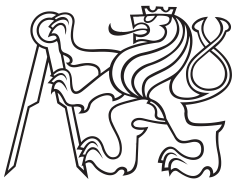
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

# Decision Forests in the Task of Semi-Supervised Learning

Bc. Jan Brabec

Supervisor: Ing. Lukáš Machlica, Ph.D
January 2017

**Czech Technical University in Prague**
**Faculty of Electrical Engineering**

**Department of Cybernetics**

# DIPLOMA THESIS ASSIGNMENT

**Student:** Bc. Jan B r a b e c

**Study programme:** Open Informatics

**Specialisation**: Computer Vision and Image Processing

**Title of Diploma Thesis:** Decision Forests in the Task of Semi-Supervised Learning

**Guidelines:**

1. Get familiar with decision forests and related classification techniques.
2. Prepare and get familiar with the training and testing data sets.
3. Study the recommended literature.
4. Propose and implement methods in a chosen programming language.
5. Evaluate the methods on real world data sets.

**Bibliography/Sources:**

[1] Criminisi A., et al.: Decision Forests for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning, Microsoft TechReport, 2011, available at: http://research.microsoft.com/apps/pubs/default.aspx?id=155552
[2] Lakshminarayanan B., et al.: Mondrian Forests: Efficient Online Random Forests, Advances in Neural Information Processing Systems 27 (NIPS), 2014
[3] Leistner Ch., et al.: Semi-Supervised Random Forests, IEEE 12th International Conference on Computer Vision, 2009
[4] Liu X., et al.: Semi-supervised Node Splitting for Random Forest Construction, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2013
[5] Geurts P., et al.: Extremely randomized trees, Machine Learning, Springer, 2006

**Diploma Thesis Supervisor:** Ing. Lukáš Machlica, Ph.D.

**Valid until:** the end of the winter semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
**Head of Department**

prof. Ing. Pavel Ripka, CSc.
**Dean**

Prague, May 26, 2016

# Acknowledgements

I would like to thank my supervisor Ing. Lukáš Machlica, Ph.D for his valuable guidance and advice. I would like to thank the CTA team at Cisco Systems, Inc. for providing me with the opportunity and resources to work on this project.

My thanks also goes to my family and to my girlfriend Vanda for all their support.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.


# Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date .....

.....
signature

# Abstract

We focus on the task of classification with random forests trained both on fully and on partially labeled data. Standard semi-supervised learning approaches cannot be applied on our problem because they usually assume that the unlabeled dataset is sampled from the same underlying distribution as the labeled dataset. In our case, the majority of objects in the unlabeled dataset belongs to a class that is not present in the labeled dataset at all. This problem has an application in network traffic classification, where the labeled dataset is made of objects that were associated with some category of malware and the objects in the unlabeled dataset are mostly benign (non-malware) traffic but there are also some malicious objects that we were unable to detect and label correctly. We implement and analyze several random forest types. They are analyzed both on publicly available datasets, that have been modified to contain imbalanced and unlabeled data, and also on our own network dataset that is composed from proxy logs. In addition, we adapt and implement several algorithms that can be used together with random forests in the above mentioned task and we show that they significantly improve the classification performance on public datasets.



**Keywords:** random forests, classification, semi-supervised learning, imbalanced data, mislabeled data, malware



**Supervisor:** Ing. Lukáš Machlica, Ph.D Cisco Systems, Inc., Charles Square Center, Karlovo Naměstí 10, 12000, Praha



# Abstrakt

Zabýváme se úlohou klasifikace s využitím náhodných lesů trénovaných jak na plně označených, tak i na částečně označených datech. Standardní přístupy k učení v přítomnosti neúplně označených dat nemohly být aplikovány na náš problém. Obvykle totiž předpokládají, že označená a neoznačená datová sada mají stejné pravděpodobnostní rozdělení. V našem případě většina objektů v neoznačené datové sadě patří do třídy, která se v označené datové sadě vůbec nevyskytuje. Tento problém se objevuje při klasifikaci síťového provozu, kde označená datová sada je tvořena objekty, které byly spojeny s některou z kategorií malwaru. Objekty v neoznačené datové sadě jsou většinou benigní (ne malware) provoz, ale také existují škodlivé objekty, které jsme nebyli schopni detekovat a správně označit. Implementujeme a analyzujeme několik druhů náhodných lesů. Jsou analyzovány na veřejně dostupných datových sadách, které byly upraveny, aby obsahovaly nevyvážená a neoznačená data, a také na naší síťové datové sadě, která je tvořena z proxy logů. Také jsme adaptovali a implementovali několik algoritmů, které mohou být použity společně s náhodnými lesy ve výše zmíněné úloze. Ukazujeme, že výrazně zvyšují kvalitu klasifikace na veřejně dostupných datových sadách.



**Klíčová slova:** náhodné lesy, klasifikace, kombinace učení s učitelem a bez učitele, nevyvážená data, chybně označená data, malware

**Překlad názvu:** Modely rozhodovacích lesů a jejich využití v úloze neúplně anotovaných dat

# Contents

# Chapter 1

# Introduction

In this thesis, we focus on the task of classification by using several random forest types and on implementing methods that improve their performance. We use random forests because they achieve state of the art results in various machine learning tasks and they fulfil our basic requirements such as the natural ability for multiclass classification, fast evaluation, and reasonable handling of imbalanced datasets. We begin with the standard supervised learning scenario where the algorithms are provided with a labeled dataset for training. In this scenario we compare different random forest types and it also serves as a baseline for the methods we introduce later. Then we focus on a more specific problem where the classes in the training dataset are imbalanced and in addition to labeled data the training dataset also contains unlabeled objects. The difference to the common semi-supervised learning problem, as it is usually understood, is that in our case the unlabeled data are not distributed in the same way as the labeled data. The majority of unlabeled data belongs to a class we call *negative* which is not present among the labeled data at all. The remaining part of the unlabeled objects belongs to classes present in the labeled data and we call these classes positive. The problem can also be reformulated that the data are not unlabeled but they are assigned the *negative* label instead. In that case, we call the objects in the negative class that should belong to some positive class *mislabeled*. We further use both formulations of this problem because each of them is useful in different contexts.

There are many applications of the problem above, because it occurs in situations where it is easier or cheaper to provide labels only for a subset of classes that are present in the data and there is a possibility that some of the objects that should have been labeled were missed.

The problem occurs in the task of classification of network traffic data for malware, where the training dataset is formed from proxy logs. Some of the objects are labeled as malware traffic by a human analyst or by leveraging various blacklists. The majority of remaining objects that are left unlabeled is benign (non-malware) traffic, but there are still some objects which should have been associated with a malware category but they escaped our detection. In addition, even the positive classes alone are greatly imbalanced.

The main contributions of this thesis are:

- We analyze the algorithms on imbalanced datasets and compare them in terms of precision and recall rather than just in accuracy.
- We successfully adapt the DAS-RF algorithm [32], which was originally proposed for standard semi-supervised learning problems, to handle the imbalanced and unlabeled problem.
- We implement the PU learning spy technique [38, 35] and use random forests internally instead of the original Naive Bayes classifier and apply it on multiclass problems that are different from text classification for which it was originally proposed.
- We perform detailed experiments with Mondrian forests [30] in supervised setting, comparing them to Extremely Randomized Trees [25] and Breiman forests [9] and in addition to their accuracy we also benchmark size of their models and discover a limitation that prevents them from being applied to big datasets.
- All of the methods are benchmarked in detail on well-known publicly available datasets as well as on our own network dataset which is more challenging.

The thesis is structured into several chapters. In Chapter 2 we briefly review the necessary background about classification, that is useful in the following chapters. We also explain the evaluation measures that are used in chapters with experiments.

Chapter 3 is about decision trees which are the major building block of random forests. We describe used notation and the *CART algorithm* [7] which is commonly used for growing of decision trees.

In Chapter 4 we go into detail about random forests. In addition to *Random Forests* that were described and analyzed by Leo Breiman [9] and are by far the most commonly used type of random forest, we also describe random forests known as *Extremely Randomized Trees* [25] and *Mondrian forests* [30].

Chapter 5 moves away from the supervised learning scenario, that was the focus of previous chapters, and considers the imbalanced and unlabeled problem. This does not prevent the application of supervised methods but it makes them less effective. This chapter introduces various methods that attempt to tackle this problem from different angles. The methods presented in this chapter can be used in conjunction with any type of random forest presented earlier or they can even be used together to increase the effect.

Chapter 6 contains experiments with different random forests and methods for handling mislabeled data introduced earlier. The algorithms are evaluated on various publicly available datasets. The chapter is divided to two large sections. In the first section all random forest types are evaluated on the datasets in their original form and compared in the supervised learning scenario. In the second section the datasets are modified to be imbalanced and contain mislabeled data in a similar way to our network dataset.

Chapter 7 contains the summary of experiments that we performed on our network dataset.

# Chapter 2
# Classification

*Random forests* belong to a family of algorithms known as *classifiers*. Classifiers deal with a problem of assigning data from a *domain of interest* to some discrete *classes*. This problem is called *classification* and is very common. An example is the task of classifying emails to classes SPAM or NOSPAM, or classification of image to the class corresponding to the letter which is in the image. If the *output variable* is a real number, such as in the task of prediction of temperature, and the prediction's penalty depends on the magnitude of the difference between the true value and the predicted value [42], then the problem is called *regression* instead of classification. Random forests can also be used for regression but it is not the focus of this thesis.

In the context of machine learning, classifiers are general algorithms that can be applied to multiple problems. To achieve this, classifiers do not use some specific domain knowledge to solve each problem, but rather use training data to learn the concepts in the domain. In this chapter we summarize the common approaches used in machine learning to build classifiers, and describe how classifiers are evaluated. Since this is a broad topic we restrict ourselves only to techniques that are relevant to the rest of the thesis.

## 2.1 Supervised learning

*Supervised learning* is the most common approach used to train classifiers. From now on, we assume that our data reside in an *instance space* $X = \mathbb{R}^d$. To get the raw data into this format we typically need to perform *feature extraction*. In the above mentioned example of letter classification the typical *features* extracted from each raw image can be: overall lightness, light pixel count, dark pixel count, number of horizontal edges, number of vertical edges, etc. For each image those features are then put together and they form a vector in the instance space. Instance space is also often called *feature space* and we will use these terms interchangeably. In this thesis we will not further deal with feature extraction and selection (the process of selecting the relevant features). If not stated otherwise, when we talk about the datasets and data objects we mean the data residing in the feature space and not the initial raw data.

In the task of supervised learning, our input consists of fully labeled dataset

$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i$ are vectors from the feature space $X$ and the $y_i$ are their corresponding class labels from the discrete class labels set $Y$. This dataset is split into *training dataset* and *testing dataset*. The classifier is trained on the training dataset and it's performance is evaluated on the testing dataset. Trained classifier is formally a mapping $f : X \mapsto Y$ and the concrete trained classifier is also sometimes called *hypothesis* or *model* depending on the context. When talking about classifier's parameters it is important to distinguish if we mean the parameters of a model that are fitted to the data from training dataset or if we mean the parameters of the method used for training. To disambiguate, the parameters of the method are often called *hyperparameters*. In the context of decision trees, a hyperparameter can be the maximum depth of a tree and parameters of a model are the trained nodes.

For the classifier to learn correctly, the training dataset has to be representative of the data that the classifier will encounter in the testing dataset and production environment. Formally, we assume that there is some underlying distribution $P_{XY}$ on $X \times Y$ [54] and that the objects in the datasets are drawn independently from this distribution. We also assume that the distribution will remain the same in the production environment. In practice these assumptions may not completely hold. The more they are broken the less likely the classifier is to work well out of the box and the classifier or the learning process in general would require some special modifications. This is exactly our case when we deal with the classification of network data, since the underlying distribution probably changes in time and due to the nature of the usage patterns of the network the single data objects both in the training and testing datasets are also not completely independent from each other.

When evaluating classifiers we need to define a *loss function* $L : Y \times Y \mapsto R$ which assigns different penalties to different errors [54]. This might be useful in the mentioned example of email classification, when classifying clean email as spam is way more costly than the other way round. In this thesis we assume that all errors have equal importance and use the usual loss function [54]:

$$L_{01}(y, y') = \begin{cases} 0 & \text{if } y = y' \\ 1 & \text{if } y \neq y' \end{cases} \tag{2.1}$$

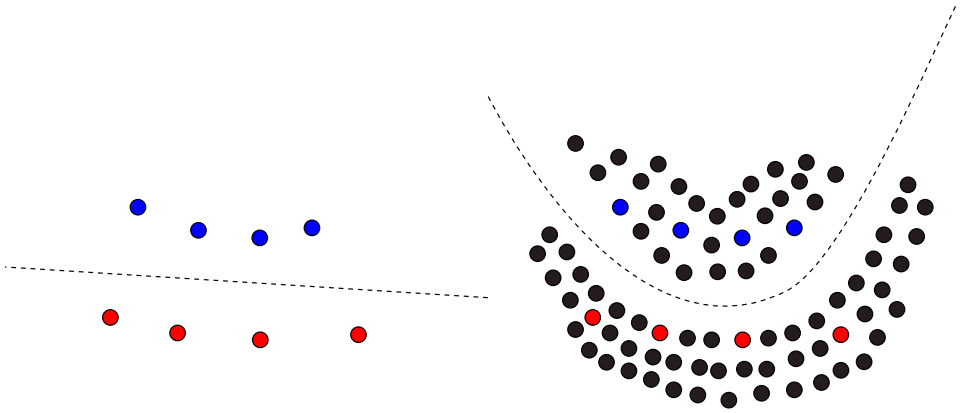
Because we are using the zero-one loss function we can simply define the classifier's error on a dataset as a fraction of points on which the label from the classifier and the true label disagree. More formally, for a dataset $D$ with $N$ objects the error of a trained classifier $f$ is:

$$err(f, D) = \frac{\sum_1^N L_{01}(f(\mathbf{x}_i), y_i)}{N} \tag{2.2}$$

The complementary metric to error is accuracy and it is defined as $acc(f, D) = 1 - err(f, D)$. If the error is computed on the training dataset it is called *training error* or *empirical error*. The error on the testing dataset is also called *generalization error* [27] or *classification error*.

## 2.2 Semi-supervised learning

In *semi-supervised learning* our input consists of labeled dataset and an additional unlabeled dataset. The goal is to use the unlabeled data in training to obtain a classifier with better performance than we would get if we only trained on labeled data. A simple reason why this might be possible is that the classifier is provided with more information than in the supervised learning case and it may therefore be able to make a more informed decision. Figure 2.1 shows an example where additional unlabeled objects improve classifier's performance.



**(a) :** labeled data only **(b) :** unlabeled data added

**Figure 2.1: Semi-supervised learning:** Addition of unlabeled data to the training phase can help the classifier make more informed decision.

The motivation for this approach is that it is often expensive and difficult to obtain labeled data because it often requires manual work, sometimes even from experts such as doctors or malware analysts. On the other hand, there might be abundance of unlabeled data available that would otherwise be left unused.

In most of semi-supervised learning algorithms, there is often an assumption that the labeled and unlabeled data are sampled from the same distribution and many methods are built on this assumption. This is often reasonable but in our case this assumption does not hold. In our case, when we are interested in identifying malware in network data, the labeled data are objects that were identified to belong in some of the malware classes. The unlabeled data are mostly benign, ordinary network traffic, but there might also be a small portion of malicious data mixed in, because it might have escaped our detection. This means that the unlabeled dataset contains a whole new class (benign ordinary network traffic) which does not even occur in the labeled dataset. This problem is known as *positive-unlabeled learning* (*PU learning*). The term first appeared in [34], but some work on the problem was published

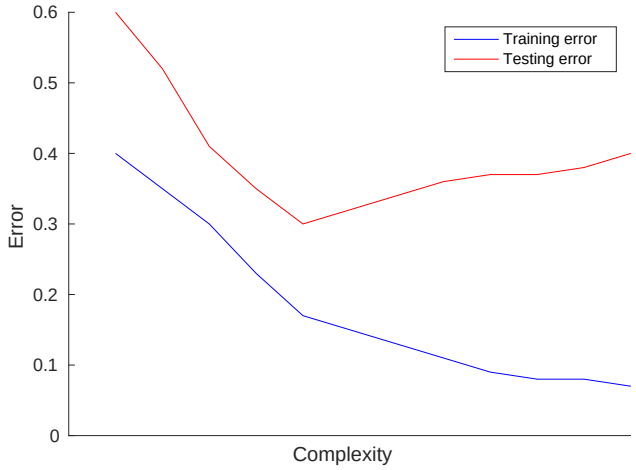



**Figure 2.2:** The typical behavior of the training error and testing error as a function of model's complexity.

earlier. PU learning is described in detail in Chapter 5.

## 2.3 Model complexity

Minimizing the error on the training dataset does not guarantee that the error on the testing dataset will also be minimal. This happens because of several reasons. First we have to realize that most of the real world data contains noise and maybe even some erroneous outliers. Overly complex models will start fitting on the noise in the training data at the expense of the underlying concepts. When this happens we say that the classifier *overfits* on the data. Also, since the training dataset is always finite it is also possible for sufficiently complex classifiers to simply memorize the correct labels for all the training data points. In fact, this is exactly what 1-NN classifier does and because of that, it is not a problem to have zero training error with 1-NN classifier.

On the other hand, when the model has too few parameters and it's complexity is too low, it is unable to fit on the training data. This is called *underfitting* and results in high training error and testing error.

In 2.2 we can see the typical behavior of the training error and testing error based on the classifier's complexity. We see that as we increase the complexity both errors decrease, but only to a certain point. After that the training error still decreases but the testing error starts to increase. We see that the error of a classifier can have two opposing causes when considering the classifiers complexity. This behavior is also known as *bias-variance tradeoff*. The exact definitions of bias and variance, and error decomposition to bias and variance can be found at: [18]. For the purposes of this thesis intuitive understanding of bias and variance is sufficient.

We can imagine a scenario when we repeat the training of the classifier many times. Each time the training dataset is randomly sampled from the

underlying distribution $P_{XY}$. If the classifier has high bias then the trained models from each experiment can be weak, but they will be very similar. On the other hand, if the classifier has high variance then the trained models will be very different from each other, because each of them will overfit to it's particular training set.

High variance is most problematic when the training dataset is small. As the training dataset's size grows we can generally allow to have more variance in our classifier because the datasets would be less different from each other and will more closely resemble the underlying distribution. Still, less complex models are generally more practical and easier to reason about, so when we have to choose from two competing models with similar generalization error the good rule of thumb is to pick the less complex one. This rule of thumb is an application of the well known principle known as Occam's razor.

The above mentioned interpretation of Occam's razor should not be confused with a different stronger interpretation that states that if two models have similar training error then the less complex one should be preferred because it is likely to generalize better. This interpretation has been rejected by Domingos in [17]. Empirical tests on thirty different public datasets were performed in [53] with results confirming Domingos' hypothesis on the 0.05 significance level that this strong interpretation of Occam's razor is false.

## 2.4 Evaluation measures

In Section 2.1 we mentioned classification error and accuracy as measures for measuring classifier's performance. While classification error is intuitive and common, it only offers a very high-level insight into the classifier's performance.

Furthermore, classification error does not work well for imbalanced datasets where some classes are rare. For example, if we wanted to build a classifier on patients' symptoms to test them for disease with prevalence of 0.01 % then the classifier could classify all of the patients as negative and it would have classification error only 0.01 %. The situation is similar in our problem, the classification of network data, where the data from interesting classes are also very rare. The following measures were invented for the binary classification problem. In binary classification there are only two classes of data. At first, we will introduce them in the binary classification context and at the end of this section we will show how we use them for multiclass classification.

At first, we want to calculate the confusion matrix. For now we assume that we are doing only binary classification and there are two classes: 'positive' and 'negative'. In the confusion matrix, the classification results are divided into the following categories:

- **TP**: True positives. Number of positive objects that were correctly classified as positive.
- **TN**: True negatives. Number of negative objects that were correctly classified as negative.

- **FP**: False positives. Number of negative objects that were incorrectly classified as positive.
- **FN**: False negatives. Number of positive objects that were incorrectly classified as negative.

These four measures are not that interesting by themselves, but they can be used to compute several other measures that provide greater insight into the performance of the classifier. Also, the classification error can be computed directly from the confusion matrix:

$$Classification\_Error = \frac{FP + FN}{FP + FN + TP + TN} \tag{2.3}$$

### 2.4.1 Precision

*Precision*, also sometimes called *positive predictive value*, is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

It is the fraction of predicted positive objects that are truly positive. In other words, it can also be interpreted as the probability that an object predicted as positive is truly positive, so it can be used to express the confidence of the classifier in positive predictions.

It is also interesting to note that precision is affected by the prevalence of the positive class. If the prevalence changes then precision might change accordingly. This follows from the observation that precision is a probability conditioned on the predicted class label and not on the true class label. This means that the measured precision is only meaningful if we assume that the prevalence will not change.

### 2.4.2 Recall

*Recall*, also sometimes called *sensitivity* or *true positive rate*, is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{2.5}$$

It is the fraction of truly positive objects that are correctly classified as positive. It gives us an insight into the ability of the classifier to find positive objects. Recall is not much useful measure by itself, because it is trivially possible to achieve 100 % recall by simply marking all of the objects as positive.

