Anomaly Detection in Streaming Time Series Data Using Active Learning and Metalearning

Jonas Lundgren

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6101
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2020 by Jonas Lundgren. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2020

Abstract

In this thesis a framework for finding anomalies in streaming data is proposed. The framework proposed is not necessarily applicable only to problems in anomaly detection, but could be applied to other problems as well. There are three main concepts at play in the framework: (i) *Active Learning*, a learning algorithm which can query a human specialist for labels of instance such that the model can improve, from an otherwise unlabeled data set. (ii) *Ensemble* which is a combination of models, often weaker models, where the idea is that the combined result from all models will mitigate the error in every single model and thus provide better results. (iii) *Metalearning* which is the concept of having a second model learn model characteristics for a problem. In this thesis metalearning will be used to weight ensemble members.

The framework is displayed in Figure 0.1. The meta learner takes instances as input and output weights for each ensemble member according to its performance of previous similar instances. Thus the total output is a dynamically weighted ensemble output where the weighting is based on the input. When a human expert provides label feedback on misclassified instances only the meta learner is updated in order to provide new weights for the ensemble to suppress the error and not the entire ensemble.

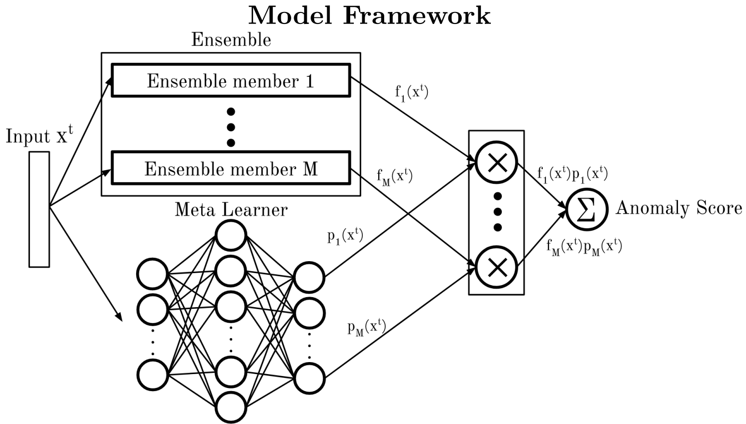


Figure 0.1: The active anomaly detection framework used.

We want to leverage the fact that different ensemble members have different characteristics which makes them more or less suitable to make predictions for certain instances. We weight the ensemble members using a neural network, taking the instance as input to weight the ensemble members in accordance with their capacity to make a prediction for certain instances. The loss to train the neural network is composed of two parts, the first a supervised part $loss_{AAD}$, using the labels provided by a human expert, and a second part $loss_{prior}$ which places a uniform prior on the ensemble members. When new labels are provided the meta learner is updated so as not to misclassify any of the labeled instances.

The framework was tested on the Yahoo Webscope benchmark dataset consisting of four different types of time series. The proposed framework had an AUC of 0.9088, 0.9787, 0.8998 and 0.8123 for the four datasets corresponding to the second highest AUC for 2 data sets and third highest for the remaining 2 data sets out of the models that were used for comparison.

Acknowledgements

This master's thesis was written as a part of the Master of Science in Engineering Mathematics degree at the Faculty of Engineering at Lund University. It was written during the spring of 2020 and was done in collaboration with Sentian.ai in Malmö.

I would like to thank Sentian for giving me the opportunity to let me write about something that interests me. I would like to thank Kenneth Ulrich for guidance. I would like to thank Ponuts Giselsson for his time and feedback.

Contents

1. Introduction **9**
1.1 Problem Description 10
1.2 Technologies 10
2. Theoretical Background **11**
2.1 Machine Learning 11
2.2 Anomaly Detection 12
2.3 Learning 13
2.4 Structures 19
2.5 Performance Measures 31
3. Methodology **35**
3.1 Data Preprocessing 35
3.2 Framework 37
3.3 The Ensemble 40
3.4 The Meta Model 45
3.5 Setup 48
4. Results **49**
4.1 Yahoo Webscope Dataset 49
5. Discussion **53**
5.1 Conclusion 55
Bibliography **56**

1

Introduction

This thesis aims to investigate how active learning and metalearning can be applied to anomaly detection in a streaming setting. There are numerous application areas in which the thesis could be applied, including fraud prevention, medical diagnostics, prevention of machine failure, security.

The problem of anomaly detection is a classification problem where we would like to separate normal (*nominals*) and unusual (*anomalous*) instances. Anomaly detection itself poses some challenges not present in a usual learning problem. (i) Since unusual instances are, by definition, unusual, we will have a skewed distribution in the target distribution and notably fewer anomalous than nominal instances. (ii) There is no hard decision boundary separating the anomalies from nominal instances.

The framework proposed in the thesis requires a human expert to be able to provide true labels to the model. This could be seen as a limiting factor since a truly unsupervised approach would not require a human. However there are instances where a human will be present regardless, and possess expertise that can be used to improve the model compared to an unsupervised model. One such scenario would be in self driving cars where the driver is the human expert and driver disengagement from the autopilot could be considered as feedback to the model. Another example could be in detecting fraudulent transactions, in which a human expert reviews only the most interesting transactions[1].

Most algorithms used for anomaly detection are designed neither to allow for human intervention nor to deal with data in a streaming setting[Das et al., 2019]. Using an active approach where human feedback is allowed to improve the performance of such algorithms is a way to continuously improve the model. In real world applications we usually don't want the model to be static and data is often sampled continuously and does not come in batches. Streaming time series data is steadily increasing across industries [Subutai

[1] `https://youtu.be/6OKJz1BVTyU?t=762`

and Scott, 2016] and it has been observed that research in the area is lacking[2]. We would therefore like the model to be able to deal with data in a streaming setting and to continuously improve over time.

1.1 Problem Description

This thesis aims to address the following questions:

1. **Can we create a framework which can be used to find anomalies in and adjust to streaming time series data?**
2. **Can we update the framework such that it can improve by human intervention?**
3. **How well does our framework compare to other anomaly detection systems for streaming data?**

1.2 Technologies

All programming done in this thesis was done in the programming language *Python*. This is the programming language used at Sentian. Numerous Python packages were used throughout the thesis. Modeling of neural networks was done using *PyTorch*. For most other models used in the thesis *scikit-learn* was used. *Pandas* was used for data handling and *numpy* for other numeric computation to name a few.

All coding and testing was done on a Lenvo ThinkPad T470 with an Intel i5-7200U CPU provided by Sentian.

[2] `https://youtu.be/0VH1Lim8gL8?t=4219`

2

Theoretical Background

The theoretical background needed in this thesis can be divided into two parts *learning* and *structures*. The structures provide the framework in which the learning can occur, and learning in turn provides a way for the structure to improve. In the machine learning world the two go hand in hand. Before looking at the learning techniques and structures used in the thesis, we first give a background of *machine learning* and the main application in thesis: *anomaly detection*.

2.1 Machine Learning

What is machine learning? There are numerous definitions and explanations of what machine learning is in books and online. Most explanations of machine learning are some kind of variant of the definition made by Tom Mitchell in his book *Machine Learning* [Mitchell, 1997]:

> ***Definition***: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

From my point of view the key in the definition is that the program is said to learn, hence does not need to be explicitly programmed. However, I feel that these definitions exclude a lot of what I consider to be machine learning. The definition only accounts for a small part of what it to me means to work in machine learning, namely the modeling part. I would therefore rather define machine learning as a framework in which we are moving from (i) *fundamentals* to (ii) *extraction* to (iii) *applications*. Where (i) the fundamentals include probability theory, statistics, mathematics and computer science. Further (ii) extraction is how to, from observations in the world, with the use of the fundamentals being able to know what to do with the observation. And finally with the use of analysis of causality, uncertainty,

modeling, optimization, simulation, to name some ways, we want to create (iii) applications in healthcare, energy, safety, technology, advancing science and climate to name a few areas.

The American astronomer Clifford Stoll has been quoted saying:

> Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom.[1]

Human biases are not present when a program is not explicitly programmed, therefore one could make an argument of it being wise. To me machine learning is a problem solving framework which lets us go from data to wisdom.

2.2 Anomaly Detection

Anomaly Detection (AD) refers to the problem of detecting unusual, deviating and/or interesting instances among a set of normal instances. The term *anomaly detection* is often referred to as either *rare event detection*, *outlier detection* or *novelty detection*. The line between what is what is blurry. An attempt to clarify how the terms have been used in the literature was done by Carreño, Inza and Lozano [Carreño et al., 2019]; the result is presented in Table 2.1. Even though this thesis will mainly focus on time series data which should be refered to rare event detection according to Table 2.1, the problem of detecting unusual, deviating and/or interesting instances will be referred to as *anomaly detection* in this thesis.

Relative to	Characteristics	Rare Events	Anomaly	Novelty	Outlier
Data	Temporal data	Yes	No	No	Possible
Data	All classes represented in training data	Yes	Yes	No	-
Problem	Unbalanced Classification	Yes	Yes	Possible	-
Problem	Supervised Classification	Yes	Yes	Yes	No

Table 2.1: Summary of how the terms Rare Events, Anomaly, Novelty and Outlier have been used in the literature according to [Carreño et al., 2019] in relation to characteristics for either the type of data and/or problem.

The problem of anomaly detection can look rather different depending on the application, the data available and the remedy put in place when an anomaly is found. However most algorithms for anomaly detection try to model some probability density function of normal instances $f_X(x)$, where the set of instances X often contains both the anomalous and normal instances. The model labels instances as anomalous or normal based on the likelihood that these instances belong to the modeled distribution. Therefore

[1] `https://www.brainyquote.com/quotes/clifford_stoll_212166`

anomaly detection models tend to present outliers in the modeled distribution as anomalies. The problem is thus reduced to measuring distances from an expected value to the instance [Wang et al., 2019]. This type of approach has shortcomings, as it, for cases where anomalies occur within the distribution and for cases where normal instances happen to lie far from the center of the distribution and thus result in an unnecessarily high amount of false categorisations [Das et al., 2019]. Working with high dimensional data exposes another shortcoming; distance measures loses their usefulness, one of several components of the *curse of dimensionality*[Wikipedia, 2020c].

Anomaly detection is essentially "finding a needle in a haystack" in data... but you don't know it's a needle you're looking for.

2.3 Learning

Learning is what makes machine learning algorithms able to solve problems that would be too difficult to solve with a rule based program. We often want the learning to occur with as little human guidance as possible, partly because we do not want to infer our human biases of how a problem should be solved and partly because we in many times don't know how to guide the learning algorithm ourselves. We as humans can for example distinguish between a cat and a dog in a picture, but to express what makes a cat a cat and a dog a dog from pixel values in a computer program is a hard task [Goodfellow et al., 2016].

Learning itself is not the ability to solve a problem, but rather the way to obtain the capacity to solve the problem. If we want a computer program to distinguish between cats and dogs, to discriminate between the two is the problem. We could try to create a program based on typical cat and dog features, or we could could create a program that learns to distinguish between the two. The learning problem is most of the time distilled to some optimization problem, where we optimize the model performance for the problem.

Problems in machine learning often stem from observations in the real world. The idea is to teach the learning algorithm by showing it examples of previous observations of the same type it will face in the future. In the case of distinguishing between cats and dogs, an image of a cat or a dog would correspond to such an observation. The observations are typically represented as a vector $\mathbf{x} \in \mathbb{R}^n$ where the ith entry in the vector x_i corresponds to a feature, for example the pixel value for pixel i in an image [Goodfellow et al., 2016].

There are mainly three types of learning used in the context of machine learning, *supervised learning*, *unsupervised learning* and *reinforcement learning* [Géron, 2019]. Reinforcement learning deals with problems where agents

take actions in some environment [Wikipedia, 2020d]. It will not be used in this thesis. We are instead going to look into other types of learning techniques, *active learning* and *metalearning*.

Supervised Learning

For supervised learning we have labels available when training for all observations. In the dogs and cats example this would correspond to our having a correct label available for each image. An image of a dog $\mathbf{x}$ would have a corresponding label y indicating if the image is representing a dog or cat. The task of anomaly detection is also a classification task since we try to classify as anomaly or normal.

Typically there are two types of tasks withing supervised learning, *classification* and *regression*. Distinguishing between cats and dogs would be a classification task. Regression is the task of predicting some numeric value, such as price of a house. Training a model would require examples of houses with different features $\mathbf{x}$ and the price y [Géron, 2019].

The use of labels and supervised learning is rarely used for anomaly detection. Since the goal when using supervised learning is to discriminate between prelabeled classes, the model is likely to perform poorly on previously unseen anomaly classes [Goernitz et al., 2014]. Anomaly detection algorithms also typically have to be adaptable and continue learning as there often are drifts and/or changes in streaming data, illustrated in Figure 2.1. If a model was trained on data up to 2014-02-25 and then deployed it could potentially flag all instances as anomalous. Therefore it is essential for the model to be able to adapt to the change in the data stream.

