Consistency Metrics for LR-Systems

A Comparative Analysis

MSc Applied Mathematics: Thesis Project

Hester Klomp

Delft University of Technology

Consistency Metrics for LR-Systems

A Comparative Analysis

by

Hester Klomp

Student Name	Student Number
Hester Klomp	5850797

Instructor TU Delft:	J. Söhl
Instructor NFI:	L. van der Ham
Project Duration:	January, 2024 - October, 2024
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Sciences, Delft

Cover:	The Public Health Image Library from the Centers for Disease Control and Prevention (CDC)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

Preface

Working on this thesis has been both challenging and rewarding. During the better part of this year, I have spent day in day out thinking about likelihood ratios and consistency metrics. To have it come to an end is bittersweet: while I am proud of the work that I have delivered and extremely satisfied with everything that I have learned, I am also sad to say goodbye to it and with that conclude my Master's degree.

Having never worked in forensic science before, a whole new world has opened up to me and I have learned so much over the past months. Writing a master's thesis is never easy, but doing it about such an interesting subject and in such a stimulating environment certainly helps. What helps even more is having supportive people around you that you can fall back on if you need to.

I would first like to thank my supervisors, Jakob Söhl from TU Delft and Leen van der Ham from the Netherlands Forensic Institute. All of our meetings and chats have helped me immensely. It is a privilege to be able to work with such intelligent and helpful people, and I am very grateful for this opportunity.

Secondly, I would like to thank my boyfriend Gonçalo for always encouraging me and proofreading everything even though he has no idea what it is all about. I would also like to thank my friend Rosalie for being a shoulder to cry on, not only during the thesis but during this whole degree. There is no way I would have made it through without you.

Lastly I would like to thank my family, and in particular my parents, for their continuous support during the highs and the lows of everything.

Thank you all for being part of this journey with me.

Hester Klomp
Delft, September 2024

Abstract

An important tool in forensic science is the likelihood ratio (LR), which quantifies the strength of evidence. It does so by comparing the probabilities of the evidence under two mutually exclusive hypotheses, the prosecution hypothesis H_p and the defense hypothesis H_d. However, if the underlying probability model to determine these probabilities is not correct, this can lead to misleading conclusions, for example biases towards one of the hypotheses. The ability for an LR-system to produce LR-values that reflect the true probabilities of the evidence under the hypotheses is called 'consistency'. Ensuring the consistency of LR-systems is necessary to prevent biases and inaccuracies.

Several methods to evaluate consistency of LR-systems have been developed over the past decade, but there has been a lack of thorough comparisons to identify which one is the most effective with real case data. In this thesis, the aim is to fill this gap by developing and optimizing the existing methods, and comparing them to one another. An in-depth comparative analysis will be conducted of various existing metrics, as well as some newly introduced metrics, to evaluate the consistency of LR-systems.

This study evaluates the consistency metrics C_{llr}^{cal} and devPAV. The individual metrics are optimized before comparing them to each other. A third metric is introduced, which is named Fid. This metric is based on advanced calibration techniques. It is compared to the previous two metrics to see which one performs best on different datasets. This performance is evaluated based on the metrics' abilities to distinguish between consistent and inconsistent LR-systems, their reliability in terms of their output and their sensitivity to dataset size. To measure this, several different datasets are used.

The results show that C_{llr}^{cal} outperforms the other metrics in distinguishing between consistent and inconsistent LR-systems. However, it falls short in terms of reliability, as it fails to assign the same values to different LR-systems that are all consistent. On the other hand, devPAV demonstrates high reliability, showing both reliability across different datasets and across different dataset sizes. The Fid metric shows similar performance to devPAV, but has the disadvantage of not working on smaller datasets. Therefore, as a metric it might not be preferred, although the method itself definitely shows interesting insights into the consistency of LR-systems.

These findings improve the tools we have for forensic evidence interpretation, helping to make forensic practices more accurate and reliable. By identifying the best metric for consistency, this research helps to make the criminal justice system fairer and more precise.

Contents

Preface **i**

Abstract **ii**

1 Introduction **1**

2 Literature Review **3**
- 2.1 Likelihood Ratios . . . 3
  - 2.1.1 Common source versus specific source models . . . 4
  - 2.1.2 Feature-based versus score-based methods . . . 6
- 2.2 Properties of LR-systems . . . 6
  - 2.2.1 Consistency of LR-systems . . . 6
  - 2.2.2 Discrimination of LR-systems . . . 9
  - 2.2.3 Calibration methods for LR-systems . . . 9
  - 2.2.4 The PAV Algorithm . . . 9
  - 2.2.5 Kernel Density Estimation . . . 11
- 2.3 Metrics to measure consistency . . . 11
  - 2.3.1 Moments metric . . . 12
  - 2.3.2 Probability of misleading evidence . . . 13
  - 2.3.3 C_{llr}^{cal} metric . . . 14
  - 2.3.4 devPAV metric . . . 15
  - 2.3.5 Calibration discrepancy metric . . . 16
- 2.4 Building LR-systems . . . 18
  - 2.4.1 The data . . . 18
  - 2.4.2 The scores . . . 19
  - 2.4.3 The LR values . . . 19
  - 2.4.4 Selection and validation . . . 19
- 2.5 Comparing the metrics . . . 20

3 Methods **21**
- 3.1 Step 1: Optimization of the metrics . . . 21
  - 3.1.1 Optimization of C_{llr}^{cal} . . . 22
  - 3.1.2 Optimization of devPAV . . . 22
  - 3.1.3 Optimization of fiducial metric . . . 24
- 3.2 Step 2: Comparing the metrics . . . 25
  - 3.2.1 Datasets . . . 25
  - 3.2.2 Generate same-source data . . . 26
  - 3.2.3 Test metrics on new data . . . 27

4 Results **28**
- 4.1 Results of optimization of the metrics . . . 28
  - 4.1.1 Optimization of C_{llr}^{cal} . . . 30
  - 4.1.2 Optimization of devPAV . . . 33
  - 4.1.3 Optimization of fiducial metric . . . 37
- 4.2 Results of comparing the metrics . . . 40
  - 4.2.1 Consistent versus inconsistent data . . . 40
  - 4.2.2 Reliability of the metrics across datasets . . . 49
  - 4.2.3 Reliability of the metrics across dataset sizes . . . 53

5 Conclusions and Discussion **55**

References **57**

A Source Code **59**

1 Introduction

Forensic science plays an important role in the criminal justice system by providing scientific analyses that support legal decisions. One of the key challenges in forensic science is the evaluation and interpretation of evidence to determine its relevance and strength. A commonly used statistical tool to determine this strength of evidence is the likelihood ratio (LR).

A likelihood ratio is a measure used to quantify the strength of evidence by comparing the probability of observing the evidence under two competing hypotheses: the prosecution hypothesis (H_p) and the defense hypothesis (H_d) [13]. These evidence assess whether or not the evidence comes from a given source (for example, a suspect). The LR is calculated as the ratio of these probabilities. It provides a numerical value that indicates how much more likely the evidence is to be observed under one hypothesis compared to the other. When determined correctly, the LR gives an objective assessment of evidence strength, which can be used in legal decision-making.

Likelihood ratios are used in various forensic disciplines, such as DNA analysis, fingerprint examination, and glass fragment analysis [8], [11], [10]. In these areas, we speak of LR-systems. These LR-systems allow for the automatic computation of LRs based on mathematical models trained on relevant data. By automating this process, forensic scientists can provide more consistent and objective evaluations of evidence, which is of great importance in supporting the judicial process. Moreover, it becomes more reproducible and significantly quicker to calculate LRs, enhancing the efficiency of forensic evaluations. However, a potential downside of using LR-systems is that the model could overlook certain aspects of the evidence, and it may not account for the unique details of each individual case.

An important aspect of using LR-systems is ensuring their consistency, i.e., their ability to produce reliable and accurate LRs that reflect the true probabilities of the evidence under the different hypotheses [9]. Inconsistent LR-values can lead to misleading conclusions, possibly affecting the outcomes of legal cases. For instance, if an LR-system is not consistent, it might overestimate the strength of evidence in favor of the prosecution or the defense, leading to potential biases in the legal decision-making process.

In the past decade, different methods and metrics have been developed to evaluate the consistency of LR-systems. These methods range from simple statistical checks to complex calibration techniques. Some of these metrics have already been compared in [27]. This comparison gives us a lot of initial insight into the performance of the metrics and their reliability. Many of the insights and methods from this paper will be applied in this thesis. However, a normal distribution is assumed for the LR-data, which is usually not in line with reality. There has not yet been a thorough comparison to see which one of these methods work best with real forensic LR-data. This lack of comprehensive analysis means we still do not have a clear understanding of which metrics are the most reliable across various types of evidence and forensic contexts.

This thesis aims to fill this gap by evaluating and comparing already existing metrics, as well as a newly introduced one, used to measure the consistency of LR-systems. By developing and optimizing these metrics, this study hopes to provide a robust framework for evaluating the reliability of assessing forensic evidence. This, in turn, will help increase the fairness and accuracy of the criminal justice system. A reliable consistency metric ensures that forensic evidence is evaluated correctly, minimizing the risk of wrongful convictions or acquittals based on misinterpreted evidence.

The structure of this thesis is as follows: in Chapter 2, the relevant existing literature on LR-systems and consistency metrics is reviewed. The theoretical framework is outlined, and all the important definitions are introduced and explained. The chapter provides a solid foundation for understanding the various approaches to measuring consistency and their applications in forensic science. Chapter 3 describes the methodology used in this study for the comparison of the metrics, including the construction and optimization of the metrics. The chapter describes in detail the experimental setup, data collection, and analytical techniques used to ensure a thorough comparison. Chapter 4 presents the results of the comparative analysis, showcasing the performance of different metrics across various datasets and forensic scenarios. The chapter includes detailed statistical analyses and visualizations. Chapter 5 discusses the findings, implications, and conclusions of this research. It also highlights the practical applications of the developed framework and suggests directions for future research.

This research does not only identify the most effective metrics for evaluating the consistency of LR-systems, but also contributes to the broader field of forensic science by improving the tools available for evidence interpretation. By providing a clear and comparative analysis of existing metrics, this thesis lays the groundwork for more reliable and accurate methods in forensic science.

2 Literature Review

2.1. Likelihood Ratios

In criminal cases, we often encounter situations where some evidence, denoted as E, is available, but its strength or direction is not immediately clear. This is where forensic experts come into play: to help quantify the strength of the evidence. A common tool for this purpose is the likelihood ratio (LR), first introduced in [13], which is used to measure the strength of the evidence. A few key components are required to compute the LR.

Firstly, two competing hypotheses are commonly used: the prosecution hypothesis H_p, and the defense hypothesis H_d. The hypotheses depend on the question that needs to be answered. The question, in turn, usually depends on the evidence and background information available. Sometimes there is a suspect. Other times, there is no suspect yet, but only two traces, and the aim is to compare the traces to each other. Typically, the prosecution hypothesis asserts that the source of the evidence E and the suspect are the same person, or in the case where there is no suspect, that two traces come from the same source. For example, the trace found on the crime scene belongs to the suspect, the two fingerprints are from the same person, etc. The defense hypothesis on the other hand, asserts that the evidence does not belong to the suspect, or that two traces come from different sources. One can use different formulations of hypotheses that might influence the values of the LRs in different ways. The hypotheses do not need to be defined on source-level, as we did now. They can also be on activity-level, stating that a certain activity has taken place, or on subject-level, emphasizing on the people involved. It is important to be very specific and clear about what exactly the hypotheses are. More on this in Section 2.1.1.

Normally, there is some background information available, which is referred to as I. This can be anything varying from other evidence to background information on the subject or crime.

In a criminal case, it is interesting to know the following ratio:

$$\frac{\mathbb{P}(H_p|E,I)}{\mathbb{P}(H_d|E,I)}. \tag{2.1}$$

This ratio says something about the likeliness of H_p being true compared to H_d being true, given the evidence and the context of the case. However, it is often not possible to directly determine the value of this ratio.

According to Bayes's theorem, as discussed in the textbook [4], Equation (2.1) provides a factorization of the posterior odds:

$$\frac{\mathbb{P}(H_p|E,I)}{\mathbb{P}(H_d|E,I)} = \frac{\mathbb{P}(E|H_p,I)}{\mathbb{P}(E|H_d,I)} \times \frac{\mathbb{P}(H_p|I)}{\mathbb{P}(H_d|I)}. \tag{2.2}$$

Normally, the notation of the I is omitted and Equation (2.2) simplifies to

$$\frac{\mathbb{P}(H_p|E)}{\mathbb{P}(H_d|E)} = \frac{\mathbb{P}(E|H_p)}{\mathbb{P}(E|H_d)} \times \frac{\mathbb{P}(H_p)}{\mathbb{P}(H_d)}. \tag{2.3}$$

Now, the middle term is what is referred to as the 'Likelihood Ratio', or LR. In words:

Posterior Odds = LR $\times$ **Prior Odds**.

The job of the forensic examiner is only to determine the LR for a given situation. It is out of his/her field of expertise to determine the prior odds or the posterior odds. This is the responsibility of the legal expert. Sometimes, the LR is erroneously interpreted as the posterior odds. This phenomenon is called the ‘prosecutor’s fallacy’, as explained in more detail in [12].

The LR tells us how much more likely it is to find the evidence E when H_p is true, compared to finding E when H_d is true. In other words, it says something about the direction in which the evidence points, and the strength with which it does so. An LR with a value greater than one indicates that the evidence supports the prosecution hypothesis H_p, whereas an LR of a value smaller than one points towards the defense hypothesis H_d. An LR of exactly one gives neutral information. In this case, the evidence does not support a specific hypothesis. Given the evidence, both scenarios are equally likely.

Recently, ways have been developed to automatically compute LRs based on raw data, using mathematical models. These models have been trained on data relevant to the specific case. Such methodologies are referred to as ‘LR-systems’, as first introduced in [10]. The underlying models define the LR-system. Examples of LR-systems already in use include those for analyzing glass fragments [11], DNA profiles [8], and fingerprints [10].

Ideally, an LR-system outputs a value greater than one every time that H_p is true, and a value smaller than one every time that H_d is true. Realistically, this almost never happens. Sometimes, an LR-system gives a value that is indicative of the wrong hypothesis. When the LR-system outputs a value smaller than one when H_p is true, or a value greater than one when H_d is true, we call this ‘misleading evidence’ [22].

The more extreme the LR, the stronger the evidence is considered to be. Theoretically, an LR can become infinitely large or small. However, it is intuitively clear that when an LR-system is trained on a very small dataset, it is undesirable to express infinite value of evidence. The reason is that an LR-system trained on little data can potentially have very high variance, and not be an accurate representation of reality. When there is more data available, there is more confidence in the LR values. In practice, the LR is usually bounded from above by the number of H_d-true elements in the dataset used to build it, and from below by one divided by the number of H_p-true elements. These bounds are called the empirical lower and upper bounds, or ELUB bounds, and they were first introduced in [28]. So if there are N_d H_d-true elements in the dataset and N_p H_p-true elements, in practice the LR is often bounded as follows:

$$\frac{1}{N_p} \leq \textbf{LR} \leq N_d. \tag{2.4}$$

When the LR-system outputs a value smaller than the lower bound or larger than the upper bound it is just set equal to the corresponding bound.

2.1.1. Common source versus specific source models

Equipped with a foundational understanding of likelihood ratios, a crucial distinction has to be made: common source versus specific source models. This distinction was first introduced in [20].

As previously mentioned, LRs quantify the strength of evidence. The nature of the evidence itself guides our hypotheses. In this section, the common source and specific source questions and models will be introduced. Note that the terms ‘common source’ and ‘specific source’ will be used both to address the type of question as well as the model used to solve it. This can be quite confusing, but it is in line with the literature. It should be clear from the context, or else emphasized, which one of the two options is referred to.

Firstly, it is important to distinguish between two types of questions that need to be addressed, known as the specific source and common source questions. In the specific source question, there is one or more first traces coming from a crime scene, and one or more second traces coming from a suspect. The suspect is the specific source in this case. The question of interest is whether or not the suspect is the donor of this/these trace(s). Samples can be taken both from the trace(s) with the unknown source and the trace(s) from the suspect (the specific source). The results from the analysis of these samples form the evidence E that the LR will be based on. The hypotheses in this case are of the following form [19], [20]:

H_p: The trace originates from the specific source.

H_d: The trace does not originate from the specific source, but from some other source in the alternative source population.

Here, the alternative source population can vary. All possible different sources can be considered, or only those of a specific type. The LR might change depending on how the defense hypothesis is chosen. It is important to be very specific in the phrasing of the hypotheses.

Another type of question is possible, which is referred to as the common source question. Here, two traces have been found, either at the same site or different sites. A question of interest could be if these two traces originate from the same, possibly unknown, source. This is an example of a common source question. Samples will be taken from both of the traces, of which the analysis result will give the evidence for the LR. In this case, the hypotheses will be structured as follows [19], [20]:

H_p: The two traces both originate from the same (unknown) source.
H_d: The two traces originate from different sources.

Consider the following two examples to help clarify the different types of scenarios.

Example 1 (Specific source). Suppose a cartridge casing has been found on a crime scene, and the forensic expert is in possession of the gun of the suspect. Naturally, the question arises if the cartridge casing was fired from the gun of the suspect (the specific source). In this case, it would make sense to phrase the hypotheses as we have seen in the specific source scenario.

Example 2 (Common source). Suppose bullet cartridge casings have been found at two different crime scenes. One can be interested in knowing whether or not the cartridge casings come from the same firearm or not, without being in possession of this specific firearm. In this case, the hypotheses would be phrased as we have seen in the common source scenario. This way we can say something about the origin of the casings without having a reference weapon.

The following description, based on [26], compares two models for evaluating evidence: the common source model and the specific source model. The difference between the two models is that the evidence is compared to either a fixed (specific) source or a random one under H_p. This distinction impacts the background population considered and thus the sampling model. In the common source model, where a random source is considered, only a single background population is taken into account, namely the one that the sources are presumably a part of. Samples are taken from subjects of this population and compared with samples from the same subject and from different ones. The distribution of pairs from the same source and pairs from different sources are modelled and compared to the evidence.

In the specific source model, where the evidence is compared to a fixed source, two background populations are of interest: one related to the specific source itself, and one to the random sources. So samples are taken both from the specific source and from the random sources, and for both, distributions are modelled. The evidence is compared to both.

It depends on the specific case and on the data available which one of the models one should use. If the question is phrased as in the common source scenario, there is only one possibility. In this case there are two traces and no source of reference, so the only option is to use the common source model, as there is no specific source to compare it to. If the question is of the specific source scenario, both ways are theoretically possible. The specific source model can be used, in which case many samples need to be taken from the specific source as well as from the alternative random sources. However, it is also possible to take the common source approach. In this case, only few samples from the specific source are needed, as well as samples from the alternative random source population. Now, instead of checking if the trace comes from the specific source, one checks if the samples from the specific source and the trace have the same common source (namely, the specific source).

In the specific source scenario where a specific reference source is available, using the specific source model may be expected to give the best results, as some information gets lost when switching to the common source model. However, there are some benefits to using the common source approach. Firstly, it is not always possible to take a lot of samples from the specific source. More often than not, it is only possible to take a few samples. With only a few samples at hand, it is difficult and often unreliable to model a distribution for the specific source samples. In this case, one might want to switch to the common source approach.

Another advantage of using the common source approach is the following. With each case, the data of the traces can be added to a relevant database, for example a glass evidence database. This database can again be used for each new case when using the common source approach, without needing a lot of new samples. When using the specific source approach, however, one must 'start over' every time, because of the nature of the approach.

In practice, the advantages of the common source approach usually outweigh the advantages of the specific source approach, and the common source approach is taken.

2.1.2. Feature-based versus score-based methods

When building an LR-system, both for a common source and a specific source approach, a choice needs to be made between using a feature-based or score-based approach. In a feature-based approach, the evidence consists of feature vectors from reference subjects. The features of the subjects are modeled directly, so a distribution is assumed for the specific features given the hypotheses. If the feature vectors of subjects to be compared are denoted by x and y, then the LR in a feature-based approach can be represented as

$$LR = \frac{f(x, y|H_p)}{f(x, y|H_d)}, \tag{2.5}$$

where f denotes the density of the feature vectors under the competing hypotheses, following the notation in [3].

In score-based systems, the focus is not on examining specific features but rather on comparing differences between various subjects. The features of all possible pairs of subjects are taken into account, and a similarity score is calculated for each pair. The evidence consists of these similarity scores, say $s(x, y)$. In this case, the LR can be represented as follows:

$$LR = \frac{f(s(x, y)|H_p)}{f(s(x, y)|H_d)}. \tag{2.6}$$

Now, the distribution of the scores are modeled both for the pairs that come from the same subject, as well as pairs that come from different subjects. Note that these are now functions of univariate variables, as the multivariate features are reduced to a single score. Examples of scores are Pearson correlation and Euclidean distance. It is also possible to use machine learning algorithms to determine the scores.

The type of method used, depends mostly on the data available. For example, for speaker recognition, typically a score-based method is used [23]. For glass comparison, a feature-based method is used [11]. Both of the methods have advantages and disadvantages.

According to [3], a downside of using score-based methods is that you may lose some information when combining the features into a score. In turn, reducing the multivariate structure to a single dimension improves the robustness of score-based methods against minor fluctuations in one or multiple features, albeit at the expense of their ability to accurately distinguish between weak and strong evidence. Moreover, score-based methods usually do not take rarity into account. When two traces both contain a rare feature, this detail is disregarded when using score-based methods because only the difference between the traces is considered. In such instances the difference will be small, but so will the difference between two traces who both have the same common feature. Ideally, more importance would be given to these rare features. Feature-based models can account for that.

Although feature-based methods are more preserving of the available information, they are generally more difficult to implement. As previously mentioned, the data is often multi-dimensional, and it is hard to find well-fitting models. Moreover, features often have correlation between them, and for this reason one might accidentally be giving too much or too little weight to specific features. Although some information is lost, the scores are much easier to work with than the features. For this reason, score-based methods are usually preferred in practice.

For a more detailed discussion and examples, refer to [3].

2.2. Properties of LR-systems

LR-systems can have different types of properties. Understanding the properties of LR-systems is essential for evaluating their effectiveness and reliability. Some desirable properties of LR-systems are discrimination and consistency. This section will be devoted to explaining these properties, their importance and how to measure them.

2.2.1. Consistency of LR-systems

In an LR-system, the main aim is to ensure that the reported LRs align accurately with the true probabilities of observed evidence under the different hypotheses. The numerical outputs of the LR-system should accurately reflect the likelihood of a given scenario based on the available data. When this is the case, the LR-system is said to be ‘consistent’ or ‘well-calibrated’, as first described by [9]. Consistency is a spectrum: an LR-system can be more or less consistent.

As an LR only provides us with a ratio of probabilities, it is not possible to retroactively say whether or not it was correct. The LR is just a number that says something about the likeliness of an event, but it is not a classification system. An intuitive explanation for consistency of LR-systems is as follows. In a consistent LR-system, at an LR of 5 (or $5:1$), one would expect the relative frequency of H_p divided by the relative frequency of H_d at an LR of 5 to be equal to 5. Here, our evidence is expressed by the LR, which in this case is equal to 5. In other words, the relative frequency of H_p should be 5 times the relative frequency of H_d at an LR of 5. This is equivalent to saying that the probability of the evidence (in this case an LR of 5) given H_p divided by this probability given H_d is equal to the LR, which is 5.

It should be clear that consistency is an important property of LR-systems. If an LR-system is not sufficiently consistent, it is not possible to draw any reasonable conclusions from it.

In Figure 2.1 the LR distributions of data simulated based on a perfectly consistent LR-system is shown. Along the x-axis is the base 10 logarithm of the LR value intervals, and along the y-axis is their relative frequency.

There are many ways in which an LR-system can be inconsistent. Some common ones are the following, as described in [27]. An LR-system can consistently have LRs that are too large, shown by a shift to the right in Figure 2.2. Similarly, the system can also consistently output too small LR-values, shown by a shift to the left, as shown in Figure 2.2. Both these ways of inconsistency contribute to an increase in misleading evidence. It is also possible for LR-values to be too extreme: too large for an LR greater than one and too small for an LR smaller than one. Likewise, LR-values can be too weak. These two ways of inconsistency can be seen by stretched out (too extreme) or pressed together (too weak) LRs. This can be seen in Figure 2.3. For clarity, a dotted line is drawn at log LR (LLR) values of 0 in each of the figures.

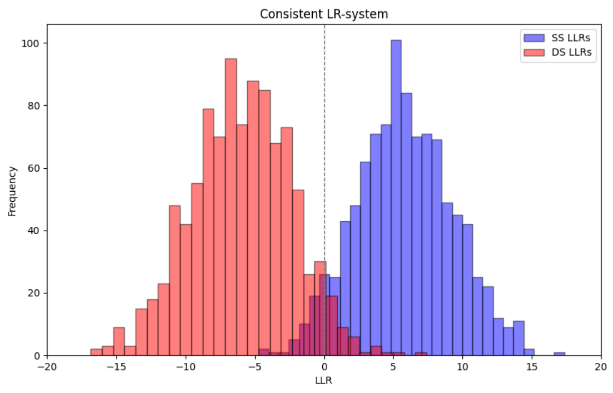

Figure 2.1: Distribution of LRs based on data simulated from a perfectly consistent LR-system.

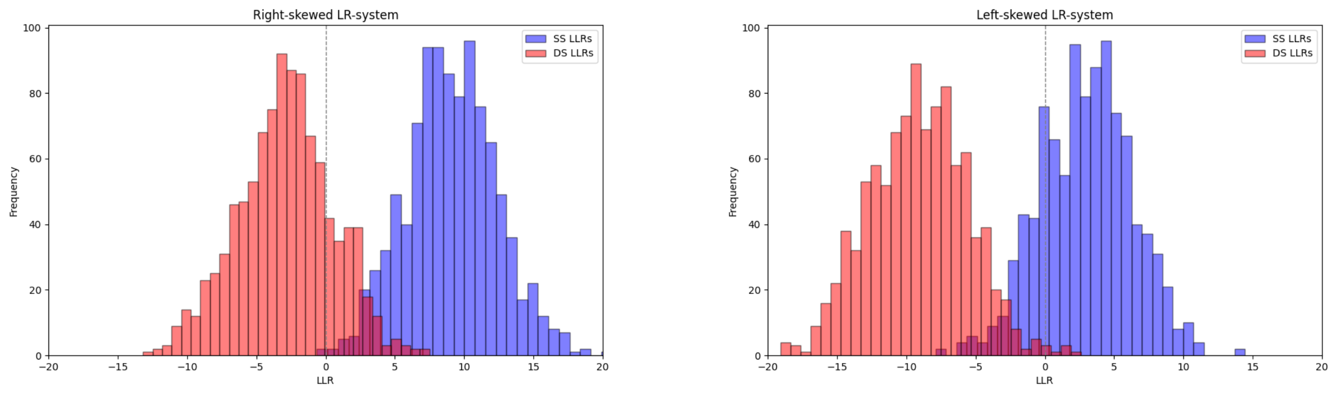

Figure 2.2: Distribution of LRs based on data simulated from LR-systems skewed to the right (left) and to the left (right).

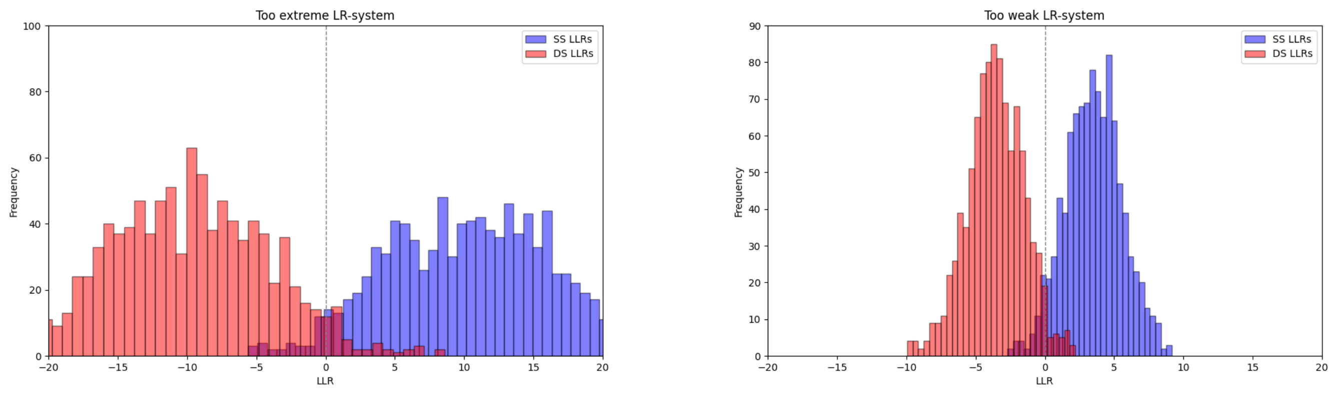

Figure 2.3: Distribution of LRs based on data simulated from too extreme (left) or too weak (right) LR-systems.

For a consistent LR-system, it should hold that 'the LR of the LR is the LR' [27]. For evidence E, the following notation is employed:

$$LR(E) = \frac{\mathbb{P}(E|H_p)}{\mathbb{P}(E|H_d)}$$

Now, saying that 'the LR of the LR is the LR' is equivalent to saying that for a consistent LR-system denoted LR_c, the following equality should hold, as discussed in [14] and further elaborated in [5]:

$$LR(LR_c = V) = V. \tag{2.7}$$

Here, LR_c is the consistent LR-system used, and LR just means ratio of the relative frequencies. So $LR(LR_c = V)$ is the relative frequency with which $LR_c = V$ occurs. This corresponds to the intuitive explanation given earlier on in this section. The proof of Equality 2.7 goes as follows:

Proof.

$$LR(LR_c = V) = \frac{\mathbb{P}(LR_c = V|H_p)}{\mathbb{P}(LR_c = V|H_d)} \tag{2.8}$$

$$\overset{(1)}{=} \frac{\mathbb{P}(H_p|LR_c = V)}{\mathbb{P}(H_d|LR_c = V)} \frac{\mathbb{P}(H_d)}{\mathbb{P}(H_p)} \tag{2.9}$$

$$\overset{(2)}{=} V \frac{\mathbb{P}(H_p)}{\mathbb{P}(H_d)} \frac{\mathbb{P}(H_d)}{\mathbb{P}(H_p)} \tag{2.10}$$

$$= V, \tag{2.11}$$

where (1) follows from isolating the middle term in Equation (2.3), and (2) follows from consistency. Namely, in a consistent LR-system for an LR-value of V, the relative frequency of H_p if V times the relative frequency of H_d. This gives the V multiplied with the prior odds (because it is about the relative frequency). $\square$

A different proof for the equality is given in [5].

It is often possible to improve the consistency of an inconsistent LR-system. This process is called calibration [21]. Possible ways to do so are isotonic regression, kernel density estimation and logistic regression. Some of these methods will be discussed in Section 2.2.3.

2.2.2. Discrimination of LR-systems

In addition to consistency, another desirable characteristic in LR-systems is discrimination or discriminating power. Discrimination refers to the system's capability to effectively distinguish between different hypotheses based on the presented evidence, as first introduced in [9] and further explained in the context of LR-systems in [21]. Ideally, every time H_p is true, the LR-system would report an LR of infinity, and every time H_d is true, the LR-system would report an LR of zero. This system makes a clear distinction between H_p and H_d.

An example of very poor discrimination, is when the LR is always equal to one. This LR-system could still be consistent. However, it doesn't provide any information whatsoever. Therefore, it is of no use in casework. Ideally, the LR-system consistently assigns high values to cases where H_p was true, and low values to cases where H_d was true.

There are several ways in which one can test the discriminating power of an LR-system. They will not be discussed in this report as it is out of scope for this project.

2.2.3. Calibration methods for LR-systems

In this section, some methods will be discussed that can help transform an inconsistent LR-system into a more consistent one. The methods that we will discuss are the PAV algorithm in Section 2.2.4 and kernel density estimation in Section 2.2.5.

2.2.4. The PAV Algorithm

The Pooling Adjacent Violators (PAV) algorithm is a non-parametric method used to calibrate LRs. It is an isotonic regression algorithm, fitting a monotonically increasing function to a dataset. Given a specific dataset and when using a binary scoring method, it has been shown that the PAV algorithm gives optimal consistency for LR-systems [7].

The PAV algorithm was first introduced in [1] and first applied in the context of LR-systems in [6]. The idea of the PAV algorithm is as follows. The data consists of pairs (x_i, y_i), where x_i corresponds to the LR-value and y_i corresponds to the corresponding label (0 for H_d, 1 for H_p). Begin by sorting the data points in non-decreasing order, based on the scores x_i. After sorting, the label for each data point should not be greater than the label of the subsequent data point. If this monotonicity constraint is violated, the algorithm merges the non-conforming data points into their weighted average, where the weights are determined by the number of data points in each group being merged. This procedure continues until the entire sequence satisfies the monotonicity constraint. An example is shown in Figure 2.4.

Once a monotonic sequence of labels is achieved, the next step is to compute the new LRs based on the adjusted, non-decreasing empirical probabilities derived from the merged data points. Specifically, for each group of data points (or bucket) resulting from the PAV algorithm, calculate the empirical probability p_i of the class H_p as the ratio of the sum of labels in the bucket to the total points in that bucket. Now, $1 - p_i$ represents the likelihood of the class H_d. The new likelihood ratio for each bucket is then calculated using the formula:

$$LR_i = \frac{p_i}{1 - p_i}.$$

This new LR value is assigned to all the original data points within the corresponding bucket. Now, the new LRs are monotonic and reflect the adjusted empirical probabilities.

From plotting the PAV transform against the original log LR values, it is possible to deduce in which way the LR-system was inconsistent. Figure 2.5 shows the LR-values of data simulated based on a perfect LR-system (on the x-axis) and its PAV transform (on the y-axis) plotted against each other.

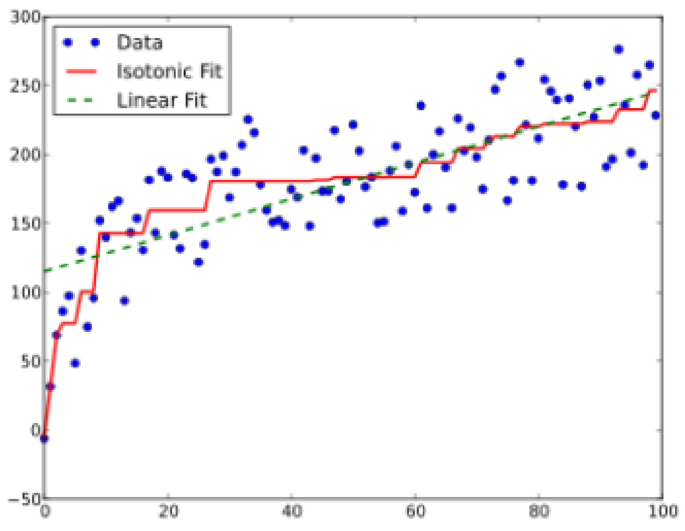

Figure 2.4: Example of PAV algorithm application.