In contrast to precision, recall is a probability conditioned on the true class label and it is not affected when the prevalence of the positive class changes.

If no objects are positive then recall is undefined.

### 2.4.3 Specificity

*Specificity*, or *true negative rate*, is very similar to recall, but instead of positive objects it measures the percentage of negative objects that are correctly classified. It is defined as:

$$Specificity = \frac{TN}{TN + FP} \tag{2.6}$$

As with recall, it is trivially possible to achieve 100 % specificity by marking all of the objects as negative, the specificity of a classifier is not dependent on the prevalence, and if there are no negative objects in the dataset then specificity is undefined.

Complementary measure to specificity is called *fall-out* and it is defined as:

$$FallOut = 1 - Specificity \tag{2.7}$$

Fall-out measures the proportion of negative objects that are marked as positives.

### 2.4.4 ROC curve

In most non-trivial settings there is usually a trade-off between recall[1] and specificity. Improvement in recall might lead to performance decrease in specificity and otherwise. Most classifier algorithms have some way to implicitly or explicitly set thresholds determining if the classifier is preferring *false positives* or *false negatives*. The behavior of a classifier algorithm with different thresholds is commonly expressed by the *ROC curve*.

ROC curve is a graphical plot with fall-out on the horizontal axis and recall on the vertical axis. Because both fall-out and recall are independent of the class prevalences the ROC curve is also independent of the class prevalences. This is a nice property to have because it captures the intrinsic behavior of the classifier and is not affected if the classifier is used in a population with different distribution.

The curve starts at point $[0; 0]$, is monotonously increasing and ends at point $[1; 1]$. Point $[0; 0]$ describes a classifier which classifies all of the objects as negative and point $[1; 1]$ describes a classifier that classifies all of the objects as positive. The point $[0; 1]$ is an ideal point that would describe a classifiers that makes zero errors. The usual goal is to get as close to point $[0; 1]$ as possible. The overall performance of a classifier over all thresholds can also be expressed by the *area under curve* ($AUC$) where bigger area implies that the classifier performs better. ROC curve of a random classifier is the diagonal and it's AUC is 0.5. Perfect classifier that makes zero errors irrespective of the threshold has AUC equal to 1. Most of the classifiers will be somewhere in between.

[1] In the context of ROC analysis recall is almost exclusively called sensitivity.

### 2.4.5 Precision-Recall curve

Although ROC curve is used very commonly, it is not the best tool in our scenario. The reason is that it is not very informative when the data are severely imbalanced. This is best demonstrated by an example:

We will need the probabilistic interpretations of precision, recall and fall-out. In this example we denote positive sample as $p$ and positive prediction as $\tilde{p}$. Prevalence of positive objects is $P(p)$, precision is $P(p|\tilde{p})$, recall is $P(\tilde{p}|p)$ and fall-out is $P(\tilde{p}|\neg p)$.

A classifier in our example has recall 0.99 and fall-out only 0.01. It is very close to to ideal point $[0; 1]$ in the ROC space and it's AUC is almost 1. But what happens when the prevalence of the positive class is really low? We set the prevalence to be 0.001 and compute the precision with the Bayes' theorem.

$$P(p|\tilde{p}) = \frac{P(\tilde{p}|p)P(p)}{P(\tilde{p}|p)P(p) + P(\tilde{p}|\neg p)(1 - P(p))} \doteq 0.09 \tag{2.8}$$

We see that the precision is only 9 % even though recall and specificity are very high. Because of that we will use the precision-recall plot, where recall is on the horizontal axis and precision is on the vertical axis. In that plot the ideal point is $[1; 1]$ and even thought the classifier is close to optimal in the ROC space, we can see in the PR space that there is still very large room for improvement. PR curve is closely related to the ROC curve. It has been proven that if a curve dominates in ROC space it also dominates in PR space and vice versa [14].

Precision and recall are common measures used in machine learning when dealing with imbalanced datasets. There is some criticism of this practice in [43]. The first major point is that precision and recall ignore performance on the negative class. In our case, we are not that interested in precision and recall on the negative class because the class itself is composed of multiple classes that are unknown to us. There is no obstacle in computing precision and recall on the negative class but we do not include it in the results of our experiments in Chapter 6 and Chapter 7. The second major point is that precision is often used without understanding that it is meaningful only with the prevalences it was computed with.

### 2.4.6 Multiclass environment

All of the previous measures were described in the binary classification context. In multiclass environment we compute the precision and recall for each class. This can be done by treating the given class as positive and all of the other classes as negative and then we can use the formulas for precision and recall as in the binary case.

We also compute the average precision and recall over all positive classes. These values are very informative and provide us with quick insight into the general performance of a classifier on a given dataset. However, it is also

important to always look at the per-class values to analyze further what is causing the changes in the average values.

It is also worth considering how to treat classes with zero precision or recall when computing the averages. We decided to ignore zero precision values. That is because we assign zero precision in a case when no objects are marked as positive and precision is undefined. This has downside that we ignore results if they contain only false positives, because the precision is zero too in that case. We investigated this and these results do not occur in our experiments or they are so rare that it is not an issue.

Because we are dealing with classes that can be very rare it is not uncommon for classes to have zero recall. There are two ways how to compute the averages. One includes classes with zero recall and the other ignores them. Both averages are informative. The one with zeros included provides us with somewhat objective measure of the classifier's performance on all classes. The problem is that it is not as sensitive to changes in recall in already discovered classes as we would like. The recall average with zeros ignored captures those slight improvements in already discovered classes better but it can be misleading. When a class that was previously discovered with low recall is not discovered in the current evaluation then the average recall with zeros ignored might actually increase even thought the classifier has more incorrect predictions. Because of that, we decided to use only the version with zeros included in Chapter 6 and Chapter 7.

Plotting PR curve (or ROC curve) in multiclass environment is not straightforward and we do not do it. It would theoretically be possible to plot PR curve for each class by treating the class as positive and all of the other classes as negative. The problem is that there is no straightforward way how to treat objects that would be rejected for the positive class even if the confidence in the class is maximal of all of the classes. Publications such as [15], deal with the problem of setting the correct thresholds and decision points in multiclass environment but it is not in the scope of this thesis.

# Chapter 3

# Decision trees

In this chapter we describe decision trees which are the major building block of random forests. Understanding single decision trees is therefore mandatory to understand random forests. Decision trees have been around for a long time and a lot of research has been done around them. The most defining publication about decision trees, although not first, is probably [7] which features the *CART algorithm* for growing of binary decision trees which can be used both for classification and regression. Other algorithms for growing decision trees that are often used are the *ID3* algorithm [44] and it's successor *C4.5* [45]. Both ID3 and C4.5 produce nonbinary trees that are usually a bit shallower than CART trees. According to [3] there are not significant performance differences between binary and nonbinary trees. We did not use ID3 or C4.5 in this thesis so we do not explain those algorithms in detail.

Decision trees have tree-like structure known from graph theory. An example of a simple decision tree is available in Figure 3.1. The evaluation of an object starts in the root and each inner node is associated with a test that determines which edge to follow. Each leaf is associated with the result of classification. The result is either a class label or it can be a probability distribution across all classes where each value reflects the confidence of the tree that the object belongs to the given class.

The main strengths of decision trees are that they are able to handle lots of data without a need for much preparation, intrinsically implement feature selection and they are easily interpretable. That means that the reasoning behind a particular decision can be easily understood by looking at the tree structure even by a person without statistical background. This is in contrast with methods like *neural networks* which act more like a black box and it is difficult to explain their decisions. In addition decision trees usually do not require much customisation of their hyperparameters and often work well out of the box.

As with every method there are also several limitations of decision trees. It is known that the problem of training an optimal (smallest possible) decision tree on data is NP-Complete [29]. Because of that most of the algorithms only try to approximate the best solution by some heuristic. The second major limitation is that some patterns in data might result in very large trees that generalize badly to unseen data.

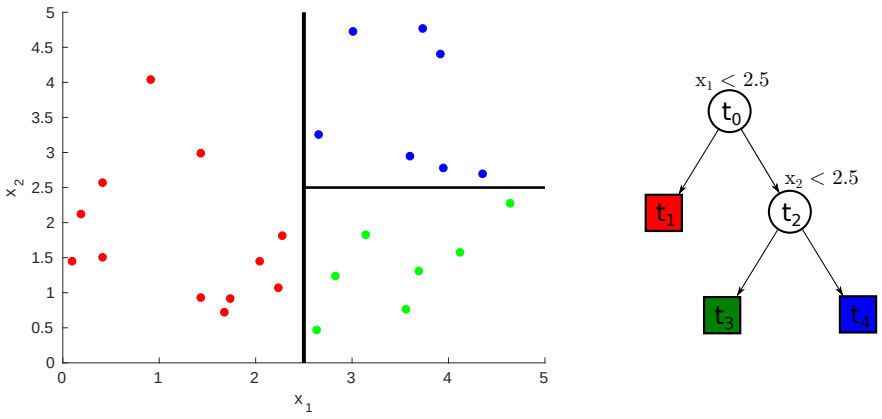



**Figure 3.1: Decision tree** trained on the data points on the left. Leafs contain the classification results. The inner nodes contain decision rules that correspond to the axis-aligned splits in the left image. These splits define decision boundaries in the feature space between regions with the same classification result. Nodes are numbered in the breadth-first order.

## 3.1 Notation

We use mathematical notation inspired by [3] and [41] to describe the algorithms for decision trees and random forests formally. This notation is compatible with the notation introduced in Section 2.1.

A single data point is denoted by a vector $\mathbf{x} = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ where $x_i$ are the numerical features of the data point. In general, decision trees have no problem dealing with nonnumerical features, but in this thesis we restrict ourselves to only numerical ones[1]. The data point's label, if available, is denoted by $y$. The set of all classes is denoted by $\mathcal{C}$. Nodes are denoted as $t_i$ and are numbered in the breadth-first order starting from root which is always $t_0$. Because we are dealing only with binary trees in this thesis, the left child of node $t_i$ can also be denoted as $t_i^L$ and the right child can be denoted as $t_i^R$.

All of the described training algorithms build the tree by recursively partitioning the training dataset. Because of that we denote the subset of training samples belonging to node $t_i$ by $\mathcal{S}_i$. The subset belonging to the left child of node $t_i$ is denoted as $\mathcal{S}_i^L$ and subset belonging to the right child as $\mathcal{S}_i^R$. It applies that $\mathcal{S}_i = \mathcal{S}_i^L \cup \mathcal{S}_i^R$ and $\mathcal{S}_i^L \cap \mathcal{S}_i^R = \emptyset$ for every inner node.

As with parts of the training dataset we can also associate nodes with regions in the feature space $X$ defined by the decision boundaries of the given node and all of the nodes on the path to the root. Each node $t_i$ is associated with a region $X_i$ and the notation $X_i^L$ and $X_i^R$ for regions associated with left or right child of $t_i$ is also available. The identities $X_i = X_i^L \cup X_i^R$ and $X_i^L \cap X_i^R = \emptyset$ also hold.

[1] Categorical features can be converted to numerical by adding dummy variables and using the one-hot encoding.

Each inner node $t_i$ is associated with a split function. Although the concept of split function is not hard to understand or implement, there are many ways how the split function can be formalized. We use the formalization provided in [3] which is a bit more general than what we need, but it also captures the flexibility that is allowed in the split function:

$$h(\mathbf{x}, \theta_j) \in \{0, 1\} \tag{3.1}$$

Depending on the result of the test, the data point $\mathbf{x}$ is either sent to the left child or the right child of node $t_i$. The split in node $t_i$ is characterized by it's parameters $\theta_i = (\phi, \psi, \tau)$[3]. $\psi$ defines the geometric primitive that separates the data. In this thesis we only use axis-aligned hyperplanes as the geometric primitives, but in general this can be anything (general oriented hyperplane, quadratic, general surface, etc.). $\tau$ is the vector of thresholds used in the test. In the case of axis-oriented hyperplane as a separator, there is only need for a single threshold. $\phi$ is a filter function that selects a subset of features from the vector $\mathbf{x}$.

The following is one of more ways how the axis-aligned hyperplane separator might look like:

$$h(\mathbf{x}, \theta_j) = [\phi(\mathbf{x}) \cdot \psi \geq \tau] \tag{3.2}$$

$[\cdot]$ is the indicator function which returns zero if the argument is false and one if it is true. We can define $\phi(\mathbf{x}) = (\mathbf{x}, 1)$, $\tau = 0$ and $\psi = (\mathbf{n_j}, \psi_j)$ where $\mathbf{n_j}$ is the normal vector of the hyperplane and $\psi_j$ is it's position on the axis. The calculation takes place in homogeneous coordinates. The exact rule for the node $t_0$ in Figure 3.1 would then look like this:

$$h(\mathbf{x}, \theta_0) = [(x_1, x_2, 1) \cdot (-1, 0, 2.5) \geq 0] \tag{3.3}$$

Axis-aligned splits can be expressed by a more readable notation $x_i < \tau$, where $i$ is the index of an axis and $\tau$ is the threshold on the axis. We will prefer this shorter notation when applicable, because, as we have shown above, it can be easily converted to the (3.2) form.

## 3.2 Training of decision trees

As stated earlier, training of the smallest decision tree that classifies the training dataset without errors is an NP-Complete problem. All of the well known algorithms mostly work in a greedy way by recursively partitioning the training dataset. In this section we explain the induction of binary decision trees as introduced in [7]. The algorithm described in the book is often known as the *CART algorithm*, although it became such a common thing that it is often not even mentioned by a specific name.

Algorithm 1 shows the basic high-level algorithm for the induction of a decision tree from a training dataset. In the following subsections we examine the specific parts of the algorithm.

**Algorithm 1** High-level overview of an algorithm for induction of a binary decision tree from a training dataset $\mathcal{S}_i$. The algorithm returns a handle to the root of the tree.

```
1: function BuildTree(S_i)
2:     if should create leaf node then
3:         return CreateLeafNode(S_i)
4:     θ_i ← FindBestSplitParameters(S_i)
5:     S_i^L, S_i^R ← SplitDataset(S_i, θ_i)
6:     t_i^L ← BuildTree(S_i^L)
7:     t_i^R ← BuildTree(S_i^R)
8:     return t_i                          ▷ Return node handle.
```

### 3.2.1 Deciding when to create a leaf node

At first the algorithm checks if it should create a leaf node. In practice the following conditions are checked:

- Check if the dataset $\mathcal{S}_i$ contains samples from only one class. If so, there is no need to split further and we can create a leaf node.
- Check if there is at least one split possible. In practice, it can happen that all of the vectors in $\mathcal{S}_i$ are the same, but the samples do not belong to the same class. In that case we create a leaf node.
- The algorithm usually contains hyperparameter determining the minimum number of samples necessary to create a leaf node. In our implementation this hyperparameter is called *minSamplesToSplit*. The larger this hyperparameter is the greater bias and smaller variance the algorithm has. If $|\mathcal{S}_i| < minSamplesToSplit$ then we create a leaf node. In random forests this parameter is usually set low because we want the trees to grow deep.
- Another hyperparameter with similar purpose is the *maxDepth* hyperparameter. Instead of limiting the number of samples in inner nodes it limits the maximal depth of a tree. We do not use this hyperparameter in our implementation.
- Sometimes the change in the *impurity measure* (usually information gain) during node splitting is used as a stopping condition. As with *maxDepth* we do not use it in our implementation.

### 3.2.2 Creating leaf nodes

Once in a leaf node, the goal is to make the actual classification based on the subset of samples associated with the node. In general, any classifier can be used for this. However, by far the most common choice is to make prediction based on the normalized histogram of classes in the subset. The node can either return the class with maximum frequency in the histogram or it can

return the histogram itself where each value can be interpreted as an estimate of probability that the object belongs to the given class.

### 3.2.3 Finding the best split

To find the best split we need the concept of *impurity measure*. Impurity measure of a node's subset $\mathcal{S}_i$ is a function $i(\mathcal{S}_i) \in \mathbb{R}$. The purer the data in the node are, the more confidence we have in it's prediction. The goal is to find a split $\theta_i$ that reduces the impurity in the node's children the most. To do that all of the possible splits are evaluated. The *decrease in impurity* due to a split is defined as:

$$\Delta i(\mathcal{S}_i, \theta_i) = i(\mathcal{S}_i) - \frac{|\mathcal{S}_i^L| i(\mathcal{S}_i^L) + |\mathcal{S}_i^R| i(\mathcal{S}_i^R)}{|\mathcal{S}_i|} \tag{3.4}$$

Several impurity measures are available, but only *Gini impurity* and *entropy* are commonly used. Studies, such as [46], show that there is not much difference in performance between Gini impurity and entropy, because most of the time they are consistent with each other [50].

Although, the original book [7] uses Gini impurity, we decided to use entropy[2] which is defined as:

$$H(X) = -\sum_{c \in \mathcal{C}}^{n} p(c) \log p(c) \tag{3.5}$$

The formula (3.5) sums over all classes and $p(c)$ is the frequency of class $c$ in the node's subset $\mathcal{S}_i$. If the frequency of a class $c$ is zero then the corresponding term in the sum is by definition zero. The entropy (and therefore impurity) is maximal if the classes are equally distributed in the subset $\mathcal{S}_i$. The entropy is zero if all of the objects in the dataset belong to the same class. When entropy is used as the impurity measure then the decrease in impurity $\Delta i(\mathcal{S}_i, \theta_i)$ corresponds to the *information gain*. Because of that, instead of decreasing impurity, we can look at the problem as information gain maximization.

To select the best axis-aligned split, all of the possible splits along all dimensions have to be evaluated. Algorithm 2 describes in pseudocode how this can be done efficiently. The major trick is that before performing any computation along dimension, the data points are grouped together to reduce duplicities. For example, if we have 50 data points with the same coordinate in the given dimension, we just remember the class counts of the 50 data points with the coordinate and further work only with the class counts. If we are working with real valued feature vectors then we should perform quantization to allow grouping of data that are sufficiently close together and avoid noise. Quantization can happen as a preprocessing step on the dataset so we do not mention it further and expect that the training vectors are quantized.

[2] The usage of entropy was popularized in the ID3 and C4.5 algorithms.

**Algorithm 2** Algorithm finding the best axis-aligned split on dataset $\mathcal{S}_i$.

**function** FindBestSplitParameters($\mathcal{S}_i$)
  $max\_gain_\Delta \leftarrow -\infty$
  **for** *all dimensions* **do**
    $counts^L \leftarrow (0, 0, \ldots, 0)$
    $counts^R \leftarrow GetClassCounts(\mathcal{S}_i)$
    $coordinates \leftarrow$ find unique coordinates in dimension
    $Sort(coordinates)$
    **for** $point \in coordinates$ **do**
      $counts^L \leftarrow$ add class counts present on point
      $counts^R \leftarrow$ remove class counts present on point
      $current\_gain_\Delta = ComputeGain_\Delta(counts^L, counts^R)$
      **if** $current\_gain_\Delta > max\_gain_\Delta$ **then**
        $max\_gain_\Delta \leftarrow current\_gain_\Delta$
        $\theta_i \leftarrow$ create split parameters from point and neighbor
  **return** $\theta_i$
**function** $ComputeGain_\Delta(counts^L, counts^R)$
  $size^L \leftarrow Sum(counts^L)$
  $size^R \leftarrow Sum(counts^R)$
  **return** $-\frac{size^L \cdot H(counts^L) + size^R \cdot H(counts^R)}{size^L + size^R}$ ▷ H is entropy. Eq. (3.5).

This trick can provide major performance boost because many features can have only a very limited set of values. If the feature is boolean we have to always sort only 2 values independently of the number of data points in $\mathcal{S}_i$. By having the coordinates sorted we are able to add and remove the class counts to $counts^L$ and $counts^R$ in an organized way. We do that by iterating over all of the sorted coordinates along the dimension. In each step we add all of the class counts that are present on the coordinate to $counts^L$ and subtract them from $counts^R$. In our implementation, we keep the class counts for each coordinate in table precomputed before the iteration where each row represents class counts for a single coordinate. In each iteration we then just advance to the next row.

The major improvement of this approach is that time of each entropy calculation is dependent only on the number of classes, which is constant, instead of the number of objects in $\mathcal{S}_i$. Also it is interesting to note that we only need to compute the second term in the information gain formula (3.4), because the term $i(\mathcal{S}_i)$ is constant and can be ignored if we only care about maximization.

The split is created in the middle of the interval between the current point and the following point along the dimension. This is done to maximize the margin of the classifier and improve it's generalization performance.

If we consider the number of dimensions and classes to be constant then the worst case time complexity of the splitting algorithm is $\mathcal{O}(|\mathcal{S}_i| \log \mathcal{S}_i)$. This is because in the worst case there will be no data points with duplicate coordinates and the dominant part of the algorithm will be the sort, which

will sort the whole dataset. As mentioned, in practice we can expect the performance to be better but it is highly dependent on the data.

## 3.3 Average weighted depth

Decision tree makes a prediction by descending from the root node to some leaf node. The time complexity of each prediction depends on the length of this path. For a given decision tree, we would like to know how long this path is on average. The average path length can be computed by computing the average depth of all leaves.

