FEERCI: A Package for Fast Non-Parametric Confidence Intervals for Equal Error Rates in Amortized O(m log n)

1st Erwin Haasnoot
Faculty of EEMCS, DMB Group
University of Twente
Enschede, The Netherlands
e.haasnoot@utwente.nl

2nd Ali Khodabakhsh
Norwegian Biometrics Laboratory
NTNU
Gjovik, Norway
ali.khodabakhsh@ntnu.no

3rd Chris Zeinstra
Faculty of EEMCS, DMB Group
University of Twente
Enschede, The Netherlands
c.g.zeinstra@utwente.nl

4th Luuk Spreeuwers
Faculty of EEMCS, DMB Group
University of Twente
Enschede, The Netherlands
l.j.spreeuwers@utwente.nl

5th Raymond Veldhuis
Faculty of EEMCS, DMB Group
University of Twente
Enschede, The Netherlands
r.n.j.veldhuis@utwente.nl

***Abstract*—Equal Error Rates (EERs), or other weighted relations between False Match and Non-Match Rates (FMR/FNMR), are often used as a performance metric for biometric systems. Confidence Intervals (CIs) are used to denote the uncertainty underlying these EERs, with many methods existing to estimate said CIs in both parametric and non-parametric ways. These confidence intervals provide, foremost, a method of comparing scoring/ranking functions. Non-parametric methods often suffer from high computational costs, but do not make assumptions as to the shape of the EER- and score distributions. For both EERs and CIs, contemporary open-source toolkits leave room for improvement in terms of computational efficiency. In this paper, we introduce the Fast EER (FEER) algorithm that calculates an EER in $O(\log n)$ on a sorted score list, we show how to adapt the FEER algorithm to calculate non-parametric, bootstrapped EER CIs (FEERCI) in $O(m \log n)$ given m resamplings, and we introduce an opinionated open-source package named feerci that provides implementations of the FEER and FEERCI algorithm. We provide speed and accuracy benchmarks for the feerci package, comparing it against the most-used methods of calculating EERs in Python and show how it is able to calculate EERs and CIs on very large score lists faster than contemporary toolkits can calculate a single EER.**

***Index Terms*—Receiver operating characteristic, Equal Error Rate, Bootstrap Confidence Interval, Open Source**

I. Introduction

An Equal Error Rate (EER) can be defined as the point where the False Match Rate (FMR) and False Non-match Rate (FNMR) of a biometric system are equal, and the difference between decision thresholds to attain said rates is minimized or is zero. EERs are used to summarize performance of a biometric system, often in combination with a full report of the Decision Error Trade-off (DET) and/or the Receiver Operator Characteristic (ROC) curve, but oftentimes without such curves. Given the prevalence of reporting EERs without ROC/DET curves, toolkits/methods currently in use leave room for improvement, by not accepting pre-sorted score lists, and only calculating EERs in $O(n)$ after sorting.

Efficiently calculating EERs is relatively easy when given sorted genuine and impostor score lists. We can imagine the problem of finding an EER as doing a binary search along the line where FMR = FNMR, to find the intersection point with the DET/ROC curve. These algorithms are worth using when the cost of the $O(n \log n)$ initial sorting can be amortized across enough runs of the algorithm.

Use cases of such an algorithm include 1) calculating multiple different weighted (EER-like) subtractive[1] performance measures on the same sorted score list and 2) comparing multiple genuine and impostor score lists to eachother, where the amount of comparisons outweighs the amount of sorts. Further, it is possible to adapt such an EER algorithm so that it can efficiently calculate non-parametric EER confidence intervals (CIs) through a bootstrap-equivalent procedure, which we show in this paper.

We first explicate an example of a $O(\log n)$ algorithm for calculating EERs on the empirical ROC, which we have called the Fast Equal Error Rate (FEER) algorithm. Second, we show how to extend this algorithm to calculate a single bootstrapped EER, which allows us to calculate an EER CI in $O(m \log n)$ when repeated m times. This algorithm we named FEERCI, for Fast EER CIs. We provide both a speed and accuracy benchmark for the algorithms.

Included in this paper is the introduction of the feerci package, an opinionated, open source package for the Python programming language. This package provides implementa-

[1]For measures such as Minimal Cost-Decision Function (minDCF) the $\min_{0 \leq e \leq 1} |FMR + FNMR|$ is calculated, whereas for EER the function to minimize is $\min_{0 \leq e \leq 1} |FMR - FNMR|$. The former is not monotonic, whereas the latter is.

