resulted in an accuracy decrease greater than 5%. In addition to the RPIF algorithm version with parameters set based on experiments described in 7.1, a version labeled as **RPIF_2** was used, where the maximum number of

iterations (*max_iter*) was set to 2. This was done to evaluate whether additional iterations of the algorithm lead to a "significant" loss of classification accuracy at the expense of only a "small" reduction in the size of reduced sets.

Table 7.4 and Figure A.1 contain the values of the $M_{AccSize}$ metric achieved by IS algorithms depending on the sizes of reduced sets. For edition algorithms, the values of $M_{AccSize}$ decrease with increasing size of the reduced set. In the case of condensation and hybrid algorithms, the trend is opposite, i.e., $M_{AccSize}$ increases with the increasing size of the reduced set. An exception is the ICF algorithm, where a decreasing trend appeared for set sizes 300k and 479,952. The RPIF-AllKNN algorithm achieved the best result, while the ENN algorithm achieved the worst result in terms of $M_{AccSize}$. If we exclude edition algorithms from the evaluation, mainly included in this thesis due to their use in the hybrid algorithm PIF, the ICF algorithm achieved the worst values of the $M_{AccSize}$ metric. According to this metric, IS algorithms can be divided into five groups in the graph. The order of these groups depends on the achieved results (from the best to the worst). In the first group, which achieves the highest values of $M_{AccSize}$, are all versions of the RPIF algorithm, namely RPIF, RPIF_2, RPIF-AllKNN, and RPIF-RENN. For this group, the values of $M_{AccSize}$ during the reduction of the entire training set range from 15 to 18. The DROP3 algorithm follows, which achieved a value of 12.22 for a size of 300k. The third group consists of versions of the PIF algorithm, with the best results achieved by PIF-AllKNN. The range of values for this group during the reduction of the entire training set ranges from 9 to 12. The CNN, MSS, and ICF algorithms form the fourth group. The values of $M_{AccSize}$ for the fourth group range from 6 to 9. The last group consists of edition algorithms, which achieved values around one.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	
1000	0.79	1.02	1.08	1.41	1.87	3.73	2.22	
2000	0.83	1.00	1.07	1.29	2.22	2.94	2.48	
5000	0.86	1.00	1.03	1.19	2.80	4.14	3.01	
10000	0.88	1.00	1.03	1.15	3.32	4.60	3.50	
20000	0.90	1.00	1.02	1.13	3.80	4.88	4.06	
30000	0.91	1.00	1.03	1.11	4.25	4.96	4.38	
40000	0.91	1.00	1.02	1.10	4.47	4.69	4.63	
50000	0.92	1.00	1.02	1.10	4.66	5.19	4.90	
60000	0.92	1.01	1.02	1.09	4.95	4.96	5.11	
75000	0.93	1.01	1.02	1.09	5.21	5.07	5.37	
100000	0.93	1.00	1.02	1.08	5.57	5.18	5.65	
200000	0.94	1.00	1.02	1.06	6.53	6.66	6.61	
300000	0.95	1.00	1.01	1.06	7.20	6.74	7.18	
479952	0.95	1.00	1.01	1.05	8.10	6.39	7.96	
Original size	**PIF**	**PIF-AllKNN**	**PIF-RENN**	**RPIF**	**RPIF-AllKNN**	**RPIF-RENN**	**DROP3**	**RPIF_2**
1000	4.81	6.30	6.35	6.52	8.95	8.31	5.24	6.23
2000	4.04	5.68	4.49	6.38	8.36	6.39	4.13	5.61
5000	4.16	5.74	4.29	6.02	8.30	6.28	4.91	5.66
10000	5.22	6.92	5.34	8.32	10.62	8.48	5.53	7.68
20000	6.01	7.46	6.39	9.91	11.99	9.80	6.78	8.78
30000	6.37	7.79	6.72	10.51	12.12	10.32	7.28	8.90
40000	6.82	8.18	7.11	11.15	0.14	11.61	7.72	9.99
50000	7.22	9.03	7.41	11.94	13.92	11.49	8.22	10.66
60000	7.68	8.89	7.67	11.91	13.64	11.59	8.40	11.04
75000	7.62	9.11	7.80	12.55	14.33	12.46	8.73	11.31
100000	7.94	9.41	8.18	12.91	14.43	12.75	9.10	11.82
200000	8.72	10.19	8.87	14.47	16.15	13.96	11.49	13.12
300000	9.34	10.46	9.26	15.76	16.95	14.68	12.22	14.33
479952	10.08	11.33	9.86	16.94	18.03	15.63	-	15.50

Table 7.4 Values of the $M_{AccSize}$ metric achieved by IS algorithms - EMBER

Sizes of the reduced sets are provided in Table 7.5. The visualization of this table is presented in Figure A.2. When examining the results, a similarity with the outcomes related to $M_{AccSize}$ is evident, confirming the influence of the reduced set size on the overall metric value. The graph depicting the sizes of reduced sets in relation to the original sizes almost mirrors Figure A.1. The algorithm RPIF-AllKNN achieved the best results in terms of reducing the training set, while the algorithm ENN achieved the least reduction. Ignoring edition algorithms, the ICF algorithm achieved the least favorable results as the size of the reduced set increased. In this case as well, algorithms can be divided into five groups, listed from the best to the worst. The first group consists of the RPIF algorithm and all its modifications. The sizes of reduced sets when using these algorithms on the entire training set ranged from 5% to 6% compared to the original size. In the second group is the DROP3 algorithm, which managed to reduce the set of size 300k to 7.36%. All three versions of the PIF algorithm form the third group. The sizes of the reduced sets when using these algorithms ranged from 8% to 10%. The MSS, CNN, and ICF algorithms, which achieved a reduction in the training set size between 11% and 15%, form the fourth group. The fifth group is composed of edition algorithms. The sizes of reduced sets ranged from 89% to 94% compared to the original size of the training set. For condensation algorithms CNN and MSS, it can be observed that with increasing size of the reduced set, the level of reduction compared to hybrid algorithms decreases more rapidly and approaches the reduction level of the PIF algorithm. Edition algorithms achieved a lower level of reduction with the increasing size of the original set.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	
1000	100	73.80	67.50	53.50	42.10	19.50	34.00	
2000	100	78.05	69.15	59.60	36.70	25.75	31.00	
5000	100	81.50	75.34	67.38	30.24	18.84	27.04	
10000	100	83.43	77.97	72.14	26.05	17.66	23.86	
20000	100	86.17	81.16	75.83	23.37	17.36	21.08	
30000	100	87.10	82.99	78.17	21.09	17.31	19.66	
40000	100	88.07	84.44	79.50	20.18	18.42	18.81	
50000	100	88.65	85.01	80.57	19.44	16.72	17.91	
60000	100	89.07	85.39	81.16	18.36	17.61	17.25	
75000	100	89.69	86.15	82.17	17.54	17.45	16.52	
100000	100	90.39	87.33	83.46	16.55	17.13	15.80	
200000	100	91.93	89.62	86.29	14.28	13.24	13.69	
300000	100	92.70	90.65	87.62	13.03	13.26	12.71	
479952	100	93.48	91.70	89.01	11.68	14.14	11.57	

Original size	PIF	PIF-AllKNN	PIF-RENN	RPIF	RPIF-AllKNN	RPIF-RENN	DROP3	RPIF_2
1000	15.50	11.80	11.80	11.30	8.30	8.90	13.80	11.80
2000	19.45	13.30	17.00	12.25	8.60	11.45	18.60	13.90
5000	18.92	13.50	17.82	12.90	9.08	12.22	16.10	13.78
10000	15.68	11.80	15.30	9.80	7.43	9.67	14.84	10.73
20000	14.09	11.27	13.23	8.53	6.92	8.55	12.36	9.63
30000	13.41	10.95	12.55	8.04	6.81	8.11	11.77	9.34
40000	12.72	10.35	12.02	7.68	6.18	7.34	11.05	8.57
50000	12.16	9.61	11.57	7.28	6.24	7.34	10.38	8.16
60000	11.49	9.82	11.47	7.32	6.23	7.35	10.26	7.94
75000	11.69	9.59	11.29	6.99	6.05	7.04	10.10	7.84
100000	11.31	9.48	10.85	6.87	5.91	6.94	9.58	7.51
200000	10.34	8.87	10.19	6.16	5.55	6.37	7.80	6.82
300000	9.78	8.62	9.86	5.76	5.37	6.15	7.36	6.36
479952	9.21	8.13	9.24	5.39	5.04	5.80	-	5.95

Table 7.5 Sizes (%) of reduced sets - EMBER

Classification accuracies of IS algorithms depending on the sizes of original sets are provided in Table 7.6. Graph A.3 visually represents the values from this table. In terms of achieved classification accuracy, the CNN algorithm achieved the best results, surpassing the edition algorithms. The ICF algorithm achieved the worst results as the sizes of reduced sets increased. Excluding the edition algorithms from the evaluation, the PIF algorithm ranked second for the entire training set of the EMBER dataset, and the third place was occupied by the RPIF algorithm with the *max_iter* parameter set to 2. The MSS and PIF-AllKNN algorithms were close to the RPIF_2 algorithm in terms of the achieved accuracy. The remaining versions of the PIF and RPIF algorithms achieved the lowest classification accuracy. The RPIF-AllKNN algorithm, which achieved the largest reduction in the training set, was also the third worst algorithm in terms of achieved accuracy. The DROP3 algorithm, which also performed among the best in terms of reduction, achieved the second worst classification accuracy for the size of 300k.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	
1000	79.44	75.51	73.10	75.23	78.52	72.75	75.43	
2000	82.83	78.39	73.80	77.18	81.46	75.69	76.88	
5000	85.74	81.25	77.69	80.07	84.59	78.04	81.29	
10000	88.12	83.62	80.28	82.70	86.56	81.15	83.48	
20000	89.96	86.15	83.15	85.49	88.75	84.76	85.50	
30000	90.74	87.42	85.12	86.90	89.68	85.81	86.01	
40000	91.46	88.33	86.31	87.71	90.29	86.45	87.11	
50000	91.85	88.92	86.90	88.32	90.66	86.82	87.84	
60000	92.23	89.54	87.40	88.77	90.97	87.26	88.12	
75000	92.64	90.15	88.13	89.61	91.47	88.44	88.67	
100000	93.18	90.77	89.04	90.27	92.14	88.75	89.28	
200000	94.26	92.26	90.98	91.88	93.24	88.15	90.51	
300000	94.82	93.04	91.90	92.62	93.90	89.37	91.29	
479952	95.43	93.78	92.75	93.49	94.54	90.37	92.16	
Original size	**PIF**	**PIF-AllKNN**	**PIF-RENN**	**RPIF**	**RPIF-AllKNN**	**RPIF-RENN**	**DROP3**	**RPIF_2**
1000	74.57	74.35	74.92	73.70	74.28	73.97	72.31	73.50
2000	78.60	75.61	76.35	78.21	71.87	73.17	76.76	77.97
5000	78.76	77.46	76.40	77.67	75.41	76.68	79.11	78.00
10000	81.85	81.61	81.72	81.55	78.88	82.00	82.04	82.41
20000	84.66	84.09	84.48	84.45	82.92	83.79	83.76	84.60
30000	85.39	85.29	84.33	84.49	82.53	83.68	85.71	83.19
40000	86.72	84.63	85.45	85.64	85.19	85.27	85.31	85.66
50000	87.75	86.73	85.79	86.97	86.90	84.28	85.31	86.95
60000	88.22	87.29	87.90	87.11	85.02	85.25	86.17	87.70
75000	89.09	87.37	88.04	87.74	86.72	87.69	88.19	88.71
100000	89.77	89.24	88.73	88.72	85.21	88.45	87.15	88.79
200000	90.25	90.32	90.43	89.14	89.66	88.95	89.53	89.48
300000	91.42	90.12	91.26	90.86	91.02	90.27	89.85	91.18
479952	92.83	92.08	91.15	91.35	90.91	90.62	-	92.22