Figure 2.6 shows the PAV-plots of LR-systems that are skewed to the right, meaning the LR values are shifted to the right (biased in favour of the prosecution hypothesis) and skewed to the left (biased in favour of the defense hypothesis. In Figure 2.7, PAV-plots of LR-systems with either too extreme or too weak values are shown.

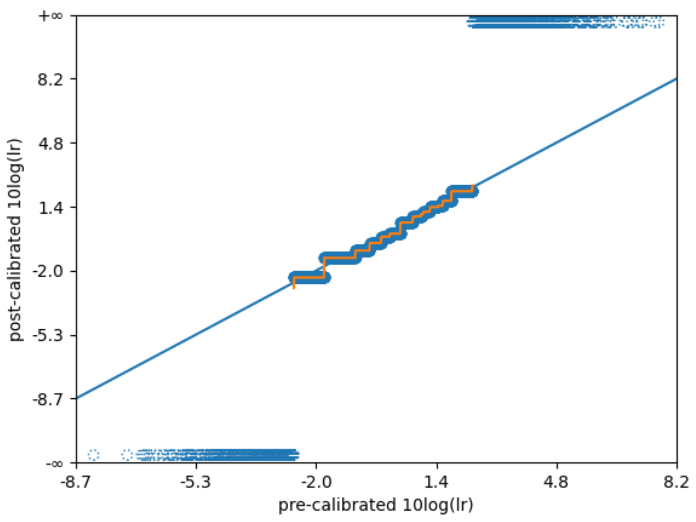

Figure 2.5: PAV plot of data simulated based on a perfectly consistent system.

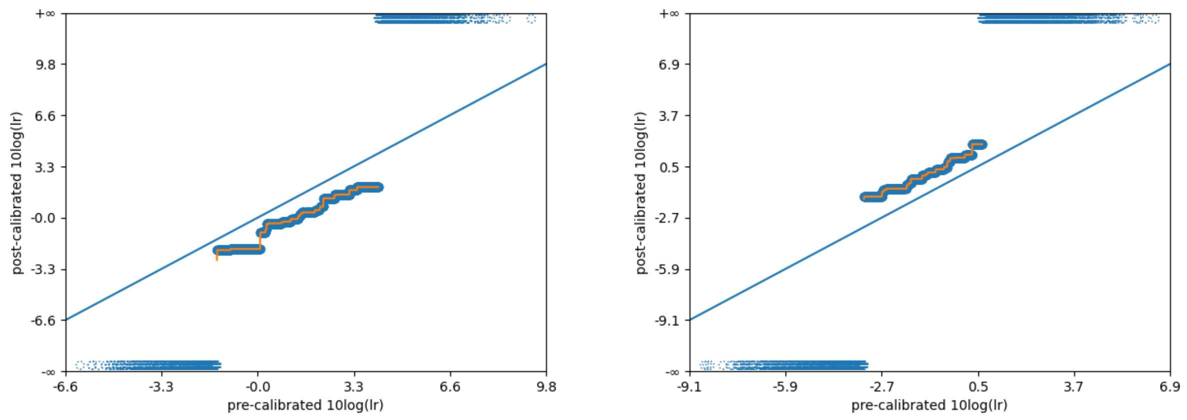

Figure 2.6: PAV plots of data simulated based right-skewed (left) and left-skewed (right) LR-systems.

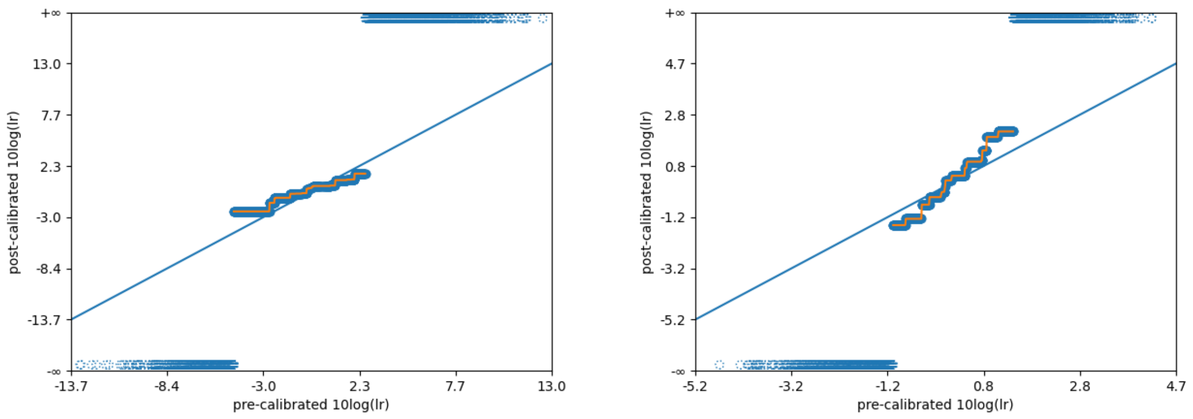

Figure 2.7: PAV plots of data simulated based on too strong (left) or too weak (right) LR-systems.

2.2.5. Kernel Density Estimation

Usually, scores or features cannot immediately be transformed into LRs. A commonly used method to transform scores or features into probability distributions is kernel density estimation (KDE), first introduced in [24]. The idea of KDE is similar to normalizing and smoothing out a histogram. KDE is a non-parametric method to estimate the probability density function of a random variable. This means that it does not assume a specific form for the underlying distribution. However, it does require the selection of one key parameter—the kernel width, which determines the standard deviation of the distribution centered around each data point.

In KDE, a kernel function is applied to each data point, modeling a distribution with the data point as the mean and the chosen standard deviation (kernel width). While the normal distribution is commonly used as the kernel, other distributions can also be applied. The final KDE estimate is obtained by summing and normalizing all these individual kernels. An example is shown in Figure 2.8. For more detailed information on KDE, please consult [23].

2.3. Metrics to measure consistency

While discrimination is definitely important, the number one priority for an LR-system is to be consistent. If there are multiple consistent LR-systems available, the one with the highest discriminating power is generally the one that will be used. Naturally, this leads to the question: how can we measure consistency? This has been a topic of research over the last few years. Several metrics have been proposed, which will be presented in this section.

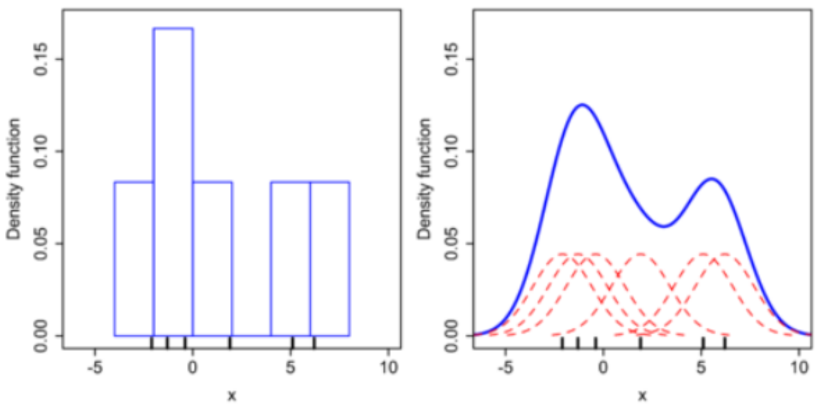

Figure 2.8: Histograms and corresponding kernel density fit.

Some of these methods have been introduced and compared in [27] as well. However, in [27] the data under H_p and H_d were assumed to have a normal distribution. This is definitely not always the case! In this thesis, a comparison will be made without assuming normality of the data. Moreover, a metric will be introduced that has not been discussed yet in [27].

Throughout this section, the letter m will be used to denote the number of LRs under H_d, and n to denote the number of LRs under H_p.

2.3.1. Moments metric

The first metric takes the random variable LR and considers its moments. This metric has been introduced by Good in [14]. He says that for a consistent LR-system, the following inequality should hold:

$$\mathbb{E}(LR^n|H_p) = \mathbb{E}(LR^{n+1}|H_d). \tag{2.12}$$

These moments can be approximated by the empirical data available, and an estimation of the consistency of the LR-system can be made.

The following proof is for the case where the LR has a continuous distribution. The case for the discrete distribution is almost identical. The only difference is that the integral needs to be replaced with a sum, summing over the possible values of the LR.

Proof.

$$\begin{aligned}
\mathbb{E}(LR^n|H_p) &= \int_0^\infty LR^n \ \mathbb{P}(LR|H_p) \ dLR \\
&= \int_0^\infty \left(\frac{\mathbb{P}(LR|H_p)}{\mathbb{P}(LR|H_d)}\right)^n \ \mathbb{P}(LR|H_p) \ dLR \\
&= \int_0^\infty \left(\frac{\mathbb{P}(LR|H_p)}{\mathbb{P}(LR|H_d)}\right)^{n+1} \ \mathbb{P}(LR|H_d) \ dLR \\
&= \int_0^\infty LR^{n+1} \ \mathbb{P}(LR|H_d) \ dLR \\
&= \mathbb{E}(LR^{n+1}|H_d).
\end{aligned}$$

□

Especially interesting in Equation (2.12) are the cases $n = 0$ and $n = -1$, because they give a one on one side of the equality [25]. For $n = 0$ this gives:

$$1 = \mathbb{E}(LR|H_d), \tag{2.13}$$

and for $n = -1$ this gives:

$$\mathbb{E}\Big(\frac{1}{LR}|H_p\Big) = 1. \tag{2.14}$$

Equipped with a set of LR-values, one can estimate the unknown expectations by averaging over the H_d and H_p sets, respectively. This gives, for a consistent LR-system:

$$1 \approx \frac{\sum_{i=1}^{m} LR_{d_i}}{n}, \tag{2.15}$$

and

$$\frac{\sum_{i=1}^{m} 1/LR_{p_i}}{n} \approx 1 \tag{2.16}$$

where the summations are over the sets of LRs for which H_d and H_p are true respectively. LR_{d_i} denotes the i'th LR value for which H_d is true, and LR_{p_i} denotes the i'th LR value for which H_p is true. The sums are divided by the number of elements in the respective sets.

A reasonably consistent LR-system in practice usually has a distribution that's slightly skewed towards misleading evidence, so generally the values of Equations (2.15) and (2.16) will be smaller than 1 [27].

The approximate equalities in 2.15 and 2.16 naturally give rise to metrics to calculate the consistency of LR-systems: the difference between the mean values and one.

2.3.2. Probability of misleading evidence

Intuitively, it is clear that when an LR-system leads to a lot of misleading evidence, this might indicate inconsistency. Royall originally defined a metric in terms of the probability of misleading evidence in [22]. This method is clearly summarized and explained in [27]. As it turns out, the following inequality must hold for a consistent LR-system, for every constant $k \geq 1$:

$$\mathbb{P}(LR \leq \frac{1}{k}|H_p) \leq \frac{1}{k}. \tag{2.17}$$

In words, the chance that we find a small LR when H_p is true, is small and can be bounded. Equivalently, for constant $k \geq 1$, the following must also hold:

$$\mathbb{P}(LR \geq k|H_d) \leq \frac{1}{k}. \tag{2.18}$$

So when H_d is true, the chances of finding a large LR are small and bounded.

It follows that for a consistent LR-system, the chances of misleading evidence are bounded. Filling in values for k gives explicit bounds for the rates of misleading evidence. Specifically, for $k = 2$ this yields the following inequalities:

$$\mathbb{P}(LR \geq 2|H_d) \leq \frac{1}{2}, \tag{2.19}$$

and

$$\mathbb{P}(LR \leq \frac{1}{2}|H_p) \leq \frac{1}{2}. \tag{2.20}$$

Now, a similar approach can be taken as done in Section 2.3.1. The probabilities in Equations (2.19) and (2.20) can be approximated as follows:

$$\frac{\sum_{H_d} \mathbb{1}_{LR\geq 2}}{m}, \tag{2.21}$$

and

$$\frac{\sum_{H_p} \mathbb{1}_{LR\leq 1/2}}{n}, \tag{2.22}$$

counting the times that LR exceeds the values specified in Equations (2.19) and (2.20), divided by the total number of LRs under H_d and H_p, respectively.

To define a metric, we can look at the difference between the quantities defined in (2.21) and (2.22), and $\frac{1}{k}$, where usually k is taken to be 2. It is also possible to look at the limiting rates of the probabilities specified in Equations (2.17) and (2.18), as a function of k.

2.3.3. C_{llr}^{cal} metric

Accuracy of assessments can also be measured by means of strictly proper scoring rules. Strictly proper scoring rules are loss functions, assigning penalties to the posterior probability depending on the ground-truth label. A scoring rule is proper if it is maximized for predictions that align with the true distribution. It is strictly proper if, in addition to being proper, it is only maximized for predictions that align with the true distribution. There are no other predictions that can maximize this score.

To measure consistency in LR-systems, the Empirical Cross-Entropy (ECE) is often used. This method was introduced in [21], on which this section is based. It is based on the logarithmic strictly proper scoring rule. This is a binary scoring rule, defined as follows:

$$LS = -\frac{1}{N_1}\sum_{x:\theta_1 \text{ true}} \log_2(\mathbb{P}(\theta_1|x)) - \frac{1}{N_2}\sum_{x:\theta_2 \text{ true}} \log_2(\mathbb{P}(\theta_2|x)), \tag{2.23}$$

where N_1 and N_2 are the number of comparisons where θ_1 and θ_2 are true. It can be seen that this scoring rule penalizes misclassifications, where the penalty goes to infinity the worse the misclassification gets.

The ECE for LR-systems is defined as follows:

$$ECE = -\frac{\mathbb{P}(H_p)}{n}\sum_{i:H_p \text{ true}} \log_2(\mathbb{P}(H_p|E_i)) - \frac{\mathbb{P}(H_d)}{m}\sum_{j:H_d \text{ true}} \log_2(\mathbb{P}(H_d|E_j)), \tag{2.24}$$

where the sums are taken over the validation sets where H_p and H_d are true respectively, and n and m denote the number of elements in these sets. The ECE measures the cost in terms of the base 2 logarithm of the posterior probability of the ground truth, weighed by the priors. As it is a cost function, it follows that the smaller the ECE value, the better the performance of the LR-system.

The following equality for the odds O is used, which can in turn be used to go from odds to probabilities:

$$O(\cdot) = \frac{\mathbb{P}(\cdot)}{1-\mathbb{P}(\cdot)}.$$

It can now be derived for $i = p, d$ that

$$\mathbb{P}(H_i|E) = \frac{LR \cdot O(H_i)}{1 + LR \cdot O(H_i)}$$

Writing everything out, we find that

$$ECE = \frac{\mathbb{P}(H_p)}{n}\sum_{i:H_p \text{ true}} \log_2\left(1 + \frac{1}{LR \cdot \frac{\mathbb{P}(H_p)}{\mathbb{P}(H_d)}}\right) + \frac{\mathbb{P}(H_d)}{m}\sum_{j:H_d \text{ true}} \log_2\left(1 + LR \cdot \frac{\mathbb{P}(H_p)}{\mathbb{P}(H_d)}\right). \tag{2.25}$$

Usually, the prior odds are not known and the ECE is plotted as a function of the prior odds, or the base 10 logarithm thereof.

An example of the ECE plot in a system where the LR is always equal to 1, is as follows:

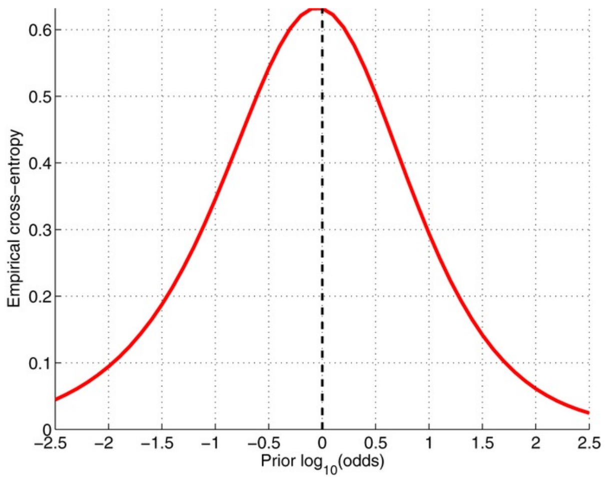

Figure 2.9: ECE plot of LR=1 always.

Often times, the ECE gets plotted for an LR-system both before and after applying isotonic regression (using the PAV algorithm) to the LR dataset, as a function of the prior log odds. As the PAV algorithm reduces the ECE, a big difference between the curves is an indication that the original LR-system was not consistent.

The 'cost log likelihood ratio', or C_{llr}, is a metric used to evaluate the performance of an LR-system. It specifically measures how well the system's likelihood ratios reflect the true probabilities at prior odds of 1 (equivalently, prior log odds of 0). To assess how much the consistency of an LR-system improves after calibration, the 'calibrated cost log likelihood ratio', C_{llr}^{cal}, is used. C_{llr}^{cal} is calculated as the difference between the C_{llr} values before and after applying isotonic regression (using the PAV algorithm) at prior odds of one. C_{llr}^{cal} is often used as a metric to measure consistency of LR-systems.

A large C_{llr}^{cal} value indicates a significant difference between the pre- and post-calibration performance, suggesting that the original LR-system was inconsistent and that calibration was necessary to improve its consistency. This is illustrated in Figure 2.10, where the ECE plot shows the effect of calibration on the LR values.

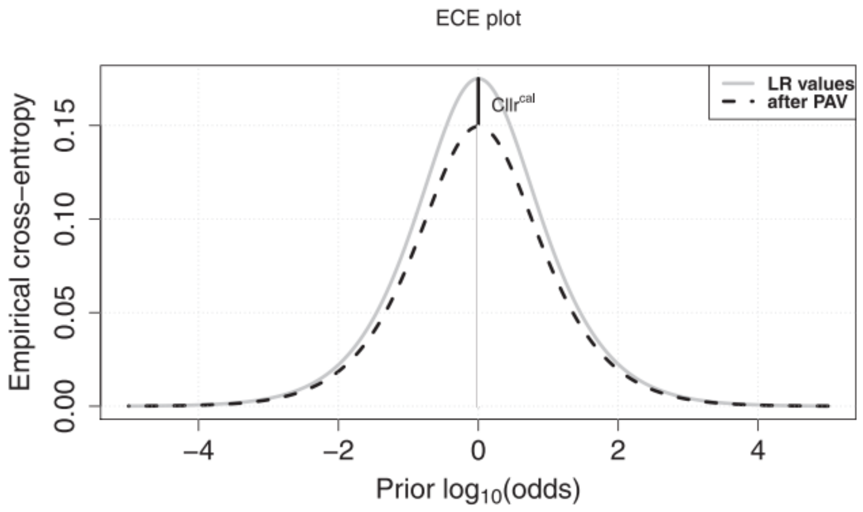

Figure 2.10: Example of ECE plot of LR values before and after PAV.

2.3.4. devPAV metric

The devPAV metric is a more recently introduced metric. It was introduced in [27]. This metric makes use of the PAV-transform of the LR dataset, just like the C_{llr}^{cal} metric which was introduced in Section 2.3.3.

The devPAV metric is defined as the average absolute deviation of the PAV-transform to the identity line $x = y$, both on a logarithmnic scale with base 10. An example is shown in Figure 2.11.

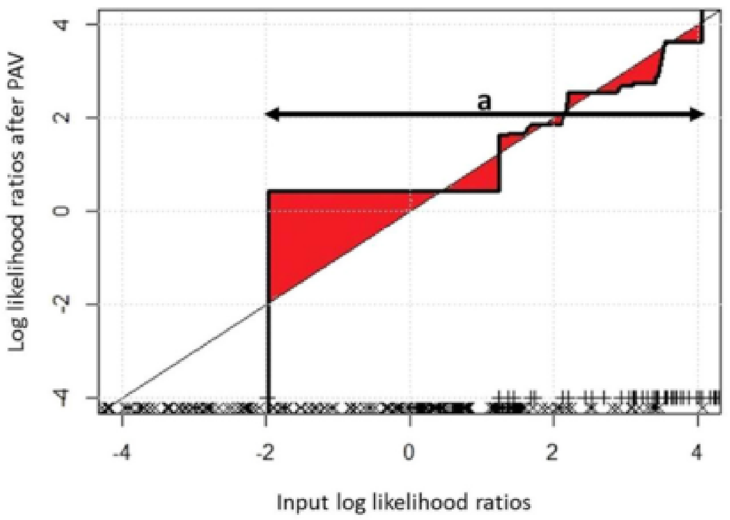

Figure 2.11: Example of the devPAV metric. a is the interval where the PAV-transform is finite. The red area is the value of devPAV.

The PAV transform is seen as the optimal calibration given a specific dataset, and so the devPAV measures the deviation from this optimal calibration. Naturally, a large value indicates inconsistency, whereas a small value indicates better consistency.

For a more detailed explanation of the devPAV metric, please consult [27].

2.3.5. Calibration discrepancy metric

For this approach, introduced in [16], the property that 'the LR of the LR is the LR' is used, as introduced in Section 2.2.1. This section is based completely on [16] and [15]. First, some notation is needed.

Let $g(r)$ denote the probability density function (pdf) of the LR under H_p, so

$$g(r) = f(r|H_p).$$

Note that r here represents the LR as a random variable. Similarly, let $h(r)$ denote the pdf of the LR under H_d. Let $G(r)$ and $H(r)$ denote the corresponding cumulative distribution functions (cdf). Now, from Section 2.2.1 it is known that for a consistent LR-system, the following must hold (for $r \geq 0$, $f(r) \neq 0$):

$$\frac{g(r)}{h(r)} = r,$$

or equivalently,

$$g(r) = rh(r).$$

Integrating both sides over the interval (a, b) (where $0 < a < b < \infty$), using partial integration on the right-hand side, gives the following equality:

$$G(b) - G(a) = bH(b) - aH(a) - \int_a^b H(r)dr. \tag{2.26}$$

Now, taking the logarithm and subtracting the right-hand side on both sides in equation 2.26, it follows that the following must hold for a consistent LR-system:

$$\log_{10}(G(b) - G(a)) - \log_{10}\left(bH(b) - aH(a) - \int_a^b H(r)dr\right) = 0. \tag{2.27}$$

The distributions $G(r)$ and $H(r)$ can be estimated with fiducial distributions obtained from the ground-truth known empirical data, which consists of a set of LR values from cases where it is known H_p was true, and a set of LR values from cases where it is known H_d was true.

Now, define the 'interval specific calibration discrepancy' $d_{(a,b)}(G,H)$ for the interval (a,b) as

$$d_{(a,b)}(G,H) = \log_{10}(G(b) - G(a)) - \log_{10}(bH(b) - aH(a) + \int_a^b H(r)dr). \tag{2.28}$$

If both of the terms within the log are zero, we set $d_{(a,b)}(G,H)$ to zero as well.

Note that $G(b) - G(a)$ is the probability of observing an LR value in the interval (a,b) when H_p is true. If $d_{(a,b)}(G,H)$ is smaller than zero, the value of evidence is overstated by the LR-system, because fewer observations are falling in the interval (a,b) than would be expected for a consistent LR-system. Equivalently, if the value of $d_{(a,b)}(G,H)$ is greater than zero, the value of the evidence is understated by the LR-system.

Not only do the values of $d_{(a,b)}(G,H)$ tell us whether the evidence is being overstated or understated in the interval (a,b), they also tell us the factor with which this has been done. Suppose $d_{(a,b)}(G,H) = x$. If $x > 0$, then, the LR values between a and b have been overstated by 10^x in favour of the defense hypothesis, on average. Equivalently, if $x < 0$, the evidence has been overstated by the same factor in favour of the prosecution hypothesis.

Using the fiducial distributions $G(r)$ and $H(r)$ allow us to form confidence intervals for $d_{(a,b)}(G,H)$. If the confidence bounds for an interval (a,b) do not include zero, it can be concluded that the LRs in this interval are not consistent.

Now for a sequence $0 < a_1 < \cdots < a_n < \infty$, define simultaneous confidence intervals

$$d(G,H) = (d_{(a_1,a_2)}(G,H), d_{(a_2,a_3)}(G,H), \ldots, d_{(a_{n-1},a_n)}(G,H))^T. \tag{2.29}$$

Every interval that does not contain zero indicates inconsistency for that specific interval.

The approach can be visually represented using 'calibration discrepancy plots', an example of which is given in Figure 2.12, taken from [15].

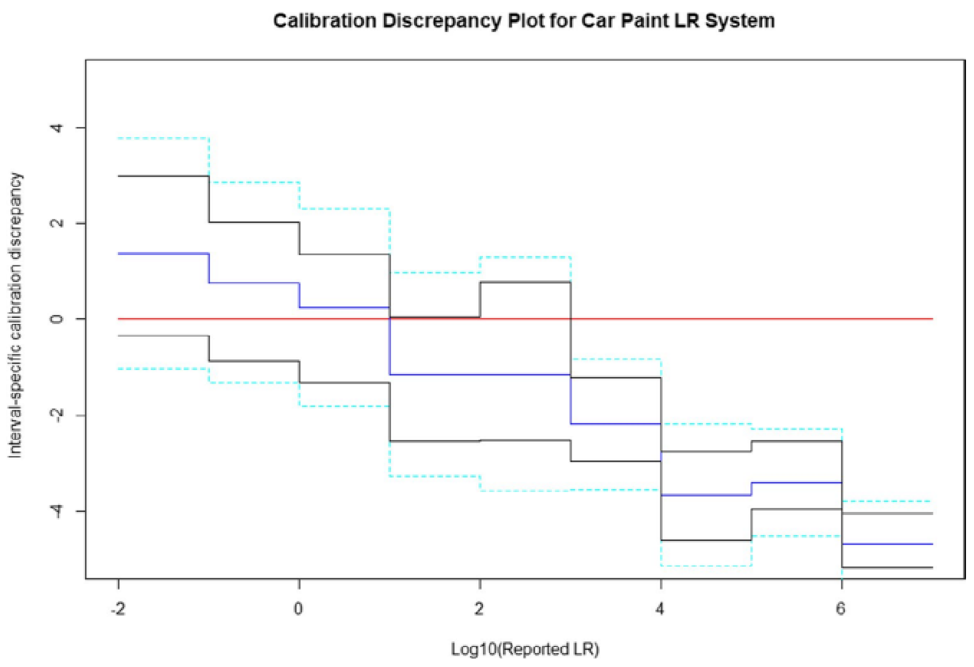

Figure 2.12: An example of a calibration discrepancy plot for an LR-system.

The vertical axis represents the discrepancy $d_{(a,b)}(G,H)$ and the horizontal axis represents the base 10 logarithm of the LR. The red line indicates perfect consistency with zero discrepancy. The light blue lines are the 95% simultaneous confidence bounds and the black lines are the 95% pointwise confidence bounds. The dark blue line is the median of the fiducial distribution and is therefore the estimate of the calibration discrepancy. These confidence intervals can be determined as follows, following the approach taken in [15].

The first step is fiducial sampling. The original data is sorted, leaving us with two sorted datasets: the H_p and the H_d dataset. Now, fiducial samples are determined. The number of fiducial samples we used in this thesis is 100, but can be varied. In each fiducial sample, for the datasets sorted random uniform numbers are generated between zero and one (the same amount as the number of elements in the dataset), adding one at the beginning and zero at the end. These generated numbers represent positions along the cumulative distribution function of the sorted data. Each number corresponds to a specific quantile. Now, for each of the fiducial samples, these quantiles of the data are determined using the empirical data at hand, resulting in a new sample that captures the distributional characteristics of the observed data.

The second step is determining the survival and integral functions. To compute the survival functions, we look at the probability that the LR is greater than a given value. This is done by using a grid and capturing the relative frequencies of LR-values exceeding each grid point. The integral function is determined by summing up contributions from adjacent data points, weighed by the corresponding survival probability. The survival functions are used to determine the left-hand side of equation 2.26, and for the right-hand side the survival functions are multiplied with the grid points and summed with the integral functions.

In the third step, the fiducial difference is calculated by measuring the difference between the right-hand side and the left-hand side of equation 2.26, using the values found in the previous step to approximate them. A positive outcome indicates higher LR-values for H_p data, whereas a negative value indicates higher values for H_d data.

Lastly, the fiducial confidence intervals are determined. This is done by first calculating the median of the fiducial difference distribution for each grid point, determining the maximal deviation from the median and then determining the cut-off value to ensure the desired coverage probability. This way, confidence intervals for the fiducial difference are constructed at each grid point.

A more detailed explanation on how these confidence bounds are determined can be found in [15].

The intervals where the red line is outside of the confidence bounds, are the intervals where the LR-values are inconsistent. For example: the LRs between 10^4 and 10^5 are overstated by a factor of at least 10^2.

An advantage of this method over other methods such as the commonly used C_{llr}^{cal} is that it does not only give information about whether or not the LR-system is consistent, but also the degree with which the evidence is being overstated. Theoretically, this could be used to make an LR-system more consistent.

2.4. Building LR-systems

In this section, a step-by-step approach to build an LR-system will be presented. Several ingredients are needed and several choices need to be made along the way. Most of this section is based on [17]. In this section we assume we are in a common-source scenario, using a score-based approach.

2.4.1. The data

First of all, the data for the LR-system needs to be collected and pre-processed. It is important to understand the data well and format it correctly. Extreme values need to be investigated; errors need to be distinguished from results caused by natural variability, and removed. Variables that have strong correlation might be combined into one variable. For example, if data is collected from people, and you have the variables weight, length and BMI, one of these might be removed, as BMI is just a combination of weight and length.

Often times, the data is standardized to avoid the domination of certain variables just because they have bigger values (e.g.: length in centimeters will usually give higher values than weight in kilos).

It is important to store the data appropriately so that it is easy to understand and easy to access. Usually this is done using tables.

Now, the data must be split in subsets. Around 20% of the data will be used as a validation set. If we are testing multiple LR-systems, the remaining data needs to be split into a selection set, containing around 10 to 20% of the remaining data, and the training data, which is the rest. As observations over the same subject are usually extremely correlated, it is advisable to split the data so that observations on the same subject are in the same set.

First, the system is trained on the training data. Then, using the selection set, the best model is selected. The validation set is then used to validate the specific model.

It is possible to use cross-validation. In k-fold cross-validation, the data is split into k non-overlapping sets or 'folds'. Each round, one of the k folds is used as validation set and the model is trained on the rest of the data. The performance characteristics are based on the combination of these k sets of LRs. For a visual representation of k-fold cross-validation, see Figure 2.13. The same process can also be done with the selection set.

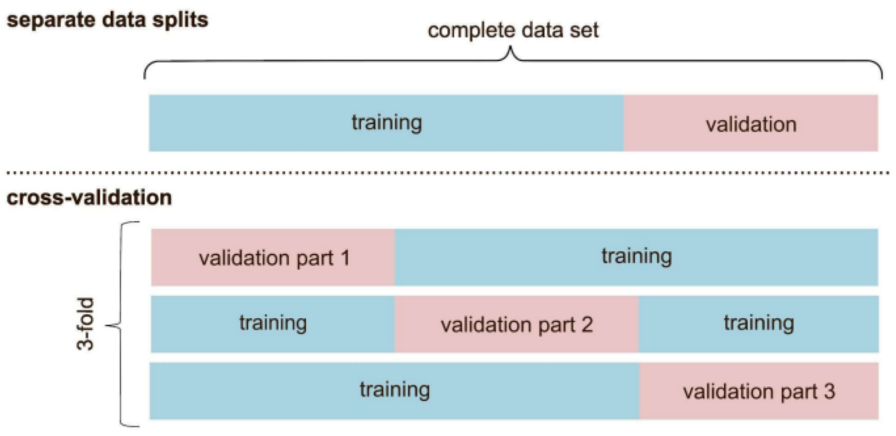

Figure 2.13: Example of 3-fold cross-validation.

2.4.2. The scores

Now, a choice needs to be made, namely between a score-based and a feature-based LR-system. In this section we will assume a score-based LR-system is used, as this is common practice. This step needs to be adjusted accordingly when using a feature-based method.

First, we need to construct a new dataset containing a set of paired features for H_p-comparisons and a set of paired features for H_d-comparisons. From these pairs, scores need to be deducted. Examples are cosine similarity (the higher, the more similar) and Euclidean distance (the higher, the less similar). Many score and distance functions are possible here. The score should differentiate between pairs from the H_p set and pairs from the H_d set. It is also possible to use machine learning algorithms, outputting high values for H_p pairs and low values for H_d pairs.

Now, the values of interest are the scores for pairs and the corresponding label (H_p-true pair or H_d-true pair, they can be denoted by 1's and 0's respectively).

2.4.3. The LR values

Equipped with scores, we have to construct a function to transform them to LRs. There are several options to do this. One option is to fit pdf's to each of the two sets of scores. A commonly used method is KDE, which was discussed in more detail in Section 2.2.5. It is also possible to fit a function that relates the score to the posterior probability that H_p is true. Examples of such methods are logistic regression and isotonic regression. Isotonic regression was discussed in Section 2.2.4.

2.4.4. Selection and validation

If multiple LR-systems have been built, the best one can now be selected by training the systems on the training data and using the selection data to select the one that performs best. This can be done by looking at different performance metrics, as discussed in Section 2.3 and by plotting the LR values and looking at the discrimination. Usually, a simple model is preferred over a complicated one.

Now, with one LR-system left, the LR-system needs to be validated. The aim of validation of the system is to study its performance on independent data. There are several methods that test performance of LR-systems, most of which were discussed in Section 2.3.

Ideally (and usually), the system performs better when the quality or quantity of data increases.

2.5. Comparing the metrics

In this thesis, a comparison will be done between different metrics, to find the optimal one. In [27], several metrics have already been compared with regards to differentiation between more and less consistent LR-systems, and stability across dataset size: the moments metric, the probability of misleading evidence, C_{llr}^{cal} and the devPAV metric. It was shown that the moments metric and the probability of misleading evidence performed significantly worse than C_{llr}^{cal} and devPAV. Between C_{llr}^{cal} and devPAV, devPAV seemed to come out on top,although the difference was small.

However, an important assumption was made about the distribution of the LR data, which is definitely not always true: it was assumed that both the H_p- and the H_d-data are distributed according to a normal distribution with equal variance and mirrored means. This is usually not in line with reality.

The moment metric and the probability of misleading evidence will not be taken into account in this thesis, because of their poor performance in the comparison done in [27]. The calibration discrepancy method has not yet been compared, so this one will be compared to C_{llr}^{cal} and devPAV.