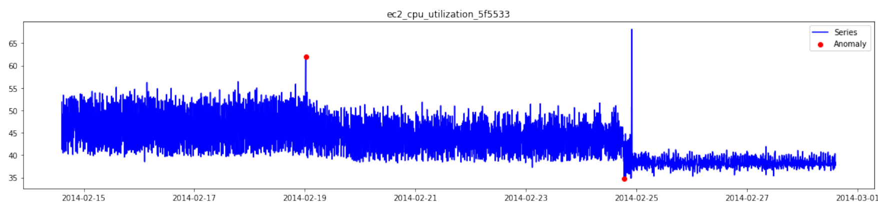


Figure 2.1: The CPU utilization for an Amazon EC2 instance. At about 2014-02-25 a change in the streaming CPU utilization is present. It is essential for a model to be able to adapt to the change in the data stream at 2014-02-25.

Unsupervised Learning

For unsupervised learning we do not have any labels available during training. In the dogs and cats example this would correspond to our having only the images $\mathbf{x}$, with no labels y indicating if the image is representing a dog or

cat. We would have to try to group the images based on similarity, using some *clustering algorithm*. Even if this algorithm succeeded in separating the images into two clusters, one with dog images and one with cat images, it would not be able to tell us which is which, but rather that the images in one cluster are more similar to each other than images in the other cluster [Géron, 2019].

For the task of anomaly detection, unsupervised learning is more common than supervised learning [Goernitz et al., 2014]. In an unsupervised setting, models are trained for data characterization and there is a difference when models are trained using both normal and anomalous instances during training or only normal instances. The difference is that an algorithm trained on only normal instance only expects to encounter normal instances and therefore essentially tries to model the distribution of normal instances $f_{normal}(x)$. Everything outside $f_{normal}(x)$ is thus anomalous. While if the model is trained on both normal and anomalous instances the algorithm tries to model the combined distribution of normal and anomalous instances $f_{normal \cup anomalous}(x)$. Classical methods used are thresholds, clustering and exponential smoothing. Anomalies are then determined based on some distance measure [Ahmad and Purdy, 2016]. The performance of these methods are often sensitive to window sizes for time series and what thresholds to use.

There are two main problems using unsupervised learning in anomaly detection. The first is that results will be sensitive to what threshold to use, illustrated in Figure 2.2. Most models are soft in their classification, meaning that the model assigns an anomaly score to each instance. Categorization as normal or anomalous is then determined by some cut-off in the anomaly score, the threshold. Instances are essentially categorized as anomalous based on some distance from the mean of the distribution. The threshold will directly influence the number of type 1 (false positive) and type 2 (false negative) errors. Sometimes we prefer one error over the other and could adjust the threshold accordingly. For example when determining if someone is sick and the available cure has no side effects we might have a lower threshold while if the cure instead had severe side effects we might want a higher threshold. The other problem of using unsupervised learning for anomaly detection is that we assume that there is a difference in the distributions $f_{normal}(x)$ and $f_{anomalous}(x)$ and that we can separate the two. In real world cases this is often not the case. Figure 2.2 displays the problem when $f_{normal}(x)$ and $f_{anomalous}(x)$ are overlapping and the same unsupervised learning algorithm (*isolation forest*) classifies instances as normal or anomalous based on two different thresholds. It should be mentioned that since we are in an unsupervised setting, the model, isolation forest, does not have access to the labels. Imagine trying to predict the anomalous instances in the figure to the left, if all dots were displayed to you in white. Adjusting the threshold for the unsupervised algorithm will not result in a sufficiently good model.

Unsupervised Learning for Toy Data

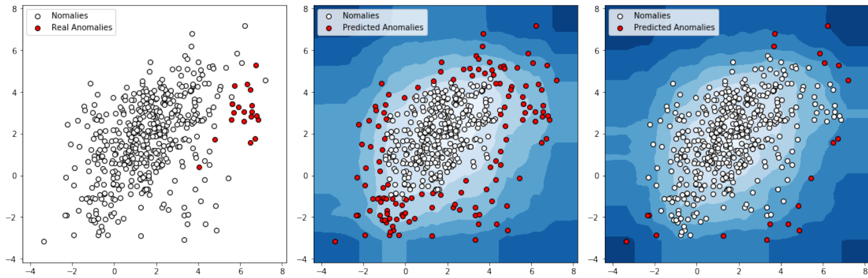


Figure 2.2: The figure to the left displays a toy dataset where the red dots indicate the real anomalies. The middle and right figures display the result of how the same unsupervised learning algorithm would classify the instances as normal or anomalous based on two different thresholds. The anomaly score is indicated by the blue scale: darker blue indicates higher anomaly score than lighter blue in the middle and right figures. Inspiration for plotting code [2].

Active Learning

At the 2019 International Conference on Machine Learning Yoshua Bengio, recipient of the Turing Award 2018, Professor at University of Montreal said in an interview [Synced, 2019]:

> "I see a move from passive machine learning, where the learner gets a big dataset and trains; to **active machine learning**, where the learner interacts with its environment. It's not just reinforcement learning. It is **active learning**, things like dialogue systems where the interaction allows the learner to improve and to seek information."

Active learning is not considered as one of the typical types of learning in machine learning. The idea is to design the model such that it can query a human expert for labels of some instances to improve. The process is displayed in Figure 2.3 where the model queries a human expert (step 1) who provides labels for the requested instances (step 2) and the model updates according to the label feedback (step 3) [Wikipedia, 2020a].

[2] https://scikit-learn.org/stable/auto_examples/ensemble/plot_isolation_forest.html#sphx-glr-auto-examples-ensemble-plot-isolation-forest-py

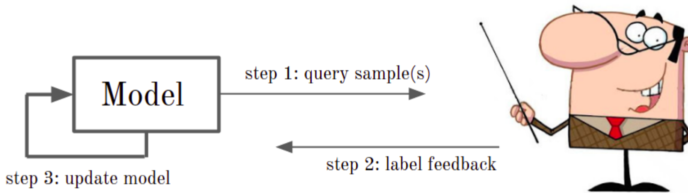


Figure 2.3: **Step 1**: the model queries a human expert. **Step 2**: the human expert provides labels for the requested instances. **Step 3**: the model updates according to the label feedback.

The process now has two goals: to minimize the number of queries and to obtain a sufficiently accurate model [Das et al., 2019].

Metalearning

Metalearning has been around since the mid-1970s but started to appear in the machine learning community in the 1990s[Lemke et al., 2013]. To distinguish between a learning system in general and a metalearning system can be somewhat tricky. One definition made by [Schaul and Schmidhuber, 2010] states:

> Metalearning is the process of learning to learn. Informally speaking, a metalearning algorithm uses experience to change certain aspects of a learning algorithm, or the learning method itself, such that the modified learner is better than the original learner at learning from additional experience.

Another definition made by [Lemke et al., 2013] states:

1. A metalearning system must include a learning subsystem, which adapts with experience.
2. Experience is gained by exploiting metaknowledge extracted
    a) ...in a previous learning episode on a single dataset, and/or
    b) ...from different domains or problems.

The idea is to use a framework consisting of two systems: an unsupervised learning anomaly detection system and a metalearning system, which adapts with experience and affects the unsupervised learning system. The second system is the one mentioned in the first statement in the above definition.

The first learning system will consist of several models i.e. an *ensemble*. The role of the second metalearning system is to weight the several models in the first system, depending on the input. The systems thus adapt the

weighting of the models in the first system based on the input. The second, metalearning system will in the thesis consist of a neural network, taking an instance as input to weight the models in the first system, in accordance with their capacity to make a prediction for certain instances.

Ensemble learning

Combing several models to make a prediction is called *ensemble learning*, the models used in an ensemble are often simpler models. The idea is that the different models in the ensemble should be different to each other. The bias introduced by any single model is reduced by averaging over the large number of ensemble members. We often want to introduce more variance on purpose for every single model in the ensemble. One way to introduce more variance among the models is to train each model with a randomly sampled subset of the data. This is used in for example the *Random Forest* algorithm which uses committee of trees to make a final prediction. Another tree based ensemble algorithm which will be used in this thesis is the *Isolation Forest* which is more suited for the problem of anomaly detection [Hastie et al., 2001].

There are two benefits in using ensembles in combination with active learning in a streaming seating. The first benefit is that using an active learning approach, we gain control over the ensemble members which we otherwise would lack control over. The common approach in ensemble learning is to weight all ensemble members equally. Using metalearning, the second system allows us to weight the ensemble members based on their performance, thus gaining control over them. The ensemble itself is based on an unsupervised algorithm that would not allow the incorporation of label feedback, which could improve performance of the algorithm. The improved performance is displayed in Figure 2.4 where the same ensemble members are used to categorize anomalies in the middle and right figures. The difference is that all ensemble members are equally weighted in the middle figure while a feed forward neural network is used as a second metalearning system to weight the ensemble members based on the instance in the right figure.

Active Learning for Toy Data

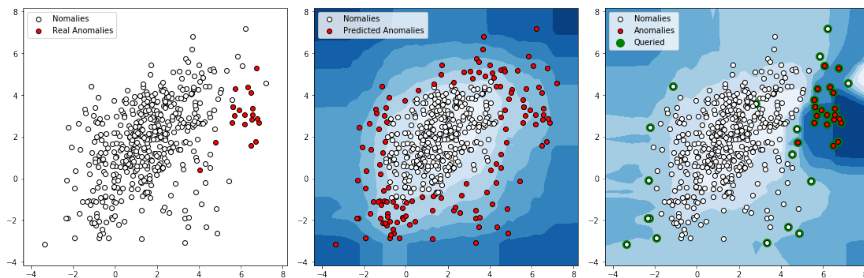


Figure 2.4: The red dots in the figure to the left indicates the real anomalies. The anomaly score is indicated by the blue scale, darker blue indicates higher anomaly score than lighter blue. The middle and right figure uses the same ensemble members. While the model in the middle figure weights all ensemble members equally the model in the left figure uses a feed forward neural network to weight the ensemble members. Inspiration for plotting code [3].

The second benefit is associated with streaming data, which potentially is unlimited and fed to the model in windows. Using an ensemble we can use the first window of data to train all ensemble members. When new data is provided to the model, we can update subset of the models used in the ensemble. Consequently, we remove the oldest fraction of the members in the ensemble and train the same number new members on the new data. Therefore we have a way to deal with drifts or shifts in streaming data, such as the data displayed in Figure 2.1 [Das et al., 2019].

2.4 Structures

In order to create a structure in which learning can occur we have to make some assumptions. (i) We have some vector $\mathbf{x}$ with p parameters. (ii) We have some variable y. (iii) There exists a function f returning y given $\mathbf{x}$. (iii) The function f is optimal when satisfying minimizing some loss function L. The loss function L will return large values if the output from f given $\mathbf{x}$ is far from the true y and small if the output is close to y. The goal is to find the optimal f i.e. the structure. The structure consists either of parameters, such as *linear regression*, that can be tuned in order to change an output $\hat{y}$ from a structure for some input $\mathbf{x}$ or some algorithm that applied to an input $\mathbf{x}$ produces an output $\hat{y}$, such as an *isolation forest*. These algorithms can vary

[3] https://scikit-learn.org/stable/auto_examples/ensemble/plot_isolation_forest.html#sphx-glr-auto-examples-ensemble-plot-isolation-forest-py

in complexity, giving rise to large research fields such as *deep learning*[4] while others, such as decision trees, can be explained in a short YouTube clip[5]. What they all have in common is that they try to model data and usually generalize better with more data.

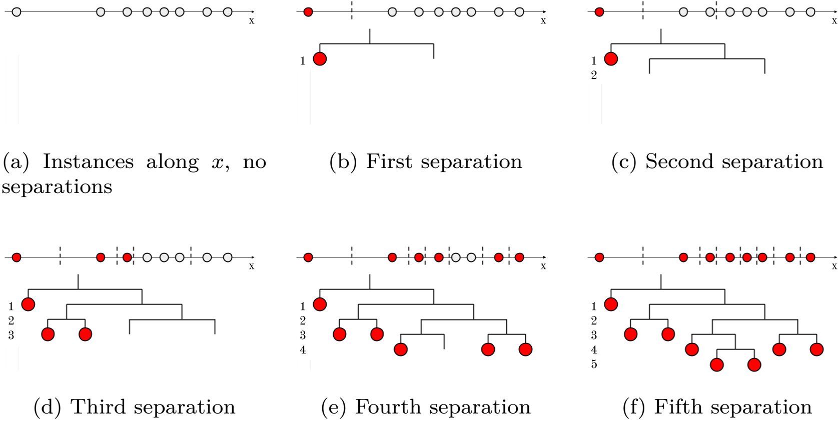
Isolation Forest

The Isolation forest algorithm, first proposed in [Liu et al., 2008] is a tree algorithm designed for anomaly detection. The assumption the algorithm makes is that anomalies lie further away, distance wise, from other observations. The idea is that if we separate all instances less than and larger than some random point into two groups, we are likely to make the separation such that the anomaly is on one side of the random point and the remaining normal instances on the other side of the point. In extension, we continue to make random separations until all instances have been fully separated. The random separations are made on each interval where instances have not yet been isolated. The partitioning can be described by tree structure and is displayed for 8 instances along some axis x in Figure 2.5. When the final tree is constructed, the tree in Figure 2.5 (f), the depth from the root node to each isolated instance is measured. An anomaly score is assigned to each sample based on the depth, where a higher anomaly score corresponds to a more shallow depth. When the first tree is completed new trees with different random splits are created.