It is likely that not all of the leaves will be visited with the same frequency. To take this into account we will weight the leaves with the number of objects that arrived in them during the training phase. We call the result the *average weighted depth* of a tree and it is an estimate of the expected length of the path from root to leaf during a single prediction.

# Chapter 4

# Random forests

Single decision trees have tendency to overfit to the training dataset. A random forest is an ensemble of randomized decision trees. Ensemble methods in general work by combining multiple weak classifiers to build a stronger one. There are two main families of ensemble methods:

- **Boosting methods** work by sequentially constructing weak classifiers and each classifier is built to improve the performance of the previous ones. In terms of bias-variance error decomposition, the weak classifiers usually have large bias and the boosting method tends to reduce this bias and introduces variance into the model. Examples of boosting methods are the *Adaboost* [22] algorithm and *Gradient Tree Boosting* [8, 23, 24]. Gradient Tree Boosting has proven itself to be very effective. The downside is that it requires lots of hyperparameter tuning. It is not in the scope of this thesis.
- **Averaging methods** on the other hand, build the classifiers independently and average their predictions. Each of the weak classifiers usually has large variance, which is decreased by the averaging. This is a bit counterintuitive, because in this case, increasing the complexity of the model by adding more weak classifiers decreases the variance instead of increasing it. Random forests belong in this category.

Generally, there are two ways how the predictions of weak classifiers can be aggregated. In *majority voting* each tree in the forest votes for one class. The result is the class which received the majority of the votes. In *soft voting* [55] the prediction of the forest is obtained by averaging the class probability results from each tree. It was empirically shown that there is not much difference in results between these two methods [6]. Soft voting also has an advantage that it provides smoother class probability estimates. We use soft voting in all our implementations of different random forest variations.

For random forests to work correctly, it is necessary, that the individual trees in the forest are all different from each other. Each random forest algorithm has it's ways to inject randomness into the training process to control the correlation of individual trees. Randomness is usually injected either by modifying the training dataset for each tree, or by introducing randomness to the node splitting process.

Authors in [16] studied ensemble methods and identified three main reasons why they work better than single models:

- **Statistical:** If the training dataset is small compared to the size of hypothesis space, then the learning algorithm can find multiple hypotheses that have similar performance. By using these hypotheses in ensemble the algorithm can 'average' their votes and reduce the risk that a wrong hypothesis is selected.
- **Computational:** Algorithms often work by performing greedy search in the hypothesis space and variations in the training dataset can lead the algorithm to different local minima. Because of that, an ensemble model, where individual models have different starting points, can better approximate the true function than any single model.
- **Representational:** Sometimes, the true function can not be represented completely by any hypothesis in the hypothesis space. Ensembles can expand this hypothesis space by forming weighted sums of the weak hypotheses to create hypotheses outside of the hypothesis space.

## 4.1 Breiman forests

Breiman forests are by far the most commonly used random forest algorithm. They were described and rigorously analyzed by Leo Breiman [9] and he holds a trademark for the term "Random Forests".

Breiman forests are using the standard CART decision trees and combine *bagging* and *random feature subsets* at each node to inject randomness into the forest.

### 4.1.1 Bagging

*Bagging*, also called *bootstrap aggregating*, was first introduced and thoroughly studied in [6]. For a forest of size $m$ and training dataset of size $n$, bagging creates $m$ datasets of size $n'$ from the training dataset. The datasets are created by randomly sampling with replacement from the training dataset. It follows that some data objects from the training dataset are not present in a particular sample and some might be repeated. Each tree in the forest is trained on it's corresponding sample and the results are aggregated as mentioned above.

The size of the individual samples in bagging is a hyperparameter of the forest. The most common choice, by far, is to choose $n = n'$. In this special case the individual samples are called *bootstrap samples* [19]. The probability that a particular object from the training dataset is not present in a particular bootstrap sample can be expressed with the following formula:

$$(1 - \frac{1}{n})^n = \frac{1}{e} \cong 36.8\% \tag{4.1}$$

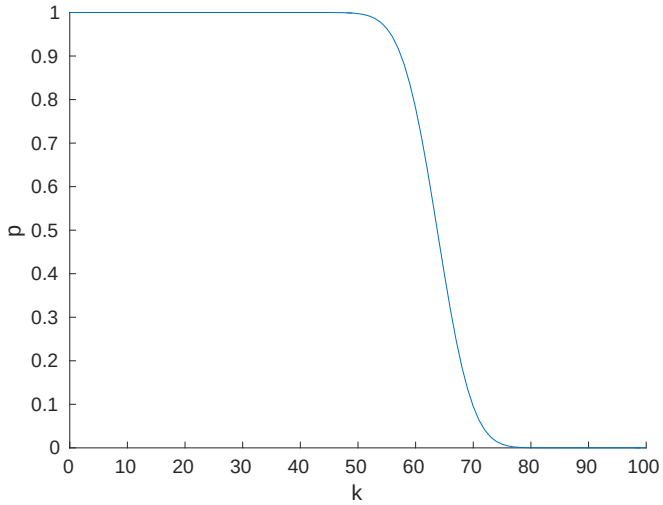



**Figure 4.1:** Plot of the probability that an object from the training dataset will occur in at least $k$ bootstrap samples. It is assumed that each bootstrap sample contains the same number of objects as the training dataset and therefore equation (4.2) is used to compute the probability. The number of trees in the forest is assumed to be 100, which is often used as a default value. We can see that it is practically certain that the object will occur in more than 50 trees.

The probability that a particular object from the training dataset occurs in a particular bootstrap sample is $1 - \frac{1}{e} \doteq 63.2\%$. When we have result for the probability of object occurrence in a single bootstrap sample it is easy to compute the probability of object occurrence in at least $k$ of $m$ bootstrap samples. Since, the bootstrap samples are independent from each other this probability can be computed using the binomial distribution as:

$$P(X \geq k) = \sum_{i=k}^{m} \binom{m}{i} \left(1 - \frac{1}{e}\right)^{i} \left(\frac{1}{e}\right)^{m-i} \tag{4.2}$$

When using bagging, it is possible to compute the *out-of-bag (OOB) error* on the training dataset. Out-of-bag error is the mean of prediction errors of objects in the training dataset. Each object is evaluated only on trees that did not contain the object in their bootstrap sample. Out-of-bag error is interesting, because it was shown that it is an accurate estimate of the generalization error. It was shown in [51] that the result is as good or even better than k-fold cross-validation, while it is more computationally efficient because it is not necessary to retrain the model k times. Breiman [6] shows that if we use the usual assumption that training dataset and test dataset are sampled from the same underlying distribution, then out-of-bag error on the training dataset is as accurate as using test dataset of the same size.

The standard bagging approach may not be sufficient when dealing with imbalanced datasets. When using sampling from the whole population it is possible that the bootstrap sample for a tree does contain very little objects from the minority classes and in extreme cases it can even contain none. This results in a tree with very poor performance.

There are several approaches to bagging modifications to handle imbalanced data. Simple approaches include oversampling the minority classes or downsampling the majority classes. Oversampling of minority classes leads to increased duplication of objects in the bootstrap samples and is therefore related to increasing the weights of the objects in classification. Oversampling leads to more correlated trees. Interesting method that is similar to oversampling is *SMOTE* [11]. SMOTE creates synthetic objects for the underrepresented classes. Undersampling of the majority class on the other hand may lead to bad utilization of the objects in the majority class. This is most striking when the majority class contains thousands of times more samples than the minority class. An example of undersampling scheme is in [12]. A problem with all of these approaches is that they disturb the actual distribution of the data which might be problematic and has to be considered when using them.

Method we are using is called *stratified bootstrap* [12] and we will also refer to it as *balanced bagging*. It is a simple method that samples adequate number of objects from each class to keep the resulting bootstrap sample balanced. When the number of objects in a class is so low that it would hurt too much to ignore 36.8 % of the samples, we do not perform any sampling and add all objects from the class. In the experiments we set this threshold to 50. This, of course, invalidates the reasoning behind the properties of out-of-bag error so we do not use it together with balanced bagging. The advantage of the method is that it works naturally in multiclass classification problem, while some of the other methods are better suited for binary classification.

### 4.1.2 Random feature subset sampling in nodes

The second technique that Breiman forests use to inject randomness into individual trees is the random feature subset sampling in nodes. It was first used in [2]. The standard algorithm for node splitting 2 uses all of the dimensions in the training data vectors to find the best split. However, trees in Breiman forests use only a random subset of the dimensions that is sampled for each split node individually.

The size of these random subsets is another hyperparameter of the forest, that we denote *splitFeaturesCount*. The value of this hyperparameter directly affects the correlation between individual trees. For $splitFeaturesCount = 1$ each node finds the best split only along a single random dimension. This leads to the least correlation between the individual trees, but it also increases bias of the individual trees. For $splitFeaturesCount = numberOfDimensions$ there is zero randomness injected to the node splitting process. All of the randomness comes from bagging and that is usually not enough, particularly when the training dataset is huge. The forest as a whole is then not much stronger than individual trees, because the individual trees are strongly correlated. Luckily, there exists an empirically tested rule of thumb that says $splitFeaturesCount = \sqrt{numberOfDimensions}$ is a good default value for classification tasks.

There is also another benefit of setting lower *splitFeaturesCount* and that

is the computation cost of the training phase. The time complexity of node splitting depends linearly on *splitFeaturesCount* and does not depend on *numberOfDimensions* anymore. This is beneficial because it allows us to add more features to the data without worrying about the performance impact, because that is now decoupled into a separate hyperparameter. It also opens doors for various performance optimisation techniques. For example, the feature values can be computed lazily only when they are specifically requested in the node splitting process, instead of precomputing them all ahead of training of the forest.

## 4.2 Extremely Randomized Trees

*Extremely Randomized Trees (ERT)* are another random forest variant that was introduced in [25]. In some scenarios, that are available in the original publication, they have been shown to perform better than Breiman forests.

The defining feature of Extremely Randomized Trees is the node splitting Algorithm 3 which randomizes both the split dimension and location. ERTs have a hyperparameter $k$, which specifies how many different split candidates in each node should be considered. The algorithm randomly samples $k$ split candidates, where each candidate is created by uniformly sampling random splittable dimension[1] and then also uniformly sampling threshold along this dimension. All of the split candidates are then compared using some impurity measure, usually information gain, and the best one is selected.

**Algorithm 3** Algorithm used in Extremely Randomized Trees for finding the best split on dataset $\mathcal{S}_i$.

**function** FindBestSplitParameters($\mathcal{S}_i, k$)
  $max\_gain_\Delta \leftarrow -\infty$
  **for** $j \in \{1, 2, \ldots, k\}$ **do**
    $\delta_j \leftarrow$ uniformly sample random splittable dimension
    $min_j \leftarrow \min\{\mathbf{x}_{\delta_j} | \mathbf{x} \in \mathcal{S}_i\}$
    $max_j \leftarrow \max\{\mathbf{x}_{\delta_j} | \mathbf{x} \in \mathcal{S}_i\}$
    $\xi_j \leftarrow$ uniformly sample split location from interval $(min_j, max_j)$
    $current\_gain_\Delta = ComputeGain_\Delta(\mathcal{S}_i, \delta_j, \xi_j)$
    **if** $current\_gain_\Delta > max\_gain_\Delta$ **then**
      $max\_gain_\Delta \leftarrow current\_gain_\Delta$
      $\theta_i \leftarrow$ create split parameters from $\delta_j$ and $\xi_j$
  **return** $\theta_i$

Extremely Randomized Trees do not perform bagging and use the whole dataset for each tree. The reasoning behind this is that the splitting process provides enough randomness and bagging is not necessary anymore.

[1] Splittable dimension in dataset $\mathcal{S}_i$ is a dimension for which exist at least two objects from $\mathcal{S}_i$ that have different values in this dimension.

From the bias-variance perspective, the idea is that the randomization of the threshold in combination with the ensemble averaging should be able to reduce the variance more than the scheme used in Breiman Forests. The usage of the whole dataset for each tree is motivated to reduce bias [25].

In theory, Extremely Randomized Trees should be faster to train than Breiman forests, because of simpler node splitting algorithm. In our experiments we did not notice a significant difference, because Extremely Randomized Trees usually contained more nodes than trees in Breiman Forests and therefore even if the node splitting is faster it had to be performed more times.

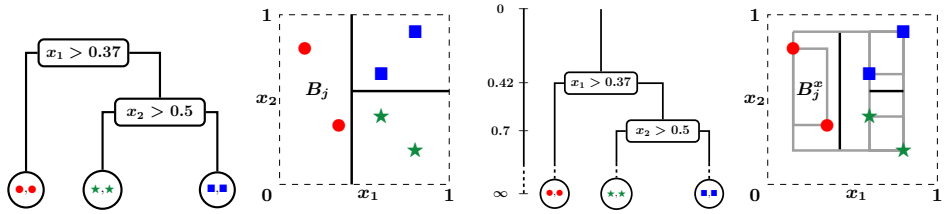
The hyperparameter $k$ controls the amount of randomness injected into the trees. [25] states that $k = \sqrt{numberOfDimensions}$ is a reasonable default value for classification. In a special case when $k = 1$ Extremely Randomized Trees are sometimes called *Totally Randomized Trees* [41] or *1-ERT*. Totally Randomized Trees are interesting, because they do not need any impurity measure and do not need to look at the class labels during the node splitting. It can be said that the tree structure is learned in an unsupervised way. This allows for many interesting approaches, for example, in semi-supervised setting, the trees can be inducted by joining the labeled and unlabeled data and it is possible to additionally provide labels for some of the unlabeled data later, which may improve performance, without the need for the induction of the trees again. Publications [30] have shown that Totally Randomized Trees do not fall too short in performance when all of the features are relevant. If however, there are some noisy, irrelevant features the performance of Totally Randomized Trees drops dramatically.

As a stopping criterium *minSamplesToSplit* is used instead of *maxDepth* because, in general, ERTs tend to be deeper and less balanced than Breiman forests and *maxDepth* would limit them too much.

## 4.3 Mondrian forests

Mondrian forests [30] are an interesting class of random forests. They are more similar to Extremely Randomized Trees than Breiman forests. The ensemble behaves in the same way as in ERTs but the individual trees in the ensemble represent finite restrictions of so-called Mondrian processes [48]. There exist both *online* and *offline* versions of the *Mondrian Tree* induction algorithm and both versions agree with each other. Although Mondrian forests were developed primarily because of their ability to be trained in online fashion, we are concerned only with the offline version of the training algorithm in this thesis.

We were interested in Mondrian forests, because their structure is uncommon and they do not use data labels during node splitting, which made them interesting candidates for possible modifications to semi-supervised learning algorithm. However, we show that with common hyperparameter settings they are very similar to 1-ERTs and we also present an obstacle that makes their use on larger datasets complicated.




**(a) :** Decision Tree **(b) :** Mondrian Tree

**Figure 4.2:** Example of a difference between a decision tree in Figure 4.2a and Mondrian tree in Figure 4.2b inducted on the same training data in two dimensions. Note that each node $t_i$ in the Mondrian tree is associated with a time that is shown on the vertical axis. Nodes in Mondrian trees are also associated with blocks that are denoted by the gray rectangles and Mondrian trees may not commit to a decision outside of these blocks. Both figures are from [30].

In this section we assume that all of the data are normalized to $[0; 1]$ intervals along each dimension. This is done to simplify the algorithms. The normalization of the training dataset is straightforward. To normalize the training dataset along dimension $j$ we compute values $min_j$ and $max_j$ which represent the minimum and maximum values along the dimension $j$. The object $\mathbf{x}$ in the training dataset is then transformed with the formula (4.3) along each dimension to obtain normalized object $\mathbf{x}'$. For simplicity when we refer to objects $\mathbf{x}$, we actually mean the normalized objects $\mathbf{x}'$ in this section.

$$x'_j = \frac{x_j - min_j}{max_j - min_j} \tag{4.3}$$

The normalization of objects in the testing dataset is performed with $min_j$ and $max_j$ computed on the training dataset so it is possible that values in some dimensions can actually be outside the $[0; 1]$ interval. This is necessary, because we require that training and testing data are transformed with the same formula. It also simulates the real use case, where $min_j$ and $max_j$ are parameters of the model trained on the training dataset and each object during testing is normalized on the fly inside the evaluation of the prediction of the Mondrian forest.

### 4.3.1 Mondrian tree

There are fundamental differences in the structure of Mondrian trees and standard decision trees. Figure 4.2 shows Mondrian tree and decision tree inducted on the same data. The main difference in the structure between Mondrian trees and decision trees is that each node $t_i$ in a Mondrian tree stores the bounding box of it's training subset $\mathcal{S}_i$. This bounding box is denoted as $B_i$. The stored bounding boxes are necessary both for the online induction algorithm and also for the prediction algorithm. We will discuss the prediction algorithm later.

Algorithm 4 describes the outline of the offline induction of Mondrian trees. The offline algorithm is simpler than the online version.

Each node $t_i$ is associated with a *time* $\tau_i$ which always increases on the path from root to leaf. The increase of time $\tau_i$ from $\tau_{parent(i)}$ is sampled from exponential distribution with rate $\sum_d (u_{id} - l_{id})$. This rate is so called linear-dimension of the bounding box $B_i$. As a result, it is more probable that large bounding boxes will further split, because the increase in time will tend to be smaller for larger bounding boxes.

The algorithm has a hyperparameter $\lambda$ which is called *lifetime* of the tree. Hyperparameter $\lambda$ determines the maximum time of a node in a tree and acts as a stopping condition similar to *splitFeaturesCount* or *maxDepth* that were discussed in Section 3.2.1. If $\lambda$ is set to $\infty$ then the tree grows until there are no more splits available, or all of the data in $\mathcal{S}_i$ are from the same class. These standard stopping conditions are not in the Algorithm 4, but they are always present. Also, when $\lambda$ is equal to $\infty$ the offline algorithm for tree induction is almost similar to 1-ERTs. The only remaining difference in the induction from 1-ERTs is that the split dimension probability is not uniform, but is proportional to the length of the training data bounding box along the dimension. Authors of [30] used $\lambda$ equal to $\infty$ in all of their experiments. In our experiments in Section 6.1.2, we show that Mondrian forests usually achieve the best accuracy when $\lambda$ is set to $\infty$.

**Algorithm 4** Offline algorithm for the induction of Mondrian trees [30].

**function** CreateMondrianTree($\mathcal{S}_i, \lambda$)
  **for** $d \in$ *dimensions* **do** ▷ Compute bounds of $\mathcal{S}_i$ in each dimension.
    $l_{id} \leftarrow \min \{\mathbf{x}_d | \mathbf{x} \in \mathcal{S}_i\}$
    $u_{id} \leftarrow \max \{\mathbf{x}_d | \mathbf{x} \in \mathcal{S}_i\}$
  $E \leftarrow$ Sample from exponential distribution with rate: $\sum_d (u_{id} - l_{id})$
  **if** $\tau_{parent(i)} + E < \lambda$ **then**
    $\tau_i \leftarrow \tau_{parent(i)} + E$
    $\delta_i \leftarrow$ sample split dimension, choosing $d$ with $p(d) \propto (u_{id} - l_{id})$
    $\xi_i \leftarrow$ sample split location, uniformly from interval $(l_{i\delta_i}, u_{i\delta_i})$
    $\theta_i \leftarrow$ create split parameters from $\delta_i$ and $\xi_i$
    $\mathcal{S}_i^L, \mathcal{S}_i^R \leftarrow SplitDataset(\mathcal{S}_i, \theta_i)$
    $t_i^L \leftarrow CreateMondrianTree(\mathcal{S}_i^L, \lambda)$
    $t_i^R \leftarrow CreateMondrianTree(\mathcal{S}_i^R, \lambda)$
  **else**
    $\tau_i \leftarrow \lambda$
    create $t_i$ as leaf node
  **return** $t_i$ ▷ Return node handle.

The state stored with each node $t_i$ includes $\mathbf{l}_i$, $\mathbf{u}_i$ and $\tau_i$. This is problematic because the nodes occupy much more memory than in standard decision trees. As an example, consider a case where the vectors in $\mathcal{S}_i$ have 500 dimensions and the trained Mondrian tree contains 50000 nodes. If each value in the training dataset has size of 8 bytes then each node has at least

$2 * 500 * 8 = 8000$ bytes just by storing the bounding box in vectors $\mathbf{l}_i$, $\mathbf{u}_i$. A single tree with 50000 nodes will then occupy more than 400 MB of memory space. A forest of 100 Mondrian trees would occupy over 40 GB of memory space. Now, if we consider that certain datasets contain much more features and that Mondrian trees usually have large number of nodes because they do not consider labels during splitting, we can see that in some cases memory can be a major limitation of Mondrian forests.

The prediction algorithm of Mondrian trees is much more complicated than the prediction algorithm of normal decision trees. However, when we tried replacing this algorithm with the standard histogram based prediction in leafs that is used in common decision trees we did not notice a significant increase in performance on the datasets we used. Because of that, and because we used Mondrian forests only marginally in our implementation, we discuss the prediction algorithm very briefly. Complete explanation can be found in the appendix of [30].

Mondrian trees use hierarchical Bayesian approach for prediction. Each node is associated with a probability distribution and as we descend from the root to the leaf, we combine those distributions to arrive at the final distribution in the leaf.