In this comparison, several factors will be taken into account. Most importantly, it should differentiate well between consistent and inconsistent LR-systems, and the degree of inconsistency. Moreover, it should be stable for different means, different sample sizes etc. For a reliable metric, it is possible to define a range of values in which the consistent LR-systems would fall.

In the upcoming sections, the aim is to determine which metrics best meet these criteria. By a systematic evaluation of these metrics, we hope to gain insights into their respective strengths and limitations, allowing us to select the best metric for different purposes.

3 Methods

The main goal of this thesis is to evaluate different metrics used to determine the consistency of LR-systems, and to see which one works best. The process is split up into two steps. The first step is the optimization of the metrics that we will be comparing. The second step is the comparison of these optimized metrics.

The metrics we will be taking into account are the commonly used C_{llr}^{cal} as described in Section 2.3.3, the devPAV metric as described in Section 2.3.4 and the calibration discrepancy as described in Section 2.3.5. Note that the last one is not a metric yet: it doesn't output a single value, it is just a way to gain insights into the consistency of LR-systems. We will derive several metrics from this method and compare them to each other.

3.1. Step 1: Optimization of the metrics

Before comparing them to each other, the three above-mentioned metrics will be optimized. For each of the metrics, a small investigation will be performed to find the optimal version of it. An example of the code can be found in Appendix A.2.

For the metrics, the different versions will be compared to each other in the following manner, similar to the approach taken in [27]. Nine different sets of values will be generated, which will be the log LR-values (). The logarithm here is the natural logarithm (e-based). Unless differently specified, this holds true in the rest of this report for all logarithms. Each set will be made up of same-source (referred to as SS) and different-source (referred to as DS), where both groups will follow a normal distribution. The nine sets can be divided into five categories, which will be generated as follows:

- Consistent LR-values: $SS \sim N(\mu_{SS}, \sigma^2 = 2\mu_{SS})$, $DS \sim N(-\mu_{SS}, \sigma^2)$,
- LR-values with bias favouring H_p: $SS \sim N(\mu_{SS} + c_1, \sigma^2)$, $DS \sim N(-\mu_{SS} + c_1, \sigma^2)$,
- LR-values with bias favouring H_d: $SS \sim N(\mu_{SS} - c_1, \sigma^2)$, $DS \sim N(-\mu_{SS} - c_1, \sigma^2)$,
- Too extreme LR-values: $SS \sim c_2 \times N(\mu_{SS}, \sigma^2)$, $DS \sim c_2 \times N(-\mu_{SS}, \sigma^2)$,
- Too weak LR-values: $SS \sim \frac{1}{c_2} N(\mu_{SS}, \sigma^2)$, $DS \sim \frac{1}{c_2} N(-\mu_{SS}, \sigma^2)$.

The set of consistent LR-values will be generated in Python, and the other sets are obtained by shifting and rescaling these values.

For this comparison, μ_{SS} is chosen to be equal to 6. This corresponds to a log 10-LR of about 2.6 and a normal LR of about 403. For c_1, the values 1 and 2 are used. For c_2, we use 1.5 and 2.5. These values are chosen equivalently to the values in [27].

We will look at three different sizes of the datasets to see how the metrics are affected by the amount of data at hand. First, for each dataset, $n_{SS} = 150$ same source LR-values are generated, and $n_{DS} = 3 \cdot 150 = 450$ different-source LR-values. This gives a decently sized dataset where one may expect the metrics to make an accurate distinction. We will also look at a smaller dataset, namely $n_{SS} = 50$ same-source LR-values and $n_{DS} = 150$ different source LR-values. Lastly, we will look at datasets of sizes $n_{SS} = 300$ and $n_{DS} = 900$. It is common for k same-source LR-values, to have $\binom{k}{2}$ different-source LR-values. However, this is quite computationally demanding, especially for the fiducial method. For this reason we do not use this size of dataset for different-source LRs.

After generating the data, for each of the dataset the metrics we want to compare are calculated. The values are stored and this process is repeated N times, to minimize the impact of randomness. For this project, we have taken N to be equal to 1000. For each metric, 1000 values are stored. The distributions of these values are plotted to see if the metric makes a good distinction between consistent and inconsistent LR-systems. This can be deducted from the overlap in the values of the metrics for consistent and inconsistent LR-systems, where a lot of overlap indicates that the metric does not distinguish well between systems. The overlap will be quantified with a percentage, where a small percentage of overlap is preferable over a large percentage. This will be explained in more detail in Section 4.1.

3.1.1. Optimization of C_{llr}^{cal}

For C_{llr}^{cal} the optimization will be done as follows. As previously explained, the C_{llr}^{cal} metric is based on the Empirical Cross Entropy. The logarithmic scoring function at prior odds of 1 (or equivalently prior log odds of 0) is evaluated both for the normal LR-values and for the LR-values after applying the PAV-algorithm to them. The difference between these two values is C_{llr}^{cal}.

In the optimization step, we will research if it is possible to replace the logarithmic scoring rule used for C_{llr}^{cal} with other possible scoring rules. The logarithmic scoring rule that is originally used for this metric, has the important property that it is strictly proper, meaning that it is optimized for LR-values that align with the real underlying distribution. It gives a high penalty for LRs that point in the wrong direction. The stronger they point, the higher the penalty, limiting in infinity. This penalty grows logarithmically. Although this might be useful in real-life situations, it is also interesting to look at more 'symmetric' score functions, that do not penalize misleading LR-values as extremely. The scoring rules we are going to look into are the Brier score, the spherical scoring rule and the zero-one loss.

The Brier score (BS) is strictly proper, just like the logarithmic scoring rule originally used for the C_{llr}^{cal}. The formula for the Brier score is given by

$$BS(p, y) = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)^2,$$

where the p_i's are the posterior probabilities, derived from the LRs similarly as in Section 2.3.3, and the y_i are the corresponding ground truth labels, 1 for H_p true scenarios and 0 for H_d true scenarios. In this formula, N equals the total number of LRs.

The spherical scoring rule (SP) is strictly proper as well. In the case of binary classification, it is defined as follows:

$$SP(p, y) = \frac{1}{N}\sum_{i=1}^{N}\frac{y_i p_i + (1 - y_i)(1 - p_i)}{\sqrt{p_i^2 + (1 - p_i)^2}}.$$

Lastly, the zero-one loss (S) is defined as follows:

$$S(p, y) = \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}_{\{p_i > 0.5 \text{ and } y_i = 0\}} + \mathbb{1}_{\{p_i < 0.5 \text{ and } y_i = 1\}}.$$

The zero-one loss basically determines the percentages of misclassifications, where a misclassification is defined as a posterior of greater than 0.5 with a corresponding ground-truth label of 0, or a posterior of smaller than 0.5 with a corresponding ground-truth label of 1. The zero-one loss is not proper.

3.1.2. Optimization of devPAV

The devPAV metric will be optimized in the following manner. As can be seen in Figure 2.11, the devPAV metric is calculated by summing over a set of surfaces of a step function, obtained by performing isotonic regression on the original LR-values. This surface is scaled by the x-range.

There are several ways in which this could be adjusted. Firstly, the devPAV value could be normalized by dividing the value of the sum of the surfaces over the total surface, instead of just over the length of the x-axis.

Another possible way to adjust devPAV is to 'cut off half of the corners' of the steps. This smooths out the function a little bit, decreasing the effect of the big steps that usually appear at the beginning and end of the function. The steps can be large here because these regions typically involve more extreme LR-values, where the PAV algorithm needs to make more substantial adjustments to enforce monotonicity. These adjustments result in larger deviations of the line $x = y$, especially in areas where the original LRs are sparse or where there are significant jumps in their values.

In Figure 3.1, the calculation of the original devPAV is shown: the x-axis shows the original LR-values, the blue curve shows the corresponding LR-values after the PAV algorithm and the marked area is summed up and divided by the x range to get devPAV. Now, in Figure 3.2, the graphs are rotated so that the line $x = y$ lies on the x-axis. The original step function is shown in blue, and the smoothed function by cutting the corners is shown in orange.

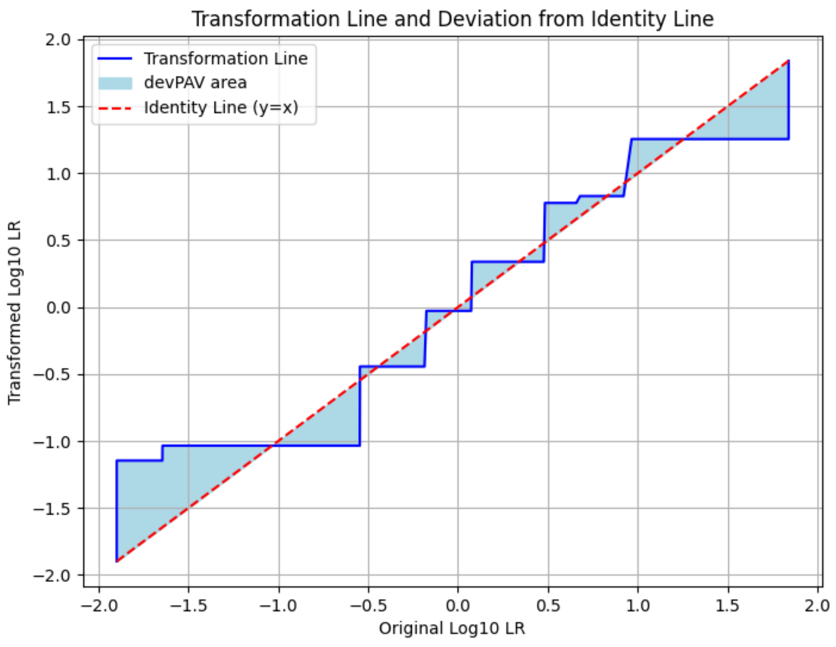

Figure 3.1: The normal calculation of devPAV.

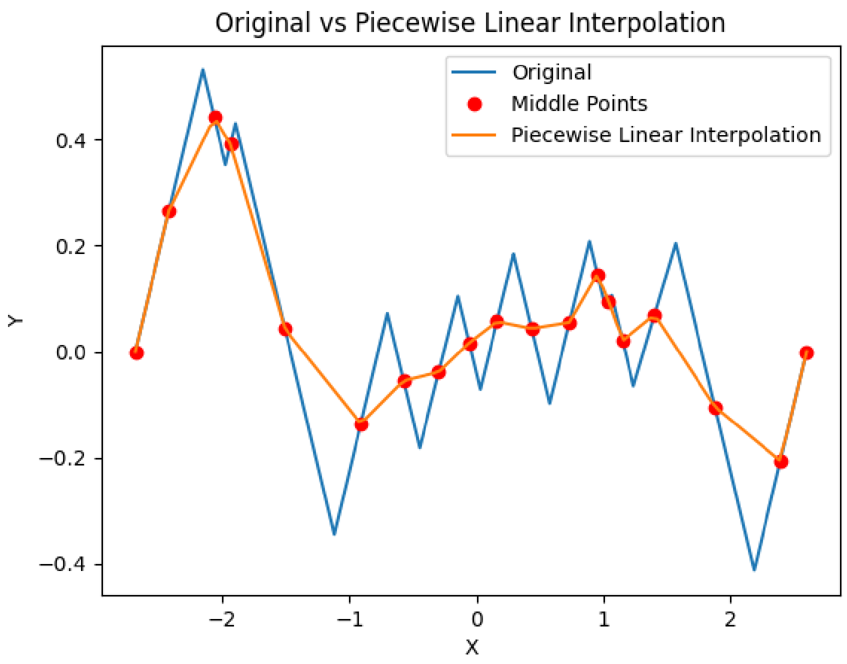

Figure 3.2: The normal step function of the LR-values before versus after isotonic regression in blue, as well as the smoothed function by cutting off corners of the steps in orange. On the x-axis the original LR-values are shown, and on the y-axis the LR-values after applying the PAV-algorithm. The graph is rotated.

3.1.3. Optimization of fiducial metric

For the fiducial method, a metric has not been defined yet. We will look into several options and, using the aforementioned method, check which one works the best in distinguishing between the consistent and inconsistent systems. The three metrics we have defined are the following three.

The first metric we will take into consideration is perhaps the most intuitive one: simply averaging over the median values of the fiducial confidence intervals for the specified LR-values. Since we do not want positive and negative medians to cancel each other out, we take the absolute values of the medians. The formula is given by

$$FI_{\textbf{med}} = \sum_{i=1}^{k} \frac{|M_i|}{k},$$

where M_i is the median of the i'th interval, and k is the total number of intervals. The smaller the values of the metric, the better.

Ideally, LR-systems which result in wider confidence intervals would give larger values of the metric (where a smaller value is better) then LR-systems which result in smaller confidence intervals, if the medians are the same. This results in a second metric to be taken into consideration: again, we sum over the medians, but we weigh by the width of the confidence intervals:

$$FI_{\textbf{scaled}} = \sum_{i=1}^{k} \frac{|M_i| \times (U_i - L_i)}{k},$$

where U_i is the upper bound of the confidence intervals and L_i is the lower bound.

Lastly, a very simple metric is taken into account, namely simply counting the amount of times that the value 0 does not lie within the confidence bounds. As explained in Section 2.3.5, this is an indication of inconsistency. The metric is formulated as follows:

$$FI_{01} = \sum_{i=1}^{k} \frac{\mathbb{1}_{\{0 \notin C_i\}}}{k},$$

where C_i is the i'th confidence interval.

3.2. Step 2: Comparing the metrics

After optimizing the three metrics, they will now be compared to each other. This will be done differently than the optimization process in the previous section. So far, we have only looked at normally distributed LR-values, similar to the methods used in [27]. However, in a real-life setting the LR-values are not usually distributed in this way. We would like to compare the metrics on distributions that are more similar to distributions that we can find in real-life LR-systems.

We will generate more realistic datasets in the following way. We use seven existing datasets of log10 LR-values that have been used in publications, as well as normally distributed generated data as per Section 3.1. We will use the different-source , as we usually have more of these than of the same-source , to generate new same-source . These same-source will be generated in such a way that the different-source and the same-source together form a consistent LLR-system. From here, we can adjust the as we have done in Section 3.2 by shifting and scaling them to create inconsistent systems. By evaluating the different metrics on both the consistent and the inconsistent systems, we can calculate the overlap percentage to see how well the metrics differentiate between them. We will do this for different sizes of datasets, to also see how much the metrics depend on the size of the dataset. Moreover, we will determine the values of the metrics for the different consistent LR-systems we generated, to see how much the value differs between different datasets. Ideally, the values of the metrics would be roughly the same for different consistent LR-systems, so that when we have a new system and obtain a value for a metric, we can say something about the consistency of this system.

The exact procedure is explained in the following subsections.

3.2.1. Datasets

As previously mentioned, seven datasets have been used for this part of the thesis. The first five are from [2] and contain LLRs about firearm toolmarks on cartridge case primers. Four of these datasets correspond to those used in Figure 9 of [2], encompassing both the stages of cross-validation and the final model. Each of these stages includes one dataset where the ELUB bounds are applied and one dataset before applying the ELUB bounds. The fifth dataset corresponds to the data of the final model used for Figure 29 of [2].

The other two datasets contain data about gun shot residue and gasoline traces.

Lastly, we use LR-data generated as per Section 3.1, where the follow a normal distribution. For this part, we will use $\mu_{SS} = 6$.

The datasets used are summarized in Tables 3.1 and 3.2. All percentiles are expressed using log10 LR-values.

	Type of data	ELUB bounds	Number of SS	Number of DS
Data 1	Firearm toolmarks on cartridge case primers	Yes	188	477
Data 2	Firearm toolmarks on cartridge case primers	No	188	477
Data 3	Firearm toolmarks on cartridge case primers	Yes	188	1327
Data 4	Firearm toolmarks on cartridge case primers	No	188	1327
Data 5	Firearm toolmarks on cartridge case primers	Yes	199	6501
Data 6	Gun shot residue	No	37	835
Data 7	Petrol	Yes	99	5964
Normal data	Generated data	No	-	-

Table 3.1: Characteristics of the datasets.

	SS 5th Percentile	SS Median	SS 95th Percentile	DS 5th Percentile	DS Median	DS 95th Percentile
Data 1	-0.85	1.55	1.55	-0.85	-0.85	0.07
Data 2	-0.87	2.75	29.07	-0.97	-0.85	0.07
Data 3	-0.88	1.81	1.81	-0.88	-0.88	0.01
Data 4	-0.89	2.85	29.28	-0.92	-0.88	0.01
Data 5	1.63	3.76	3.76	-2.1	-2.1	-0.95
Data 6	-0.42	1.65	1.91	-1.88	-1.18	-0.07
Data 7	0.73	2.69	3.96	-100.0	-100.0	0.37
Normal data	0.13	2.61	5.08	-5.08	-2.61	-0.13

Table 3.2: Percentile values for SS and DS LRs.

Note in Tables 3.1 and 3.2 that the characteristics of some datasets are very similar. This can be explained as follows: Data 1 and Data 2 are based on the same data, only on Data 1, the ELUB bounds are applied and in the second they are not. The same is true for Data 3 and 4.

3.2.2. Generate same-source data

In this section, the approach taken to generate same-source LLRs will be explained. All the corresponding code can be found in Appendix A.1. To obtain a consistent LR-system from a given dataset, we start by taking only the different-source LLR-values from the dataset. Note that, using the relation in Equation 2.7, we can now generate same-source data so that this relation is satisfied and hence, the system is consistent.

The straightforward approach would be to divide the different-source LLRs into bins. Now for each bin, the average value is used as LLR and using the frequency of different-source LLRs in this bin and the relation in 2.7, a corresponding same-source LLR frequency can be calculated. However, this approach has a problem. As this relation uses the relative frequencies, we want the total of the relative frequencies for both same-source and different-source to sum up to one. This is per construction the case for the different-source . However, for the same-source it is not. Consider the following example: if we have 100 different-source , all of which are equal to log(0.5). Then, as per 2.7 we would generate $0.5 * 100 = 50$% of our same-source LLR-values as being log(0.5), and we would not generate any other values because we do not have different-source LLRs of different values.

To overcome this problem, we do the following. First, we use a kernel density estimate (KDE) to approximate the distribution of the different-source LLR-values. This gives us a smooth approximate rather than just the discrete bin values. Now, we generate a new curve for the same-source LLR-values using the KDE so that Equation 2.7 holds. We then determine the surface under both of the curves, which we want to be nearly equal to each other. At this point, the area under the KDE will be roughly equal to one.

If the surface under the KDE curve is larger than the surface under the same-source curve, it means that the total of the relative frequencies of the different-source is higher than the total of the relative frequencies of the same-source LLRs. In this case, we 'stretch' the different-source LLR-values and the corresponding KDE-values to the right, so we get larger LLR-values for the same frequencies. We do this by shifting so that the smallest different-source LLR is zero, then multiplying by $1 + \alpha$, where $\alpha = 0.001$ and then shifting back. This way, the values of the are stretched out to the right. Because our original frequencies are now tied to larger LLR-values (the height of the graph does not change, it is just stretched out to the right), we generate higher frequencies of same-source LLR-values because of the relation in 2.7. We keep repeating this procedure with the new stretched LLR-values and corresponding curve until the areas under curves are almost the same, using a tolerance of 0.01. Note that the areas under the curves no longer need to be equal to one: by stretching the KDE curve, we have changed the area under it. We want the area under the same-source LLR-curve to be equal to the area under the different-source KDE to have the same frequency of same-source , and we could normalize them to both equal one, however, this is not necessary: we are interested in the ratio of same-source versus different-source , which does not change by normalizing.

It is also possible that the surface under the same-source curve is larger than the surface under the different-source KDE curve. In this case, we want to generate lower frequencies of the same-source . We do this by shifting the KDE curve to the left. Similarly as before, we shift the (and so the KDE) so that the smallest LLR has value zero, then multiply by $1 - \alpha$ to obtain smaller frequencies of same-source LLR values. After that, we shift back. So instead of stretching to the right, we now stretch to the left. Again, this procedure is repeated until the areas under the curves are almost equal.

After performing the previously explained procedure, we have two curves corresponding to the different-source and same-source LLR-frequencies as a function of the LLR-values. Using these curves, we can now generate data according to these distributions that together specify a consistent LR-system. In Section 4.2.1, the procedure is visualized.

3.2.3. Test metrics on new data

Now that we are equipped with a way to generate data based on a consistent LR-system, we can go from here and shift and scale this data to make it inconsistent. In the exact same way as in Section 3.1, for each of the seven previously mentioned datasets, we test the metric on nine sets, divided into five categories:

- Consistent LR-values,
- LR-values with a bias favouring H_p: shifted to the right by c_1,
- LR-values with a bias favouring H_d: shifted to the right by c_2,
- Too extreme LR-values: scaled by c_2,
- Too weak LR-values: scaled by $\frac{1}{c_2}$.

The consistent LR-values are generated as per Section 3.2.2. Again, for c_1 the values 1 and 2 are used, and for c_2 the values 1.5 and 2.5 are used.

Similarly as before, we will look at small datasets, medium-sized datasets and large datasets. For the small datasets, $n_{SS} = 50$ and $n_{DS} = 150$. For the medium-sized datasets we will use $n_{SS} = 150$ and $n_{DS} = 450$. For the large datasets $n_{SS} = 300$ and $n_{DS} = 900$.

We calculate the metrics for the datasets and repeat this process N times. N is chosen to be equal to 1000.

The metrics will then be compared in three different ways.

- Differentiation: we see how the metrics distinguish between consistent and inconsistent systems. For each dataset, each metric, and each dataset size (small, medium, large), we calculate the metric values for both consistent and inconsistent systems. The goal is to assess how well a given metric can distinguish between consistent and inconsistent systems. A good metric should clearly differentiate between these two types of systems.
- Reliability across dataset type: for each dataset, we focus on the consistent data generated according to this dataset. For a given dataset size, we compare the metric values of the consistent data across all datasets. This means that we are not comparing within a single dataset but rather across different datasets of the same size. Ideally, the metric should evaluate different consistent systems similarly, indicating that the metric is reliable across different datasets of the same size.
- Reliability across dataset size: for each metric and each dataset size, we combine all consistent data of a given size from all datasets. We then compare the metric values of this aggregated consistent data across different dataset sizes. This comparison helps us understand how dependent a given metric is on the size of the dataset, allowing us to determine if the metric behaves consistently across varying dataset sizes.

4 Results

The results will be split up into two parts. In the first part, the results of the optimization of the individual metrics will be discussed. Then, we will discuss the results of the comparison between the different metrics.

4.1. Results of optimization of the metrics

In this section, we will discuss the results of the optimization of the three metrics individually. We have selected three metrics or methods to gain insight into the consistency of LR-systems. For each of these metrics, we have adapted them in different ways to see if we can improve them. We will look at different ways to adjust them and finally select the version of the metrics that works the best, i.e. that makes the best distinction between consistent and inconsistent LR-systems. This will leave us with three optimized metrics, which we will then compare to each other in the next section of this report.

All the results will be shown using violin plots. An example of a violin plot is shown in Figure 4.1. Violin plots are used as a visualization technique to represent the distribution of data. It combines the aspects of boxplots and kernel density plots. Violin plots can be read as follows. The inner part of a violin plot is a boxplot. Hence, the black 'box' shows the interquartile range (IQR), capturing the middle 50% of the data. The white line in the middle of the box represents the median. The lines above and below the box are commonly referred to as 'whiskers'. They extend from the top and bottom of the box to indicate the range of the data distribution beyond which points are considered outliers. The whiskers extend to the minimum and maximum values within 1.5 times the IQR from the lower and upper quartiles, respectively. The shape of the violin represents the kernel density estimate of the distribution. This can be seen by turning the violin on its side. The width of the violin represents the density of the data distribution at different values. Wider sections indicate higher density of data points, and narrower sections indicate lower density.

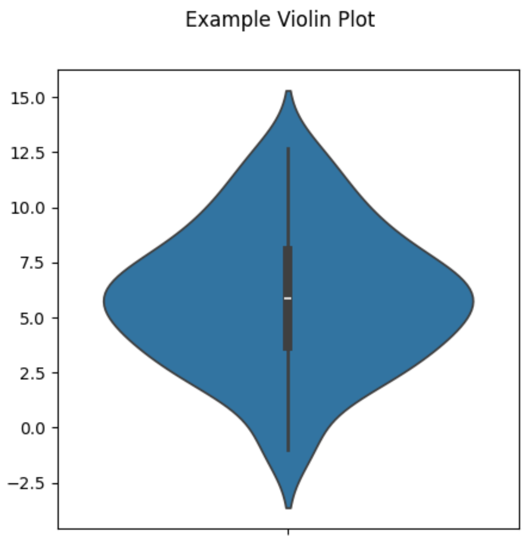

Figure 4.1: Example of a violin plot.

When comparing metrics, we look at the violin plots of the values of metrics for both the inconsistent and the consistent LR-systems. If there is a lot of overlap between two violin plots, it means that the given metric does not distinguish well between those two LR-systems as it attributes similar values to the different systems.

The overlap will be quantified as follows. For each LR-system (the consistent one and all the inconsistent ones), the values corresponding to the fifth and 95th percentiles will be calculated, determining the boundaries of the 90% confidence interval. Now, these boundaries of the consistent LR-system will be compared individually with each of the inconsistent systems in the following manner. The amount of points in the 90% confident of the consistent system that also fall within the confidence bounds of the inconsistent system is determined, as well as the amount of points in the 90% confidence interval of the inconsistent system, that also fall within the confidence bounds of the consistent system. The minimum of these to amounts is taken and divided by the total amount of points within the confidence points, which is the same for both the consistent and inconsistent system. This number is then multiplied by 100 to obtain a percentage.

In formula form, the overlap between two systems is determined as follows:

$$\textbf{Overlap} = \frac{\text{min}\,(P_C, P_I)}{P_T} \times 100, \tag{4.1}$$

where P_C is the number of points within the 90% confidence interval of the consistent LR-system that also fall within the confidence bounds of the inconsistent LR-system, P_I is the number of points within the 90% confidence interval of the inconsistent LR-system that also fall within the confidence bounds of the consistent LR-system, and P_T is the total amount of points in the confidence intervals.

A schematic example of how the overlap percentage is calculated for two datasets is shown in Figure 4.2.

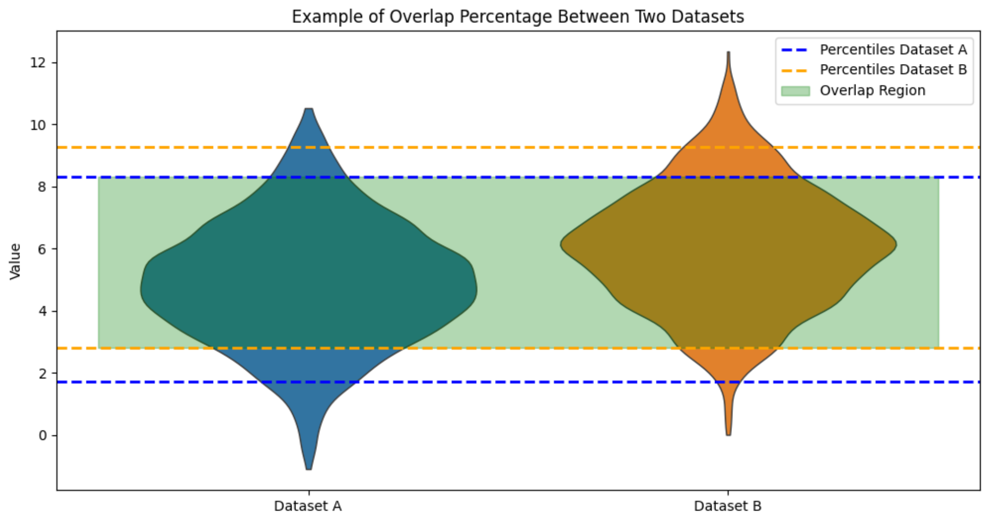

Figure 4.2: Schematic example of overlap percentage of two datasets.

For a given metric, the total overlap is taken to be the average of the overlap percentages of the consistent LR-system with all the inconsistent ones.

4.1.1. Optimization of C_{llr}^{cal}

For the metric C_{llr}^{cal} we have looked into different scoring rules to determine the value of the metric. The original version uses the logarithmic scoring rule, defined in Section 2.3.3. We have compared this with using the Brier score, the spherical score and zero-one loss, all three of which are defined in Section 3.1.1.

The results of the values obtained by using the method explained in Section 3.1 are displayed in the following figures. In these figures, the sizes of the datasets are $n_{SS} = 150$ and $n_{DS} = 450$. In Figure 4.3 the results are shown for using the normal C_{llr}^{cal}, i.e. using the logarithmic scoring rule. Figure 4.4 shows the results when using the Brier scoring rule. In Figure 4.5 the results are shown using spherical scoring rule. Lastly, in Figure 4.6 the results using the zero-one loss are visualized.

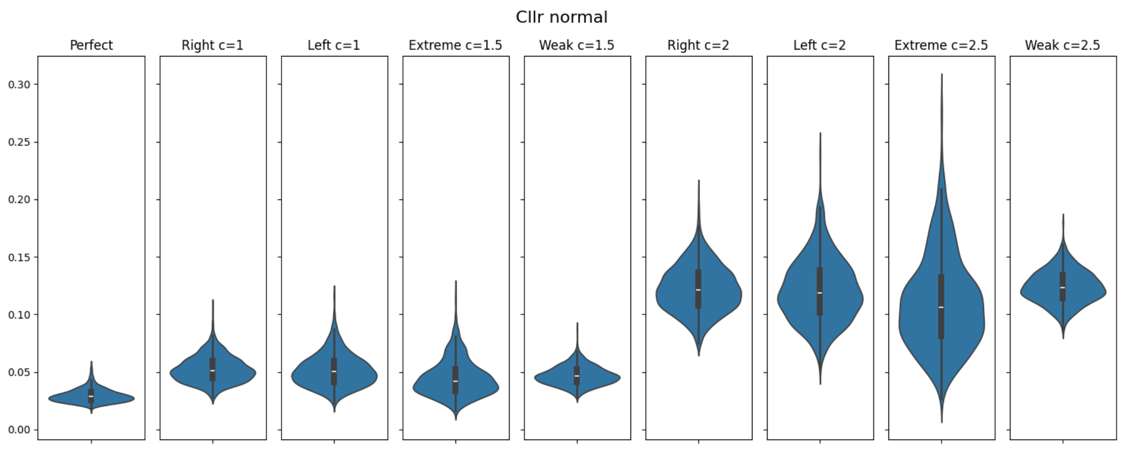

Figure 4.3: Values of normal C_{llr}^{cal} for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

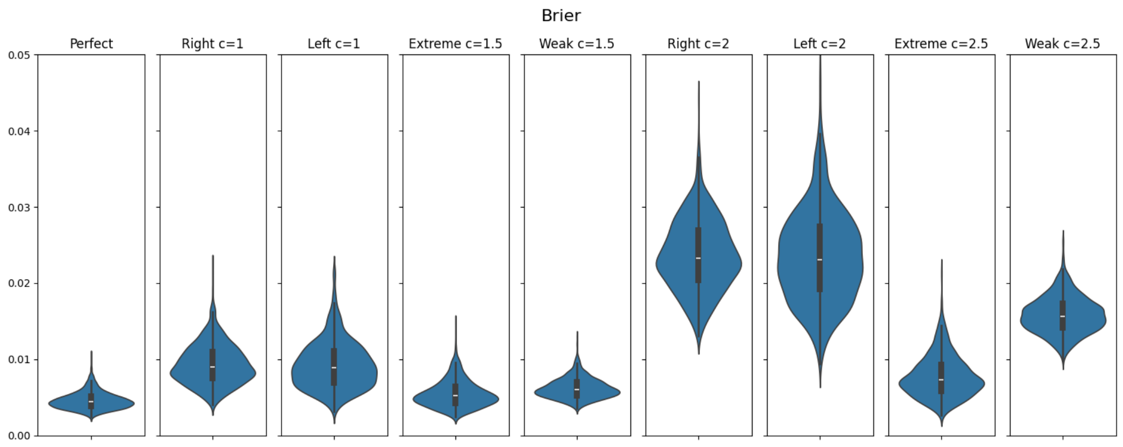

Figure 4.4: Values of C_{llr}^{cal} using the Brier score for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

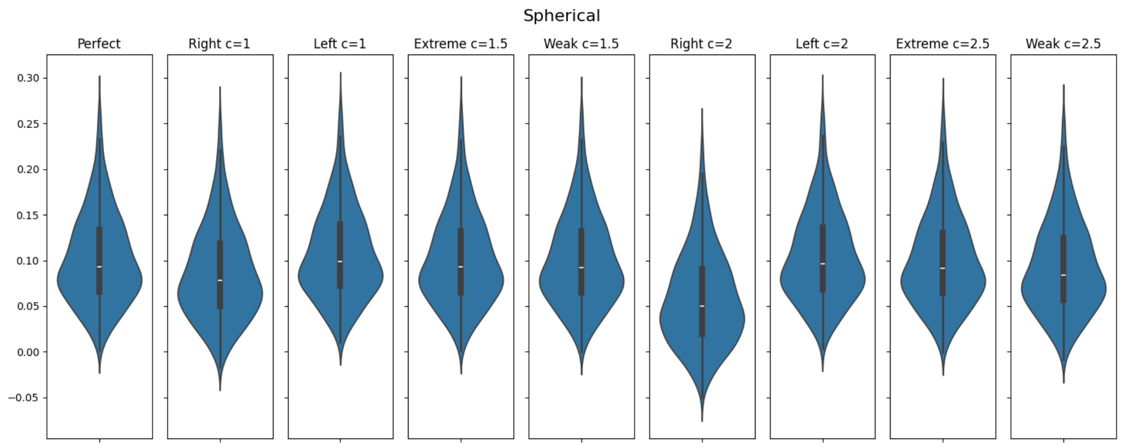

Figure 4.5: Values of C_{llr}^{cal} using the spherical score for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

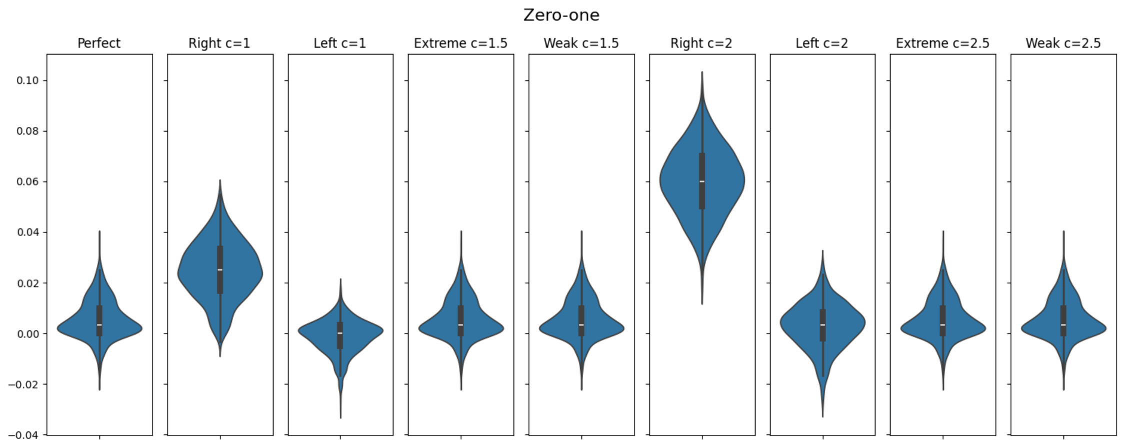

Figure 4.6: Values of C_{llr}^{cal} using zero-one loss for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

As is visible in the figures, the spherical scoring rule does a very poor job distinguishing between the different LR-systems. This can be seen because there is a lot of overlap in the values of the metrics for the given LR-systems. The total overlap percentage is 93.4%, meaning that it barely distinguishes between consistent and inconsistent systems at all. The zero-one loss does slightly better, but only distinguishes for the shifted LR-values, not the more weak and extreme ones. Intuitively, this makes sense: the zero-one loss counts the number of misclassifications, which does not change in the more extreme and more weak systems. This results in an overlap percentage of 74.0%. Note that both for the spherical scoring rule and zero-one loss, the values of the metrics reach values below zero sometimes. This means that sometimes these scoring rules attribute higher scores to the PAV LR-values than to the original ones, even though we know that the PAV LR-values are more consistent than the original ones. This implies that the values can be nonsensical and they do not give us a lot of accurate information.