[4] `https://scholar.google.com/scholar?hl\=en\&as_sdt=0\%2C5\&q=deep+learning\&btnG=`

[5] `https://www.youtube.com/watch?v=7VeUPuFGJHk`

Isolation Forest Partitioning of Instances with Corresponding Trees



(a) Instances along x, no separations

(b) First separation

(c) Second separation

(d) Third separation

(e) Fourth separation

(f) Fifth separation

Figure 2.5: The splitting process for the Isolation Forest algorithm for 8 instances displayed in (a) along axis x. The first random separation displayed in (b) isolates the instance furthest to the left. The second separation displayed in (c) isolated no additional instances. The third separation displayed in (d) isolated 2 additional instances, the fourth separation, displayed in (e), isolated three additional instances and the fifth separation, displayed in (f), isolated the remaining two instances. In total five separations were required to isolate all instances. The tree corresponding to each random partitioning is displayed for each seperation.

The depth of each observation for all trees determines the final anomaly score for each instance. Algorithm 1 displays how the anomaly score $a(x_n) = 2^{\frac{E[h(x_n)]}{E[h(x_n)|\psi]}}$ for an instance x_n is a power. The base, 2, arises from the fact that the height of an isolation tree grows in the order of $log(\psi)$ where ψ is the size of the sample size of the data. The depth $h(x_n)$ is determined by the number of edges e from the root node to the leaf node where x_n has been isolated in the isolation tree. The average $E[h(x_n)]$ is thus the average over all isolation trees. Not all observations x_n will be used in the construction of each isolation tree since we are using a subsample of the data $\mathbf{X}'$ of size ψ when constructing each isolation tree. We therefore need to normalize the depth of each instance $h(x_n)$ which is the reason we divide the exponent with $E[h(x_n)|\psi]$ when calculating the anomaly score [Liu et al., 2008].

Algorithm 1: Isolation Forest

Inputs: $\mathbf{X}$ - input data, N - number of trees, ψ - subsample size
Outputs: Anomaly score for each instance
for *i = 1 to N* **do**
  $\mathbf{X}' = sample(\mathbf{X}, \psi)$
  Initialize isolation tree
  Choose random feature j in $\mathbf{X}'$
  while *Not all instances are isolated* **do**
    Randomly pick threshold $\theta \in [min(x_j), max(x_j)]$ uniformly
    Recursively split $\mathbf{X}'$ on θ
  end
end
Let $E[h(x)]$ be the average depth over all isolation trees for $x \in \mathbf{X}$
Let $E[h(x)|\psi]$ be the average depth over all isolation trees for $x \in \mathbf{X}$ given ψ
Return: Anomaly score of each instance $a(x) = 2^{\frac{E[h(x)]}{E[h(x)|\psi]}}$

In Algorithm 1 a function *sampling* was used in order to resample the input data $\mathbf{X}$ to some new data set $\mathbf{X}'$. This result in each isolation tree being trained on different data, resulting in a variation among the trees. The sampling method used is *bootstrap*, which means that we are going to sample with replacement from $\mathbf{X}$. The probability that an instance from the dataset $\mathbf{X}$ of size n is in $\mathbf{X}'$ is given accordingly:

$$\text{P}(\text{"included in } \mathbf{X}'\text{"}) = 1 - \text{P}(\text{"excluded in } \mathbf{X}'\text{"})^n = 1 - (\frac{n-1}{n})^n$$

As n is large we get:

$$\lim_{n \to \inf} 1 - (\frac{n-1}{n})^n = 1 - \frac{1}{e} \approx 0.63$$

Each isolation tree is therefore constructed of roughly two thirds of the observations where half of the observations are duplicates.

Linear Regression

Linear Regression is a useful statistical approach for solving supervised problems. The goal of linear regression is to investigate relationships in data, and especially the relationship between instances $\mathbf{x_i}$ and their corresponding target y_i. In the example from section **2.3 Supervised Learning**, it was stated that one task in the domain of supervised learning could be to predict a numeric value, such as the `price` of a house y based on some house features $\mathbf{x}$ e.g. `number_of_rooms`, `indoor_area` and `plot_area`. Questions we are interested in when looking for a house could then be: *what is the relationship between* `number_of_rooms` *and the* `price`? *Between* `indoor_area` *and*

price? *Between* plot_area *and* price? The idea with linear regression is to estimate these relationships. If we assume that the relationships are linear we could write this as:

$$\texttt{price} \approx \beta_0 + \beta_1 \texttt{number_of_rooms} + \beta_2 \texttt{indoor_area} + \beta_3 \texttt{plot_area}$$

We would now like to make estimations of the coefficients $\beta_0, \beta_1, \beta_2, \beta_3$ in order to figure out the price. These estimations are made with the use of data from previous house sales. We could arrange the housing features of the 10 most recent house sales in the same neighbourhood in a matrix $\mathbf{X}$ of size 10×4 where each house corresponds to a row and the first column corresponds to the number_of_rooms for each house, the second the indoor_area and the third the plot_area as well as their corresponding prices in a vector $\mathbf{y}$. The problem can thus be written in the form:

$$\hat{\mathbf{Y}} = \mathbf{X}\hat{\beta}$$

Where $\hat{\beta} = [\hat{\beta}_0\, \hat{\beta}_1\, \hat{\beta}_2\, \hat{\beta}_3]^\top$ are our estimations of the coefficients $\beta_0, \beta_1, \beta_2, \beta_3$ and $\hat{\mathbf{Y}}$ our estimation of the 10 house prices. The vector $\hat{\beta}$ is estimated by minimizing the sum of squares difference between our estimation and the true values, i.e. least squares problem which is for the example above is defined as:

$$\begin{aligned}
\hat{\beta} &= \underset{\beta}{minimize} \sum_{i=1}^{10} (y_i - \beta_0 + \beta_1 \texttt{number_of_rooms}_i + \beta_2 \texttt{indoor_area}_i + \beta_3 \texttt{plot_area}_i)^2 \\
&= \underset{\beta}{minimize} \sum_{i=1}^{10} (y_i - \hat{y}_i)^2 \\
&= \underset{\beta}{minimize} \, \|\mathbf{Y} - \mathbf{X}\beta\|_2^2 \text{[James et al., 2014].} \qquad (2.1)
\end{aligned}$$

If the matrix $\mathbf{X}$ has full column rank, the matrix $\mathbf{X}^\top\mathbf{X}$ will be invertible and there exists a unique optimal β^*, with a zero gradient, which satisfies the normal equations:

$$\begin{aligned}
&F(\beta) = \|\mathbf{X}\beta - \mathbf{Y}\|_2^2 = (\mathbf{X}\beta - \mathbf{Y})^\top(\mathbf{X}\beta - \mathbf{Y}) = \beta^\top\mathbf{X}^\top\mathbf{X}\beta - 2\beta^\top\mathbf{X}^\top\mathbf{Y} + \mathbf{Y}^\top\mathbf{Y} \\
&\Rightarrow F'(\beta) = 0 = 2\mathbf{X}^\top\mathbf{X}\beta^* - 2\mathbf{X}^\top\mathbf{Y} \Leftrightarrow \mathbf{X}^\top\mathbf{X}\beta^* = \mathbf{X}^\top\mathbf{Y} \Leftrightarrow \beta^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}.
\end{aligned}$$

However, the normal equations are usually not used when solving the least squares problem, since we have to invert $\mathbf{X}^\top\mathbf{X}$ which is a costly operation [Trefethen and Bau, 1997]. In this thesis the least squares problem was solved using the function LinearRegression from the Python package sklearn. The function is based on the software package LAPACK and the algoritm *xGELSD* which uses the singular value decomposition and an

algorithm based on divide and conquer to solve the least squares problem [scikit-learn, 2020][scipy, 2020][LAPACK, 2020].

It should be mentioned that Equation 2.1 makes no assumptions of the distribution of the data in $\mathbf{X}$ and $\mathbf{Y}$, but rather finds the best linear fit to the data. We would have to make assumptions about the data if we were to make inference regarding any parameters of the model. If that is the case the first assumption we have to make is that all observations are independent and identically distributed. Further there is a linear relationship between the target $\mathbf{Y}$ and the features $X_1, ..., X_p$ which are the columns of $\mathbf{X}$. And we assume normality in the *residual* i.e. the distance for the model space to the observations. The regression is fitted such that the distributions of the residuals is Gaussian with zero mean and constant variance:

$$Y = \beta_0 + \sum_{i=1}^{p} X_i \beta_i + \epsilon$$

Where the error $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a Gaussian random variable [Hastie et al., 2001].

Gradient Descent

Another way of finding the optimal solution for linear regression is use the algorithm *Gradient Descent*. The idea with gradient descent is to iteratively update the parameters of a model to reduce the error of the model. The error is calculated by a *cost function* or *loss function*, which has the purpose of reducing the model performance to a single metric, which we want to minimize.

The loss, i.e. the value of the loss function will be depending on the model parameters. If we let θ be a parameter in a model, the loss will change if we increase/decrease θ by some small value. We want to increase/decrease θ such that the loss decreases. Hence we want to get the gradient of the loss with respect to the parameter θ and change θ in the direction of the decreasing gradient. We want to change the parameter θ iteratively until the loss is minimized, displayed in Figure 2.6[Géron, 2019].

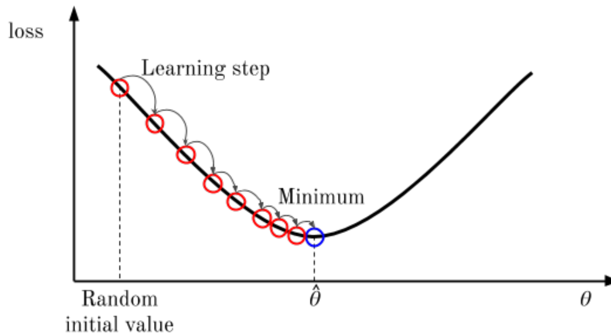


Figure 2.6: Visualisation of gradient descent where the loss is being minimized with respect to a model parameter θ. The loss in minimized at when $\theta = \hat{\theta}$.

There are three problems with gradient descent which have to be addressed. (i) In Figure 2.6 the learning steps are taken such that the gradient descent algorithm ends up at the minimum $\hat{\theta}$. We could imagine taking learning steps that are either too small, such that it takes too long for the algorithm to converge, or too large, such that the algorithm would jump back and forth over the minimum never converging. (ii) In Figure 2.6 the initial value for θ was randomized, regardless of the initial value for θ we would converge to $\hat{\theta}$. This would be the case if the loss function was a convex function with respect to θ. If the loss function contains local minima different from the global minimum gradient descent risk getting stuck in a local minimum instead of the global. In the case of linear regression, this is not a problem since the function to minimize is convex ($\underset{\beta}{minimize}\ \|\mathbf{X}\beta - \mathbf{Y}\|_2^2$). There are multiple ways to tackle these problems for the gradient descent algorithm, especially when working with neural networks where the loss function (almost) never is convex, one way, used in this thesis to train neural networks is an algorithm called *Adam*[Géron, 2019]. (iii) For large amounts of data, computing the gradient can be time consuming. The calculations can be sped up using *stochastic gradient descent*[Goodfellow et al., 2016].

Stochastic Gradient Descent Calculating the gradient with respect to an entire data set can be computationally expensive. A method for reducing the computation time is to use *stochastic gradient descent* where the gradient is estimated using a randomly sampled subset of the dataset. This means that each gradient step will not be taken aimed straight towards the minima. However each gradient step can be calculated faster. There is therefore a

trade off between faster interactions and convergence rate. Especially when dealing with large amounts of data, we would typically favour the faster iterations over faster convergence rate [Goodfellow et al., 2016].

Adaptive Moment Estimation (Adam) If normal gradient descent is like a frog jumping down a valley, getting stuck in the first pond it reaches (local minima), then Adaptive Moment Estimation (Adam) is more like a large ball rolling down the valley through the pond, continuing past the bottom of the valley, going back and forth until finally stoping at the bottom (or that's the idea at least). Adam [Kingma and Ba, 2015] does not have a static learning rate, but instead makes use of an exponentially decaying average of previous gradients m_t and the exponentially decaying average of previous squared gradients v_t in order to have a momentum to be able to continue to take learning steps through flat regions and local minimas. These could be seen as estimates of the mean m_t and variance v_t of the gradients.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

When initializing the algorithm both m_t and v_t will be zero making them biased towards 0, especially at the start and when β_1 and β_2 are close to 1. To counteract this bias let

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

The final update of the parameter θ is determined by:

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Where η is the learning rate, ϵ some small error term. The authors of the paper that first propose the Adam algorithm [Kingma and Ba, 2015] suggested the default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. The learning rate η is specific to the problem and will have to be determined by the user of the algorithm.

Cosine Annealing Another way to improve the convergence rate especially for neural networks is to use schedule schemes for the learning rate η such that η becomes a function of t where t is the batch index i.e. η_t. One way of changing η_t is with warm restarts. The learning rate η_t is scheduled to decrease for some predetermined number of batches t. After t batches we restart the learning rate η_t and set it to its initial value. The process is then

repeated. One type of warm restart is called *cosine annealing* [Loshchilov and Hutter, 2016] given by:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + cos(\frac{t}{T_0}\pi)).$$

Where η_{min} and η_{max} are ranges for the learning rate, t the current batch index and T_0 the number of batch indices before a reset. The function is displayed in Figure 2.7.

