The Potential of User Behavioural Data for a Preventive Exception Handling Mechanism

Niklas Eschner[1] and Paul Sutterer[1]

[1] Technical University of Munich, Department of Informatics, Munich, Germany
niklas.eschner@tum.de, paul.sutterer@tum.de

Abstract. Current exception handling mechanisms in Java and .NET frameworks are limited to handle exceptions once they occur. Non-existing or failing exception handling can cause several problems. This study contributes to the development of a preventive exception handling mechanism where the occurrence of an exception is predicted and proactively prevented. A database of 1.5 million user sessions recorded on a stand-alone C# software application was used in this study. We evaluate the potential of user behavioural data to predict the occurrence of exceptions. Five classification methods are benchmarked with stratified 10-fold-cross validation. K-Nearest Neighbour shows superiority to the other methods with an average Matthews correlation coefficient of 0.66. Complementing hardware and system environmental data by user behavioural data improves the prediction quality of exceptions. Our study provides evidence for the ability to predict exceptions regardless of their type. It is a step towards a self-learning mechanism that improves software reliability post-release.

Keywords: Preventive Exception Handling Mechanism, Software Defect Prediction, Exception Prediction, Software Reliability.

1 Introduction

Software is omnipresent in modern society which highly relies on its proper operation. Especially when software is responsible for human lives or has high impact on business decisions, software is expected to operate reliable and flawless. Sometimes this expectation is dismissed as there is no guarantee for a *software defect* free application. Therefore, intelligent technical mechanisms that increase software's maturity, fault tolerance, and recoverability are desirable.

In contrast to material goods, the intangible nature of software enables the occurrence of software defects [1]. Arora et al. analyse the software companies' trade-off between selling software with defects and investments in patching. They conclude that in large markets, software companies release software early, despite unfinished software testing. Reasons include high time-to market pressure, a fixed cost structure to remedy software defects, and marginal costs that are effectively zero for fixing software defects per product post release [1].

Multikonferenz Wirtschaftsinformatik 2018,
March 06-09, 2018, Lüneburg, Germany

To prevent a software from crashing due to defects, the exception handling mechanism was invented in 1975 [2]. This paradigm is reactive, defining actions which are triggered when a defect occurs. These actions aim at bringing the software back into a known state and prevent it from crashing. Cabral and Marques empirically evaluate the mechanism for Java and .NET in [3]. Their research reveals two shortcomings. First, software developers misuse the exception handling mechanism. They either throw generic *exceptions*, leave the exception handling code empty, or use the mechanism for writing log-files. This makes error handling impossible and therefore disables its use as an error handling tool. Second, ever since its invention in 1975, the exception handling mechanism's design has not changed much [3].

The exception handling mechanism is subject to current research which addresses the presented shortcomings. The latest developments in machine learning algorithms and the decreasing costs in computational power enable a paradigm shift. Lourenço et al. propose a preventive exception handling mechanism (PreX) [4,5]. A classifier learning from observed exceptions could make predictions about their occurrence. If an exception is predicted, countermeasures can be initiated to bring the software into a state that prevents it from happening. The best classifier's performance ranges from 3 to 5 percent false positive rate (FPR) and false negative rate (FNR) [4].

Besides PreX, the research field of software defect prediction is close to our study. The majority of the studies in this research field use a classification algorithm to predict whether software modules have defects [6]. Lessmann et al. offer a benchmark framework to compare different classification methods for this task [7]. They propose the area under the receiver operating characteristics curve (AUC) as an evaluation metric for comparative studies in this research field. Open issues in software defect prediction are listed in [8].

