calzone

Release develop

Kwok Lung "Jason" Fan, Qian Cao

Oct 25, 2024

CONTENTS:

calzone is a Python package for calculation of various calibration metrics. This work is credited to Kwok Lung (Jason) Fan and Qian Cao.

The calzone package provides a suite of tools for assessing and improving the calibration of machine learning models, particularly in binary classification tasks. It offers various calibration metrics and visualization tools for reliability diagrams. Please read the summary and guide first before using the package.

Key features of calzone include:

- Calculation of various calibration metrics (e.g., ECE, MCE, Hosmer-Lemeshow test, spiegelhalter z test, etc.)
- Visualization functions for reliability diagrams
- Bootstrapping capabilities for confidence interval estimation
- Subgroup analysis for calibration metrics
- Provides command line interface scripts for batch processing
- Multi-class extension by 1-vs-rest or top-class only

To accurately assess the calibration of machine learning models, it is essential to have a comprehensive and reprensative dataset with sufficient coverage of the prediction space. The calibration metrics is not meaningful if the dataset is not representative of true intended population.

We hope you find calzone useful in your machine learning projects!

CHAPTER

ONE

QUICK START

The tutorial provides a quick start to the calzone package. Including the installation and the basic command line interface useage.

1.1 Installation

calzone dependencies are numpy, scipy, statsmodels and matplotlib. If you are experienced developer, you probably already have numpy, scipy and matplolib installed. If you don't have a package install, you can install them with conda

```
[ ]: conda install numpy
     conda install scipy
     conda install matplotlib
     conda install statsmodels
```

Alternatively, you can install the dependencies with pip

```
[ ]: pip install numpy
     pip install scipy
     pip install matplotlib
     pip install statsmodels
```

Then, you can process to install the calzone package from github if you only want to use the calculator inside you python script:

```
[ ]: pip install -e "git+https://github.com/DIDSR/calzone.git"
```

If you want to run the command line interface or GUI interface, you need to clone the github repository. Notice that the GUI interface require install nicegui using `pip install nicegui`.

```
[ ]: git clone https://github.com/DIDSR/calzone.git
     cd calzone
     # install calzone
     pip install -e .
```

1.2 Command line interface

First, you need to prepare your dataset in a specified format. The dataset should be a CSV file with the following columns:

proba_0, proba_1, . . . , proba_n, label

where n >= 1.

Or if you have subgroups, the dataset should be a CSV file with the following columns:

proba_0, proba_1, . . . , proba_n, subgroup_1, subgroup_2, . . . , subgroup_m, label

where n >= 1 and m >= 1.

In the case of multi-class, you need to specify the class-of-interest, and the problem will be treated as 1-vs-all binary classification. To test the full calibration of the whole model, you need to test the calibration of each class.

The program also works if your csv file has no header.It will assume the first [:-1] columns are the probabilities and the last column is the label.

[6]:
```
### For illuration purpose , I will use the functions in the helper.py in the local␣
↪directory
### The data is generated using beta-binomial distribution


from helper import * #import local helper functions for example data generation

generate_wellcal_data(5000,"../../../example_data/simulated_welldata.csv",alpha_val=0.5,␣
↪beta_val=0.5,random_seed=123)
generate_miscal_data(5000,"../../../example_data/simulated_misdata.csv", miscal_scale=2,
↪alpha_val=0.5, beta_val=0.5,random_seed=123)
generate_subgroup_data(5000,"../../../example_data/simulated_data_subgroup.csv",miscal_
↪scale=2,alpha_val=0.5, beta_val=0.5,random_seed=123)
```

To use CLI, you can use the script in the calzone directory. The program will save the metrics into the output csv file with the CI(if you turn on boostrap). The program will also save the relibaility diagram if you apply –plot flag. There is an optional flag –prevalence_adjustment which try to derive the original model prevalence and apply prevalence adjustment. See more on prevalence adjustment in the prevalecne adjustment notebook.

[1]:
```
# Use -h to see the help message and options
%run ../../../cal_metrics.py -h
```

```
usage: cal_metrics.py [-h] [--csv_file CSV_FILE] [--metrics METRICS]
                      [--prevalence_adjustment] [--n_bootstrap N_BOOTSTRAP]
                      [--bootstrap_ci BOOTSTRAP_CI]
                      [--class_to_calculate CLASS_TO_CALCULATE]
                      [--num_bins NUM_BINS]
                      [--hl_test_validation HL_TEST_VALIDATION] [--topclass]
                      [--save_metrics SAVE_METRICS] [--plot]
                      [--plot_bins PLOT_BINS] [--save_plot SAVE_PLOT]
                      [--save_diagram_output SAVE_DIAGRAM_OUTPUT] [--verbose]

Calculate calibration metrics and visualize reliability diagram.

options:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --csv_file CSV_FILE   Path to the input CSV file. (If there is header,it
                        must be in: proba_0,proba_1,...,subgroup_1(optional),s
                        ubgroup_2(optional),...label. If no header, then the
                        columns must be in the order of
                        proba_0,proba_1,...,label)
  --metrics METRICS     Comma-separated list of specific metrics to calculate
                        (SpiegelhalterZ,ECE-H,MCE-H,HL-H,ECE-C,MCE-C,HL-
                        C,COX,Loess,all). Default: all
  --prevalence_adjustment
                        Perform prevalence adjustment (default: False)
  --n_bootstrap N_BOOTSTRAP
                        Number of bootstrap samples (default: 0)
  --bootstrap_ci BOOTSTRAP_CI
                        Bootstrap confidence interval (default: 0.95)
  --class_to_calculate CLASS_TO_CALCULATE
                        Class to calculate metrics for (default: 1)
  --num_bins NUM_BINS   Number of bins for ECE/MCE/HL calculations (default:
                        10)
  --hl_test_validation HL_TEST_VALIDATION
                        Using nbin instead of nbin-2 as HL test DOF. Use it if
                        the dataset is validation set.
  --topclass            Whether to transform the problem to top-class problem.
  --save_metrics SAVE_METRICS
                        Save the metrics to a csv file
  --plot                Plot reliability diagram (default: False)
  --plot_bins PLOT_BINS
                        Number of bins for reliability diagram
  --save_plot SAVE_PLOT
                        Save the plot to a file. Must end with valid image
                        formats.
  --save_diagram_output SAVE_DIAGRAM_OUTPUT
                        Save the reliability diagram output to a file
  --verbose             Print verbose output
```

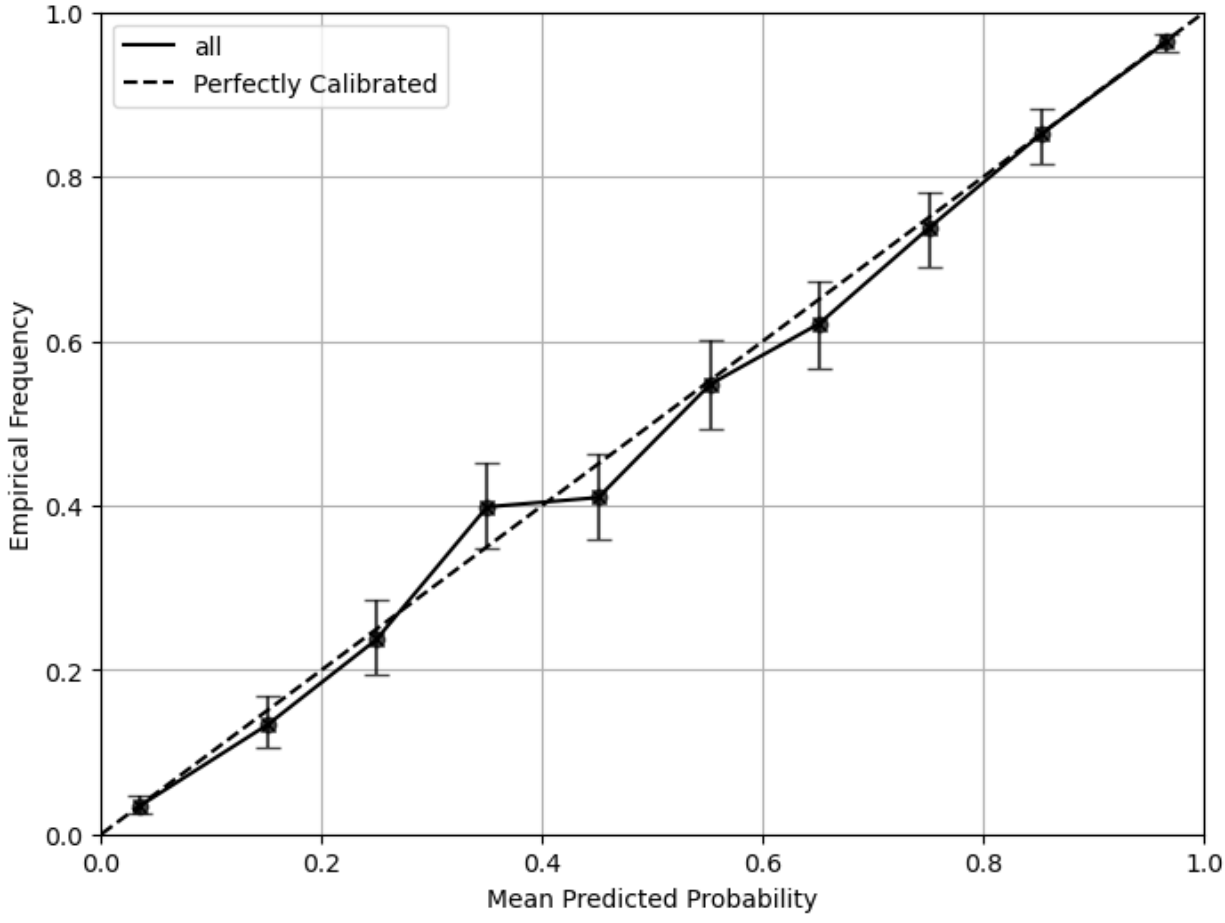
[8]:
```
%run ../../../cal_metrics.py \
--csv_file '../../../example_data/simulated_welldata.csv' \
--metrics all \
--n_bootstrap 1000 \
--bootstrap_ci 0.95 \
--class_to_calculate 1 \
--num_bins 10 \
--save_metrics '../../../example_data/simulated_welldata_result.csv' \
--plot \
--plot_bins 10 \
--save_plot '../../../example_data/simulated_welldata_result.png' \
--verbose \
--save_diagram_output '../../../example_data/simulated_welldata_diagram_output.csv'
### save_diagram_output only when you want to save the reliability diagram output
#--prevalence_adjustment # only when you want to apply prevalence adjustment
#--hl_test_validation #use it only when the data is from validation set
```

```
Metrics with bootstrap confidence intervals:
```

(continues on next page)

(continued from previous page)

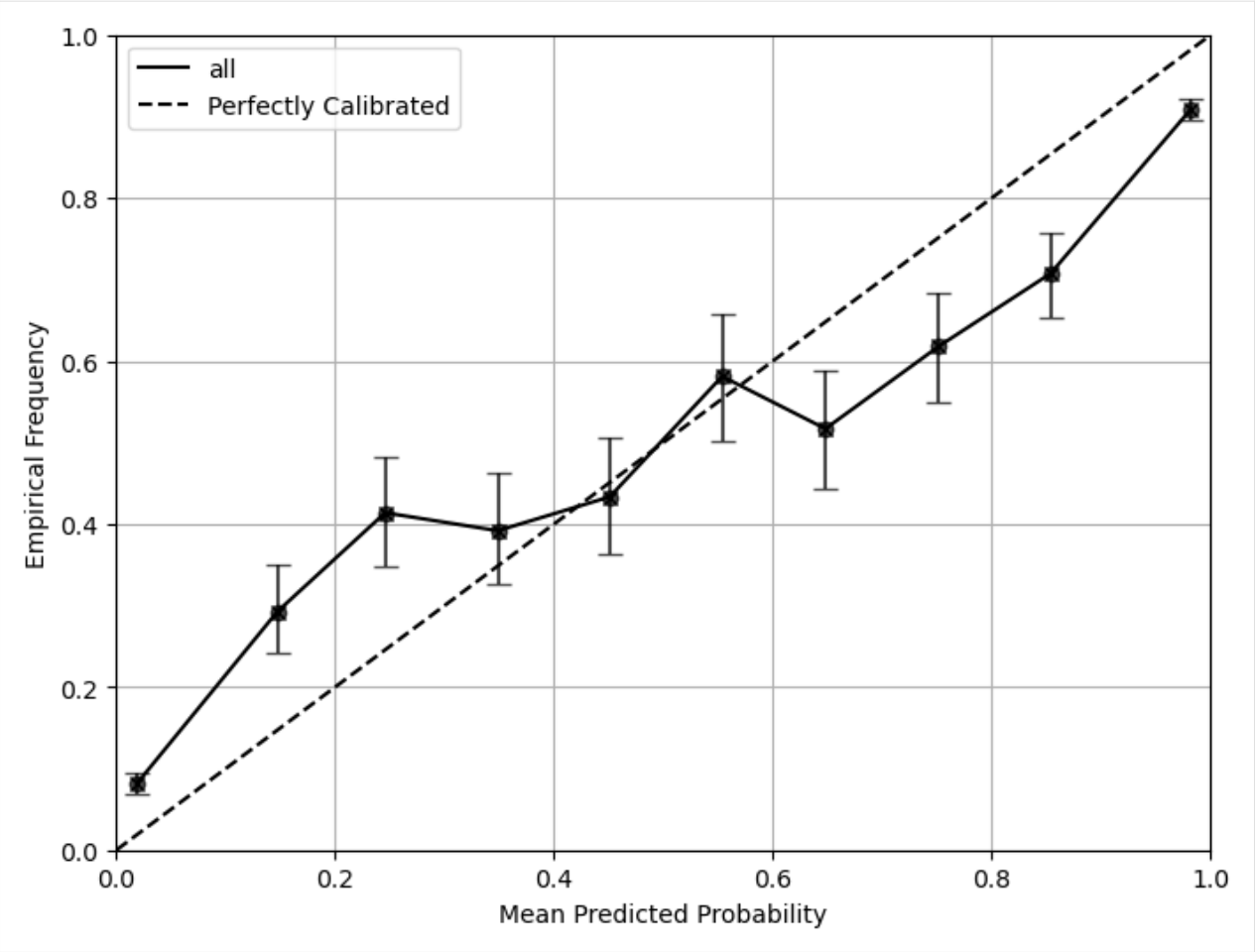
```
SpiegelhalterZ score: 0.376 (-1.581, 2.396)
SpiegelhalterZ p-value: 0.707 (0.017, 0.971)
ECE-H topclass: 0.01 (0.006, 0.021)
ECE-H: 0.012 (0.011, 0.025)
MCE-H topclass: 0.039 (0.016, 0.076)
MCE-H: 0.048 (0.034, 0.107)
HL-H score: 8.885 (7.392, 34.058)
HL-H p-value: 0.352 (0.000, 0.495)
ECE-C topclass: 0.009 (0.007, 0.021)
ECE-C: 0.009 (0.008, 0.022)
MCE-C topclass: 0.021 (0.018, 0.072)
MCE-C: 0.023 (0.020, 0.071)
HL-C score: 3.695 (4.778, 28.666)
HL-C p-value: 0.884 (0.000, 0.781)
COX coef: 0.994 (0.940, 1.054)
COX intercept: -0.045 (-0.126, 0.031)
COX coef lowerci: 0.937 (0.886, 0.994)
COX coef upperci: 1.051 (0.994, 1.114)
COX intercept lowerci: -0.123 (-0.205, -0.049)
COX intercept upperci: 0.034 (-0.048, 0.109)
COX ICI: 0.006 (0.001, 0.016)
Loess ICI: 0.006 (0.004, 0.016)
```



We can also test it on a miscalibrated dataset

[9]:
```
%run ../../../cal_metrics.py \
--csv_file '../../../example_data/simulated_misdata.csv' \
--metrics all \
--n_bootstrap 1000 \
--bootstrap_ci 0.95 \
--class_to_calculate 1 \
--num_bins 10 \
--save_metrics '../../../example_data/simulated_misdata_result.csv' \
--plot \
--plot_bins 10 \
--save_plot '../../../example_data/simulated_misdata_result.png' \
--verbose
```

```
Metrics with bootstrap confidence intervals:
SpiegelhalterZ score: 29.626 (26.37, 33.185)
SpiegelhalterZ p-value: 0. (0., 0.)
ECE-H topclass: 0.081 (0.073, 0.091)
ECE-H: 0.081 (0.073, 0.092)
MCE-H topclass: 0.151 (0.127, 0.202)
MCE-H: 0.168 (0.145, 0.244)
HL-H score: 1027.940 (818.524, 1292.977)
HL-H p-value: 0. (0., 0.)
ECE-C topclass: 0.079 (0.069, 0.09)
ECE-C: 0.08 (0.07, 0.091)
MCE-C topclass: 0.168 (0.141, 0.204)
MCE-C: 0.158 (0.139, 0.203)
HL-C score: 1857.584 (1355.637, 3341.319)
HL-C p-value: 0. (0., 0.)
COX coef: 0.497 (0.470, 0.524)
COX intercept: -0.045 (-0.125, 0.038)
COX coef lowerci: 0.469 (0.443, 0.494)
COX coef upperci: 0.526 (0.497, 0.554)
COX intercept lowerci: -0.123 (-0.203, -0.041)
COX intercept upperci: 0.034 (-0.046, 0.116)
COX ICI: 0.078 (0.071, 0.085)
Loess ICI: 0.074 (0.065, 0.083)
```



If your data has subgroup in it, simply run the script with the same argument as the one above. It will automatically detect the subgroup and generate the corresponding plots and metrics for each subgroup as well as the overall plot and metrics.