The normal C_{llr}^{cal} using the logarithmic score and the C_{llr}^{cal} using the Brier score show better performance in distinguishing between the different LR-systems. However, using the logarithmic scoring rule shows slightly better results in the weaker and more extreme systems, resulting in a total overlap of only 10.4% for the normal scoring rule versus 26.1% for the Brier score . Therefore, it is preferred to use the original version of C_{llr}^{cal} over using the Brier score for this dataset size.

It is important to note that C_{llr}^{cal} performs significantly worse if the size of the dataset decreases, both when using the logarithmic and the Brier scoring rule. In Figures 4.7 and 4.8, the results for the two metrics are shown when using $n_{SS} = 50$ and $n_{DS} = 150$.

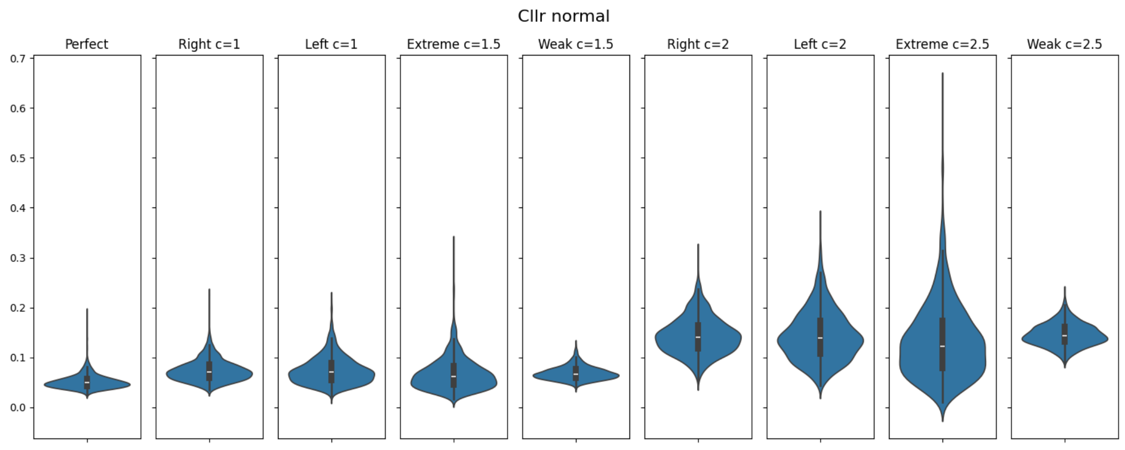

Figure 4.7: Values of normal C_{llr}^{cal} for different LR-systems, using $n_{SS} = 50$ and $n_{DS} = 150$.

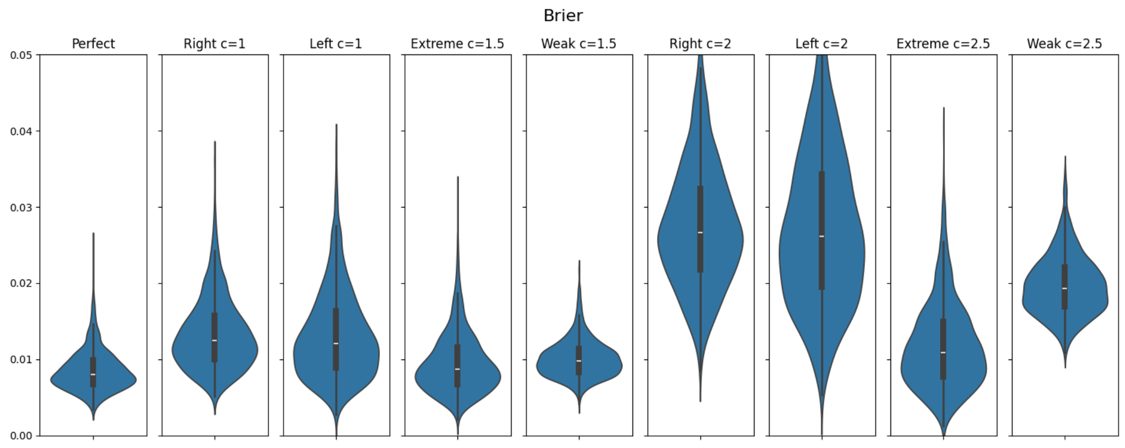

Figure 4.8: Values of C_{llr}^{cal} using the Brier score for different LR-systems, $n_{SS} = 50$ and $n_{DS} = 150$.

It is clear that both of the metrics now do a rather poor job at distinguishing between different systems. The 90% confidence intervals show a lot of overlap. For the normal C_{llr}^{cal}, we find an overlap percentage of 32.9%. As for the version using the Brier score, the overlap is 44.3%. Therefore it is not possible to make a very accurate assessment of consistency using these metrics when the data set is small.

The metric was also tested on a larger dataset. For $n_{SS} = 300$ and $n_{DS} = 900$, the results for C_{llr}^{cal}

using the logarithmic score can be seen in Figure 4.9 and the results for using the Brier score can be seen in Figure 4.10.

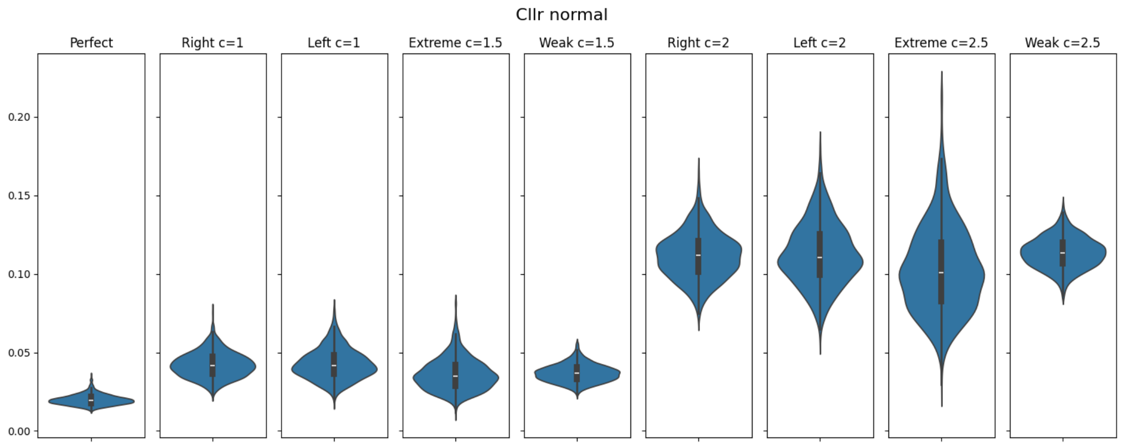

Figure 4.9: Values of normal C_{llr}^{cal} for different LR-systems, using $n_{SS} = 300$ and $n_{DS} = 900$.

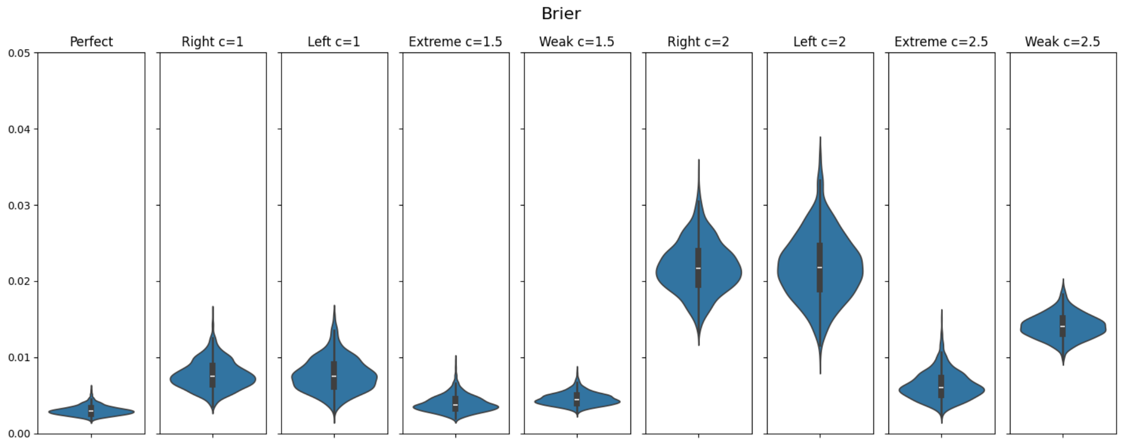

Figure 4.10: Values of C_{llr}^{cal} using the Brier score for different LR-systems, using $n_{SS} = 300$ and $n_{DS} = 900$.

We can see an improvement in the performance of both metrics, especially of the C_{llr}^{cal} using the logarithmic score, which still works better than using the Brier score. The overlap percentages are now 1.4% and 11.3% respectively. Although the metrics clearly improve with the larger dataset, the difference is not as big as we will see later on with the devPAV metric in Section 4.1.2.

Overall, the original C_{llr}^{cal} works best of each dataset size. Therefore, we will continue with this metric in the next comparisons.

4.1.2. Optimization of devPAV

To optimize the devPAV metric, we have looked into different variations, as described in Section 3.1.2. The results of the three metrics when using $n_{SS} = 150$ and $n_{DS} = 450$ are shown in the following figures. In Figure 4.11, we see the results for the original devPAV metric. Figure 4.12 shows the results for the devPAV metric scaled by the total surface. Lastly, Figure 4.13 shows the results for the 'smoothed' version of devPAV where the corners of the steps in the graph are cut off, as can be seen in Section 3.1.2.

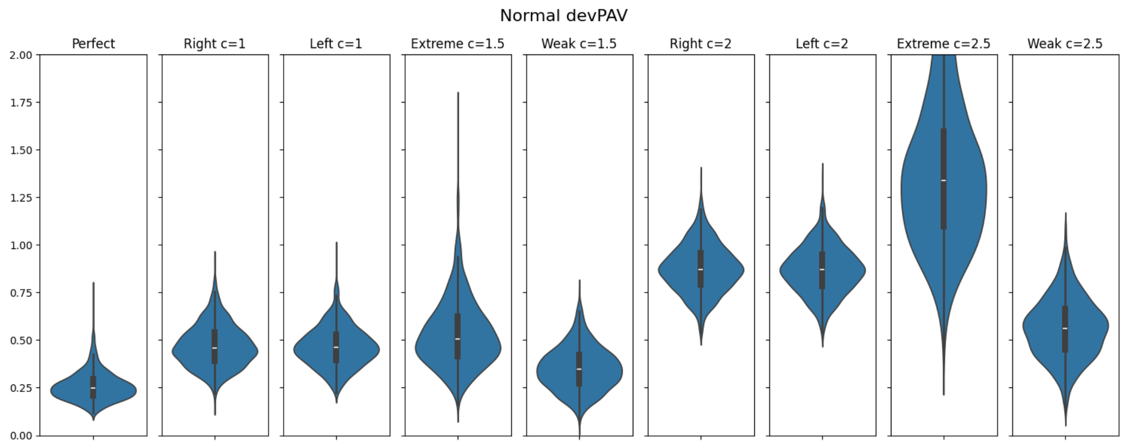

Figure 4.11: Values of the original devPAV metric, using $n_{SS} = 150$ and $n_{DS} = 450$.

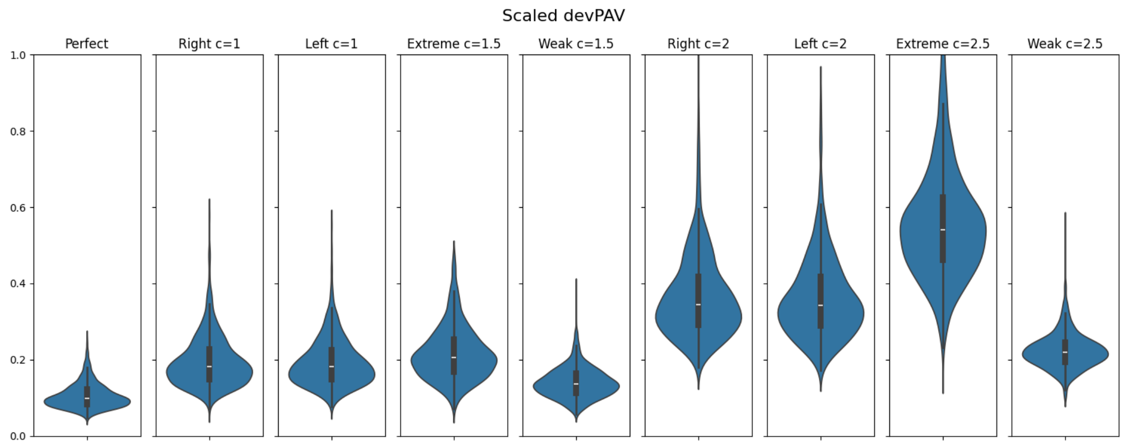

Figure 4.12: Values of the scaled devPAV metric, using $n_{SS} = 150$ and $n_{DS} = 450$.

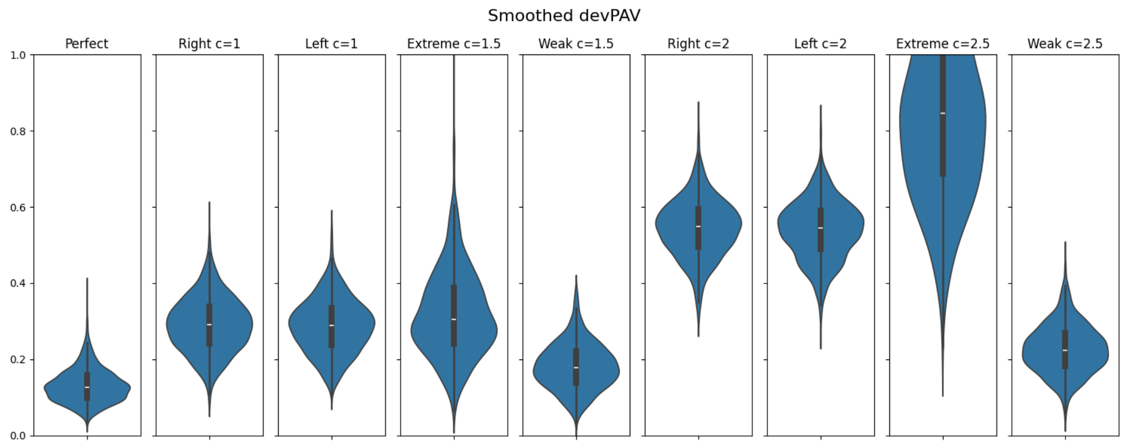

Figure 4.13: Values of the smoothed devPAV metric, using $n_{SS} = 150$ and $n_{DS} = 450$.

From the figures, it can be seen that devPAV distinguishes decently well between different LR-systems for this size of datasets. The distribution of the original devPAV values and the scaled ones are almost identical, only the scaled values are smaller, which is to be expected. This is reflected in the overlap percentage, which is 19.0% for the original devPAV and 18.7% for the scaled version. As for the smoothed devPAV, the distributions are similar as well, although for most systems it makes a slightly better distinction. On the weak systems (W1.5 and W2.5) it does a little bit worse, because by cutting the corners the PAV transformation line comes very close to the identity line, resulting in a small value for the smoothed devPAV. The total overlap percentage for the smoothed devPAV is 18.3%. So on this data with this size, the three devPAVs are very similar performance-wise.

For a smaller dataset, all versions of devPAV perform extremely poorly. This is visualized in Figures 4.14, 4.15 and 4.16.

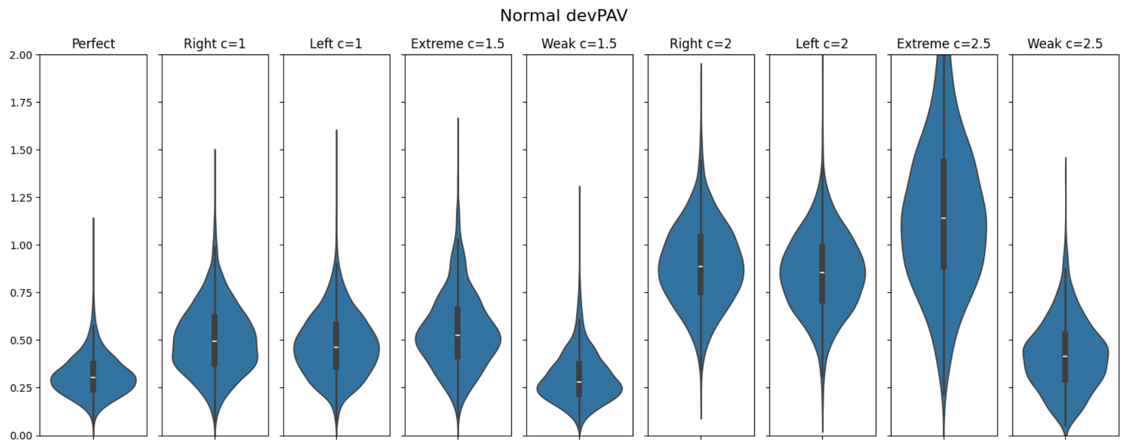

Figure 4.14: Values of the original devPAV metric, using $n_{SS} = 50$ and $n_{DS} = 150$.

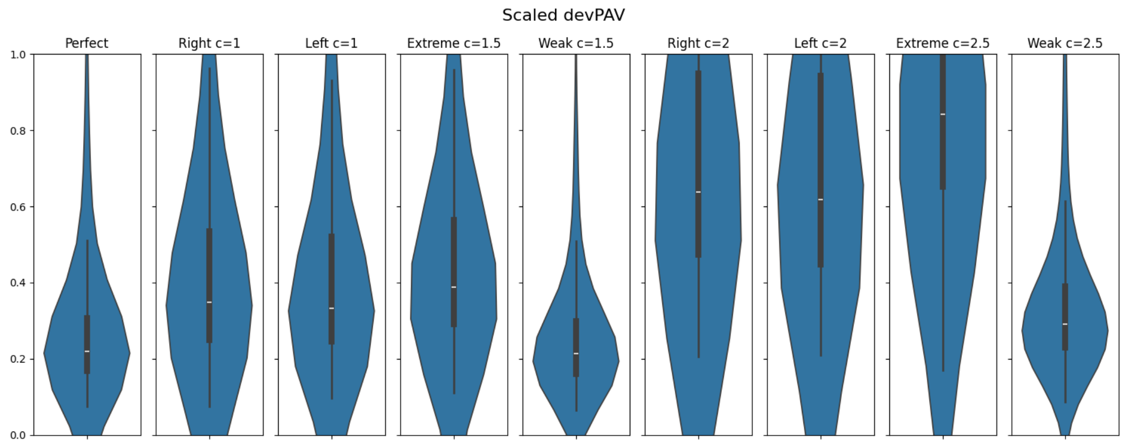

Figure 4.15: Values of the scaled devPAV metric, using $n_{SS} = 50$ and $n_{DS} = 150$.

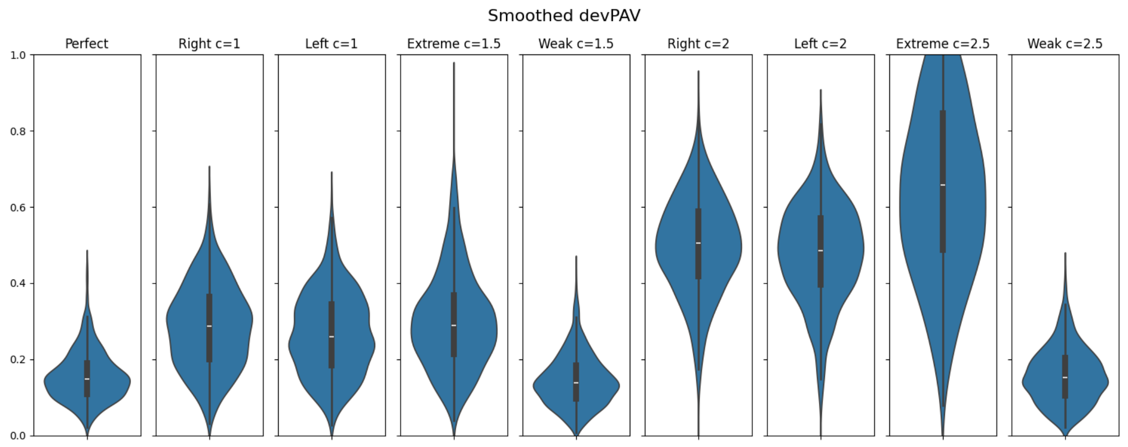

Figure 4.16: Values of the smoothed devPAV metric, using $n_{SS} = 50$ and $n_{DS} = 150$.

Especially the scaled devPAV makes almost no distinction at all between consistent and inconsistent LR-systems. It has an overlap percentage of 58.1%. This is very undesirable. The normal and the smoothed devPAV perform poorly as well, showing a lot of overlap between the different systems. They have overlap percentages of 41.8% and 42.6%, respectively.

With a larger dataset, all versions of devPAV do noticeably better at distinguishing between the consistent and inconsistent LR-systems. The results of using $n_{SS} = 300$ and $n_{DS} = 900$ can be seen in Figures 4.17, 4.18 and 4.19.

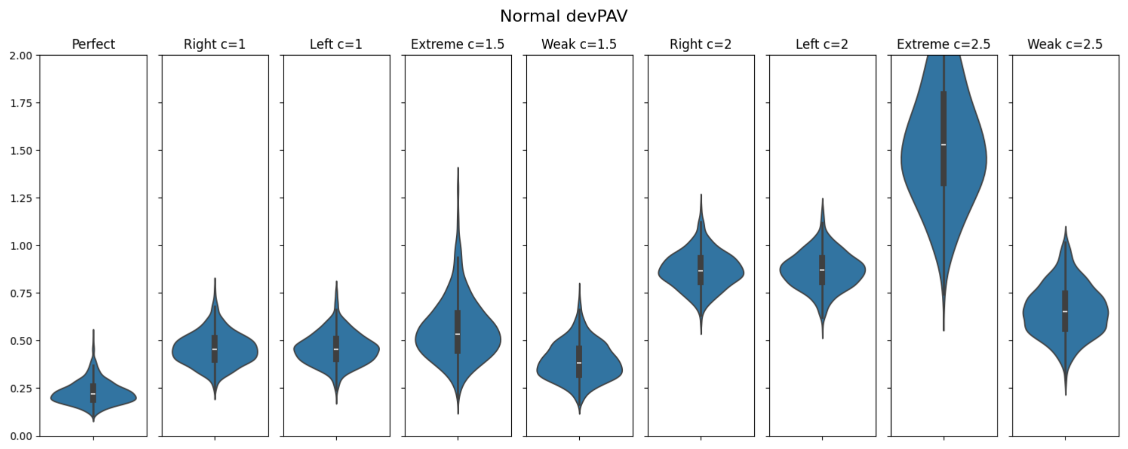

Figure 4.17: Values of the original devPAV metric, using $n_{SS} = 300$ and $n_{DS} = 900$.

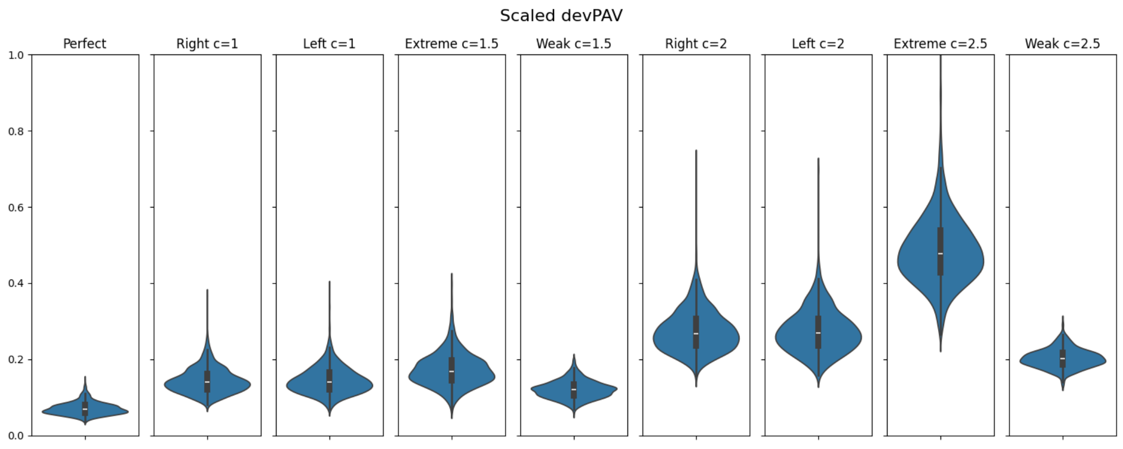

Figure 4.18: Values of the scaled devPAV metric, using $n_{SS} = 300$ and $n_{DS} = 900$.

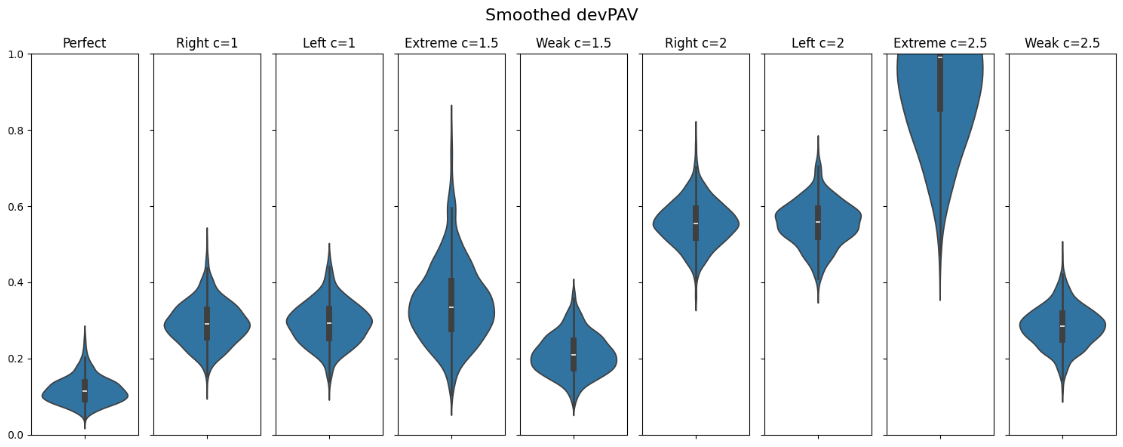

Figure 4.19: Values of the smoothed devPAV metric, using $n_{SS} = 300$ and $n_{DS} = 900$.

For each of the three metrics, there is almost no overlap between the values for the perfect LR-system and the other ones. The overlap percentage of the 90% confidence intervals is 3.9% for the normal devPAV, 2.3% for the scaled version and 3.3% for the smoothed version. All these percentages are very low, which is of course very desirable.

Overall, all three of the devPAV metrics perform quite similarly on the different dataset sizes. The scaled devPAV shows the worst performance, but not by much. The normal version and the smoothed version show almost identical performance. For the rest of this report we will continue with the smoothed version, as this one is expected to be a slightly more robust to different types of datasets. For the rest of the Results section, when we refer to devPAV, we mean the smoothed version, unless specified otherwise.

4.1.3. Optimization of fiducial metric

As described in Section 3.1.3, three possible metrics are considered for the fiducial method. In Figure 4.20, the results are shown for the first metric, taking the average of the medians of the fiducial confidence intervals. In Figure 4.21, the results for taking the average of the medians scaled by the width of the confidence intervals is shown. Lastly, in Figure 4.22 the results are shown when using the zero-one metric.

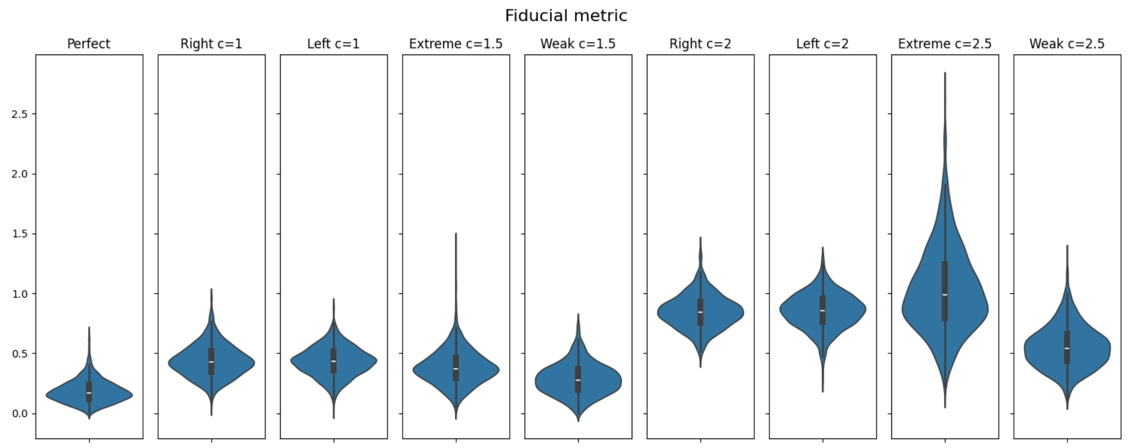

Figure 4.20: Values of the fiducial metric using the average of the medians for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

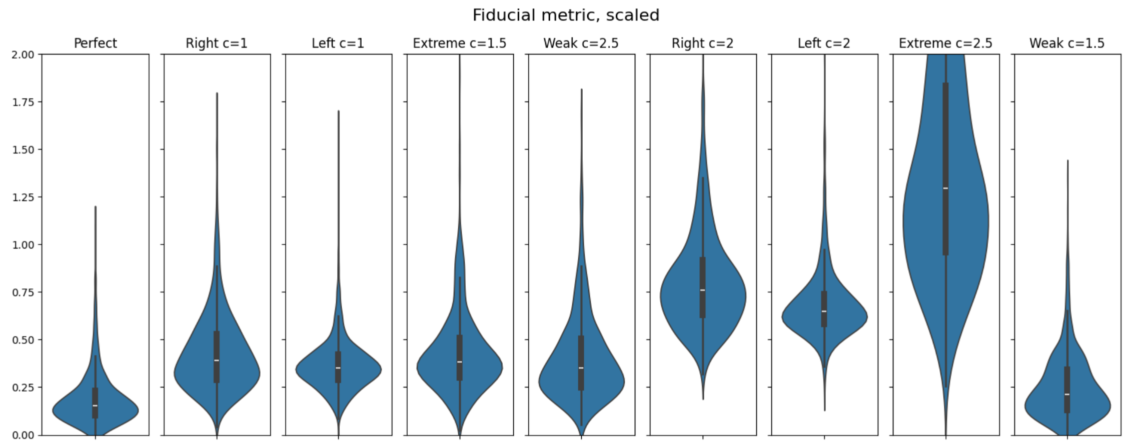

Figure 4.21: Values of the fiducial metric using the scaled metric for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

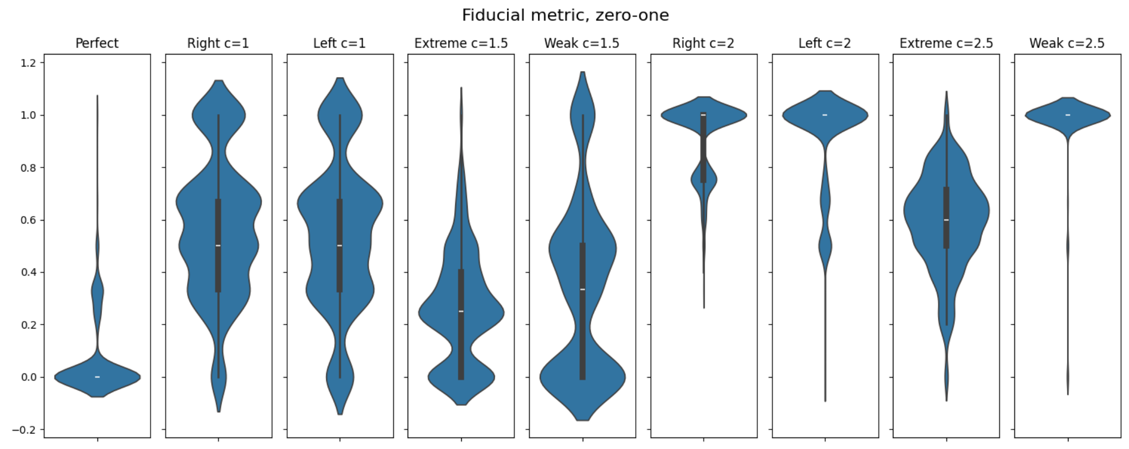

Figure 4.22: Values of the fiducial metric using the zero-one loss for different LR-systems, using $n_{SS} = 150$ and $n_{DS} = 450$.

Firstly it is clear that the zero-one metric gives a rather weird violin-plot. The plots show many bumps. This can be accredited to the discrete characteristic of the metric. There are usually four or five intervals and the times that they don't contain zero is being counted. This gives bumps at exactly those fractions. It can be seen that for the perfect system, they all contain zero most of the time. For the R2, L2 and W2.5 distributions, it seems that all the intervals almost never contain zero. It can be seen from the picture that this metric does not distinguish well between consistent and inconsistent LR-systems. While the lower part of the violin plot for the 'perfect' system lies under the values for most of the other systems, the higher part overlaps with almost every single one. This results in an overlap percentage of 23.8%.