Table 7.6 Classification accuracies (%) achieved by IS algorithms - EMBER

Table 7.7 and graph A.4 contain data for the evaluation and comparison of IS algorithms in terms of computational time. When looking at the graph, IS algorithms can be divided into four groups based on computational time. The first group is formed by the DROP3 algorithm, which was clearly the slowest. Processing a set of size 300k took approximately 77,570 seconds, which is several times longer than the computation time of other IS algorithms when processing the entire training set. The CNN algorithm, with a computation time of approximately 9,909 seconds for processing the entire training set, forms the second group. This time is roughly three times longer than the computation time of all other hybrid algorithms (except DROP3) and the MSS algorithm. These algorithms form the third group, and their

computation times ranged between 2,709 and 3,733 seconds. The fastest in this group was the ICF algorithm, making it the fastest among hybrid and condensation algorithms for the EMBER dataset. Following were the versions of PIF and RPIF algorithms without replaced edition algorithms. Even longer computation times were achieved by PIF and RPIF algorithms used in combination with the AllKNN algorithm, and the slowest in this group were the algorithms using RENN for editing. The fourth group is formed by edition algorithms. Computation times required for processing the entire training set in this group of algorithms ranged between 57 and 697 seconds. The fastest was the ENN algorithm. The AllKNN algorithm was the second fastest, and the slowest among the edition algorithms was RENN. With increasing sizes of the reduced sets, the computation time of this algorithm increases faster than in the case of the ENN and AllKNN algorithms. This fact was also evident in the algorithms from the third group, which use this algorithm for editing.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	
1000	-	0.0	0.1	0.5	41.8	0.4	0.1	
2000	-	0.1	0.3	1.0	95.6	0.6	0.2	
5000	-	0.2	0.9	2.5	203.6	1.8	0.5	
10000	-	0.3	1.7	4.9	380.4	5.4	1.8	
20000	-	0.6	3.4	10.1	803.6	20.2	6.4	
30000	-	1.0	8.3	15.3	971.9	13.1	13.1	
40000	-	1.4	10.9	21.1	1173.5	27.1	22.3	
50000	-	1.8	21.0	26.4	1506.0	40.5	36.1	
60000	-	2.2	37.3	32.1	1611.1	52.6	51.0	
75000	-	2.3	25.3	39.8	1681.4	68.6	71.0	
100000	-	3.7	47.2	53.7	1751.1	192.5	126.2	
200000	-	12.1	124.3	111.9	3436.1	426.7	529.0	
300000	-	24.4	364.4	175.3	5488.3	1006.9	1212.0	
479952	-	57.3	697.0	298.7	9909.0	2709.6	3242.5	
Original size	**PIF**	**PIF-AllKNN**	**PIF-RENN**	**RPIF**	**RPIF-AllKNN**	**RPIF-RENN**	**DROP3**	**RPIF_2**
1000	0.2	0.5	0.4	0.4	0.4	0.5	65.6	0.2
2000	0.4	0.9	0.6	0.6	0.9	0.6	169.2	0.4
5000	1.3	2.7	1.7	1.7	2.4	1.9	388.6	1.4
10000	4.1	6.7	5.3	4.8	6.2	5.4	716.9	4.2
20000	16.9	20.3	20.0	18.3	19.2	20.4	1213.2	17.2
30000	17.0	25.9	27.9	18.4	27.8	28.1	1698.6	18.2
40000	31.0	40.7	45.9	32.1	44.0	46.6	2316.8	31.1
50000	47.3	67.9	61.4	47.7	61.6	62.0	3118.7	47.7
60000	68.5	88.2	83.3	66.2	79.6	83.9	4034.1	69.0
75000	94.3	116.2	100.4	93.2	111.4	101.5	5707.2	95.3
100000	181.4	200.2	200.5	160.4	198.9	202.9	9401.9	183.4
200000	587.1	610.2	711.8	595.7	602.0	716.4	34861.1	591.5
300000	1340.8	1378.6	1533.7	1363.8	1370.9	1541.0	77578.9	1349.7
479952	3317.5	3350.8	3709.0	3348.7	3368.8	3732.9	-	3335.9

Table 7.7 Durations (s) of IS algorithms - EMBER

7.2.2 SOREL-20M

The experiments described in this subsection can be divided into two parts. The first part, where subsets of the training set of the SOREL-20M dataset are experimented with, is described in Section 7.2.2.1. Section 7.2.2.2 details the reduction of the entire training set using stratification.

7.2.2.1 Reduction of subsets of the training set without using stratification

Reduction of the SOREL-20M dataset without using stratification was performed on subsets of the following sizes: 1k, 2k, 5k, 10k, 20k, 30k, 40k, 50k, 60k, 75k, 100k, 200k, 300k, 500k, 750k, 1000k. Exceptions are the MSS and DROP3 algorithms, for which processing was only conducted up to the size of 300k. For the DROP3 algorithm, experiments were halted due to high computational times, and for the MSS algorithm, the implemented version encountered a memory shortage problem during the reduction of larger sets. This section includes tables and graphs containing values for the same metrics as in 7.2.1. In addition to experiments with versions of the RPIF algorithms, RPIF-AllKNN, and RPIF-RENN, where the max_iter parameter was set based on experiments, experiments were also conducted with the $max\_iter = 2$ parameter setting. These versions are further referred to as RPIF_2, RPIF-AllKNN_2, and RPIF-RENN_2.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	DROP3	
1000	0.78	0.93	1.05	1.26	1.84	4.27	1.98	6.44	
2000	0.81	0.96	1.00	1.20	2.14	4.49	2.27	8.86	
5000	0.83	0.96	0.99	1.13	2.59	5.22	2.68	7.27	
10000	0.85	0.96	0.99	1.10	3.00	6.11	3.03	7.68	
20000	0.86	0.96	0.99	1.07	3.43	6.21	3.40	9.13	
30000	0.87	0.97	0.99	1.07	3.70	6.65	3.66	9.60	
40000	0.88	0.96	0.98	1.06	3.98	7.09	3.84	9.46	
50000	0.88	0.97	0.98	1.05	4.19	7.26	4.07	9.64	
60000	0.89	0.97	0.98	1.05	4.38	7.10	4.19	10.47	
75000	0.89	0.97	0.98	1.04	4.66	7.33	4.36	11.08	
100000	0.90	0.97	0.98	1.04	4.80	7.13	4.55	11.40	
200000	0.90	0.97	0.98	1.03	5.61	7.78	5.14	13.93	
300000	0.90	0.97	0.98	1.02	6.22	7.99	5.57	14.66	
500000	0.90	0.96	0.97	1.00	6.83	8.12	-	-	
750000	0.91	0.96	0.97	1.00	7.37	8.87	-	-	
1000000	0.91	0.96	0.96	1.00	7.89	8.96	-	-	
Original size	**PIF**	**PIF-AllKNN**	**PIF-RENN**	**RPIF**	**RPIF-AllKNN**	**RPIF-RENN**	**RPIF_2**	**RPIF-AllKNN_2**	**RPIF-RENN_2**
1000	3.10	5.22	5.13	4.30	9.18	5.92	4.47	8.51	9.06
2000	4.93	6.71	5.86	7.92	13.52	16.29	7.20	10.08	8.28
5000	4.63	5.99	7.32	7.37	11.46	9.69	6.65	9.16	11.80
10000	4.83	6.43	5.91	7.54	12.86	8.76	6.83	10.59	9.20
20000	5.28	6.47	5.95	8.72	12.61	9.67	7.75	10.42	9.22
30000	5.49	6.66	6.18	9.17	12.84	10.26	8.19	11.16	10.17
40000	5.47	6.64	6.37	9.08	12.92	10.88	8.17	11.09	10.40
50000	5.58	6.71	6.39	9.53	12.61	10.90	8.42	10.73	10.34
60000	5.68	6.74	6.57	9.73	12.50	11.20	8.58	10.72	10.56
75000	5.93	6.84	6.72	10.19	13.32	11.66	8.98	11.34	10.92
100000	6.19	7.06	6.90	10.56	13.91	12.22	9.35	11.94	11.31
200000	6.90	7.67	7.62	12.25	15.53	13.58	10.74	13.33	12.64
300000	7.22	8.10	8.01	13.50	16.38	14.76	11.44	14.12	13.48
500000	7.80	8.67	8.62	14.66	18.60	15.81	12.66	15.40	14.81
750000	8.23	9.16	8.79	16.20	19.52	17.18	13.52	16.32	15.30
1000000	8.77	9.52	9.15	17.38	20.63	17.99	14.58	17.20	16.06

Table 7.8 Values of the $M_{AccSize}$ metric achieved by IS algorithms - SOREL-20M

The values of the $M_{AccSize}$ metrics for IS algorithms, depending on the sizes of the reduced sets, are displayed in Table 7.8 and the graph A.5. Similar to the EMBER dataset, the $M_{AccSize}$ values for edition algorithms decrease with increasing sizes of the reduced sets. In the case of condensation and hybrid algorithms, the metric shows an increasing trend. When looking at the graph, IS algorithms can be divided into six groups based on the results. The reported metric values relate to the results on a subset of the SOREL-20M training set with a size of 1000k unless otherwise specified. The RPIF-AllKNN algorithm, achieving the highest $M_{AccSize}$ metric value, forms the first group. This algorithm reached a value of 20.63. The second-best group consists of the algorithms RPIF-RENN, DROP3, RPIF, RPIF-AllKNN_2, and RPIF-RENN_2. The order of these algorithms was chosen based on the $M_{AccSize}$ metric

values (for the DROP3 algorithm, it is an estimate). The $M_{AccSize}$ values ranged from 16 to 18. The third group is composed of the RPIF_2 algorithm with an achieved metric value of 14.58. The fourth group consists of all three versions of the PIF algorithm along with the ICF algorithm. The PIF-AllKNN algorithm, with an achieved value of 9.52, was the best algorithm in this group. The lowest value of 8.77 was achieved by the PIF algorithm. The condensation algorithms CNN and MSS form the fifth group. The MSS algorithm reached a value of 5.57 on a set of size 300k. The CNN algorithm, which was better from this pair, reached a value of 7.89 on a set of size 1000k. The last group is composed of edition algorithms, with values hovering around one. If we exclude edition algorithms from the evaluation, then, in the case of the SOREL-20M dataset, the ICF algorithm did not achieve the worst result in terms of $M_{AccSize}$, but MSS did.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	DROP3	
1000	100	82.00	67.80	60.40	40.30	17.00	37.50	11.60	
2000	100	81.70	77.25	63.75	35.65	16.85	34.35	8.05	
5000	100	83.90	78.02	70.94	30.66	14.22	29.86	10.80	
10000	100	86.06	82.56	74.80	27.36	12.13	26.86	10.35	
20000	100	87.43	84.29	77.97	24.36	12.51	24.27	8.82	
30000	100	88.29	85.64	79.57	22.56	12.04	22.84	8.52	
40000	100	89.15	86.46	80.93	21.14	11.39	21.90	8.84	
50000	100	89.53	86.96	82.01	20.36	11.15	21.03	8.58	
60000	100	89.98	87.51	82.80	19.63	11.44	20.40	8.11	
75000	100	90.39	88.14	83.61	18.60	11.19	19.69	7.67	
100000	100	91.05	88.92	84.68	17.76	11.55	18.99	7.30	
200000	100	92.44	90.55	87.03	15.46	10.81	17.08	6.09	
300000	100	93.02	91.49	88.14	14.22	10.58	15.79	5.53	
500000	100	93.83	92.45	89.48	12.86	10.43	-	-	
750000	100	94.32	93.12	90.40	11.87	9.54	-	-	
1000000	100	94.68	93.60	91.05	11.15	9.50	-	-	