Fig. 1. Fast Equal Error Rates (FEER)

Input: [is, gs] // sort impostors descending, genuines ascending

Output: EER

Initialize EER point and step, and check for overlap

```
e=.5, step=.5
if (is[0] < gs[0]) then
    return 0.0
else if (gs[size(gs) − 1] < is[size(is) − 1]) then
    return 1.0
end if
```

Loop to find overlap point, or point where overlap is as close as possible

```
for i = 0; i < 2 × ceil(max(log2(size(is),log2(size(gs))); i = i + 1 do
    gmin, gmax = gs[[floor(e*size(gs)),ceil(e*size(gs))]]
    imax, imin = is[[floor(e*size(is)),ceil(e*size(is))]]
    step /= 2
    if (gmin > imax) then
        e += step
    else if (imin > gmax) then
        e -= step
    end if
end for
return EERPostProcessing(gmin,gmax,imax,imin)
```

tions for both the FEER and FEERCI algorithms, and we use it as the basis of our benchmark code, which we also release for inspection. In the feerci package's current state, it can calculate non-parametric CIs (and EERs) on large score lists (of millions of scores) in only seconds on a mid-range laptop, for large enough score lists CIs are calculated faster than the commonly used toolkit BOB can calculate a single EER .

II. Fast Equal Error Rates (FEER)

The Fast Equal Error Rate (FEER) algorithm works as follows: We want to find a point $e \in [0, 1]$ on the EER line where thresholds $t_{EER} = t_G = t_I$, given a list of genuine similarity scores G, and impostor similarity scores I. Functions G_t and I_t are defined such that both map a given e to the minimum interval of empirical genuine and impostor scores surrounding the "real" threshold for e. $t_G \in G_t(e)$ and $t_I \in I_t(e)$, an optimal e would be the point where $t_G \in I_t(e)$ and $t_I \in G_t(e)$. This is the case when $G_t(e) \cap I_t(e) \neq \phi$. Both G_t and I_t are monotonic, so that it becomes a binary search problem and logarithmic in G_t and I_t. Given sorted G and I, G_t and I_t can be performed in constant time. As such this algorithm has a running time of $O(\log n)$. See Algorithm 1 for a pseudocode implementation of the algorithm.

The EERPostProcessing step in Algorithm 1 consists of two binary searches to expand a genuine score interval around $min(G_t(e))$ bounded by $I_t(e)$ if $G_t(e) \subset I_t(e)$ and vice-versa if $I_t(e) \subset G_t(e)$. The bounding box defined by $(\min(t_G), \max(t_I))$ and $(\max(t_G), \min(t_I))$ allows us to calculate where the EER-line intersects the empirical ROC curve. This results in a an algorithm that can calculate pessimistic, expected and optimistic EERs on empirical ROCs. A visual representation of the algorithm can be seen at Figure 2, including the post-processing, with the intersection between the EER line and the ROC depicting the pessimistic estimate of EER, as well as what constitutes expected and optimistic. Full implementation details can be found at https://github.com/feerci/feerci .

The EER calculated by FEER differs from that one determined through the ROC Convex Hull (ROCCH), which can be defined as that subset of points on the empirical ROC that maximizes the convexity of the ROC. Bob makes use of this to calculate EERs [2]. If provided this set of maximum convexity, the FEER algorithm can calculate the ROCCH EERs. Unfortunately, given a sorted score list, it is not obvious how one can efficiently determine the points closest to the intersection of the ROCCH with the EER line.

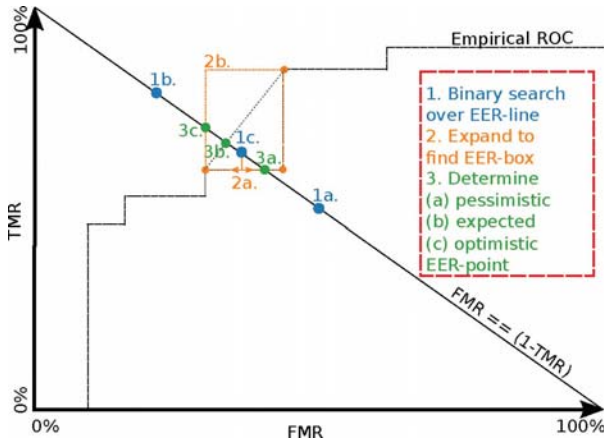


Fig. 2. Visual representation of FEER algorithm. 1) Search on the line FMR==(1-TMR) a point e in the EER Bounding box. It checks the cases $e = .5$, $e = .25$ and $e = .375$. 2) Project the found point e on the horizontal axis or vertical axis and expand within the bounds of the given genuine (or impostor if vertical projection) scores. 3) Find where the line FMR==(1-TMR) intersects the EER Bounding Box (diagonal) and report the pessimistic, expected or optimistic point.