If the test point $\mathbf{x}$ lies in the bounding box of the leaf then we can return result that was precomputed after the tree induction using approximation of the *Chinese restaurant process* [1]. If the test point $\mathbf{x}$ lies outside of the bounding box of the current node during the descend, then a possibility of branching out into a new node is considered with a probability determined by the sum of distances of the point from the bounding box along all dimensions. The further the point is, the greater is the possibility of branching out. Expected prediction in the virtual leaf node is then combined into the current prediction state and the standard path to the leaf node is followed in the descend.

The expected advantage is that this should produce smoother decision boundaries than the standard histogram based approach in leafs. The prediction algorithm requires to store 2 arrays in each node with length equal to the number of classes. This effectively makes the memory complexity of nodes linearly dependent both on the number of features and the number of classes.

# Chapter 5

# Methods for handling unlabeled data

So far, we were only concerned with the standard supervised learning scenario. In this chapter we consider a scenario where part of the training dataset is made of unlabeled data. The common semi-supervised learning approaches, that are used for learning from labeled and unlabeled data together, usually assume that the class distribution in the unlabeled data is the same as in the labeled data. However, we consider a different problem where this assumption does not hold. In our case the majority of unlabeled data belongs to a class that we call *negative* which is not present in the labeled data. The labeled data are made of classes that we call positive. The remaining part of unlabeled objects should belong to the positive classes but for various reasons they were missed when the dataset was labeled. The situation is described in Figure 5.1. Standard supervised learning methods can be applied on this problem if we label all of the unlabeled objects with the *negative* label and assume that mislabeled objects are rare enough that the performance does not suffer too much.

In this chapter, we explain the methods we implemented to better handle training datasets that are structured the way described above. These methods (*DAS-RF* [32], *PU learning* [38, 35] and *SMOTE* [11]) have the benefit that they can be used together with the supervised learning random forest algorithms presented in Chapter 4. In case of PU learning and SMOTE, the methods act as a preprocessing step on the training dataset. In the case of DAS-RF, the method acts as a wrapper around the chosen random forest algorithm. In the end, all of the methods that are presented in this chapter can theoretically be combined if they prove to be beneficial.

## 5.1 Related work

Published methods related to the positive and unlabeled problem mainly focus on the binary case where the labeled dataset contains only single class. Because of that, and because they often originate in the field of text classification, it is common that they are tightly coupled to specific types of classifiers, typically Naive Bayes [38] or SVM [37, 20].

The most common approach is the application of some two-step strategy where the first step focuses on the identification of true negatives in the

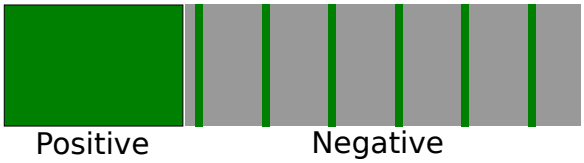



**Figure 5.1:** Diagram illustrating our situation with mislabeled data. In the training dataset we have subset marked as positive and objects in this subset have their labels assigned correctly. There are multiple different classes in the positive subset. The majority of the subset marked as negative indeed belongs to the negative class (gray color), but the negative subset also contains mislabeled objects whose true label belongs to some positive class. It might make sense to not call the subset negative but call it unlabeled instead. We chose to call it negative because this subset is used with negative labels when performing the standard supervised learning.

unlabeled dataset and the second step trains a classifier of choice on the labeled dataset joined with the objects identified in the first step. We use the most well-known of these methods [38, 35] and it is the subject of Section 5.3.

Other class of methods views the unlabeled data as negatives and applies different weights on the positive and unlabeled data. The popular methods in this class are tailored to specific classifier's and binary problems. Logistic regression is used in [31] and weighted SVM in [20].

We decided to examine methods that were not specifically developed to solve this problem, but are more compatible with random forests and multiclass classification. Method DAS-RF [32] was developed for the usual semi-supervised learning problem where the labeled and unlabeled datasets originate from the same distribution. It is one of the few existing representative attempts at semi-supervised learning that is applicable to random forests [39]. SMOTE [11] is a well-known method with numerous extensions [26, 28, 10] that is used in presence of imbalanced datasets [40]. We saw potential in both of the methods to also improve performance in the positive and unlabeled problem.

## 5.2 Deterministic Annealing based Semi-Supervised Random Forests

*Deterministic Annealing based Semi-Supervised Random Forests (DAS-RF)* is a method for semi-supervised learning introduced in [32]. In addition to the labeled dataset we now also have an additional unlabeled dataset available for training. As usual with semi-supervised methods, DAS-RF assumes that the class distribution in the unlabeled dataset is similar to the distribution in the labeled dataset. As explained earlier in this chapter, our network data do not follow this assumption. Nevertheless, we found the approach used in DAS-RF to be interesting and promising, so we implemented them in order to further modify them for our use case. First we explain them in the context of standard semi-supervised learning and in the end of the section we show

how we attempted to modify them to handle mislabeled data.

DAS-RFs are based on an optimisation method called *deterministic annealing* that was used for clustering in [47]. Deterministic annealing, in general, aims to find global optima of a function. The algorithm works in an iterative fashion. During each step it randomly perturbs the current solution in an attempt to escape local minima. The amount of perturbation in each iteration is controlled by a value called *temperature (T)*. If the temperature is high then the algorithm performs large perturbation allowing it to explore the search space further. If the temperature is low then the perturbation is low allowing the algorithm to optimize in a closer neighbourhood of the current solution. The iterative procedure starts at some initial temperature and cools down the temperature for each iteration. This allows the algorithm to explore the search space in the beginning, but forces it gradually to settle in some region. This is somewhat related to another global optimisation method called *graduated optimisation* [5], which works by smoothing the function to some easily optimizeable version and then iteratively making the smoothed function more similar to the original function and using the previous solution as a starting point for optimisation in the next iteration.

DAS-RF applies deterministic annealing to unlabeled data to optimize their label assignments. Each iteration of the algorithm trains a random forest and classifies the unlabeled data. As a result of the classification we obtain a probability distribution of labels for each unlabeled point. The current temperature determines how this distribution is transformed. High temperatures transform the distribution close to a uniform distribution, while low temperatures transform the distribution to a distribution similar to Dirac delta function that prefers the label with maximum probability in the result from classification. In the next iteration of DAS-RF each point from the unlabeled dataset is sampled a label from the transformed distribution. The random forest is then trained both on the labeled data and on the unlabeled data with assigned labels. To get the initial label distribution, before deterministic annealing is started, DAS-RF trains a random forest just on the labeled dataset.

Because unlabeled data might not be always helpful, DAS-RF also features a safety mechanism called *airbag*. It computes the out-of-bag error of the initial random forest on the labeled data. If the out-of-bag error of the forest obtained from deterministic annealing is worse than the initial error then DAS-RF returns the initial random forest instead of the optimized one. This should guarantee that the unlabeled data do not worsen the performance.

The forests trained inside of the algorithm are Breiman Forests, but we have also experimented with ERTs and it did not affect the results significantly. However, when using standard ERTs it is not possible to compute the out-of-bag error because ERTs do not perform bagging.

### 5.2.1 Label probability distribution

The overall DAS-RF loss function is [32]:

$$\begin{aligned}\mathcal{L}_{DA}(\mathbf{g}, \hat{\mathbf{p}}) = &\frac{1}{|\mathcal{X}_l|} \sum_{(\mathbf{x},y)\in\mathcal{X}_l} \ell(g_y(\mathbf{x})) \\ &+ \frac{\alpha}{|\mathcal{X}_u|} \sum_{\mathbf{x}\in\mathcal{X}_u} \sum_{i=1}^{K} \hat{p}(i|\mathbf{x})\ell(g_i(\mathbf{x})) \\ &+ \frac{T}{|\mathcal{X}_u|} \sum_{\mathbf{x}\in\mathcal{X}_u} -H(\hat{\mathbf{p}})\end{aligned} \tag{5.1}$$

The labeled training dataset is denoted as $\mathcal{X}_l$ and the unlabeled training dataset is denoted as $\mathcal{X}_u$. $T$ is the current temperature value. The probability $p(i|\mathbf{x})$ denotes the probability that object $\mathbf{x}$ is sampled label $i$ in the DAS-RF relabeling step.

Vector $\mathbf{g}(\mathbf{x})$ is called a *margin vector* and it has to hold that $\forall\mathbf{x} : \sum_{i=1}^{K} g_i(\mathbf{x}) = 0$. If the random forest classification returns probability vector of length $K$ then the margin vector can be obtained from this vector by simply subtracting $\frac{1}{K}$ from each value in the classification result vector.

Function $\ell(g_i(\mathbf{x}))$ is a *loss function*. In [32] the loss function is defined as margin maximizing if $\ell'(g_i(\mathbf{x})) \leq 0$ holds for all values of $g_i$. This is a natural requirement, because it means that a larger margin should not have worse loss function value than a smaller margin. A large number of margin maximizing loss functions can be used. It is not clear from [32] which loss function the authors used. A candidate function seems to be

$$\ell(g_i(\mathbf{x})) = -\log(g_i(\mathbf{x}) + \frac{1}{K}) \tag{5.2}$$

because authors hint that the loss function used should be similar to the loss function used in split nodes. Then they assume equality between formula for entropy used in split nodes $-\sum_{i=1}^{K} p_i \log(p_i)$ and formula $\sum_{i=1}^{K} p_i \ell(p_i - \frac{1}{K})$. Because $g_i(\mathbf{x}) = p_i - \frac{1}{K}$, we can come up with the formula (5.2).

DAS-RF has a hyperparameter $\alpha$ that can be used to balance the weight of the second term in the overall loss function (5.1) and determine if the loss function prioritizes labeled or unlabeled objects.

If the temperature is high, the overall loss function puts more weight on the third term which can be minimized by increasing the entropies of the probability distributions. As the temperature decreases, more weight is put on the second term which forces $\hat{\mathbf{p}}$ to prioritize labels that maximize margins.

The formula for computation of label probability density inside of the unlabeled data relabeling step in each iteration of DAS-RF is

$$p(i|\mathbf{x}) = \exp\left(-\frac{\alpha\ell(g_i(\mathbf{x})) + T}{T}\right)/Z(\mathbf{x}) \tag{5.3}$$

where $Z(\mathbf{x})$ is the normalizing constant obtained by summing all $p(i|\mathbf{x})$ to normalize the values into the $[0, 1]$ interval. Formula (5.3) minimizes the overall DAS-RF loss function for a given value of temperature [32].

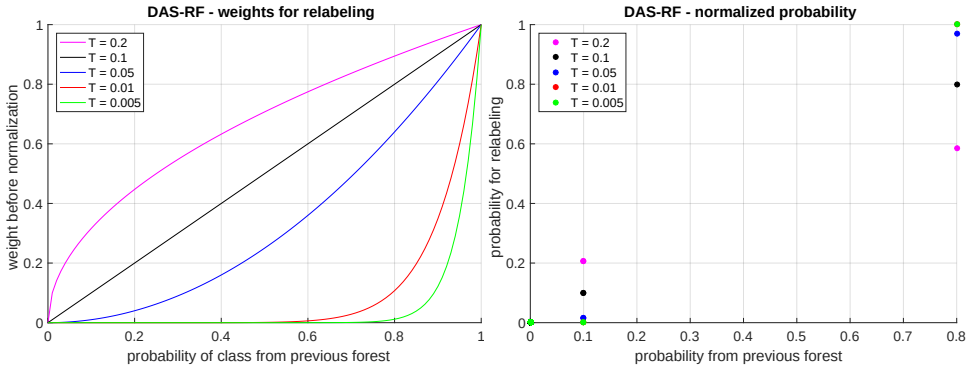



**Figure 5.2:** The left figure plots how the numerator of equation (5.3) behaves with different values of temperature. The $\alpha$ hyperparameter is fixated to 0.1 and loss function (5.2) is assumed. The behavior is independent of the number of classes because this specific loss function can be computed without computing margin. The results are normalized into the $[0, 1]$ interval to easily compare the shapes of the functions. The right figure shows the resulting probabilities for sample relabeling that are created by computing the weights from the left figure for each label and normalizing them. This figure shows only a specific example where we assume that there exist 150 classes, one of these classes has probability 80 % from the previous forest and there exist two other classes with probabilities 10 %. The rest of the classes has zero probability. This example was chosen because it reflects our use case with mislabeled data.

### 5.2.2 Hyperparameters

DAS-RFs have several hyperparameters and some of them are not trivial to set. In addition to the following hyperparameters which are only specific to DAS-RF, we also have to supply hyperparameters for the underlying random forest type.

The $\alpha$ hyperparameter occurs only in formula (5.2). It was set to 0.1 in all experiments in [32]. In addition to the $\alpha$ hyperparameter, we also need to supply the loss function $\ell(g_i(\mathbf{x}))$ as mentioned in the previous section.

Concerning the temperature, it is necessary to supply the initial temperature $T_0$ and a cooling function. There are several possible candidates for the cooling function and they have been used in literature [52] depending on the current requirements. The authors of DAS-RF state that they used "a simple exponential cooling function" in their experiments. We assume that the function they used was $T_i = T_0 \cdot r^i$ but we do not know what was the value of the constant $r$ which determines the rate of cooling.

Figure 5.2 shows how the label probabilities are transformed inside of DAS-RF when different temperatures are used. We can see that as the temperature cools down, more emphasis is put on the majority label.

We also need to specify a hyperparameter determining the maximum number of iterations of DAS-RF. DAS-RF usually always runs for the max number of iterations but alternatively some more sophisticated stopping condition may be added. The authors of DAS-RF used 20 iterations.

### 5.2.3 Modifications for handling mislabeled data

In the context of Figure 5.1, we want to use the positive subset as the labeled DAS-RF dataset and the negative subset as the unlabeled DAS-RF dataset.

The original DAS-RF assumes that the unlabeled data are sampled from the same distribution as the labeled data. In our case, the majority of unlabeled data belongs to a class *negative* that does not even occur in the labeled data. If, in the initial step, we trained the initial random forest on labeled data only, then this initial random forest would not be able to ever assign objects into the *negative* class.

To handle this, we sample a subset of the unlabeled dataset for each tree and append it to the initial dataset with class label *negative*. The idea is that majority of this subset will truly belong to the *negative* class and the initial random forest will become able to represent it. Of course, the problem with this approach is that the subset also contains small portion of mislabeled data that may throw of the initial random forest.

Then, ideally, during the iteration of DAS-RF, the assigned labels of at least a portion of mislabeled data should converge to their true labels, while the truly negative data should stay in the *negative* class.

We also tried modifying DAS-RF, so that instead of creating random distributions for each unlabeled object and sampling labels from those distributions randomly, we assigned the object to the majority class instead. Alternatively, we also added a threshold hyperparameter that would assign object to the majority class only if the majority class' margin exceeded this threshold. If the margin was smaller than the threshold we ignored the object in this iteration of DAS-RF. In this setting, which we internally called *threshold DAS-RF*, the temperature hyperparameter and the cooling function was unused.

However, it appears that threshold DAS-RF is not much useful and it did not improve the performance in our experiments. We believe, that the reason is that if the labels are added deterministically in accordance with the prediction of the previously trained random forest, we only add objects that are already correctly classified by the previous model and it does not affect it's behaviour. The most valuable objects are the ones close to the decision boundary of the classifier and these objects are ignored in threshold DAS-RF because their margin is not large enough to exceed the threshold.

## 5.3 PU learning spy technique

PU learning[1] spy technique is originally described in [38, 35] in the context of text classification as part of an algorithm that the authors called *S-EM*. We did not implement the whole S-EM algorithm since it is built around the Naive Bayes classifier but we implemented the initial step – the spy technique. In the context of this thesis, when we further refer to PU learning we always refer to the PU learning spy technique algorithm.

[1] PU learning stands for Positive-Unlabeled learning.

**Algorithm 5** DAS-RF algorithm from [32] without airbag but with our modification for handling mislabeled data.

**function** DAS-RF($\mathcal{S}_{labeled}, \mathcal{S}_{unlabeled}$)
  $\mathcal{S}_{unlabeled\_init} \leftarrow SampleSubset(\mathcal{S}_{unlabeled})$
  $\mathcal{S}_{init} \leftarrow \mathcal{S}_{labeled} \cup \mathcal{S}_{unlabeled\_init}$
  $\mathcal{F} \leftarrow TrainForest(\mathcal{S}_{init})$
  $i \leftarrow 0$
  **while** not stopping condition **do**
    $T_i \leftarrow c(T_0, i)$ ▷ $c(T_0, i)$ is the cooling function
    $i \leftarrow i + 1$
    $\forall \mathbf{x}_u \in \mathcal{S}_{unlabeled}, k \in \mathcal{C}$ : compute $p^*(k|\mathbf{x}_u)$ according to eq. (5.3)
    **for** $j \in \{1, 2, \ldots, N\}$ **do** ▷ N is the number of trees in a forest
      $\forall \mathbf{x}_u \in \mathcal{S}_{unlabeled}$ draw $\hat{y}_u$ from $p^*(k|\mathbf{x})$
      $\mathcal{S}^i_j \leftarrow \mathcal{S}_{labeled} \cup \{(\mathbf{x}_u, \hat{y}_u) | \mathbf{x}_u \in \mathcal{S}_{unlabeled}\}$
      $f_j \leftarrow BuildTree(\mathcal{S}^i_j)$ ▷ retrain the tree $f_j$
  **return** $\mathcal{F}$

PU learning is made exactly for the situation described in Figure 5.1. We use PU learning as a preprocessing step before the application of random forest in standard supervised learning setting. The goal of PU learning is to identify the mislabeled objects inside of the negative dataset and filter them out. The objects that remain are called *real negatives*.

PU learning can increase the recall of positive classes because it removes the mislabeled objects from the negative subset and should therefore lower the possibility that positive objects are wrongly classified as negative.

Algorithm 6 is almost similar to the algorithm proposed in [38]. The only difference is that the original algorithm uses naive Bayes classifier inside, while our version uses a random forest. This might possibly cause problems, because the classifier inside PU learning tests objects from the same dataset as it was trained on. Random forests generally have more variance and less bias than Naive Bayes classifiers so we have to be more careful not to overfit on the dataset. For example, if we used ERTs inside with usual hyperparameter selection (*minSamplesToSplit* set to 2) then the forest would classify all of the training objects perfectly, which would not be helpful. Luckily, if we use Breiman forests with bagging, we can control the variance more, but we still have to be careful to set the hyperparameters reasonably. Since we deal with multiclass problem we also have to include an additional minor implementation detail and join the labels of all positive classes to a common label inside of PU learning.

The main idea behind PU learning spy technique is to sample a small subset of the positive objects, called spies, and add them to negative dataset in expectation that those spies will behave in a similar way to the unknown mislabeled objects in the dataset. The ratio of positive objects sampled as spies is a hyperparameter and authors of [38] advise to sample around 15 %. Because we are dealing with imbalanced dataset, we perform stratified

**Algorithm 6** PU learning spy technique algorithm. The algorithm returns filtered unlabeled dataset that should contain mostly only the real negatives.

**function** PuLearning($\mathcal{S}_{input\_positive}, \mathcal{S}_{input\_negative}$)
  $\mathcal{S}_{positive}, \mathcal{S}_{spies} \leftarrow Split(\mathcal{S}_{input\_positive})$ ▷ sample spy set
  $\mathcal{S}_{negative} \leftarrow \mathcal{S}_{input\_negative} \cup \mathcal{S}_{spies}$ ▷ add spies to the negative set
  Assign every object in $\mathcal{S}_{positive}$ to class 1.
  Assign every object in $\mathcal{S}_{negative}$ to class 0.
  $\mathcal{F} \leftarrow BuildForest(\mathcal{S}_{positive} \cup \mathcal{S}_{negative})$
  Classify $\mathbf{x} \in \mathcal{S}_{spies}$ and decide threshold $t$ on $p(1|\mathbf{x})$.
  $\mathcal{S}_{real\_negative} \leftarrow \{p(1|\mathbf{x}) < t | \mathbf{x} \in \mathcal{S}_{input\_negative}\}$
  **return** $\mathcal{S}_{real\_negative}$

sampling so that the prevalences of classes in the spy set follow the prevalences in the positive set. We then create a classifier (in our case random forest) on the newly created positive and negative sets.

To find the threshold $t$, we classify all of the spies by the classifier and sort them by $p(1|\mathbf{x}), \mathbf{x} \in \mathcal{S}_{spies}$. There exists a hyperparameter called *noiseRatio* that specifies how many of the spies should be under the threshold. This is to avoid some possible noise in the training data. By experimentation we discovered to set *noiseRatio* to 0.5 % in our experiments on network datasets.

Because we assume that the mislabeled objects behave similarly to spies, we filter out objects in the negative set whose predicted probability that the object is positive is above the threshold. In the ideal case, if the assumption truly holds, we should be able to filter out $1 - noiseRatio$ of the mislabeled objects.

Because we have abundance of negative data in our experiments on network data we are mostly interested only in the amount of mislabeled objects that we correctly filter out. PU learning also filters out some real negatives incorrectly, but we do not mind much if the number is reasonable. However, experiments on public datasets in Chapter 6 suggest that the real negatives incorrectly filtered out are the important objects that lie on the boundary between classes. During our experiments on network data we did not have problems with incorrectly filtering out real negatives but we would be happy to filter out more mislabeled data, because we still missed a considerable portion of them. This is because the assumption that the spies behave the same as mislabeled data does not completely hold in our case. We elaborate more about this in Chapter 7 with network experiments.