Original size	PIF	PIF-AllKNN	PIF-RENN	RPIF	RPIF-AllKNN	RPIF-RENN	RPIF_2	RPIF_AllKNN_2	RPIF_RENN_2
1000	24.40	14.40	14.20	17.40	8.00	12.20	16.60	8.80	7.90
2000	15.85	11.60	13.10	9.60	5.65	4.60	10.75	7.55	9.00
5000	17.42	13.04	10.58	10.56	6.56	8.08	11.84	8.48	6.50
10000	16.84	12.61	13.60	10.67	6.18	9.04	11.79	7.60	8.59
20000	15.87	12.83	13.93	9.38	6.47	8.49	10.66	7.92	8.86
30000	15.46	12.61	13.61	8.95	6.40	7.99	10.20	7.42	8.19
40000	15.54	12.75	13.27	9.20	6.45	7.60	10.29	7.53	8.02
50000	15.35	12.85	13.26	8.79	6.62	7.61	10.06	7.83	8.13
60000	15.14	12.83	12.97	8.67	6.70	7.51	9.92	7.87	7.97
75000	14.57	12.57	12.76	8.30	6.33	7.21	9.49	7.50	7.76
100000	14.16	12.28	12.59	8.12	6.11	6.99	9.23	7.16	7.57
200000	12.76	11.35	11.52	6.93	5.47	6.29	8.03	6.44	6.86
300000	12.07	10.72	11.02	6.43	5.12	5.88	7.52	6.05	6.45
500000	11.26	10.11	10.34	5.83	4.60	5.36	6.87	5.62	5.91
750000	10.65	9.61	9.91	5.34	4.37	5.00	6.46	5.35	5.68
1000000	10.24	9.27	9.68	5.05	4.16	4.82	6.08	5.11	5.44

Table 7.9 Sizes (%) of reduced sets - SOREL-20M

Table 7.9 and graph A.6 display the sizes of the reduced sets depending on the sizes of the original sets. In the case of the SOREL-20M dataset, there was a similarity between these results and the $M_{AccSize}$ results. The RPIF-AllKNN algorithm again achieved the best reduction levels, and the ENN algorithm reduced individual sets the least. If we exclude edition algorithms from the evaluation, then the MSS algorithm achieved the smallest reduction levels. Based on the graph, IS algorithms can be divided into three groups. The first group consists of all versions of the RPIF algorithm. When applying these algorithms, the sizes of the reduced sets ranged from 4% to 6%. All versions of the PIF, ICF, CNN, and MSS algorithms form the second group. Although the size of the reduced sets using the CNN algorithm is larger than other algorithms at small original sizes, with increasing original set sizes, the reduction level of CNN approached the other algorithms in this group. They were able to reduce the original size of 1000k to sizes ranging from 9% to 12%. The MSS algorithm reduced the set of size 300k to 15.79%. The

third group consists of edition algorithms, where, in the case of the SOREL-20M dataset, the reduction level decreased with increasing original set sizes. The reduced sizes ranged from 91% to 95% compared to the original size of 1000k.

Table 7.10 contains classification accuracies achieved by IS algorithms depending on the sizes of the reduced sets. This table is visualized in graph A.7. Since the ranking of IS algorithms in terms of classification accuracy varies for different sizes, they cannot be clearly divided into groups. The mentioned classification accuracies in this paragraph are related to the set of size 1000k unless stated otherwise. In most cases, edition algorithms achieve the best results. Classification accuracies for edition algorithms range between 90% and 91%. Among the worst algorithms in terms of achieved accuracy are ICF, DROP3, RPIF-AllKNN, and RPIF-RENN. For these algorithms, accuracies ranged from 85% to 87%. Modifications of RPIF with the parameter $max\_iter = 2$ set (at the cost of a lower reduction level), all three versions of the PIF algorithm, and the condensation algorithm CNN achieved better results. Classification accuracies for these algorithms ranged between 87% and 90%. The MSS algorithm achieved an accuracy of 88% for the set of size 300k.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	DROP3	
1000	77.53	76.67	71.26	76.20	73.96	72.51	74.08	74.65	
2000	80.89	78.15	77.48	76.81	76.42	75.66	78.11	71.36	
5000	82.67	80.80	77.50	80.45	79.47	74.29	79.93	78.53	
10000	84.64	82.85	81.46	82.23	81.98	74.13	81.35	79.49	
20000	86.20	84.19	83.11	83.72	83.55	77.61	82.59	80.55	
30000	87.29	85.53	84.49	84.91	83.45	80.06	83.69	81.74	
40000	87.77	86.03	84.95	85.65	84.09	80.73	84.10	83.59	
50000	88.44	86.82	85.49	86.36	85.23	80.89	85.63	82.71	
60000	88.56	87.12	85.77	86.79	85.88	81.15	85.45	84.93	
75000	89.28	87.43	86.34	87.28	86.77	82.00	85.88	85.01	
100000	89.61	88.33	87.39	88.10	85.23	82.37	86.44	83.15	
200000	90.29	89.48	88.76	89.22	86.69	84.12	87.83	84.83	
300000	90.42	89.91	89.29	89.70	88.44	84.60	88.00	81.01	
500000	90.39	89.94	89.78	89.71	87.89	84.73	-	-	
750000	91.10	90.37	90.05	90.64	87.47	84.61	-	-	
1000000	91.29	90.67	90.21	90.60	87.96	85.10	-	-	
Original size	**PIF**	**PIF-AllKNN**	**PIF-RENN**	**RPIF**	**RPIF-AllKNN**	**RPIF-RENN**	**RPIF_2**	**RPIF_AllKNN_2**	**RPIF_RENN_2**
1000	75.53	75.21	72.84	74.89	73.44	72.22	74.20	74.89	71.57
2000	78.21	77.85	76.77	76.05	76.40	74.95	77.38	76.12	74.48
5000	80.63	78.12	77.42	77.79	75.19	78.33	78.72	77.67	76.68
10000	81.36	81.09	80.34	80.48	79.50	79.20	80.52	80.45	78.99
20000	83.77	82.99	82.87	81.79	81.54	82.04	82.57	82.52	81.65
30000	84.81	83.96	84.12	82.10	82.21	81.96	83.56	82.78	83.27
40000	84.99	84.59	84.49	83.48	83.33	82.64	84.02	83.44	83.42
50000	85.67	86.15	84.79	83.81	83.48	82.88	84.71	84.01	84.03
60000	85.94	86.41	85.17	84.40	83.80	84.12	85.15	84.28	84.18
75000	86.33	86.01	85.69	84.53	84.32	84.09	85.27	85.09	84.69
100000	87.73	86.66	86.93	85.71	84.98	85.37	86.36	85.49	85.54
200000	87.98	87.08	87.73	84.97	84.94	85.49	86.17	85.83	86.72
300000	87.16	86.83	88.27	86.79	83.84	86.83	86.00	85.34	87.00
500000	87.89	87.68	89.16	85.47	85.58	84.68	87.03	86.59	87.56
750000	87.68	87.96	87.15	86.59	85.25	85.94	87.37	87.33	86.85
1000000	89.78	88.20	88.60	87.71	85.88	86.73	88.56	87.88	87.34

Table 7.10 Classification accuracies (%) achieved by IS algorithms - SOREL-20M

Table 7.11 contains recorded computational times of IS algorithms depending on the sizes of the reduced subsets of the training set of the SOREL-20M dataset. The data from this table is visualized in graph A.8. Based on the computational times, IS algorithms can be divided into six groups. The mentioned times in this paragraph are related to the reduced set of size 1000k unless stated otherwise. Edition algorithms form the first group. These algorithms achieved the lowest computational times, with reduction times ranging between 200 and 2200 seconds. ENN algorithm was the fastest, and RENN algorithm was the slowest, with its computational time increasing faster with the growing size of the

reduced set compared to ENN and AllKNN algorithms. This effect was also observed in the modified versions of PIF and RPIF algorithms that used this algorithm for editing. The second group consists of PIF, PIF-AllKNN, RPIF-AllKNN, RPIF-AllKNN_2, RPIF, and RPIF_2 algorithms. For this group, the reduction time ranged between 11594 and 12560 seconds. The next group is formed by PIF-RENN and RPIF-RENN_2 algorithms. The computational time of the PIF-RENN algorithm was 13713 seconds, and RPIF-RENN_2 reduced the set of size 1000k in approximately 13799 seconds. Group four is composed of MSS, ICF, and RPIF-RENN algorithms. The slowdown of the RPIF-RENN algorithm was influenced by the repetition of subset filtration without a set *max_iter* parameter in combination with RENN, which was the slowest representative of editing algorithms. The computational time of this algorithm was 15352 seconds. The ICF algorithm completed the computation in 15272 seconds. The set of size 300k was reduced by the MSS algorithm in approximately 1624 seconds. In this algorithm, due to non-parallelizable parts, a slowdown can be expected. The fifth group is represented by the CNN algorithm with a computational time of almost 34000 seconds, which is more than twice the time compared to algorithms from the previous group. In the case of the SOREL-20M dataset, seemingly the slowest algorithm was DROP3, with a computational time exceeding 53409 seconds for the reduction of the set of size 300k.

Original size	KNN	ENN	RENN	AllKNN	CNN	ICF	MSS	DROP3	
1000	-	0.0	0.3	0.5	35.0	0.4	0.1	36.9	
2000	-	0.1	0.4	1.0	94.8	0.6	0.2	96.6	
5000	-	0.2	1.2	2.5	201.8	2.1	0.6	252.9	
10000	-	0.3	2.2	4.9	378.4	6.9	2.2	466.7	
20000	-	0.6	5.1	10.2	675.8	26.6	7.9	822.5	
30000	-	1.0	8.1	15.3	924.8	15.1	16.0	1155.6	
40000	-	1.4	9.0	20.7	1469.6	26.2	28.3	1596.7	
50000	-	1.9	15.7	26.3	1235.1	39.3	44.3	2189.3	
60000	-	2.3	28.2	32.0	1662.1	56.2	62.7	2854.3	
75000	-	2.4	20.8	39.3	1524.1	78.8	87.4	4074.8	
100000	-	3.6	35.4	52.8	1834.3	143.8	163.0	6795.3	
200000	-	11.4	141.2	110.9	3336.1	563.8	665.9	25019.2	
300000	-	23.3	195.7	175.0	6015.1	1307.2	1623.8	53409.4	
500000	-	58.5	719.4	311.3	10462.1	3814.4	-	-	
750000	-	124.5	1790.8	516.3	24328.3	8729.5	-	-	
1000000	-	216.7	2170.6	734.3	33976.8	15272.0	-	-	

Original size	PIF	PIF-AllKNN	PIF-RENN	RPIF	RPIF-AllKNN	RPIF-RENN	RPIF_2	RPIF_AllKNN_2	RPIF_RENN_2
1000	0.2	0.7	0.4	0.4	0.7	0.7	0.2	0.7	0.4
2000	0.5	1.3	0.7	0.8	1.4	0.9	0.5	1.3	0.8
5000	1.7	3.7	2.8	2.0	4.0	3.6	1.7	3.8	2.8
10000	5.1	8.8	7.0	6.2	9.3	6.9	5.2	9.1	7.1
20000	20.9	27.0	24.8	22.4	27.4	24.1	21.4	27.5	25.3
30000	16.8	29.9	23.8	18.5	31.0	24.1	17.1	30.4	24.0
40000	28.1	40.9	33.9	31.2	44.3	34.2	28.8	44.0	34.3
50000	35.5	60.5	47.4	38.5	59.8	49.1	36.1	61.4	48.1
60000	50.6	77.6	75.5	54.1	79.0	67.3	51.8	78.1	76.5
75000	66.0	106.5	83.9	71.7	108.1	96.5	67.7	107.9	85.5
100000	127.1	164.8	157.4	139.0	168.1	159.5	129.9	168.1	159.6
200000	518.5	575.7	657.6	572.9	585.1	626.9	527.0	581.8	663.8
300000	1305.9	1240.7	1368.8	1291.2	1255.3	1440.6	1199.2	1242.8	1361.6
500000	3181.3	3246.0	3906.1	3332.9	3268.8	4126.1	3215.9	3275.1	3933.0
750000	7409.6	7552.3	9131.6	7483.7	7210.2	8558.3	7469.9	7593.5	9188.7
1000000	11835.3	11594.9	13713.3	12559.3	11695.3	15351.9	11961.2	11668.6	13798.5

Table 7.11 Durations (s) of IS algorithms - SOREL-20M

7.2.2.2 Reduction of the training set using stratification

The entire training set of the SOREL-20M dataset was reduced using stratification [59], meaning the training set was randomly divided into a chosen number of equally-sized subsets while preserving the class distribution. IS algorithms were then applied to these subsets, and the results were combined into the reduced training set. The number of subsets was selected based on experiments with values of 100,

200, 300, and 400. Due to computational complexity, CNN and DROP3 algorithms were excluded from experiments with the number of subsets. Two values of the dividing parameter were chosen based on the $M_{AccSize}$ metric. Edition algorithms, including ENN, RENN, and AllKNN, achieved the best results when using a parameter value of 400. Higher values of $M_{AccSize}$ were thus achieved by applying edition algorithms to more smaller subsets. For the remaining algorithms, the lowest tested value of 100 was selected. This value was also used for CNN and DROP3 algorithms.