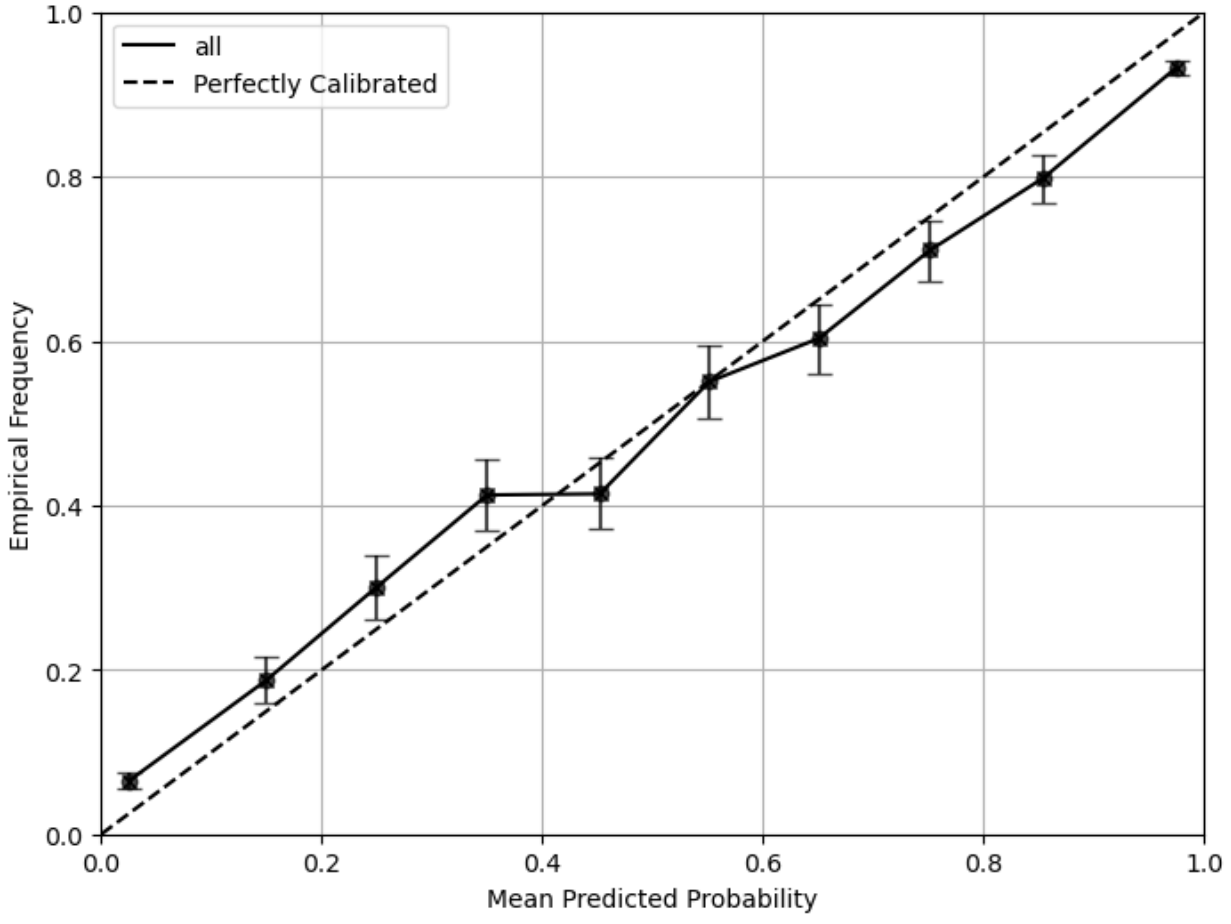
[10]:
```
%run ../../../cal_metrics.py \
--csv_file '../../../example_data/simulated_data_subgroup.csv' \
--metrics all \
--n_bootstrap 1000 \
--bootstrap_ci 0.95 \
--class_to_calculate 1 \
--num_bins 10 \
--save_metrics '../../../example_data/simulated_data_subgroup_result.csv' \
--plot \
--plot_bins 10 \
--save_plot '../../../example_data/simulated_data_subgroup_result.png' \
--verbose
```

```
Metrics with bootstrap confidence intervals:
SpiegelhalterZ score: 18.327 (15.794, 21.009)
SpiegelhalterZ p-value: 0. (0., 0.)
ECE-H topclass: 0.042 (0.035, 0.049)
ECE-H: 0.042 (0.036, 0.049)
MCE-H topclass: 0.055 (0.043, 0.087)
```

(continues on next page)

(continued from previous page)

```
MCE-H: 0.063 (0.055, 0.109)
HL-H score: 429.732 (335.116, 584.729)
HL-H p-value: 0. (0., 0.)
ECE-C topclass: 0.042 (0.035, 0.049)
ECE-C: 0.038 (0.032, 0.046)
MCE-C topclass: 0.065 (0.055, 0.091)
MCE-C: 0.064 (0.052, 0.086)
HL-C score: 1138.842 (779.577, 1844.15)
HL-C p-value: 0. (0., 0.)
COX coef: 0.668 (0.638, 0.698)
COX intercept: -0.02 (-0.073, 0.03)
COX coef lowerci: 0.641 (0.611, 0.67)
COX coef upperci: 0.696 (0.664, 0.727)
COX intercept lowerci: -0.074 (-0.127, -0.025)
COX intercept upperci: 0.034 (-0.019, 0.084)
COX ICI: 0.049 (0.044, 0.055)
Loess ICI: 0.037 (0.032, 0.043)
```

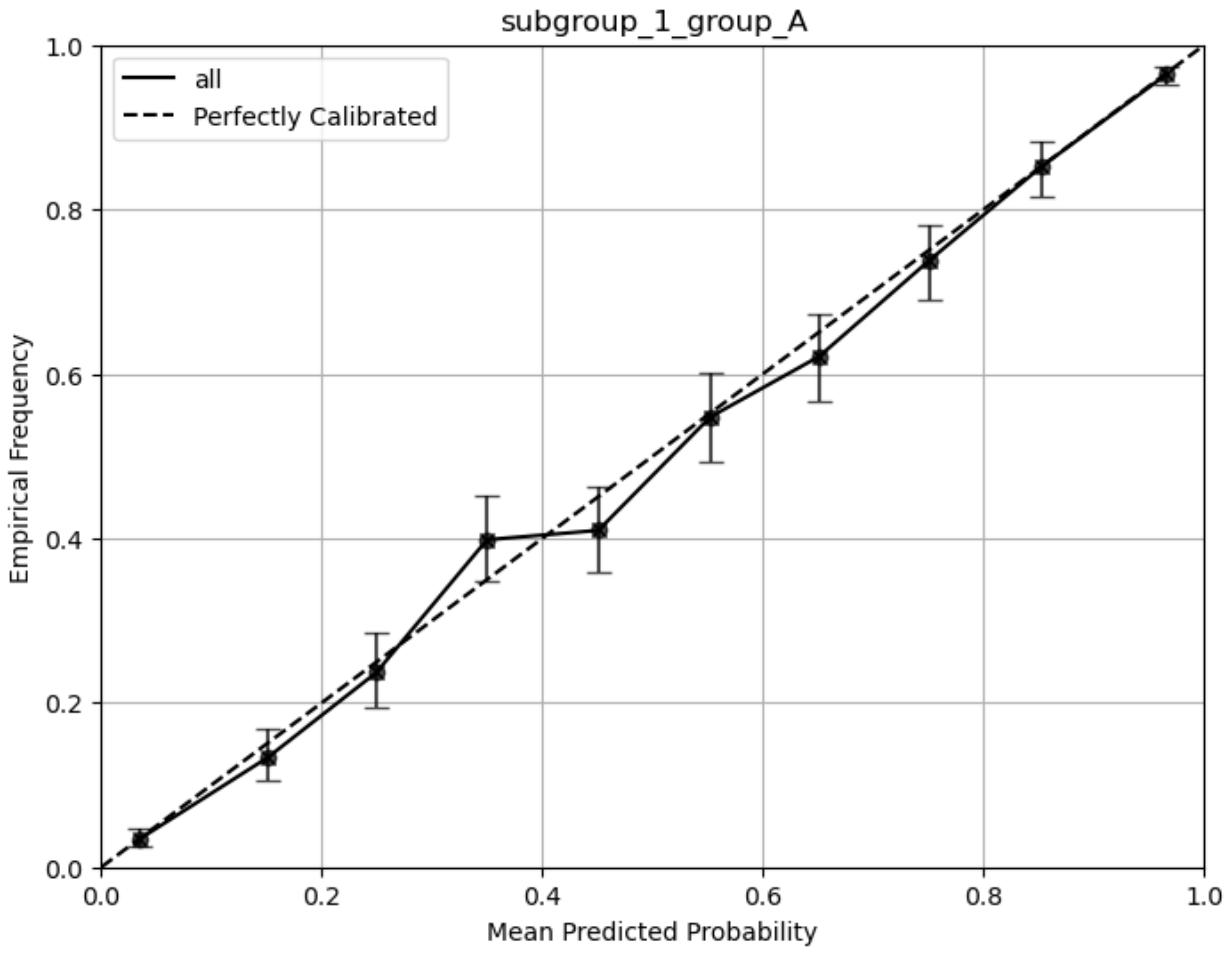



```
Metrics for subgroup_1_group_A with bootstrap confidence intervals:
SpiegelhalterZ score: 0.376 (-1.536, 2.249)
SpiegelhalterZ p-value: 0.707 (0.018, 0.981)
ECE-H topclass: 0.01 (0.006, 0.021)
```

(continues on next page)

(continued from previous page)

```
ECE-H: 0.012 (0.011, 0.025)
MCE-H topclass: 0.039 (0.017, 0.077)
MCE-H: 0.048 (0.034, 0.107)
HL-H score: 8.885 (7.273, 34.6)
HL-H p-value: 0.352 (0.000, 0.507)
ECE-C topclass: 0.009 (0.007, 0.022)
ECE-C: 0.009 (0.007, 0.023)
MCE-C topclass: 0.021 (0.018, 0.072)
MCE-C: 0.023 (0.018, 0.075)
HL-C score: 3.695 (4.463, 29.686)
HL-C p-value: 0.884 (0.000, 0.813)
COX coef: 0.994 (0.942, 1.050)
COX intercept: -0.045 (-0.135, 0.028)
COX coef lowerci: 0.937 (0.888, 0.990)
COX coef upperci: 1.051 (0.996, 1.111)
COX intercept lowerci: -0.123 (-0.214, -0.051)
COX intercept upperci: 0.034 (-0.056, 0.107)
COX ICI: 0.006 (0.001, 0.017)
Loess ICI: 0.006 (0.003, 0.016)
```

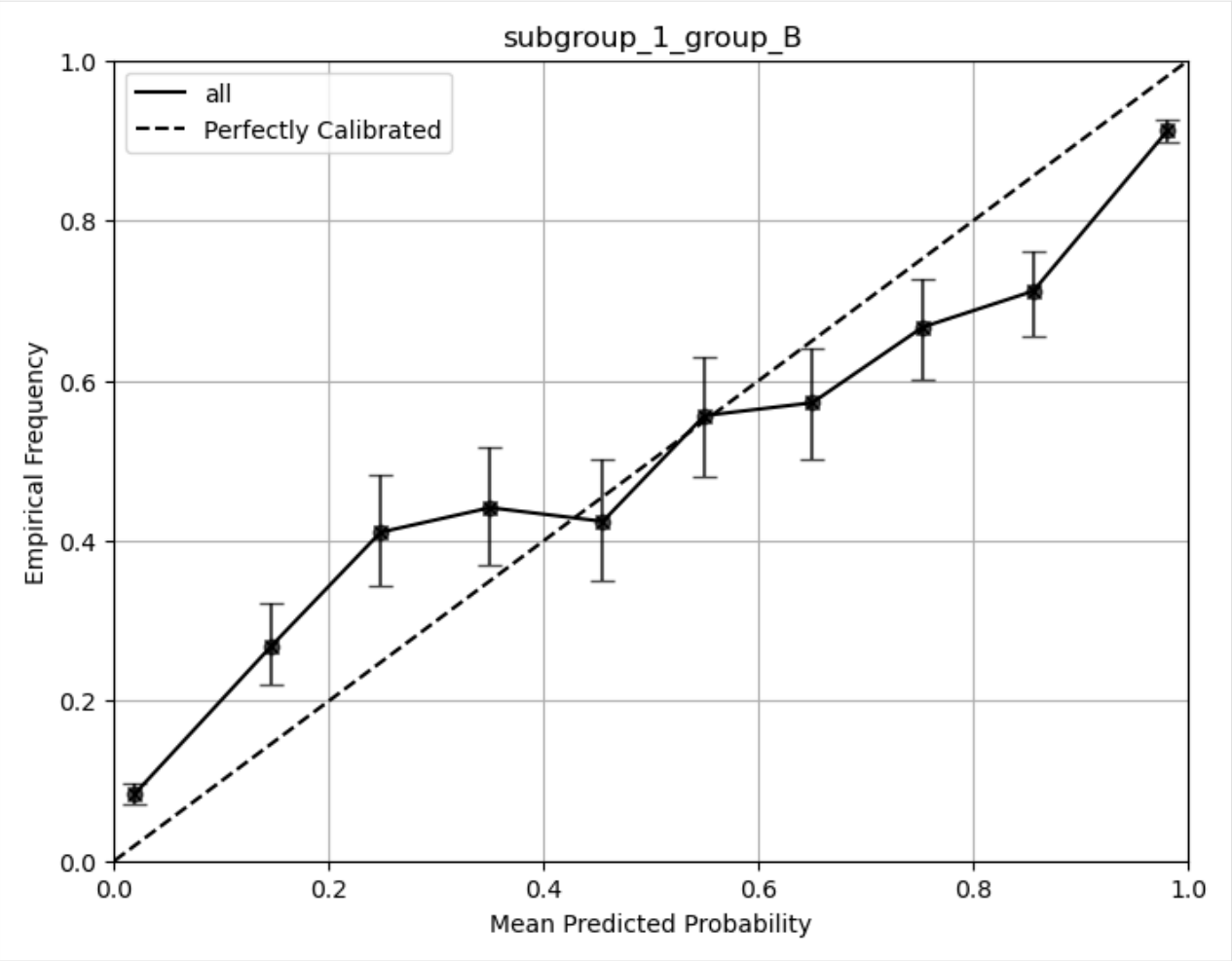



```
Metrics for subgroup_1_group_B with bootstrap confidence intervals:
```

(continues on next page)

(continued from previous page)

```
SpiegelhalterZ score: 27.936 (24.631, 31.175)
SpiegelhalterZ p-value: 0. (0., 0.)
ECE-H topclass: 0.077 (0.068, 0.086)
ECE-H: 0.077 (0.069, 0.087)
MCE-H topclass: 0.133 (0.108, 0.175)
MCE-H: 0.163 (0.13, 0.232)
HL-H score: 910.439 (716.681, 1156.971)
HL-H p-value: 0. (0., 0.)
ECE-C topclass: 0.074 (0.066, 0.084)
ECE-C: 0.075 (0.065, 0.085)
MCE-C topclass: 0.141 (0.124, 0.182)
MCE-C: 0.140 (0.115, 0.182)
HL-C score: 2246.171 (1393.175, 3900.391)
HL-C p-value: 0. (0., 0.)
COX coef: 0.507 (0.481, 0.539)
COX intercept: 0.000 (-0.073, 0.077)
COX coef lowerci: 0.478 (0.454, 0.508)
COX coef upperci: 0.536 (0.508, 0.569)
COX intercept lowerci: -0.078 (-0.152, -0.002)
COX intercept upperci: 0.079 (0.005, 0.155)
COX ICI: 0.077 (0.07, 0.085)
Loess ICI: 0.07 (0.062, 0.078)
```



1.3 Using calzone in python

Instead of running the command line tool, you can also use calzone in python directly

[1]:
```
from calzone.metrics import CalibrationMetrics
from calzone.utils import data_loader

loader = data_loader('../../../example_data/simulated_welldata.csv')
cal_metrics = CalibrationMetrics(class_to_calculate=1)
cal_metrics.calculate_metrics(loader.labels, loader.probs, metrics='all')
```

[1]:
```
{'SpiegelhalterZ score': 0.3763269161877356,
 'SpiegelhalterZ p-value': 0.7066738713391099,
 'ECE-H topclass': 0.009608653731328977,
 'ECE-H': 0.01208775955804901,
 'MCE-H topclass': 0.03926468843081976,
 'MCE-H': 0.04848338618970194,
 'HL-H score': 8.884991559088098,
 'HL-H p-value': 0.35209071874348785,
 'ECE-C topclass': 0.009458033653818828,
```

(continues on next page)

(continued from previous page)

```
'ECE-C': 0.008733966945443138,
'MCE-C topclass': 0.020515047600205505,
'MCE-C': 0.0232403122348625 6,
'HL-C score': 3.694947603203135,
'HL-C p-value': 0.8835446575708198,
'COX coef': 0.9942499557748269,
'COX intercept': -0.04497652296600376,
'COX coef lowerci': 0.9372902801721911,
'COX coef upperci': 1.0512096313774626,
'COX intercept lowerci': -0.12348577118577644,
'COX intercept upperci': 0.03353272525376893,
'COX ICI': 0.005610391483826338,
'Loess ICI': 0.00558856942568957}
```

CHAPTER

TWO

SUMMARY AND GUIDE FOR CALZONE

We provide a summary of the calibration metrics provides by calzone, including the pros and cons of each metrics. For a more detailed explanation of each metrics and how to calculate them using calzone, please refer to the specific notebook.