This study addresses the two research fields: exception handling mechanism and software defect prediction. In contrast to software defect prediction, our classifier predicts the occurrence of exceptions in a post release live system rather than the occurrence of defects in the software code. While Lourenço et al. [4,5] focus on predicting null pointer exceptions only, our classifier predicts all types of exceptions. Furthermore, we evaluate the ability of PreX with recorded data on a C# stand-alone software application. Besides *hardware and system environmental data*, we include user behavioural data for the prediction of exceptions. We aim to answer three research questions with this study:

Research Question 1 (RQ1): What potential does user behavioural data have for the prediction of exceptions in user sessions? Software defects and exceptions are usually predicted with hardware and system environmental data. However, the user's actions trigger exceptions.

Research Question 2 (RQ2): How important is the beginning of a user session for the occurrence of an exception during the session? The first events in a user session should reveal which task a user pursues. We assume that exceptions are related to certain tasks. Therefore, the beginning of a user session should already reveal a later occurring exception.

Research Question 3 (RQ3): What potential does the modelling of user behaviour have for predicting exceptions in a live system? For the development of PreX, it is not

sufficient to answer only RQ2, since the exact moment of an occurring exception is unknown. However, for PreX, it is essential to predict that exact moment.

The remainder of this article is structured as follows. In Chapter 2, we provide required definitions and a description of the applied machine learning and evaluation methods. In Chapter 3, we describe the data set design and its features. In Chapter 4, the machine learning methods are applied to two data sets and their performance is evaluated. We conclude with a summary of this study's contribution in Chapter 5, as well as its limitations, and provide some directions for future research.

2 Theoretical Background

In the following, we differentiate software defects from exceptions and provide the reader with the concept of PreX. Next, we present the chosen classifiers and justify the selected evaluation metrics.

2.1 Software Defects and Exceptions

The term software defect describes unintended software behaviour. It includes *software failures* and *software faults* [9]. A software failure describes the software's inability to produce the user's expected result [10]. The cause for a software failure is a software fault [9]. Only a developer can eliminate a software fault in the code before compiling the application. Some software faults remain forever undiscovered in software.

In order to define the term exception, Flaviu distinguishes three different software states that an executed software method can result in [11]. These are: standard domain (SD), anticipated exceptional domain (AED), and unanticipated exceptional domain (UED). To control a software system, it is necessary to know the state of all its variables and objects. If this condition is violated, the software is in an unintended state and terminates. Usually, a called software method terminates at the SD. When a method is invoked and an exception appears, the application terminates in AED or UED. Therefore, Flaviu defines an exception as unintended software state [11]. Flaviu's study is the foundation for current exception handling mechanisms. We use the term exception to refer to an unintended software state [11].

2.2 Preventive Exception Handling Mechanism

The current exception handling mechanisms in Java and .NET follow a `try-catch-finally` syntax. Lourenço et al. [4,5] adopt and extend this syntax as shown in the following.

```
1 try (<prediction_context>) {
2 // ... Prediction Block.
3 // Exceptions can be caught and alarms can be triggered
4 } prevent ( <exception_name>, <information_object> ) {
5 // ... Prevention Block.
6 // Execution follows the resumption model.
7 } catch ( ... ) {
8 ... Exception Handling code
9 }
```

The `try` block turns into a prediction block. A classifier predicts the probability for an upcoming exception. When a certain probability threshold is exceeded, an alarm is triggered and the `prevent` block is executed. This block is responsible for preventing the exception from happening by executing the resumption model. When the classifier fails to predict an exception, no alarm is triggered and the traditional exception handling mechanism in the `catch` block is executed.

Lourenço et al.'s aim is to offer developers a sophisticated and powerful tool to develop and apply new kinds of exception handling strategies. The developer is responsible to define what kind of exception should be predicted and what suitable countermeasures for this specific kind of exception are. Lourenço et al. simulate a python client-server application [4,5]. For this environment, they predict connection timeouts between the client and the server. When the probability for such a timeout is high, they slow down the clients execution rate to prevent the connection timeout[4,5].

While Lourenço et al. focus on predicting connection timeouts in this specific environment, we focus on predicting any kind of exception types in an empirical data set. Since the design of the resumption model always is an individual decision of the developer, we do not address its design further.