IS Algorithm	Stratification	Size (%)	Accuracy (%)	F1(%)	$M_{AccSize}$	Duration (s)
KNN	-	100	92.04	86.54	-	-
ENN	400	87.31	90.00	81.01	1.03	**219.1**
RENN	400	83.80	88.35	77.30	1.05	1540.9
AllKNN	400	77.53	89.18	78.98	1.15	3823.3
CNN	100	14.11	87.95	81.10	6.23	54364.6
ICF	100	11.52	89.36	82.19	7.76	7892.9
MSS	100	20.12	**91.61**	**85.73**	4.55	8066.5
DROP3	100	7.64	91.43	84.58	11.96	366067.1
PIF	100	14.99	90.56	82.77	6.04	6883.5
PIF-AllKNN	100	12.68	90.05	81.31	7.10	10357.2
PIF-RENN	100	13.15	89.80	80.84	6.83	9818.5
RPIF	100	8.48	90.29	82.44	10.65	8309.2
RPIF-AllKNN	100	**6.24**	89.94	81.39	**14.43**	10563.7
RPIF-RENN	100	7.48	89.84	81.27	12.01	9902.7
RPIF_2	100	9.69	90.33	82.50	9.32	7824.2
RPIF-AllKNN_2	100	7.42	89.88	81.21	12.11	10481.8
RPIF-RENN_2	100	8.00	89.63	80.81	11.20	9895.6

Table 7.12 Results of IS algorithms when using stratification - SOREL-20M

The experimental results of IS algorithms in reducing the entire training set of the SOREL-20M dataset using stratification are presented in Table 7.12. In terms of the $M_{AccSize}$ metric, the algorithm RPIF-AllKNN achieved the best result with a value of 14.43. In this case, it managed to outperform RPIF-AllKNN_2 both in the achieved level of reduction and in classification accuracy. Among the algorithms with the best $M_{AccSize}$ values were the remaining versions of the RPIF algorithm and the DROP3 algorithm. These algorithms, however, performed the worst according to $M_{AccSize}$. MSS, on the other hand, was the best algorithm in terms of achieved classification accuracy and F1 score, with a loss compared to the original accuracy and F1 score not exceeding 0.5%. The low $M_{AccSize}$ value for the MSS algorithm was due to the lowest achieved level of reduction among non-edition IS algorithms, with a size of 20.12% compared to the original training set. "Non-edition" is the term used for hybrid and condensation algorithms. The second-best algorithm in terms of accuracy and F1 score was the DROP3 algorithm, which also ranked among the best in reducing the training set, with a reduced set size of 7.64%. The lowest size of 6.24% was achieved by the RPIF-AllKNN algorithm. The CNN algorithm achieved the lowest accuracy, and among non-edition algorithms, RPIF-RENN_2 obtained the lowest F1 score. The reduction of subsets created through stratification proceeded gradually. In the case of parallel algorithms, parallelization was applied to individual subsets. This processing method disadvantaged mainly the DROP3 and CNN algorithms. For these two algorithms, lower computational times can be expected if the subsets

were reduced concurrently. DROP3 was unequivocally the slowest algorithm with a computation time of 366067.1 seconds. The ENN algorithm was the fastest, managing to reduce the training set by 219.1 seconds. Excluding edition algorithms, PIF was the fastest with a time of 6883.5 seconds.

Considering computational times, the reduction of the entire training set with a size of 6926181 using stratification took less time for the RENN, ICF, PIF, PIF-AllKNN, PIF-RENN, RPIF, RPIF-AllKNN, RPIF-RENN, RPIF_2, RPIF-AllKNN_2, and RPIF-RENN_2 algorithms than when reducing a set of size 1000k without stratification. In some cases, the reduction time even approached the computational times of reducing a set of size 750k. MSS and DROP3 algorithms were processed without stratification only up to a size of 300k. For the remaining algorithms, reduction took longer with stratification than without stratification for a set of size 1000k. The remaining metric values achieved with stratification (referred to as 'with stratification' reduction) are compared with the metric values achieved when reducing sets of size 1000k without stratification (referred to as 'without stratification' reduction), with the exception of MSS and DROP3 algorithms, where metric values for 'without stratification' reduction are related to a set size of 300k. Unlike 'without stratification' reduction, in the case of 'with stratification' reduction, none of the algorithms achieved a decrease in accuracy of more than 5% compared to the original accuracy. For edition algorithms and the CNN algorithm, classification accuracy was lower in the case of 'with stratification' reduction, while the remaining algorithms achieved improvements. For metrics $M_{AccSize}$ and Size, the comparison results are similar. For edition algorithms, 'with stratification' reduction was better, while the remaining algorithms achieved better results in the case of reduction 'without stratification.' However, it is necessary to emphasize that the reported Size values are expressed as percentages relative to the original subset size. The training set size was almost seven times larger than the subset size of 1000k. This means that in all cases, the sizes of the reduced sets when applying 'with stratification' reduction were several times bigger compared to the sizes when applying 'without stratification' reduction. If we do not consider edition algorithms, when using stratification, an improvement in classification accuracies was achieved with similar computational times at the cost of deterioration of $M_{AccSize}$ and several times larger reduced sets. An exception is the CNN algorithm, which also deteriorated in terms of achieved accuracy.

Conclusion

Within this thesis, two publicly available datasets, EMBER and SOREL-20M, containing metadata of PE binary files were processed. The parsed data from the databases underwent preprocessing. Initially, a cleaning process took place, involving the imputation of missing values with constant values, removal of duplicates, constant features, and features containing a unique value for each instance, as well as the detection and subsequent replacement of outliers. In the case of the SOREL-20M dataset, values detected by the IQR method were replaced with the mean, while the EMBER dataset achieved better results without outlier handling. After the data cleaning phase, the transformation of categorical features into numerical ones occurred. One-hot encoding and feature hashing were used for this purpose. Subsequently, feature scaling was applied to all features in both datasets. For the EMBER dataset, feature values were transformed using min-max normalization, while robust scaling was chosen for the SOREL-20M dataset. Feature extraction was performed in both cases using the PCA method. The aforementioned methods were applied through scripts created in the Python programming language, utilizing available libraries.

For the purposes of experiments with instance selection algorithms, eight state-of-the-art algorithms were used. In the case of edition algorithms, these were ENN, RENN, and AllKNN. The category of condensation algorithms was represented by CNN and MSS, while the hybrid algorithms chosen for experiments were ICF, DROP3, and PIF. A custom implementation was created for all eight algorithms. Python was the main programming language, but some parallel and computationally intensive parts were implemented in the C programming language. The experiments also included modifications to the PIF algorithm. The first modification involved repeated filtration of subsets created based on the nearest enemies of elements. The second modification replaced the edition algorithms, specifically the AllKNN and RENN algorithms. This resulted in five modified versions of the PIF algorithm, namely PIF-AllKNN, PIF-RENN, RPIF, RPIF-AllKNN, and RPIF-RENN. The modified versions were also implemented in Python, combined with the C programming language for computationally intensive parts. The parameters of the instance selection algorithms were set based on experiments.

The main goal of this thesis was to create or modify existing instance selection algorithms and compare them with existing state-of-the-art algorithms. The findings from the comparison of state-of-the-art algorithms are summarized in the following paragraph.

Edition algorithms were unequivocally the fastest among the compared IS algorithms. They were also among the best in terms of classification accuracy achieved on reduced datasets. However, they were unequivocally the worst in terms of reduction capability and, consequently, in the achieved values of the metric $M_{AccSize}$. In the case of condensation algorithms, the level of reduction and values of $M_{AccSize}$ were higher than for edition algorithms but lower compared to hybrid algorithms. For the EMBER dataset, CNN algorithm was the best in terms of accuracy, while for the SOREL-20M dataset, the achieved accuracy was comparable to MSS and hybrid algorithms. In terms of computational times, the CNN algorithm was the second slowest, and with increasing sizes of reduced sets, the difference in reduction speed compared to faster algorithms grew. Computational times of the MSS algorithm were comparable to ICF and PIF algorithms at smaller sizes. However, from the size of the reduced set of 500k, the MSS implementation encountered a memory shortage issue. The best in terms of $M_{AccSize}$ and reduction level up to size 300k was the hybrid DROP3 algorithm. However, in terms of computational time, it was significantly slower compared to other algorithms, with computation times several times higher. The remaining two hybrid algorithms, ICF and PIF, achieved lower reduction and $M_{AccSize}$ values compared to DROP3 algorithms, but due to greater parallelization, they were better in terms of computational times. With increasing sizes, the PIF algorithm became the fastest among hybrid and condensation algorithms, and considering its full parallelization, this trend is expected to continue.

All modifications of the PIF algorithm, thanks to a higher degree of set reduction, achieved higher values of $M_{AccSize}$ compared to the original version. However, as the sizes of the reduced sets decreased, the classification accuracies of the new versions also decreased. Modified algorithms can be divided into two groups. The first group consists of algorithms PIF-AllKNN and PIF-RENN, while the second group is formed by algorithms RPIF, RPIF-RENN, and RPIF-AllKNN. The first group achieved lower values of $M_{AccSize}$ compared to the second group, but the losses in classification accuracy compared to non-reduced sets were lower. Algorithms from the second group were among the overall best in terms of $M_{AccSize}$ and reduction level; however, there was a greater loss in classification accuracy compared to the first group. Specifically, the RPIF-AllKNN algorithm was able to reduce sets to the smallest sizes, thus achieving the highest values of $M_{AccSize}$. However, in terms of accuracy, it ranked among the worst. The computation time was extended the most for versions that used the RENN algorithm for the initial filtration.

In conclusion, the created modifications of the PIF algorithm were able to achieve greater reduction, although at the cost of a loss in accuracy. According to the $M_{AccSize}$ metric, the results can be considered better; however, if preserving higher accuracy is more important for someone, then $M_{AccSize}$ may not sufficiently reflect this requirement. Future work related to this issue could therefore involve introducing a penalty that takes into account the loss of accuracy on reduced sets. Further experiments could explore the replacement of the editing method in other hybrid algorithms. Regarding the PIF algorithm, experiments could involve changing the filtration rule applied to subsets or experimenting with the application of multiple filtration rules simultaneously.