## 5.4 SMOTE: Synthetic Minority Over-sampling Technique

SMOTE [11] is originally a technique developed for dealing with imbalanced data. It is an interesting approach to the problem because it creates synthetic objects of the underrepresented classes. We see a possibility to apply SMOTE

also to the mislabeled data problem, because due to the nature how SMOTE works it might be able to somewhat neutralize mislabeled objects, by creating new synthetic positive objects in their neighbourhood. As with PU learning, SMOTE is also applied as a preprocessing step on the training dataset.

**Algorithm 7** SMOTE algorithm for oversampling a single data point $\mathbf{q}$ by finding neighbours in $\mathcal{S}_{input}$. If the multiplier for the input data point is $m$ then the function returns set $\mathcal{O}$ including the input data point and $m-1$ synthetic data points. This function is called repeatedly for each data point that should be oversampled.

1: **function** SMOTEPOINT($\mathbf{q}, \mathcal{S}_{input}$)
2: $\quad\mathcal{N} \leftarrow FindKNearestNeighbours(\mathbf{q}, \mathcal{S}_{input}, k)$
3: $\quad\mathcal{O} \leftarrow \{\mathbf{q}\}$
4: $\quad$**for** $i \in \{1, 2, \ldots, m-1\}$ **do**
5: $\quad\quad\mathbf{n} \leftarrow$ sample random neighbour uniformly from $\mathcal{N}$
6: $\quad\quad\alpha \leftarrow$ sample random number uniformly from $(0, 1)$
7: $\quad\quad\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathbf{q} + \alpha(\mathbf{n} - \mathbf{q})\}$
8: $\quad$**return** $\mathcal{O}$

First, the SMOTE algorithm 7 finds k-nearest neigbours of each object that should be oversampled. The number of neighbours is a parameter and the authors mention that they used 5 nearest neighbors in their implementation. Before computing the nearest neighbors, the data points' coordinates should be normalized so that each dimension has the same scale. The normalization can be performed by using formula (4.3) or alternatively it can be performed lazily inside of the formula for distance computation in k-NN algorithm. The k-NN algorithm is a major performance bottleneck, and the standard CPU based algorithms for computing it do not work well in high dimensions.

The second parameter of SMOTE is the multiplier for a given data point (query point) specifying how many synthetic objects should be created from it. This multiplier can be different for different data points.

A synthetic data point is created by randomly selecting a neighbour from the k-nearest neighbours of the query point and sampling the synthetic data point uniformly on the line joining the query point and the selected neighbour. This process is repeated as many times as many synthetic objects should be created for the given query point.

The selection of multipliers is an important decision. In our experiments we create synthetic objects only for greatly underrepresented classes, because the performance cost increases linearly with the number of objects that are being oversampled. Even then, the cost of SMOTE dominates the costs of PU learning, training and testing of random forests combined.

# Chapter 6

# Experiments on public datasets

In this chapter, we perform experiments with random forests and other methods presented in this thesis on publicly available datasets. All of the datasets we used are available in a common *LibSVM* format on site [36]. The site aggregates various well known datasets for multiclass classification. We used datasets *usps*, *dna*, *letter* and *satimage* because they are the same datasets that the authors of Mondrian forests used in [30]. Additionally, we also used datasets *aloi* and *mnist* because they are considerably larger than first four datasets. Table 6.1 compares the datasets in terms of size, number of classes and number of features. All of the datasets are available to us in a form of numerical feature vectors.

| | **Train size** | **Test size** | **# of classes** | **# of features** |
|---|---|---|---|---|
| **usps** | 7291 | 2007 | 10 | 256 |
| **dna** | 1400 | 1186 | 3 | 180 |
| **letter** | 15000 | 5000 | 26 | 16 |
| **satimage** | 3104 | 2000 | 6 | 36 |
| **aloi** | 98000 | 10000 | 1000 | 128 |
| **mnist** | 60000 | 10000 | 10 | 780 |

**Table 6.1:** Summary of public datasets. If there was an option to choose how many objects to use in training, we used the same number as the authors of Mondrian forests in [30]. In case of the aloi dataset, it was not specified how the dataset should be split to training and testing. We shuffled the dataset and chose the size of test set to be 10000.

Dataset usps originates from U.S. Postal service by scanning handwritten digits on envelopes. The images were transformed by linear transformation resulting in $16 \times 16$ greyscale images [13].

Dataset dna is interesting because it contains high number of irrelevant features. It therefore has higher demands on the feature selection. It is common practice in some experiments that only 60 best features in the dataset are used and it drastically improves results. We have always used the original 180 features.

The dataset satimage originates from satellite images and each object in the dataset consists of $3 \times 3$ multi-spectral pixel neighbourhood and the goal

is to classify the type of soil in the central pixel [49].

The letter dataset consists of images of English alphabet letters in 20 unique fonts. Each of these images was randomly distorted to produce 20000 unique images. Each distorted image was then transformed into 16 primitive features such as edge counts or statistical moments [33].

The aloi dataset is made of images of 1000 small objects (shoe, light bulb, socks, etc.) taken in a controlled setting with varying illumination, orientation, etc. The goal is to correctly classify the objects. This dataset stands out from other datasets we used with it's size and also with number of classes it contains. We used the version where features consist of 128 raw pixel values.

Mnist dataset is well known. It consists of images of greyscale handwritten digits and the features are raw pixel values. The dataset has the greatest number of features of all of the public datasets we used.

## 6.1 Experiments on the original datasets

None of the datasets is significantly imbalanced, so we use only accuracy as the classification performance evaluation measure. We also compute the average number of nodes in each tree to estimate memory requirements (although nodes in Mondrian trees are much bigger than in the rest of forests) and we compute average weighted depth of trees in the forests to get an insight into how much the trees are balanced. Average weighted depth also provides information about the time cost of prediction of the forests.

In the first experiment we compare different random forest implementations when their hyperparameters are set to default values that are commonly used. Each forest has 100 trees and the stopping criterium *minSamplesToSplit* is always set to 2 so we allow the trees to grow as deep as possible. Breiman forests use $\sqrt{numberOfDimensions}$ features in splits and the size of bagged samples is equal to the size of training dataset. k-ERT trees use $\sqrt{numberOfDimensions}$ split candidates in each split and 1-ERT trees use just a single candidate. Mondrian forests have the lifetime parameter $\lambda$ set to $\infty$ and the discount parameter for their prediction algorithm is set to default value $\gamma = 10 \cdot numberOfDimensions$.

The results from the first experiment are available in figures 6.1, 6.2 and 6.3. Each point was computed by repeating the experiment 10 times on subsets of training data randomly sampled from the whole training dataset and the results were averaged. The forests were trained on the same subsets.

Mondrian forests were trained only on the first four datasets because on the large datasets they were so big that they required more memory than we had available. We can see on Figure 6.2 that there is a trend that Mondrian forests require similar number of nodes as 1-ERTs. If this would hold for the mnist dataset too then Mondrian forests would require about 60000 nodes per tree. Each node needs to keep it's bounding box and with 780 dimensions that means that each node would require at least $780 \cdot 8 \cdot 2 = 12480$ bytes. This means that Mondrian forest model would require more than $12480 \cdot 60000 \cdot 100 \doteq 75$ GB of memory.

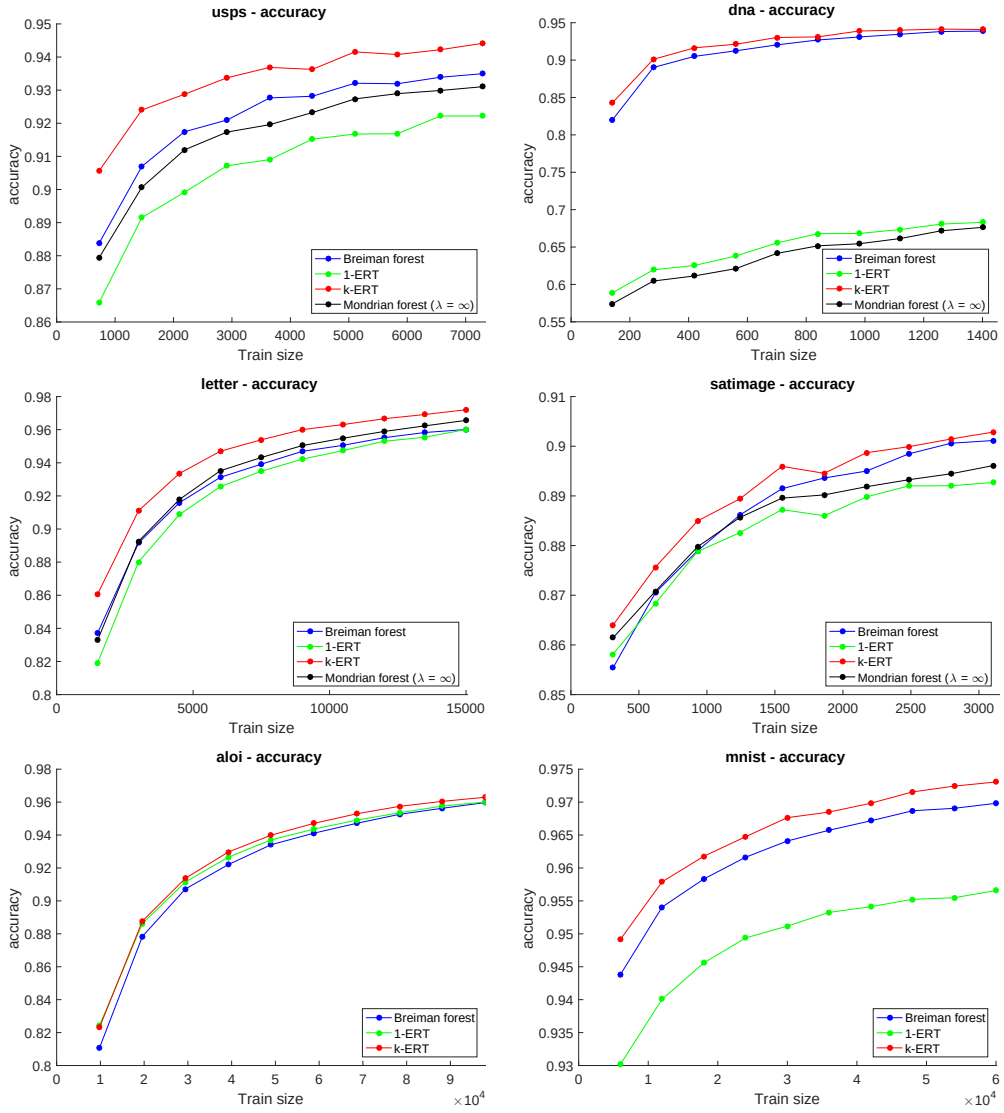



**Figure 6.1:** Accuracies of different random forest types with hyperparameters set to default values.

In terms of accuracy, we see that k-ERTs dominate across all datasets, while Breiman forests are mostly in the second place. Still, it is remarkable how well 1-ERTs and Mondrian forests perform, because they do not use labels during splits in any way. An exception is the dna dataset, which contains lots of irrelevant features and 1-ERTs and Mondrian forests can not handle this. It is interesting that the absolute differences in accuracies between the methods often stay more or less the same on datasets mnist, aloi or usps irregardless of the training size.

When comparing the average number of nodes in a tree the clear winners are Breiman forests, but k-ERTs are not much behind. As mentioned, the number of nodes in 1-ERTs and Mondrian forests tends to be very similar, but we have to take into account that nodes in Mondrian forests have much bigger

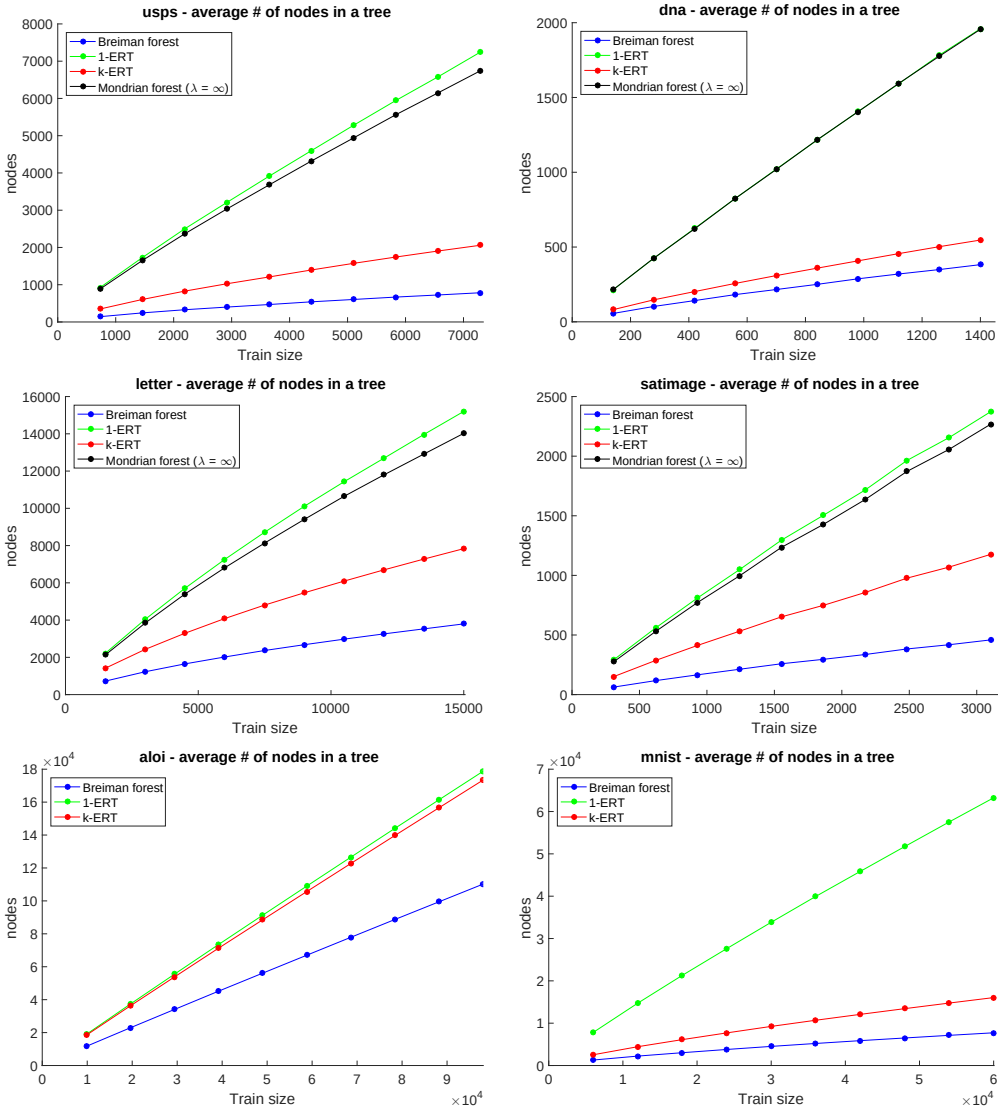



**Figure 6.2:** Average number of nodes in trees of different random forest types with hyperparameters set to default values.

memory footprint than nodes in other methods. Aloi dataset is interesting because in contrast to other datasets k-ERTs tend to have the same number of nodes as 1-ERTs. An interesting observation is that in case of 1-ERTs and Mondrian forests, the number of nodes in a tree seems to be almost equal to the number of nodes in the training dataset. We have also tested 1-ERTs on our network dataset and luckily this rule does not remain true when the size of the training dataset increases even further. When we used training dataset with ten million training object, 1-ERTs trained to a size of approximately ten thousand nodes per tree.

The average weighted depth of trees is often correlated to the number of nodes. An interesting exception is the aloi dataset, because even though 1-ERTs and k-ERTs tend to have similar number of nodes, k-ERTs have much

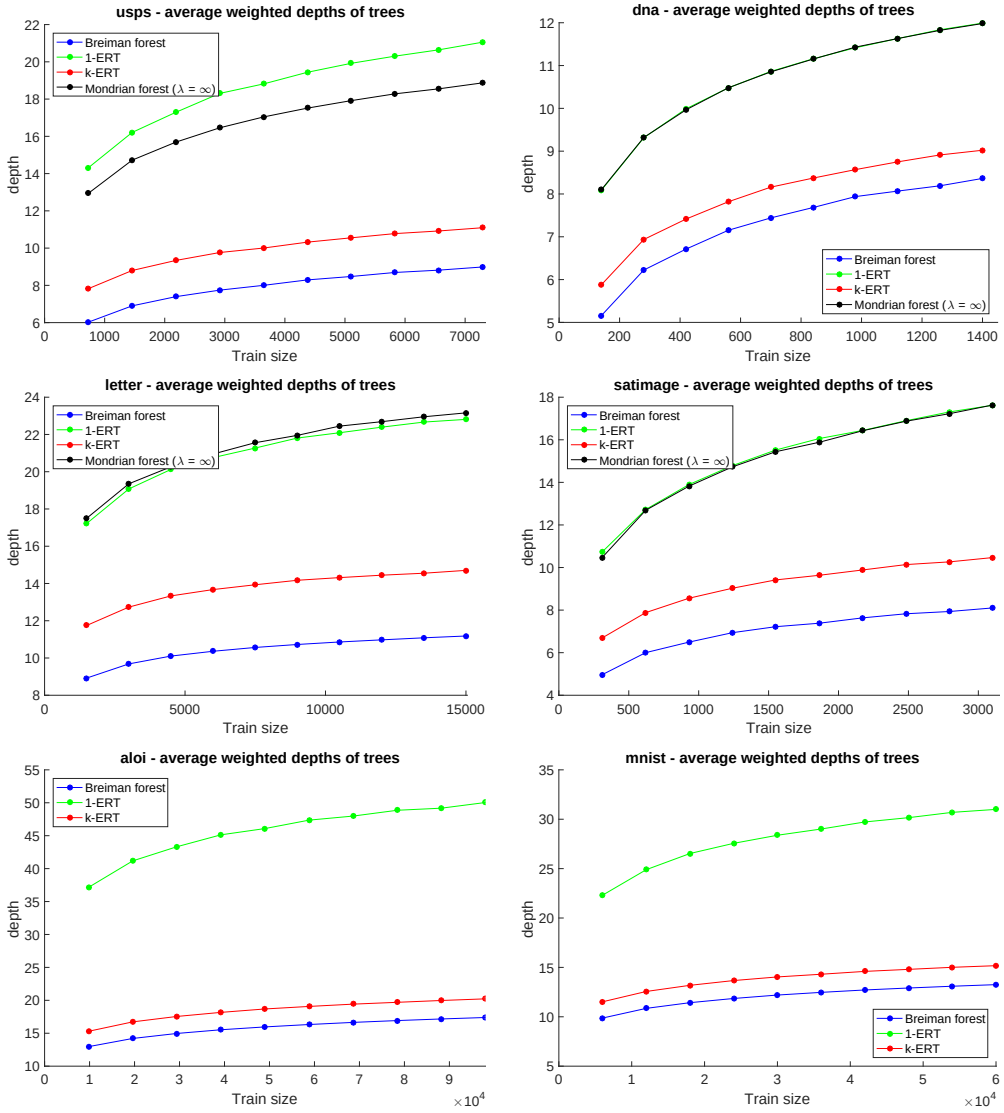



**Figure 6.3:** Average weighted depth of different random forest types with hyperparameters set to default values.

more reasonable average weighted depth. This implies that k-ERTs are much more balanced on the aloi dataset than 1-ERTs.

### 6.1.1 DAS-RF – semi-supervised learning

The experiment with DAS-RF was set up in a similar way to the previous supervised experiments. The part of training dataset that was not used in training in the supervised case was used as unlabeled training dataset for DAS-RF. We did not use more than 50 % of labeled training data, because semi-supervised learning is most helpful if the majority of training data is unlabeled.

Inside of DAS-RF we used Breiman forest with default parameters. The

number of DAS-RF iterations was set to 10 and the $\alpha$ hyperparameter was set to 0.1. This left us to experiment with the temperature and the cooling function. We used exponential cooling function and the rate was computed so that the last iteration would have temperature 0.005. We tested several starting temperatures $T_0$ that we picked according to Figure 5.2. The results are available at Figure 6.4.

The most interesting point in the graphs is the first, where the percentage of labeled training data is 10 %. There is no significant increase in accuracy on datasets letter and satimage, but plots for datasets usps and dna show an accuracy increase in units of percents. We did not implement the airbag mechanism that should ensure that DAS-RF is not worse than the default Breiman forest. With airbag implemented, applying DAS-RF can provide performance gain without much risk. The downside is that the computation time required to train DAS-RF is multiple times larger than the time required to train a single forest because, aside from other heavy computation, a new forest is trained in each iteration of DAS-RF.