2.3 Classification Methods and Evaluation Metric

Table 1 displays the seven most frequently used classification methods in software defect prediction [6] and indicates which of these are applied in this study's benchmark.

Table 1. Application of classification methods in the field of software defect prediction

Classification Method	*Number of Studies [6]*	*Applied in Current Study*
Naïve Bayes	14	Yes
Decision Tree	11	Yes
Neural Network	9	No
Random Forest	6	Yes
Logistic Regression	5	Yes
K-Nearest Neighbour	4	Yes
Support Vector Machine	4	No

We do not apply Neural Networks due to the high resource requirements regarding memory and time. Support Vector Machines are excluded also since they require parameter optimisation which is not the focus of this study.
Lessmann et al.'s framework [7] evaluates the classification methods based on the area under the curve (AUC) metric. It describes the area under the receiver operating characteristics (ROC) curve which is defined as

$$ROC = \frac{True\ Postive\ Rate\ (TPR)}{False\ Positive\ Rate\ (FPR)} \quad (1)$$

The AUC value can be interpreted as the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance [12]. In recent years, there has been some criticism about the metric. In particular for imbalanced data, it can give a misleading impression about a model's performance by over-estimating a classifiers performance [13]. Due to these shortcomings, we use Matthews correlation coefficient (MCC) [14] as our main evaluation metric. It is defined as follows

$$MCC = \frac{TP \times TN - \quad \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (2)$$

with TP=True Positive, TN=True Negative, FP=False Positive, and FN=False Negative [15]. Since it involves all values of the confusion matrix and considers accuracies and error rates of both classes, it is less biased by an imbalanced data set and considered the best singular assessment metric by some authors [16–18]. It ranges from -1 to 1, with -1 indicating the worst possible, 1 indicating a perfect, and 0 indicating a random prediction performance [17]. Since the MCC is a contingency matrix method of calculating the Person correlation coefficient [15], MCC values can be interpreted as such. Following Evans, a Pearson correlation coefficient between 0.40 and 0.59 is accounted as "moderate", as "strong" between 0.60 and 0.79, and as "very strong" between 0.80 and 1.0 [19]. For comparison reasons, we report both, MCC and AUC.

3 Data Sets Design

The data originates from a software which is primarily used for product portfolio analysis and variant management in the manufacturing industry. The risk of a software crash only results in a software restart and therefore bad user experience. The number of recorded attributes in many user sessions plus the tracking of exceptions, makes it a qualified software application for our study. A database of 1.5 million recorded user sessions was provided for this study. It covers customers' and developers' sessions of two different software products for a period of 35 months. During this period, the recording of sessions has been adjusted several times. We can only ensure the same level of data quality for sessions recorded in the past 12 months. We derive two data sets to answer the research questions. Both contain only data of customers' sessions and the first software product. For technical reasons, they differ in the number of covered months. The two data sets contain 12 hardware and system environmental features, hereinafter referred to as *system features* that are displayed in Table 2. The

system features contain information, which is available at the beginning of a user session.

Table 2. Overview of system features

System Feature Category	*Number of Features*	*Example*
Hardware Environment	6	Processor Architecture
Software Environment	5	Operating System Version
Software Product	1	Software Version

In addition to system features, both data sets contain triggered *session events*. These are individual events in a session triggered by the user. A click on the application's start menu is one example for such an event. For this study, we observe 415 unique session events, which cover a click path through the software application. This represents the user's behaviour in a session. Each session event is represented by a unique identifier. Not all 415 session events are necessarily triggered in one single user session. Figure 1 illustrates an exemplary user session with $n \in \mathbb{N}$ session events and $m \in \mathbb{N}$ exceptions.