III. Fast EER Confidence Intervals (FEERCI)

Confidence bands (CB) of ROC/DET curves, and confidence intervals (CIs) of EERs provide, foremost, a method of comparing scoring/ranking functions, e.g. biometric classifiers, against each other. Where CBs overlap, one scoring function can not be said to be better than the other. Of all methods that exist to estimate CBs, bootstrapped re-sampling based methods are generally preferable over others that assume some limiting (likely Gaussian) distribution [3]. Although naive bootstrapping methods, where a raw empirical distribution is re-sampled, are sub-optimal for estimating CBs compared to smoothed bootstraps, where independent (Gaussian) noise is added to each bootstrapped sample [5], the only point on ROC CBs where there usually is no difference between bootstrapping methods is at the EER point [6]. Therefore a

good case can be made for naively bootstrapped CIs for EERs on empirical ROCs. We refer to [3] for a more thorough discussion of EERs on empirical ROCs vs EERs on ROCCH.

Smoothed bootstraps are computationally (more) expensive than naive bootstraps, but this does not mean that naive bootstraps are computationally cheap. A naive algorithm for it would look like below.

1 Bootstrap both genuine and imposter lists.
2 Sort bootstrapped genuine and impostor lists.
3 Calculate bootstrapped EER
4 Repeat previous two steps m times and store each generated EER.
5 Sort re-sampled EERs and determine confidence interval.

With the $O(n \log n)$ sort at step 2 dominating the algorithmic complexity, and making the full algorithm for a naive CI cost $O(m * n \log n)$. Given a sorted sampling[2], this could be brought down to $O(m * n)$. In this section, we show to adapt the FEER algorithm, so that it can draw bootstrapped EERs in amortized $O(\log n)$, resulting in an efficient $O(m \log n)$ algorithm for EER confidence intervals.

A. Algorithm

A bootstrap sampling is equivalent to repeatedly 1) drawing from the discrete uniform distribution Uniform$(0, 1)$, 2) transforming this to an index over the range $[0, n)$ and 3) indexing the original score list with said index. The kth order statistic in case of n repeated draws Uniform$(0, 1)$ is well known to be Beta$(k, n - k + 1)$ distributed [4]. In practice, a draw $z \sim$ Beta$(\alpha, \beta) = x/(x + y)$, where $x \sim$ Gamma(α) and $y \sim$ Gamma(β).

The FEER algorithm as exposed in the previous section repeatedly draws kth-order statistics from the original score lists using the functions G_t and I_t. If we correctly administrate the ranges over which we're drawing, one for the re-sampled set, one for the original set, we can calculate an equivalent of a bootstrapped EER in effectively only $O(\log n)$. Sampling a gamma distribution results in a "pseudo-constant" operation, as it depends on the average amount of accept-reject trials [1] and is inversely dependent on n (becomes lower as n is increased). Please see Algorithm 3 for a pseudocode implementation of the FEERCI algorithm.

We omit details from the pseudocode of Algorithm 3, e.g. how to handle cases so that ks are not accidentally resampled. For details on this part of the algorithm, please refer to the source code of FEERCI implementation at https://github.com/feerci/feerci .

IV. Benchmarks

In this section, we present the result of our benchmark tests, testing the implementation of the FEER and FEERCI algorithm as is available in our Python feerci package.