Metrics	Description	Pros	Cons	Meaning
ECE	Using binned reliability diagram (equal-width or equal-count binning), sum of absolute difference, weighted by bin count.	• Intuitive • Easy to calculate	• Depend on binning • Depend on class-by-class/top-class	Average deviation from true probability
MCE	Using binned reliability diagram (equal-width or equal-count binning), Maximum absolute difference.	• Intuitive • Easy to calculate	• Depend on binning • Depend on class-by-class/top-class	Maximum deviation from true probability
Hosmer-Lemeshow test	Using binned reliability diagram (equal-width or equal-count binning), Chi-squared based test using expected and observed.	• Intuitive • Statistical meaning	• Depend on binning • Low power • Wrong coverage	Test of calibration
Spiegelhalter's z test	Decomposition of brier score. Normal distributed	• Doesn't rely on binning • Statistical meaning	• Doesn't detect prevalence shift	Test of calibration
Cox's analysis	Logistic regression of the logits	• Doesn't rely on binning • Hints at miscalibration type	• Failed to capture some miscalibration	A logit fit to the calibration curve
Integrated calibration index (ICI)	Similar to ECE, using smooth fit (usually losse) instead of binning to get the calibration curve	• Doesn't rely on binning • Capture all kind of miscalibration	• Depend on the choice of curve fitting • Depend on fitting parameters	Average deviation from true probability

2.1 Guide to calzone and calibration metrics

calzone aims to access whether a model achieves moderate calibration, meaning whether $\mathbb{P}(\hat{Y} = Y|\hat{P} = p) = p$ for all $p \in [0, 1]$.

To accurately assess the calibration of machine learning models, it is essential to have a comprehensive and reprensative dataset with sufficient coverage of the prediction space. The calibration metrics is not meaningful if the dataset is not representative of true intended population.

calzone takes in a csv dataset which contains the probability of each class and the true label. Most metrics in calzone only work with binary classification and which transforms the problem into 1-vs-rest when calcualte the metrics. Therefore, you need to specify the class-of-interest when using the metrics. The only exception is the ECE_{top} and MCE_{top} metrics which works for multi-class problems. See the corresponding documentation for more details.

We recommend visualizing calibration using reliability diagrams. If you observe general over- or under-estimation of probabilities for a given class, consider applying a prevalence adjustment to determine if it's solely due to prevalence shift. After prevalence adjustment, plot the reliability diagrams again and examine the results of calibration metrics.

For a general sense of average probability deviation, we recommend using the Cox and Loess integrated calibration index (ICI). To test for calibration, we suggest using Spiegelhalter's z-test. Other metrics such as Expected Calibration Error (ECE), Cox slope/intercept, and Hosmer-Lemeshow (HL) test depends strongly on binning and should be used with caution.

Please refer to the notebooks for detailed descriptions of each metric.

CHAPTER

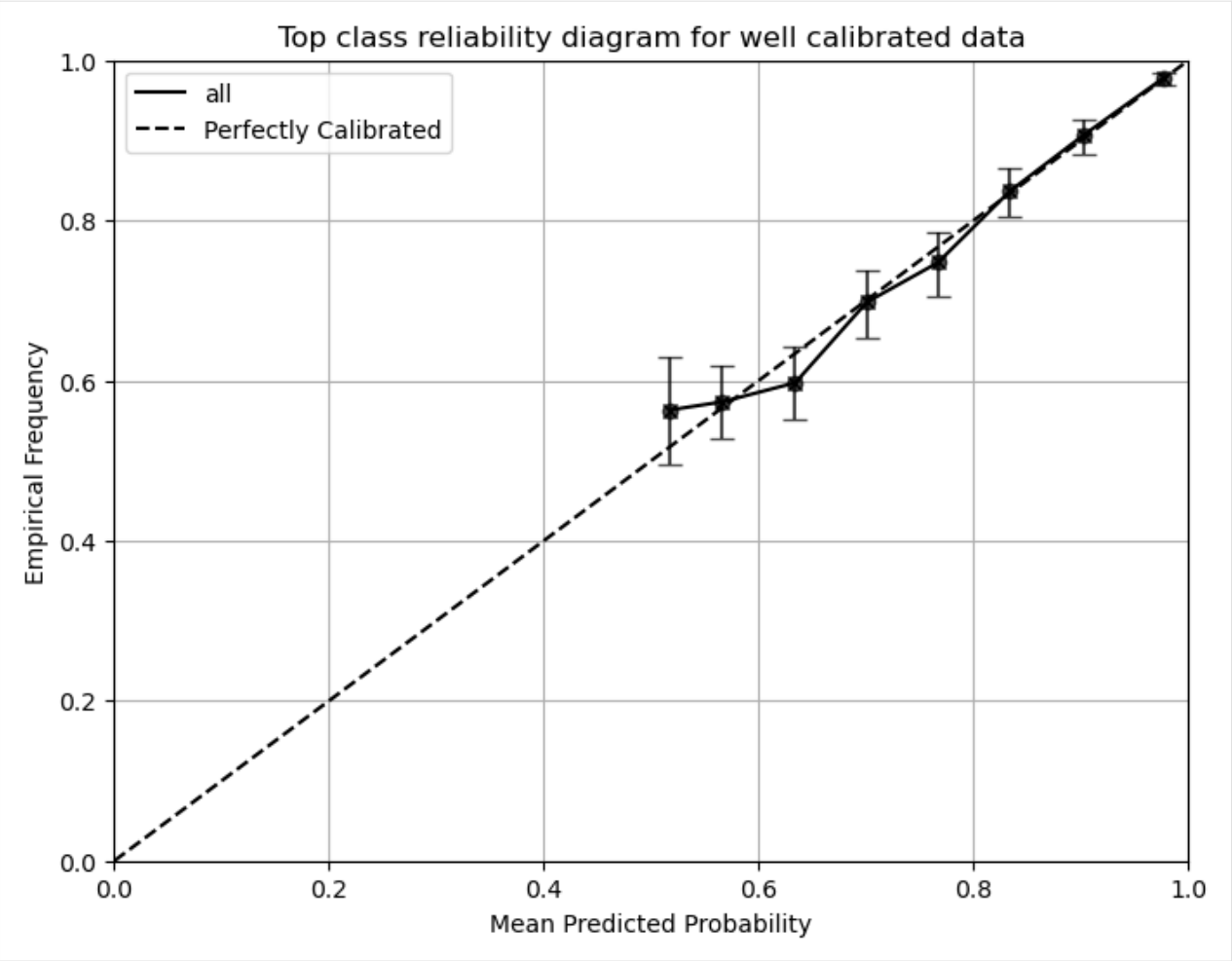
THREE

RELIABILITY DIAGRAM

Reliability Diagram is a tool to visualize the calibration of a model given a set of data. It groups the data into bins and plots the accuracy of each bin against the average predicted value for that bin. The reliability diagram can be plotted for top-class prediction only or for a given class. The calzone package provides a function to calculate and plot the reliability diagram.

[1]:
```
### Import the necessary libraries and load the data
import numpy as np
from calzone.utils import reliability_diagram,data_loader
from calzone.vis import plot_reliability_diagram
### loading the data
wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")
```

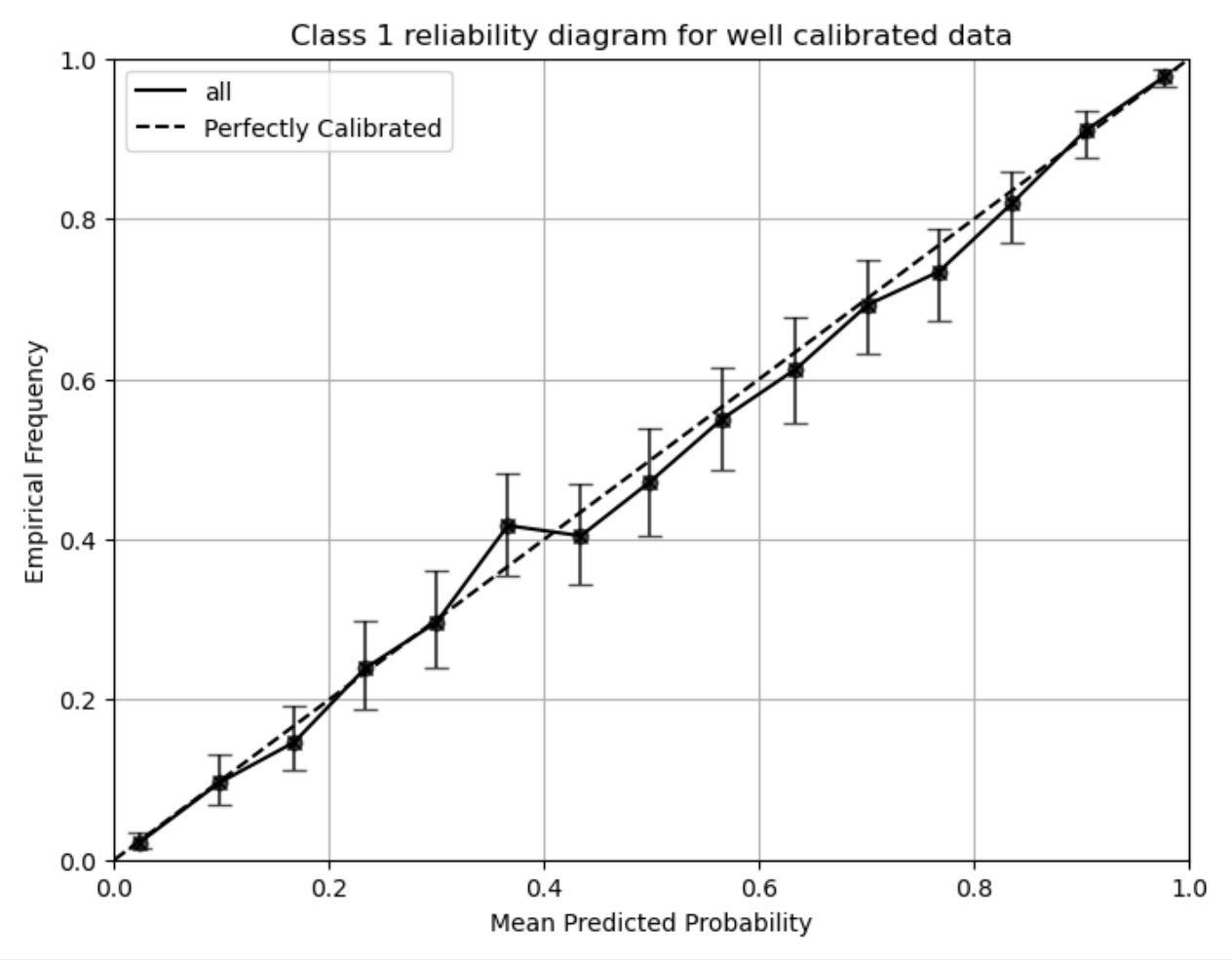
[13]:
```
### Create and plot the top-class well calibrated data
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels, wellcal_dataloader.probs, num_bins=15,class_to_plot=None) #clasto plot is None␣
↪mean calcuate for top-class
# Plot the reliability diagram
plot_reliability_diagram(reliability,confindence,bin_counts,error_bar=True,title='Top␣
↪class reliability diagram for well calibrated data')
```



The error bar in the reliability diagram is the 95% confidence interval calculated using wilson score interval which assume that samples in a bin is a series of Bernoulli trials with the success probability equal to the mean predicted probability. The confidence interval is only for reference and might not be exact.

Since we have a binary classification problem, The mean predicted probability will not go below 0.5 for top-class reliability diagram. We will proceed to plot the class 1 reliability diagram.

[14]:
```
### Create and plot the class 1 well calibrated data
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels, wellcal_dataloader.probs, num_bins=15,class_to_plot=1)
# Plot the reliability diagram
plot_reliability_diagram(reliability,confindence,bin_counts,error_bar=True,title='Class␣
↪1 reliability diagram for well calibrated data')
```



Class-by-class reliability diagram reveal more information about the model's calibbration. The top-class reliability diagram could be misleading as it could shows reasonable calibration for the top-class, but the model could be over-confident for the other classes. We can demonstrate in the following example.

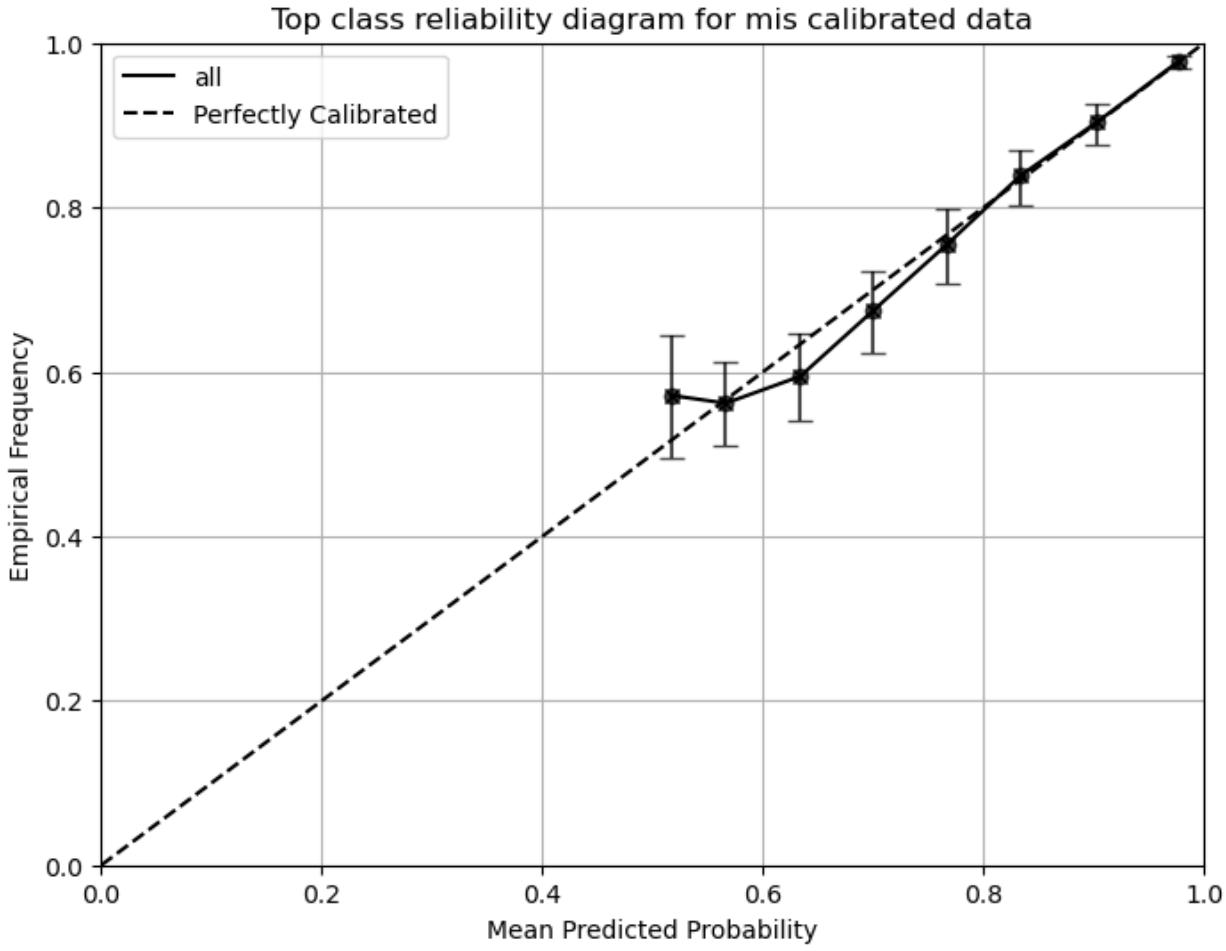
[40]:
```
### We will artificially drop the prevalence of class 1
# The top-class reliability diagram will still look good
# But the class-1 reliability diagram will be very bad
from calzone.utils import softmax_to_logits
from scipy.special import softmax
import numpy as np

test_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv")
class_1_index = (test_dataloader.labels==1)

# We will drop 50% of class 1 samples
class_1_samples = np.where(class_1_index)[0]
drop_indices = np.random.choice(class_1_samples, size=int(len(class_1_samples)/2),␣
↪replace=False)

mask = np.ones(len(test_dataloader.labels), dtype=bool)
mask[drop_indices] = False
```

(continues on next page)

(continued from previous page)

```
test_dataloader.labels = test_dataloader.labels[mask]
test_dataloader.probs = test_dataloader.probs[mask]
test_dataloader.data = test_dataloader.data[mask]
```

[42]:
```
### Create and plot the top-class reliability diagram
reliability,confindence,bin_edges,bin_counts = reliability_diagram(test_dataloader.
↪labels, test_dataloader.probs, num_bins=15,class_to_plot=None)
plot_reliability_diagram(reliability,confindence,bin_counts,error_bar=True,title='Top␣
↪class reliability diagram for mis calibrated data')
```