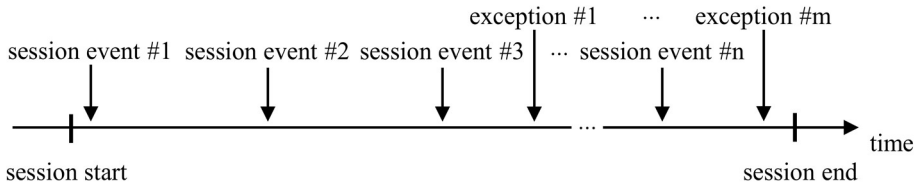

Figure 1. Model of a single user session with session events and exceptions

For both data sets, we model session events as sparse matrices representing the chronological order of events in a user session.

3.1 Modelling of Individual User Behaviour in Data Set I

We design Data Set I to answer RQ1 and RQ2. It covers a time frame of 12 months. A single user session is represented as exactly one instance. The session events only represent sequences of the beginning of a user session with a length of $n \in \{3, 6, 9\}$. For example, assume one single user session consists of the following five session events $(376, 379, 387, 295, 296)$. In case of $n = 3$, only the first three session events (376, 379, 387) are considered.

It is possible to observe more than one exception during a user session. This is irrelevant for the dependent variable, since it only indicates whether an exception will occur at some point during an entire user session.

3.2 Modelling of Individual User Behaviour in Data Set II

We design Data Set II to answer RQ1 and RQ3. It covers a time frame of 3 months. In contrast to Data Set I, a single user session is now represented by several instances. We split all observed session events during one single user session into sequences of length $n \in \{3, 6, 9\}$. For example, assume one single user session consists of the same five session events $(376, 379, 387, 295, 296)$. In case of $n = 3$, three session events sequences $(376, 379, 387)$, $(379, 387, 295)$, and $(387, 295, 296)$ are created. This results in three instances representing the user session.

In case of more than one observed exception in a user session, all occurring exceptions are considered. The dependent variable indicates whether an exception occurs immediately after the session events sequence. In consequence of modelling user behaviour as session events sequences, the system features are replicated and added to each session events sequence.

3.3 Distribution of Data Sets' Dependent Variable

Figure 3 displays both data sets' number of instances and the distribution of the dependent variable.

Table 3. Overview of Data Set I and II

Data	*Number of Instances*	*Percentage With no Exception*	*Percentage With Exception*
data_set_I_$n = \{3, 6, 9\}$	49,085	91.27	8.73
data_set_II_$n = 3$	73,998	99.46	0.54
data_set_II_$n = 6$	73,294	99.46	0.54
data_set_II_$n = 9$	72,352	99.45	0.55

The instances of Data Set I represent 49,085 user sessions covering session events from the beginning of a session only. The number of instances is independent of n. One or more exceptions occur in 8.73 percent of these user sessions.

Data Set II only covers 316 user sessions each with at least one exception. The number of instances depends on n since each user session is divided into session events sequences of length $n \in \{3, 6, 9\}$. This results in three data sets with a slightly different total number of instances. The number of instances is equal to the number of created session events sequences. Compared to Data Set I it is even more imbalanced.

4 Evaluation of Classification Methods

Five classification methods are benchmarked on both data sets using R (version 3.4.1) and the mlr package [20,21]. The benchmarks are computed on cloud computing services, requiring memory in the range of 10 to 72 GB. Stratified 10-fold-cross-validation is chosen for both benchmarks since it is a generally accepted validation method [7,22,23]. At first, only system features are used for both benchmarks.

Afterwards, session events sequences with $n \in \{3, 6, 9\}$ are added to the system features and the benchmarks are performed again.

4.1 Benchmark of Classification Methods with Data Set I

In Data Set I, only the first session events of a session are considered. The classification task at hand is to predict whether at some point during the session an exception will occur. The benchmarking results of all five classification methods on Data Set I are displayed in Table 4.