Appendix A

Graphs for the tables from Chapter 7

This appendix contains graphs related to the tables from Chapter 7. These tables include the results of evaluating IS algorithms on datasets of various sizes, without using stratification. The graphs have been placed in the appendix due to their size.

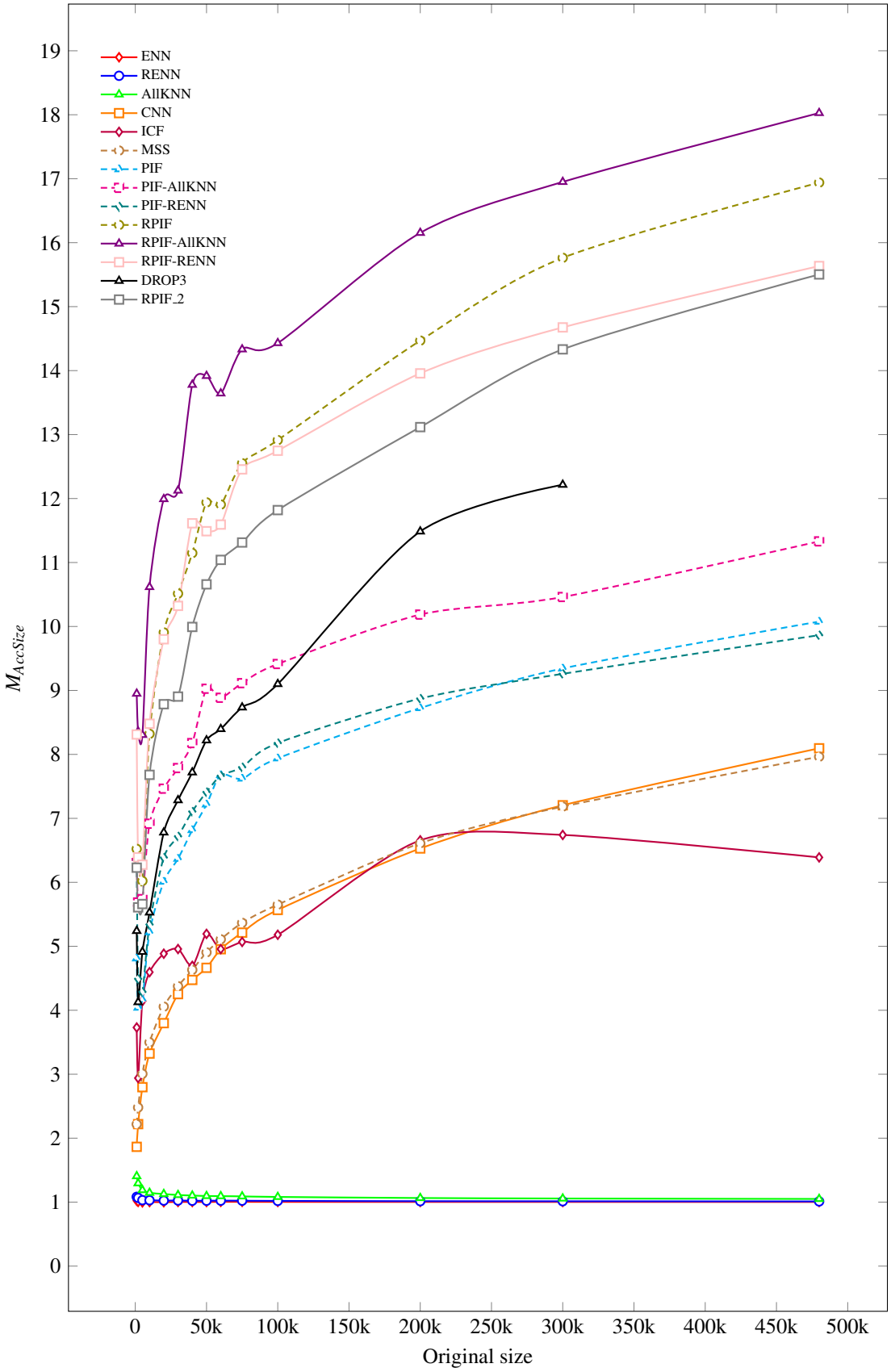

Figure A.1 Values of $M_{AccSize}$ achieved by IS algorithms depending on the size of the reduced set - EMBER

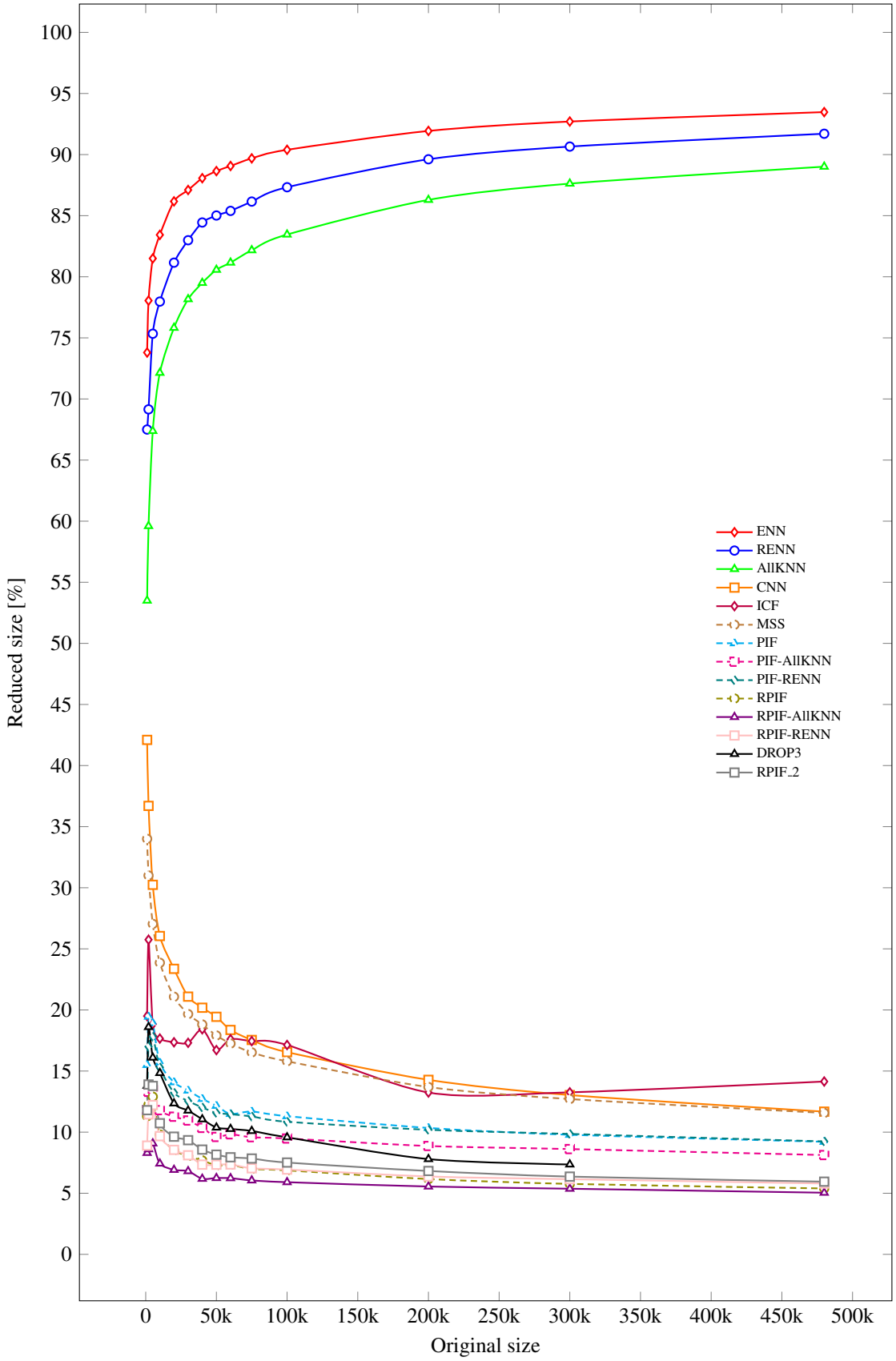

Figure A.2 Sizes of reduced sets depending on sizes of original sets - EMBER

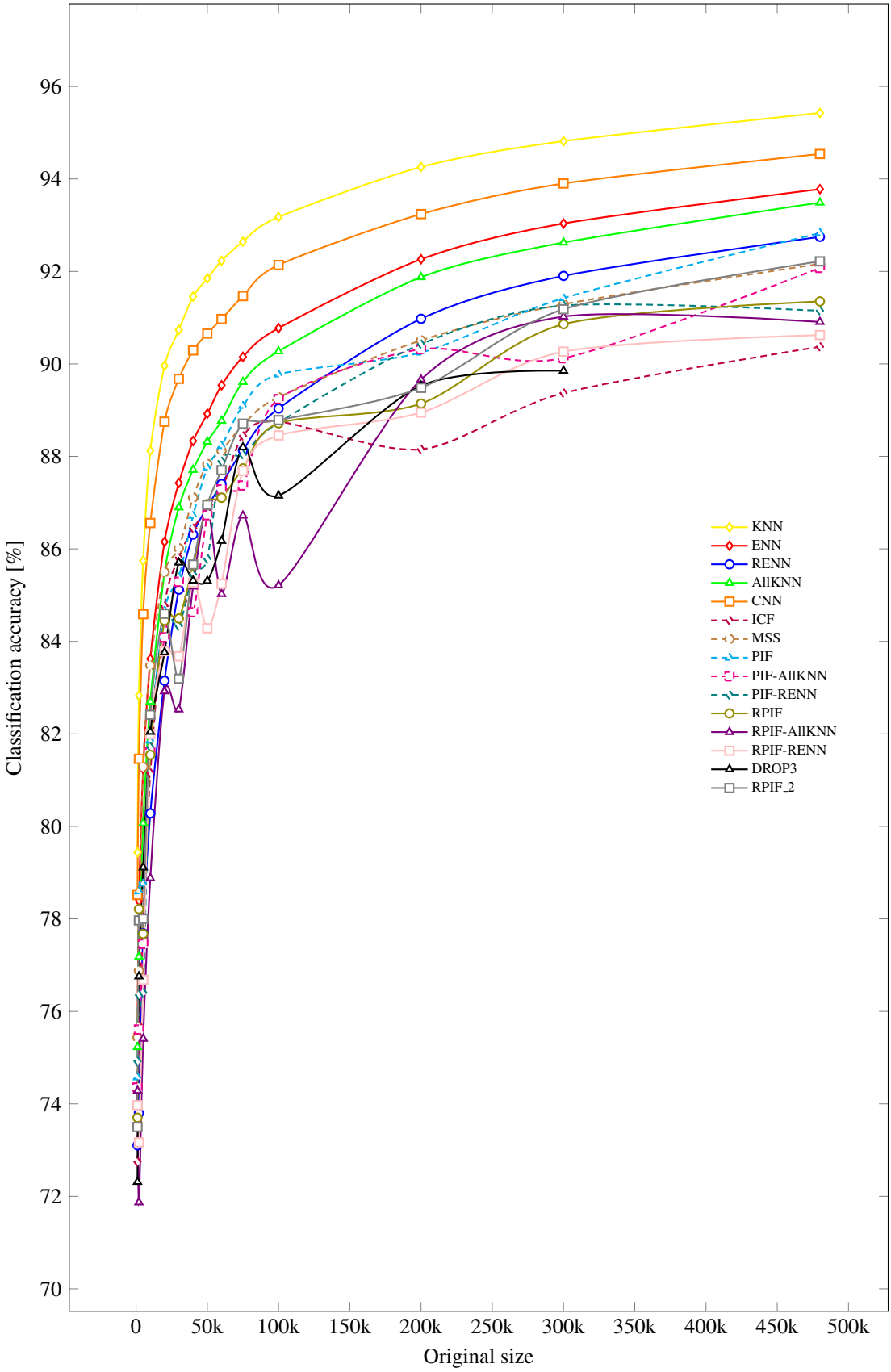

Figure A.3 Classification accuracies depending on the sizes of original sets - EMBER