Cosine Annealing

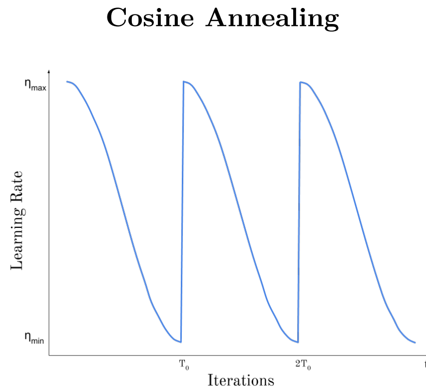


Figure 2.7: Cosine annealing where the learning rate η_t is a function of time.

The intuition for why we would want to use warm resets is that we converge to some local minima where we get stuck, then we take a large leap and start over, potentially jumping out of the local minima, then we get stuck in another and so on. Used in combination with the Adam optimizer, cosine annealing is like increasing the friction and giving legs to the ball rolling down the valley, the ball is slowing down all the time and every now and then the ball takes a large leap.

Neural Network

A neural network is a type of structure in which learning can occur. The goal of a neural network is to approximate some function f mapping an input $\mathbf{x}$ to an output y. The most elementary neural network, a single perceptron, displayed in Figure 2.8, consists of 1 node, an input vector $\mathbf{x}$ a bias b and an activation function ϕ. The output y is thus given by $y = \varphi\big(\sum_{i=1}^{N} \omega_i x_i + b\big)$. We see that without the activation function φ or φ being the identity function $(\varphi(.) = .)$, the perceptron is a linear regression [Géron, 2019].

Single Perceptron

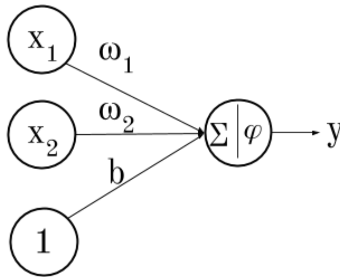


Figure 2.8: The perceptron mapping an input $\mathbf{x}$ of length 2 to an output y accordingly $y = \varphi\big(\sum_{i=1}^{N} \omega_i x_i + b\big)$.

The choice of activation function is determined by the architect of the neural network. Some typical activation functions are the *Heaviside*, *sigmoid* and *rectified linear unit* (relu).

Heaviside	*Sigmoid*	*Relu*
$\varphi(x) = \begin{cases} 0 \;\; if \; x < 0 \\ 1 \;\; if \; x \geq 0 \end{cases}$	$\varphi(x) = \dfrac{1}{1+\exp(-x)}$	$\varphi(x) = max(0,x)$

A slightly more complicated neural network is created when adding single perceptions, nodes, in a layer, where each node is connected to all inputs and each node has its own bias, as displayed in Figure 2.9. The layer is then called a fully connected layer or a *dense layer* and maps an input $\mathbf{x}$ of length n to an output $\mathbf{y}$ of length m and $\mathbf{y} = \varphi(\mathbf{x}\mathbf{W} + \mathbf{b})$. The matrix $\mathbf{W}$ is of size $n \times m$ consisting of the weights (excluding bias terms) with one row per input and one column per node in the fully connected layer. The bias vector $\mathbf{b}$ of size m contains a bias term for each of the nodes in the fully connected layer.

Fully Connected Layer

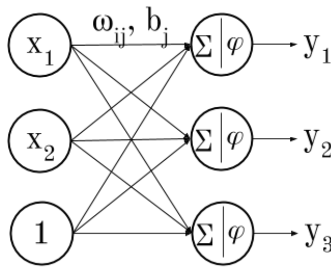


Figure 2.9: A fully connected layer mapping an input $\mathbf{x}$ of length 2 to an output $\mathbf{y}$ of length 3, accordingly $\mathbf{y} = \varphi(\mathbf{x}\mathbf{W} + \mathbf{b})$. The matrix $\mathbf{W}$ of size 2×3 consist of the weights (excluding bias terms) with one row per input and one column per node in the fully connected layer. The bias vector $\mathbf{b}$ of size 3 contain a bias term for each of the nodes in the fully connected layer.

We want to find values for the parameters, called *weights*, ω_{ij} in the matrix $\mathbf{W}$ and the bias terms b_j in $\mathbf{b}$ that create the mapping f that best fit our data. When the perceptron is given an instance $\mathbf{x}$ it returns an output $\hat{\mathbf{y}}$. If the prediction is incorrect we update the weights to improve f. The weight update is made using the *perceptron learning rule (weight update)* accordingly:

$$\omega_{ij}^{(next\ step)} = \omega_{ij} + \eta(y_j - \hat{y}_j)x_i$$

Where x_i is the value at the ith position in the input $\mathbf{x}$, the ω_{ij} is the connection between the and the jth node in the fully connected layer, $\hat{y}_j$ the output from the jth node, y_j the target for the jth node, corresponding to the current input $\mathbf{x}$ and η a learning rate, which is determined by the model architect[Géron, 2019].

Stacking several fully connected layers results in a slightly more complicated neural network structure called *Multi Layer Perceptron* (MLP) displayed in Figure 2.10. A MLP can consist of several layers between the input and output layers, these layers are called *hidden layers*.

Multi Layer Perceptron

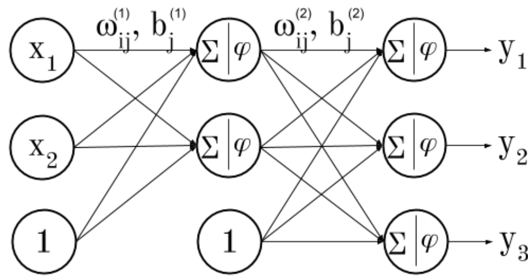


Figure 2.10: A fully connected layer mapping an input $\mathbf{x}$ of length 2 to an output $\mathbf{y}$ of length 3, accordingly $\mathbf{y} = \varphi(\varphi(\mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$. The matrix $\mathbf{W}^{(\mathbf{1})}$ of size 2×2 and $\mathbf{W}^{(\mathbf{2})}$ of size 2×3 consist of the weights (excluding bias terms) with one row per input node and one column per output node in corresponding layer. The bias vectors $\mathbf{b}^{(1)}$ of size 2 and $\mathbf{b}^{(2)}$ of size 3 contain a bias term for each of the nodes in the fully connected layer.

In order to update the weights and biases of a MLP we make use of the training algorithm called *backpropagation* [Rumelhart et al., 1986]. Backpropagation is the process of computing the gradients needed for applying Gradient Descent to a neural network. By passing an input $\mathbf{x}$ to a neural network we calculate the network output $\hat{\mathbf{y}}$, i.e. a forward pass. Based on the error or the network output we want to go backward through the network, a *backward pass*, calculating the impact of each weight and bias on the error, i.e. the gradient of error with respect to each parameter in the network. When all gradients are computed we take a gradient step (or some other gradient descent algorithm, such as Adam) and update all parameters. This process is repeated until we (in theory) converge to a solution [Géron, 2019].

The complexity of neural networks can further increase with more layers, nodes, connections, activation functions and stacking different networks. Different types of neural networks are suitable for different tasks. Some examples would be *Convolutional Neural Networks* suitable when working with images, *Recurrent Neural Networks* suitable when working with sequences or *Autoencoders* suitable when working with dimensionality reduction.

Loss function Training the weights in the neural network is done through optimizing a *loss function* and applying backpropagation. The loss function is minimized using gradient descent, discussed in section 2.4 **Gradient Descent**, or some gradient based method such as Adam, discussed in section 2.4 **Adaptive Moment Estimation (Adam)**. In order to use gradient based

algorithms partial gradients of the loss function with respect to all weights in the neural network have to be calculated. Computing these gradients are done using the chain rule, iterating backward, layer by layer, through the neural network [Wikipedia, 2020b].

Common types of loss functions are: *mean squared error*, $l_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ where n is the number of instances $\hat{y}_i$ is the prediction for the ith observation and y_i the true label, for regression problems while, or Cross entropy used for classification $l_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_i^{(k)} log(\hat{p}_i^{(k)})$, where n is the number of instances, K the number of classes, $\hat{p}_i^{(k)}$ the predicted probability that the ith instance belong to class k and $y_i^{(k)}$ the true class. Another common loss function used for binary classification is the hinge loss $l_{hinge} = max(0, c - y\hat{y})$, where c is some constant, $y \in \{-1, 1\}$ the true label and $\hat{y} \in [-1, 1]$ the predicted value [Goodfellow et al., 2016].

2.5 Performance Measures

Metrics are needed in order to measure the performance of models. For classification models, such as anomaly detection models determining whether an instance is anomalous or normal, the evaluation itself can be challenging. The choice of evaluation metric should be picked with the application in mind. For the purpose of the thesis, the metrics used were largely determined by the papers, which we are comparing our results to. The papers used well established metrics.

Model Validation

Cross validation is closely related to the concept of the phenomenon of *overfitting*. Overfitting is essentially to train a model with complexity such that it follows the noise of the training data too closely. This will lead to poor model model performance for new, unseen data and we say that the model is not generalizing well to unseen data [James et al., 2014]. When evaluating the performance of a model an *out of sample* data set or testing set, should be used. Evaluating the model on the same data it was trained on is essentially meaningless since by just having a complex enough model we could get a perfect score. Therefore we use the practice of cross validation.

The idea is to split the available data into a training and test set, where the test set is only used to check the model performance. Further, we have to make decisions for our model, such as the architecture of the neural network, the number of trees used in an isolation forest or what learning rate to use. Since we don't want these decisions to create a model that overfits to the training data we need to evaluate our model decisions on data that was not

used when training the model. Part of the training data is therefore set aside for model validation, called the validation set.

The process of finding the best model is therefore to (i) train several different models on training data, where different model choices are made. Preferably models for all combinations of considered model choices should be trained. (ii) the models are evaluated based on their performance on the validation data, which was not used in the training process. (iii) The best model based on the validation set is chosen to be the final model. (iv) The final model is evaluated on the test set. This is the performance which is released to the public since it is the best estimation of how the model would perform on unseen data [Jake, 2016].

Confusion Matrix

Several metrics for evaluating classification models, such as an anomaly detection model, make use of a *confusion matrix*. The idea for a confusion matrix is to compare predictions with their true labels. In a confusion matrix each row represents a predicted class and each column represents the true class. A confusion matrix for a binary classification is therefore composed of four values displayed in Figure 2.11.

Confusion Matrix

		True Class	
		Anomaly	Normal
Predicted Class	Anomaly	True Positive (TP)	False Positive (FP)
	Normal	False Negative (FN)	True Negative (TN)

Figure 2.11: Confusion Matrix

For a perfect classifier, all instances would either be true positives or true negatives. The false positives are sometimes referred to as type I error and false negatives as type II error [Géron, 2019].

F1 score

From the confusion matrix we can deduce more concise metrics. *Precision* and *recall* are two such metrics. Precision is the fraction of predicted positive instances that are real positive instances while recall is the fraction of real positive instances that are correctly predicted as positive. The metrics can be thought of as measures along each of the axes in the confusion matrix,

precision along the first row and recall along the first column[Powers, 2011]. They are defined accordingly:

$$\text{Precision} = \frac{TP}{TP+FP} \qquad \text{Recall} = \frac{TP}{TP+FN}$$

There is a tradeoff between precision and recall. Moving a threshold to increase the precision will reduce the recall and vice versa. F1-score is a metric which combines the precision and recall into one metric and favours both precision and recall equally[Géron, 2019]. F1-score is given by:

$$\text{F1-Score} = 2\frac{Precision \cdot Recall}{Precision + Recall}$$

Using F1-score as a metric in an anomaly detection setting can be problematic. There are few positive labels, anomalies present in a data set. Splitting the data into a training and test set can result in there not being any anomalies present in the one or the other. The absence of anomalous data points will result in no true positive or false negative values in the confusion matrix since no predicted positive (anomalous) instances will be correct and all predicted negative (normal) will be correct. This will lead to the recall being undefined since we will be dividing by zero and thus also the F1-score being undefined.

AUC

Another metric commonly used is the *Receiver Operating Characteristic* (ROC) curve. The purpose of the ROC curve is to express the capability to discriminate between classes regardless of a threshold. The ROC curve is a curve where the *False Positive Rate* (FPR), x-axis, has been plotted against *True Positive Rate* (TPR), y-axis, where the TPR (which is the same as recall) and FPR are given by:

$$\text{TPR} = \frac{TP}{TP+FN} \qquad \text{FPR} = \frac{FP}{FP+TN}$$

The *Area Under the Curve* (AUC) is the area under the ROC curve. A perfect model would have a AUC of 1 and a model making classifications on random would have an AUC of 0.5 [Powers, 2011]. The goal of any classifier is to try to distinguish between two (or more) distributions. We are able to perfectly separate the two distributions if they don't overlap, resulting in an AUC of 1. There will be false negatives and false positives if the two distributions overlap, illustrated in Figure 2.12, regardless of threshold.