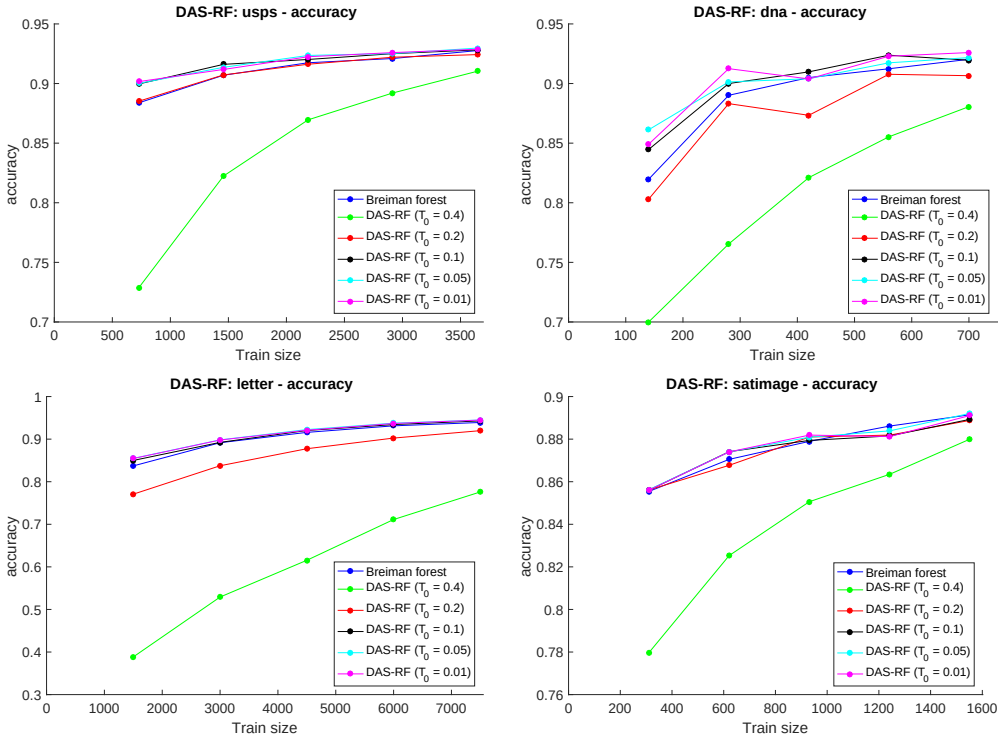



**Figure 6.4:** Accuracies of DAS-RF with different starting temperature values. Breiman forest with default hyperparameters is included for comparison.

### 6.1.2 Mondrian forests – hyperparameter $\lambda$

Because Mondrian forests have enormous memory footprint, we performed experiment to determine if setting the hyperparameter $\lambda$ to different values than $\infty$ can reduce the forest size while retaining a reasonable accuracy. We

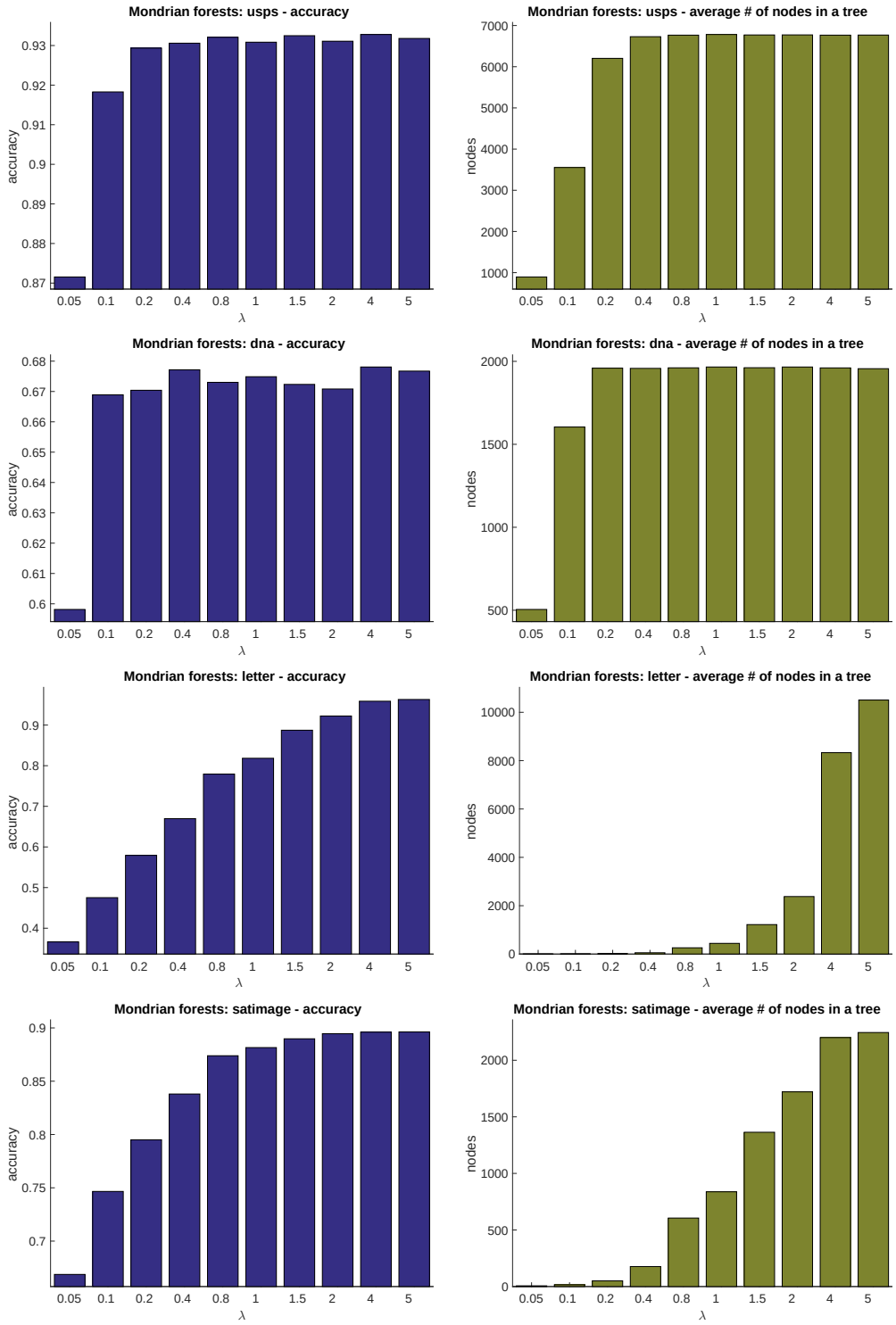



**Figure 6.5:** Accuracies and average numbers of nodes in Mondrian trees with different values of hyperparameter $\lambda$.

determined the values of $\lambda$ to include in the experiment by inspecting the node times $\tau_i$ in nodes of forests trained with $\lambda = \infty$. Only the first four datasets were included in the experiment and we always trained only on full training sets. All of the other hyperparameters than $\lambda$ were set as in the first experiment.

The results are available at Figure 6.5. First of all, we can see that even though all of the datasets are normalized to the $[0, 1]$ interval, there is not a single value of $\lambda$ that would work well on all datasets. Also it looks like that if we aim only for maximum accuracy, then $\lambda = \infty$ is the best choice, because lowering the lambda seems to lower the accuracy.

For dataset letter, the results look good for $\lambda = 2$, because there is a major drop in the number of nodes required but only a minor drop in accuracy. Dataset usps seems to work well with $\lambda = 0.1$ where the number of nodes is reduced by a half but accuracy still remains reasonable. The results on dataset satimage are different in the way that the number of nodes and accuracy reduces more gradually and there is not a single value for $\lambda$ that we would select as the best. Dataset dna is included for completeness but the accuracy is bad even with $\lambda = \infty$ and other values of $\lambda$ do not improve it.

## 6.2 Experiments on datasets modified to contain mislabeled data

We modified the original datasets to explore the imbalanced and mislabeled problem. At first, the datasets were made imbalanced by joining roughly the bottom 80 % of classes together to form a single majority class. An exception to this rule was the dna dataset where the 3rd class was majority already so we left the dna dataset as it was. The exact number of classes for each dataset is shown in Table 6.2.

| | **Total # of classes** | **# of classes after relabeling** |
|---|---|---|
| **usps** | 10 | 3 |
| **dna** | 3 | 3 |
| **letter** | 26 | 7 |
| **satimage** | 6 | 3 |
| **aloi** | 1000 | 200 |
| **mnist** | 10 | 3 |

**Table 6.2:** Summary of the way we made the datasets unbalanced. First column shows the original number of classes in the dataset and the second column shows the number of classes in the modified dataset. One of these classes is always the majority class so, for example, the dataset letter contains 1 majority class and 6 minority classes.

In addition to making the datasets imbalanced we also added mislabeled data to make the datasets resemble Figure 5.1. This was done by randomly sampling 10 % of objects from the minority classes and changing their label

to the majority class. The mislabeling was performed only on the training datasets and the testing datasets were kept clean.

We adapt the terminology here to the terminology we use when dealing with our network data and call the majority class *negative* and the minority classes *positive*. Because the datasets are imbalanced we are not interested in accuracy anymore for the reasons explained in Section 2.4, but measure average precision and recall on the positive classes. The negative class is not included in the averages, because it is not a single class but is made of several distinct classes and the performance on it is not that important to us.

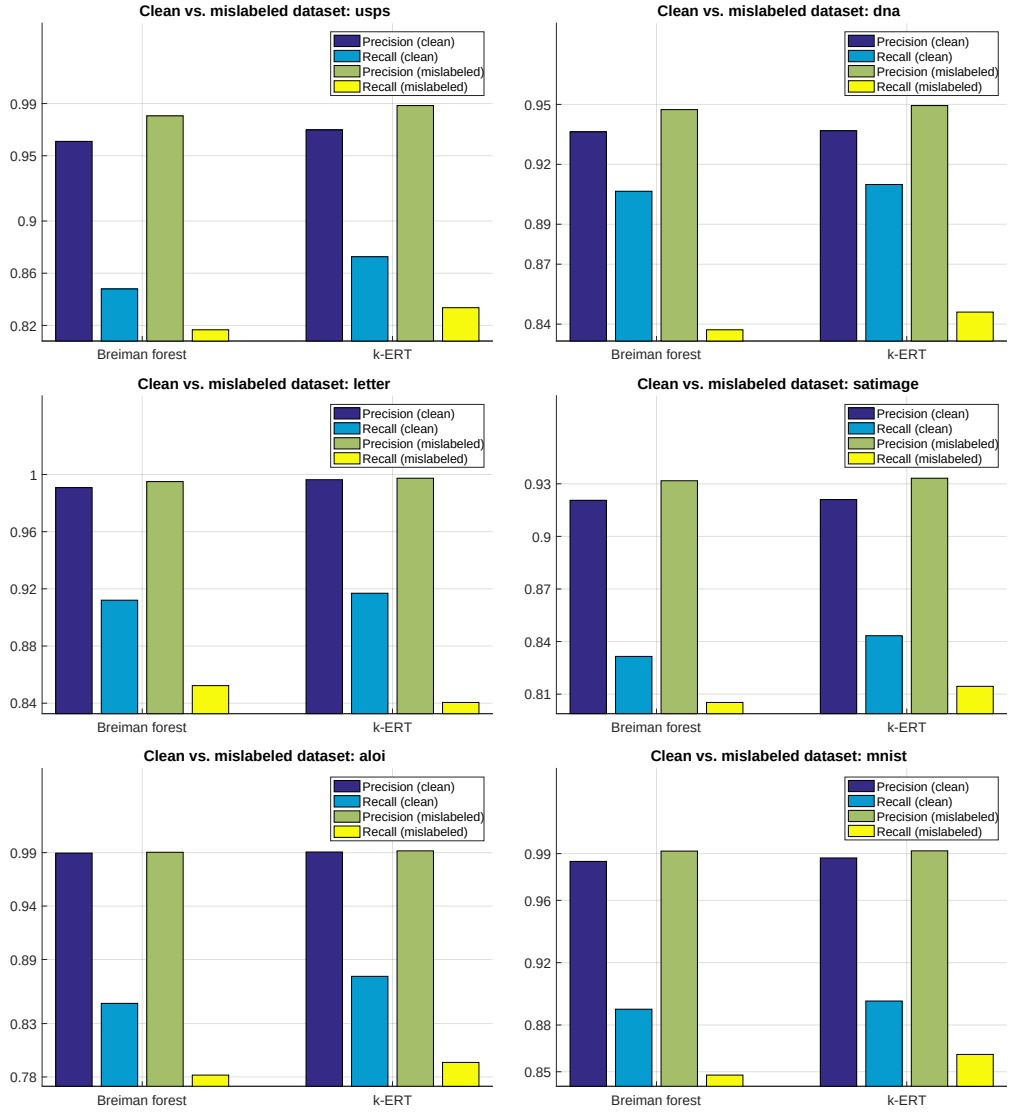



**Figure 6.6:** Comparison of perfomance between scenarios when the training datasets were clean and when they contained mislabeled data. The reported precisions and recalls are averages over all positive classes.

Because of the results of the experiments on the original datasets, we decided to further use only Breiman forests and k-ERTs with the same

hyperparameters as we used before. We always trained only on the full training datasets and each experiment was repeated 5 times and we show the average values.

Figure 6.6 shows the differences in performance, when the training dataset is clean (the mislabeling was not performed) and when the training dataset contains mislabeled data. We can see that the presence of mislabeled data always reduced the average recalls over positive classes. This happens because some of the positive objects that would be classified correctly are now classified as the negative class. Interestingly, we can observe increase in precisions in the mislabeled cases. This is understandable, because by mislabeling positives to negatives the number of objects that are marked as positives is decreased and some objects that would be false positives are now classified correctly. In the following experiments, we aim to increase the recalls in the mislabeled cases while keeping the precisions preferably as high as in the cases with clean training datasets.

### 6.2.1 SMOTE

We applied SMOTE on the positive classes as a preprocessing step on the training dataset before running training either Breiman forest or k-ERT. The number of nearest neighbours in SMOTE was always set to 5 as in the original publication [11]. We performed five experiments with different values of the SMOTE multipliers. In this section, we denote the SMOTE algorithm with the number of multipliers set to $k$ as 'SMOTE k'. The results are available at Figure 6.7.

From the results, it looks like there is not a single value of multiplier that would be the best in all cases. Generally, recall increases and precision decreases when we increase the number of multipliers. The exception to this rule is the mnist dataset where recall slightly decreases if we use Breiman forests. We encounter a similar phenomenon on our network dataset. Both datasets have in common that the number of features is very large and it might be possible that the nearest neighbour does not work well in those cases. However, that does not explain why k-ERTs suffer as well.

When we compare the results with the best results on the clean datasets achieved by k-ERTs in Figure 6.6, we see that we can get very close and sometimes we can even improve the overall performance.

On dataset usps k-ERTs achieved 97 % precision and 87 % recall on the clean dataset. SMOTE 4 with k-ERTs achieved 97 % precision and 90 % recall. SMOTE also did get close in performance to the benchmark on clean datasets on datasets letter (SMOTE 4 with k-ERTs) and aloi (SMOTE 8 with k-ERTs). There is a 2 % drop in precision on dataset satimage with SMOTE 3 and k-ERTs but on the other hand there is a 4 % increase in recall. On mnist and dna the results on clean datasets are slightly better but the best results with SMOTE are still better than the results without SMOTE on mislabeled datasets.

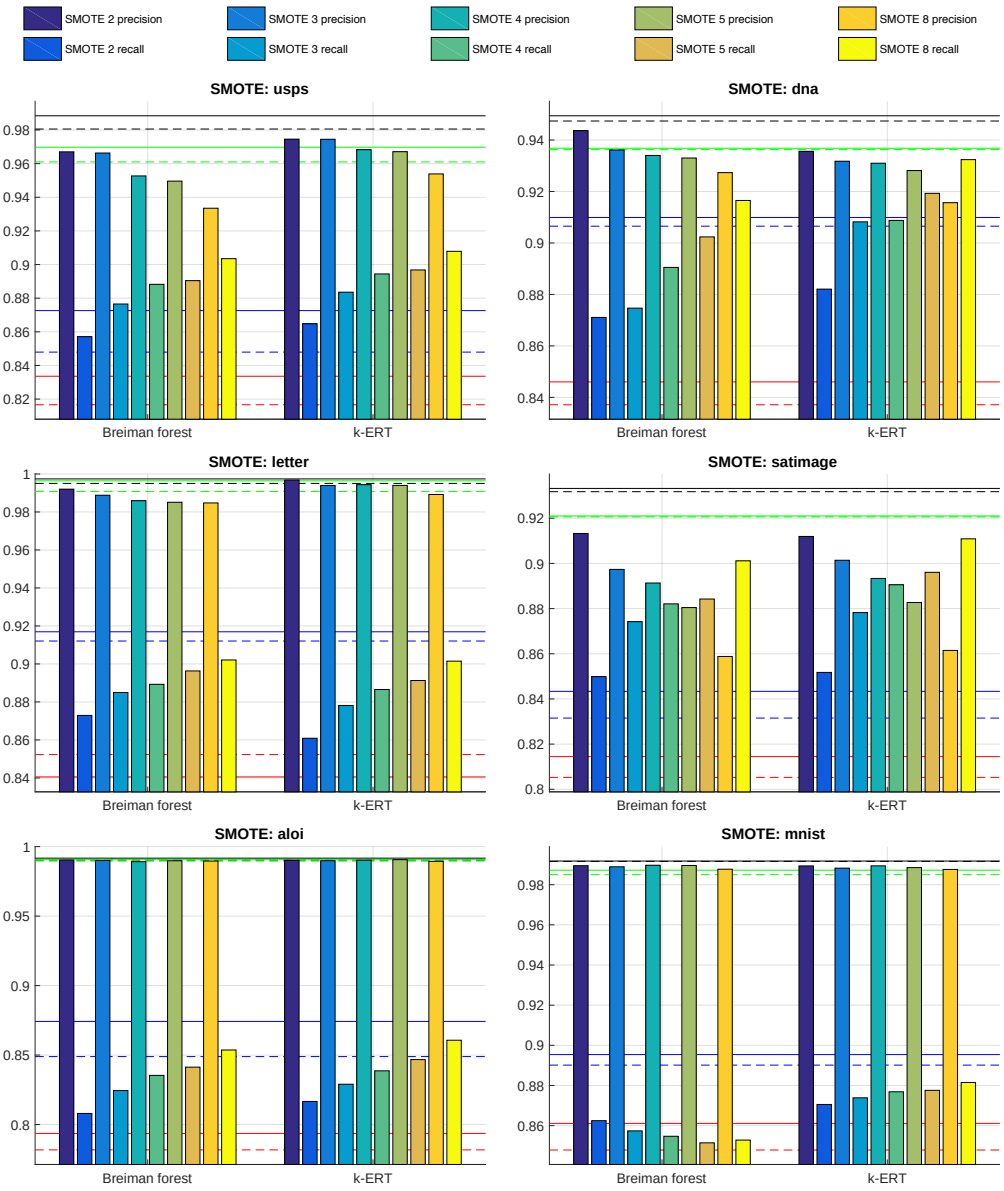



**Figure 6.7:** SMOTE applied as a preprocessing step on mislabeled datasets. The number of nearest neighbours in SMOTE was always fixed to 5. We tested SMOTE with several number of multipliers i.e. *SMOTE 2* means that after SMOTE the positive classes contain twice the number of objects, *SMOTE 3* means that they contain three times the number of objects etc. The reported precisions and recalls are averages over all positive classes. The horizontal lines refer to values in Figure 6.6 with clean and mislabeled data. Breiman forest results are drawn with full lines and k-ERT results with dashed lines. Recalls on clean dataset are blue, recalls on mislabeled dataset are red, precisions on clean dataset are green and precisions on mislabeled dataset are black.

### 6.2.2 PU learning

PU learning aims to identify the mislabeled objects in the negative class and filter them out. Before evaluating the performance of PU learning together with random forests, we performed experiment to identify reasonable values for the algorithm's hyperparameters. The only hyperparameter we did not experiment with was the spies ratio which we always set to 15 % as did the authors of the PU learning algorithm in [35]. We always used Breiman forest inside of the PU learning algorithm because they work better on network data. For other hyperparameters we selected base values that we found reasonable. Base value of the bagging ratio of the forest inside of PU learning was set to 100 %, *minSamplesToSplit* was set to 20 because we intended to give the forest inside slightly more bias, number of trees in the forest was set to 100 and the noise ratio which determines the threshold inside of PU algorithm was set to 1 %.

Then we performed several experiments. In each experiment, we modified value of a single hyperparameter and the rest was set to their base values. We hoped that this approach, although it was naive because it assumed independence between the hyperparameters, would provide us with some insight into the behaviour of PU learning with different hyperparameter settings and help us identify the important hyperparameters. In each experiment we measured recall (true positive rate) of the mislabeled objects that were correctly identified in the negative class and filtered out by the PU learning algorithm. We also measured the fall-out (false positive rate) which tells us about the ratio of true negatives that the algorithm filtered incorrectly. Each experiment was performed five times and we report the average numbers. The results are available in Table 6.3.

When we repeated the same experiment five times, TPRs were stable but, on the other hand, the standard deviation of FPRs is often around 5 % and in some rare cases it is even more. Because of that, it is not possible to make any conclusions when FPRs are different by single units of percents, because that can be attributed to random noise. After studying the results, we came to conclusion that the only hyperparameter that we will modify in the following experiment where we perform classification after PU learning is the noise ratio. Noise ratio looks like it can control the tradeoff between TPRs and FPRs most effectively. In fact it is directly related to TPR, because it sets the ratio of spies that will not be identified as positives. We decided to set the rest of hyperparameters to their base values because there is not apparent any consistent significant improvement in Table 6.3 when their base values were modified.