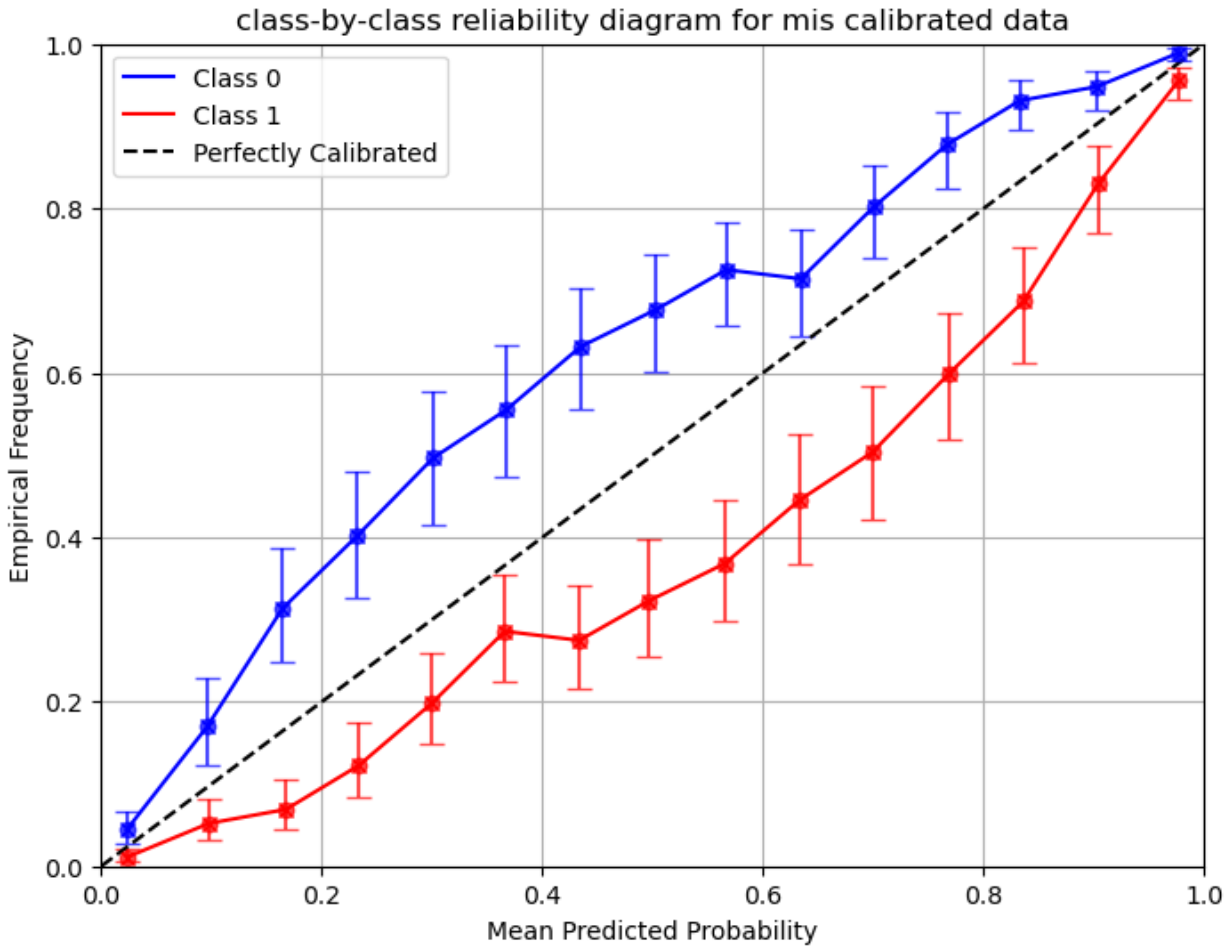


[43]:
```
### Create and plot the class-by-class reliability diagram
reliability_0,confindence_0,bin_edges,bin_counts_0 = reliability_diagram(test_dataloader.
↪labels, test_dataloader.probs, num_bins=15,class_to_plot=0)
reliability_1,confindence_1,bin_edges,bin_counts_1 = reliability_diagram(test_dataloader.
↪labels, test_dataloader.probs, num_bins=15,class_to_plot=1)
reliability = np.vstack((reliability_0,reliability_1))
confindence = np.vstack((confindence_0,confindence_1))
```

(continues on next page)

(continued from previous page)

```
bin_counts = np.vstack((bin_counts_0,bin_counts_1))
plot_reliability_diagram(reliability,confindence,bin_counts,error_bar=True,title='class-
↪by-class reliability diagram for mis calibrated data',custom_colors=['blue','red'])
```



The example shows us that the prevalence shift in the testing data could lead to mis-calibration in a posterior sense. The calzone provide a method to calibrate it. The prevalence is not related to the model and the model could still have the correct likelihood ratio. See more discussion in the prevalence adjustment section.

3.1 References

Bröcker, J., & Smith, L. A. (2007). Increasing the Reliability of Reliability Diagrams. Weather and Forecasting, 22(3), 651–661. https://doi.org/10.1175/WAF993.1

CHAPTER

FOUR

EXEPECTED CALIBRATION ERROR(ECE) AND MAXIMUM CALIBRATION ERROR (MCE)

4.1 Theoretical Background

(True) Expected Calibration Error is defined as the average difference between the predicted probability and the true probability for a particular class. We will label the predicted probability as $\hat{P}$ and the true probability as P and drop the class of interest label for simplicity.

$$\text{ECE} = \mathbb{E}_{\hat{P}}[|\mathbb{P}(\hat{Y} = Y|\hat{P} = p) - p|] = \int_0^1 |\mathbb{P}(\hat{Y} = Y|\hat{P} = p) - p| \, dF_{\hat{P}}(p)$$

Some also define the ECE as the average difference between the predicted probability and the true probability of the predicted class. This is similar to the top-class vs class-by-class reliability diagram. We will refer to it as ECE_{top}.

$$\text{ECE}_{top} = \mathbb{E}_{\hat{P_{top}}}[|\mathbb{P}(\hat{Y_{top}} = Y_{top}|\hat{P_{top}} = p_{top}) - p_{top}|])$$

Similarly, we can also define the Maximum Calibration Error (MCE) as the maximum difference between the predicted and true probabilities:

$$\text{MCE} = \max_p |\mathbb{P}(\hat{Y} = Y|\hat{P} = p) - p|$$

$$\text{MCE}_{top} = \max_{p_{top}} |\mathbb{P}(\hat{Y_{top}} = Y_{top}|\hat{P} = p_{top}) - p_{top}|$$

4.1.1 Estimated ECE and MCE

We can't compute ECE and MCE directly from the data. Alternatively, we can group the data into bins and compute an Estimated ECE and MCE from the grouped data. In most literature, the estimated ECE and MCE are simply referred to as ECE and MCE. We adopt the same convention but want to remind the reader that the estimated ECE and MCE are not the same as the true ECE and MCE and they heavily depend on the binning method. In the text below, we will use the terms ECE and MCE to refer to the estimated ECE and MCE, and true ECE and MCE to refer to the true ECE and MCE.

$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} |acc(B_m) - conf(B_m)|$$

where M is the number of bins, B_m is the m-th bin, $acc(B_m)$ is the accuracy of the m-th bin, $conf(B_m)$ is the mean predicted probability of the m-th bin, and n is the total number of samples. The ECE-topclass is a simple extension where only the top-class probability is used to group the data. We can define MCE similarly.

$$\text{MCE} = \max_m |acc(B_m) - conf(B_m)|$$

Both ECE and MCE group the data based on the predicted probability. This means the resulting ECE and MCE depend on the binning method used. Traditionally, the data are grouped into 10 equal-width bins. However, if the data is not evenly distributed, the resulting ECE and MCE may not be a good representation of the model's calibration.

Nixon et al. (2019) proposed a new binning method which bins the data into equal-count bins. They refer to the resulting ECE as Adaptive Calibration Error (ACE). The ACE also accounts for all the predictions for all classes. This is equivalent to a sum of ECE for all classes.

We will adopt a modified version of the ACE and aim to measure the true ECE with equal-count binning but ignore the part about summing over all classes. This is equivalent to the ECE but only with equal-count binning. We will refer to the equal-width binning ECE as ECE-H and the equal-count binning ECE as ECE-C. Similarly, we will refer to the equal-width binning MCE as MCE-H and the equal-count binning MCE as MCE-C.

4.2 Pros of ECE and MCE

ECE and MCE are perhaps the most intuitive metrics for measuring the calibration of a probabilistic model. They are simply the average deviation of the predicted probability from the true probability and the maximum deviation of the predicted probability from the true probability. The ECE and MCE are also easy to compute and interpret. ECE could be computed by doing a weighted average of the reliability diagram based on bin counts and MCE is just the maximum difference between the accuracy and the confidence. Because of that, ECE and MCE are widely used in the machine learning literature.

4.3 Cons of ECE and MCE

The biggest disadvantage of using ECE and MCE is that they rely on the binning scheme, and results will depend on the binning scheme and the number of bins. It can be shown that the average ECE and MCE will always increase with the number of bins and the average ECE and MCE will change with the number of samples for a fixed number of bins. This causes problems when interpreting the results. We will do a simple experiment in the following section to show this. Because of the above reasons, we recommend using ECE and MCE with other metrics that are not dependent on the binning scheme.

4.4 Calculating ECE and MCE with calzone

There is two way to calculate the ECE and MCE in calzone. The first way is by calling the function explicitly. Notice ECE_H is the equal-width binning and ECE_C is the equal-count binning.

[1]:
```
from calzone.utils import reliability_diagram,data_loader
from calzone.metrics import calculate_ece_mce
import numpy as np

### loading the data
wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")

### calculating the top-class ECE-H
### This is done by setting the class_to_plot=None
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=None, is_equal_freq=False)
ece_h_top_class,mce_h_top_class = calculate_ece_mce(reliability,confindence,bin_
```

(continues on next page)

(continued from previous page)

```
↪counts=bin_counts)
print("Top-class ECE-H: ",ece_h_top_class)
print("Top-class MCE-H: ",mce_h_top_class)
```

```
Top-class ECE-H:  0.009608653731328977
Top-class MCE-H:  0.03926468843081976
```

[2]:
```
### We can calculate the class 1 ECE and MCE by setting class_to_plot=1
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=1, is_equal_freq=False)
ece_h_classone,mce_h_classone = calculate_ece_mce(reliability,confindence,bin_counts=bin_
↪counts)
print("Class 1 ECE-H: ",ece_h_classone)
print("Class 1 MCE-H: ",mce_h_classone)
```

```
Class 1 ECE-H:  0.01208775955804901
Class 1 MCE-H:  0.04848338618970194
```

[3]:
```
### Similarily we can calculate the class 1 ECE-C and top-class ECE-C by setting is_
↪equal_freq=True
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=None, is_equal_freq=True)
ece_c_top_class,mce_c_top_class = calculate_ece_mce(reliability,confindence,bin_
↪counts=bin_counts)
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=1, is_equal_freq=True)
ece_c_classone,mce_c_classone = calculate_ece_mce(reliability,confindence,bin_counts=bin_
↪counts)
print("Top-class ECE-C: ",ece_c_top_class)
print("Top-class MCE-C: ",mce_c_top_class)
print("Class 1 ECE-C: ",ece_c_classone)
print("Class 1 MCE-C: ",mce_c_classone)
```

```
Top-class ECE-C:  0.009458033653818828
Top-class MCE-C:  0.020515047600205505
Class 1 ECE-C:  0.008733966945443138
Class 1 MCE-C:  0.02324031223486256
```

The second method is much simpler. We can use the calzone.metrics.CalibrationMetrics class to calculate all type of metrics.

[4]:
```
from calzone.metrics import CalibrationMetrics
calmetrics = CalibrationMetrics()
calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_dataloader.probs,␣
↪metrics=['ECE-H', 'MCE-H', 'ECE-C', 'MCE-C'])
```

[4]:
```
{'ECE-H topclass': 0.009608653731328977,
 'ECE-H': 0.01208775955804901,
 'MCE-H topclass': 0.03926468843081976,
 'MCE-H': 0.04848338618970194,
 'ECE-C topclass': 0.009458033653818828,
 'ECE-C': 0.008733966945443138,
 'MCE-C topclass': 0.020515047600205505,
 'MCE-C': 0.02324031223486256}
```

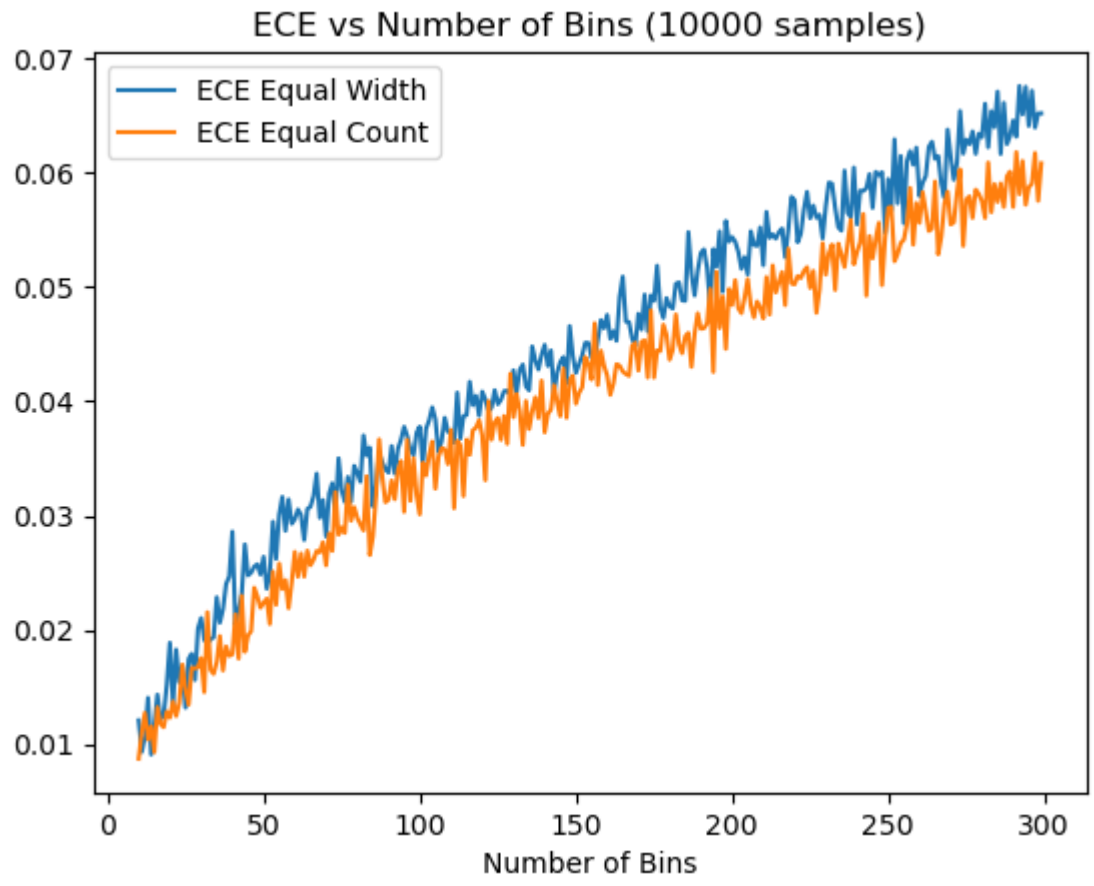
4.5 ECE and MCE as function of bin size

In this section, we want to quickly demonstrate how binning could affect the ECE and MCE.

[5]:
```
range_of_binning = np.arange(10,300,1)
result = np.zeros((len(range_of_binning),8))
for i in range(len(range_of_binning)):
    calmetrics = CalibrationMetrics(class_to_calculate=1,num_bins=range_of_binning[i])
    result[i,:] = calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_
↪dataloader.probs, metrics=['ECE-H', 'MCE-H', 'ECE-C', 'MCE-C'],return_numpy=True)
```

[6]:
```
import matplotlib.pyplot as plt
plt.plot(range_of_binning,result[:,1], label='ECE Equal Width')
plt.plot(range_of_binning,result[:,5], label='ECE Equal Count')
plt.title('ECE vs Number of Bins (10000 samples)')
#plt.ylim(0, np.max(ece_equal_width)+0.1)
plt.xlabel('Number of Bins')
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x7fe048e8b470>
```



We can see that the error goes up as the number of bins increases while equal count and equal width always return different results. Opposite effects can be observed for the number of sample. Therefore, ECE and MCE can only give us a rough estimate of the calibation error of the model.

4.6 Reference

Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). On Calibration of Modern Neural Networks (No. arXiv:1706.04599). arXiv. http://arxiv.org/abs/1706.04599

Pakdaman Naeini, M., Cooper, G., & Hauskrecht, M. (2015). Obtaining Well Calibrated Probabilities Using Bayesian Binning. Proceedings of the AAAI Conference on Artificial Intelligence, 29(1). https://doi.org/10.1609/aaai.v29i1.9602

Arrieta-Ibarra, I., Gujral, P., Tannen, J., Tygert, M., & Xu, C. (2022). Metrics of Calibration for Probabilistic Predictions.

Nixon, J., Dusenberry, M. W., Zhang, L., Jerfel, G., & Tran, D. (2020). Measuring Calibration in Deep Learning.

CHAPTER

FIVE

HOSMER-LEMESHOW TEST (HL TEST)

5.1 Theoretical Background

The Hosmer-Lemeshow test (HL Test) is a statistical test that can be used to assess the calibration of a probabilistic model. The test works by dividing the predicted probabilities into groups (typically deciles) and comparing the observed and expected frequencies of events in each group. A non-significant p-value (usually greater than 0.05) indicates we cannot reject the hypothesis that the model is well-calibrated, while a significant p-value suggests the opposite. The Hosmer-Lemeshow test is widely used in the literature and industry since it is simple to implement and interpret.