[2]We can draw a sorted sample from a sorted list by first drawing a histogram of index counts (which are sorted), then iteratively building the sample list from this histogram

Fig. 3. Fast EER Confidence Intervals (FEERCI)

Input: is, gs, m //both is & gs sorted, m is amount of bootstrap samples
Output: EERS

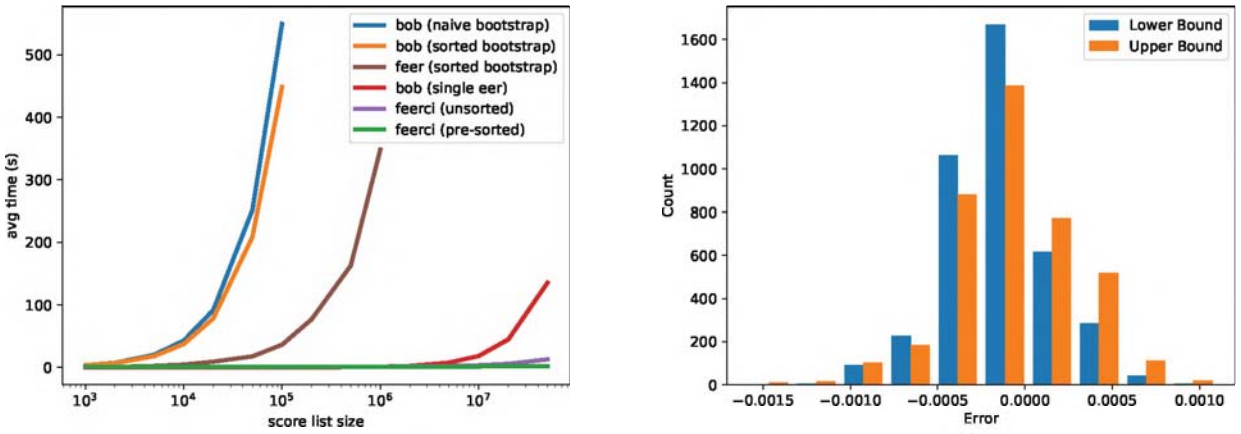
```
EERS = new list[m]
for ind = 0; ind < m; ind+ = 1 do
    iomin,iomax = ibmin,ibmax = 0,size(is)-1
    gomin,gomax = gbmin,gbmax = 0,size(gs)-1
    while gbmax − gbmin > 1 or ibmax − ibmin > 1 do
        kg1, ki1 = (gbmax + gbmin) / 2, (ibmax + ibmin) / 2
        kg2, ki2 = kg1 + 1, ki1 + 1
        ig1 = round(gomin + Beta(kg1 - gbmin + 1, gbmax - kg1 + 1) * (gomax - gomin))
        ii1 = round(iomin + Beta(ki1 - ibmin + 1, ibmax - ki1 + 1) * (iomax - iomin))
        ig2 = round(gomin + Beta(1, gbmax - kg1 + 1) * (gomax - gs[ig1]))
        ii2 = round(iomin + Beta(1, ibmax - ki1 + 1) * (iomax - is[ii1]))

        if gs[ig1] > is[ii1] then
            gbmax, gomax = kg1, ig1
            ibmax, iomax = ki1, ii1
        end if
        if is[ii2] > gs[ig2] then
            gbmin, gomin = kg1, ig1
            gomin, iomin = ki1, ii1
        end if
        EERS[ind] = EERPostProcessing(kg1,kg2,ki1,ki2)
        break
    end while
end for
return EERS
```

A. Speed Benchmark

Our speed benchmark consisted of calculating CIs (or single EERs) in six different cases in Python, see below.

1 Naive sampling using bob.measure.eerrocch
2 Naive sorted sampling using bob.measure.eerrocch
3 Naive sorted sampling using FEER
4 Single EER on unsorted list with bob.measure.eerrocch
5 FEERCI on unsorted list
6 FEERCI on pre-sorted list

We ran each benchmark case 10 times on set sizes ranging from 1,000 to 50,000,000, as long as a run could be finished within 10 minutes for a case. We set this limit because we expected all non-FEERCI algorithms to take days rather than seconds on large enough sets (of >500,000 scores). We generated artificial scores by drawing impostor and genuine scores from a normal distribution parameterized such that the EER should be around 20%, drawn using [7].



(a) Speed Benchmark

(b) Accuracy Benchmark

Fig. 4. a) Speed benchmark on 6 cases. b) Accuracy benchmark, histogram plotted of errors between naive bootstrap + feer CI and FEERCI CIs, for both the upper and lower CI bound

B. Accuracy Benchmark

Our accuracy benchmark consisted of testing how the CIs of our FEERCI algorithm compared to calculating a naive bootstrap using FEER. We drew samples from two normal distributions parameterized in such a way that the resulting EER would be around 2%, 5%, 10% and 20%. We repeated this procedure 1,000 times for each EER. We reported on errors between the lower and upper bounds of CIs calculated through both algorithms.