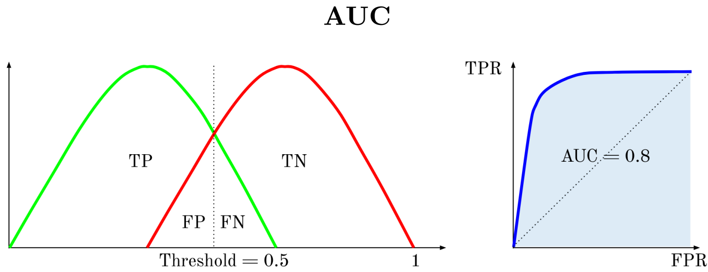


Figure 2.12: We are trying to separate the green and the red distributions. False positives and False Negatives shows up regardless of threshold. The corresponding ROC curve is displayed to the right in blue with the AUC corresponding to the filled area.

For the model in Figure 2.12 with an AUC = 0.8, there is a 80% probability that the model can distinguish between the classes. A classifier with an AUC < 0.5 can be inverted and thus have a AUC > 0.5.

Similar to F1-score, using AUC can be problematic if there is an absence of anomalous data points. This will result in no true positive or false negative values in the confusion matrix, leading to the TPR being undefined since we will be dividing by zero and thus also the AUC being undefined.

3

Methodology

We can now take a look at the models that were taken into consideration in this thesis and the framework that was used. Some implementation details will be provided.

The main aim of the thesis is to address anomalies in a streaming data setting. The model would have to be able to deal with shifts in the data stream and have ways of dealing with missing values. In addition we would like to have some way of updating the model without having to retrain the entire model.

3.1 Data Preprocessing

To evaluate the model performance of our model we measure the performance on benchmark data sets. The benefit by using a public standard benchmark for time series data is that others also have used the dataset and we will be able to compare our results with theirs.

We want to make the comparison with other published models as much of an apples-to-apples comparison as possible. The fact that we are working in an active anomaly detection setting, where the model has access to queried labels, makes it impossible to make the comparison completely fair, due to other benchmarks models being either supervised or unsupervised. However in order to get an indication of the model performance we are going to compare results to other benchmark models. The choice of metrics and proportion of train and test sets were thus determined by the paper containing the benchmark we are comparing our results to.

The performance of the few published active anomaly detection models are dependent on the number of observations that the model was allowed to query. More queries would result in more labels and thus a better model [Das et al., 2019] [Zhang et al., 2019].

Yahoo Webscope Dataset

The publicly available [Yahoo! Labs, 2020] Yahoo Webscope dataset created by Yahoo Labs to benchmark anomaly detection systems was used. The data set includes 367 time series, each containing 1420 - 1680 instances, it is divided into four categories: A1, A2, A3, A4 [Nikolay et al., 2015]. The A1 consists of real time series data of aggregated user logins to the Yahoo network. The remaining A2, A3, A4 data sets consist of synthetic data.

Preprocessing

Each time series was normalized by subtracting the mean and dividing by the standard deviation of the series i.e.

$$x^{(t)}_{normalized} = \frac{x^{(t)} - \bar{x}}{\sigma}.$$

The time series in the Yahoo Webscope dataset were then converted into sliding windows of data displayed in Table 3.1. Using the data itself for both input and target can be seen as unsupervised learning since we don't need anyone to label the data, or it can be seen as supervised since we have data and corresponding labels. In order to mitigate confusion Yann LeCun has proposed calling this type of learning *self-supervised learning*[1]. As displayed in Table 3.1 a time series can thus be split into either $\mathbf{X}$(+ the last row, corresponding to 12:07) if we are using an unsupervised model such as isolation forest, or it can be split into $\mathbf{X}$ and $\mathbf{y}$ if we are using a supervised model such as linear regression. The size of the sliding window was set as a hyperparameter and was determined through cross validation.

Timestamp	x_t
12:01	1
12:02	2
12:03	3
12:04	4
12:05	5
12:06	4
12:07	3

⇒

Timestamp	x_t	x_{t-1}	x_{t-2}	x_{t-3}		y_t	
12:01	1					2	
12:02	2	1				3	
12:03	3	2	1			4	
12:04	4	3	2	1		5	
12:05	5	4	3	2	**X**	4	**y**
12:06	4	5	4	3		3	
12:07	3	4	5	4			

Table 3.1: The time series to the left is converted into sliding windows with a maximum lag of 3. Only the instances highlighted in grey are used as input data $\mathbf{X}$ after creating the sliding windows since we are missing values for the first 3 sliding windows and the last time point is used as corresponding targets $\mathbf{y}$.

[1] `https://twitter.com/ylecun/status/1123235709802905600`

In accordance, with the comparison paper, each time series was split into a training set containing the first 40% and a testing set containing the remaining 60% of each time series in the data set.

3.2 Framework

All anomaly detection systems in this thesis were constructed according to the model framework displayed in Figure 3.1. The model consists of an ensemble of anomaly detection models, each providing an anomaly score $f_i(x_t)$, for the ith ensemble member, for the current input x_t and a Neural Network Meta Model which outputs weights $p_i(x_t)$ corresponding to each ensemble member. The final anomaly score is given by $\sum_{i=1}^{M} p_i(x_t) f_i(x_t)$, where M is the total number of ensemble members.

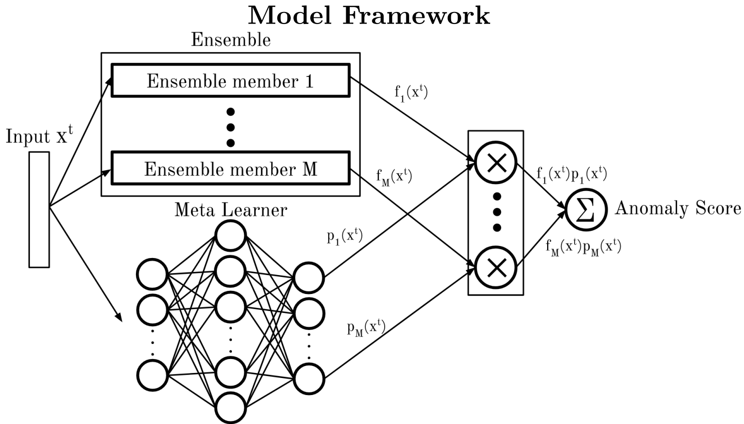


Figure 3.1: The the active anomaly detection framework used.

The idea is that since the meta learner takes instances as input and output weights for each ensemble member, each ensemble member should be weighted according to its previous performance of similar instances. In Figure 3.2 the idea is displayed. If a new instance x_t^* lies in the green region, i.e. 1, and *Ensemble member 1* is most accurate among the ensemble members for instances in that region, then we would like to give that ensemble member a higher weight $p_1(x_t^*)$. In the same way, we want to increase the weight $p_M(x_t^*)$ of *Ensemble member M* if it performs better than the other ensemble members in the orange region, M, and the new instance x_t^* lies in that region. Thus the total output is a dynamically weighted ensemble output where the weighting is based on the input.

Idea of Model Framework

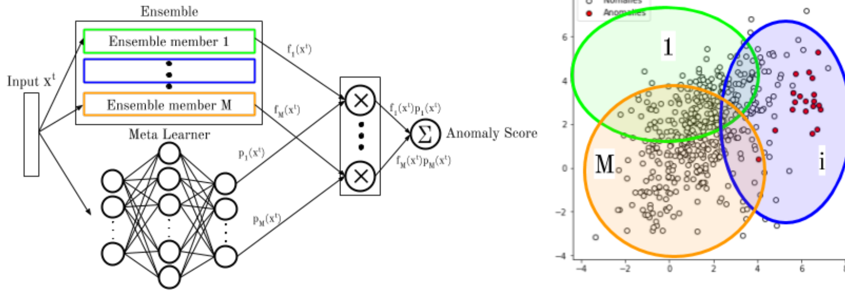


Figure 3.2: The active anomaly detection framework used.

The framework is at the start in an unsupervised setting, i.e. we don't have any labels available. However the model can query a human expert who provides the true labels for the queried instances. The main algorithm for the framework is displayed in Algorithm 2.

Algorithm 2: Framework

Inputs: $\mathbf{X}$ - input data, `ensemble` - Ensemble, `meta_model` - Meta Model, Q - number of queries

Outputs: `meta_model` - Meta Model, `ensemble` - Ensemble, `queried_instances` - queried instances and labels

`queried_instances` = [ ] // Initiate empty list for queried instances and their labels

train(`ensemble`($\mathbf{X}$)) // Train ensemble in an unsupervised setting

for k = *1 to Q* **do**

- $\mathbf{f}_{score}^{(ensemble)}$ = `ensemble`($\mathbf{X}$) // Get anomaly scores from ensemble for each instance
- $\mathbf{p}^{(meta_model)}$ = `meta_model`($\mathbf{X}$) // Get weights from meta_model for each instance
- $\mathbf{a} = \sum_{i=1}^{N} \mathbf{f}_{score}^{(ensemble)} \cdot \mathbf{p}^{(meta_model)}$ // Final anomaly score for each instance
- Let $\tilde{x} = x_i$ where $i = argmax_i(\mathbf{a})$ // Greedy query label for most anomalous instance
- Get label $\tilde{y} = \{-1, 1\}$ for $\tilde{x}$ from human expert
- Append $(\tilde{x}, \tilde{y})$ to `queried_instances`
- Update `meta_model` such that all instances in `queried_instances` are correctly classified

end

return `meta_model`, `ensemble`, `queried_instances`

We start out by training the ensemble without any labels i.e. unsupervised. The ensemble then remains constant throughout the process and we are only adjusting the meta learner. There are two things required in order to make the algorithm work. The first is that there is variation among the models in the ensemble. The second is that we do not restrict the output from the meta model too much, for example a softmax function as output activation in the meta learner, i.e. $\sum_{i=1}^{M} p_i = 1$ and $p_i \in [0, 1]$ for all output

weights p_i could restrict the output too much. The model will have trouble correctly classifying all queried instances if these two requirements are not fulfilled. This is due to the fact that we are only able to make model changes using the output from the meta learner, weights.

Some adjustments to Algorithm 2 will have to be made in order to deal with streaming time data. We would have to have some way to be able to adapt to shifts in the data without having to retrain the entire model. Therefore we will not be keeping all queried instances, but rather some fixed number, determined through cross validation, of labeled instances. When we reach the limit of queried instance we disregard the first queried instance and add the new one. The reason that we don't want to keep old queries is that the data stream is continuously changing and we don't want to update the meta model with regard to data which is not representative of the current data.

In streaming data, queries will be performed slightly differently. At each time t a new data point x_t will be considered and the model will provide an anomaly score. The new instance x_t will be classified as anomalous if the anomaly score is above some threshold th. We are going to query labels for instances which have an anomaly score of $(1 - qf) \cdot th$ where $qf \in [0, 1]$ is a constant determining the query frequency. A higher value for qf will result in more queries and a more accurate model with more human intervention. A lower value for qf will result in less queries being made and a less accurate model. The choice of qf will have to be determined depending on use case of the model. There could be reasons to have a higher value for qf at the training phase of the model and to lower the it later. For example if the model were to be deployed, for a limited amount of time we could request more human intervention while the model is being calibrated. When the model has reached an adequate level, the value for qf could be decreased and less human interaction is needed.

To summarize the framework in a streaming setting, displayed in Algorithm 3, we first initiate the ensemble by training it at some window of length m of data. While there is data stream we (i) get the anomaly score for the instance. (ii) classify the instance as anomalous or normal. (iii) if anomaly score is above the threshold to query, we query a human expert for the true label and (iv) if our prediction does not correspond to the true label, update the oldest part of the ensemble and the meta learner.

Algorithm 3: Framework Streaming Data

```
Inputs: X - input data, ensemble - Ensemble, meta_model - Meta Model, th -
  Threshold, qf - Query Frequency Constant
Outputs: meta_model - Meta Model, ensemble - Ensemble, queried_instances - queried
  instances and labels

queried_instances = [ ]        // Initiate empty list for queried instances and their labels
train(ensemble(X))    // Train ensemble in an unsupervised setting on the first m instances

while data stream do
    f_score^(ensemble)(t) = ensemble(x(t))    // Anomaly scores from ensemble for current instance
    p^(meta_model)(t) = meta_model(x(t))    // Weights from meta_model for current instance
    a(t) = sum_{i=1}^{N} f_score^(ensemble)(t) · p^(meta_model)(t)    // Get weighted sum of anomaly scores

    if a(t) > th then
        ŷ(t) = 1                                   // Classify instance as anomalous
    else
        ŷ(t) = −1                                  // Classify instance as normal
    end

    if a(t) > qf · th then
        Get label y(t) = {−1,1} for x(t) from human expert.
        Append (x(t),y(t)) to queried_instances and remove oldest query.
        if ŷ(t) ≠ y(t) then
            Retrain oldest *% of ensemble members at last m instances
            Update meta_model such that all instances in queried_instances are
              correctly classified
        end
    end
end
return meta_model, ensemble, queried_instances
```

In Algorithm 3 the number of instances to keep in `queries_instances` before removing the oldest queried instances and the percentage * of oldest ensemble members to update are both set as hyperparameters and are chosen through cross validation.

3.3 The Ensemble

Different ensemble compositions were used for different problems.