In order to calculate the Hosmer-Lemeshow test statistic, we need to first determine the binning scheme used to divide the predicted probabilities into groups. Conventionally, the predicted probabilities are divided into 10 equal-width bins. We will label the equal-width binning Hosmer-Lemeshow test as HL-H and equal-count binning Hosmer-Lemeshow test as HL-C. The Hosmer-Lemeshow test statistic is then calculated as follows:

$$\text{HL} = \sum_{m=1}^{M} \left[\frac{(O_{1,m} - E_{1,m})^2}{E_{1,m}} + \frac{(O_{0,m} - E_{0,m})^2}{E_{0,m}} \right] = \sum_{m=1}^{M} \frac{(O_{1,m} - E_{1,m})^2}{E_{1,m}(1 - \frac{E_{1,m}}{N_m})} \sim \chi^2_{M-2}$$

where $E_{1,m}$ is the expected number of class 1 events in the mth bin, $O_{1,m}$ is the observed number of class 1 events in the mth bin, N_m is the total number of observations in the mth bin, and M is the number of bins. The HL test statistic is distributed as a chi-squared distribution with $M - 2$ degrees of freedom. We can then use this test statistic to calculate the p-value for the test and determine whether we can reject the null hypothesis that the model is well-calibrated. Notice that the degree of freedom of HL test is $M - 2$ by default but sthe degree of freedom should be M instead when the samples is not used for training. We provides the option to specify the degree of freedom in the `calzone`. The default value is still $M - 2$.

5.2 Pros of HL test

The Hosmer-Lemeshow test offers several advantages in assessing calibration. It is a non-parametric test, meaning it does not require any assumptions about the distribution of the predicted probabilities, and it provides statistical meaning to the resulting test statistic. It is also very intuitive and easy to understand since it is just a chi-square based test. It can be calculated from the result of the reliability diagram. The HL test is widely used in the literature as a validation method for model calibration.

5.3 Cons of HL Test

Many studies have shown that the HL test is not an ideal way to examine the calibration of a model. The biggest problem is that the HL test depends on the binning scheme used. Whether equal-width or equal-count binning and the number of bins used can affect the results of the HL test. It is shown that the standard 10 equal-width bins often have the wrong size and low statistical power. Therefore, it is recommended not to use the HL test to examine the calibration of a model. However, the HL test is still a useful tool to quickly check the calibration of a model and provide a reference for the calibration of a model.

5.4 Calculating HL test statistics and p-value with calzone

There are again two ways to calculate the HL test statistics and p-value with calzone. One is to call the function explicitly, and the other is to use the calzone.metrics.CalibrationMetrics class.

[1]:
```
from calzone.utils import reliability_diagram,data_loader
from calzone.metrics import hosmer_lemeshow_test
import numpy as np

### loading the data
wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")

### calculating the HL-H TS
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=1, is_equal_freq=False)

#df = 10 instead 8 since it is validation data
HL_H_ts,HL_H_p,df = hosmer_lemeshow_test(reliability,confindence,bin_count=bin_counts,␣
↪df=10)
print("HL-H Test Statistic: ",HL_H_ts)
print("HL-H p-value: ",HL_H_p)
```

```
HL-H Test Statistic:  8.884991559088098
HL-H p-value:  0.5430520576015005
```

[2]:
```
### similar for HL-C model
reliability,confindence,bin_edges,bin_counts = reliability_diagram(wellcal_dataloader.
↪labels,wellcal_dataloader.probs,num_bins=10, class_to_plot=1, is_equal_freq=True)

HL_C_ts,HL_C_p,df = hosmer_lemeshow_test(reliability,confindence,bin_count=bin_counts,␣
↪df=10)
print("HL-C Test Statistic: ",HL_C_ts)
print("HL-C p-value: ",HL_C_p)
```

```
HL-C Test Statistic:  3.694947603203135
HL-C p-value:  0.9600610057855211
```

We can see the result from equal-width binning and equal-count binning are different. We will demostrate how to use the calzone.metrics.CalibrationMetrics class.

[3]:
```
### using the CalibrationMetrics class
from calzone.metrics import CalibrationMetrics
```

(continues on next page)

(continued from previous page)

```
calmetrics = CalibrationMetrics(class_to_calculate=1)
calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_dataloader.probs,␣
↪metrics=['HL-H', 'HL-C'],df=10)
```

[3]:
```
{'HL-H score': 8.884991559088098,
 'HL-H p-value': 0.5430520576015005,
 'HL-C score': 3.694947603203135,
 'HL-C p-value': 0.9600610057855211}
```

5.5 Size of HL test

We will show the size of HL test. Notice that the size of HL test had been shown to depend on sample size, number of bin and binning scheme (Hosmer et. al. 1997). We will generate fake data to show the size of HL test.

[4]:
```
### The size of HL Test
from calzone.utils import fake_binary_data_generator
np.random.seed(123)
fakedata_generator = fake_binary_data_generator(alpha_val=0.5, beta_val=0.5)
cal_metrics = CalibrationMetrics()
sample_size = 1000
simulation_size = 10000
results = []
# generate data
for i in range(simulation_size):
    X, y = fakedata_generator.generate_data(sample_size)
    if i == 0:
        tempresult = cal_metrics.calculate_metrics(y, X, ['HL-H', 'HL-C'],return_
↪numpy=False, df=10)
        keys = list(tempresult.keys())
        results.append(np.array(list(tempresult.values())))
    else:
        tempresult = cal_metrics.calculate_metrics(y, X, ['HL-H', 'HL-C'],return_
↪numpy=True, df=10)
        results.append(tempresult)
results = np.array(results)
```

[5]:
```
### Showing the size of the model
import matplotlib.pyplot as plt
hl_h_pvalue = results[:,1]
hl_c_pvalue = results[:,3]
size_h = np.mean(hl_h_pvalue < 0.05)
size_c = np.mean(hl_c_pvalue < 0.05)
print("The size of HL-H is :", round(size_h,3))
print("The size of HL-C is :", round(size_c,3))
```

```
The size of HL-H is : 0.047
The size of HL-C is : 0.055
```

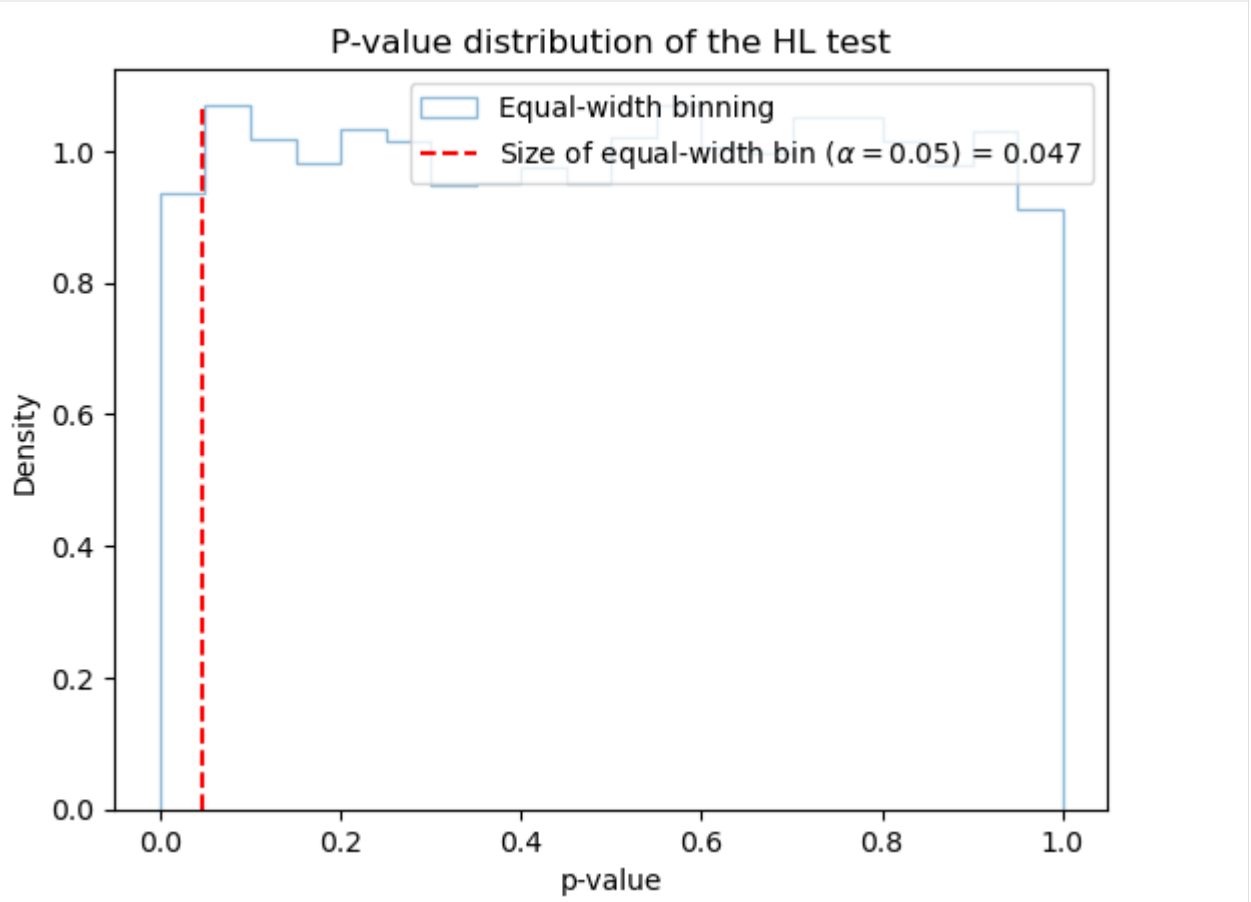
[6]:
```
value,_,_=plt.hist(hl_h_pvalue, bins=20, density=True, alpha=0.5, label='Equal-width␣
↪binning',histtype='step')
```

(continues on next page)

(continued from previous page)

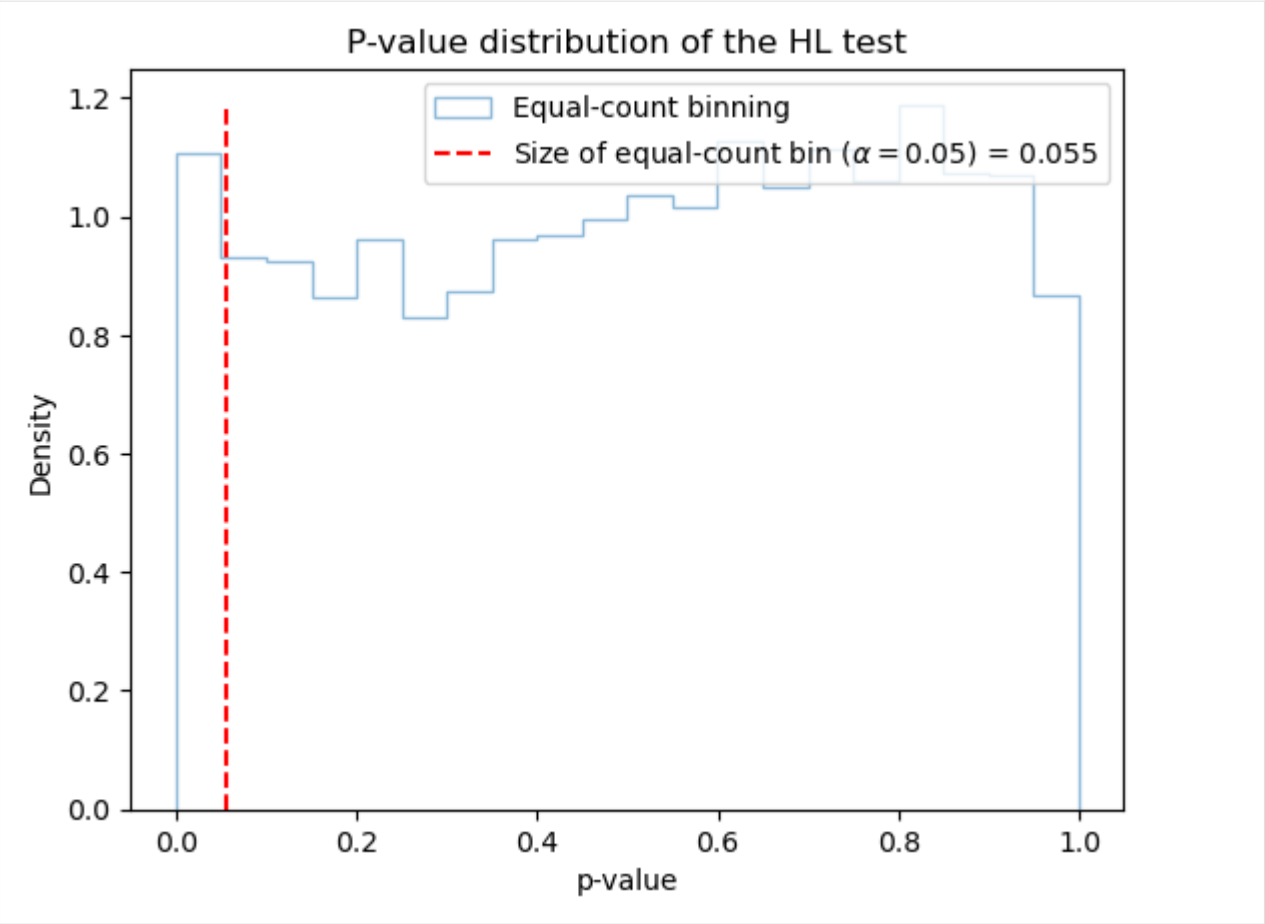
```
plt.vlines(size_h,ymin=0,ymax=np.max(value),linestyle='--',label=r'Size of equal-width␣
↪bin ($\alpha=0.05$) = '+str(round(size_h,3)),colors='r')
plt.legend(loc='upper right')
plt.xlabel('p-value')
plt.ylabel('Density')
plt.title('P-value distribution of the HL test')
```

[6]:
```
Text(0.5, 1.0, 'P-value distribution of the HL test')
```



[7]:
```
value,_,_=plt.hist(hl_c_pvalue, bins=20, density=True, alpha=0.5, label='Equal-count␣
↪binning',histtype='step')
plt.vlines(size_c,ymin=0,ymax=np.max(value),linestyle='--',label=r'Size of equal-count␣
↪bin ($\alpha=0.05$) = '+str(round(size_c,3)),colors='r')
plt.legend(loc='upper right')
plt.xlabel('p-value')
plt.ylabel('Density')
plt.title('P-value distribution of the HL test')
```

[7]:
```
Text(0.5, 1.0, 'P-value distribution of the HL test')
```



We can see the equal-width and equal-count binning gives a different result.

5.6 Reference

Hosmer, D. W., & Lemesbow, S. (1980). Goodness of fit tests for the multiple logistic regression model. Communications in statistics-Theory and Methods, 9(10), 1043-1069.

Hosmer, D. W., Hosmer, T., Cessie, S. L., & Lemeshow, S. (1997). A COMPARISON OF GOODNESS-OF-FIT TESTS FOR THE LOGISTIC REGRESSION MODEL. 16.

CHAPTER

SIX

COX CALIBRATION ANALYSIS

6.1 Theoretical Background

Cox calibration analysis is both a logistic recalibration technique and a method to examine the current calibration of a model. To perform the analysis, we first need to fit a new logistic regression model using logit (log odds, aka $\log \frac{\hat{p}}{1-\hat{p}}$) as the predictor variable and the outcome as the target variable.

$$p_{new} = \frac{1}{1 + e^{-(a+b \cdot \log \frac{\hat{p}}{1-\hat{p}})}}$$

In the case of perfect calibration, $P(Y = 1|p = \hat{p}) = \hat{p}$ and the new probability p_{new} is equal to the original probability $\hat{p}$. That means $a = 0$ and $b = 1$. If $b > 1$, the model is under-confident at high probabilities and over-confident at low probabilities for the class-of-interest. If $b < 1$, the model is over-confident at high probabilities and under-confident at low probabilities for the class-of-interest. If $a > 0$, the model is over-confident at all probabilities for the class-of-interest. If $a < 0$, the model is under-confident at all probabilities for the class-of-interest. The confidence interval of a and b can be used to guide the calibration of the model. The user can also choose to fix $a = 0$ and fit for b only and vice versa, then there will be no interaction between a and b and the confidence interval can be used as a statistical test to test for perfect calibration.

6.2 Pros of Cox calibration analysis

Cox calibration analysis doesn't depend on binning of data, which is a big advantage since common metrics such as ECE/MCE and HL test all depend on binning and we have shown that changing binning can lead to different results. We can also use it to perform statistical tests by fixing a to 0 and test whether $b = 1$ and the other way around to test for perfect calibration. Also, the fitted values of a and b can tell us how the model is miscalibrated, whether it is an overall under- or over-confidence or if it is over-confident in some ranges and under-confident in others. For example, if a is not close to 0 while b is close to 1, it likely indicates a prevalence shift. See more details in the prevalence adjustment notebook.

6.3 Cons of Cox calibration analysis

Cox Calibration analysis can only assess weak calibration and only captures certain types of miscalibration (general over/under-confidence). A model can have $a = 0$ and $b = 1$ and still be miscalibrated.

6.4 Calculating Cox slope and intercept with calzone

There are two ways to calculate the Cox slope and intercept. Calling the Cox function gives you more control over the calculation, including fixing $a = 0$ or $b = 1$.

[1]:
```
from calzone.utils import reliability_diagram,data_loader
from calzone.metrics import cox_regression_analysis
import numpy as np

### loading the data
wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")

### calculating cox slope and intercept
cox_slope, cox_intercept,cox_slope_ci,cox_intercept_ci = cox_regression_analysis(wellcal_
↪dataloader.labels, wellcal_dataloader.probs,class_to_calculate=1,print_results=True)
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 5000
Model:                          Logit   Df Residuals:                     4998
Method:                           MLE   Df Model:                            1
Date:                Wed, 02 Oct 2024   Pseudo R-squ.:                  0.4438
Time:                        15:53:27   Log-Likelihood:                -1927.5
converged:                       True   LL-Null:                       -3465.6
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0450      0.040     -1.123      0.262      -0.123       0.034
x1             0.9942      0.029     34.212      0.000       0.937       1.051
==============================================================================
```

You can also fix the intercept $a = 0$ by using the `fix_intercept=True` option. Similarly, you can fix the slope $b = 1$ by using the `fix_slope=True` option.