The other two metrics (the average of the medians and the scaled average of the medians) show similar distributions, with the second one being scaled. However, the non-scaled average of medians shows better performance, with an overlap percentage of 20.3%. The scaled average shows a higher overlap percentage of 34.0%.

All metrics perform significantly worse with smaller data sets. Often times, the confidence intervals can not be determined and the metrics do not produce any results at all. When they do show results, the results are inconsistent and show large overlap percentages. Because of the unreliability we exclude any results of the fiducial metrics on the small datasets, both here and in the rest of the thesis.

On a larger dataset, the three metrics all perform better. For the metric using the normal average of the medians, the results are shown in Figure 4.23. It has an overlap percentage of 5.9%. For the scaled average, the results can be seen in Figure 4.24. It has a significantly higher overlap percentage of 19.9%. Lastly, the results of the zero-one metric, that are visualized in Figure 4.25, give us an overlap percentage of 8.3%.

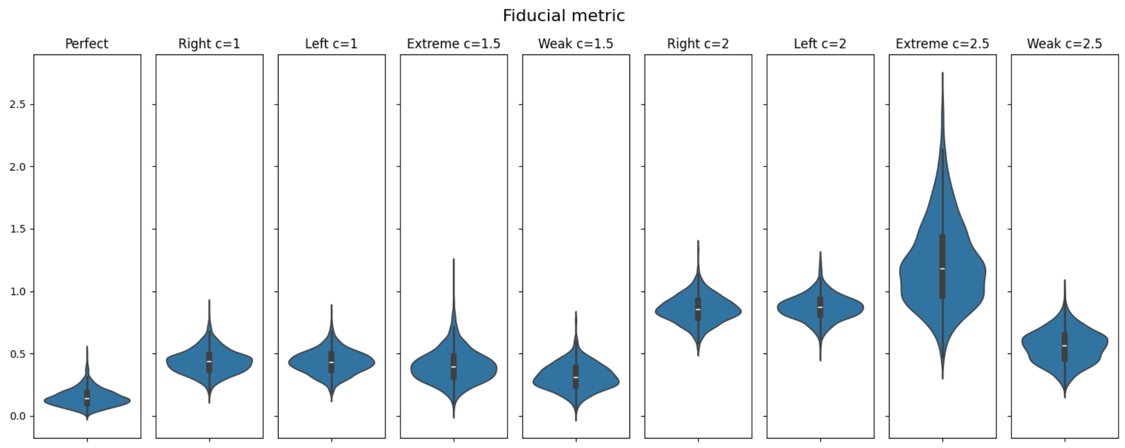

Figure 4.23: Values of the fiducial metric using the average of the medians for different LR-systems, using $n_{SS} = 300$ and $n_{DS} = 900$.

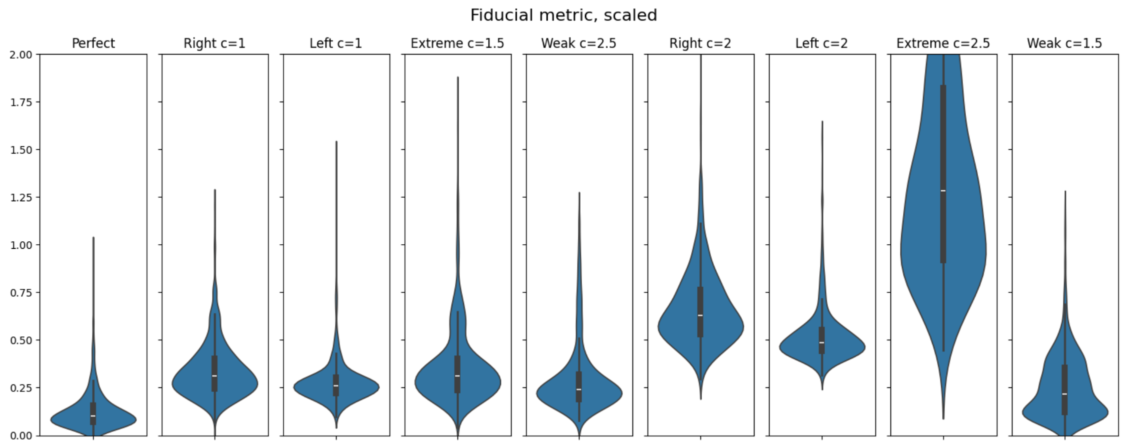

Figure 4.24: Values of the fiducial metric using the scaled metric for different LR-systems, using $n_{SS} = 300$ and $n_{DS} = 900$.

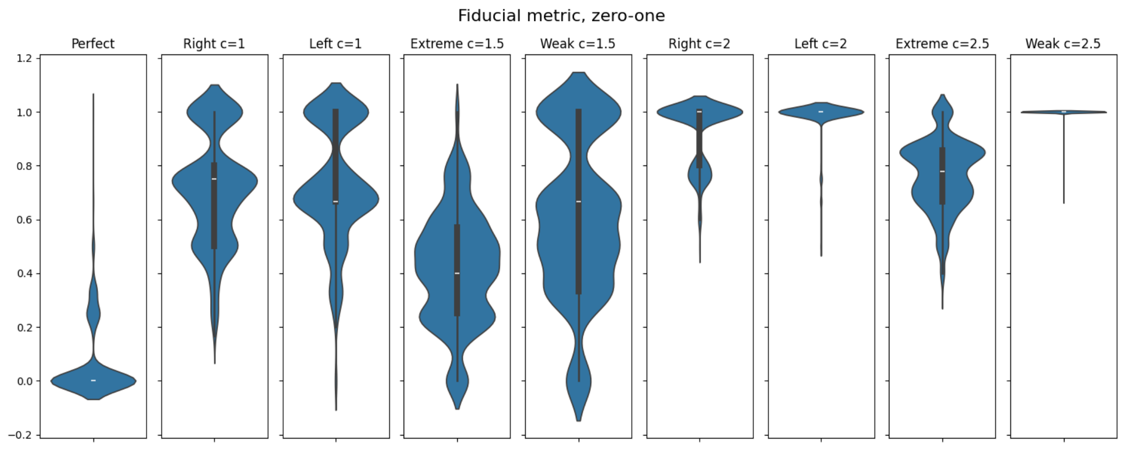

Figure 4.25: Values of the fiducial metric using the zero-one loss for different LR-systems, using $n_{SS} = 300$ and $n_{DS} = 900$.

The simple average of the medians performs significantly better than the other possible metrics. Therefore, this will be the metric used for further comparisons. From now on, this metric will be referred to as 'Fid'.

4.2. Results of comparing the metrics

In this section, we will discuss the results of comparing the different metrics. As mentioned in Chapter 3, the three different metrics (C_{llr}^{cal}, devPAV and Fid) will be compared on eight different datasets to see how well they distinguish between consistent and inconsistent LR-systems, how they are affected by the size of the dataset and how reliable they are over different datasets. Part of this has already been done in Section 3.1: we have already seen how the metrics differentiate between consistent and inconsistent LR-systems when the systems are based on normally distributed LR-data. However, in this section we will look at differently distributed data, while keeping the normally distributed data as a reference point. The distributions we will look at are not known distributions. Rather, we use the distribution of real-life different-source LR-systems.

4.2.1. Consistent versus inconsistent data

In this section the results will be presented on how well the different metrics distinguish between consistent and inconsistent datasets. This will be done using the overlap percentages presented in Section 4.1. For illustration purposes, we will show the procedure in detail using one dataset, specifically Data 3. This dataset consists of 188 same-source LR's and 1327 different-source LR's. The same calculations apply to the remaining datasets presented in Section 3.2.1; hence, we will only present their final results in the table without going into each step.

The distribution of the original same- and different-source data is shown in a histogram in Figure 4.26.

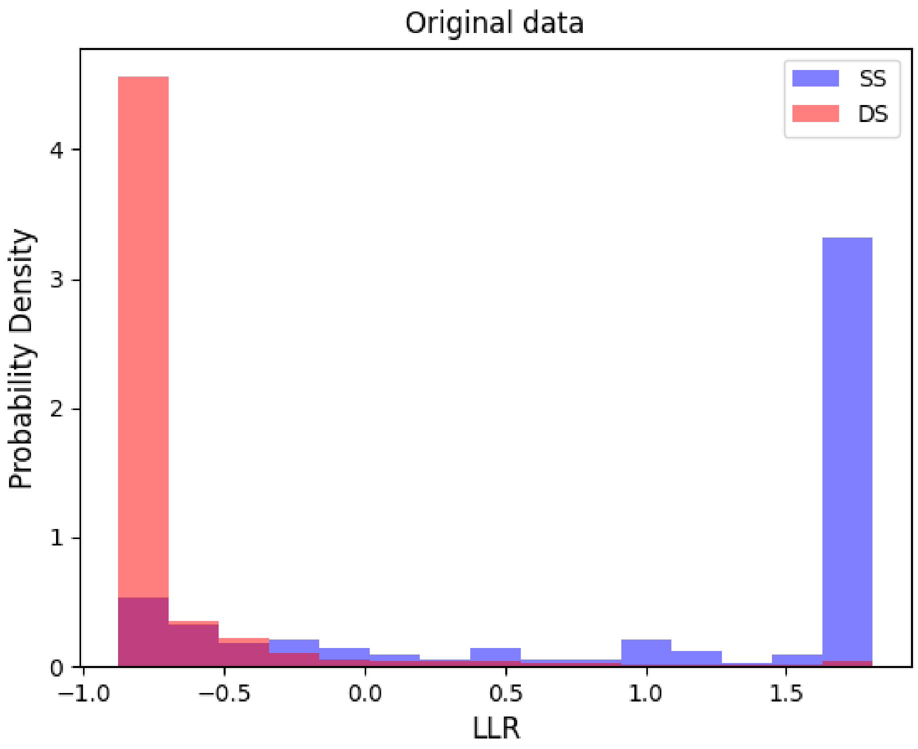

Figure 4.26: Distribution of the 188 same-source and 1327 different-source LR's of Data 3.

Note that the ELUB bounds are applied to this dataset, which results in peaks at the far-right and far-left bins. Values larger than the right bound are moved to the far-right bin, while smaller values than the left bound are moved to the far-left bin.

As explained in Section 3.2, we only use the different-source LR-data from the dataset, which we use to generate new same-source data, thereby ensuring a consistent LR system. This is achieved by applying a kernel density estimate (KDE) to the distribution of the different-source LR's. For the bandwidth selection of the KDE, we experimented with various methods across the datasets. The objective was to get a smooth estimate without losing too much information. Additionally, we aimed to determine a single method that could be consistently applied to all datasets, eliminating the need for manual selection each time. We have experimented with the Silverman bandwidth, the Scott bandwidth and manually changed both to be wider. We have also experimented with the Sheather-Jones plug-in bandwidth. While the optimal bandwidth varied for each dataset, the Scott bandwidth performed the best on average. It provided a good balance between smoothing the distributions and retaining information.

The resulting KDE is illustrated in Figure 4.27 below.

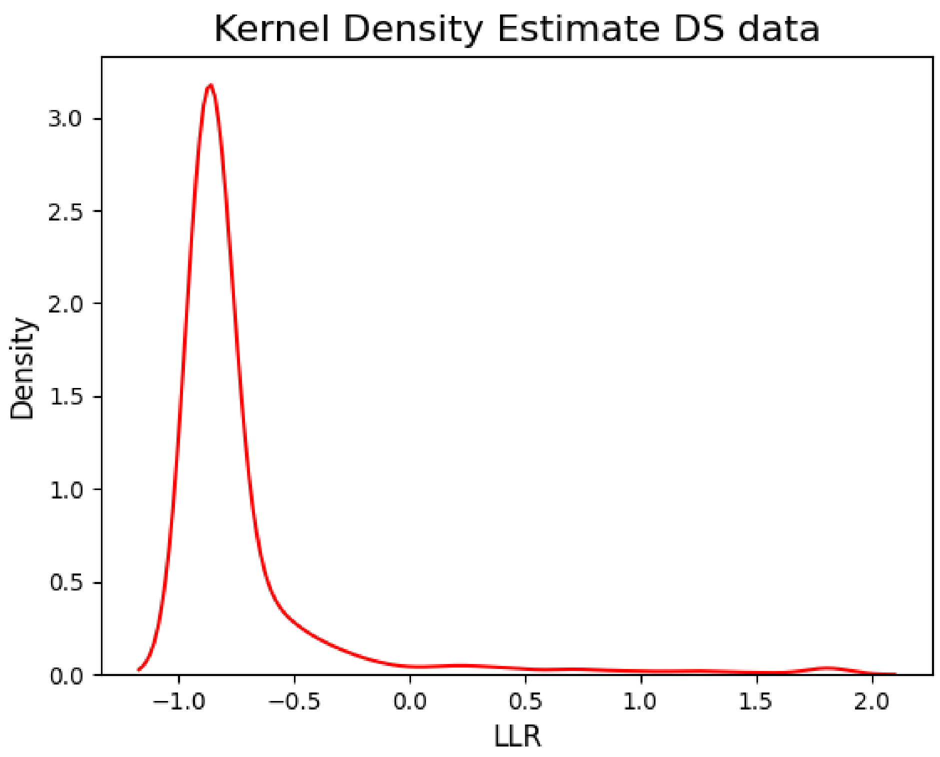

Figure 4.27: Kernel density estimate applied to distribution of different-source LR's.

As explained in Section 3.1, we can now generate a corresponding distribution for the same-source data so that Equation 2.7 holds. However, we also want the surfaces under the two curves to be close to equal. So we stretch the KDE from Figure 4.27 out to the left or right, depending on which area under the curve is larger, to generate new curves until we have the one that meets the criteria, namely that Equation 2.7 holds and that the same-source and different-source curve have the same area under the curve. This results in the same-source and different-source distributions shown in Figure 4.28

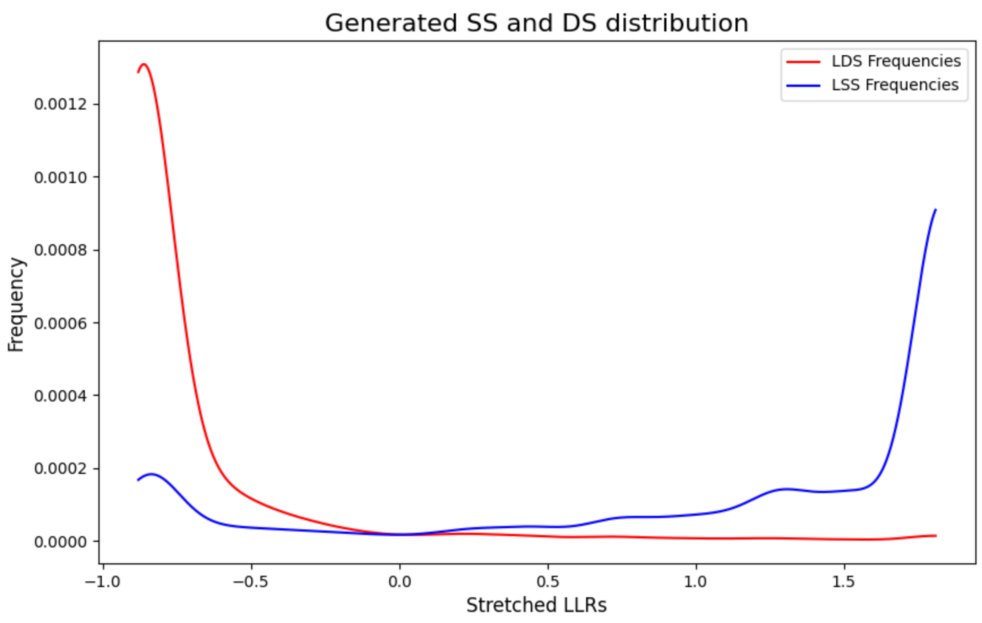

Figure 4.28: Generated SS and DS distributions.

It can be seen that the two curves cross each other at LLR-values of 0 (or equivalently LR-values of 1), which is always the case for a consistent LR-system. Left from 0, the DS-frequencies should always be higher than the SS-frequencies, and vice versa on the right side of 0.

Now, n_{SS} LR-values are sampled from the SS distribution given by the blue curve, and n_{DS} LR-values are sampled from the DS distribution given by the red curve. An example of a dataset generated according to these distributions is shown in Figure 4.29.

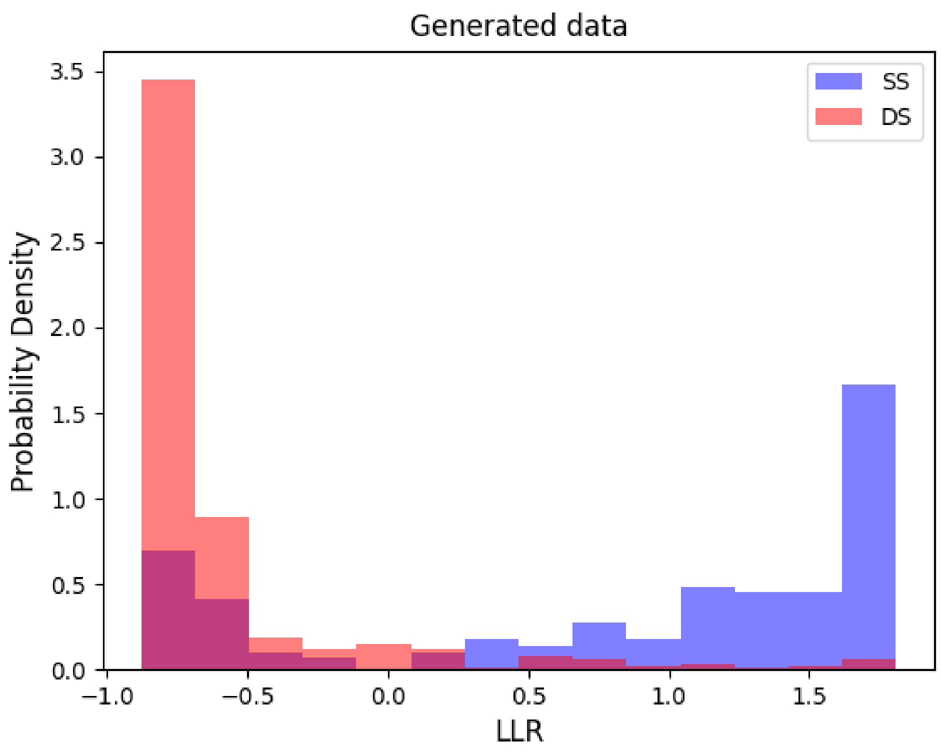

Figure 4.29: Generated consistent data according to SS and DS distributions.

These values form our consistent LR-system. The metrics are then calculated for this consistent LR-system, as well as for the LR-systems we obtain by shifting them and scaling them, as described in Section 3.2.3. This process is then repeated 1000 times for each set of values for n_{SS} and n_{DS}.

Just like before, we start with the medium-sized datasets: for $n_{SS} = 150$ and $n_{DS} = 450$, the results are shown in Figures 4.30, 4.31 and 4.32.

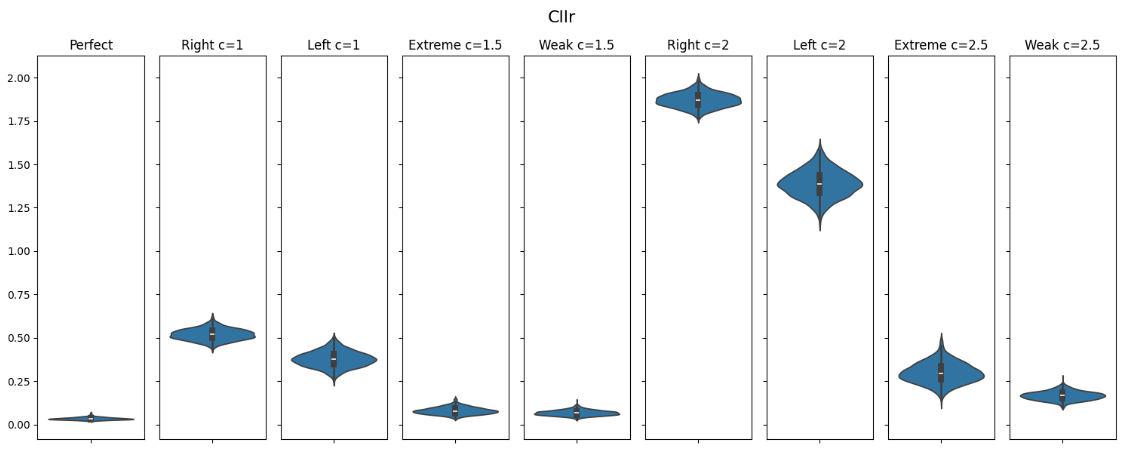

Figure 4.30: Values of C_{llr}^{cal} for $n_{SS} = 150$ and $n_{DS} = 450$.

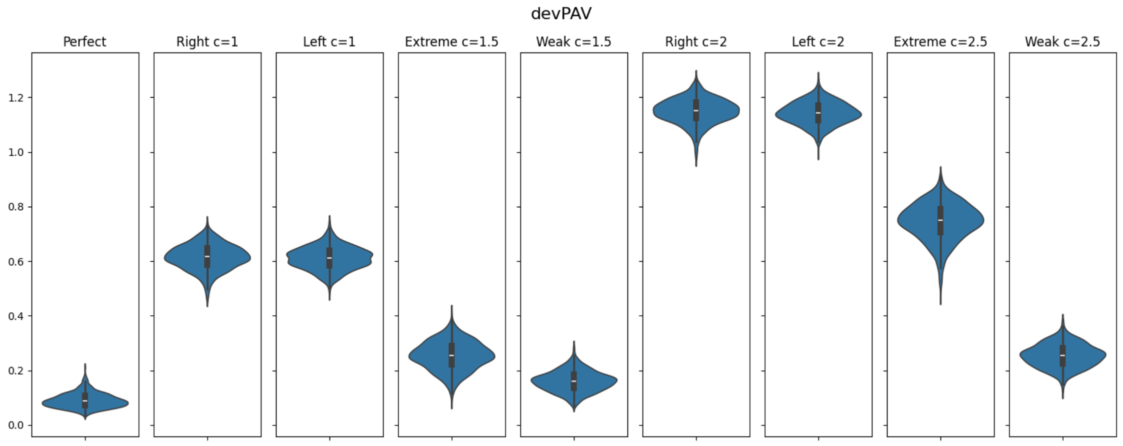

Figure 4.31: Values of devPAV for $n_{SS} = 150$ and $n_{DS} = 450$.

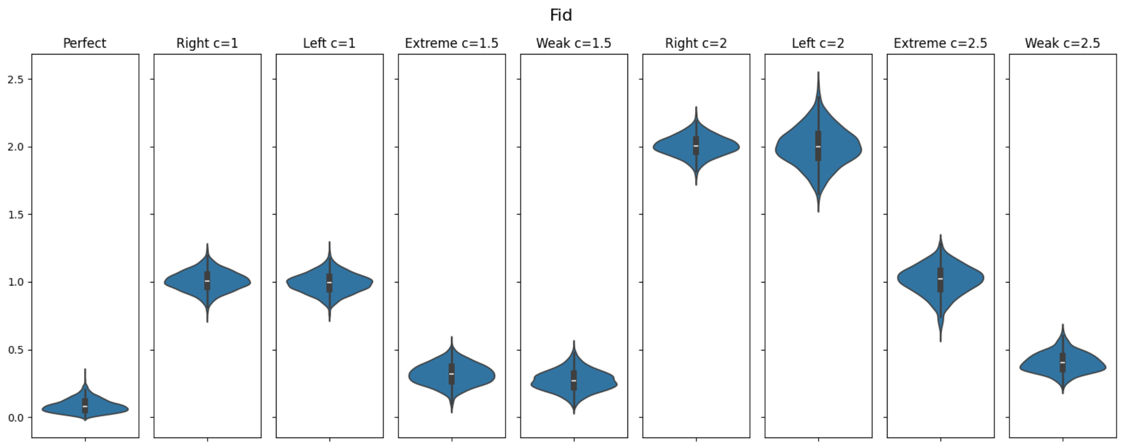

Figure 4.32: Values of Fid for $n_{SS} = 150$ and $n_{DS} = 450$.

For this dataset size, the overlap percentages are the following:

- C_{llr}^{cal}: 0.3%,
- devPAV: 3.4%,
- Fid: 0.8%.

It can be seen that C_{llr}^{cal} does the best job at distinguishing between consistent and inconsistent LR-systems, followed rather closely by Fid. devPAV does the worst, but not by much. All three metrics give very good results for this dataset size.

For the small dataset with $n_{SS} = 50$ and $n_{DS} = 150$, the results are shown in Figures 4.33 and 4.34 for C_{llr}^{cal} and devPAV, respectively. Unfortunately, Fid does not work consistently on small datasets so we will exclude any results for Fid on datasets of this size.

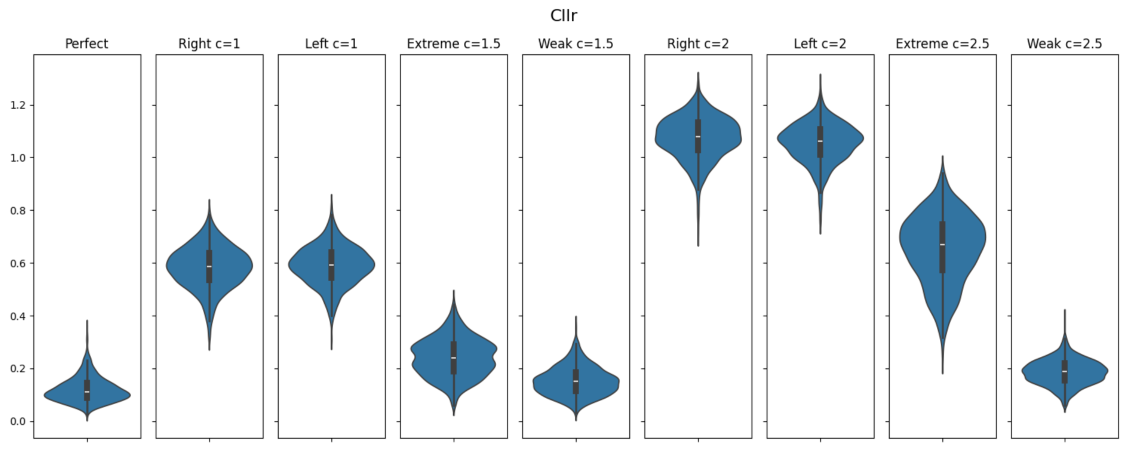

Figure 4.33: Values of C_{llr}^{cal} for $n_{SS} = 50$ and $n_{DS} = 150$.

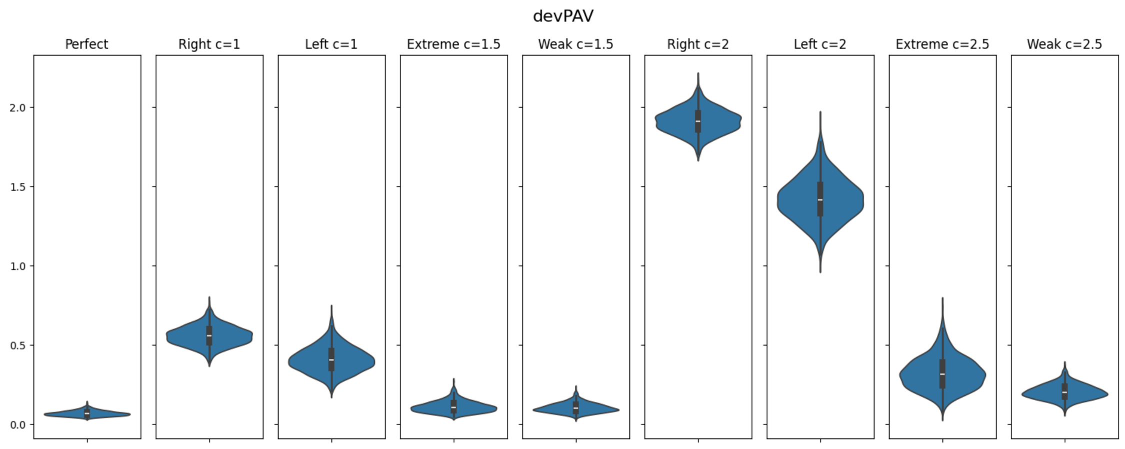

Figure 4.34: Values of devPAV for $n_{SS} = 50$ and $n_{DS} = 150$.

The overlap percentages are as follows:

- C_{llr}^{cal}: 21.9%,
- devPAV: 11.3%,
- Fid: -.

So it can be concluded that for this dataset with $n_{SS} = 50$ and $n_{DS} = 150$, devPAV performs better than C_{llr}^{cal} at distinguishing between consistent and inconsistent LR-systems. Overall, both of the metrics perform worse on the smaller dataset than on the medium-sized one, as expected.

Lastly, for $n_{SS} = 300$ and $n_{DS} = 900$, the results are shown in Figures 4.35, 4.36 and 4.37.

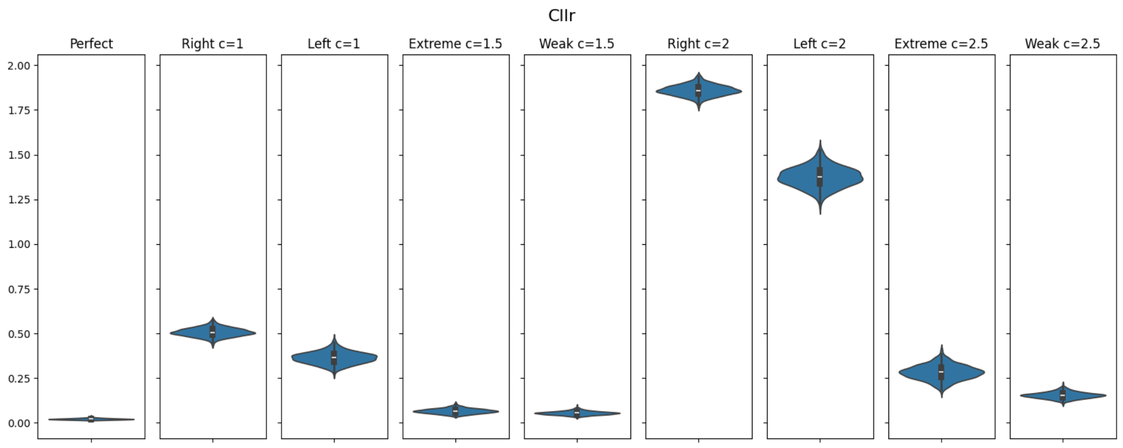

Figure 4.35: Values of C_{llr}^{cal} for $n_{SS} = 300$ and $n_{DS} = 900$.

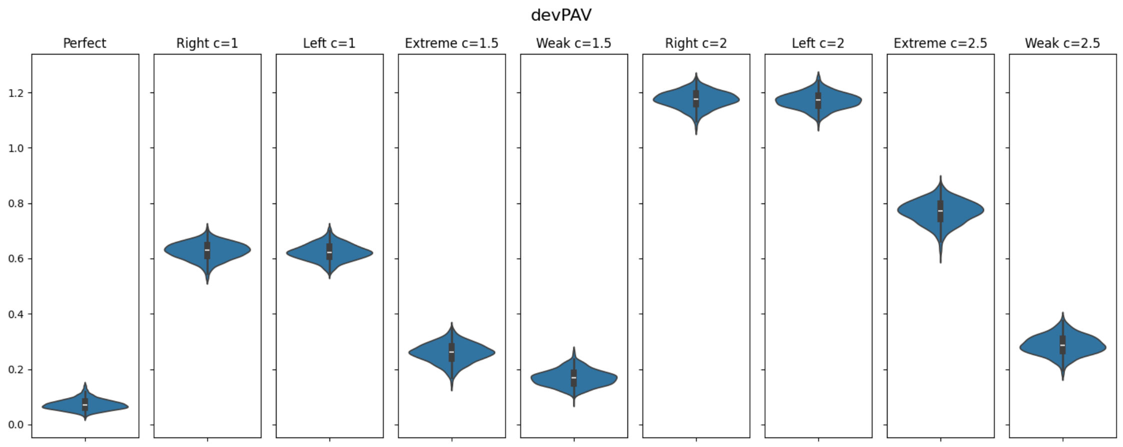

Figure 4.36: Values of devPAV for $n_{SS} = 300$ and $n_{DS} = 900$.

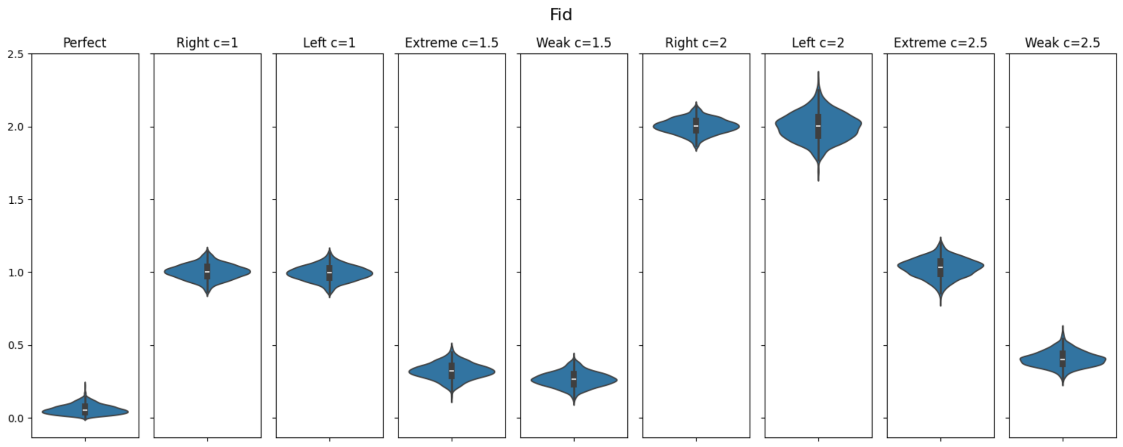

Figure 4.37: Values of Fid for $n_{SS} = 300$ and $n_{DS} = 900$.

This dataset size results in the following overlap percentages:

- C_{llr}^{cal}: 0.0%,
- devPAV: 0.0%,

- Fid: 0.0%.

As expected, the performance of each metric improves with the enlargement of the dataset size. All three of the metrics now show zero percent overlap, meaning that they make a perfect distinction between consistent and inconsistent LR-systems at this dataset size.

This procedure is performed for every one of the datasets, resulting in different overlap percentages.

Tables 4.1, 4.2 and 4.3 present the overlap percentages for the various data sets under the different metrics.

Medium	C_{llr}^{cal}	devPAV	Fid
Data 1	0.8	3.8	1.2
Data 2	2.2	25.0	14.8
Data 3	0.3	3.4	0.8
Data 4	1.8	25.9	12.8
Data 5	4.3	19.3	15.4
Data 6	0.5	1.8	6.8
Data 7	4.2	6.4	30.3
Normal data	10.4	18.3	20.3

Table 4.1: Values of metrics for consistent LR-systems based on different datasets, for $n_{SS} = 150$ and $n_{DS} = 450$.