Ensemble of Isolation Trees

Isolation forest was used when working with non-time series data such as the Toy Data displayed in Figure 2.2 in Section 2.3 **Unsupervised learning**. Each ensemble member was composed of a single isolation tree. Each tree was trained on a bootstrap sample from the training data. Each tree assigned an anomaly score to each instance and the final anomaly score was a weighted sum of all tree outputs.

However, in a streaming setting the isolation forest approach did not perform well. Isolation forest makes splits along a randomly chosen dimension, which in a time series setting with sliding windows, are represented by the

time series itself, where each dimension is shifted one step compared to the previous. Theoretically this results in the anomalous instance being predicted as an anomaly in any dimensions, i.e. all sliding windows passing over the anomaly will predict its window as anomalous shown in Figure 3.3. We want to detect the anomaly as soon as it appears and therefore want to tag the sliding window first reaching the anomaly as anomalous, i.e. the upper time series in Figure 3.3. We preferably don't want to tag a later sliding window, lower time series in Figure 3.3, as anomalous as well. Hence we are only interested in categorizing instances along dimension 5 as normal/anomalous which defeats the purpose of having a sliding window. Additionally, in practice isolation forest will not be able predict each anomaly in all dimensions, therefore only some of the sliding windows sliding over an anomaly will be anomalous, resulting in an inaccurate model.

Isolation Forest for Streaming Time Series Data

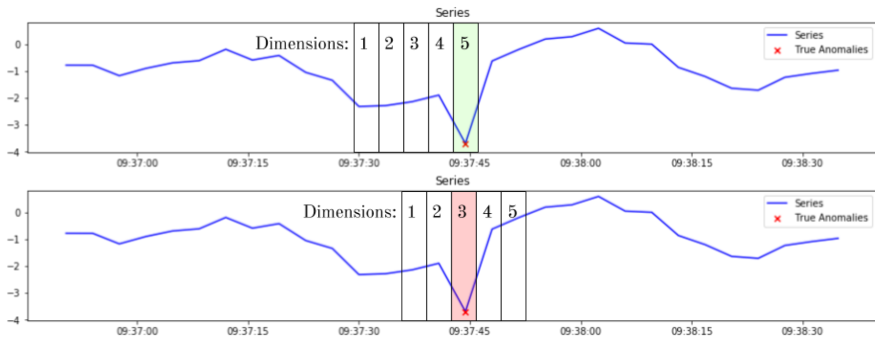


Figure 3.3: A sliding window of length five is sliding over a time series containing an anomaly tagged with a red cross. When isolation forest is used for streaming time series data we want to tag the sliding window first reaching the anomaly as anomalous, i.e. the upper time series. We preferably don't want to tag a later sliding window, lower time series, as anomalous as well. Hence we are only interested in categorizing instances along dimension 5 as normal/anomalous.

Ensemble of Linear Regression Models

In order to better deal with streaming data, a different approach was needed. Instead of using a model such as isolation forest, which is specifically used for anomaly detection, regression models were used. The first approach was an ensemble composed of linear regression models. Each of the models in the ensemble made a prediction $\hat{y}$ based on previous time steps. The number of previous time steps to take into account for each model was randomized in

an interval where the interval ranges were treated as hyperparameters which were determined through cross validation.

In order to (i) introduce more randomness, (ii) differentiate between model using the same number of previous steps and (iii) to better deal with missing values in the data for the ensemble members, a different random proportion of the time series was imputed by the previous value for each ensemble member displayed in Table 3.2.

Timestamp	x_t
12:01	1
12:02	2
12:03	3
12:04	4
12:05	5
12:06	4
12:07	3

⇒

Timestamp	x_t	x_{t-1}	x_{t-2}	x_{t-3}		y_t	
12:01	1					2	
12:02	2	1				2	
12:03	2	2	1			4	
12:04	4	2	2	1		5	
12:05	5	4	2	2	**X**	5	**y**
12:06	5	5	4	2		3	
12:07	3	5	5	4			

Table 3.2: The highlighted values 3 and 4 from the original time series to the left are imputed by the previous value, 2 and 5. The time series is then converted using sliding windows with a maximum lag of 3 as described in section 3.1 **Preprocessing**. Corresponding imputed values are highlighted after creation of sliding windows.

The imputation ensures that the prediction will be different for two models in the ensemble even if they are based on the same number of previous instances. This increases the variation within the ensemble. Since the model is trained using imputed values it's more robust when dealing with missing values when doing predictions. We simply impute these missing values with the previous value.

The final decision on whether an instance is an anomaly or not is then based on a threshold compared to the true value. We make the assumption that we are able to predict the next value in the data stream accurately enough for our prediction to be within some threshold of the true value, if the prediction is outside of the threshold interval we categorize the instance as anomalous. To further explain the process, at time t each ensemble member make a prediction for the value in the time series at time $t+1$, $\hat{y}_{t+1}^{(i)}$ for the ith ensemble member. At time $t+1$ when we have access to the true value y_{t+1} we calculate the anomaly score for the ensemble member as $f_i(x_{t+1}) = |y_{t+1} - \hat{y}_{t+1}^{(i)}|$.

How is the threshold determined? Normally a poor threshold will ruin the entire model. An overly large threshold will never categorize any instances as anomalous and a too small threshold will categorize all instance as anomalous and in between the two extremes there is a trade-off between the number of type I and type II errors. The answer is that when using our framework, it

doesn't matter what we set the threshold to. The framework will adapt to it for us. When weighting the anomaly scores from the different ensemble members the meta learner will provide the weighting to the anomaly scores that satisfies the queried labels accordingly.

$$\begin{aligned}
&\sum_{i=1}^{M} p_i(x) f_i(x) > th \\
\Leftrightarrow &\sum_{i=1}^{M} c p'_i(x) f_i(x) > th \\
\Leftrightarrow &\sum_{i=1}^{M} p'_i(x) f_i(x) > \frac{th}{c}
\end{aligned}$$

Here M are the number of models in the ensemble and c a constant. We see that since the meta model can choose c, the threshold can be set to any value and the anomaly score will adapt accordingly. The phenomena of the meta learner changing the anomaly score to address the set threshold can be seen in Figure 3.4. Here the anomaly score, displayed in the bottom figure, initially is above the anomaly threshold for all instances in the first window (100 instances) of training data. The last instance in the window is therefore labeled as anomalous and the model asks for the true label of the instance, which is non-anomalous. The prediction is thus wrong, since the instance is not an anomaly, and the meta model is updated to categorize the labeled instance correctly. After the query, the anomaly score drops below the anomaly threshold where it stays as training continues. This procedure is repeated every time when training a model. The first instance is categorized as anomalous and the meta model updates the weighting of the ensemble members to accommodate the chosen threshold.

Anomaly Score at Start of Training

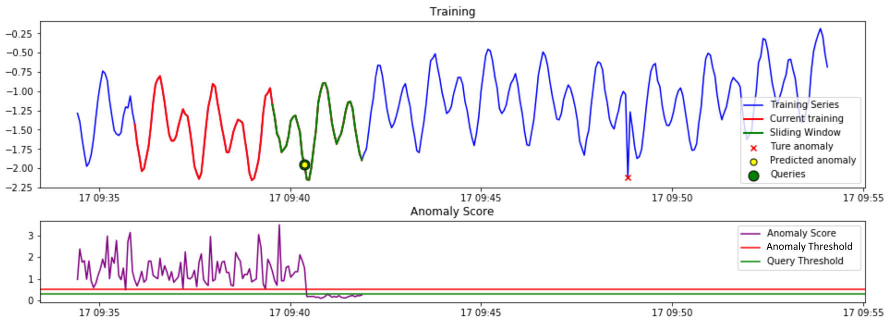


Figure 3.4: The training process for the time series in blue in the upper figure. The red cross indicates the true anomaly, the green line the maximum length used by the ensemble members to make a prediction for the next time step, the red line the training data used to train the ensemble members when they are updated, the yellow circle a predicted anomaly and the green circle the queried instance. The corresponding anomaly score is displayed in the bottom figure and thresholds for tagging an instance as anomalous and to query. Initially the anomaly score is above the anomaly threshold for all instances in the first window (100 instances) of training data. The last instance in that window is therefore labeled as anomalous and the model asks for the true label of the instance, which is non-anomalous. The prediction is thus wrong, since the instance is not an anomaly, and the meta model is updated to categorize the labeled instance correctly. After the query the anomaly score drops below the anomaly threshold where it stays as training continues.

It is worth mentioning that when using regression models in the ensemble, each model outputs its own anomaly score rather than a prediction. If they were to output a prediction, the final classification as normal/anomalous would be determined by

$$|(\sum_{i=1}^{M} p_i \hat{y}_{t+1}^{(i)}) - y_{t+1}| > th.$$

Where M is the number of models in the ensemble $\hat{y}_{t+1}^{(i)}$ the predicted value at the next time step $t+1$ for model $i \in [1, M]$, y_{t+1} the true time series value at time $t+1$ and p_i the weight for model i given by the meta model. We would therefore not have the same influence over the threshold by adjusting the weights p_i.

3.4 The Meta Model

The meta model is a multi layer perceptron consisting of 1 hidden layer where the number of nodes are three times the number of output nodes, which is the same as the number of ensemble members. The sigmoid function was used as the activation function both in the hidden and output layer. The model architecture was taken from [Das et al., 2019]. The choice of the sigmoid function as the output function over for example the softmax function is motivated by the reduced restriction of the weighting of the ensemble members. Using a softmax function instead resulted in the model sometimes getting stuck during the training process due to the restriction, $\sum_{i=1}^{M} p_i = 1$. For example if the ensemble consists of three models giving the anomaly scores 0.62, 0.65, 0.70 for an non-anomalous instance and our threshold is 0.5 to categorize an instance as anomalous. If $p_1 + p_2 + p_3 = 1$ and $p_i \in [0, 1]$ for $i = 1, 2, 3$ then $p_1 0.62 + p_2 0.65 + p_3 0.70 > 0.5$. However relieving the model of the restriction, $\sum_{i=1}^{M} p_i = 1$ will result in the model being able to classify the instance as non-anomalous.

Worth mentioning is that [Das et al., 2019] choose to use Xavier initialization [Glorot and Bengio, 2010] (named normalized initialization in the paper) when initializing the weights. The meta learner was able to train and converge to a solution fast enough using Xavier initialization and sigmoid as activation function in all layers. However, it is specifically mentioned by Glorot and Bengio in [Glorot and Bengio, 2010] that:

> - For tanh networks, the proposed normalized initialization can be quite helpful, presumably because the layer-to-layer transformations maintain magnitudes of training iterations is a powerful investigative tool for understanding training difficulties in deep nets.
> - Sigmoid activations (not symmetric around 0) should be avoided when initializing from small random weights, because they yield poor learning dynamics, with initial saturation of the top hidden layer.

This indicates that we could expect faster training using tanh as activation function (at least in the hidden layer). The reason we normalized the input to the meta model was in order for the network to consider each input feature equally and to have both positive and negative values in the input, which improves the expressiveness of the layer. Ideally, we would like the input to each layer in the neural network to be normally distributed for the same reasons. Using Xavier initialization and sigmoid as activation function in a neural network (as in [Das et al., 2019]) will result in non normally distributed outputs from each layer. The impact on the output mean and standard deviation using Xavier initialization can be seen here [2].

[2] https://nbviewer.jupyter.org/github/LurreMcFly/masters_thesis/blob/master/xavier_initialization.ipynb

Loss Function

The loss function used to train the meta model has to be designed in a way where we are able to train the weights in the neural network based on weighted ensemble output. The final loss function l_{MM}, equivalent to the one used in [Das et al., 2019], consists of two main parts:

$$l_{MM} = \underbrace{\frac{1}{|Q_t|} \sum_{(\mathbf{x},y) \in Q_t} l_{AAD}(\mathbf{x},y)}_{(1)} + \underbrace{\frac{\lambda_1}{|X_t|} \sum_{\mathbf{x} \in X_t} l_{prior}(\mathbf{x})}_{(2)} + \underbrace{\lambda_2 \sum \omega^2}_{(3)}.$$

Where Q_t is the set and $|Q_t|$ the number of queried instances at time t, X_t the set and $|X^{(t)}|$ the number of instances at time t, ω the weights in the neural network, λ_1 and λ_2 hyperparameters, scaling the impact of (2) and (3). The first part (1) of the loss function l_{AAD} is defined in equation 1.

$$l_{AAD}(\mathbf{x}_t, y) = \underbrace{max(0, y(th - a(\mathbf{x}_t))}_{(a)} + \underbrace{max(0, y(a(\mathbf{x}_{t-1}) - a(\mathbf{x}_t))}_{(b)} \tag{1}$$

Where $y \in \{-1, 1\}$ (-1 normal instance and 1 anomalous instance) is the true label, $a(\mathbf{x}_t) = \sum_{i=1}^{N} \mathbf{f}_{score}^{(ensemble)}(\mathbf{x}_t) \cdot \mathbf{p}^{(meta_model)}(\mathbf{x}_t)$ the anomaly score for for $\mathbf{x}_t$ and q the threshold over which we categorize an instance as anomalous. Both of the terms (a) and (b) displayed in Figure 3.5 making up l_{AAD} are hinge losses. The first term (a) penalizes a wrongly classified instance by $|th - a(\mathbf{x}_t)|$ the difference between the threshold th and the predicted anomaly score $a(\mathbf{x}_t)$. The second term (b) penalizes a decrease in model performance after a label feedback by $|a(\mathbf{x}_{t-1}) - a(\mathbf{x}_t)|$, the difference between the anomaly score at previous query and current anomaly score.