C. System

Both benchmarks ran on a Lenovo W550s laptop running Ubuntu 16.04 with 16 GB of RAM and a 2.6 GHz quad-core Intel i7 processor. We used Python 3.5.2. We pinned the bob library to version 4.0.1, and bob.measure to 3.0.0. For full code and version information, please look at our repository here: https://github.com/feerci/benchmark .

D. FEERCI package

The python feerci package is an easy to install package that allows one to easily calculate both the EER and its CI in one command. The package can be found by following the link here: https://github.com/feerci/feerci [3]. It provides a single method with the signature:

```
feerci.feerci(impostors, genuines,
  is_sorted = False,n_iterations=10000,
  ci=.95) ->
    (eer, bootstrapped_eers, ci)
```

We use this package as-is throughout the benchmark. The default parameters were chosen so that a valid CI is calculated by default on the score lists.

[3]NOTE: This is for the review paper only, we will change this to the real repository link for a possible camera-ready paper.

V. Results & Discussion

The results of our speed benchmark can be found in Figure 4a. We show how the average run-time of all 6 benchmark cases change as the set size is increased. Running times quickly pass the 10 minute threshold for the two bob-based algorithms. It is also apparent bob is not able to handle pre-sorted sets efficiently, as bob with a sorted bootstrap procedure can only handle sets twice as large. The FEER algorithm is able to handle set sizes up to 10x larger than bob's within 10 minutes of running-time, showing that it is able to handle sorted sets significantly faster than the bob cases.

We also note that for set sizes larger than 1 million, a full FEERCI run is faster than a single run of the bob EER function on unsorted lists. Similarly, there is only a slight difference between running times for FEERCI on pre-sorted and unsorted lists if n is not large enough. This is due to the final $O(m \log m)$ sort on the bootstrapped EERs contributing significantly to the running time if $m \approx n$. If $n \gg m$, the initial score list sort becomes dominant, and we see running times increase significantly for FEERCI on unsorted lists.

The results of our accuracy benchmark can be found in Figure 4b. We show a histogram of differences between bounds the lower and upper bounds of our calculated and show a normal distribution of errors, indicating no bias between our naive bootstrapping based on the FEER algorithm, and the results from the FEERCI algorithm.

VI. Conclusion

We have presented two efficient algorithms for calculating EERs and their confidence intervals. We have benchmarked the performance of these algorithms as they are implemented in the python feerci package, against other major contemporary biometrics toolkits in Python, of which there is currently only one: bob and found significant speed-ups for both the FEER

and FEERCI algorithm. We introduced the feerci package, an opinionated, open source Python package that gives an implementation of both previously mentioned algorithms. We believe the combination of speed, accuracy and ease-of-use of the FEERCI algorithm as implemented in the feerci package takes away many barriers holding back wide-spread adoption of CIs across the field of biometrics, and hope to see them employed more often.

REFERENCES

[1] AHRENS, J. H., AND DIETER, U. Computer methods for sampling from gamma, beta, poisson and bionomial distributions. *Computing 12*, 3 (1974), 223–246.

[2] ANJOS, A., EL-SHAFEY, L., WALLACE, R., GÜNTHER, M., MCCOOL, C., AND MARCEL, S. Bob: a free signal processing and machine learning toolbox for researchers. In *Proceedings of the 20th ACM international conference on Multimedia* (2012), ACM, pp. 1449–1452.

[3] BERTAIL, P., CLÉMENÇCON, S. J., AND VAYATIS, N. On bootstrapping the roc curve. In *Advances in Neural Information Processing Systems* (2009), pp. 137–144.

[4] DEHLING, H. G., AND KALMA, J. N. *Kansrekening: het zekere voor het onzekere*. Epsilon Uitgaven, 2005. p. 186:189.

[5] FALK, M., AND REISS, R.-D. Weak convergence of smoothed and nonsmoothed bootstrap quantile estimates. *The Annals of Probability* (1989), 362–371.

[6] SCHUCKERS, M. E., MINEV, Y., AND ADLER, A. Curvewise det confidence regions and pointwise eer confidence intervals using radial sweep methodology. In *International Conference on Biometrics* (2007), Springer, pp. 376–385.

[7] VAN DER WALT, S., COLBERT, S. C., AND VAROQUAUX, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.