Small	C_{llr}^{cal}	devPAV	Fid
Data 1	11.9	24.0	-
Data 2	15.1	36.8	-
Data 3	21.4	11.3	-
Data 4	15.0	37.8	-
Data 5	14.8	34.1	-
Data 6	12.6	19.7	-
Data 7	21.7	29.5	-
Normal data	32.9	42.6	-

Table 4.2: Values of metrics for consistent LR-systems based on different datasets, for $n_{SS} = 50$ and $n_{DS} = 150$.

From Table 4.1 it can be deduced that for $n_{SS} = 150$ and $n_{DS} = 450$ on average, C_{llr}^{cal} shows an overlap percentage of 3.1%. For devPAV and Fid, this average is slightly higher with 13.0% and 12.8% respectively.

For the datasets of sizes $n_{SS} = 50$ and $n_{DS} = 150$, the averages of the overlap percentages are higher, as can be seen in Table 4.2. This is in line with the expectation: on smaller datasets, we expect the metrics to perform worse. For C_{llr}^{cal} it goes up by 15.1%, resulting in an overlap percentage of 18.2%. For devPAV the overlap percentage is 29.1%, which is 16.1% higher than for the medium-sized datasets. For Fid, we don't have any results for this dataset size.

Lastly, for the datasets of sizes $n_{SS} = 300$ and $n_{DS} = 900$, the averages are the lowest, shown in Table 4.3. The average overlap percentage of C_{llr}^{cal} drops by 2.8% compared to the medium-sized

Large	C_{llr}^{cal}	devPAV	Fid
Data 1	0.0	0.0	0.0
Data 2	0.0	13.3	9.1
Data 3	0.0	0.0	0.0
Data 4	0.0	12.8	7.4
Data 5	1.1	4.8	5.2
Data 6	0.0	0.0	0.0
Data 7	0.0	1.3	18.7
Normal data	1.4	3.3	5.9

Table 4.3: Values of metrics for consistent LR-systems based on different datasets, for $n_{SS} = 300$ and $n_{DS} = 900$.

dataset, giving an overlap percentage of 0.3% for this dataset size. For devPAV it drops by 8.6%, resulting in an overlap percentage of 4.4%. For Fid, the average overlap percentage is the highest by just a bit with 5.8%, dropping by 7.0% from the medium-sized dataset.

Altogether, we conclude that C_{llr}^{cal} does the best job at distinguishing between consistent and inconsistent datasets. It outperforms the other two metrics at every dataset size that we have tested.

4.2.2. Reliability of the metrics across datasets

In this section, the results will be presented on how consistent the values of the metrics are across consistent LR-systems based on different datasets. Instead of speaking of consistency, we will speak of reliability, to avoid any confusion with the consistency of LR-systems. So reliability of a metric is defined as its capability to to assign similar values to different consistent LR-systems. Specifically, for each of the seven datasets introduced in Section 3.2.1, as well as for the normally distributed data, we will evaluate the metrics for the consistent LR-systems generated from these datasets. We will determine whether each metric provides similar values across the different consistent LR-systems, indicating its reliability. This reliability is desirable because if we know a metric distinguishes well between consistent and inconsistent LR-systems and it is reliable in the values it gives for consistent LR-systems, then we can confidently use it to assess the consistency of new LR-systems based on the value the metric assigns to it.

The seventh dataset is excluded in this section due to the computational cost.

As explained in Section 3.2.3, we will test the reliability of the metrics on the consistent LR-systems generated using the different datasets. We will compare the distributions of the values of the metrics across the consistent LR-systems. We do this for each dataset size and each metric. Again, we look at an overlap percentage, but note that in this case, we want the overlap to be high, i.e. we want the metrics to assign similar values to different consistent LR-systems. The overlap percentage we use now is slightly different from the overlap percentage previously used in the following manner: we will not compare one distribution to all the others, but we will compare each pair and take the average overlap. This will become more clear from looking at the results.

In Figures 4.38, 4.39 and 4.40, the results when using $n_{SS} = 150$ and $n_{DS} = 450$ are visualized for the three metrics. Specifically, in Figure 4.38, the distribution of the value of C_{llr}^{cal} is shown per consistent LR-system based on a given dataset, specified above each column. For devPAV and Fid, these distributions are shown in Figures 4.39 and 4.40.

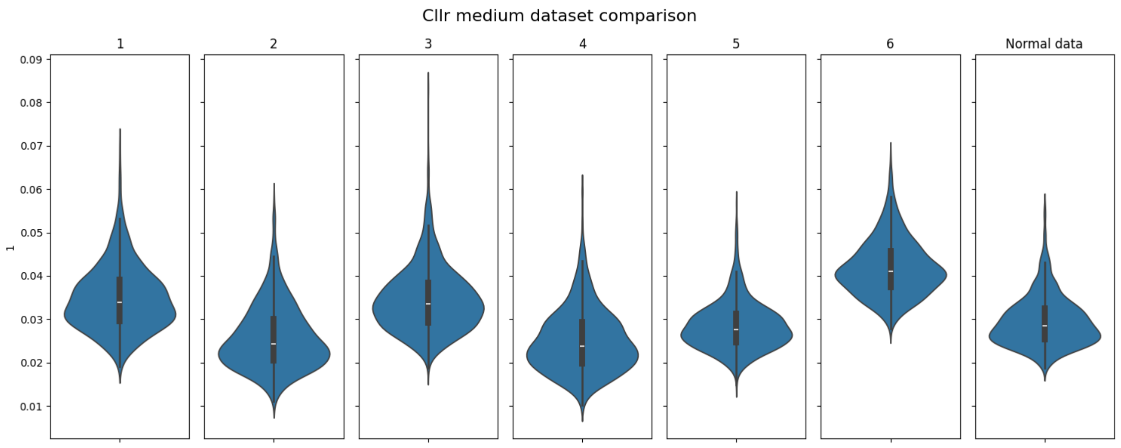

Figure 4.38: Values of C_{llr}^{cal} across datasets for $n_{SS} = 150$ and $n_{DS} = 450$.

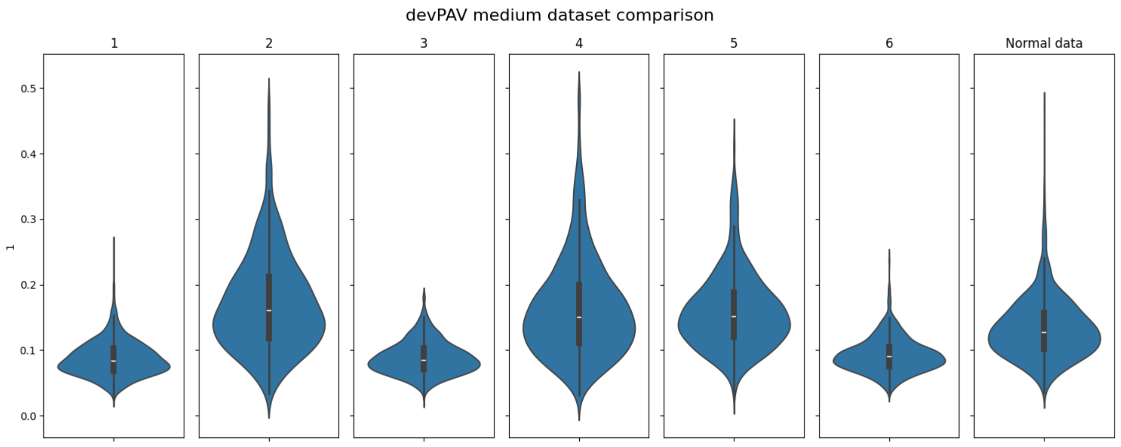

Figure 4.39: Values of devPAV across datasets for $n_{SS} = 150$ and $n_{DS} = 450$.

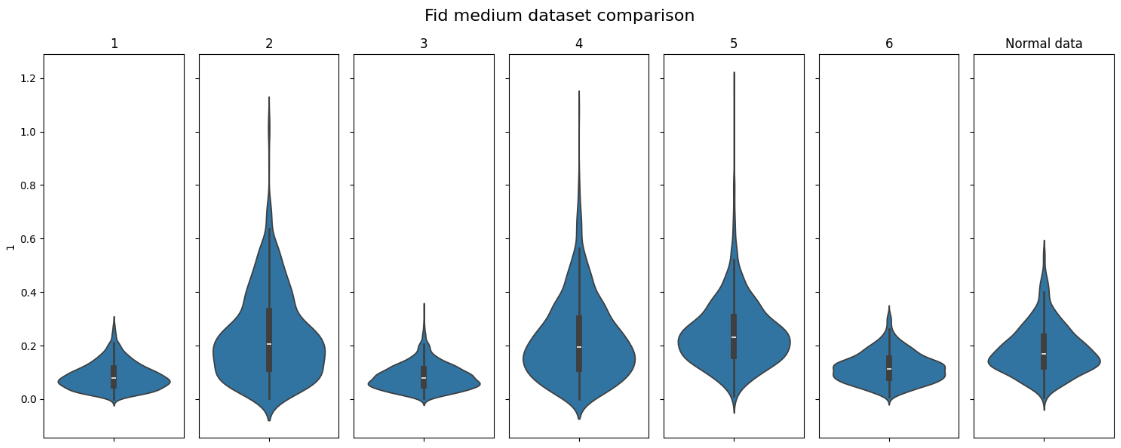

Figure 4.40: Values of Fid across datasets for $n_{SS} = 150$ and $n_{DS} = 450$.

Now, as previously mentioned, the overlap percentages are not taken just from the first column with the other columns, as we did before, but for each pair of columns, and then the average is taken. This dataset size results in overlap percentages of 59.1% for C_{llr}^{cal}, 63.0% for devPAV and 64.3% for Fid.

Recall that we want the overlap percentage here to be high, implying reliability of the metrics across various consistent datasets. It follows that here, Fid outperforms C_{llr}^{cal} and devPAV by a little bit.

In Figures 4.41 and 4.42, the results are shown using dataset sizes $n_{SS} = 50$ and $n_{DS} = 150$. For Fid, we do not have results for this dataset size, as explained in Section 4.1.3.

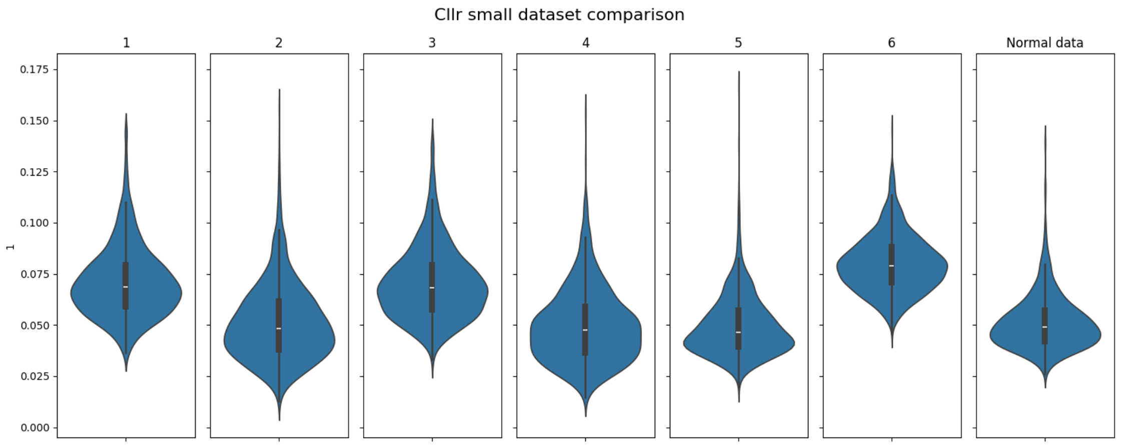

Figure 4.41: Values of C_{llr}^{cal} across datasets for $n_{SS} = 50$ and $n_{DS} = 150$.

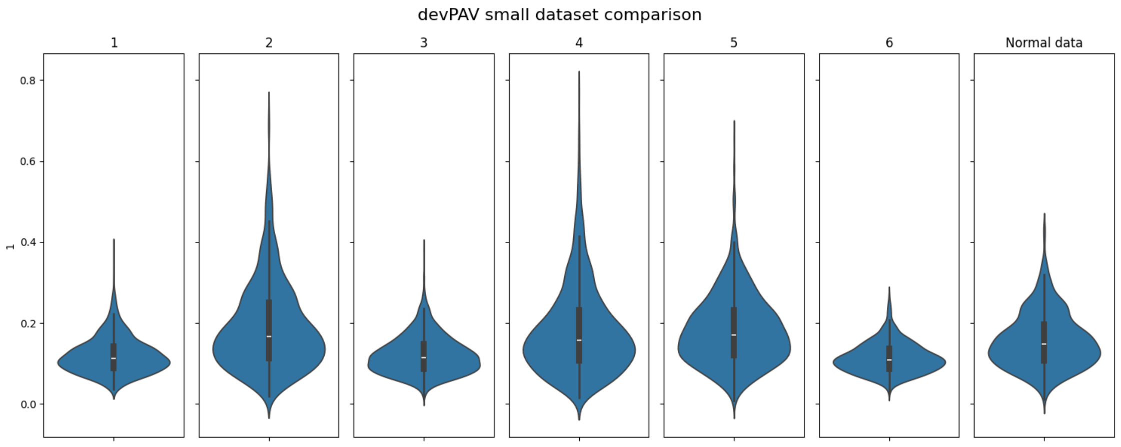

Figure 4.42: Values of devPAV across datasets for $n_{SS} = 50$ and $n_{DS} = 150$.

For this dataset size, we find an overlap percentage of 60.3% for C_{llr}^{cal}, and 77.8% for devPAV. The overlap percentages of C_{llr}^{cal} and devPAV have increased compared to the medium-sized datasets.

The results for $n_{SS} = 300$ and $n_{DS} = 900$ can be seen in Figures 4.43, 4.44 and 4.45.

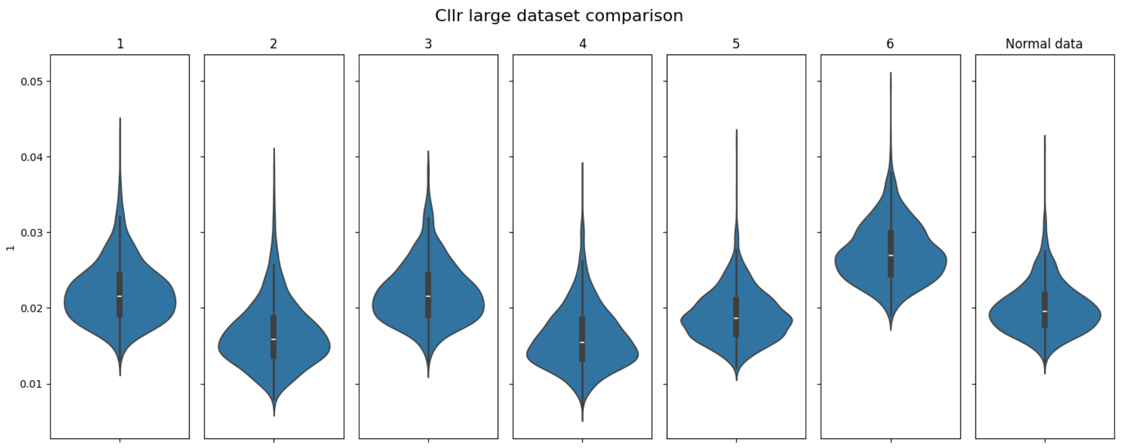

Figure 4.43: Values of C_{llr}^{cal} across datasets for $n_{SS} = 300$ and $n_{DS} = 900$.

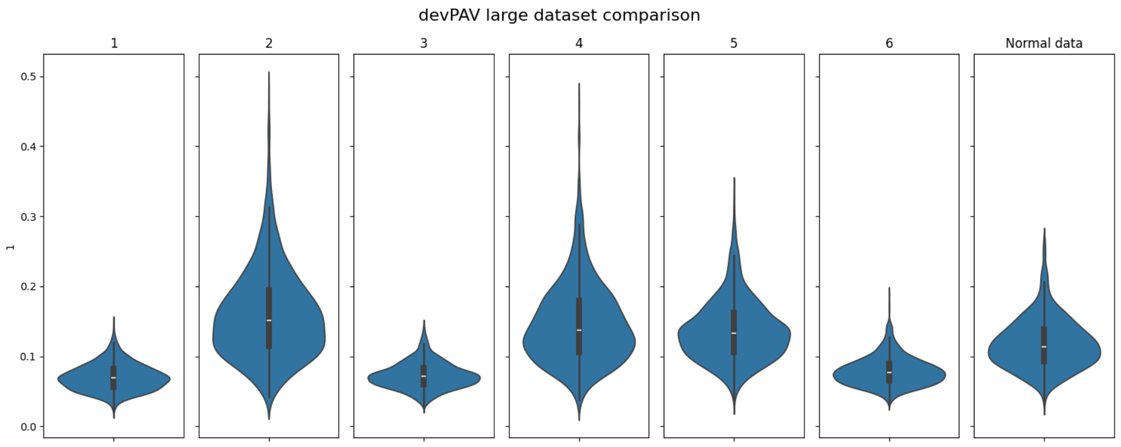

Figure 4.44: Values of devPAV across datasets for $n_{SS} = 300$ and $n_{DS} = 900$.

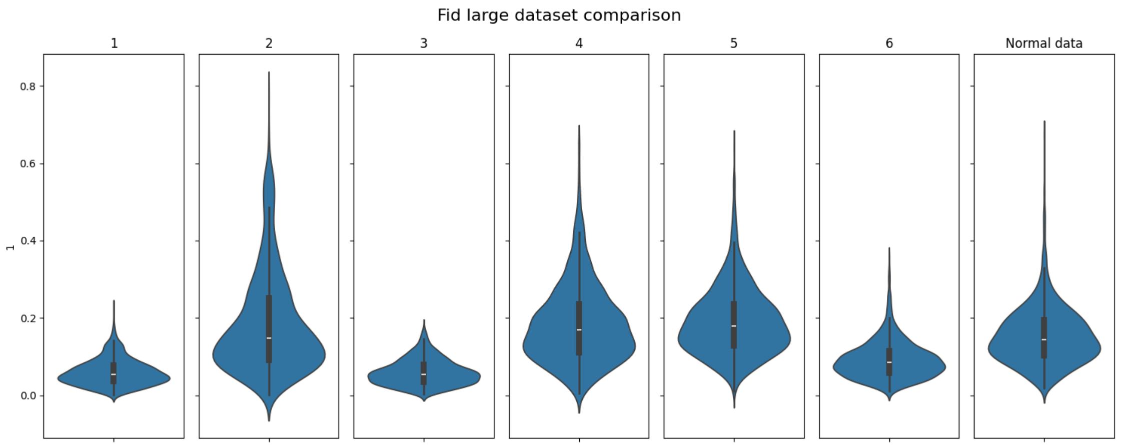

Figure 4.45: Values of Fid across datasets for $n_{SS} = 300$ and $n_{DS} = 900$.

With the large datasets, we find an overlap percentage of 54.8% for C_{llr}^{cal}, 54.0% for devPAV and 60.4% for Fid. The overlap percentages have decreased in comparison with the smaller datasets. Just like before, Fid outperforms the other two metrics, but again not by much.

The findings show that Fid consistently provides the most reliable metric values across different datasets, even though its performance decreases with larger dataset sizes. C_{llr}^{cal} and devPAV show slightly more variability and less reliability across datasets, especially with larger dataset sizes. This reliability is very important in practical application, as it shows that the metric can provide dependable evaluations across different datasets. This observation is corroborated by [18], which collected and compared values of C_{llr}^{cal} across various publications have been collected and compared to each other to see if a 'good' value for C_{llr}^{cal} can be deduced. They also found that the values of C_{llr}^{cal} vary a lot and so they do not give us a lot of information about the consistency of LR-systems.

4.2.3. Reliability of the metrics across dataset sizes

In this section, the results on the reliability of the metrics across dataset sizes will be presented. In other words, we will see how much the metrics' values for consistent LR-systems depend on the size of the datasets. The sizes used here are equal to the sizes we used before: for the small dataset we used $n_{SS} = 50$ and $n_{DS} = 150$, for the medium dataset we used $n_{SS} = 150$ and $n_{DS} = 450$ and for the large dataset we used $n_{SS} = 300$ and $n_{DS} = 900$.

The seventh dataset is excluded in this section due to the computational cost.

First, we look at C_{llr}^{cal}. The value of C_{llr}^{cal} for all consistent LR-datasets of a given size are added together. Then, the distributions of the values per dataset size are compared to each other. This is visualized in Figure 4.46. Note that again, a higher overlap percentage is preferred, indicating reliability of a metric across different dataset sizes.

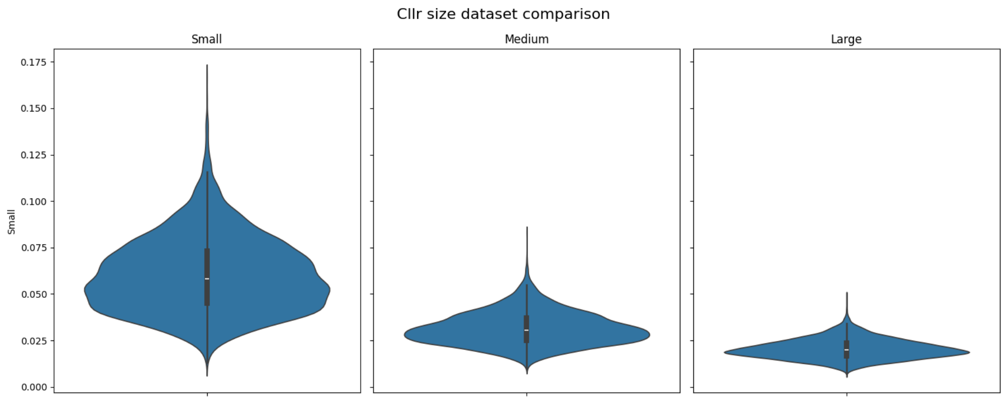

Figure 4.46: Values of C_{llr}^{cal} for consistent datasets of different sizes.

For C_{llr}^{cal}, we find an overlap percentage of 25.0%. This is quite low. It can be seen from Figure 4.46 that this can mostly be attributed to the distribution of the values for the small datasets, as C_{llr}^{cal} performs significantly worse on smaller datasets. For the larger datasets, there seems to be more overlap.

For devPAV, the results across dataset sizes are shown in Figure 4.47.

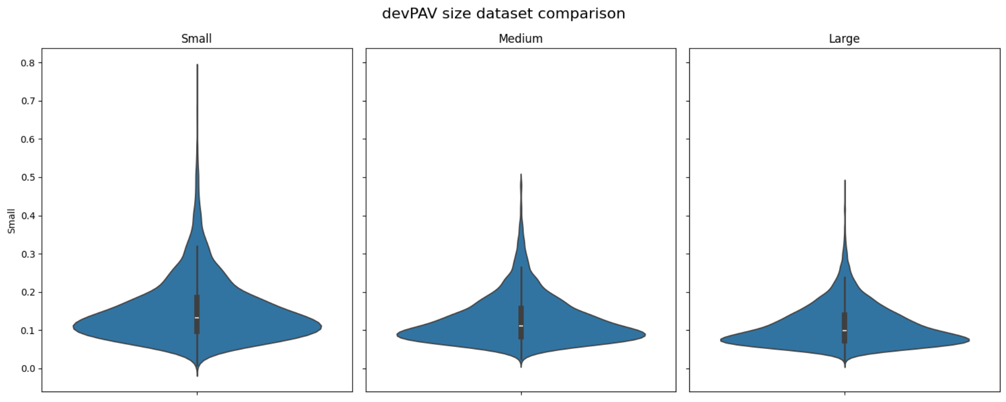

Figure 4.47: Values of devPAV for consistent datasets of different sizes.

As can be seen in Figure 4.47, the overlap percentage is rather high: devPAV gives an overlap percentage of 91.9% across dataset sizes.

Lastly, for Fid the results are shown in Figure 4.48

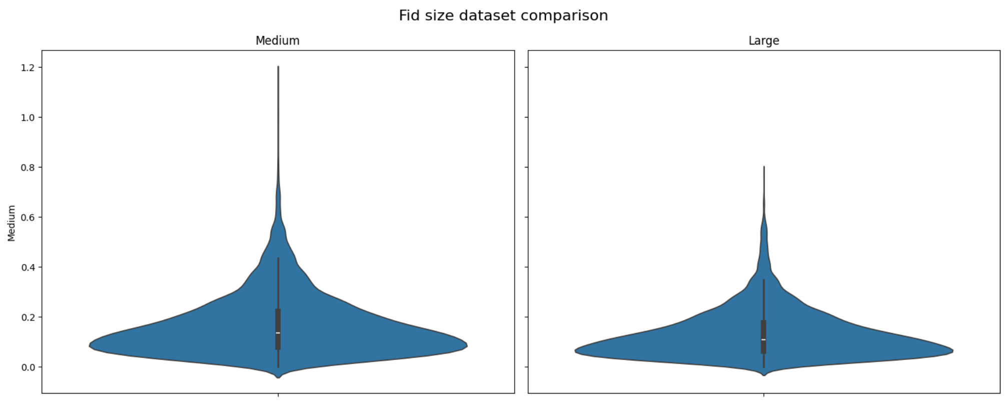

Figure 4.48: Values of Fid for consistent datasets of different sizes.

Fid gives the highest overlap percentage of 93.0% across dataset sizes, outperforming devPAV by just a little bit. Note, however, that for Fid we are only comparing two dataset sizes instead of three, like for the other metrics. This probably results in a higher overlap percentage, as the medium and large dataset values are often closer to each other for each metric.

These results indicate that both devPAV and Fid demonstrate high reliability. The high overlap percentages for devPAV and Fid show that these metrics are less sensitive to the size of the dataset, making them more robust choices for evaluating the consistency of LR-systems across varying dataset sizes. This insight is important for the application of these metrics in practical scenarios, ensuring that the evaluations stay reliable regardless of the dataset size.

5 Conclusions and Discussion

This study aimed to evaluate and compare different consistency metrics for LR-systems on realistic data in order to identify the most effective metric. The evaluation focused on three types of metrics: C_{llr}^{cal}, devPAV, and metrics based on fiducial distributions. Each metric was first optimized individually by performing various adaptations and assessing their effect on the performance of the metric in distinguishing between consistent and inconsistent LR-systems. The dataset sizes used were $n_{SS} = 50$ and $n_{DS} = 150$ for the smaller datasets, $n_{SS} = 150$ and $n_{DS} = 450$ for the medium-sized datasets and $n_{SS} = 300$ and $n_{DS} = 900$ for the larger datasets, where n_{SS} denotes the number of same-source LR-values and n_{DS} denotes the number of different-source LR-values.

The individual metrics were optimized on normally distributed LR-data.

For C_{llr}^{cal}, the optimization was performed by comparing the original that uses the logarithmic score to the same method using other scoring rules. This was tested on the different dataset sizes. The original C_{llr}^{cal} performed the best on all dataset sizes, with overlap percentages of 32.9%, 10.4% and 1.4% for small, medium and large datasets, respectively. The versions of C_{llr}^{cal} using the other scores, although improvements were seen on the large datasets, still had higher overlap percentages on each of the dataset sizes.

For devPAV, several variations of the original metric were considered, namely scaling it by both the x- and y-axis, instead of just the x-axis, and smoothing it by cutting off the corners from the step function. The smoothed version of the devPAV metric was found to be the most reliable across different dataset sizes. On smaller datasets, all versions performed poorly, with high overlap percentages. However, on large datasets, the smoothed devPAV showed an overlap of only 3.3%, making it highly effective in distinguishing between consistent and inconsistent systems. Although the original devPAV showed similar performance, the smoothed devPAV was expected to be more robust to different datasets because it is less sensitive to large steps in the isotonic regression function, and was hence chosen to go forward with.

Among the fiducial metrics, the simple average of the medians performed better than the scaled average and the zero-one metric. The overlap percentage for the average of the medians was 5.9% on large datasets, indicating that it might still be useful despite being less effective compared to C_{llr}^{cal} and devPAV. This metric was named 'Fid'. Unfortunately, for small datasets, the fiducial method does not work.

When tested on real LR-data, C_{llr}^{cal} consistently outperformed the other two metrics in distinguishing between consistent and inconsistent LR-systems. On the medium-sized datasets, the overlap percentage for C_{llr}^{cal} was 3.1%, for devPAV it was 13.0% and for Fid it was 12.8%. On average, C_{llr}^{cal} showed an overlap percentage of 18.2% for the smaller datasets, compared to 29.1% for devPAV. On the large datasets the overlap percentages were 0.3%, 4.4% and 5.8%, respectively. On every dataset size, C_{llr}^{cal} shows the smallest overlap percentage, meaning that it distinguishes most accurately between consistent and inconsistent LR-systems. devPAV and Fid show similar results, but devPAV has the advantage of working on small datasets, which Fid does not.

Testing the metrics' reliability across different consistent LR-systems showed that devPAV and Fid outperform C_{llr}^{cal}. For medium-sized datasets, devPAV and Fid showed overlap percentages of 63.0% and 64.3%, respectively, compared to an overlap percentage of 59.1% for C_{llr}^{cal}. When testing reliability, we want the overlap to be high, showing similar metric values across varying consistent LR-systems. At the small dataset, the overlap percentage for devPAV was 77.8%, again outperforming C_{llr}^{cal} with an overlap percentage of 60.3%. On the larger datasets the difference was smaller: C_{llr}^{cal} had an overlap percentage of 54.8%, devPAV had an overlap percentage of 54.0% and Fid outperformed the other two metrics with an overlap percentage of 60.4%.

Lastly, the reliability of the metrics across varying dataset sizes was tested. C_{llr}^{cal} showed poor results with an overlap percentage of 25.0%, although it should be noted that this can mostly be attributed to the smaller datasets. Both devPAV and Fid performed very well, showing overlap percentages of 91.9% and 93.0%, respectively. However, Fid was only compared on two dataset sizes, resulting in a higher overlap percentage.

In conclusion, the findings of this study provide a thorough comparison of three consistency metrics for LR-systems, highlighting the strengths and limitations of each. C_{llr}^{cal}, using the original logarithmic score, showed the best performance in distinguishing between consistent and inconsistent LR-systems, while the smoothed devPAV emerged as the most reliable metric, assigning similar values to different consistent LR-systems and being most robust to changes in the dataset size. Although Fid did not perform worse than devPAV in most cases, it has the big disadvantage of not working on small datasets. While as a metric it might be the poorer choice for this reason, the method itself still uses interesting ideas and the fiducial plots give a lot of insight into the consistency of LR-systems.

While this study provided us with some valuable insights into the effectiveness of different consistency metrics, there are a few limitations and areas for improvement that future work could explore.

The study was conducted on specific dataset sizes and distributions. For the first part of the optimization, only normally distributed data was used. Although real-life LR-data was used for the second part of the comparison, it was still a small amount of datasets, many of which contained LR-values of the same type of LR-system. Exploring a wider range of dataset characteristics, including different distributions (e.g. skewed distributions, bimodal distributions) and real-world variations, could provide a more comprehensive understanding of metric performance.

Although several adaptations were explored, there may be other creative approaches to modify the metrics that were not considered. Moreover, there are many options for new metrics altogether, for example using scoring rules as metrics. Future research could look into adaptations and combinations of existing metrics to improve the performance further.

The parameter tuning used in this study was quite basic. For example, the bandwidth for the KDE was mostly chosen based on trying a few different methods and seeing which one seemed to work the best, without a deeper analysis of why certain bandwidths might be better than others. More parameter tuning could have optimized performance further.

Lastly, the implementation of fiducial metrics was slow and computationally heavy. Optimizing the code for better efficiency, maybe through algorithmic improvements or faster computational techniques, could make these metrics more practical for larger datasets.

In summary, this study provided a detailed evaluation of three consistency metrics for LR-systems, with C_{llr}^{cal} emerging as the most effective metric for distinguishing between consistent and inconsistent systems. The smoothed devPAV demonstrated the highest reliability across datasets of varying sizes, making it a strong alternative. While the Fid metric performed similarly to devPAV in most cases, its inability to handle small datasets limits its practical application. However, there is still room for refinement and improvement. By addressing the limitations identified and exploring new approaches, this study lays a foundation for future research to make LR-system evaluations even more accurate and reliable. In this way it contributes to the development of better forensic analysis techniques.

References

[1] R. Ahuja and J. Orlin. “A Fast Scaling Algorithm for Minimizing Separable Convex Functions Subject to Chain Constraints”. In: *Operations Research* 29.5 (2001), pp. 629–805. DOI: `10.1287/opre.49.5.784.10601`.

[2] M. Baiker-Sørensen et al. “Automated interpretation of comparison scores for firearm toolmarks on cartridge case primers”. In: *Forensic Science International* 353.Article 111858 (2023). DOI: `10.1016/j.forsciint.2023.111858`.

[3] A. Bolck, H. Ni, and M. Lopatka. “Evaluating score- and feature-based likelihood ratio models for multivariate continuous data: applied to forensic MDMA comparison”. In: *Law, Probability and Risk* 14.3 (2015), pp. 243–266. DOI: `10.1093/lpr/mgv009`.

[4] W.M. Bolstad. *Introduction to Bayesian Statistics*. 1st ed. New Jersey: John Wiley & Sons, 2004.

[5] N. Brümmer and D.A. van Leeuwen. “The distribution of calibrated likelihood-ratios in speaker recognition”. In: *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France* (2013), pp. 1619–1623.

[6] N. Brümmer and J. Du Preez. “Application-Independent Evaluation of Speaker Detection”. In: *Comput. Speech Lang* 20.2-3 (2006), pp. 230–275.

[7] N. Brümmer and J. Du Preez. “The PAV algorithm optimizes binary proper scoring rules”. PhD thesis. University of Stellenbosch, 2013.

[8] A. Collins and N.E. Morton. “Likelihood ratios for DNA identification”. In: *Proceedings of the National Academy of Sciences* 91.13 (1994), pp. 6007–6011.

[9] M. DeGroot and S. Fienberg. “The Comparison and Evaluation of Forecasters”. In: *Journal of the Royal Statistical Society. Series D (The Statistician)* 12 (1983), pp. 12–22.

[10] N. Egli, C. Champod, and P. Margot. “Evidence evaluation in fingerprint comparison and automated fingerprint identification systems—Modelling within finger variability”. In: *Forensic Science International* 167.2-3 (2007), pp. 189–195. DOI: `10.1016/j.forsciint.2006.06.054`.

[11] A. van Es et al. “Implementation and assessment of a likelihood ratio approach for the evaluation of LA-ICPMS evidence in forensic glass analysis”. In: *Science & Justice* 57.3 (2017), pp. 183–194.