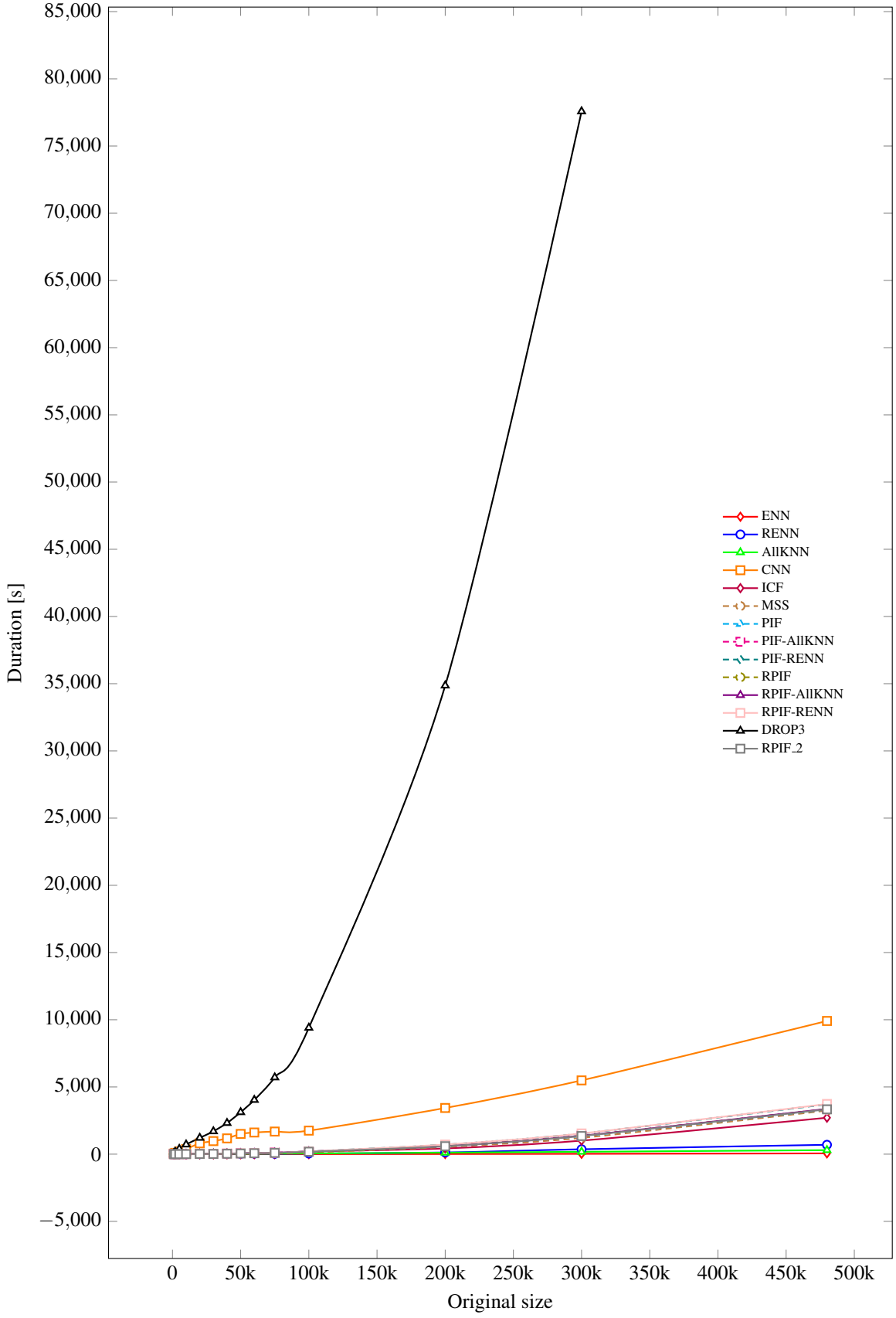

Figure A.4 Run times of IS algorithms depending on the sizes of the original sets. - EMBER

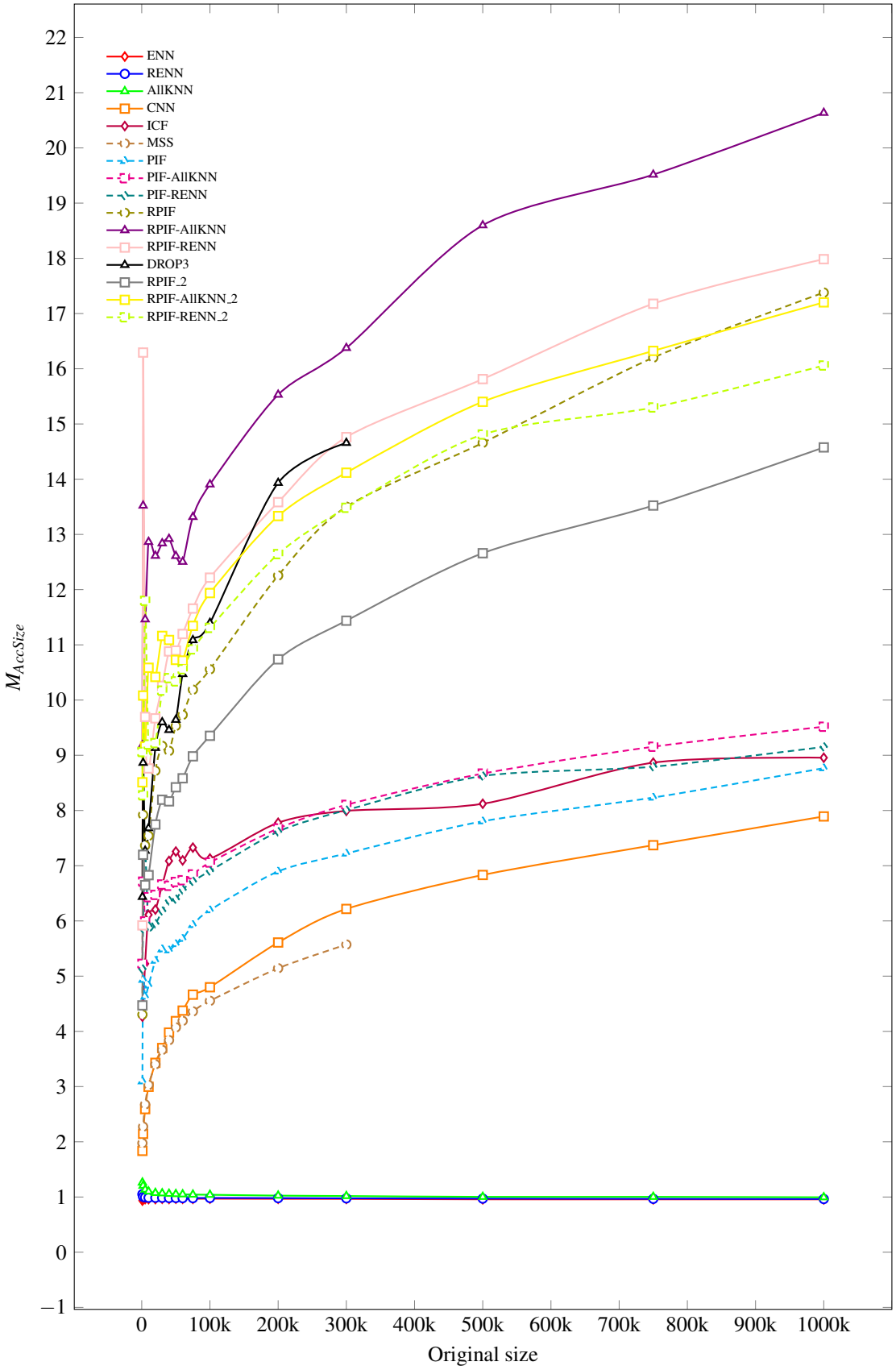

Figure A.5 Values of $M_{AccSize}$ achieved by IS algorithms depending on the size of the reduced set - SOREL-20M