Hinge Loss Functions.

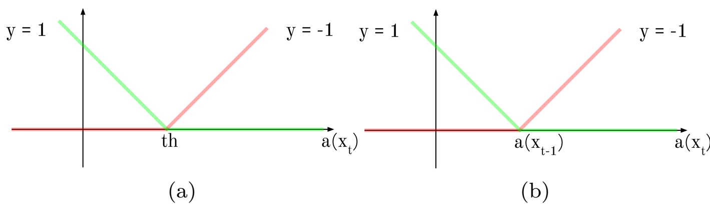


Figure 3.5: The two hinge loss terms making up l_{AAD}. The first term (a) penalizes a wrongly classified instance by $|th - a(\mathbf{x}_t)|$ the difference between the threshold th and the predicted anomaly score $a(\mathbf{x}_t)$. The second term (b) penalizes a decrease in model performance after a label feedback by $|a(\mathbf{x}_{t-1}) - a(\mathbf{x}_t)|$, the difference between the anomaly score at previous query and current anomaly score.

The loss terms contribute to the loss depending on the anomaly score $a(\mathbf{x}_t)$ as follows:

$$\text{(a)} \begin{cases} \text{Instance is correctly classified i.e. } a(\mathbf{x}_t) \text{ is on correct side of } th : 0 \text{ loss} \\ \text{Instance is wrongly classified i.e. } a(\mathbf{x}_t) \text{ is on wrong side of } th : |th - a(\mathbf{x}_t)| \text{ loss} \end{cases}$$

$$\text{(b)} \begin{cases} \text{Anomaly score } a(\mathbf{x}_t) \text{ improves over previous anomaly score } a(\mathbf{x}_{t-1}) : 0 \text{ loss} \\ \text{Anomaly score } a(\mathbf{x}_t) \text{ deteriorate over previous anomaly score } a(\mathbf{x}_{t-1}) : |a(\mathbf{x}_{t-1}) - a(\mathbf{x}_t)| \text{ loss} \end{cases}$$

The second part (2) of the final loss function l_{MM} is a cross entropy loss defined in equation 2.

$$l_{prior}(\mathbf{x}_t) = -\sum_{m=1}^{M} b\log(p_m(\mathbf{x}_t)) + (1-b)\log(1-p_m(\mathbf{x}_t)) \tag{2}$$

Where M is the number of ensemble members, $p_m(\mathbf{x}_t)$ the weight set by the meta model for the mth member and b a prior, set to $\frac{1}{M}$. The loss puts a prior over the weighting of the ensemble members to weight the members equally. Deviation from the prior is penalized according to Figure 3.6, where the cross entropy function is displayed for one dimension.

Cross Entropy in One Dimension with Prior b.

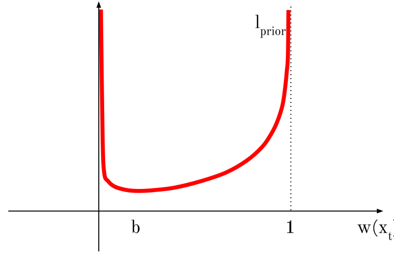


Figure 3.6: The cross entropy loss in the dimension $p(x_t)$ with the prior b.

Minimizing the loss function thus obtains three main goals. (i) Reduces misclassified queried instances, (ii) prevents the anomaly from previously queried instances to deteriorate and (iii) places a prior over the ensemble members, such that they are equally weighted.

The last part (3) of the loss function is an L2 regularization term, penalizing the squared magnitude of the weights in the neural network.

3.5 Setup

Model 1

Model 1 was composed of an ensemble with linear models described in section 3.3 **Ensemble of Linear Regression Models**. Each linear regression model containing a randomized number of coefficients and each with imputed values as described in Table 3.2. The meta model was the MLP first described in section 3.4 **The Meta Model**, composed of the same number of outputs as the number of ensemble members and a single hidden layer with three times the number of nodes as the output layer. No transformations other than normalizing were done to the time series data.

The model hyperparameters that were tuned for this model using validation set were:

- Number of ensemble members.
- Interval from which the number of previous instances to use in each linear regression models is drawn randomly.
- The impute rate in the linear models.
- Number of previous instances to take into account when updating the ensemble.
- The proportion of ensemble members to update when updating the ensemble.
- Number of queried labels to keep.
- Learning rate for the Adam optimizer.
- The coefficients λ_1 and λ_2 in the loss function.
- Coefficient T_0 associated with the cosine annealing.

Hyperparameters were selected using a grid search where the model resulting in the maximum validation score was selected for each data set.

4

Results

4.1 Yahoo Webscope Dataset

The models were evaluated on the four parts of Yahoo Webscope Dataset and compared to results presented in [Mohsin et al., 2019][1]. The metric used in [Mohsin et al., 2019] was AUC. For each of the four parts of the data set a new model was trained with its own hyperparameter tuning. Similar to [Mohsin et al., 2019] the average AUC for all time series in that part of the data set was used as metric. As mentioned in section 2.5 **AUC** the use of AUC as a metric is problematic when there are no positive label (anomalies) present in the data set. After splitting the data into training (first 40%) and test (remaining 60%) sets some of the time series lacked anomalous data points. These time series were removed [2] both during hyperparameter tuning and during testing. However in the second part of the Yahoo Webscope Dataset A2 all anomalies occurred in the last 60% of the data set, meaning that we have no way of evaluating the hyperparameter tuning using AUC[3]. The model used for A2 part is therefore the same model used for A1 part. There is no reason to believe that the hyperparameters used when evaluating A2 are optimal. The results are displayed in Table 4.1. It is also worth to remind the reader that this comparison is made with unsupervised learning algorithms without any access to the true labels.

[1] The most recent published results I could find for Yahoo Webscope Dataset.

[2] I had an email correspondence with the authors of the paper [Mohsin et al., 2019] to be sure I dealt with this problem in the same way they did.

[3] The authors of the paper [Mohsin et al., 2019] did not respond to how they dealt with this problem.

Results for Yahoo Webscope Dataset

Benchmark	A1	A2	A3	A4
iForest[**Liu et al., 2008**]	0.8888	0.6620	0.6279	0.6327
OCSVM[**Ma and Perkins, 2003**]	0.8159	0.6172	0.5972	0.6036
LOF[**M.M. et al., 2000**]	0.9037	0.9011	0.6405	0.6403
PCA[**M.L. et al., 2003**]	0.8363	0.9234	0.6278	0.6100
TwitterAD[**twitter**]	0.8239	0.5000	0.6176	0.6534
DeepAnT[**Munir et al., 2019**]	0.8976	0.9614	0.9283	0.8597
FuseAD[**Mohsin et al., 2019**]	**0.9471**	**0.9993**	**0.9987**	**0.9657**
My Framework (0.4)	0.9088	0.9787	0.8998	0.8123
My Framework (0.1)	0.9042	0.9769	0.8993	0.7990

Table 4.1: The average AUC per part of Yahoo Webscope Dataset for comparison with other state-of-the-art anomaly detection methods. Results for My Framework are displayed with a query frequency of 0.4 and 0.1.

Two versions of the framework were used, the first with a query frequency of 0.4 and the second with a query frequency of 0.1 both displayed in Table 4.1. My Framework has the second highest AUC for the benchmarks A1 and A2, and third highest AUC for A3 and A4 benchmark. The second model queries fewer instances from the human expert but has a lower AUC for all four benchmarks.

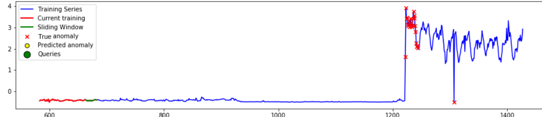
Snapshots of the prediction process for My Framework with query frequency 0.4 and 0.1 are displayed in Figure 4.1 for time series number 10 in the A1 benchmark data set. The time series displays data of aggregated user logins to the Yahoo network. The optimal hyperparameters were used for My Framework, determined through cross validation. The corresponding AUC for the series displayed in Figure 4.1 was 0.9987 for the model with query frequency 0.4 and 0.9995 for the model with query frequency 0.1. The F1-score was 0.8571 and 0.8727 respectively. The model using a lower query frequency thus performed better in terms of AUC and F1-score for this time series, which was not the general case as displayed in Table 4.1.

Comparing the snapshots in Figure 4.1 a lower query frequency results in less queries being made. The model with the higher query frequency made a total of 53 queries, while the model with lower query frequency made 39 queries. This is especially observable when comparing (e) and (f) where the interval over which the queries spread is much narrower in (e) than the interval in (f) meaning that more queries are being made over the same amount of time. The same phenomenon is observable when comparing (g) and (h).

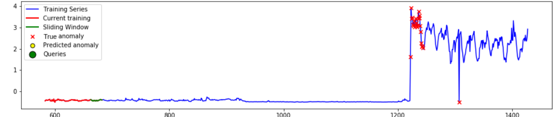
It is worth mentioning here that in Figure 4.1 it is observable how the model adjusts to the new level after the jump in values after instance 1200. When the time series settles at the new level, and we no longer consider instances at the new level as anomalous, only three instances (for both models)

My Framework, Different Query Frequencies Prediction Process for Series 10 in A1 Benchmark

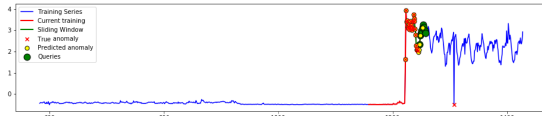
Query Frequency: 0.4 | **Query Frequency: 0.1**



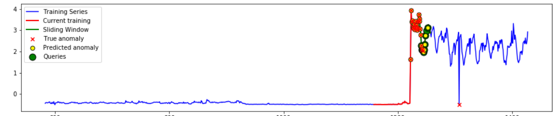
(a) Instance 662



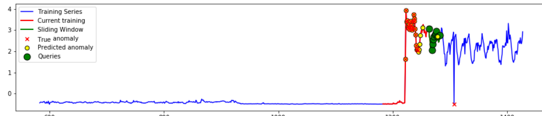
(b) Instance 662



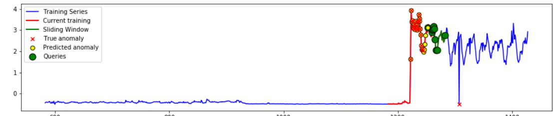
(c) Instance 1237



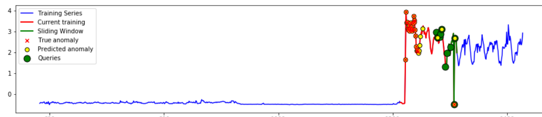
(d) Instance 1237



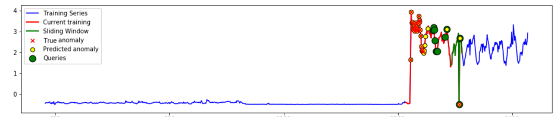
(e) Instance 1262



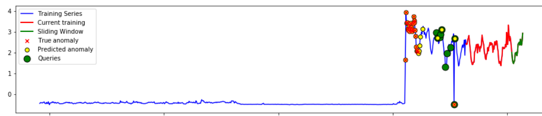
(f) Instance 1262



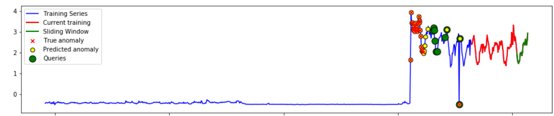
(g) Instance 1292



(h) Instance 1292



(i) Instance 1407



(j) Instance 1407

Figure 4.1: Snapshots of the prediction process for My Framework with tuned hyperparameters for time series 10 displaying aggregated user logins to the Yahoo network in the A1 benchmark data set. The model used in the left column uses a query frequency of 0.4 while the right column uses a query frequency of 0.1. The AUC and F1-score were 0.9987 and 0.8571 respectively for the model used in the left column. The AUC and F1-score were 0.9995 and 0.8727 respectively for the model used in the right column. Snapshots have been taken for each of the two models when making predictions for instances 662, 1237, 1262, 1292 and 1407 (corresponding to instances 100, 575, 600, 630 and 745 of test set). The entire prediction process is displayed at `https://gifyu.com/image/n2hr` for query frequency 0.4 and at `https://gifyu.com/image/n2h1` for query frequency 0.1.

are flagged as anomalous while not being real anomalies (the yellow circles without a red cross at around 1230 observable in Figure 4.1 (c), (d), (e), (f), (g), (h), (i) and (j)) after the initial surge in anomalies after instance 1200.

The anomaly at approximately instance 1300, which is at the level that was previously considered to be normal, is correctly flagged as being an anomaly by the framework. The remaining instances are all rightfully flagged as normal. This shows that the framework has adjusted to the change in the time series.

5

Discussion

The proposed framework works as intended. It is able to adapt to the streaming data, updating the ensemble members to accommodate the data stream and it can be updated based on human intervention. In the thesis the framework has been used to detect anomalies in time streaming data. The application of the framework is however not restricted to anomaly detection and could be applied to other problems.