Table 4. Benchmark I, average MCC (average AUC)

Classification Method	*System Features*	*System Features + Session Events*		
		n = 3	*n = 6*	*n = 9*
Naïve Bayes	0.04 (0.69)	0.11 (0.70)	0.04 (0.70)	0.02 (0.60)
XGBoost	0.04 (0.72)	0.27 (0.77)	0.41 (0.75)	0.51 (0.72)
Random Forest	0.00 (0.53)	0.21 (0.66)	0.58 (0.80)	0.69 (0.86)
Logistic Regression	0.04 (0.59)	0.23 (0.60)	0.48 (0.71)	0.59 (0.78)
K-Nearest Neighbour	0.16 (0.73)	0.33 (0.84)	0.63 (0.92)	0.78 (0.96)

Regardless of the classifier and based on system features only, the prediction whether an exception occurs in a user session is random. Except for the Naïve Bayes classifier, the prediction performances improve, when session events from the session's beginning are added. The more session events at the session's beginning are included, the better the prediction performance of all classification methods is, except for Naïve Bayes. K-Nearest Neighbour based on system features and $n = 9$ session events predicts the occurrence of an exception best, leading to an average MCC of 0.78, which is considered strong. It indicates a very high degree of correlation between the classifiers prediction and the actual outcome. The results of XGBoost are further evidence for the unsuitability of the AUC as evaluation metric. For $n = \{3, 6, 9\}$ the average AUC is almost the same while the average MCC shows considerable differences.

4.2 Benchmark of Classification Methods with Data Set II

In Data Set II, all possible sequences of size $n \in \{3, 6, 9\}$ in a session are considered. The classification task at hand is to predict whether an exception occurs immediately after the corresponding sequence. The Benchmarking results of all five classification methods on Data Set II are displayed in Table 5.

Table 5. Benchmark II, average MCC (average AUC)

Classification Method	*System Features*	*System Features + Session Events Sequences*		
		n = 3	*n = 6*	*n = 9*
Naïve Bayes	0.03 (0.73)	NaN (0.58)	NaN (0.50)	NaN (0.50)
XGBoost	NaN (0.62)	0.36 (0.69)	0.44 (0.69)	0.53 (0.71)
Random Forest	0.00 (0.52)	0.12 (0.73)	0.17 (0.88)	0.27 (0.92)
Logistc Regression	0.05 (0.75)	0.14 (0.83)	0.26 (0.81)	0.42 (0.84)
K-Nearest Neighbour	0.03 (0.57)	0.40 (0.83)	0.60 (0.91)	0.66 (0.88)

Again, regardless of the classifier and based on system features only, the prediction whether an exception occurs in a user's session is random. For XGBoost, there is no MCC available, since FP = TP = 0. As observed in Benchmark I, the prediction performance improves when the actual session events sequences are considered in addition to the system features. The larger the session events sequences, the better the prediction performance is. K-Nearest Neighbour, based on system features and session events sequences of length n = 9, predicts the occurrence of an exception best, leading to an average MCC of 0.66, which is considered strong.

4.3 Discussion of Results

In both benchmarks, the Naïve Bayes classifier shows poor prediction performance. The reason is that session events data violates the Naïve Bayes assumption that all features should be independent. Session events are clearly dependent on previous session events, since they cover the user's click tree.

Both benchmarks differ in the classification task. Benchmark I's prediction performance is slightly better than Benchmark II's. One reason for that difference might be the degree of class imbalance which is worse in Data Set II. With the obtained benchmark results, we can answer the research questions.

RQ1: What potential do user behavioural data have for the prediction of exceptions in user sessions? Both Benchmark I and II show a high potential of behavioural data, here session events, for the prediction of exceptions in user sessions. Complementing system features by behavioural data improves all classifiers' prediction performance, except for Naïve Bayes. We conclude that the reason for this performance improvement is that the user's actions are responsible for triggering exceptions. System features by itself do not contain any information about which software methods are used. This is evidence to include both behavioural data and system features for PreX.