[2]:
```
### fixing intercept and calculating cox slope
cox_slope, cox_intercept,cox_slope_ci,cox_intercept_ci = cox_regression_analysis(wellcal_
↪dataloader.labels, wellcal_dataloader.probs,class_to_calculate=1, fix_intercept=True,
↪print_results=True)
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.385628
         Iterations: 0
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 5000
Model:                          Logit   Df Residuals:                     4999
Method:                           MLE   Df Model:                            0
Date:                Wed, 02 Oct 2024   Pseudo R-squ.:                  0.4436
Time:                        15:53:27   Log-Likelihood:                -1928.1
converged:                      False   LL-Null:                       -3465.6
Covariance Type:            nonrobust   LLR p-value:                       nan
==============================================================================
```

(continues on next page)

(continued from previous page)

```
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const               0          0        nan        nan          0          0
x1             0.9939      0.029     34.210      0.000      0.937      1.051
==============================================================================

Model has been estimated subject to linear equality constraints.
```

Alternatively, we can use the CalibrationMetrics class to compute the COX slope and intercept.

[3]:
```
from calzone.metrics import CalibrationMetrics
calmetrics = CalibrationMetrics()
calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_dataloader.probs,␣
↪metrics=['COX'],print_results=True)
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 5000
Model:                          Logit   Df Residuals:                     4998
Method:                           MLE   Df Model:                            1
Date:                Wed, 02 Oct 2024   Pseudo R-squ.:                 0.4438
Time:                        15:53:27   Log-Likelihood:                -1927.5
converged:                       True   LL-Null:                       -3465.6
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0450      0.040     -1.123      0.262      -0.123       0.034
x1             0.9942      0.029     34.212      0.000       0.937       1.051
==============================================================================
```

[3]:
```
{'COX coef': 0.9942499557748269,
 'COX intercept': -0.04497652296600376,
 'COX coef lowerci': 0.9372902801721911,
 'COX coef upperci': 1.0512096313774626,
 'COX intercept lowerci': -0.12348577118577644,
 'COX intercept upperci': 0.03353272525376893,
 'COX ICI': 0.005610391483826338}
```

The resulting COX slope and intercept can be used to calibrate the model but it is beyond the scope of this package.

6.5 Size of COX slope and intecept test

Although Cox calibration analysis is usually only used to estimate the overall calibration trend, the resulting estimates of the slope and intercept can also be used to test whether the model is well calibrated (moderate calibration). We will do a demostrate on the size of the slope and intercept test below

[ ]:
```
### The size of slope test
from calzone.utils import fake_binary_data_generator
np.random.seed(123)
fakedata_generator = fake_binary_data_generator(alpha_val=0.5, beta_val=0.5)
cal_metrics = CalibrationMetrics()
```

(continues on next page)

(continued from previous page)

```
sample_size = 1000
simulation_size = 1000
results = []
# generate data
for i in range(simulation_size):
    X, y = fakedata_generator.generate_data(sample_size)
    if i == 0:
        tempresult = cal_metrics.calculate_metrics(y, X, ['COX'],return_numpy=False,fix_
↪intercept=True) #we need to fix the intercept to be 0
        keys = list(tempresult.keys())
        results.append(np.array(list(tempresult.values())))
    else:
        tempresult = cal_metrics.calculate_metrics(y, X, ['COX'],return_numpy=True,fix_
↪intercept=True) #we need to fix the intercept to be 0
        results.append(tempresult)
results = np.array(results)
```

[5]:
```
Cox_slope = results[:,0]
Cox_slope_lowerci = results[:,2]
Cox_slope_upperci = results[:,3]
chance = np.logical_and(Cox_slope_lowerci<=1, Cox_slope_upperci>=1)
print('The size of the Cox slope test is: ', 1-np.mean(chance))
```

```
The size of the Cox slope test is:  0.039000000000000035
```

We can also do the intercept test:

[ ]:
```
### The size of intercept test
from calzone.utils import fake_binary_data_generator
np.random.seed(123)
fakedata_generator = fake_binary_data_generator(alpha_val=0.5, beta_val=0.5)
cal_metrics = CalibrationMetrics()
sample_size = 1000
simulation_size = 1000
results = []
# generate data
for i in range(simulation_size):
    X, y = fakedata_generator.generate_data(sample_size)
    if i == 0:
        tempresult = cal_metrics.calculate_metrics(y, X, ['COX'],return_numpy=False,fix_
↪slope=True) #we need to fix the slope to be 1
        keys = list(tempresult.keys())
        results.append(np.array(list(tempresult.values())))
    else:
        tempresult = cal_metrics.calculate_metrics(y, X, ['COX'],return_numpy=True,fix_
↪slope=True) #we need to fix the slope to be 1
        results.append(tempresult)
results = np.array(results)
```

[7]:
```
Cox_intercept = results[:,1]
Cox_intercept_lowerci = results[:,4]
```

(continues on next page)

(continued from previous page)

```
Cox_intercept_upperci = results[:,5]
chance = np.logical_and(Cox_intercept_lowerci<=0, Cox_intercept_upperci>=0)
print('The size of the Cox intercept test is: ', 1-np.mean(chance))
```

```
The size of the Cox intercept test is:  0.05600000000000005
```

Both test doesn't have the exact size but it is closer than the HL test.

6.6 References

Cox, D. R. (1958). Two Further Applications of a Model for Binary Regression.

Calster, B. V., & Steyerberg, E. W. (2018). Calibration of Prognostic Risk Scores. In R. S. Kenett, N. T. Longford, W. W. Piegorsch, & F. Ruggeri (Eds.), Wiley StatsRef: Statistics Reference Online (1st ed., pp. 1–10). Wiley. https://doi.org/10.1002/9781118445112.stat08078

Huang, Y., Li, W., Macheret, F., Gabriel, R. A., & Ohno-Machado, L. (2020). A tutorial on calibration measurements and calibration models for clinical prediction models. Journal of the American Medical Informatics Association, 27(4), 621–633. https://doi.org/10.1093/jamia/ocz228

CHAPTER

SEVEN

INTEGRATED CALIBRATION INDEX (ICI)

7.1 Theoretical Background

Integrated Calibration Index (ICI) is essentially the same as expected calibration error (ECE) in terms of the idea. They both try to measure the average deviation of the predicted probabilities from the true probabilities. However, ECE is calculated by grouping the samples into bins and then calculating the weighted average of the deviation of the mean predicted probabilities from the empirical accuracy. ICI, on the other hand, is calculated by fitting a smooth curve using the samples itself and therefore doesn't require binning. However, the choice of the curve fitting method can affect the result and is arbitrary. The most common choice is locally estimated scatterplot smoothing (LOESS) (Cleveland, 1979). People also use other methods such as polynomial fitting and spline fitting. Interestingly, not many people have looked into using COX regression results to calculate ICI, which is implemented in calzone. Notice that the Cox-ICI can be way off from the truth if the logistic regression is not a good fit.

The formula for ICI is:

$$\text{ICI} = \int_0^1 |\mathbb{S}(p) - p| \, dF_{\hat{P}}(p)$$

where $\mathbb{S}(p)$ is the fitted function and $F_{\hat{P}}(p)$ is the cumulative distribution function of the empirical probabilities. In the actual implementation, we calculate

$$\text{ICI} = \frac{1}{N} \sum_{i=1}^{N} |\mathbb{S}(p_i) - p_i|$$

where p_i is the predicted probability of the i-th sample.

7.2 Pros of ICI

The main advantage of ICI is that it skips the need for binning while still giving you an easily interpretable metric. It is essentially the same as ECE and can be interpreted as the average deviation from the true probability. ICI can capture any type of miscalibration if the calibration curve is well-described by the fitting method.

7.3 Cons of ICI

The main disadvantage of ICI is the need for a fitting method. Locally estimated scatterplot smoothing (LOESS) is a non-parametric regression method that fits a smooth line through the data. It is the most common method used in ICI. However, it still requires hyperparameters like the span (window width) which could affect the fitting result and ICI greatly.

7.4 Calculating LOESS ICI and COX ICI using calzone

To calculate LOESS ICI and COX ICI using calzone, we can call the function directly

[1]:
```
from calzone.utils import reliability_diagram,data_loader
from calzone.metrics import cox_regression_analysis,lowess_regression_analysis,cal_ICI_
↪cox

### loading the data
wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")

### calculating cox ICI
cox_slope, cox_intercept,cox_slope_ci,cox_intercept_ci = cox_regression_analysis(wellcal_
↪dataloader.labels, wellcal_dataloader.probs,class_to_calculate=1,print_results=False)
cox_ici = cal_ICI_cox(cox_slope, cox_intercept, wellcal_dataloader.probs, class_to_
↪calculate=1)

### calculating loess ICI
loess_ici, lowess_fit_p, lowess_fit_p_correct = lowess_regression_analysis(wellcal_
↪dataloader.labels, wellcal_dataloader.probs, class_to_calculate=1, span=0.5, delta=0.
↪001, it=0)

print(f"Cox ICI: {cox_ici}")
print(f"Loess ICI: {loess_ici}")
```

```
Cox ICI: 0.005610391483826338
Loess ICI: 0.00558856942568957
```

Alternatively, we can use the CalibrationMetrics class to compute the COX and Loess ICI

[2]:
```
from calzone.metrics import CalibrationMetrics
calmetrics = CalibrationMetrics()
calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_dataloader.probs,␣
↪metrics=['COX','Loess'])
```

[2]:
```
{'COX coef': 0.9942499557748269,
 'COX intercept': -0.04497652296600376,
 'COX coef lowerci': 0.9372902801721911,
 'COX coef upperci': 1.0512096313774626,
 'COX intercept lowerci': -0.12348577118577644,
 'COX intercept upperci': 0.03353272525376893,
 'COX ICI': 0.005610391483826338,
 'Loess ICI': 0.00558856942568957}
```

7.5 Visualization of the fitted curve

We can also plot the loess curve and the COX curve.

[3]:
```
### We will use linear miscalibrated data to demostrate the plot. Notice that the COX␣
↪should capture the miscalibration perfectly in the example case.
miscal_dataloader = data_loader(data_path="../../../example_data/simulated_misdata.csv")

### calculating cox ICI
cox_slope, cox_intercept,cox_slope_ci,cox_intercept_ci = cox_regression_analysis(miscal_
↪dataloader.labels, miscal_dataloader.probs,class_to_calculate=1,print_results=False)
cox_ici = cal_ICI_cox(cox_slope, cox_intercept, wellcal_dataloader.probs, class_to_
↪calculate=1)

### calculating loess ICI
loess_ici, lowess_fit_p, lowess_fit_p_correct = lowess_regression_analysis(miscal_
↪dataloader.labels, miscal_dataloader.probs, class_to_calculate=1)

### We also try a different span
loess_ici2, lowess_fit_p2, lowess_fit_p2_correct = lowess_regression_analysis(miscal_
↪dataloader.labels, miscal_dataloader.probs, class_to_calculate=1, span=0.3, delta=0.
↪001, it=0)

print("Cox ICI: ", cox_ici)
print("Loess ICI (span = 0.5): ", loess_ici)
print("Loess ICI (span = 0.3): ", loess_ici2)
```

```
Cox ICI:  0.0984122810555748
Loess ICI (span = 0.5):  0.07356445428053172
Loess ICI (span = 0.3):  0.07692218401743334
```

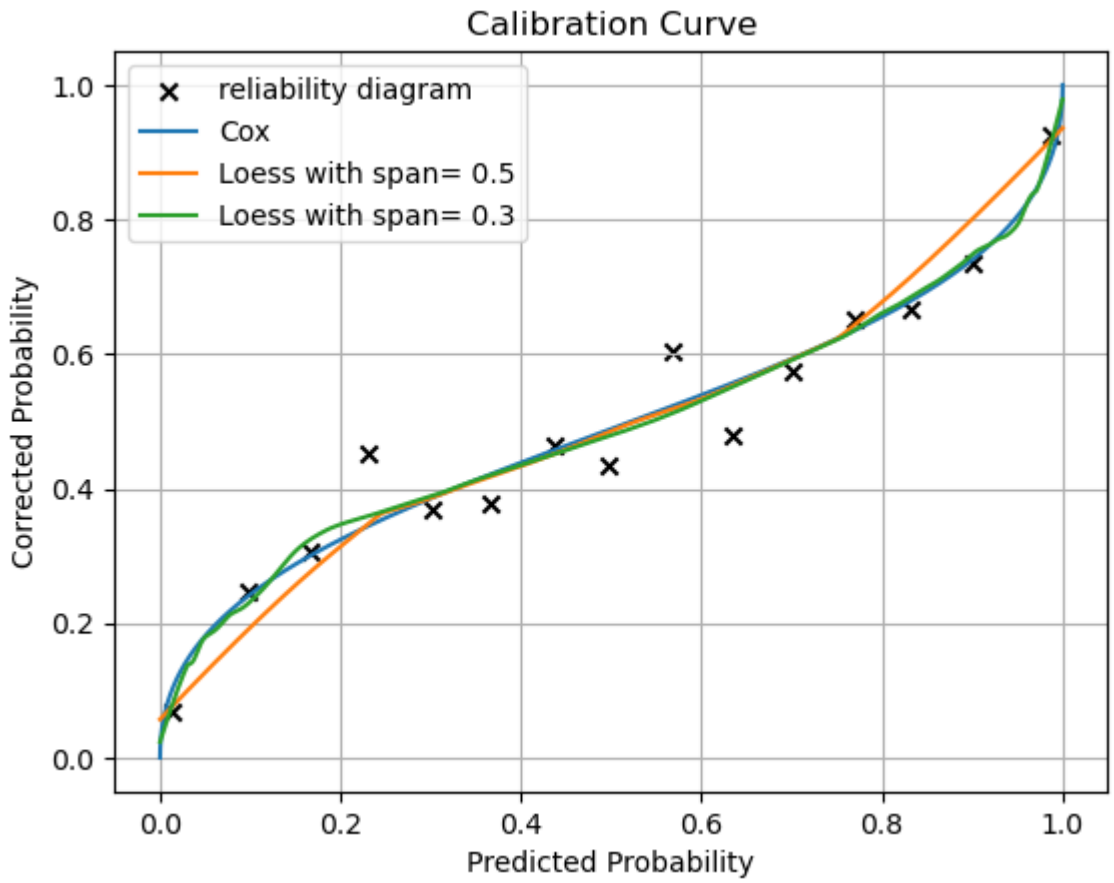
[4]:
```
### ploting the curve
from calzone.metrics import logit_func
import matplotlib.pyplot as plt
from calzone.utils import reliability_diagram
from calzone.vis import plot_reliability_diagram
import numpy as np
cox_func= logit_func(coef=cox_slope, intercept=cox_intercept)
proba_class1 = np.sort(np.clip(miscal_dataloader.probs[:,1],1e-10,1-1e-10))
cox_p_correct=cox_func(proba_class1)
reliability,confindence,bin_edges,bin_counts = reliability_diagram(miscal_dataloader.
↪labels, miscal_dataloader.probs, num_bins=15,class_to_plot=1)
plt.scatter(confindence,reliability, label="reliability diagram", marker="x",color='k')
#ig = plot_reliability_diagram(reliability, confindence, bin_counts)
#plt.close()
#print(fig)
plt.plot(proba_class1, cox_p_correct, label="Cox")
plt.plot(lowess_fit_p, lowess_fit_p_correct, label="Loess with span= 0.5")
plt.plot(lowess_fit_p2, lowess_fit_p2_correct, label="Loess with span= 0.3")
plt.xlabel("Predicted Probability")
plt.ylabel("Corrected Probability")
plt.legend()
plt.grid()
```

(continues on next page)

(continued from previous page)

```
plt.title("Calibration Curve")
```

```
[4]: Text(0.5, 1.0, 'Calibration Curve')
```



7.6 Reference

Cleveland, W.S. (1979) "Robust Locally Weighted Regression and Smoothing Scatterplots". Journal of the American Statistical Association 74 (368): 829-836.

Austin, P. C., & Steyerberg, E. W. (2019). The Integrated Calibration Index (ICI) and related metrics for quantifying the calibration of logistic regression models. Statistics in Medicine, 38(21), 4051–4065. https://doi.org/10.1002/sim.8281

Huang, Y., Li, W., Macheret, F., Gabriel, R. A., & Ohno-Machado, L. (2020). A tutorial on calibration measurements and calibration models for clinical prediction models. Journal of the American Medical Informatics Association, 27(4), 621–633. https://doi.org/10.1093/jamia/ocz228

CHAPTER

EIGHT

SPIEGELHALTER'S Z-TEST

8.1 Theoretical background

Spiegelhalter's Z-test is a statistical test that tests whether a probabilistic model is calibrated. It is named after the statistician David Spiegelhalter, who proposed it in 1986. It is a non-parametric test that does not require any binning.

The Spiegelhalter's Z-test was inspired by the fact that the Brier score (mean squared error) can be decomposed into reliability and resolution. In fact, any proper scoring rule can be decomposed into reliability and resolution, as shown by Brocker (2008). For example, the cross-entropy can be decomposed into KL-divergence (reliability) and entropy (resolution).

The Brier score can be decomposed into reliability and resolution as follows:

$$B = \frac{1}{N}\sum_{i=1}^{N}(x_i - p_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - p_i)(1 - 2p_i) + \frac{1}{N}\sum_{i=1}^{N} p_i(1 - p_i)$$

Where the first term measure the reliability/calibration and the second term measure the resolution/discrimination.