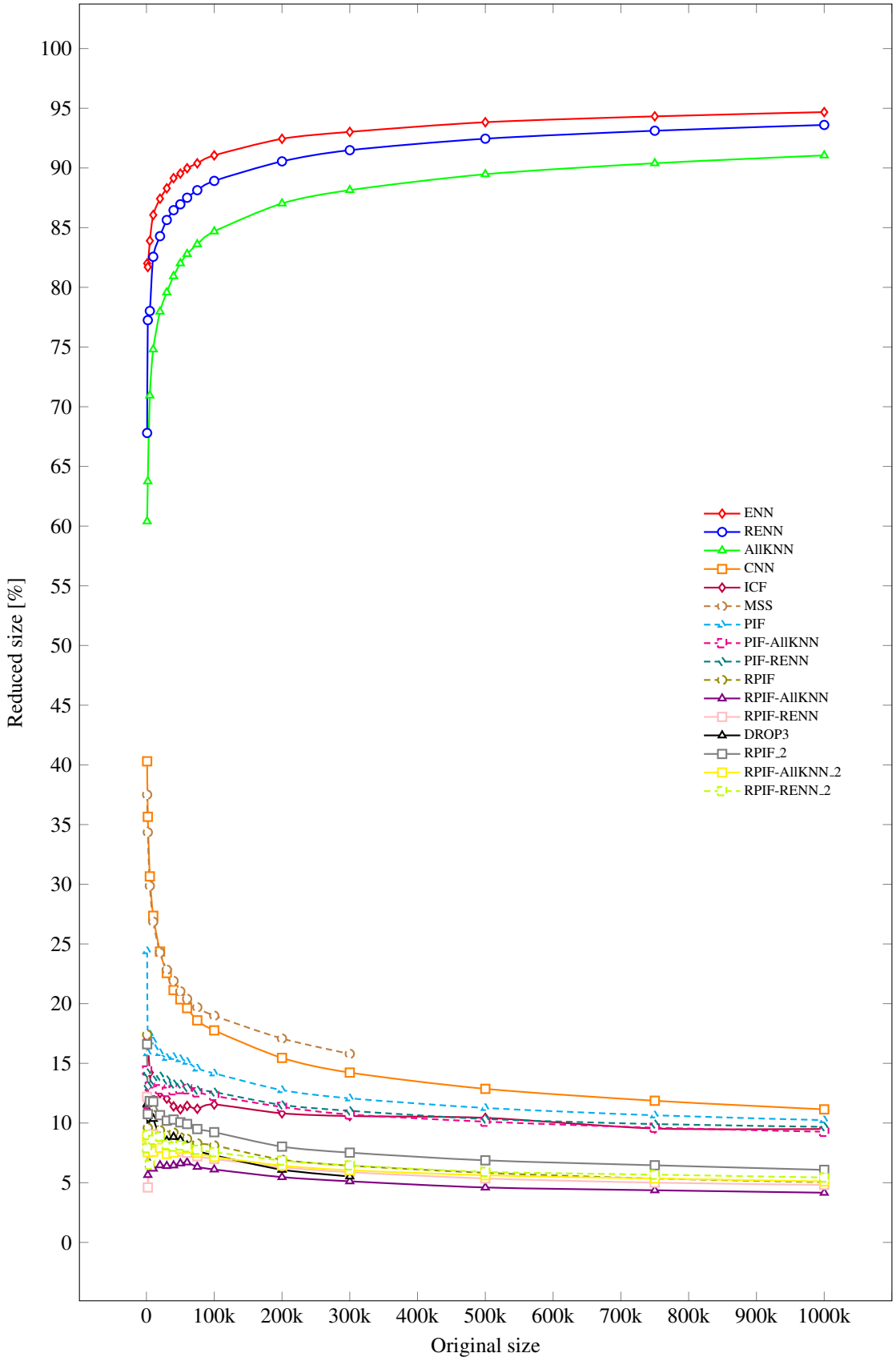

Figure A.6 Sizes of reduced sets depending on sizes of original sets - SOREL-20M

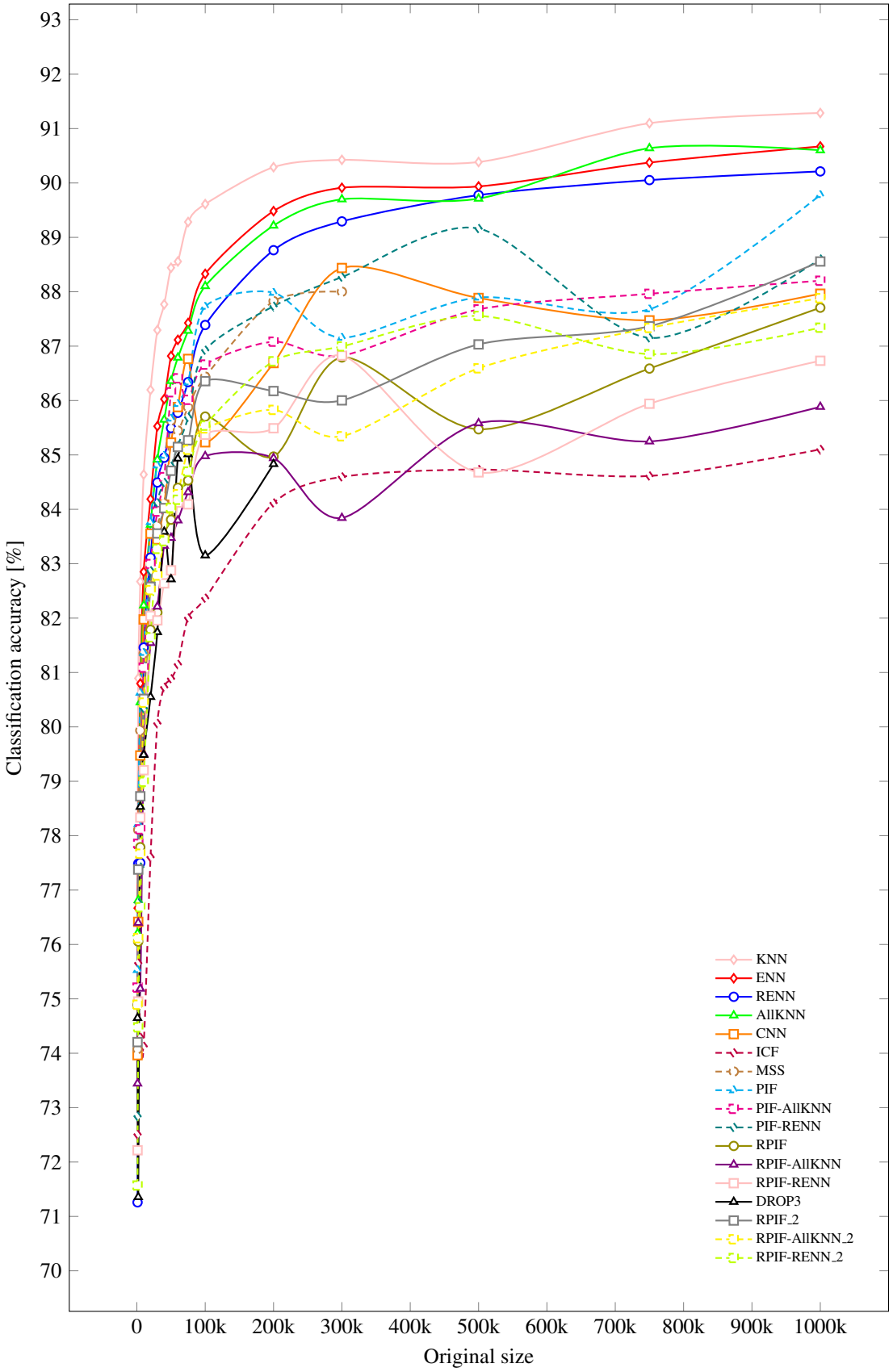

Figure A.7 Classification accuracies depending on the sizes of original sets - SOREL-20M

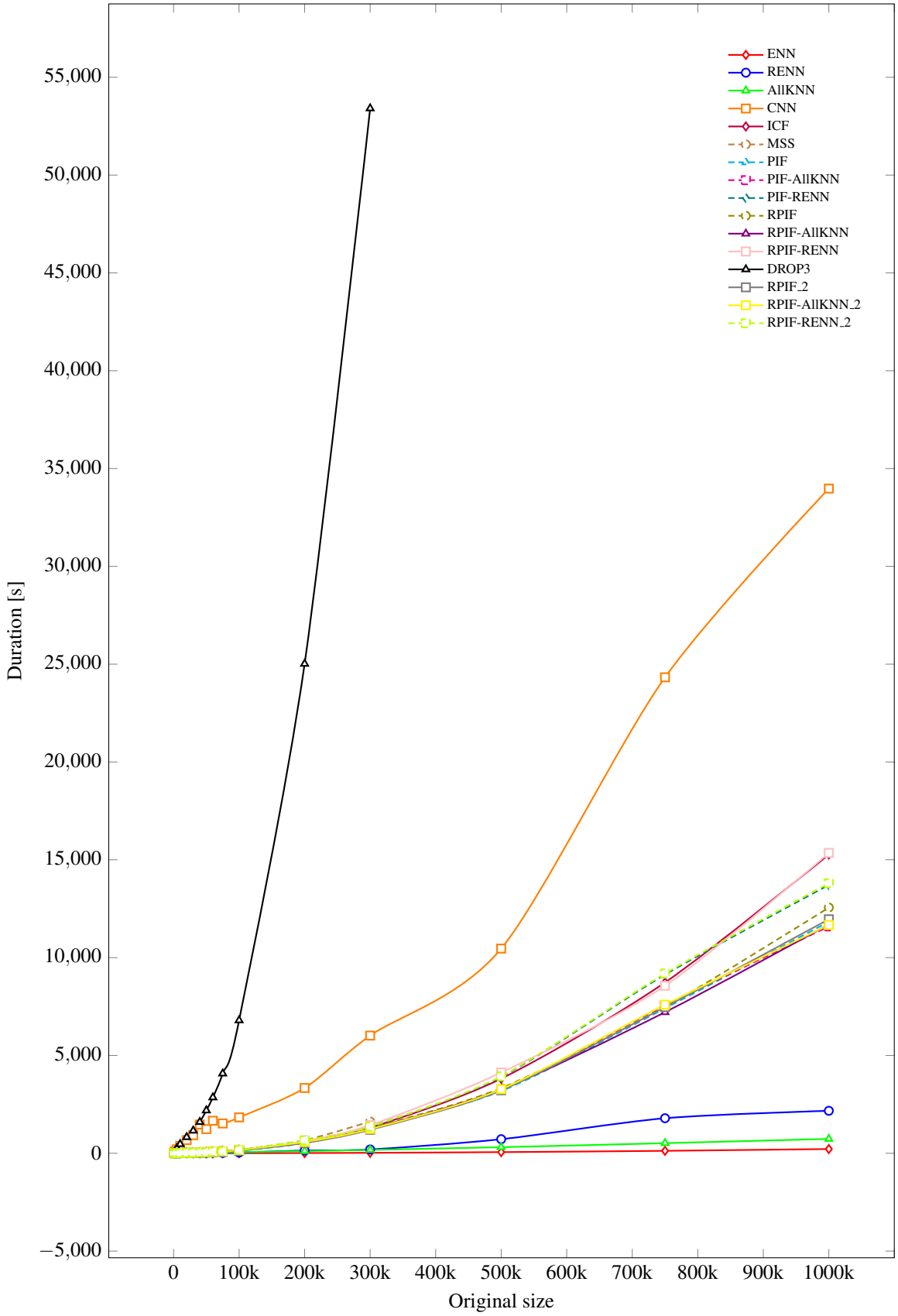

Figure A.8 Run times of IS algorithms depending on the sizes of the original sets - SOREL-20M

Appendix B

Description of the attached files

This page describes only the content of the root directory. This directory is further divided into subdirectories. Each subdirectory contains its own file **description.txt**, where its content is described.

`Ded_masters_thesis_appendices.zip` ..... root directory
- `1_data_parsing` ..... downloading datasets and parsing into CSV files
- `2_data_preprocessing` ..... preprocessing datasets before feature extraction
- `3_feature_selection` ..... feature extraction (PCA)
- `4_instance_selection` ... parameters tuning of IS algorithms and their subsequent comparison
- `description.txt` ..... a brief description of the content of the directory

Bibliography

1. WATTERS, Ashley. *25 Crucial Information Technology Statistics and Facts to Know* [online]. [N.d.]. Available also from: https://connect.comptia.org/blog/information-technology-stats-facts. Accessed: 2024-01-15.
2. JAIN, Prati; RAJVAIDYA, Ishita; SAH, Keshav Kumar; KANNAN, Jayanthi. Machine Learning Techniques for Malware Detection- a Research Review. In: *2022 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*. 2022, pp. 1–6. Available from DOI: 10.1109/SCEECS54111.2022.9740918.
3. PALATTY, Nivedita James. *30+ Malware Statistics You Need To Know In 2024* [online]. [N.d.]. Available also from: https://www.getastra.com/blog/security-audit/malware-statistics/. Accessed: 2024-01-15.
4. AGARKAR, Sanket; GHOSH, Soma. Malware Detection Classification using Machine Learning. In: *2020 IEEE International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC)*. 2020, pp. 1–6. Available from DOI: 10.1109/iSSSC50941.2020.9358835.
5. OLVERA-LÓPEZ, José; CARRASCO-OCHOA, Jesús; MARTÍNEZ-TRINIDAD, José Francisco; KITTLER, Josef. A review of instance selection methods. *Artif. Intell. Rev.* 2010, vol. 34, pp. 133–143. Available from DOI: 10.1007/s10462-010-9165-y.
6. ANDERSON, H. S.; ROTH, P. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*. 2018. Available from DOI: 10.48550/arXiv.1804.04637.
7. HARANG, Richard; RUDD, Ethan M. *SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection*. 2020. Available from arXiv: 2012.07634 [cs.CR].
8. KUMAR, Virender; KHOSLA, Cherry. Data Cleaning-A Thorough Analysis and Survey on Unstructured Data. In: *2018 8th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. 2018, pp. 305–309. Available from DOI: 10.1109/CONFLUENCE.2018.8442950.