RQ2: How important is the beginning of a user session for the occurrence of a software exception during the session? Benchmark I clearly demonstrates that the beginning of a user session contains information about the later occurrence of exceptions. The first session events in a user session indicate which task a user pursues and this task is assumed to be correlated with the later occurrence of an exception. The larger the sequence of leading session events, the better the prediction performance of all classifiers, except for Naïve Bayes, is.

RQ3: What potential does the modelling of user behaviour have for predicting exceptions in a live system? For the development of PreX, it is essential to predict the exact moment of an occurring exception. Otherwise, it is very difficult to initiate countermeasures to prevent the exception. Modelling session events in sequences of a given length mitigates the time problem of Data Set I. The results of Benchmark II show that modelling user behaviour by session events can lead to a substantial improvement on the prediction performance of exceptions. For all classifiers except Naïve Bayes, session events sequences of length $n = 9$ result in better prediction performance than sequences of length $n = \{3, 6\}$.

This study's results measured with the AUC metric are comparable to Lessmann et al. [7], even though we predict exceptions and Lessmann et al. predict software defects. The results of Benchmark I range between an average AUC of 0.53 and 0.96, while the results of Benchmark II range between 0.50 and 0.91. Lessmann et al. report results between an average AUC of 0.50 and 0.97 in their benchmark study [7]. Overall the benchmark results measured with the MCC show a positive correlation between the classifiers prediction and the actual outcome. K-Nearest Neighbour is identified as the single superior classifier.

5 Conclusion

A self-learning mechanism which improves software reliability post release is of interest in research and practise. PreX is at the starting point of its development and is a new use case for applied machine learning. While PreX remains invisible for software users, they are leveraged as large-scale testing resources who unconsciously contribute to the software's reliability. PreX is a generic mechanism which can be used in any software application regardless of its use case. Different software has different requirements concerning the importance of minimising FP and FN predictions. Therefore, we apply the MCC metric which balances those. When software developers implement PreX they can adjust the costs of FP and FN predictions in line with the application's requirements.

We contribute to PreX's development in three areas. First, this study is evidence for the ability to predict exceptions regardless of their type based on a C# stand-alone software application. Second, the choice of data set design has a substantial impact on the classifier's prediction performance. Third, for the classification task of exception prediction, the choice of classifier also has a substantial impact on the prediction performance. Clearly, the Naïve Bayes classifiers assumptions are violated, making it unsuitable for behavioural data. In both benchmarks, K-Nearest Neighbour shows superiority to the other classification methods. However, the choice of classifier is closely related to the data set design and hence cannot be generalised.

The limitations of this study are a domain specific data set, the application of only five classification methods, and some data set design decisions. Especially, Data Set II is restricted to 316 user sessions. However, more user sessions with exceptions are desirable. Another limitation is the decision to model session events sequences with fixed length of $n = \{3, 6, 9\}$.

This study's database offers opportunities for further research in the four areas of sequence analysis, modelling of session events, prediction problems and classification runtime.

First, future work should analyse the relationship between session events and exceptions in further detail. One approach might be to identify single session events likely to cause exceptions, another approach is to apply sequence analysis to discover sequences which appear frequently and cause exceptions.

Second, further modelling of user behaviour should be analysed. A third model could combine the approaches of Data Set I and II. Session events sequences always include all session events from the session's start until the last observed user event. When a new user event is recorded, a new instance is created with a session events sequence covering all user events up to this point.

Third, the exception location and exception type are of interest for PreX's resumption model. Therefore, future work should predict the exception location as typical for the research field of software defect prediction and the exception type. Nonetheless, the resumption model itself should be topic of future work.

Fourth, the runtime of classifying a user session is essential for implementing PreX. To ensure an unimpaired user experience, future work should study how much time each classifier needs for the predictions in a session.

In summary, PreX aims to improve software reliability based on the occurrence of exceptions only. Exceptions are the effect of software defects. Hence, PreX does not correct any software defects, but rather is a concept to improve software reliability by controlling the effects of the actual cause. While PreX is no substitute for proper software testing, it is a powerful concept to mitigate the effects of software defects.