The Variance of the Brier score is:

$$\text{Var}(B) = \frac{1}{N^2}\sum_{i=1}^{N}(1 - 2p_i)^2 p_i(1 - p_i)$$

and the Speigelhalter's Z-test is defined as:

$$Z = \frac{B - E(B)}{\sqrt{\text{Var}(B)}} = \frac{\sum_{i=1}^{N}(x_i - p_i)(1 - 2p_i)}{\sum_{i=1}^{N}(1 - 2p_i)^2 p_i(1 - p_i)}$$

and Z is approximately standard normal distributed under the null hypothesis of calibration. Spiegelhalter's Z-test has the right size in many situations and it is powerful in many situations. We recommend using Spiegelhalter's Z-test to test the calibration of probabilistic models.

8.2 Pros of Spiegelhalter's Z test

Spiegelhalter's Z test is a statistical test which can provide statistical evidence that the null hypothesis (well-calibrated) is true or false. It is a non-parametric test and doesn't require any hyperparameter tuning. It also doesn't require any binning of data, which is extremely useful compared to the Hosmer-Lemeshow test.

8.3 Cons of Spiegelhalter's Z test

The power of Spiegelhalter's Z test is limited for some cases of miscalibration, such as prevalence shift. However, it is a very powerful test for many other cases of miscalibration.

8.4 Calculating the Spieegelhalter Z score and p-value using calzone

We can call functions from the calzone package to calculate the Spiegelhalter Z score and p-value directly.

[1]:
```
from calzone.utils import reliability_diagram,data_loader
from calzone.metrics import spiegelhalter_z_test
import numpy as np

wellcal_dataloader = data_loader(data_path="../../../example_data/simulated_welldata.csv
↪")
z,p_value = spiegelhalter_z_test(wellcal_dataloader.labels,wellcal_dataloader.probs,
↪class_to_calculate=1)
print(f"Z-score: {z}, p-value: {p_value}")
```

```
Z-score: 0.3763269161877356, p-value: 0.7066738713391099
```

We can also use the CalibrationMetrics class

[2]:
```
from calzone.metrics import CalibrationMetrics
calmetrics = CalibrationMetrics(class_to_calculate=1)
calmetrics.calculate_metrics(wellcal_dataloader.labels, wellcal_dataloader.probs,␣
↪metrics=['SpiegelhalterZ'])
```

[2]:
```
{'SpiegelhalterZ score': 0.3763269161877356,
 'SpiegelhalterZ p-value': 0.7066738713391099}
```

8.5 Testing the size of Spiegelhalter's z test

Like to HL test, we can check whether the Spiegelhalter's z test has the correct size.

[3]:
```
### The size of HL Test
from calzone.utils import fake_binary_data_generator
np.random.seed(123)
fakedata_generator = fake_binary_data_generator(alpha_val=0.5, beta_val=0.5)
cal_metrics = CalibrationMetrics()
sample_size = 1000
simulation_size = 10000
results = []
# generate data
for i in range(simulation_size):
    X, y = fakedata_generator.generate_data(sample_size)
    if i == 0:
        tempresult = cal_metrics.calculate_metrics(y, X, ['SpiegelhalterZ'],return_
↪numpy=False)
        keys = list(tempresult.keys())
```

(continues on next page)

(continued from previous page)

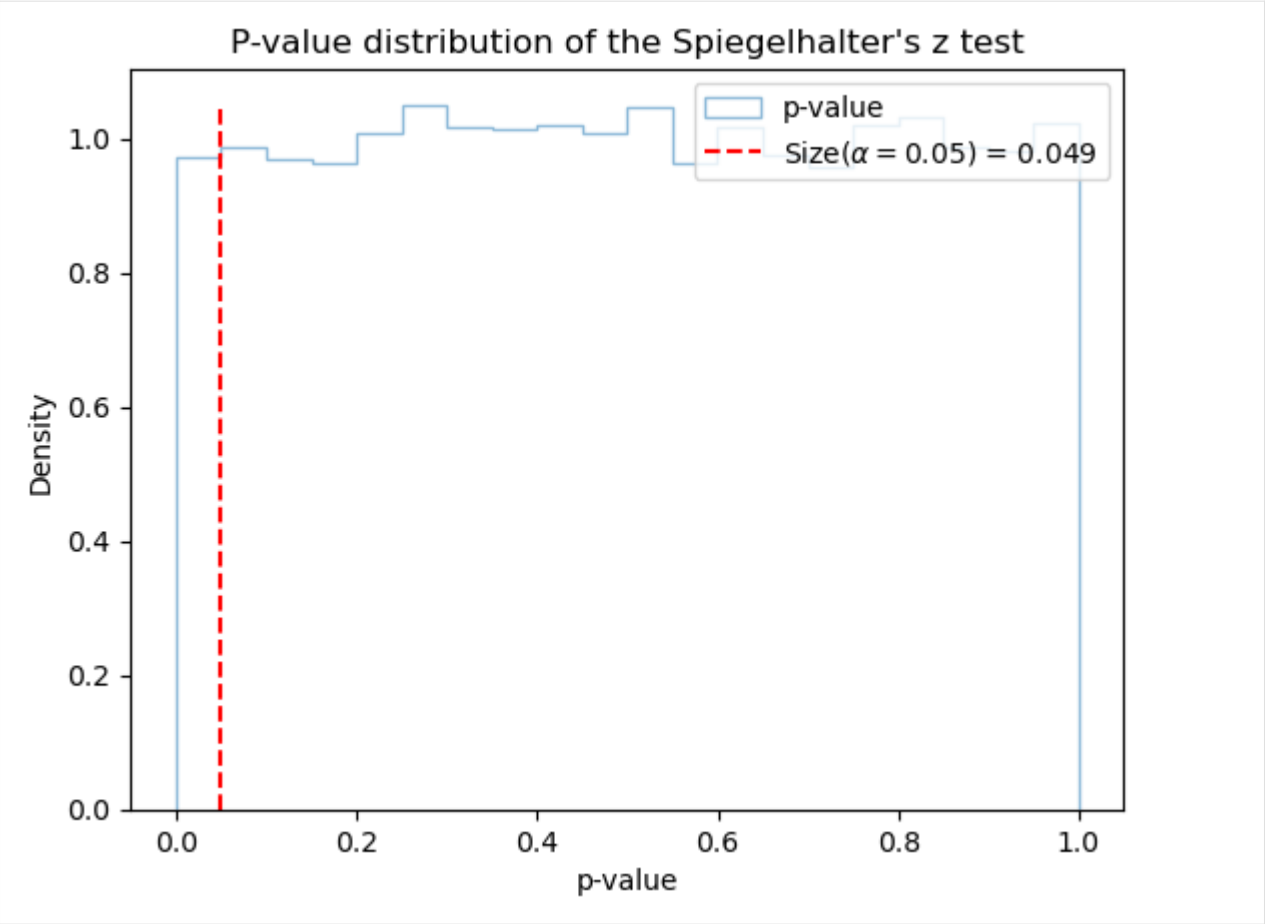
```
        results.append(np.array(list(tempresult.values())))
    else:
        tempresult = cal_metrics.calculate_metrics(y, X, ['SpiegelhalterZ'],return_
↪numpy=True)
        results.append(tempresult)
results = np.array(results)
```

[4]:
```
### Showing the size of the model
import matplotlib.pyplot as plt
z_scores = results[:,0]
p_values = results[:,1]
size = np.mean(p_values < 0.05)
print("The size of Spiegelhalter's z test is :", round(size,3))
```

```
The size of Spiegelhalter's z test is : 0.049
```

[5]:
```
value,_,_=plt.hist(p_values, bins=20, density=True, alpha=0.5, label='p-value',histtype=
↪'step')
plt.vlines(size,ymin=0,ymax=np.max(value),linestyle='--',label=r'Size($\alpha=0.05$) =
↪'+str(round(size,3)),colors='r')
plt.legend(loc='upper right')
plt.xlabel('p-value')
plt.ylabel('Density')
plt.title("P-value distribution of the Spiegelhalter's z test")
```

[5]:
```
Text(0.5, 1.0, "P-value distribution of the Spiegelhalter's z test")
```



We can see that Spiegelhalter's Z test has a accurate size.

8.6 Reference

Spiegelhalter, D. J. (1986). Probabilistic prediction in patient management and clinical trials.

Bröcker, J. (2009). Reliability, Sufficiency, and the Decomposition of Proper Scores. Quarterly Journal of the Royal Meteorological Society, 135(643), 1512–1519. https://doi.org/10.1002/qj.456

CHAPTER

NINE

PREVALENCE ADJUSTMENT

In this notebook, we will discuss how prevalence will affect the calibration of the model in a binary classification problem and how to adjust for prevalence differences.

When we discuss calibration, we usually refer to whether the probability output by the model matches the posterior probability of the true outcome.

$$P(D = 1|\hat{p} = p) = p, \forall p \in [0, 1]$$

where $\hat{p}$ is the predicted probability of the true outcome being 1.

However, the posterior probability of the true outcome being 1 depends on the prevalence of the outcome 1. Using Bayes' theorem, we can derive the following relationship:

$$P(D = 1|\hat{p} = p) = \frac{P(\hat{p} = p|D = 1)P(D = 1)}{P(\hat{p} = p)}$$

The term $P(\hat{p} = p|D = 1)$ is independent of prevalence for a given model. The term $P(D = 1)$ is the prevalence of the outcome 1. The term $P(\hat{p} = p)$ is the marginal probability of the predicted probability being p and implicitly depends on the prevalence of the true outcome. We can expand the denominator using the fact that $P(\hat{p} = p) = P(\hat{p} = p|D = 1)\eta + P(\hat{p} = p|D = 0)(1 - \eta)$. Further rearranging the above equation will lead to the following equation:

$$P(D = 1|\hat{p} = p) = \frac{\text{LR}(p) \times \eta}{\text{LR}(p) \times \eta + 1 - \eta}$$

where $\text{LR}(p) = \frac{P(\hat{p}=p|D=1)}{P(\hat{p}=p|D=0)}$ is the likelihood ratio of the predicted probability being p given the true outcome being 1 and 0 respectively, and η is the prevalence of the outcome 1.

The likelihood ratio is independent of the prevalence, so that the model can be calibrated for a specific prevalence but will become mis-calibrated for a different prevalence. We can say such a model is "intrinsically calibrated", meaning that the likelihood ratio of the model with a specific prevalence produced a correct posterior probability of the true outcome being 1.

An intrinsically calibrated model can be adapted to a population with a different prevalence but the same probability distribution within class. To adjust for prevalence differences, we rely on the fact that the likelihood ratio is independent of the prevalence. We can use the following equation to adjust the predicted probability of the true outcome being 1 for a different prevalence:

$$P(D = 1|\hat{p} = p) = \frac{\eta LR(p)}{\eta LR(p) + (1 - \eta)} = p$$

$$LR(p) = \frac{p}{1 - p} \cdot \frac{1 - \eta}{\eta}$$

$$P'(D = 1|\hat{p} = p) = \frac{\eta' LR(p)}{\eta' LR(p) + (1 - \eta')} = \frac{\eta'/(1 - \eta')}{(1/p - 1)(\eta/(1 - \eta))} = p'$$

where η is the prevalence of the derivation population (aka the population for which the model is calibrated) and η' is the prevalence of the outcome 1 in the new population. We will refer to p' as the adjusted probability.

In practice, we might have a dataset with the true label (which we can use to calculate the prevalence η) and predicted probability of the true outcome being 1. We can search for the derivation prevalence η that minimizes cross-entropy loss between the adjusted probability p' and the posterior probability of the true outcome being 1.

$$\min_{\eta} \sum_{i=1}^{N} (y_i \log(p'_i) + (1 - y_i) \log(1 - p'_i))$$

Notice that minimizing cross-entropy loss with respect to η is equivalent to minimizing the KL divergence since the prevalence adjustment is a monotonic transformation and doesn't affect the resolution component of the cross-entropy loss.

9.1 Preform prevalence adjustment in calzone

We will demonstrate how to perform prevalence adjustment in calzone. The first method is to find optimal prevalence first and apply the adjustment.

[1]:
```
##

from calzone.utils import find_optimal_prevalence,apply_prevalence_adjustment,data_
↪loader,fake_binary_data_generator
import numpy as np
# We generate data and drop the prevalence

np.random.seed(123)
fakedata_generator = fake_binary_data_generator(alpha_val=0.5, beta_val=0.5)
X, y = fakedata_generator.generate_data(5000)
### drop half the outcome 1 prevalence
class_1_index = (y==1)
class_1_samples = np.where(class_1_index)[0]
drop_indices = np.random.choice(class_1_samples, size=int(len(class_1_samples)/2),␣
↪replace=False)

mask = np.ones(len(y), dtype=bool)
mask[drop_indices] = False

y = y[mask]
X = X[mask]
optimal_prevalence,adjusted_p = find_optimal_prevalence(y, X, class_to_calculate=1)
print("Dataset prevalence: ", np.mean(y))
print("Derived prevalence: ", optimal_prevalence)
```

```
Dataset prevalence:  0.3300531914893617
Derived prevalence:  0.49863799264980607
```

The function return both the derived prevalence and the adjusted probability. We can also use the derived prevalence adjustment factor to perform the adjustment mannually.

[2]:
```
### Prevalence Adjustment
from calzone.metrics import lowess_regression_analysis
proba_adjust = apply_prevalence_adjustment(optimal_prevalence, y, X, class_to_
```

(continues on next page)

(continued from previous page)

```
↪calculate=1)
print('Loess ICI before prevalence adjustment: ', lowess_regression_analysis(y, X, class_
↪to_calculate=1)[0])
print('Loess ICI after prevalence adjustment: ', lowess_regression_analysis(y, proba_
↪adjust, class_to_calculate=1)[0])
```

```
Loess ICI before prevalence adjustment:  0.07961758926734244
Loess ICI after prevalence adjustment:  0.008745511902314453
```

calzone also provides a argument to perform prevalence adjustment directly from the CalibrationMetrics class.

[3]:
```
### We calculate the Calibration metrics before and after prevalence adjustment
from calzone.metrics import CalibrationMetrics
calmetrics = CalibrationMetrics()
before_prevalence = calmetrics.calculate_metrics(y,X, metrics=['ECE-H','COX','Loess'],
↪perform_pervalance_adjustment=False)
after_prevalence = calmetrics.calculate_metrics(y,X, metrics=['ECE-H','COX','Loess'],
↪perform_pervalance_adjustment=True)
```

[4]:
```
for key in before_prevalence.keys():
    print(key)
    print('before adjustment:',before_prevalence[key],', after adjustment:',after_
↪prevalence[key])
```

```
ECE-H topclass
before adjustment: 0.014081013182402267 , after adjustment: 0.010355911839501922
ECE-H
before adjustment: 0.0841517729106883 , after adjustment: 0.013671230516636386
COX coef
before adjustment: 0.9400481147756811 , after adjustment: 0.9400481147756811
COX intercept
before adjustment: -0.6897839569176842 , after adjustment: -0.029403495083063648
COX coef lowerci
before adjustment: 0.8754203499121679 , after adjustment: 0.8754203499121678
COX coef upperci
before adjustment: 1.0046758796391944 , after adjustment: 1.0046758796391944
COX intercept lowerci
before adjustment: -0.7837388214288888 , after adjustment: -0.12775157222121533
COX intercept upperci
before adjustment: -0.5958290924064796 , after adjustment: 0.06894458205508802
COX ICI
before adjustment: 0.0841517733462589 , after adjustment: 0.007508966220374058
Loess ICI
before adjustment: 0.07961758926734244 , after adjustment: 0.008745511902314453
```

9.2 Prevalence adjustment and constant shift in logit of class-of-interest

In the section, we will prove that the prevalence shift is equivalent to a constant shift in logit of class-of-interest. In other words, prevalence adjustment can be done by addint a constant to the logit of class-of-interest. For the calibrated case, the likelihood ratio of the two classes is:

$$LR(p) = \frac{\frac{e^{x_2}}{e^{x_1}+e^{x_2}}}{\frac{e^{x_1}}{e^{x_1}+e^{x_2}}} \cdot \frac{1-\eta}{\eta} = e^{x_2-x_1} \cdot \frac{1-\eta}{\eta}$$

Assumer we add a constant c to the logit of class-of-interest (x_2 here), the likelihood ratio becomes:

$$LR'(p) = e^{x_2-x_1+c} \cdot \frac{1-\eta}{\eta}$$

And the posterior probability becomes:

$$P'(D=1|\hat{p}=p) = \frac{\eta LR'(p)}{\eta LR'(p) + (1-\eta)} = \frac{\eta LR(p) \cdot e^c}{\eta LR(p) \cdot e^c + (1-\eta)}$$

Which is equivalent to the posterior probability after prevalence adjustment:

$$\frac{\eta' LR(p)}{\eta' LR(p) + (1-\eta')}$$

By setting

$$\eta' = \frac{1}{1 + e^a \left(\frac{1-\eta}{\eta}\right)}$$

Therefore, prevalence adjustment is equivalent to a constant shift in logit of class-of-interest.

9.3 References

Chen, W., Sahiner, B., Samuelson, F., Pezeshk, A., & Petrick, N. (2018). Calibration of medical diagnostic classifier scores to the probability of disease. Statistical Methods in Medical Research, 27(5), 1394–1409. https://doi.org/10.1177/0962280216661371

Gu, W., & Pepe, M. S. (2011). Estimating the diagnostic likelihood ratio of a continuous marker. Biostatistics, 12(1), 87–101. https://doi.org/10.1093/biostatistics/kxq045

Tian, J., Liu, Y.-C., Glaser, N., Hsu, Y.-C., & Kira, Z. (2020). Posterior Re-calibration for Imbalanced Datasets (No. arXiv:2010.11820). arXiv. http://arxiv.org/abs/2010.11820

Horsch, K., Giger, M. L., & Metz, C. E. (2008). Prevalence scaling: applications to an intelligent workstation for the diagnosis of breast cancer. Academic radiology, 15(11), 1446–1457. https://doi.org/10.1016/j.acra.2008.04.022

CHAPTER

TEN

SUBGROUP ANALYSIS

In many real-world applications, we are not just interested in the calibration of the overall population, but also interested in the calibration for subgroups within the population. calzone provides a simple way to perform subgroup analysis given some data input format. In order to perform subgroup analysis, the input csv file should contain the following columns:

proba_0, proba_1, . . . , proba_n, subgroup_1, subgroup_2, . . . , subgroup_m, label

where n >= 1 and m >= 1.