In the following experiment we performed PU learning with different noise ratios and then we trained either k-ERT or Breiman forest as in the previous experiment with SMOTE. Each experiment was repeated 5 times and we report the averages. The results are available at Figure 6.8.

When compared with SMOTE, PU learning behaves a bit differently. In contrast to SMOTE there is more often visible a significant drop in precision for each increase in recall, but sometimes this is in situations where PU

| PU hyperparams | usps | | dna | | letter | |
|---|---|---|---|---|---|---|
| | **TPR (%)** | **FPR (%)** | **TPR** | **FPR** | **TPR** | **FPR** |
| Bagging 100 % | 97.3 | 5.6 | 98.8 | 15.4 | 97.8 | 8.4 |
| Bagging 75 % | 98.8 | 6.8 | 98.2 | 14.4 | 98.4 | 14.8 |
| Bagging 50 % | 97.8 | 8.5 | 100.0 | 31.1 | 98.4 | 19.3 |
| Min. split 20 | 97.8 | 5.3 | 96.1 | 8.2 | 98.0 | 8.7 |
| Min. split 10 | 97.8 | 5.6 | 98.2 | 20.4 | 98.4 | 7.4 |
| Min. split 2 | 98.3 | 7.1 | 99.1 | 47.7 | 98.3 | 11.3 |
| 100 trees | 99.2 | 7.9 | 97.6 | 10.6 | 98.5 | 12.0 |
| 50 trees | 97.6 | 6.7 | 98.2 | 15.4 | 98.3 | 14.0 |
| 10 trees | 99.5 | 59.7 | 98.5 | 31.8 | 97.8 | 15.9 |
| **Noise 10 %** | **91.5** | **0.7** | **87.6** | **4.7** | **89.4** | **0.6** |
| **Noise 5 %** | **93.4** | **1.5** | **93.6** | **6.5** | **93.6** | **1.6** |
| **Noise 1 %** | **98.5** | **5.1** | **100.0** | **14.1** | **98.7** | **14.3** |
| **Noise 0.5 %** | **97.6** | **6.8** | **99.7** | **23.0** | **98.4** | **13.0** |
| PU hyperparams | satimage | | aloi | | mnist | |
| | **TPR (%)** | **FPR (%)** | **TPR** | **FPR** | **TPR** | **FPR** |
| Bagging 100 % | 99.3 | 16.9 | 99.0 | 9.8 | 99.2 | 7.4 |
| Bagging 75 % | 99.0 | 17.0 | 99.2 | 11.6 | 99.2 | 7.8 |
| Bagging 50 % | 98.8 | 18.2 | 99.1 | 17.3 | 99.3 | 10.5 |
| Min. split 20 | 98.3 | 13.2 | 99.1 | 9.4 | 99.0 | 6.7 |
| Min. split 10 | 98.8 | 17.9 | 99.2 | 8.1 | 99.3 | 8.2 |
| Min. split 2 | 98.3 | 30.4 | 99.5 | 18.0 | 99.3 | 12.1 |
| 100 trees | 99.3 | 15.5 | 99.3 | 11.4 | 99.2 | 7.6 |
| 50 trees | 99.0 | 18.4 | 99.3 | 12.1 | 99.4 | 10.7 |
| 10 trees | 99.3 | 38.8 | 98.8 | 28.1 | 99.4 | 26.5 |
| **Noise 10 %** | **88.7** | **2.7** | **89.6** | **0.4** | **90.0** | **0.5** |
| **Noise 5 %** | **95.9** | **6.7** | **94.6** | **1.4** | **95.1** | **1.4** |
| **Noise 1 %** | **98.6** | **15.8** | **99.1** | **10.3** | **99.2** | **7.4** |
| **Noise 0.5 %** | **97.8** | **12.5** | **99.7** | **18.0** | **99.5** | **10.4** |

**Table 6.3:** Results of experiments with different values of hyperparameters of the PU learning algorithm. TPR is the true positive rate (recall) and it tells how many of the mislabeled positives that were mixed into negatives were discovered by PU learning and removed from the dataset. FPR is the false positive positive rate (fall-out) and it tells how many of the true negatives were falsely removed by PU learning. Rows where we modified the noise ratio are highlighted because we decided that we would only modify noise ratio in the experiment where we perform classification after PU learning.

learning already outperformed SMOTE.

It is sometimes difficult to convincingly infer that one method behaves better than the other because we did not explore the full space of hyperparameters. This is the case on datasets letter, aloi and mnist. PU learning achieved similar performance to SMOTE and then with lower noise ratios it was able to score considerably higher in recall than what we see in the SMOTE results.

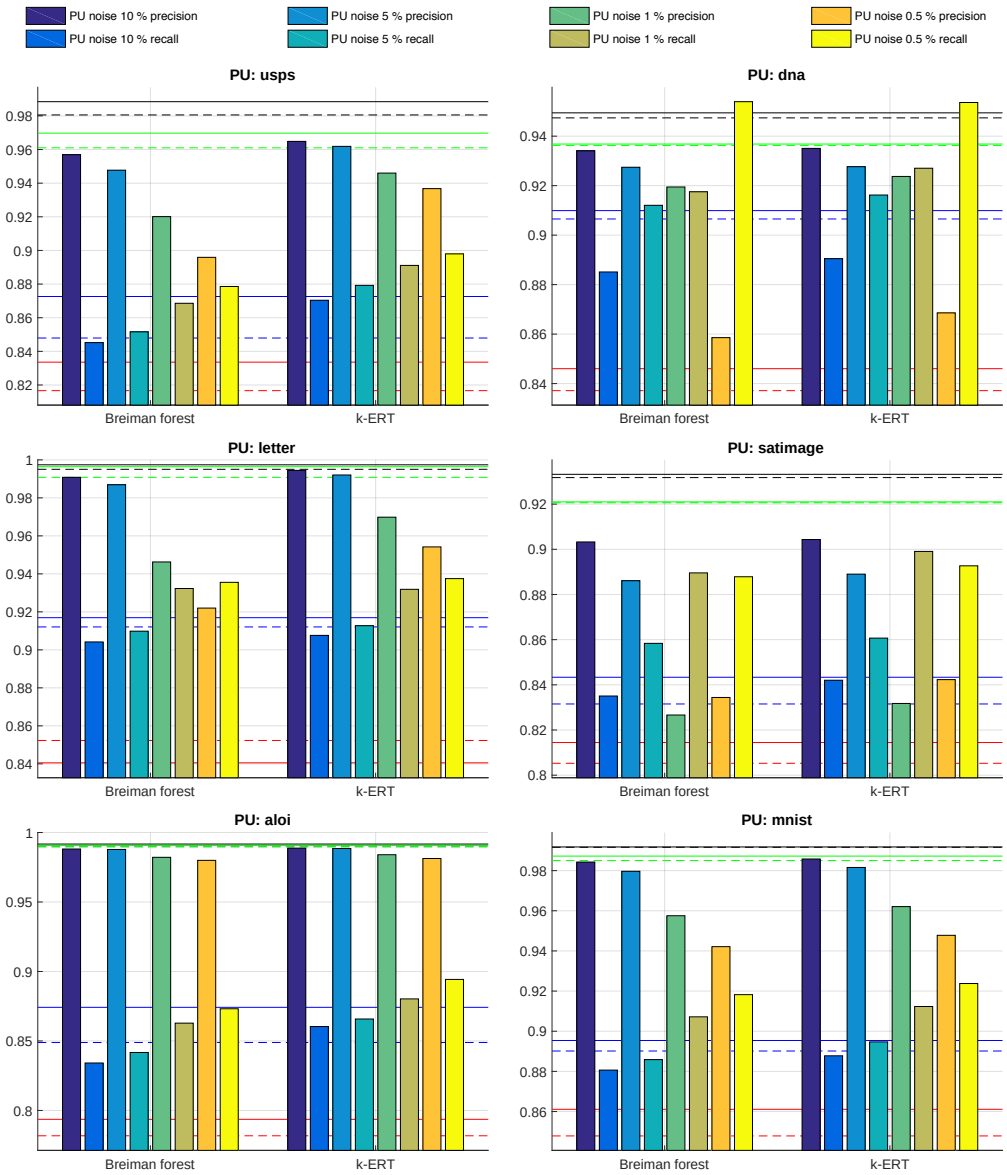



**Figure 6.8:** PU learning applied as a preprocessing step on mislabeled datasets before training either Breiman forest or k-ERT. We tested PU learning with different values of the hyperparameter *noiseRatio*. The reported precisions and recalls are averages over all positive classes. The horizontal lines refer to values in Figure 6.6 with clean and mislabeled data. Breiman forest results are drawn with full lines and k-ERT results with dashed lines. Recalls on clean dataset are blue, recalls on mislabeled dataset are red, precisions on clean dataset are green and precisions on mislabeled dataset are black.

On dataset usps SMOTE is actually better than PU learning because when PU learning achieves the same recall (90 %) it has 3 % less precision. On dataset satimage PU learning is also unable to achieve as high precision as SMOTE did with the same recalls. Performance on dataset dna is comparable.

Table 6.3 shows that on all datasets, PU learning is able to filter out almost

every mislabeled object. An interesting observation is that PU learning sometimes achieves better recall than the recall that was achieved on clean dataset in Figure 6.6. This is not only because of correctly identified mislabeled objects but also because of those true negatives that are removed from the negative class. This effectively achieves undersampling which is commonly used when dealing with mislabeled data. The removed true negatives tend to be more important than random negative objects, because they tend to be closer to the boundary between classes. This increase in recall is paid for by the decrease in precision and also by the fact that the undersampling is somewhat hidden and it can be too easy to forget about it. The nice thing about PU learning is that if we settle with smaller recall of the mislabeled objects then the fall-out can be tuned by hyperparameters to an extent and the amount of undersampling can be made insignificant. In that case, PU learning can almost provide a net benefit, because in the worst case it does nothing and it can potentionally remove at least some mislabeled objects.

Compared to SMOTE the time cost of PU learning is much smaller. The defining part of the cost is the training of the inner forest so PU learning requires similar amount of time as the actual training of the random forest after.

### 6.2.3 DAS-RF

In the experiments with DAS-RF we used only the small datasets because the datasets aloi and mnist would require too much computation time. We used the same hyperparameters as in the experiment on original datasets in Section 6.1.1. As a starting temperature we always used value 0.1, because it performed well in the first experiment.

To be able to handle mislabeled data, original DAS-RF requires modification that is described in Section 5.2.3. In short, we sampled a part of the negative class to the initial training dataset which would otherwise consist only of positive data. This allows the forest inside of DAS-RF predict negative labels. The percentage of the negative class that is sampled to the initial dataset is an additional hyperparameter. We performed experiments with different values of this ratio. Each experiment was repeated 5 times and the averages are available in Figure 6.9. The results should be compared mainly with the results using Breiman forests in previous experiments because our DAS-RF implementation uses only Breiman forests inside. Breiman forest is used because it performs better on the network dataset and we designed the algorithm primarily for that.

While not ignoring the results achieved by k-ERTs in previous experiments it is reasonable to primarily compare DAS-RF results with the results achieved by Breiman forests. By analysing the results we can see that they are greatly influenced by the negative ratio hyperparameter. When the negative ratio is really small, DAS-RF achieves recalls that are higher than in all of the previous results. This is however compensated by very poor precision. As the negative ratio increases the results stabilize at values that are closer to results of SMOTE and PU learning.

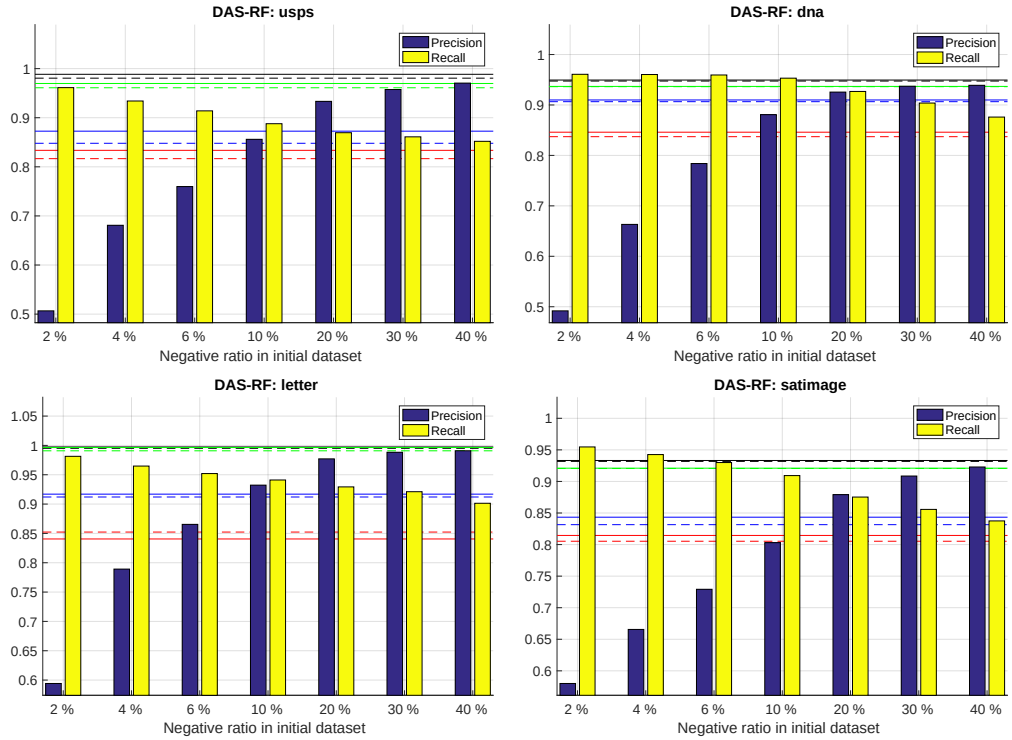



**Figure 6.9:** DAS-RF applied on mislabeled datasets. We tested different percentages of the negative class sampled to the initial training dataset in DAS-RF. The reported precisions and recalls are averages over all positive classes. The horizontal lines refer to values in Figure 6.6 with clean and mislabeled data. Breiman forest results are drawn with full lines and k-ERT results with dashed lines. Recalls on clean dataset are blue, recalls on mislabeled dataset are red, precisions on clean dataset are green and precisions on mislabeled dataset are black.

On dataset letter DAS-RF achieves 99 % precision and 92 % recall which is actually better than the results of both SMOTE and PU learning. On dataset usps, Breiman SMOTE achieves almost 97 % recall with 88 % recall. DAS-RF has worse results than that but the result with 30 % negative ratio achieves 96 % precision and 86 % recall which is also reasonable. On dataset dna the results of DAS-RF are comparable with the results of SMOTE and PU learning. We would again pick the result with negative ratio 30 % which achieves 94 % precision and 90 % recall. On dataset satimage, again with negative ratio 30 %, DAS-RF achieves precision 91 % and almost 86 % recall. This result is also comparable maybe even better than SMOTE.

In overall it looks like setting the negative ratio to 30 % could be a reasonable default choice, because all of the reported results above were achieved with that value.

# Chapter 7

# Experiments on network data

In this chapter, we present summary of the experiments we performed on our network datasets with the aim to identify and classify malware. Contrary to the experiments in Chapter 6, these datasets are not publicly available. The following description of the datasets is inspired by [4].

The datasets have the form of proxy logs. Objects in these datasets are HTTP flows, where each flow represents a single communication between user and a server. Although we had HTTPS flows available too, we did not use them, because their features were too noisy therefore we restricted ourselves only to HTTP flows. Flows from proxy logs are bidirectional so both directions of the communication are included in a single flow. Each flow contains fields like: URL, referer, source and destination IP addresses and ports, HTTP status, number of bytes transferred, etc. [4]

| **Features** | **Features on all URL parts + referer** |
|---|---|
| duration | length |
| HTTP status | digit ratio |
| is URL encrypted | lower case ratio |
| is protocol HTTPS | upper case ratio |
| number of bytes uploaded | vowel changes ratio |
| number of bytes downloaded | has repetition of '&' and '=' |
| is URL in ASCII | starts with number |
| client port number | number of non-base64 characters |
| server port number | has a special character |
| user agent length | max length of consonant stream |
| MIME-Type length | max length of vowel stream |
| number of '/' in path | max length of lower case stream |
| number of '/' in query | max length of upper case stream |
| number of '/' in referer | max length of digit stream |
| is second-level domain raw IP | ratio of a character with max occurrence |

**Table 7.1:** List of features extracted from proxy log [4]. The features in the right column are extracted from each part (protocol, second-level domain, tld, path, filename, query, fragment) of URL and referer.

In total, 357 features were extracted from each flow and the most informative

features originate from the URL. For illustration, some of the features are available in Table 7.1.

Initially all of the flows were unlabeled. Positive labels were added to some flows by using available blacklists, other malware feeds from Collective Intelligence Framework (CIF) [21] or they were created by a human analyst. The labeling was performed at the domain level and not per individual flows. Nevertheless, the majority of objects in the datasets remains unlabeled. Most of these unlabeled objects belong to the innocent negative traffic and they are therefore assigned label *negative*. Still, there are some malicious objects for which we did not have labels at the time when we labeled the datasets and they are assigned the *negative* label incorrectly as a result. This makes the structure of the datasets similar to the datasets we used for the experiments on public datasets containing mislabeled data in Section 6.2.

Newly obtained labels were not retrospectively applied on the training dataset but they were used for evaluation of methods like PU learning.

There are 86 classes of malware available in the datasets. Each of the classes is related to a traffic associated with different malware categories, such as ransomware, exploit kit, exfiltration, ad-injector, etc.

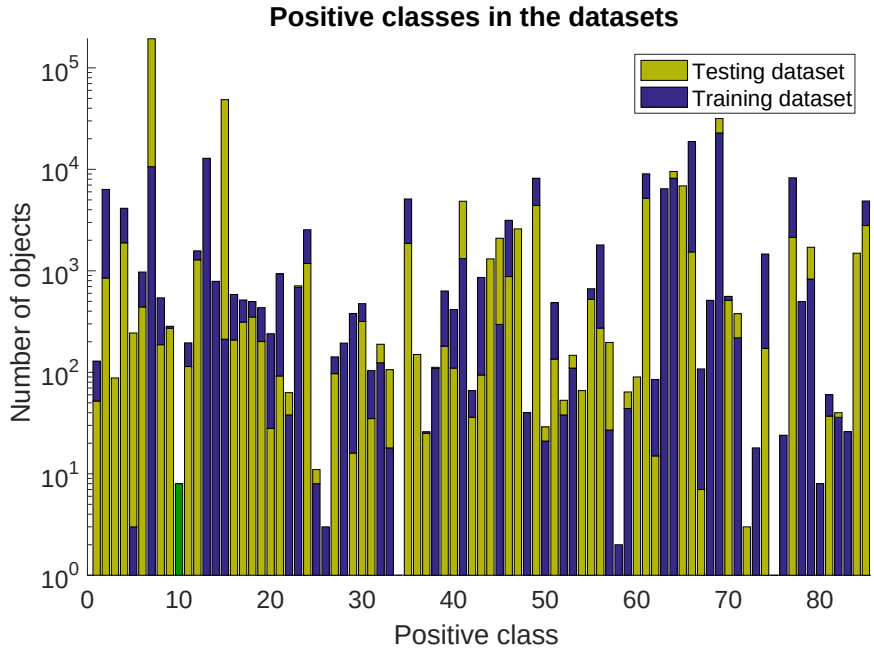



**Figure 7.1:** Size of the positive classes in the training and testing datasets. For each class the smaller bar is in the front and the larger bar is in the back. The Y axis uses log scale.

We decided to use dataset originating from proxy logs recorded during 3 days in October and November 2015 for training and dataset from proxy logs originating in January 2016 for testing. The training dataset contains $3,359,466$ objects and the testing dataset contains $10,895,786$ objects. Not all classes that are available in the testing dataset are available in the training dataset and vice versa. Figure 7.1 shows the distribution of positive classes

both in the training and testing datasets. It is apparent that even the positive classes alone are are highly imbalanced and the distributions of classes are different between the two datasets.

There are several ways how to evaluate the results. First, they can be evaluated on a per-flow basis. This is the most straightforward way, where each flow is evaluated in isolation and measures like precision and recall are computed for all classes.

Each flow is associated with a user to which it belongs. If, for example, there are 1000 flows in the testing dataset and each of them belongs to the same class and user and each is classified as the same class, it might make sense to include the result only once instead of including it thousand times. This way, the recall of a class does not represent the fraction of flows that were correctly identified to belong in this class, but it represents the fraction of users which had flows correctly identified.

The previous per-user evaluation might report the same class for a user to be a *true positive* and *false positive* at the same time. This might happen if there exist both a positive flow belonging to the user that was classified correctly and negative flow that was classified positive by mistake. We decided to also perform evaluation which we call *benevolent user statistics*, because the standard per-user evaluation is possibly unnecessarily strict. In benevolent user statistics, we do not include false positives and false negatives if there exists a single true positive for a given class-user combination, because the user was identified correctly to be infected by the given malware class.