6 Acknowledgements

This study's database has been sponsored by a software start-up company. We thank them very much for sharing their data with us and sponsoring computational resources. Furthermore, we would like to thank both reviewers for their valuable comments.

References

1. Arora, A., Caulkins, J.P., Telang, R.: Research Note—Sell First, Fix Later. Impact of Patching on Software Quality. Management Science 52, 465–471 (2006)
2. Goodenough, J.B.: Exception Handling: Issues and a Proposed Notation. Communications of the ACM 18, 683–696 (1975)
3. Cabral, B., Marques, P. (eds.): Exception Handling: A Field Study in Java and .Net. Springer (2007)
4. Lourenço, J.R.: PreX – Preventive Exception Handling. Portugal (2016)
5. Lourenço, J.R., Cabral, B., Bernardino, J.: A Predictive Model for Exception Handling. In: Rocha, Á., Correia, A.M., Adeli, H., Reis, L.P., Mendonça Teixeira, M. (eds.) New Advances in Information Systems and Technologies, pp. 767–776. Springer International Publishing, Cham (2016)

6. Wahono, R.S.: A Systematic Literature Review of Software Defect Prediction. Research Trends, Datasets, Methods and Frameworks. Journal of Software Engineering 1, 1–16 (2015)
7. Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. IEEE Transactions on Software Engineering 34, 485–496 (2008)
8. Arora, I., Tetarwal, V., Saha, A.: Open Issues in Software Defect Prediction. Procedia Computer Science 46, 906–912 (2015)
9. Lyu, M.R. (ed.): Handbook of Software Reliability Engineering. IEEE Computer Society Pr, Los Alamitos Calif. (1996)
10. Endres, A. and Rombach, H.D.: A Handbook of Software and Systems Engineering. Empirical Observations, Laws and Theories. Pearson Addison Wesley, Harlow, England, Munich (2003)
11. Flaviu, C.: Exception Handling and Software Fault Tolerance. IEEE Transactions on Computers Vol. C-31, 531–540 (1982)
12. Fawcett, T.: An Introduction to ROC analysis. Pattern Recognition Letters 27, 861–874 (2006)
13. Briggs, W.M., Zaretzki, R.: The Skill Plot. A Graphical Technique for Evaluating Continuous Diagnostic Tests. Biometrics 64, 250–256 (2008)
14. Matthews, B.W.: Comparison of the Predicted and Observed Secondary Structure of T4 Phage lysozyme. Biochimica et Biophysica Acta (BBA) - Protein Structure 405, 442–451 (1975)
15. Powers, D.M.: Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. Journal of Machine Learning, 37–63 (2011)
16. Ding, Z.: Diversified Ensemble Classifiers for Highly Imbalanced Data Learning and Their Application in Bioinformatics. Atlanta, Georgia (2011)
17. Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A.F., Nielsen, H.: Assessing the Accuracy of Prediction Algorithms for Classification. An Overview. Bioinformatics Review 16, 412–424 (2000)
18. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal Classifier for Imbalanced Data Using Matthews Correlation Coefficient Metric. PloS one 12 (2017)
19. Evans, J.D.: Straightforward Statistics for the Behavioral Sciences. Brooks/Cole, Pacific Grove, Calif. (1996)
20. Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Jones, Z., Casalicchio, G., Gallo, M., Bossek, J. and Studerus, E., et al.: R Package Machine Learning in R, https://cran.r-project.org/web/packages/mlr/mlr.pdf
21. Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G., Jones, Z.M.: R Package mlr. Machine Learning in R. Journal of Machine Learning Research 17, 1–5 (2016)
22. Salzberg, S.L.: On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach. Data Mining and Knowledge Discovery 1, 317–328 (1997)
23. Witten, I.H., Frank, E. and Hall, M.A.: Data Mining. Practical Machine Learning Tools and Techniques. Elsevier, Amsterdam (2011)