[12] I. Evett. “Avoiding the Transposed Conditional”. In: *Science & Justice* 35.2 (1995), pp. 127–131.

[13] M. O. Finkelstein and W. B. Fairley. “A Bayesian Approach to Identification Evidence”. In: *Harvard Law Review* 83.3 (1970), pp. 267–301. DOI: `10.2307/1339656`.

[14] I. Good. “Weight of Evidence: A Brief Survey”. In: *Bayesian Statistics* 2 (1985), pp. 249–270.

[15] J. Hannig and H. Iyer. “Testing for calibration discrepancy of reported likelihood ratios in forensic science”. In: *Journal of the Royal Statistical Society Series A: Statistics in Society* 185.1 (2021), pp. 267–301.

[16] J. Hannig et al. “Are reported likelihood ratios well calibrated?” In: *Forensic Science International: Genetics Supplement Series* 7.1 (2019), pp. 572–574.

[17] J. Leegwater et al. “From data to a validated score-based LR system: a practicioner’s guide”. In: *Forensic Science International* 357 (2024). DOI: `10.1016/j.forsciint.2024.111994`.

[18] S. van Lierop et al. “An overview of log likelihood ratio cost in forensic science – Where is it used and what values can we expect?” In: *Forensic Science International* 8 (2024). DOI: `10.1016/j.fsisyn.2024.100466`.

[19] D. Ommen. “Approximate Statistical Solutions to the Forensic Identification of Source Problem”. PhD thesis. South Dakota State University, 2017.

[20] D. Ommen and C. Saunders. “Building a unified statistical framework for the forensic identification of source problems”. In: *Law, Probability and Risk* 17 (2018), pp. 179–197. DOI: `10.1093/lpr/mgy008`.

[21] D. Ramos and J. Gonzalez-Rodriguez. “Reliable support: Measuring calibration of likelihood ratios”. In: *Forensic Science International* 230 (2013), pp. 156–169.

[22] R. Royall. “On the Probability of Observing Misleading Statistical Evidence”. In: *Journal of the American Statistical Association* 95.451 (2000), pp. 760–768. DOI: `10.1080/01621459.2000.10474264`.

[23] N. Scheijen. “Forensic speaker recognition. Based on text analysis of transcribed speech fragments”. MA thesis. TU Delft, 2020.

[24] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. 1st ed. London: Chapman & Hall, 1986.

[25] D. Taylor, J. Buckleton, and I. Evett. “Testing likelihood ratios produced from complex DNA profiles”. In: *Forensic Science International: Genetics* 16 (2015), pp. 165–171.

[26] P. Vergeer. “From specific-source feature-based to common-source score-based likelihood-ratio systems: ranking the stars”. In: *Law, Probability and Risk* 22 (2023).

[27] P. Vergeer, Y. van Schaik, and M. Sjerps. “Measuring calibration of likelihood-ratio systems: A comparison of four metrics, including a new metric devPAV”. In: *Forensic Science International* 321.Article 110722 (2021).

[28] P. Vergeer et al. “Numerical Likelihood Ratios Outputted by LR Systems Are Often Based on Extrapolation: When to Stop Extrapolating?” In: *Science & Justice* 56.6 (2016), pp. 482–491. DOI: `10.1016/j.scijus.2016.06.003`.

A
Source Code

Listing A.1: functions thesis.py

```python
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt, comb
from scipy.stats import gaussian_kde, norm
import lir
from scipy.interpolate import interp1d
import pandas as pd
from scipy.integrate import simps

# FUNCTIONS FOR DEVPAV

# this function is copied from the lir library
def _calcsurface(c1: (float, float), c2: (float, float)) -> float:
    """
    Helper function that calculates the desired surface for two xy-coordinates
    """
    # step 1: calculate intersection (xs, ys) of straight line through coordinates with
        identity line (if slope (a) = 1, there is no intersection and surface of this
        parrellogram is equal to deltaY * deltaX)
    x1, y1 = c1
    x2, y2 = c2
    a = (y2 - y1) / (x2 - x1)

    if a == 1:
        # dan xs equals +/- Infinite en is er there is no intersection with the identity line

        # the surface of the parallellogram is:
        surface = (x2 - x1) * np.abs(y1 - x1)

    elif (a < 0):
        raise ValueError(f"slope␣is␣negative;␣impossible␣for␣PAV-transform.␣Coordinates␣are␣{
            c1}␣and␣{c2}.␣Calculated␣slope␣is␣{a}")
    else:
        # than xs is finite:
        b = y1 - a * x1
        xs = b / (1 - a)
        # xs

        # step 2: check if intersection is located within line segment c1 and c2.
        if x1 < xs and x2 >= xs:
            # then intersection is within
            # (situation 1 of 2) if y1 <= x1 than surface is:
            if y1 <= x1:
                surface = 0.5 * (xs - y1) * (xs - x1) - 0.5 * (xs - x1) * (xs - x1) + 0.5 * (
                    y2 - xs) * (x2 - xs) - 0.5 * (
                            x2 - xs) * (x2 - xs)
            else:
                # (situation 2 of 2) than y1 > x1, and surface is:
```

```
                surface = 0.5 * (xs - x1) ** 2 - 0.5 * (xs - y1) * (xs - x1) + 0.5 * (x2 - xs) ** 2 - 0.5 * (x2 - xs) * (
                            y2 - xs)
                # dit is the same as 0.5 * (xs - x1) * (xs - y1) - 0.5 * (xs - y1) * (xs - y1) + 0.5 * (y2 - xs) * (x2 - xs) - 0.5 * (y2 - xs) * (y2 - xs) + 0.5 * (y1 - x1) * (y1 - x1) + 0.5 * (x2 - y2) * (x2 -y2)
        else:  # then intersection is not within line segment
            # if (situation 1 of 4) y1 <= x1 AND y2 <= x1, and surface is
            if y1 <= x1 and y2 <= x1:
                surface = 0.5 * (y2 - y1) * (x2 - x1) + (x1 - y2) * (x2 - x1) + 0.5 * (x2 - x1) * (x2 - x1)
            elif y1 > x1:  # (situation 2 of 4) then y1 > x1, and surface is
                surface = 0.5 * (x2 - x1) * (x2 - x1) + (y1 - x2) * (x2 - x1) + 0.5 * (y2 - y1) * (x2 - x1)
            elif y1 <= x1 and y2 > x1:  # (situation 3 of 4). This should be the last possibility.
                surface = 0.5 * (y2 - y1) * (x2 - x1) - 0.5 * (y2 - x1) * (y2 - x1) + 0.5 * (x2 - y2) * (x2 - y2)
            else:
                # situation 4 of 4 : this situation should never appear. There is a fourth sibution as situation 3, but than above the identity line. However, this is impossible by definition of a PAV-transform (y2 > x1).
                raise ValueError(f"unexpected coordinate combination: ({x1}, {y1}) and ({x2}, {y2})")
    return surface

# this function is taken from the lir library
def _devpavcalculator(lrs, pav_lrs, y):
    """
    Function that calculates davPAV for a PAVresult for SSLRs and DSLRs.

    Input:
    - Lrs: np.array with LR-values.
    - pav_lrs: np.array with LRs after PAV-transform. y = np.array with labels (1 for H1 and 0 for H2).

    Output:
    - devPAV value.

    """
    DSLRs, SSLRs = lir.Xy_to_Xn(lrs, y)
    DSPAVLRs, SSPAVLRs = lir.Xy_to_Xn(pav_lrs, y)
    PAVresult = np.concatenate([SSPAVLRs, DSPAVLRs])
    Xen = np.concatenate([SSLRs, DSLRs])

    # order coordinates based on x's then y's and filtering out identical datapoints
    data = np.unique(np.array([Xen, PAVresult]), axis=1)
    Xen = data[0, :]
    Yen = data[1, :]

    # pathological cases
    # first one of four: PAV-transform has a horizonal line to log(X) = -Inf as to log(X) = Inf
    if Yen[0] != 0 and Yen[-1] != np.inf and Xen[-1] == np.inf and Xen[-1] == np.inf:
        return np.Inf

    # second of four: PAV-transform has a horizontal line to log(X) = -Inf
    if Yen[0] != 0 and Xen[0] == 0 and Yen[-1] == np.inf:
        return np.Inf

    # third of four: PAV-transform has a horizontal line to log(X) = Inf
    if Yen[0] == 0 and Yen[-1] != np.inf and Xen[-1] == np.inf:
        return np.Inf

    # fourth of four: PAV-transform has one vertical line from log(Y) = -Inf to log(Y) = Inf
    wh = (Yen == 0) | (Yen == np.inf)
    if np.sum(wh) == len(Yen):
        return np.nan

    else:
        # then it is not a  pathological case with weird X-values and devPAV can be
```

```
            calculated

        # filtering out -Inf or 0 Y's
        wh = (Yen > 0) & (Yen < np.inf)
        Xen = np.log10(Xen[wh])
        Yen = np.log10(Yen[wh])
        # create an empty list with size (len(Xen))
        devPAVs = [None] * len(Xen)
        # sanity check
        if len(Xen) == 0:
            return np.nan
        elif len(Xen) == 1:
            return abs(Xen - Yen)
        # than calculate devPAV
        else:
            deltaX = Xen[-1] - Xen[0]
            surface = 0
            for i in range(1, (len(Xen))):
                surface = surface + _calcsurface((Xen[i - 1], Yen[i - 1]), (Xen[i], Yen[i]))
                devPAVs[i - 1] = _calcsurface((Xen[i - 1], Yen[i - 1]), (Xen[i], Yen[i]))
            # return(list(surface/a, PAVresult, Xen, Yen, devPAVs))
            return surface / deltaX


def scaled_devpavcalc(lrs, pav_lrs, y):
    """
    Function that calculates the scaled davPAV for a PAVresult for SSLRs and DSLRs.

    Input:
    - lrs: np.array with LR-values.
    - pav_lrs: np.array with LRs after PAV-transform.
    - y: np.array with labels (1 for H1 and 0 for H2).

    Output:
    - scaled devPAV value.

    """
    DSLRs, SSLRs = lir.Xy_to_Xn(lrs, y)
    DSPAVLRs, SSPAVLRs = lir.Xy_to_Xn(pav_lrs, y)
    PAVresult = np.concatenate([SSPAVLRs, DSPAVLRs])
    Xen = np.concatenate([SSLRs, DSLRs])

    # Order coordinates based on x's then y's and filtering out identical datapoints
    data = np.unique(np.array([Xen, PAVresult]), axis=1)
    Xen = data[0, :]
    Yen = data[1, :]

    # pathological cases
    # first one of four: PAV-transform has a horizonal line to log(X) = -Inf as to log(X) =
        Inf
    if Yen[0] != 0 and Yen[-1] != np.inf and Xen[-1] == np.inf and Xen[-1] == np.inf:
        return np.Inf

    # second of four: PAV-transform has a horizontal line to log(X) = -Inf
    if Yen[0] != 0 and Xen[0] == 0 and Yen[-1] == np.inf:
        return np.Inf

    # third of four: PAV-transform has a horizontal line to log(X) = Inf
    if Yen[0] == 0 and Yen[-1] != np.inf and Xen[-1] == np.inf:
        return np.Inf

    # forth of four: PAV-transform has one vertical line from log(Y) = -Inf to log(Y) = Inf
    wh = (Yen == 0) | (Yen == np.inf)
    if np.sum(wh) == len(Yen):
        return np.nan

    else:
        # then it is not a  pathological case with weird X-values and devPAV can be
            calculated

        # filtering out -Inf or 0 Y's
        wh = (Yen > 0) & (Yen < np.inf)
        Xen = np.log10(Xen[wh])
        Yen = np.log10(Yen[wh])
        # create an empty list with size (len(Xen))
        devPAVs = [None] * len(Xen)
        # sanity check
        if len(Xen) == 0:
            return np.nan
        elif len(Xen) == 1:
            return abs(Xen - Yen)
        # then calculate devPAV
        else:
            # determine the difference in x-values and y-values
            deltaX = Xen[-1] - Xen[0]
            deltaY = Yen[-1] - Yen[0]
            surface = 0
            for i in range(1, (len(Xen))):
                surface = surface + _calcsurface((Xen[i - 1], Yen[i - 1]), (Xen[i], Yen[i]))
                devPAVs[i - 1] = _calcsurface((Xen[i - 1], Yen[i - 1]), (Xen[i], Yen[i]))
            # scale by surface
            return surface / (deltaX*deltaY)

def devpav(lrs: np.ndarray, y: np.ndarray) -> float:
    """
    Function that calculates normal devPAV for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - devPAV value.
    """
    # Check if input is valid
    if sum(y) == len(y) or sum(y) == 0:
        raise ValueError('devpav: illegal input: at least one value is required for each 
            class')

    # Determine pav lrs
    cal = lir.IsotonicCalibrator()
    pavlrs = cal.fit_transform(lrs, y)

    # Return devpav
    return _devpavcalculator(lrs, pavlrs, y)

def scaled_devpav(lrs: np.ndarray, y: np.ndarray) -> float:
    """
    Function that calculates scaled devPAV for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - scaled devPAV value.
    """

    # Check if input is valid
    if sum(y) == len(y) or sum(y) == 0:
        raise ValueError('devpav: illegal input: at least one value is required for each 
            class')

    # Determine pav lrs
    cal = lir.IsotonicCalibrator()
    pavlrs = cal.fit_transform(lrs, y)

    # Return scaled devpav
    return scaled_devpavcalc(lrs, pavlrs, y)

def devpav_new(lrs: np.ndarray, y:np.ndarray) -> float:
    """
    Function that calculates smoothed devPAV for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - smoothed devPAV value.
    """
    # determine pav lrs
    cal = lir.IsotonicCalibrator()
    pavlrs = cal.fit_transform(lrs, y)

    # get original and pav lrs and ensure row vector
    x = np.ravel(lrs)
    y = np.ravel(pavlrs)

    # calculating devPAV only makes sense if the original and transformed
    # variables have the same domain; in this case they are both LRs with a
    # domain between 0 and +inf.
    if any(x < 0) or any(y < 0):
        raise ValueError('Both␣variables␣should␣be␣non-negative.')

    # Convert both coordinates to log10
    x = np.log10(x)
    y = np.log10(y)

    # Sort values
    x = np.sort(x)
    y = np.sort(y)

    # Exclude datapoints with one or two non-finite coordinates
    finite = np.isfinite(x) & np.isfinite(y)
    x = x[finite]
    y = y[finite]

    # Add initial and final points at the identity line
    x = np.concatenate(([x[0]], x, [x[-1]]))
    y = np.concatenate(([x[0]], y, [x[-1]]))

    # Rotate the transformation line clockwise by 45 degrees
    x_rot = (x + y) / np.sqrt(2)
    y_rot = (y - x) / np.sqrt(2)

    # Add new points to the line, where it crosses the (new rotated) X-axis.
    # This is when the Y-values of two adjacent points have opposite signs.
    i_cross = np.where(np.abs(np.diff(np.sign(y_rot))) == 2)[0]
    # Add new points in backwards order, so the cross indices are unchanged
    for i_p in range(len(i_cross) - 1, -1, -1):
        i_c = i_cross[i_p]
        x_dif = np.diff(x_rot[i_c:i_c + 2])
        y_dif = np.diff(y_rot[i_c:i_c + 2])

        # The added x-coordinate is shifted proportional to the y-values
        x_add = x_rot[i_c] + x_dif * np.abs(y_rot[i_c] / y_dif)
        x_add = np.array([x_add]).reshape(1, )
        x_rot = np.concatenate((x_rot[:i_c + 1], x_add, x_rot[i_c + 1:]))
        y_add = 0
        y_rot = np.concatenate((y_rot[:i_c + 1], [y_add], y_rot[i_c + 1:]))

    # Determine corners of step function
    critical_points = []
    critical_points.append((x_rot[0], y_rot[0]))
    increasing = True
    for i in range(1, len(y_rot)):
        if increasing and y_rot[i] < y_rot[i - 1]:
            # Transition from increasing to decreasing
            critical_points.append((x_rot[i-1], y_rot[i-1]))
            increasing = False
        elif not increasing and y_rot[i] > y_rot[i - 1]:
            # Transition from decreasing to increasing
```

```python
            critical_points.append((x_rot[i - 1], y_rot[i - 1]))
            increasing = True
    critical_points.append((x_rot[-1], y_rot[-1]))

    # Determine the triangles that form the steps (corners and middle point)
    tuples = []
    for i in range(0, len(critical_points) - 2, 2):
        tuples.append([critical_points[i], critical_points[i+1], critical_points[i+2]])

    # Determine all the lines of the step function
    lines = []
    for i in range(0, len(critical_points)-1, 1):
        line = [(critical_points[i][0],critical_points[i][1]), (critical_points[i+1][0],
            critical_points[i+1][1])]
        lines.append(line)

    # Determine the new points we want to interpolate linearly: corners stay, middle points
        are cut off by
    # determining middle points of lines from corner to middle point of triangle and drawing
        line between
    # middle points
    points = []
    points.append(lines[0][0])
    for line in lines:
        mid_x = (line[0][0] + line[1][0])/2
        mid_y = (line[0][1] + line[1][1])/2
        points.append((mid_x, mid_y))
    points.append((lines[-1][1]))

    # Determine x-values and y-values of points
    xvals = np.array([point[0] for point in points])
    yvals = np.array([point[1] for point in points])

    # Interpolate linearly between the points, giving a piecewise linear function
    interp_func = interp1d(xvals, yvals, kind='linear')

    # Obtain corresponding y-values from the interpolation function
    new_x = np.linspace(min(xvals), max(xvals), 100)
    new_y = interp_func(new_x)

    # Determine area
    area = np.diff(new_x) * np.abs(new_y[:-1] + new_y[1:]) / 2
    total_area = np.sum(area)

    # Determine smoothed devPAV
    x_range = np.max(new_x) - np.min(new_x)
    smoothed_devpav = total_area / x_range

    return smoothed_devpav

# FUNCTIONS FOR CLLR

def cllr(lrs, y):
    """
    Function that calculates cllr cal using logarithmic scoring rule for LR data under H1 and
         H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - cllr cal value.
    """

    # Determine total cllr and discrimination power and subtract to find cllr cal
    cllrmax = lir.metrics.cllr(lrs, y)
    cllrmin = lir.metrics.cllr_min(lrs, y)

    return cllrmax - cllrmin