Still, if for example, there are two flows belonging to the user where one is positive and the other is negative and the classifier classified both of them incorrectly, we do not classify the user to be a true positive in the given class but it remains counted both as a false positive and false negative.

The evaluation schemes considering users have a slight problem that it is a bit random which user's traffic was successfully labeled and which remained unlabeled. This makes the results not as transparent as the straightforward results computed over all flows. The interpretation of the per-user statistics requires deeper knowledge of the datasets. For that reason we mainly focus on the evaluation per-flow when comparing results.

Based on the experiments on public datasets and also on some prior experiments on other smaller network datasets, we decided to evaluate only Breiman forests and k-ERTs on these two network datasets. The following hyperparameters are used in all experiments in this chapter.

The datasets contain 357 features and therefore the hyperparameter $k$ in k-ERTs, as well as the number of features used in Breiman forest split nodes, was set to 19 because $\sqrt{357} \doteq 19$ which is a common value for these hyperparameters as described in Chapter 4. We also experimented with other values for these hyperparameters but that did not lead to any significant performance increase.

Based on prior experiments on smaller network dataset we set the hyperparameter *minSamplesToSplit* to 20 because it led to minor performance increase over the default value 2 that was used in Chapter 6. In these prior

experiments we also discovered that 20 trees in Breiman forests is enough to achieve similar performance as with 100 trees. Because of that we always used 20 trees in Breiman forests for experiments reported in this chapter. This allowed us to compute the experiments faster. The k-ERTs still use 100 trees because their drop in performance with only 20 trees was big.

Additionally, because the datasets are significantly imbalanced we used balanced bagging instead of standard bagging in Breiman forests. If the number of objects in a class was lower than 50, then no sampling was performed on the class during the training process of each tree, but all objects belonging to the class were included. That way it is guaranteed that all of the trees in the forest can classify all classes in the training dataset. However, when experimenting on smaller network datasets we discovered that the difference between balanced bagging and normal bagging is only minor if there is any. This might be explained by Figure 4.1 which shows that it is highly probable that a given object will be included in the majority of trees even if normal bagging is used.

| Method | Flows | | Users | | Benevolent users | |
|---|---|---|---|---|---|---|
| | Prec. (%) | Rec. (%) | Prec. | Rec. | Prec. | Rec. |
| Breiman | 97.4 | 62.8 | 93.0 | 59.5 | 94.2 | 66.8 |
| k-ERT | 97.5 | 57.8 | 93.2 | 58.0 | 94.5 | 64.1 |

**Table 7.2:** Comparison of results on the network datasets between just Breiman forests and k-ERTs without any additional methods. Reported precisions and recalls were averaged over all positive classes.

As a first experiment, we decided to compare just plain Breiman forests and k-ERTs on the described network datasets. The results are available in Table 7.2. We report average precisions and recalls over all positive classes in the three possible evaluation schemes mentioned earlier. Contrary to the experiments on public datasets Breiman forests achieve better recalls than k-ERTs here. The precisions are similar for both methods.

An interesting note about precision is that it is possible that some of the false positives are in fact true positives but we have not received correct labels for them yet. The reported results are therefore lower bounds on precision. When we analyzed the errors further we discovered that the vast majority of errors is between the negative class and some of the positive classes. It was very rare when positive classes were misclassified between each other.

The reported precisions have estimated standard deviation under 0.5 % and the reported recalls have standard deviation around 1 %.

## 7.1 PU learning

In the following experiment we applied PU learning on the training dataset and then we trained Breiman forest and k-ERTs on it with the same hyper-parameters as in the first experiment.

We followed the standard setup of PU learning parameters that was used in public experiments, but after some experimentation we decided to set the PU learning parameters a bit differently because it did not work well with the standard values. The number of trees of the forest inside PU learning was set to 10, *baggingRatio* was set to 60 %, *noiseRatio* to 0.5 %, *spiesRatio* to 15 % and *minSamplesToSplit* to 20. The most important parameter is the *noiseRatio* and we needed to set it lower than it was necessary in case of the public datasets, to be able to filter out a reasonable number of positive objects inside of the negative class. This might have been caused by the size of the network dataset which is incomparable to the public datasets and even 0.5 % is a sufficient number of objects here.

As mentioned earlier in this chapter, we possess additional positive labels that were not provided to the algorithms. Concretely, we possess positive labels for 7103 objects that were marked as negative in the training dataset. In the ideal case PU learning would be able to filter out all of these objects. To test this ideal case, we include result where these 7103 objects are manually removed from the training dataset and then Breiman forest is trained. Table 7.3 shows the results of the experiments with PU learning.

| Method | Flows | | Users | | Ben. users | |
|---|---|---|---|---|---|---|
| | Prec. (%) | Rec. (%) | Prec. | Rec. | Prec. | Rec. |
| Clean | 97.7 | 69.4 | 93.4 | 65.2 | 94.7 | 71.2 |
| PU + Breiman | 96.9 | 64.4 | 92.1 | 61.4 | 93.5 | 68.2 |
| PU + k-ERT | 97.3 | 62.3 | 93.1 | 59.5 | 94.5 | 66.0 |

**Table 7.3:** Comparison of results on the network datasets between PU learning followed by Breiman forests and PU learning followed by k-ERTs. The method 'Clean' denotes the experiment that is described in the text where 7103 known positive objects in the negative class are manually removed before training the Breiman forest. Present precisions and recalls were averaged over all positive classes.

PU learning on average filtered out 1460 of the positive objects for which we possess labels and it filtered out 1761 objects on average. It is possible that some of those filtered objects for which we do not have labels might also be positive but we did not perform any additional analysis of the these objects.

When compared with how well PU learning performed on the public datasets, particularly when considering the very high recall of the positives in the negative class, the results on the network dataset are not as strong. Still, PU learning provides a benefit over the baseline case when it was not used in Table 7.2, because when using it we are consistently able to achieve higher recalls while precisions remain close to the baseline.

The experiment with the known mislabeled objects removed ('Clean' in Table 7.3) shows that there is still potential to increase the recall by 5 % if PU learning were more successful.

## 7.2 SMOTE

After we have established, both in the previous section and in Chapter 6 with public experiments, that PU learning is beneficial to the performance, we decided to always use PU learning in all further experiments. In this section, we describe experiments with PU learning and SMOTE combined. We always used 5 nearest neighbours in SMOTE which is the same value as in the experiments with SMOTE on public datasets in Section 6.2.1. We performed SMOTE only on positive classes with less than 2000 objects because it is supposed to be used on the underrepresented classes and it also made the computation faster. As in Section 6.2.1, we denote the SMOTE algorithm with the number of multipliers set to $k$ as 'SMOTE k'.

| Method | Flows | | Users | | Ben. users | |
|---|---|---|---|---|---|---|
| | Prec. (%) | Rec. (%) | Prec. | Rec. | Prec. | Rec. |
| PU only | 96.9 | 64.4 | 92.1 | 61.4 | 93.5 | 68.2 |
| SMOTE 10 + PU | 96.2 | 62.3 | 90.8 | 59.4 | 92.3 | 66.0 |
| PU + SMOTE 10 | 96.8 | 60.6 | 91.9 | 57.8 | 93.1 | 63.6 |
| PU + SMOTE 3 | 96.9 | 63.9 | 91.8 | 61.7 | 93.2 | 67.8 |

**Table 7.4:** Comparison of results on the network datasets with SMOTE and PU learning combined. Breiman forest was trained in all of the experiments to perform classification so it is not explicitly mentioned. The first row 'PU only' is the results from Table 7.3 which serves as a baseline. Reported precisions and recalls were averaged over all positive classes.

The results of our experiments are available in Table 7.4. The results are a bit surprising because we observe an actual decrease in recall when SMOTE is used which is certainly counterintuitive considering the way SMOTE works.

We suspected that it was possible that by oversampling the positive classes we now made more classification errors across positive classes. However, by inspecting the individual errors we discovered that it was not the case and that the vast majority of errors were still positive objects classified as negative.

Still, it is possible that oversampling of positive classes may cause negative prediction that would not otherwise happen. Consider a simple example where the result of classification of object $\mathbf{x}$ is that $\mathbf{x}$ belongs to class $A$ with probability 60 % and it belongs to the negative class with probability 40 %. This would result in classification of $\mathbf{x}$ as $A$. Now, if we apply SMOTE on class $B$, the result of classification may be that with probability 25 % $\mathbf{x}$ belongs to class $B$, with probability 35 % it belongs to class $A$ and with probability 40 % it belongs to the negative class. Result of this would be classification of $\mathbf{x}$ as negative even though no synthetic negative objects were added to the training dataset.

We attempted to counter this scenario by implementing a simple method which we internally called *two-stage prediction*. When using two-stage prediction we do not assign the result of classification simply to the class with

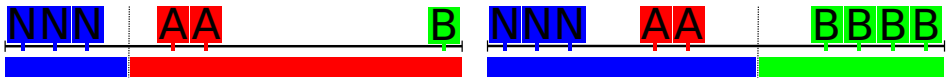



**Figure 7.2:** Example how SMOTE can expand the decision region of negative class during node splitting when SMOTE is applied on positive class. The figure on the left shows how decision tree would split three classes based on entropy. If the child nodes were leaves the left leaf would classify objects as negative ($N$) and the right leaf would classify objects as positive class $A$. In the right figure, 3 synthetic samples were created by SMOTE to positive class $B$. This forces the decision boundary to move and as a result the region which classifies objects as negative is larger than in the left figure.

maximum classification probability but we perform two stages of evaluation. In the first stage we sum the probabilities of all positive classes and compare the grouped probability to the probability of the negative class. If the probability of the negative class is larger we classify the object as negative and if the summed probability of positive classes is larger, we go to the second stage and classify the object as the maximum of all positive classes. When using two stage prediction, $\mathbf{x}$ in the previous example would still be classified as class $A$.

Two-stage prediction however did not fulfil it's purpose and did not improve the recalls of positive classes. This implies that the suggested scenario we described in the previous example is not the root cause of the decrease in recall with SMOTE. Figure 7.2 shows an example how SMOTE can make the region in feature space which classifies objects as negative larger during node splitting in decision trees. When considering more classes and dimensions than in the example then the situation gets even more complicated. We did not further explore this problem and did not perform any experiments to determine if situation similar to Figure 7.2 happens on our data. A simple approach how to handle this problem could be inspired by the two-stage prediction method that was mentioned earlier. The idea is that if the node contains both positive and negative objects during splitting then the positive objects would be treated as a single class when calculating the entropy. This would solve the situation with the expansion of negative region in Figure 7.2.

One of the main drawbacks of SMOTE is it's computation time. While each experiment with PU learning required around 5 hours of computation time on the hardware[1] we had available, each experiment with SMOTE in Table 7.4 required around 30 hours of computation time even if the algorithm was partially optimized.

## 7.3 DAS-RF

We tested DAS-RF that was executed after PU learning to see if it can further increase recall. The hyperparameters of the Breiman forest inside DAS-RF were that same as the hyperparameters of Breiman forests in previous

[1] Intel®Xeon®CPU E7- 4860 @ 2.27GHz, 32 cores, 252 GB RAM

experiments.

The DAS-RF parameters were also the same as in the experiments on mislabeled public datasets in Section 6.2.3. We performed 10 iterations, the hyperparameter $\alpha$ was set to 0.1, starting temperature was set either to 0.3 or 0.1 and the final temperature in the last iteration was 0.005. We set the negative ratio to 10 %, even though the most promising candidate ratio, in terms of tradeoff between precision and recall, from experiments in Section 6.2.3 was 30 %, because, after experiments with SMOTE, we wanted to test if it is possible to increase recall with DAS-RF even if it meant that precision would temporarily suffer.

The result of experiments compared to the result where only PU learning was used from Table 7.3 is available in Table 7.5.

| Method | Flows | | Users | | Ben. users | |
|---|---|---|---|---|---|---|
| | Prec. (%) | Rec. (%) | Prec. | Rec. | Prec. | Rec. |
| PU only | 96.9 | 64.4 | 92.1 | 61.4 | 93.5 | 68.2 |
| PU + DAS-RF 0.1 | 96.9 | 64.6 | 92.0 | 62.4 | 93.7 | 68.2 |
| PU + DAS-RF 0.3 | 90.5 | 68.1 | 85.7 | 64.6 | 87.5 | 69.7 |

**Table 7.5:** Results of DAS-RF experiments compared to the result of PU learning from Table 7.3 which serves as a baseline. Row 'PU + DAS-RF 0.1' represents the experiment where the starting temperature was set to 0.1 and 'PU + DAS-RF 0.3' represents the experiment where the starting temperature was set to 0.3. Reported precisions and recalls were averaged over all positive classes.

The number of objects from the negative class that were assigned positive label in the last iteration of DAS-RF was 835 when the starting temperature was set to 0.1 and 2416 when the starting temperature was set to 0.3. These are not small numbers when compared to the 1460 objects filtered by PU learning on average.

Since the standard deviation estimate of recalls is around 1 %, the slight improvement in recall in case of DAS-RF with starting temperature 0.1 is not significant.

DAS-RF with starting temperature 0.3 achieves more interesting results. The increase in recall is significant but it is also compensated by a significant drop in precision. Still, it is interesting that DAS-RF was able to increase recall, when SMOTE could not achieve it even with multiplier parameter set to 10 which is a very large value.

# Chapter 8

# Conclusion

In this thesis, we focused on various types of random forest classifiers and methods that can be used together with random forests to handle imbalanced and unlabeled data in the training dataset.

From different random forest types, we implemented Breiman forests as well as Extremely Randomized Trees and Mondrian forests and tested them in various scenarios both on well-known publicly available datasets and on our own network dataset created from proxy logs where we performed classification of individual network flows for malware.

Extremely Randomized Trees consistently outperformed both Breiman forests and Mondrian forests in experiments on public datasets when they performed better in almost all of the experiments. This is an interesting results given that Breiman forests are de facto the default choice among random forest types when performing classification. However, Breiman forests performed better in experiments on our network dataset which was the main motivation for this thesis. Additionally, we discovered that Mondrian forest model's memory requirements scale both with the dimension of vectors in training dataset and the number of classes which is a major obstacle when they are trained on bigger training datasets such as mnist or aloi.

We adapted DAS-RF, PU learning spy technique and SMOTE for use together with random forests on the imbalanced and unlabeled problem. All three methods were successful in experiments on public datasets when they were able to significantly and consistently outperform random forests that were used alone as a baseline. It is not clear which one performed the best because the results vary across different datasets. However, PU learning has the advantage that it behaves more transparently than the other two methods and it is the fastest. SMOTE has the potential to have the strongest effect when there is great imbalance in the data, because it is an oversampling method that was originally developed to handle imbalanced data. Theoretically all of the methods can be combined together.

On the network dataset, PU learning spy technique consistently significantly improved performance. DAS-RF combined with PU learning was able to achieve a significant increase in recall on positive classes at the cost of decrease in precision. However, SMOTE applied in combination with PU learning caused a slight drop in recall of positive classes instead of increasing it.

# Bibliography


[1] David J Aldous. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIII—1983*, pages 1–198. Springer Berlin Heidelberg, 1985.

[2] Yali Amit, Donald Geman, and Kenneth Wilder. Joint induction of shape features and tree classifiers. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(11):1300–1305, November 1997.

[3] Jamie Shotton Antonio Criminisi, Ender Konukoglu. Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning. Technical report, October 2011.

[4] Karel Bartos and Michal Sofka. *Robust Representation for Domain Adaptation in Network Security*, pages 116–132. Springer International Publishing, Cham, 2015.

[5] Andrew Blake and Andrew Zisserman. *Visual Reconstruction*. MIT Press, Cambridge, MA, USA, 1987.

[6] L. Breiman. Bagging predictors. Technical Report 421, University of California Berkeley, 1994.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, 1984.

[8] Leo Breiman. Arcing the edge. Technical report, 1997.

[9] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[10] Chumphol Bunkhumpornpat, Krung Sinapiromsaran, and Chidchanok Lursinsap. Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 475–482. Springer, 2009.

[11] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[12] Chao Chen, Andy Liaw, and Leo Breiman. Using Random Forest to Learn Imbalanced Data. Technical report, Department of Statistics, University of Berkeley, 2004.

[13] Y. Le Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. Advances in neural information processing systems 2. chapter Handwritten Digit Recognition with a Back-propagation Network, pages 396–404. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[14] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM.

[15] Kun Deng, Chris Bourke, Stephen Scott, and N. V. Vinodchandran. New algorithms for optimizing multi-class classifiers via roc surfaces. In *In Proceedings of the ICML 2006 Workshop on ROC Analysis in Machine Learning*, pages 17–24, 2006.

[16] Thomas G. Dietterich. Ensemble methods in machine learning. In *MULTIPLE CLASSIFIER SYSTEMS, LBCS-1857*, pages 1–15. Springer, 2000.

[17] Pedro Domingos. The role of occam's razor in knowledge discovery. *Data mining and knowledge discovery*, 3(4):409–425, 1999.

[18] Pedro Domingos. A unified bias-variance decomposition and its applications. In *IN PROC. 17TH INTERNATIONAL CONF. ON MACHINE LEARNING*, pages 231–238. Morgan Kaufmann, 2000.

[19] B. Efron. Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26, 01 1979.

[20] Charles Elkan and Keith Noto. Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 213–220, New York, NY, USA, 2008. ACM.

[21] Greg Farnham and Kees Leune. Tools and standards for cyber threat intelligence projects. *SANS Institute InfoSec Reading Room*, 27, 2013.

[22] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[23] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232, 10 2001.

[24] Jerome H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, February 2002.

[25] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, April 2006.

[26] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. Borderline-smote: a new over-sampling method in imbalanced data sets learning. In *International Conference on Intelligent Computing*, pages 878–887. Springer, 2005.

[27] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition.* Springer Series in Statistics. Springer New York, 2009.

[28] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1322–1328. IEEE, 2008.

[29] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[30] Balaji Lakshminarayanan, Daniel M. Roy, and Yee W. Teh. Mondrian Forests: Efficient Online Random Forests, February 2015.

[31] Wee Sun Lee and Bing Liu. Learning with positive and unlabeled examples using weighted logistic regression. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML*, page 2003, 2003.

[32] Christian Leistner, Amir Saffari, Jakob Santner, and Horst Bischof. Semi-supervised random forests. In *2009 IEEE 12th International Conference on Computer Vision*, pages 506–513. IEEE, 2009.

[33] Letter recognition data set. `https://archive.ics.uci.edu/ml/datasets/Letter+Recognition`. Accessed: 2016-12-07.

[34] Xiao-Li Li and Bing Liu. *Learning from Positive and Unlabeled Examples with Different Data Distributions*, pages 218–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[35] Xiao-Li Li, Bing Liu, and See-Kiong Ng. Negative training data can be harmful to text classification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 218–228, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[36] Libsvm data: Classification (multi-class). `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html`. Accessed: 2016-12-07.

[37] Bing Liu, Yang Dai, Xiaoli Li, Wee Sun Lee, and Philip S. Yu. Building text classifiers using positive and unlabeled examples. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 179–, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Bing Liu, Wee Sun Lee, Philip S Yu, and Xiaoli Li. Partially supervised classification of text documents. In *ICML*, volume 2, pages 387–394. Citeseer, 2002.

[39] Xiao Liu, Mingli Song, Dacheng Tao, Zicheng Liu, Luming Zhang, Chun Chen, and Jiajun Bu. Semi-supervised node splitting for random forest construction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 492–499, 2013.

[40] Victoria López, Alberto Fernández, Salvador García, Vasile Palade, and Francisco Herrera. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250:113–141, 2013.

[41] Gilles Louppe. *Understanding Random Forests: From Theory to Practice*. PhD thesis, University of Liege, Belgium, 10 2014. arXiv:1407.7502.

[42] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[43] David Martin Ward Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *International Journal of Machine Learning Technology*, 2(1):37–63, 2011.

[44] J. R. Quinlan. Induction of decision trees. *MACH. LEARN*, 1:81–106, 1986.

[45] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[46] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[47] Kenneth Rose. Deterministic annealing for clustering, compression, classification, regression, and related optimization problems. In *Proceedings of the IEEE*, pages 2210–2239, 1998.

[48] Daniel M. Roy and Yee Whye Teh. The mondrian process. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, pages 1377–1384, USA, 2008. Curran Associates Inc.

[49] Statlog (landsat satellite) data set. `https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite)`. Accessed: 2016-12-07.

[50] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[51] David H. Wolpert and William G. Macready. An efficient method to estimate bagging's generalization error. *Machine Learning*, 35(1):41–55, 1999.

[52] Makoto Yasuda. Deterministic annealing approach to fuzzy c-means clustering based on entropy maximization. *Adv. Fuzzy Sys.*, 2011:9:9–9:9, January 2011.

[53] Jan Zahálka and Filip Železný. An experimental test of occam's razor in classification. *Machine Learning*, 82(3):475–481, 2011.

[54] Filip Železný. Machine learning and data analysis learning theory: Introduction. University Lecture, Czech Technical University in Prague, 2012.

[55] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st edition, 2012.

# Appendix A

## Enclosed CD contents

The root directory on the enclosed CD contains the following items:

- **thesis.pdf:** The PDF file of this thesis.
- **[source]:** Directory containing source codes written in Java. The project can be built with Gradle.