9. RIDZUAN, Fakhitah; WAN ZAINON, Wan Mohd Nazmee. A Review on Data Cleansing Methods for Big Data. *Procedia Computer Science*. 2019, vol. 161, pp. 731–738. ISSN 1877-0509. Available from DOI: https://doi.org/10.1016/j.procs.2019.11.177. The Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia.
10. KANG, Hyun. The prevention and handling of the missing data. *Korean journal of anesthesiology*. 2013, vol. 64, pp. 402–6. Available from DOI: 10.4097/kjae.2013.64.5.402.
11. DASU, Tamraparni; JOHNSON, Theodore. *Exploratory Data Mining and Data Cleaning*. ATT Labs, Research Division Florham Park, NJ: Wiley-Interscience, 2003. ISBN 978-0471268512.
12. SESSA, Jadran; SYED, Dabeeruddin. Techniques to deal with missing data. In: *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*. 2016, pp. 1–4. Available from DOI: 10.1109/ICEDSA.2016.7818486.
13. PHAN, Quoc-Thang; WU, Yuan-Kang; PHAN, Quoc-Dung; LO, Hsin-Yen. A Study on Missing Data Imputation Methods for Improving Hourly Solar Dataset. In: *2022 8th International Conference on Applied System Innovation (ICASI)*. 2022, pp. 21–24. Available from DOI: 10.1109/ICASI55125.2022.9774453.
14. BROWNLEE, Jason. *Data Preparation for Machine Learning*. 2020.
15. KUHN, Max; JOHNSON, Kjell. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742, USA: Chapman and Hall/CRC, 2019. ISBN 978-1138079229.
16. HAN, Jiawei; KAMBER, Micheline; PEI, Jian. *Data Mining: Concepts and Techniques*. 500 Sansome Street, Suite 400, San Francisco, CA 94111, USA: Morgan Kaufmann, 2011. ISBN 978-9380931913.
17. ILYAS, Ihab F.; CHU, Xu. *Data Cleaning*. New York, NY, United States: Association for Computing Machinery, 2019. ISBN 978-1-4503-7152-0.
18. BELICHOVSKI, Martin; STAVROV, Dushko; DONCHEVSKI, Filip; NADZINSKI, Gorjan. Unsupervised Machine Learning Approach for Anomaly Detection in E-coating Plant. In: *2022 IEEE 17th International Conference on Control Automation (ICCA)*. 2022, pp. 992–997. Available from DOI: 10.1109/ICCA54724.2022.9831858.
19. BENKERT, Katharina; GABRIEL, Edgar; RESCH, Michael M. Outlier detection in performance data of parallel applications. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–8. Available from DOI: 10.1109/IPDPS.2008.4536463.
20. CERDA, Patricio; VAROQUAUX, Gaël; KÉGL, Balázs. *Similarity encoding for learning with dirty categorical variables*. 2018. Available from arXiv: 1806.00979 [cs.LG].
21. KUHN, Max; JOHNSON, Kjell. *Applied Predictive Modeling*. New York, USA: Springer, 2013. ISBN 978-1461468486.
22. SCIKIT-LEARN. *LabelEncoder* [online]. [N.d.]. Available also from: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html. Accessed: 2023-04-03.

23. WEINBERGER, Kilian; DASGUPTA, Anirban; ATTENBERG, Josh; LANGFORD, John; SMOLA, Alex. Feature Hashing for Large Scale Multitask Learning. 2009. Available from DOI: 10.1145/1553374.1553516.

24. CAO, Xi Hang; STOJKOVIC, Ivan; OBRADOVIC, Zoran. A robust data scaling algorithm to improve classification accuracies in biomedical data. *BMC bioinformatics*. 2016, vol. 17, p. 359. Available from DOI: 10.1186/s12859-016-1236-x.

25. FERREIRA, Pedro; C. LE, Duc; ZINCIR-HEYWOOD, Nur. Exploring Feature Normalization and Temporal Information for Machine Learning Based Insider Threat Detection. In: *2019 15th International Conference on Network and Service Management (CNSM)*. 2019, pp. 1–7. Available from DOI: 10.23919/CNSM46954.2019.9012708.

26. AMORIM, Lucas B.V. de; CAVALCANTI, George D.C.; CRUZ, Rafael M.O. The choice of scaling technique matters for classification performance. *Applied Soft Computing*. 2023, vol. 133, p. 109924. ISSN 1568-4946. Available from DOI: 10.1016/j.asoc.2022.109924.

27. GHOJOGH, Benyamin; SAMAD, Maria N.; MASHHADI, Sayema Asif; KAPOOR, Tania; ALI, Wahab; KARRAY, Fakhri; CROWLEY, Mark. Feature Selection and Feature Extraction in Pattern Analysis: A Literature Review. *CoRR*. 2019, vol. abs/1905.02845. Available from arXiv: 1905.02845.

28. SONG, Fengxi; GUO, Zhongwei; MEI, Dayong. Feature Selection Using Principal Component Analysis. In: *2010 International Conference on System Science, Engineering Design and Manufacturing Informatization*. 2010, vol. 1, pp. 27–30. Available from DOI: 10.1109/ICSEM.2010.14.

29. ALKANDARI, Abdulrahman; ALJABER, Soha Jaber. Principle Component Analysis algorithm (PCA) for image recognition. In: *2015 Second International Conference on Computing Technology and Information Management (ICCTIM)*. 2015, pp. 76–80. Available from DOI: 10.1109/ICCTIM.2015.7224596.

30. MURPHY, Kevin P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 978-0262018029.

31. SEHGAL, Shruti; SINGH, Harpreet; AGARWAL, Mohit; BHASKER, V.; SHANTANU. Data analysis using principal component analysis. In: *2014 International Conference on Medical Imaging, m-Health and Emerging Communication Systems (MedCom)*. 2014, pp. 45–48. Available from DOI: 10.1109/MedCom.2014.7005973.

32. JUREČEK, Martin; JUREČKOVÁ, Olha. Parallel Instance Filtering for Malware Detection. 2022. Available from DOI: 10.48550/arXiv.2206.13889.

33. MALHAT, Mohamed; MENSHAWY, Mohamed El; MOUSA, Hamdy; SISI, Ashraf El. A new approach for instance selection: Algorithms, evaluation, and comparisons. *Expert Systems with Applications*. 2020, vol. 149, p. 113297. ISSN 0957-4174. Available from DOI: https://doi.org/10.1016/j.eswa.2020.113297.

34. WILSON, D.; MARTINEZ, Tony. Instance Pruning Techniques. In: 1997, pp. 403–411.

35. HART, P. The condensed nearest neighbor rule (Corresp.) *IEEE Transactions on Information Theory*. 1968, vol. 14, no. 3, pp. 515–516. Available from DOI: 10.1109/TIT.1968.1054155.

36. BINIAZ, Ahmad; CABELLO, Sergio; CARMI, Paz; CARUFEL, Jean-Lou De; MAHESHWARI, Anil; MEHRABI, Saeed; SMID, Michiel. On the Minimum Consistent Subset Problem. 2018. Available from DOI: 10.48550/arXiv.1810.09232.

37. GARCÍA, Salvador; LUENGO, Julián; HERRERA, Francisco. *Data Preprocessing in Data Mining*. Springer, 2014. ISBN 978-3319102467.

38. BARANDELA, Ricardo; FERRI, Francesc; SÁNCHEZ, Josep. Decision boundary preserving prototype selection for nearest neighbor classification. *IJPRAI*. 2005, vol. 19, pp. 787–806.

39. WILSON, D.; MARTINEZ, Tony. Reduction Techniques for Instance-Based Learning Algorithms. *Machine Learning*. 2000, vol. 38, pp. 257–286. Available from DOI: 10.1023/A:1007626913721.

40. WILSON, Dennis L. Asymptotic Properties of Nearest Neighbor Rules Using Edited Data. *IEEE Transactions on Systems, Man, and Cybernetics*. 1972, vol. SMC-2, no. 3, pp. 408–421. Available from DOI: 10.1109/TSMC.1972.4309137.

41. TOMEK, Ivan. An Experiment with the Edited Nearest-Neighbor Rule. *IEEE Transactions on Systems, Man, and Cybernetics*. 1976, vol. SMC-6, no. 6, pp. 448–452. Available from DOI: 10.1109/TSMC.1976.4309523.

42. OLVERA-LÓPEZ, José; CARRASCO-OCHOA, Jesús; MARTÍNEZ-TRINIDAD, José Francisco; KITTLER, Josef. A review of instance selection methods. *Artif. Intell. Rev.* 2010, vol. 34, pp. 133–143. Available from DOI: 10.1007/s10462-010-9165-y.

43. BRIGHTON, Henry; MELLISH, Chris. On the Consistency of Information Filters for Lazy Learning Algorithms. In: 1999, pp. 283–288. ISBN 978-3-540-66490-1. Available from DOI: 10.1007/978-3-540-48247-5_31.

44. COVER, T.; HART, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*. 1967, vol. 13, no. 1, pp. 21–27. Available from DOI: 10.1109/TIT.1967.1053964.

45. CUNNINGHAM, Padraig; DELANY, Sarah. k-Nearest neighbour classifiers. *Mult Classif Syst*. 2007, vol. 54. Available from DOI: 10.1145/3459665.

46. JAPKOWICZ, Nathalie; SHAH, Mohak. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011. ISBN 978-0521196000.

47. OBI, Jude. A Comparative Study of Several Classification Metrics and Their Performances on Data. *World Journal of Advanced Engineering Technology and Sciences*. 2023, vol. 8, pp. 308–314. Available from DOI: 10.30574/wjaets.2023.8.1.0054.

48. ARYAL, Kshitiz; GUPTA, Maanak; ABDELSALAM, Mahmoud. Exploiting Windows PE Structure for Adversarial Malware Evasion Attacks. In: *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. Charlotte, NC, USA: Association for Computing Machinery, 2023, pp. 279–281. CODASPY '23. ISBN 9798400700675. Available from DOI: 10.1145/3577923.3585044.

49. FANG, Yong; ZENG, Yuetian; LI, Beibei; LIU, Liang; ZHANG, Lei. DeepDetectNet vs RLAttackNet: An adversarial method to improve deep learning-based static malware detection model. *PLOS ONE*. 2020, vol. 15, e0231626. Available from DOI: 10.1371/journal.pone.0231626.

50. *PE Format* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed: 2023-09-14.

51. *A dive into the PE file format - PE file structure - Part 2: DOS Header, DOS Stub and Rich Header* [online]. [N.d.]. Available also from: https://0xrick.github.io/win-internals/pe3/. Accessed: 2023-09-14.

52. NISI, Dario; GRAZIANO, Mariano; FRATANTONIO, Yanick; BALZAROTTI, Davide. Lost in the Loader:The Many Faces of the Windows PE File Format. In: San Sebastian, Spain: Association for Computing Machinery, 2021, pp. 177–192. RAID '21. ISBN 9781450390583. Available from DOI: 10.1145/3471621.3471848.

53. *A dive into the PE file format - PE file structure - Part 3: NT Headers* [online]. [N.d.]. Available also from: https://0xrick.github.io/win-internals/pe4/. Accessed: 2023-09-14.

54. REZAEI, Tina; MANAVI, Farnoush; HAMZEH, Ali. A PE header-based method for malware detection using clustering and deep embedding techniques. *Journal of Information Security and Applications*. 2021, vol. 60, p. 102876. ISSN 2214-2126. Available from DOI: https://doi.org/10.1016/j.jisa.2021.102876.

55. DUONG, Lai Van; XUAN, Cho Do. Detecting Malware based on Analyzing Abnormal behaviors of PE File. *International Journal of Advanced Computer Science and Applications*. 2021, vol. 12, no. 3. Available from DOI: 10.14569/IJACSA.2021.0120355.

56. KUMAR, Ajit; KUPPUSAMY, K.S.; AGHILA, G. A learning model to detect maliciousness of portable executable using integrated feature set. *Journal of King Saud University - Computer and Information Sciences*. 2019, vol. 31, no. 2, pp. 252–265. ISSN 1319-1578. Available from DOI: https://doi.org/10.1016/j.jksuci.2017.01.003.

57. DEVI, Dhruwajita; NANDI, Sukumar. PE File Features in Detection of Packed Executables. *International Journal of Computer Theory and Engineering*. 2012, pp. 476–478. Available from DOI: 10.7763/IJCTE.2012.V4.512.

58. CARRERA VENTURA, Ero. *pefile*. 2023. Version 2023.2.7. Available also from: https://github.com/erocarrera/pefile.

59. CANO, José; HERRERA, Francisco; LOZANO, Manuel. Stratification for scaling up evolutionary prototype selection. *Pattern Recognition Letters*. 2005, vol. 26, pp. 953–963. Available from DOI: 10.1016/j.patrec.2004.09.043.