```

```

def brier(lrs, y):
    """
    Function that calculates cllr cal using brier score for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - cllr cal value with brier score.
    """

    # Make dictionary of labels and corresponding LR-values
    grouped_LR = {}
    for label, LR in zip(y, lrs):
        if label not in grouped_LR:
            grouped_LR[label] = []
        grouped_LR[label].append(LR)

    l1 = len(grouped_LR.get(1, []))
    l2 = len(grouped_LR.get(0, []))
    sum_1 = 0
    sum_2 = 0
    for label, LR_list in grouped_LR.items():
        # Determine Brier scores for labels using posterior
        # H_p true
        if label == 1:
            for LR in LR_list:
                if LR != 0 and LR != np.inf:
                    posterior = LR / (1 + LR)
                    sum_1 += ((posterior - 1) ** 2)
        # H_d true
        else:
            for LR in LR_list:
                if LR != 0 and LR != np.inf:
                    posterior = LR / (1 + LR)
                    sum_2 += (posterior ** 2)
    # Determine ECE using Brier score
    brier = 0.5 / l1 * sum_1 + 0.5 / l2 * sum_2

    return brier

def zero_one(LRs, labels):
    """
    Function that calculates cllr using zero-one score for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - cllr cal value with zero-one score.
    """

    n = len(LRs)

    # Count misclassifications by determining posteriors
    misclas = 0
    misclas2 = 0

    for i in range(n):
        if LRs[i] != 0 and LRs[i] != np.inf:
            posterior = LRs[i] / (1 + LRs[i])
            if posterior > 0.5 and labels[i] == 0:
                misclas += 1
            elif posterior < 0.5 and labels[i] == 1:
                misclas += 1

    # Determine frequency of misclassifications
    misclas = misclas / n

    return misclas

def spherical(LRs, labels):
    """
    Function that calculates cllr using spherical scoring rule for LR data under H1 and H2.

    Input:
    - lrs: np.array with LR-values.
    - y: np.array with labels, 1 for H1 and 0 for H2.

    Output:
    - cllr cal value with spherical scoring rule.
    """
    n = len(LRs)
    spherical = 0
    m = 0

    # Determine ECE using spherical scoring rule
    for i in range(n):
        if LRs[i] != 0 and LRs[i] != np.inf:
            posterior = LRs[i] / (1 + LRs[i])
            spherical += (labels[i] * posterior + (1 - labels[i]) * (1 - posterior)) / sqrt(
                posterior ** 2 + (1 - posterior) ** 2)
            m += 1

    # To avoid error
    if m == 0:
        m = 1

    return spherical / m

# FUNCTIONS FOR FIDUCIAL METRICS

# This function is based on Jan Hannig's R-code
def fiducial_sample(data,nfid):
    """
    Function that makes fiducial samples of data.

    Input:
    - data: np.array of LR-values.
    - nfid = amount of fiducial samples.

    Output:
    - Dictionary that contains the following keys:
        - 'data': sorted data,
        - 'u': fiducial samples,
        - 'n': number of data points,
        - 'nfid': number of fiducial samples.
    """
    n = len(data)

    # Sort data
    sorted_data = np.sort(data)
    sorted_data = np.transpose(sorted_data)

    # Make nfid fiducial samples  of length ndata
    u = np.sort([np.random.uniform(size=n) for _ in range(nfid)])
    u = np.transpose(u)
    u = u[::-1]

    return {
        'data': sorted_data,  # Sorted data
        'u': u,  # Fiducial samples
        'n': n,  # Number of data points
        'nfid': nfid  # Number of fiducial samples
    }

# This function is based on Jan Hannigs R-code
def particle_grid(xgrid, lrt_fsample):
    """
    Function that defines grid for fiducial inference and calculates survival functions and
        integral on grid.

    Input:
    - xgrid = grid for x-values,
    - lrt_fsample = dictionary with following keys:
        - 'data': sorted data,
        - 'u': fiducial samples,
        - 'n': number of data points,
        - 'nfid': number of fiducial samples.
    Output:
    - Dictionary with following keys:
        - 'grid': grid,
        - 'survival': array of survival function values at each grid point for each fiducial
            sample,
        - 'bottom': array representing the integral of the survival function up to each grid
            point for each fiducial sample
        - 'nfid': number of fiducial samples,
        - 'n': number of data points.
    """

    # Sort grid points
    ngrid = len(xgrid)
    grid = np.sort(xgrid)

    # Extract and prepare the data and fiducial sample information
    data = np.concatenate(([0],lrt_fsample['data'], [np.inf]))
    n = lrt_fsample['n']
    nfid = lrt_fsample['nfid']

    # Initialize arrays for survival functions and integrals
    both_integrals_survival = []
    both_integrals_bottom = []

    for i in range(nfid):
        # Each fiducial sample processed separately
        u = np.concatenate(([1], lrt_fsample['u'][:, i], [0]))
        u = u.reshape(-1,1)
        data = data.reshape(-1,1)

        # Calculate the expression results based on fiducial sample and data
        expression_result = u[n] * (data[n] - data[n-1] +(1 / n))
        expression_result = expression_result.reshape(-1,1)

        # Concatenate and compute the integral of survival function
        concatenated = np.concatenate((expression_result, (data[n:0:-1]*u[n-1::-1] - data[n
            -1::-1]*u[n:0:-1])/2))
        # Cumulative sum to get the integral
        dataint = np.cumsum(concatenated)
        # Flip integral values for correct alignment
        flipped_int = np.flip(dataint)
        flipped_int = flipped_int.reshape(-1,1)
        dataintegral = flipped_int + (data[n]*u[n]-data[0:(n+1)]*u[0:(n+1)])/2
        dataintegral_f = dataintegral.flatten().tolist()

        # Initialize arrays for survival function and integral results
        survival_array = np.empty(ngrid)
        integral_array = np.empty(ngrid)

        # Evaluate survival and integral values at each grid point
        for j in range(ngrid):
        # Find the indices of data points that are nearest to the grid point
            indeces_ub = np.where(data>=grid[j])
            index_ub = indeces_ub[0][0]
            indeces_lb = np.where(data <= grid[j])
            index_lb = indeces_lb[0][-1]
            # If exactly hitting a grid point
            if index_ub <= index_lb:
                survival_array[j] = u[index_lb]
                integral_array[j] = dataintegral_f[index_lb]
            # If grid point is beyond the range of the data
            elif index_ub == n+1:
                gridoff = np.exp(-(grid[j] - data[n]) / (data[n] - data[n-1] + (1 / n)))
                survival_array[j] = u[n] * gridoff
                integral_array[j] = np.nan
            # If grid point is between data points
            else:
                survival_array[j] = (u[index_ub] * (grid[j] - data[index_lb]) + u[index_lb] *
                                        (data[index_ub] - grid[j]))/(data[index_ub] - data[index_lb])
                integral_array[j] = dataintegral_f[index_ub] + \
                                       (data[index_ub] - grid[j]) * (survival_array[j] + u[index_ub]) / 2

        # Append the results for the current fiducial sample
        both_integrals_survival.append(survival_array)
        both_integrals_bottom.append(grid*survival_array + integral_array)

    return {
        'grid': grid,
        'survival': np.transpose(np.array(both_integrals_survival)),
        'bottom': np.transpose(np.array(both_integrals_bottom)),
        'nfid': lrt_fsample['nfid'],
        'n': lrt_fsample['n']
    }

# This function is based on Jan Hannig's R-code
def fid_diff_log(particle_top, particle_bottom, coarse_index=None):
    """
    Function to calculate the log difference of fiducial sample values between two particles.

    Input:
    - particle_top: dictionary containing fiducial sample data for the top particle with keys:
       - 'grid': grid of x-values,
       - 'survival': survival function values for each fiducial sample,
       - 'nfid': number of fiducial samples,
       - 'n': number of data points.
    - particle_bottom: dictionary containing fiducial sample data for the bottom particle with the same keys as particle_top.
    - coarse_index: optional array of indices to coarsely sample the grid. If None, uses all indices.

    Output:
    - Dictionary with the following keys:
       - 'fsample_top': survival function values for the top particle,
       - 'fsample_bottom': bottom function values for the bottom particle,
       - 'n_top': number of data points for the top particle,
       - 'n_bottom': number of data points for the bottom particle,
       - 'nfid': number of fiducial samples,
       - 'fdiff_logratio': logarithm of the ratio of differences between the top and bottom fiducial samples,
       - 'grid': grid of x-values,
       - 'dgrid': coarse grid used for sampling.
       """

    grid = particle_top['grid']

    # Check if grids and number of fiducial samples match
    if np.sum(grid != particle_bottom['grid']) > 0 or particle_bottom['nfid'] != particle_top['nfid']:
        print('Mismatch␣of␣inputs')
        return None

    # Use all indices if coarse_index is not provided or is invalid
    if coarse_index is None or len(coarse_index) <= 1:
        coarse_index = np.arange(0, len(grid) + 1)

    # Find intersection of coarse_index with valid grid indices
    cindex = np.intersect1d(np.arange(0, len(grid) + 1), coarse_index)

    if len(cindex) < 2:
        print('Incompatible␣coarse_index')
        return None

    # Extract fiducial samples
    fidTop = particle_top['survival']
    fidBottom = particle_bottom['bottom']

    # Calculate the difference in fiducial samples
    d_fid_Top = -np.diff(fidTop[cindex], axis=0)
    d_fid_Bottom = -np.diff(fidBottom[cindex], axis=0)

    # Compute the log difference between the top and bottom fiducial samples
    fid_sample_slope_ratio = (np.log10(d_fid_Top) - np.log10(d_fid_Bottom))
    coarsegrid = grid[cindex]

    return {
        'fsample_top': fidTop,
        'fsample_bottom': fidBottom,
        'n_top': particle_top['n'],
        'n_bottom': particle_bottom['n'],
        'nfid': particle_top['nfid'],
        'fdiff_logratio': fid_sample_slope_ratio,
        'grid': grid,
        'dgrid': coarsegrid
    }

# This function is based on Jan Hannig's code
def fid_diff_CI(fid_dif_sample, alpha=0.05):
    """
    Function to calculate confidence intervals for the fiducial differences.

    Input:
    - fid_dif_sample: dictionary containing fiducial differences with keys:
        - 'fdiff_logratio': logarithmic differences of fiducial sample ratios,
        - 'dgrid': coarse grid used for sampling.
    - alpha: significance level for confidence interval (default is 0.05 for 95% CI).

    Output:
    - Dictionary with the following keys:
        - 'mean': central value of the fiducial slope,
        - 'uniform_lower': lower bound of the uniform confidence interval,
        - 'uniform_upper': upper bound of the uniform confidence interval,
        - 'median': median of the fiducial slope,
        - 'point_lower': lower bound of the pointwise confidence interval,
        - 'point_upper': upper bound of the pointwise confidence interval,
        - 'dgrid': coarse grid used for sampling.
        """

    fiducial_slope = fid_dif_sample['fdiff_logratio']

    # Calculate the central quantile of the fiducial slope
    CI_center = np.apply_along_axis(lambda x: np.quantile(x, 0.5, axis=0, keepdims=True), 1,
        fiducial_slope)

    # Calculate the scale of the confidence interval
    CI_scale = np.mean(np.abs(fiducial_slope - CI_center), axis=1, keepdims=True)

    # Calculate the scaled fiducial differences
    fid_diff = fiducial_slope - CI_center
    fid_scaled_diff = fid_diff / CI_scale
    fid_abs_scaled_diff = np.abs(fid_scaled_diff)

    # Compute the maximum of the scaled fiducial differences
    fid_max = np.nanmax(fid_abs_scaled_diff, axis=0)

    # Calculate the cutoff value for the confidence intervals
    cut_off = np.quantile(fid_max, 1 - alpha, axis=0)

    # # Calculate the confidence intervals
    mean = CI_center
    uniform_lower = CI_center - cut_off * CI_scale
    uniform_upper = CI_center + cut_off * CI_scale
    median = CI_center
    point_lower = np.apply_along_axis(lambda x: np.quantile(x, alpha / 2, axis=0, keepdims=
        True), 1, fiducial_slope)
    point_upper = np.apply_along_axis(lambda x: np.quantile(x, 1 - alpha / 2, axis=0,
        keepdims=True), 1, fiducial_slope)

    return {
        'mean': mean,
        'uniform_lower': uniform_lower,
        'uniform_upper': uniform_upper,
        'median': median,
        'point_lower': point_lower,
        'point_upper': point_upper,
        'dgrid': fid_dif_sample['dgrid'],
    }

def compare_CI(compare_sample, alpha=0.05):

    """
    Function to calculate confidence intervals for comparison samples.

    Input:
    - compare_sample: a 2D numpy array where each row represents a sample and each column
        represents a different comparison.
    - alpha: significance level for the confidence interval (default is 0.05 for 95% CI).

    Output:
    - Dictionary with the following keys:
        - 'mean': central value of the confidence interval (median of the samples),
        - 'uniform_lower': lower bound of the uniform confidence interval,
        - 'uniform_upper': upper bound of the uniform confidence interval,
        - 'median': median of the comparison samples,
        - 'point_lower': lower bound of the pointwise confidence interval,
        - 'point_upper': upper bound of the pointwise confidence interval.
    """

    # Calculate the center of the confidence interval
    CI_center = np.apply_along_axis(np.quantile, 1, compare_sample, 0.5, na_rm=True)

    # Calculate the scale of the confidence interval
    CI_scale = np.mean(np.abs(compare_sample - CI_center), axis=1)

    # Calculate fid_max (maximum of the scaled differences) and cutoff value for uniform
        interval
    fid_max = np.max(np.abs((compare_sample - CI_center) / CI_scale), axis=1, na_rm=True)
    cut_off = np.quantile(fid_max, 1 - alpha, na_rm=True)

    return {
        'mean': CI_center,
        'uniform_lower': CI_center - cut_off * CI_scale,
        'uniform_upper': CI_center + cut_off * CI_scale,
        'median': CI_center,
        'point_lower': np.apply_along_axis(np.quantile, 1, compare_sample, alpha / 2, na_rm=
            True),
        'point_upper': np.apply_along_axis(np.quantile, 1, compare_sample, 1 - alpha / 2,
            na_rm=True)
    }

def fid_AUC(fid_sample_top, fid_sample_bottom):
    """
    Function to calculate the Area Under the Curve (AUC) for fiducial samples.

    Input:
    - fid_sample_top: dictionary containing fiducial sample data for the top particle with
        keys:
        - 'data': dorted data values,
        - 'survival': survival function values for each fiducial sample,
        - 'nfid': number of fiducial samples.
    - fid_sample_bottom: Dictionary containing fiducial sample data for the bottom particle
        with the same keys as fid_sample_top.

    Output:
    - Dictionary with the following keys:
        - 'AUC': array of AUC values for each fiducial sample,
        - 'nfid': number of fiducial samples.
    """

    nfid = fid_sample_top['nfid']

    # Combine and sort unique data values from both top and bottom samples
    fullgrid = np.sort(np.unique(np.concatenate((fid_sample_top['data'], fid_sample_bottom['
        data']))))

    # Compute survival functions for the combined grid
    top_surv = particle_grid(fullgrid, fid_sample_top)['survival']
    bottom_surv = particle_grid(fullgrid, fid_sample_bottom)['survival']

    # Initialize array to store AUC values
    auc = np.zeros((nfid, 2))

    # Determine auc values
    for i in range(nfid):
        j = (i % fid_sample_bottom['nfid'])
        auc[i, 0] = 1 + np.sum(np.diff(np.concatenate(([1], top_surv[:, i], [0]))) *
                               (np.concatenate(([1], bottom_surv[:, j])) + np.concatenate(
                                   (bottom_surv[:, j], [0]))) / 2)
        auc[i, 1] = -np.sum(np.diff(np.concatenate(([1], bottom_surv[:, j], [0]))) *
                            (np.concatenate(([1], top_surv[:, i])) + np.concatenate((top_surv
                                [:, i], [0]))) / 2)

    return {'AUC': np.mean(auc, axis=1), 'nfid': nfid}

def calibrationNumber(CI_NP):
    """
    Function to determine fiducial metric 1: average of medians.

    Input:
    - CI_NP: dictionary containing confidence interval data with the key 'median', which
        holds
    the median values of the confidence intervals.

    Output:
    - Value of average of absolute value medians
    """

    # Identify the index of the last non-NaN median value
    ishow = max(np.where(~np.isnan(CI_NP['median']))[0])

    # Sum over medians and determine average of absolute values
    sum = 0
    for i in CI_NP['median'][0:ishow+1]:
        sum += np.abs(i)
    calib = sum/(ishow+1)

    return calib

def calibrationNumber2(CI_NP):
    """
    Function to determine fiducial metric 2: average of medians scaled by widths of intervals
        .

    Input:
    - CI_NP: dictionary containing confidence interval data with the key 'median', which
        holds
    the median values of the confidence intervals.

    Output:
    - Scaled value of average of absolute value medians
    """

    # Identify the index of the last non-NaN median value
    ishow = max(np.where(~np.isnan(CI_NP['median']))[0])

    # Sum over scaled medians and determine average
    sum = 0
    for i in range(ishow+1):
        sum += np.abs(CI_NP['median'][i]) * (CI_NP['point_upper'][i] - CI_NP['point_lower'][i
            ])
    calib = sum/(ishow+1)

    return calib

def calibrationNumber3(CI_NP):
    """
    Function to determine fiducial metric 3: frequency of 0 falling outside of the confidence
         interval.

    Input:
    - CI_NP: dictionary containing confidence interval data with the key 'median', which
        holds
    the median values of the confidence intervals.

    Output:
    - Frequency of zero falling outside of the confidence interval.
    """

    # Identify the index of the last non-NaN median value
    ishow = max(np.where(~np.isnan(CI_NP['median']))[0])

    # Count average amount of times that interval contains zero
    sum = 0
    for i in range(ishow+1):
        if CI_NP['point_upper'][i] >= 0 >= CI_NP['point_lower'][i]:
            sum += 1
    calib = sum/(ishow+1)

    return 1-calib

def LRtestNP(data, hlable=['P', 'D'], nfid=1000, ncores=1, GPDgrid=[], display_plot=False,
    AUC=False):
    """
    Function to perform a likelihood ratio test and analyze results using fiducial inference.

    This function processes LR data for two hypotheses, performs fiducial sampling,
        calculates survival functions,
    and optionally computes the Area Under the Curve (AUC). It can also generate plots to
        visualize the results.

    Input:
    - data: dataFrame containing columns 'LLR' (log-likelihood ratios) and 'labels' (P for
        H_p, D for H_d),
    - hlable: list of two hypothesis labels to distinguish between the two groups in the data
        ,
    - nfid: number of fiducial samples to generate for each hypothesis,
    - ncores: number of cores for parallel processing (not used in the provided code),
    - GPDgrid: custom grid for generating the particle grid. If empty, a default grid is used
        ,
    - display_plot: boolean indicating whether to display diagnostic plots,
    - AUC: Boolean indicating whether to compute and return the Area Under the Curve (AUC).

    Output:
    - A dictionary containing fiducial samples, AUC values (if computed), calibration metrics
        , and other relevant information.
    """

    # Drop NAN values from data
    data = data.dropna()

    # Create arrays for (log) LR data H_p and H_d
    log_topdata = np.sort(data['LLR'][data['labels'] == hlable[0]])
    topdata = 10 ** log_topdata
    log_bottomdata = np.sort(data['LLR'][data['labels'] == hlable[1]])
    bottomdata = 10 ** log_bottomdata

    # Generate pregrid from min value of log_topdata to maxvalue of log_topdata, stepsize = 1
    if len(GPDgrid) == 0:
        pregrid = np.power(10, np.arange(np.floor(max(-2, min(log_topdata))), np.ceil(min(10,
             max(log_topdata))) + 1, 1))
    else:
        pregrid = np.power(10, GPDgrid)

    # Make sure bottomdata does not exceed certain threshold
    bottomdata = np.minimum(bottomdata, 2 * max(pregrid))

    # Define the grid and its indices in the pregrid
    grid = np.sort(np.union1d(pregrid, pregrid))
    idgrid = np.array([np.where(x == grid)[0][0] for x in pregrid])

    # Plot the density
    if display_plot:
        plt.figure(figsize=(10, 8))
        plt.subplot(2, 2, 1)
        dtop = gaussian_kde(np.log10(topdata))
        dbottom = gaussian_kde(np.log10(bottomdata))
        x_range = np.linspace(min(log_topdata), max(log_bottomdata), 100)
        plt.plot(x_range, dtop(x_range), color='red')
        plt.plot(x_range, dbottom(x_range), color='blue')
        plt.xlabel('log(LR)')
        plt.ylabel('density')
        plt.title('Density␣of␣log␣LR')

        plt.subplot(2, 2, 2)
        plt.plot(np.sort(np.log10(topdata)), 1 - (np.arange(1, len(topdata) + 1) / len(
            topdata)), color='red')
        plt.plot(np.sort(np.log10(bottomdata)), 1 - (np.arange(1, len(bottomdata) + 1) / len(
            bottomdata)), color='blue')
        plt.xlabel('log(reported␣LR)')
        plt.ylabel('probability')
        plt.title('Survival␣function␣of␣log␣LR')

    # Make a fiducial sample for each hypothesis
    # Generates dictionary with keys: data, u (nfid arrays of length len(data)), len(data),
        nfid
    fid_sample_top = fiducial_sample(topdata, nfid)
    fid_sample_bottom = fiducial_sample(bottomdata, nfid)

    # Create particle grid and compute survival functions and integrals
    fid_sample_top_grid = particle_grid(grid, fid_sample_top)
    fid_sample_bottom_grid = particle_grid(grid, fid_sample_bottom)

    # Compute and plot AUC
    if AUC != None:
        fid_sample_auc = fid_AUC(fid_sample_top, fid_sample_bottom)
        if display_plot:
            plt.subplot(2, 2, 3)
            plt.boxplot(fid_sample_auc['AUC'])
            plt.ylabel('AUC')
            plt.title('Fiducial␣distribution␣of␣AUC')

    # Compute non-parametric fiducial differences and confidence intervals
    fid_diff_NP = fid_diff_log(fid_sample_top_grid, fid_sample_bottom_grid, idgrid)
    fid_CI_NP = fid_diff_CI(fid_diff_NP)

    # Plot calibration diagnostics
    if display_plot:
        plt.subplot(2, 2, 4)
        dgrid = np.log10(fid_CI_NP['dgrid'])
        plt.plot(dgrid, np.zeros_like(dgrid), color='red', linestyle='--')
        plt.plot(dgrid, fid_CI_NP['median'], color='blue')
        plt.plot(dgrid, fid_CI_NP['uniform_lower'], color='cyan', linestyle='--')
        plt.plot(dgrid, fid_CI_NP['uniform_upper'], color='cyan', linestyle='--')
        plt.plot(dgrid, fid_CI_NP['point_lower'], color='black')
        plt.plot(dgrid, fid_CI_NP['point_upper'], color='black')
        plt.xlabel('log10(reported␣LR)')
        plt.ylabel('interval-specific␣calibration␣discrepancy')
        plt.title('Calibration␣Diagnostic␣Plot')
        plt.show()

    # Calculate metric values
    calib = calibrationNumber(fid_CI_NP)
    calib2 = calibrationNumber2(fid_CI_NP)
    calib3 = calibrationNumber3(fid_CI_NP)


    if AUC != None:
        return {'top': fid_sample_top_grid, 'bottom': fid_sample_bottom_grid, 'AUC':
            fid_sample_auc['AUC'],
                'CI_NP': fid_CI_NP, 'calib': calib, 'calib2': calib2, 'calib3': calib3}
    else:
        return {'top': fid_sample_top_grid, 'bottom': fid_sample_bottom_grid, 'CI_NP':
            fid_CI_NP,
                'calib': calib, 'calib2': calib2, 'calib3': calib3}

def calibrationPlot(CI_NP, my_title="Calibration␣Diagnostic␣Plot", yaxis=None):
    """
    Function to create a calibration diagnostic plot.

    Input:
    - CI_NP: dictionary containing the calibration information with keys:
        'median': median of calibration discrepancies,
        'uniform_lower': lower bound of uniform confidence intervals,
        'uniform_upper': upper bound of uniform confidence intervals,
        'point_lower': lower bound of pointwise confidence intervals,
        'point_upper': upper bound of pointwise confidence intervals,
        'dgrid': the grid of log10(reported LR) values.
    - my_title: Title of the plot (default: "Calibration Diagnostic Plot").
    - yaxis: tuple specifying the y-axis limits; if None, limits are calculated automatically
        .

    Output:
    - Displays the calibration diagnostic plot.
    """

    # Determine the range of valid indices (non-NaN) for plotting
    ishow = max(np.where(~np.isnan(CI_NP['median']))[0])

    # Set y-limit if not inputted
    if yaxis == None:
        yaxis = [np.floor(min(0, np.nanmin(CI_NP['point_lower']))),
                 np.ceil(max(0, np.nanmax(CI_NP['point_upper'])))]

    # Convert grid values to log scale for x-axis plotting
    dgrid_const = np.log10(CI_NP['dgrid'][:ishow + 2])
    dgrid_const2 = np.sort(np.concatenate((dgrid_const[:-1], dgrid_const[1:])))

    # Median calibration discrepancy
    result_1 = np.repeat(CI_NP['median'][0:ishow + 1], 2)

    # Uniform confidence interval bounds
    result_2 = np.repeat(CI_NP['uniform_lower'][0:ishow+1], 2)
    result_3 = np.repeat(CI_NP['uniform_upper'][0:ishow+1], 2)

    # Pointwise confidence interval bounds
    result_4 = np.repeat(CI_NP['point_lower'][0:ishow+1], 2)
    result_5 = np.repeat(CI_NP['point_upper'][0:ishow+1], 2)


    # Create plot
    plt.figure(figsize=(10, 8))
    plt.plot([min(dgrid_const), max(dgrid_const)], [0, 0], color='red', linestyle='--')
    plt.plot(dgrid_const2, result_1, color="blue")
    plt.plot(dgrid_const2, result_2, color="cyan", linestyle="--")
    plt.plot(dgrid_const2, result_3, color="cyan", linestyle="--")
    plt.plot(dgrid_const2, result_4, color="black")
```

```python
    plt.plot(dgrid_const2, result_5, color="black")
    plt.xlabel('log10(reported␣LR)')
    plt.ylabel('interval-specific␣calibration␣discrepancy')
    plt.title(my_title)
    plt.show()

# FUNCTIONS TO DETERMINE OVERLAP
def overlap(array):
    """
    Function that determines (average) overlap percentage between one array with one or
        several others.

    Input:
    - array: array of arrays between which the overlap should be determined.

    Output:
    - Average overlap percentage
    """

    # Sort values of first array
    first_vals = np.sort([x for x in array[0] if x is not None])
    number = len(array)

    # Determine percentiles of first array to get 90%-confidence interval
    perc_first_95 = np.percentile(first_vals, 95)
    perc_first_5 = np.percentile(first_vals, 5)

    # Determine the values of the array and the range
    first_within = first_vals[(first_vals <= perc_first_95) & (first_vals >= perc_first_5)]
    ranges = (first_within[0], first_within[-1])
    number_vals = len(first_within)

    # Initialize overlapping values at zero
    overlaps = 0

    # Loop over other arrays and determine overlap percentage with first array
    for i in range(1, number):
        vals = np.sort([x for x in array[i] if x is not None])
        percentage_95 = np.percentile(vals, 95)
        percentage_5 = np.percentile(vals, 5)
        vals_within = vals[(vals <= percentage_95) & (vals >= percentage_5)]
        ranges_vals = (vals_within[0], vals_within[-1])
        number_vals2 = len(vals_within)
        count_within_range_1 = np.sum((vals_within >= ranges[0]) & (vals_within <= ranges[1])
            )
        count_within_range_2 = np.sum((first_within >= ranges_vals[0]) & (first_within <=
            ranges_vals[1]))

        # Turn into percentage
        overlap_percentage = min(count_within_range_1,count_within_range_2) / min(number_vals
            , number_vals2) * 100
        overlaps += overlap_percentage

    # Determine average of overlap percentage
    overlaps = overlaps / (number - 1)

    return overlaps


def average_overlap(array):
    """
    Function that determines average pairwise overlap percentage between several arrays.

    Input:
    - array: array of arrays between which the overlap should be determined.

    Output:
    - Average overlap percentage
    """

    # Initialize values
    number = len(array)
    total_overlap = 0
    pair_count = 0

    # Range over arrays and determine the percentiles
    for i in range(number):
        perfect_vals = np.sort([x for x in array[i] if x is not None])
        perc_perf_95 = np.percentile(perfect_vals, 95)
        perc_perf_5 = np.percentile(perfect_vals, 5)
        perf_within = perfect_vals[(perfect_vals <= perc_perf_95) & (perfect_vals >=
            perc_perf_5)]
        ranges = (perf_within[0], perf_within[-1])
        number_vals = len(perf_within)

        # Range over leftover arrays and determine overlap
        for j in range(i + 1, number):
            vals = np.sort([x for x in array[j] if x is not None])
            percentage_95 = np.percentile(vals, 95)
            percentage_5 = np.percentile(vals, 5)
            vals_within = vals[(vals <= percentage_95) & (vals >= percentage_5)]
            ranges_vals = (vals_within[0], vals_within[-1])
            number_vals2 = len(vals_within)
            count_within_range_1 = np.sum((vals_within >= ranges[0]) & (vals_within <= ranges
                [1]))
            count_within_range_2 = np.sum((perf_within >= ranges_vals[0]) & (perf_within <=
                ranges_vals[1]))
            overlap_percentage = min(count_within_range_1, count_within_range_2) / min(
                number_vals, number_vals2) * 100
            total_overlap += overlap_percentage
            pair_count += 1

    # Determine average overlap
    average_overlap = total_overlap / pair_count if pair_count > 0 else 0

    return average_overlap

# FUNCTIONS FOR SECOND PART OF RESULTS: GENERATING NEW LR-SYSTEMS

def LSS_calculator(LLRs, probabilities):
    """
    Function that determines frequencies of SS LLRs values based on frequencies of DS-LLRs so
         that the LR of the LR is the LR.

    Input:
    - LLRs: array of LLR-values (assuming log10 LR).
    - probabiities: array of frequencies with which the LLRs occur for DS.

    Output:
    - Array of same-source LLR-values.
    """

    # Initialize LSS array
    new_LSS = []
    num_bins = len(LLRs)

    # Range over LLRs and generate corresponding SS frequency
    for i in range(num_bins):
        # Determine LR from LLR
        LR = 10**LLRs[i]
        prob_hd = probabilities[i]
        # Generate SS frequency of given LR
        prob_hp = LR * prob_hd
        new_LSS.append(prob_hp)

    return new_LSS

def frequency_creator(data_DS):
    """
    Function that generates a consistent LR-system based on DS LRs and frequencies, so that
        the LR of the LR is the
    LR and both the SS and DS frequencies sum up to 100.

    Input:
    - data_DS: array of DS LLR-values

    Output:
    - array containing LDS frequencies, LSS frequencies and the corresponding LR values
    """

    # Initialize alpha
    alpha = 1

    # Create array of LLR-values, determine kde of DS-values and using kde, determine
        frequencies of LLR values
    x_values = np.linspace(min(data_DS), max(data_DS), 10000)
    kde_original = gaussian_kde(data_DS, bw_method='scott')
    LDS_frequencies = kde_original(x_values)

    # While the integrals of the SS and the DS LLRs are not almost equal, loop
    while True:
        # Stretch the LLRs by a factor alpha
        shifted_LLRs = x_values - min(data_DS)
        stretched_shifted = shifted_LLRs * alpha
        stretched_LLRs = stretched_shifted + min(data_DS)

        # Determine LSS frequencies using the LDS frequencies and the stretched LRs
        LSS_frequencies = LSS_calculator(stretched_LLRs, LDS_frequencies)

        # Integrate the two frequencies
        integral_kde = simps(LDS_frequencies, stretched_LLRs)
        integral_LSS = simps(LSS_frequencies, stretched_LLRs)

        # If they are almost equal, done
        if np.isclose(integral_kde, integral_LSS, rtol=0.01, atol=0.01):
            break
        # If there are too many LSS values, decrease alpha
        elif integral_kde < integral_LSS:
            alpha -= 0.001
        # If there are too many LDS values, increase alpha
        else:
            alpha += 0.001

    # Normalize
    LSS_frequencies = LSS_frequencies / np.sum(LSS_frequencies)
    LDS_frequencies = LDS_frequencies / np.sum(LDS_frequencies)

    return [LDS_frequencies, LSS_frequencies, stretched_LLRs]

def calculate_metrics_all(data_SS, data_DS):
    """
    Function that calculates all optimized metrics for SS and DS data.

    Input:
    - data_SS: array of SS LR-values.
    - data_DS: array of DS LR-valies

    Output:
    - values of devPAV, cllr and Fid
    """

    # Initialize metric values to prevent error
    dp = None
    c = None
    cal = None

    # Make arrays of LRs and hypotheses
    lrs = np.concatenate((data_SS, data_DS))
    all_hypotheses = np.concatenate((np.array(['H1'] * len(data_SS)), np.array(['H2'] * len(
        data_DS))))
    all_hypotheses_01 = np.where(all_hypotheses == 'H1', 1, 0)

    # Determine the metric values
    try:
        dp = devpav_new(lrs, all_hypotheses_01)
    except Exception as e:
        print(f"A␣devPAV␣error␣occurred:␣{e}")
    try:
        c = cllr(lrs, all_hypotheses_01)
    except Exception as e:
        print(f"A␣Cllr␣error␣occurred:␣{e}")
    try:
        cal = LRtestNP(pd.DataFrame({'LLR': np.log10(lrs),
                                       'labels': ['P'] * len(data_SS) + ['D'] * len(data_DS)
                                           }),
                             nfid=100, AUC=True)['calib'][0]
    except Exception as e:
        print(f"A␣Fid␣error␣occurred:␣{e}")

    return [c, dp, cal]
```

Listing A.2: cllr test.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
from math import comb, sqrt
from functions_thesis import *
import pandas as pd
from lir import *
import seaborn as sns

# Initialize mean, variation, amount of data and scaling factors
MU_ss = 6
SIGMA = sqrt(2 * MU_ss)
n_ss = 50
n_sd = 150
c = 1
c_2 = 2
d = 1.5
d_2 = 2.5

# Initialize amount of times to calculate metrics
N = 1000

# Initialize arrays to store normal Cllr values
cllr_p = []
cllr_r1 = []
cllr_r2 = []
cllr_l1 = []
cllr_l2 = []
cllr_e1 = []
cllr_e2 = []
cllr_w1 = []
cllr_w2 = []

# Initialize arrays to store Cllr values using the Brier score
brier_p = []
brier_r1 = []
brier_r2 = []
brier_l1 = []
brier_l2 = []
brier_e1 = []
brier_e2 = []
brier_w1 = []
brier_w2 = []

# Initialize arrays to store Cllr values using the zero-one score
zerone_p = []
zerone_r1 = []
zerone_r2 = []
zerone_l1 = []
zerone_l2 = []
zerone_e1 = []
```

```
zerone_e2 = []
zerone_w1 = []
zerone_w2 = []

# Initialize arrays to store Cllr values using the spherical scoring rule
spher_p = []
spher_r1 = []
spher_r2 = []
spher_l1 = []
spher_l2 = []
spher_e1 = []
spher_e2 = []
spher_w1 = []
spher_w2 = []

# Create an array of log 10 prior odds ranging from -5 to 5
log_prior_odds = np.linspace(-5, 5, num=100)  # Adjust the number of points as needed
# Calculate prior odds array
prior_odds = 10 ** log_prior_odds

for i in range(N):
    # Generate consistent data
    LSS_p = np.random.normal(MU_ss, SIGMA, n_ss)
    LDS_p = np.random.normal(-MU_ss, SIGMA, n_sd)
    SS_p = np.power(math.e, LSS_p)
    DS_p = np.power(math.e, LDS_p)

    # Generate data skewed to the right by c and c_2
    LSS_r1 = LSS_p + c
    LDS_r1 = LDS_p + c
    SS_r1 = np.power(math.e, LSS_r1)
    DS_r1 = np.power(math.e, LDS_r1)

    LSS_r2 = LSS_p + c_2
    LDS_r2 = LDS_p + c_2
    SS_r2 = np.power(math.e, LSS_r2)
    DS_r2 = np.power(math.e, LDS_r2)

    # Generate data skewed to left by c and c_2
    LSS_l1 = LSS_p - c
    LDS_l1 = LDS_p - c
    SS_l1 = np.power(math.e, LSS_l1)
    DS_l1 = np.power(math.e, LDS_l1)

    LSS_l2 = LSS_p - c_2
    LDS_l2 = LDS_p - c_2
    SS_l2 = np.power(math.e, LSS_l2)
    DS_l2 = np.power(math.e, LDS_l2)

    # Generate too extreme data, scaled by d and d_2
    LSS_e1 = d * LSS_p
    LDS_e1 = d * LDS_p
    SS_e1 = np.power(math.e, LSS_e1)
    DS_e1 = np.power(math.e, LDS_e1)

    LSS_e2 = d_2 * LSS_p
    LDS_e2 = d_2 * LDS_p
    SS_e2 = np.power(math.e, LSS_e2)
    DS_e2 = np.power(math.e, LDS_e2)

    # Generate too weak data, scaled by d and d_2
    LSS_w1 = (1 / d) * LSS_p
    LDS_w1 = (1 / d) * LDS_p
    SS_w1 = np.power(math.e, LSS_w1)
    DS_w1 = np.power(math.e, LDS_w1)

    LSS_w2 = (1 / d_2) * LSS_p
    LDS_w2 = (1 / d_2) * LDS_p
    SS_w2 = np.power(math.e, LSS_w2)
    DS_w2 = np.power(math.e, LDS_w2)

    # Determine metrics for consistent data
    all_data = np.concatenate((SS_p, DS_p))
    all_hypotheses = np.concatenate((np.array(['H1'] * len(SS_p)), np.array(['H2'] * len(DS_p
        ))))
    all_hypotheses_01 = np.where(all_hypotheses == 'H1', 1, 0)

    # PAV lrs
    cal = lir.IsotonicCalibrator()
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Determine normal cllr
    cllr1 = cllr(all_data, all_hypotheses_01)
    cllr_p.append(cllr1)

    # Determine Brier-score difference between normal and PAV data
    brierp_1 = brier(all_data, all_hypotheses_01)
    brierp_2 = brier(lrmin, all_hypotheses_01)
    brierp = brierp_1 - brierp_2
    brier_p.append(brierp)

    # Determine zero-one cllr
    zeronep_1 = zero_one(all_data, all_hypotheses_01)
    zeronep_2 = zero_one(lrmin, all_hypotheses_01)
    zeronep = zeronep_1 - zeronep_2
    zerone_p.append(zeronep)

    # Determine spherical cllr
    spherp_1 = spherical(all_data, all_hypotheses_01)
    spherp_2 = spherical(lrmin, all_hypotheses_01)
    spherp = spherp_1 - spherp_2
    spher_p.append(spherp)

    # Determine metrics for data skewed to the right
    all_data = np.concatenate((SS_r1, DS_r1))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr2 = cllr(all_data, all_hypotheses_01)
    cllr_r1.append(cllr2)

    # Brier cllr
    brierr1_1 = brier(all_data, all_hypotheses_01)
    brierr1_2 = brier(lrmin, all_hypotheses_01)
    brierr1 = brierr1_1 - brierr1_2
    brier_r1.append(brierr1)

    # Zero-one cllr
    zeroner1_1 = zero_one(all_data, all_hypotheses_01)
    zeroner1_2 = zero_one(lrmin, all_hypotheses_01)
    zeroner1 = zeroner1_1 - zeroner1_2
    zerone_r1.append(zeroner1)

    # Spherical cllr
    spherr1_1 = spherical(all_data, all_hypotheses_01)
    spherr1_2 = spherical(lrmin, all_hypotheses_01)
    spherr1 = spherr1_1 - spherr1_2
    spher_r1.append(spherr1)

    all_data = np.concatenate((SS_r2, DS_r2))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr3 = cllr(all_data, all_hypotheses_01)
    cllr_r2.append(cllr3)

    # Brier cllr
    brierr2_1 = brier(all_data, all_hypotheses_01)
    brierr2_2 = brier(lrmin, all_hypotheses_01)
    brierr2 = brierr2_1 - brierr2_2
    brier_r2.append(brierr2)

    # Zero-one cllr
    zeroner2_1 = zero_one(all_data, all_hypotheses_01)
    zeroner2_2 = zero_one(lrmin, all_hypotheses_01)
    zeroner2 = zeroner2_1 - zeroner2_2
    zerone_r2.append(zeroner2)

    # Spherical cllr
    spherr2_1 = spherical(all_data, all_hypotheses_01)
    spherr2_2 = spherical(lrmin, all_hypotheses_01)
    spherr2 = spherr2_1 - spherr2_2
    spher_r2.append(spherr2)

    # Determine metrics for data skewed to the left
    all_data = np.concatenate((SS_l1, DS_l1))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr4 = cllr(all_data, all_hypotheses_01)
    cllr_l1.append(cllr4)

    # Brier cllr
    brierl1_1 = brier(all_data, all_hypotheses_01)
    brierl1_2 = brier(lrmin, all_hypotheses_01)
    brierl1 = brierl1_1 - brierl1_2
    brier_l1.append(brierl1)

    # Zero-one cllr
    zeronel1_1 = zero_one(all_data, all_hypotheses_01)
    zeronel1_2 = zero_one(lrmin, all_hypotheses_01)
    zeronel1 = zeronel1_1 - zeronel1_2
    zerone_l1.append(zeronel1)

    # Spherical cllr
    spherl1_1 = spherical(all_data, all_hypotheses_01)
    spherl1_2 = spherical(lrmin, all_hypotheses_01)
    spherl1 = spherl1_1 - spherl1_2
    spher_l1.append(spherl1)

    all_data = np.concatenate((SS_l2, DS_l2))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr5 = cllr(all_data, all_hypotheses_01)
    cllr_l2.append(cllr5)

    # Brier cllr
    brierl2_1 = brier(all_data, all_hypotheses_01)
    brierl2_2 = brier(lrmin, all_hypotheses_01)
    brierl2 = brierl2_1 - brierl2_2
    brier_l2.append(brierl2)

    # Zero-one cllr
    zeronel2_1 = zero_one(all_data, all_hypotheses_01)
    zeronel2_2 = zero_one(lrmin, all_hypotheses_01)
    zeronel2 = zeronel2_1 - zeronel2_2
    zerone_l2.append(zeronel2)

    # Spherical cllr
    spherl2_1 = spherical(all_data, all_hypotheses_01)
    spherl2_2 = spherical(lrmin, all_hypotheses_01)
    spherl2 = spherl2_1 - spherl2_2
    spher_l2.append(spherl2)

    # Determine metrics for too extreme data
    all_data = np.concatenate((SS_e1, DS_e1))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr6 = cllr(all_data, all_hypotheses_01)
    cllr_e1.append(cllr6)

    # Brier cllr
    briere1_1 = brier(all_data, all_hypotheses_01)
    briere1_2 = brier(lrmin, all_hypotheses_01)
    briere1 = briere1_1 - briere1_2
    brier_e1.append(briere1)

    # Zero-one cllr
    zeronee1_1 = zero_one(all_data, all_hypotheses_01)
    zeronee1_2 = zero_one(lrmin, all_hypotheses_01)
    zeronee1 = zeronee1_1 - zeronee1_2
    zerone_e1.append(zeronee1)

    # Spherical cllr
    sphere1_1 = spherical(all_data, all_hypotheses_01)
    sphere1_2 = spherical(lrmin, all_hypotheses_01)
    sphere1 = sphere1_1 - sphere1_2
    spher_e1.append(sphere1)

    all_data = np.concatenate((SS_e2, DS_e2))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr7 = cllr(all_data, all_hypotheses_01)
    cllr_e2.append(cllr7)

    # Brier cllr
    briere2_1 = brier(all_data, all_hypotheses_01)
    briere2_2 = brier(lrmin, all_hypotheses_01)
    briere2 = briere2_1 - briere2_2
    brier_e2.append(briere2)

    # Zero-one cllr
    zeronee2_1 = zero_one(all_data, all_hypotheses_01)
    zeronee2_2 = zero_one(lrmin, all_hypotheses_01)
    zeronee2 = zeronee2_1 - zeronee2_2
    zerone_e2.append(zeronee2)

    # Spherical cllr
    sphere2_1 = spherical(all_data, all_hypotheses_01)
    sphere2_2 = spherical(lrmin, all_hypotheses_01)
    sphere2 = sphere2_1 - sphere2_2
    spher_e2.append(sphere2)

    # Determine metrics for too weak data
    all_data = np.concatenate((SS_w1, DS_w1))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr8 = cllr(all_data, all_hypotheses_01)
    cllr_w1.append(cllr8)

    # Brier cllr
    brierw1_1 = brier(all_data, all_hypotheses_01)
    brierw1_2 = brier(lrmin, all_hypotheses_01)
    brierw1 = brierw1_1 - brierw1_2
    brier_w1.append(brierw1)

    # Zero-one cllr
    zeronew1_1 = zero_one(all_data, all_hypotheses_01)
    zeronew1_2 = zero_one(lrmin, all_hypotheses_01)
    zeronew1 = zeronew1_1 - zeronew1_2
    zerone_w1.append(zeronew1)

    # Spherical cllr
    spherw1_1 = spherical(all_data, all_hypotheses_01)
    spherw1_2 = spherical(lrmin, all_hypotheses_01)
    spherw1 = spherw1_1 - spherw1_2
    spher_w1.append(spherw1)

    all_data = np.concatenate((SS_w2, DS_w2))
    lrmin = cal.fit_transform(to_probability(all_data), all_hypotheses_01)

    # Normal cllr
    cllr9 = cllr(all_data, all_hypotheses_01)
    cllr_w2.append(cllr9)

    # Brier cllr
    brierw2_1 = brier(all_data, all_hypotheses_01)
    brierw2_2 = brier(lrmin, all_hypotheses_01)
    brierw2 = brierw2_1 - brierw2_2
    brier_w2.append(brierw2)

    # Zero-one cllr
    zeronew2_1 = zero_one(all_data, all_hypotheses_01)
    zeronew2_2 = zero_one(lrmin, all_hypotheses_01)
    zeronew2 = zeronew2_1 - zeronew2_2
    zerone_w2.append(zeronew2)

    # Spherical cllr
    spherw2_1 = spherical(all_data, all_hypotheses_01)
    spherw2_2 = spherical(lrmin, all_hypotheses_01)
    spherw2 = spherw2_1 - spherw2_2
    spher_w2.append(spherw2)

# Collect results of normal Cllr in dictionary
results_cllr = {
    'Perfect': cllr_p,
    'Right␣c=1': cllr_r1,
    'Left␣c=1': cllr_l1,
    'Extreme␣c=1.5': cllr_e1,
    'Weak␣c=1.5': cllr_w1,
    'Right␣c=2': cllr_r2,
    'Left␣c=2': cllr_l2,
    'Extreme␣c=2.5': cllr_e2,
    'Weak␣c=2.5': cllr_w2,
}
df_results = pd.DataFrame(results_cllr)

# Compute overlap percentage
cllr_normals = np.array(list(results_cllr.values()))
overlap_normals = overlap(cllr_normals)
print('Normals␣overlap:', overlap_normals)

# Plot results
fig, axes = plt.subplots(ncols=len(df_results.columns), figsize=(15, 6), sharey=True)

for i, column in enumerate(df_results.columns):
    sns.violinplot(data=df_results[column], ax=axes[i])
    axes[i].set_title(column)
    axes[i].set_ylabel('')  # Remove y-axis label
plt.suptitle('Cllr␣normal', fontsize=16)
plt.tight_layout()
plt.show()

# Collect results of Brier Cllr in dictionary
results_brier = {
    'Perfect': brier_p,
    'Right␣c=1': brier_r1,
    'Left␣c=1': brier_l1,
    'Extreme␣c=1.5': brier_e1,
    'Weak␣c=1.5': brier_w1,
    'Right␣c=2': brier_r2,
    'Left␣c=2': brier_l2,
    'Extreme␣c=2.5': brier_e2,
    'Weak␣c=2.5': brier_w2,
}
df_resultsb = pd.DataFrame(results_brier)

# Compute overlap percentage
cllr_briers = np.array(list(results_brier.values()))
overlap_briers = overlap(cllr_briers)
print('Briers␣overlap:', overlap_briers)
```

```

# Plot results
fig, axes = plt.subplots(ncols=len(df_resultsb.columns), figsize=(15, 6), sharey=True)

for i, column in enumerate(df_resultsb.columns):
    sns.violinplot(data=df_resultsb[column], ax=axes[i])
    axes[i].set_title(column)
    axes[i].set_ylim(0.0,0.05)
    axes[i].set_ylabel('')  # Remove y-axis label
plt.suptitle('Brier', fontsize=16)
plt.tight_layout()
plt.show()

# Collect results of zero-one Cllr in dictionary
results_zerone = {
    'Perfect': zerone_p,
    'Right␣c=1': zerone_r1,
    'Left␣c=1': zerone_l1,
    'Extreme␣c=1.5': zerone_e1,
    'Weak␣c=1.5': zerone_w1,
    'Right␣c=2': zerone_r2,
    'Left␣c=2': zerone_l2,
    'Extreme␣c=2.5': zerone_e2,
    'Weak␣c=2.5': zerone_w2,
}
df_resultszer = pd.DataFrame(results_zerone)

# Compute overlap percentage
cllr_zerons = np.array(list(results_zerone.values()))
overlap_zerons = overlap(cllr_zerons)
print('Zero␣ones␣overlap:', overlap_zerons)

# Plot results
fig, axes = plt.subplots(ncols=len(df_resultszer.columns), figsize=(15, 6), sharey=True)

for i, column in enumerate(df_resultszer.columns):
    sns.violinplot(data=df_resultszer[column], ax=axes[i])
    axes[i].set_title(column)
    axes[i].set_ylabel('')  # Remove y-axis label
plt.suptitle('Zero-one', fontsize=16)
plt.tight_layout()
plt.show()

# Collect results of spherical Cllr in dictionary
results_spher = {
    'Perfect': spher_p,
    'Right␣c=1': spher_r1,
    'Left␣c=1': spher_l1,
    'Extreme␣c=1.5': spher_e1,
    'Weak␣c=1.5': spher_w1,
    'Right␣c=2': spher_r2,
    'Left␣c=2': spher_l2,
    'Extreme␣c=2.5': spher_e2,
    'Weak␣c=2.5': spher_w2,
}
df_resultssph = pd.DataFrame(results_spher)

# Compute overlap percentage
cllr_sphers = np.array(list(results_spher.values()))
overlap_sphers = overlap(cllr_sphers)
print('Sphericals␣overlap:', overlap_sphers)

# Plot results
fig, axes = plt.subplots(ncols=len(df_resultssph.columns), figsize=(15, 6), sharey=True)

for i, column in enumerate(df_resultssph.columns):
    sns.violinplot(data=df_resultssph[column], ax=axes[i])
    axes[i].set_title(column)
    axes[i].set_ylabel('')
plt.suptitle('Spherical', fontsize=16)
plt.tight_layout()
plt.show()
```