Comments on Results

It should be stated again that the results presented in Table 4.1 are not an apples-to-apples comparison. The models we are comparing to are operating in an unsupervised setting while the framework proposed have access to labels by making queries.

The results in Table 4.1 were taken from [Mohsin et al., 2019] and not verified. The way the testing of the reference models were conducted could therefore differ from how the testing of the framework in the thesis was conducted. The framework in this thesis was conducted in a streaming fashion, where the model was fed one instance at a time. In Table 4.1 the model **iForest** is the Isolation forest model discussed in Section 2.4 **Isolation Forest** with which initial testing was done. Using Isolation forest for streaming time series data runs into the problem discussed in Section 3.3 **Ensemble of Isolation Trees** resulting in worse results than the ones displayed in Table 4.1, making me question that the testing was done in the same way. Having access to the entire time series and in hindsight detect if an instance is anomalous or not is an easier problem than determining if the next instance in a stream is anomalous.

It is also worth mentioning that all hyperparameter tuning was done using a query frequency of 0.4. The model with query frequency of 0.1 could possibly improve by doing a hyperparameter tuning with query frequency of 0.1.

Model Remarks

My intuition is that a large improvement in the accuracy of the predictions made by each individual member in the ensemble would not improve the

framework as much as I first thought. This is due to the fact that the models in the ensemble are weighted by the meta model. A minor difference in how the meta model weights the ensemble members could easily ruin model, independent of the accuracy of the ensemble members. Therefore the meta model is of great importance and there is no reason to belief that the model used in the thesis is optimal. As discussed in section 3.4 **The Meta Model** we know that the initialization of weights in the model is in fact sub optimal.

Since we want the meta model to fit perfectly to all labels provided, one could argue that we essentially want the meta model to overfit to the labeled data. However, we still want the framework to generalize to new data. We are therefore using a prior to equally weight the ensemble members and L2 regularization in the loss function. Both these terms are forcing the meta model to leverage the entire ensemble rather than weighting a single ensemble member by 1 and the remaining by 0 which would probably not generalize to the observations we don't have label to.

Another thing to mention is that there is a trade off between a query frequency and model performance. More queries, more labels, better model. The benchmark dataset is constructed in such a way that anomalies are present in the time series. If the framework was to be deployed for a real world application we could imagine that some queries would have to be made when initializing the model. After learning the normal behaviour of the streaming data no queries would have to be made as displayed up until instance 1200 in Figure 4.1 where no queries were made. In a real world setting, where time series are not constructed to contain anomalies and with long times between changes in the data stream, the framework would not make frequent queries. When a query would be needed there are either changes in the data stream or anomalies present, at which point human intervention is probably needed anyways.

Improvements Future Work

The first improvement that should be made to the framework is to make use of transformations to the input data. Only normalization was used for the testing of the framework in the thesis. Since we are using an ensemble, different ensemble members could make use of different transformations for the input data. Examples of such transformations could be the difference to previous instance, i.e. $x_t - x_{t-1}$, trend, seasonality or the residuals after a time series decomposition, to name some examples. For future work I would therefore suggest looking into transformations that could be applied. My feeling is that applying transformations is the single thing that would have the most beneficial impact on the framework.

The second thing to look into is to use another meta model than the one in this thesis. I would suggest looking into models that have a track record of performing well for time series such as different types of RNN's. Another model that I would look into is the model **DeepAnT** [Munir et al., 2019]

from Table 4.1 which is a CNN. This model was used (in combination with an ARIMA model) for the best performing model in Table 4.1 **FuseAd**[Mohsin et al., 2019].

Another improvement would be to dynamical set the query frequency. The main reason to not have a constant query frequency is that the ensemble members are updated by training on the (almost) same data when two queries are made for two succeeding time steps x_t and x_{t+1}. We could therefore, for example, restrict succeeding queries to increase the variability in the models used in the ensemble.

5.1 Conclusion

To answer the questions stated in Section 1.1 **Problem Description**:

1. **Can we create a framework which can be used to find anomalies in, and adjust to streaming time series data?**
   Yes we can. The framework proposed in this thesis is able to identify anomalies in streaming time series data. However we can not guarantee that the framework is able to find all anomalies in all time series. As shown in Figure 4.1 the framework is able to adapt to changes in the data stream.

2. **Can we update the framework such that it can improve by human intervention.**
   Yes we can. The model is allowed to query samples and with the use of a second, metalearning system, we are able to update the model such that it improves after human intervention. The improvement after human intervention is displayed in Figure 3.4 showing how the framework adapts the anomaly score after a query being made.

3. **How well does our framework compare to other anomaly detection systems for streaming data?**
   A comparison of how well the framework performperforms compared to other anomaly detection systems is displayed in Table 4.1. The comparison is however not completely fair discussed in the **Model Remarks** section of the **Discussion**.

Bibliography

Ahmad, S. and S. Purdy (2016). *Real-time anomaly detection for streaming analytics*. eprint: 1607.02480.

Carreño, A., I. Inza, and J. Lozano (2019). "Analyzing rare event, anomaly, novelty and outlier detection terms under the supervised classification framework". *Artificial Intelligence Review*. DOI: 10.1007/s10462-019-09771-y.

Das, S., M. R. Islam, N. K. Jayakodi, and J. R. Doppa (2019). *Active anomaly detection via ensembles: insights, algorithms, and interpretability*. arXiv: 1901.08930 [cs.LG].

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Inc.

Glorot, X. and Y. Bengio (2010). *Understanding the difficulty of training deep feedforward neural networks*.

Goernitz, N., M. M. Kloft, K. Rieck, and U. Brefeld (2014). "Toward supervised anomaly detection". *Journal of Artificial Intelligence Research, 46:235–262*. DOI: 10.1613/jair.3623.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements of Statistical Learning*. Springer New York Inc.

Jake, V. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

James, G., D. Witten, T. Hastie, and R. Tibshirani (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated. ISBN: 1461471370.

Kingma, D. P. and J. Ba (2015). *Adam: a method for stochastic optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

LAPACK (2020). *Linear least squares (lls) problems*. `https://www.netlib.org/lapack/lug/node27.html`.

Lemke, C., M. Budka, and B. Gabrys (2013). "Metalearning: a survey of trends and technologies". *Artificial intelligence review vol. 44(1)*. DOI: `10.1007/s10462-013-9406-y`.

Liu, F. T., K. M. T. Ting, and Z.-H. Zhuo (2008). "Isolation forest". *2008 Eighth IEEE International Conference on Data Mining* **ISSN 2374-8486**.

Loshchilov, I. and F. Hutter (2016). *Sgdr: stochastic gradient descent with warm restarts*. arXiv: `1608.03983 [cs.LG]`.

M.L., S., C. S.C., S. K., and C. L. (2003). "A novel anomaly detection scheme based on principal component classifier." *Miami Univ Coral Gables FL Department of Electrical and Computer Engineering; Coral Gables, FL, USA*.

M.M., B., K. H.P., N. R.T., and S. J. (2000). "Lof: identifying density-based local outliers". *Proceedings of the 2000 ACM SIGMOD international conference on Management of data; Dallas, TX, USA* **3**, pp. 93–104.

Ma, J. and S. Perkins (2003). "Time-series novelty detection using one-class support vector machines". **3**, 1741–1745 vol.3.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill ScienceEngineeringMath.

Mohsin, M., S. S. Ahmed, C. M. Ali, D. Andreas, and A. Sheraz (2019). "Fusead: unsupervised anomaly detection in streaming sensors data by fusing statistical and deep learning models". DOI: `10.3390/s19112451`.

Munir, M., S. A. Siddiqui, A. Dengel, and S. Ahmed (2019). "Deepant: a deep learning approach for unsupervised anomaly detection in time series". *IEEE Access* **7**, pp. 1991–2005.

Nikolay, L., A. Saeed, and F. Ian (2015). *Generic and scalable framework for automated time-series anomaly detection*. eprint: `1901.08930`.

Powers, D. M. W. (2011). "Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation". *Journal of Machine Learning Technologies* **2**:1, pp. 37–63.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). *Learning internal representations by error propagation*.

Schaul, T. and J. Schmidhuber (2010). *Metalearning*. `http://www.scholarpedia.org/article/Metalearning`. DOI: `10.4249/scholarpedia.4650`.

scikit-learn (2020). *Sklearn.linear_model.linearregression*. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html`.

scipy (2020). *Scipy.linalg.lstsq.* https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lstsq.html.

Subutai, A. and P. Scott (2016). *Real-time anomaly detection for streaming analytics.*

Synced (2019). *Yoshua bengio on the turing award, ai trends, and 'very unfortunate' us-china tensions.* https://medium.com/syncedreview/yoshua-bengio-on-the-turing-award-ai-trends-and-very-unfortunate-us-china-tensions-4315d3642171. [Online; accessed 20-Feb-2020].

Trefethen, L. N. and D. Bau (1997). *Numerical Linear Algebra.* SIAM. ISBN: 0898713617.

Wang, X., Y. Du, S. Lin, P. Cui, and Y. Yang (2019). "Self-adversarial variational autoencoder with gaussian anomaly prior distribution for anomaly detection". *CoRR* **abs/1903.00904**.

Wikipedia (2020a). *Active learning (machine learning).* https://en.wikipedia.org/wiki/Active_learning_(machine_learning). [Online; accessed 20-Feb-2020].

Wikipedia (2020b). *Backpropagation.* https://en.wikipedia.org/wiki/Backpropagation.

Wikipedia (2020c). *Curse of dimensionality.* https://en.wikipedia.org/wiki/Curse_of_dimensionality.

Wikipedia (2020d). *Reinforcement learning.* https://en.wikipedia.org/wiki/Reinforcement_learning. [Online; accessed 19-Feb-2020].

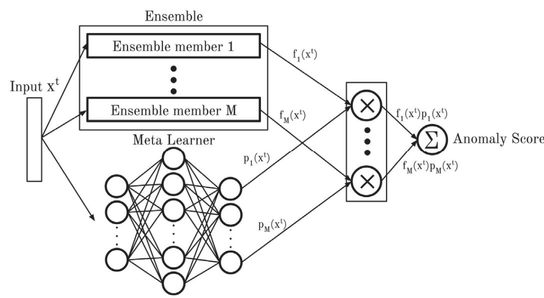
Yahoo! Labs (2020). *S5 - a labeled anomaly detection dataset, version 1.0(16m).* https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70. [Online; accessed 17-Feb-2020].

Zhang, Y., P. Zhao, S. Niu, Q. Wu, J. Cao, J. Huang, and M. Tan (2019). *Online adaptive asymmetric active learning with limited budgets.*

<table>
<tr><td rowspan="3">Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden</td><td>Document name
MASTER’S THESIS</td></tr>
<tr><td>Date of issue
June 2020</td></tr>
<tr><td>Document Number
TFRT-6101</td></tr>
<tr><td>Author(s)
Jonas Lundgren</td><td>Supervisor
Kenneth Ulrich, Sentian.ai, Malmö
Pontus Giselsson, Dept. of Automatic Control, Lund University, Sweden
Bo Bernhardsson, Dept. of Automatic Control, Lund University, Sweden (examiner)</td></tr>
<tr><td colspan="2">Title and subtitle
Anomaly Detection in Streaming Time Series Data Using Active Learning and Metalearning</td></tr>
</table>

Abstract

In this thesis a framework for finding anomalies in streaming data is proposed. The framework proposed is not necessarily applicable only to problems in anomaly detection, but could be applied to other problems as well. There are three main concepts at play in the framework: (i) Active Learning, a learning algorithm which can query a human specialist for labels of instance such that the model can improve, from an otherwise unlabeled data set. (ii) Ensemble which is a combination of models, often weaker models, where the idea is that the combined result from all models will mitigate the error in every single model and thus provide better results. (iii) Metalearning which is the concept of having a second model learn model characteristics for a problem. In this thesis metalearning will be used to weight ensemble members.
The framework is displayed in Figure 0.1. The meta learner takes instances as input and output weights for each ensemble member according to its performance of previous similar instances. Thus the total output is a dynamically weighted ensemble output where the weighting is based on the input. When a human expert provides label feedback on misclassified instances only the meta learner is updated in order to provide new weights for the ensemble to suppress the error and not the entire ensemble.



Model Framework

Figure 0.1: The active anomaly detection framework used.

We want to leverage the fact that different ensemble members have different characteristics which makes them more or less suitable to make predictions for certain instances. We weight the ensemble members using a neural network, taking the instance as input to weight the ensemble members in accordance with their capacity to make a prediction for certain instances. The loss to train the neural network is composed of two parts, the first a supervised part lossAAD, using the labels provided by a human expert, and a second part lossprior which places a uniform prior on the ensemble members. When new labels are provided the meta learner is updated so as not to misclassify any of the labeled instances. The framework was tested on the Yahoo Webscope benchmark dataset consisting of four different types of time series. The proposed framework had an AUC of 0.9088, 0.9787, 0.8998 and 0.8123 for the four datasets corresponding to the second highest AUC for 2 data sets and third highest for the remaining 2 data sets out of the models that were used for comparison.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title	*ISBN*
0280-5316	

Language	*Number of pages*	*Recipient's notes*
English	1-58	
Security classification		

http://www.control.lth.se/publications/