In this example, we will use the example simulated dataset in the calzone package with only one subgroup field and two subgroups. See quickstart for more details.

[1]:
```
### import the packages and read the data
import numpy as np
from calzone.utils import data_loader
from calzone.metrics import CalibrationMetrics

dataset = data_loader('../../../example_data/simulated_data_subgroup.csv')
print("Whether the dataset has subgroup:",dataset.have_subgroup)

### Create the CalibrationMetrics class
metrics_cal = CalibrationMetrics(class_to_calculate=1)
```

```
Whether the dataset has subgroup: True
```

[2]:
```
### subgroup analysis for each group
### You can preform other analysis during the loop (eg. plotting the reliability diagram␣
↪etc)
for i,subgroup_column in enumerate(dataset.subgroup_indices):
    print(f"subgroup {i+1}")
    for j,subgroup_class in enumerate(dataset.subgroups_class[i]):
        print(f"subgroup {i+1} class {subgroup_class}")
        proba = dataset.probs[dataset.subgroups_index[i][j],:]
        label = dataset.labels[dataset.subgroups_index[i][j]]
        result = metrics_cal.calculate_metrics(label, proba,metrics='all')
        for metric in result:
            print(f"{metric}: {result[metric]}")
```

```
subgroup 1
subgroup 1 class A
SpiegelhalterZ score: 0.3763269161877356
SpiegelhalterZ p-value: 0.7066738713391099
ECE-H topclass: 0.009608653731328977
```

(continues on next page)

(continued from previous page)

```
ECE-H: 0.01208775955804901
MCE-H topclass: 0.03926468843081976
MCE-H: 0.04848338618970194
HL-H score: 8.884991559088098
HL-H p-value: 0.35209071874348785
ECE-C topclass: 0.009458033653818828
ECE-C: 0.008733966945443138
MCE-C topclass: 0.020515047600205505
MCE-C: 0.02324031223486256
HL-C score: 3.694947603203135
HL-C p-value: 0.8835446575708198
COX coef: 0.9942499557748269
COX intercept: -0.04497652296600376
COX coef lowerci: 0.9372902801721911
COX coef upperci: 1.0512096313774626
COX intercept lowerci: -0.12348577118577644
COX intercept upperci: 0.03353272525376893
COX ICI: 0.005610391483826338
Loess ICI: 0.00558856942568957
subgroup 1 class B
SpiegelhalterZ score: 27.93575342117766
SpiegelhalterZ p-value: 0.0
ECE-H topclass: 0.07658928982434714
ECE-H: 0.0765892898243467
MCE-H topclass: 0.1327565894838103
MCE-H: 0.16250572519432438
HL-H score: 910.4385762101924
HL-H p-value: 0.0
ECE-C topclass: 0.07429481165606829
ECE-C: 0.07479369479609524
MCE-C topclass: 0.14090872416947742
MCE-C: 0.14045600565696226
HL-C score: 2246.1714434139853
HL-C p-value: 0.0
COX coef: 0.5071793536874274
COX intercept: 0.00037947714112375366
COX coef lowerci: 0.47838663128188996
COX coef upperci: 0.5359720760929648
COX intercept lowerci: -0.07796623141885761
COX intercept upperci: 0.07872518570110512
COX ICI: 0.07746407648179383
Loess ICI: 0.06991428582761099
```

[3]:
```
### An alernative way to do the same thing is through command line interface

%run ../../../cal_metrics.py \
--csv_file '../../../example_data/simulated_data_subgroup.csv' \
--metrics all \
--class_to_calculate 1 \
--num_bins 10 \
--verbose
```

```
Metrics:
SpiegelhalterZ score: 18.327
SpiegelhalterZ p-value: 0.
ECE-H topclass: 0.042
ECE-H: 0.042
MCE-H topclass: 0.055
MCE-H: 0.063
HL-H score: 429.732
HL-H p-value: 0.
ECE-C topclass: 0.042
ECE-C: 0.038
MCE-C topclass: 0.065
MCE-C: 0.064
HL-C score: 1138.842
HL-C p-value: 0.
COX coef: 0.668
COX intercept: -0.02
COX coef lowerci: 0.641
COX coef upperci: 0.696
COX intercept lowerci: -0.074
COX intercept upperci: 0.034
COX ICI: 0.049
Loess ICI: 0.037
Metrics for subgroup subgroup_1_group_A:
SpiegelhalterZ score: 0.376
SpiegelhalterZ p-value: 0.707
ECE-H topclass: 0.01
ECE-H: 0.012
MCE-H topclass: 0.039
MCE-H: 0.048
HL-H score: 8.885
HL-H p-value: 0.352
ECE-C topclass: 0.009
ECE-C: 0.009
MCE-C topclass: 0.021
MCE-C: 0.023
HL-C score: 3.695
HL-C p-value: 0.884
COX coef: 0.994
COX intercept: -0.045
COX coef lowerci: 0.937
COX coef upperci: 1.051
COX intercept lowerci: -0.123
COX intercept upperci: 0.034
COX ICI: 0.006
Loess ICI: 0.006
Metrics for subgroup subgroup_1_group_B:
SpiegelhalterZ score: 27.936
SpiegelhalterZ p-value: 0.
ECE-H topclass: 0.077
ECE-H: 0.077
MCE-H topclass: 0.133
MCE-H: 0.163
```

(continues on next page)

(continued from previous page)

```
HL-H score: 910.439
HL-H p-value: 0.
ECE-C topclass: 0.074
ECE-C: 0.075
MCE-C topclass: 0.141
MCE-C: 0.140
HL-C score: 2246.171
HL-C p-value: 0.
COX coef: 0.507
COX intercept: 0.000
COX coef lowerci: 0.478
COX coef upperci: 0.536
COX intercept lowerci: -0.078
COX intercept upperci: 0.079
COX ICI: 0.077
Loess ICI: 0.07
```

[ ]:

CHAPTER

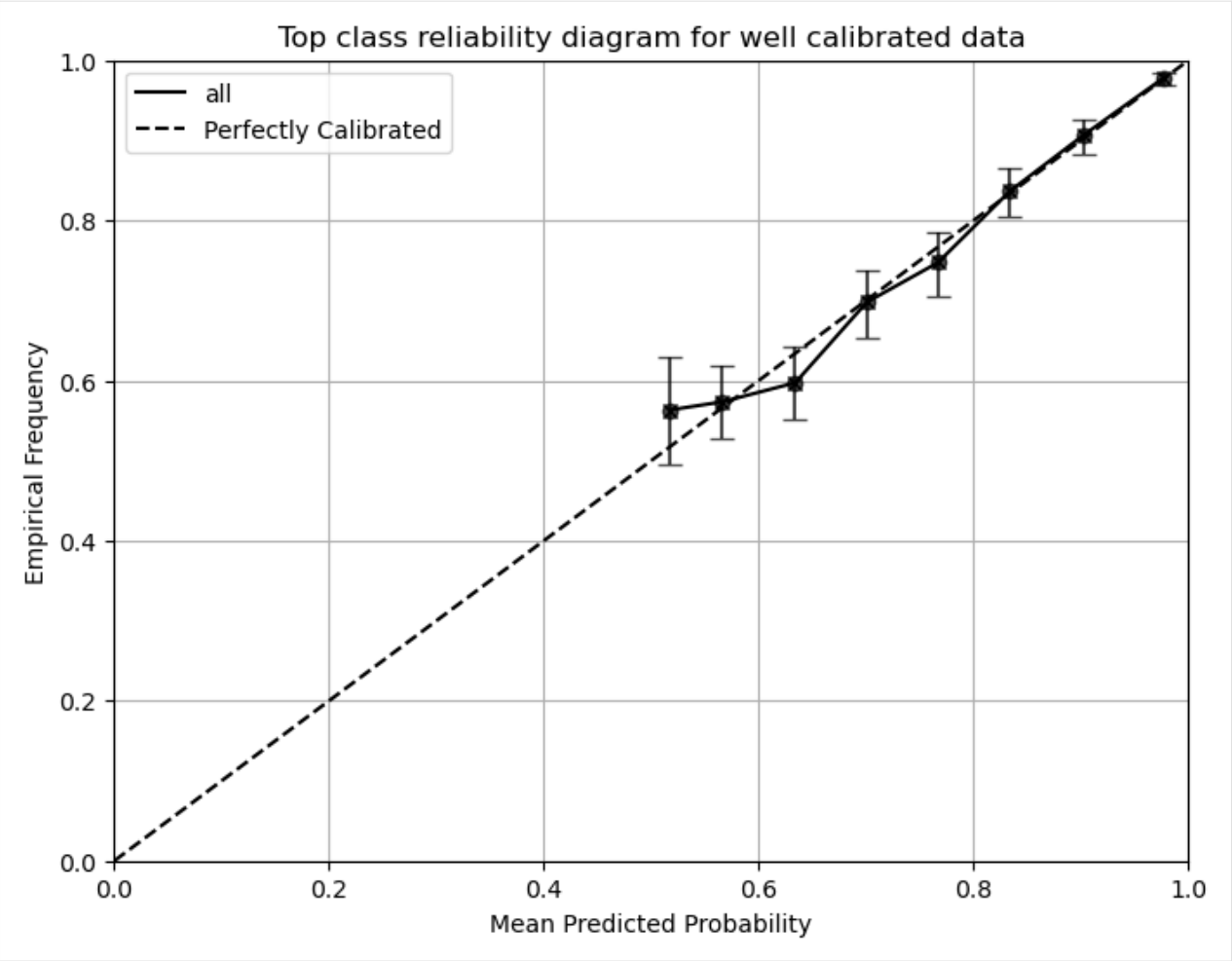
ELEVEN

MULTICLASS EXTENSION

In the previous notebooks, most metrics only consider binary classification except ECE_{top} and MCE_{top}. To handle multi-class classification, the most common way is to convert it to a series of binary classification problems by using 1-vs-rest approach. Alternativly, if the user only care about whether the predicted class of each sample is having the correct predicted probability, the user transform the data to a top-class binary classification problem. In top-class problem, the class 0 probability is the $1 - p_{top}$ and class 1 probability is p_{top} and the label is whether true label is equal to the top-class prediction. In `calzone`, user can transform the data using `data_loader.transform_topclass()` function. Most metrics will still works but the interpretation of the metrics will be different.

[3]:
```
from calzone.utils import data_loader,reliability_diagram
from calzone.vis import plot_reliability_diagram
from calzone.metrics import CalibrationMetrics

dataset = data_loader('../../../example_data/simulated_welldata.csv')
cal_metrics = CalibrationMetrics(class_to_calculate=1)
### Transform to top-class
transformed_data = dataset.transform_topclass()
```

[4]:
```
### Now the transformed data become a top-class problem when we specify the class-of-
↪interest to be 1
reliability,confindence,bin_edges,bin_counts = reliability_diagram(transformed_data.
↪labels, transformed_data.probs, num_bins=15,class_to_plot=1)
# Plot the reliability diagram
plot_reliability_diagram(reliability,confindence,bin_counts,error_bar=True,title='Top␣
↪class reliability diagram for well calibrated data')
```



[5]:
```
cal_metrics.calculate_metrics(transformed_data.labels, transformed_data.probs,metrics=
↪'all')
```

[5]:
```
{'SpiegelhalterZ score': 0.37632691618773545,
 'SpiegelhalterZ p-value': 0.7066738713391101,
 'ECE-H topclass': 0.009608653731328977,
 'ECE-H': 0.00960865373132937 2,
 'MCE-H topclass': 0.03926468843081976,
 'MCE-H': 0.03926468843081932,
 'HL-H score': 6.029652031234071,
 'HL-H p-value': 0.11017646753618837,
 'ECE-C topclass': 0.009458033653818828,
 'ECE-C': 0.009458033653818974,
 'MCE-C topclass': 0.020515047600205505,
 'MCE-C': 0.020515047600205394,
 'HL-C score': 5.0830845954424,
 'HL-C p-value': 0.7486601568004448,
 'COX coef': 0.9990721119864874,
 'COX intercept': -0.009313116424641145,
 'COX coef lowerci': 0.9097229915817346,
 'COX coef upperci': 1.0884212323912401,
```

(continues on next page)

(continued from previous page)

```
'COX intercept lowerci': -0.1322110153978738,
'COX intercept upperci': 0.1135847825485915,
'COX ICI': 0.0012819521292332472,
'Loess ICI': 0.0038637011857438034}
```

You can see the top-class ECE is the same as regular ECE.

CHAPTER

TWELVE

RUNNING GUI

GUI is currently under development and not all features from the command line are supported. To run GUI, you need to install nicegui using the following command:

```
[ ]: pip install nicegui
```

Then simply run the following command on calzone directory:

```
[ ]: python GUI_cal_metric.py
```

CHAPTER

THIRTEEN

CALZONE

13.1 calzone package

13.1.1 Submodules

13.1.2 calzone.metrics module

Metrics calculation functions for the Calibration Measure package.

class calzone.metrics.**CalibrationMetrics**(*class_to_calculate=1*, *num_bins=10*)

Bases: `object`

A class for calculating calibration metrics for classification models.

__init__(*class_to_calculate=1*, *num_bins=10*)

Initialize the CalibrationMetrics class.

Parameters

- **class_to_calculate** (*int, optional*) – The class index to calculate the metrics for. Defaults to 1.
- **num_bins** (*int, optional*) – Number of bins to use for the ECE/MCE/HL calculations. Defaults to 10.

bootstrap(*y_true*, *y_proba*, *metrics*, *perform_pervalance_adjustment=False*, *n_samples=1000*, ***kwargs*)

Run bootstrap and return a numpy structured array with correct field names.

This function performs bootstrap resampling to estimate the distribution of calibration metrics. It generates multiple samples with replacement from the input data and calculates the specified metrics for each sample.

Parameters

- **y_true** (*array-like*) – True labels.
- **y_proba** (*array-like*) – Predicted probabilities.
- **metrics** (*list of str*) – List of metric names to calculate.
- **perform_pervalance_adjustment** (*bool, optional*) – Whether to perform prevalence adjustment for each bootstrap sample. Defaults to False.
- **n_samples** (*int, optional*) – Number of bootstrap samples to generate. Defaults to 1000.
- ****kwargs** – Additional keyword arguments to pass to the metric calculation functions.

Returns

numpy.ndarray – A structured array containing the bootstrapped metrics. Each field in the array corresponds to a metric, and each row represents a bootstrap sample.

`calculate_metrics`(*y_true*, *y_proba*, *metrics*, *perform_pervalance_adjustment=False*, *return_numpy=False*, ***kwargs*)

Calculate the specified calibration metrics.

This function computes various calibration metrics for binary classification models. It supports multiple metrics and can perform prevalence adjustment if needed.

List of available metrics:

- SpiegelhalterZ: Spiegelhalter's Z-test for calibration
- ECE-H: Expected Calibration Error with equal-space binning
- MCE-H: Maximum Calibration Error with equal-space binning
- HL-H: Hosmer-Lemeshow test with equal-space binning
- ECE-C: Expected Calibration Error with equal-count binning
- MCE-C: Maximum Calibration Error with equal-count binning
- HL-C: Hosmer-Lemeshow test with equal-count binning
- COX: Cox regression analysis
- Loess: Locally Estimated Scatterplot Smoothing regression analysis

Parameters

- **y_true** (*numpy.ndarray*) – True labels.
- **y_proba** (*numpy.ndarray*) – Predicted probabilities.
- **metrics** (*list of str or 'all'*) – List of metric names to calculate. If 'all', calculates all available metrics.
- **perform_pervalance_adjustment** (*bool, optional*) – Whether to perform prevalence adjustment. Defaults to False.
- **return_numpy** (*bool, optional*) – Whether to return the results as a numpy array. Defaults to False.
- ****kwargs** – Additional keyword arguments to pass to the metric calculation functions.

Returns

dict or numpy.ndarray – A dictionary containing the calculated metrics, or a numpy array if return_numpy is True.

`optimal_prevalence_adjustment`(*y_true*, *y_proba*)

Perform optimal prevalence adjustment and return adjusted probabilities.

This function finds the optimal prevalence value that minimizes the difference between the predicted and actual positive rates, and then adjusts the input probabilities accordingly.

Parameters

- **y_true** (*array-like*) – True labels.
- **y_proba** (*array-like*) – Predicted probabilities.

Returns

- **optimal_prevalence** (*float*) – Optimal prevalence value.
- **adjusted_proba** (*array-like*) – Adjusted probabilities. First column is the adjusted probabilities for the other class, second column is the adjusted probabilities for the class of interest.

calzone.metrics.**cal_ICI**(*func*, *y_proba*, *points=1000*, *class_to_calculate=1*)

Calculate the Integrated Calibration Index (ICI) for a given calibration function.

Parameters

- **func** (*callable*) – The calibration function to evaluate.
- **y_proba** (*array-like*) – Predicted probabilities for each class. Shape (n_samples, n_classes).
- **points** (*int, optional*) – Number of points to use for numerical integration. Defaults to 1000.
- **class_to_calculate** (*int, optional*) – The class index to calculate the ICI for. Defaults to 1.

Returns

float – The Integrated Calibration Index (ICI) value.

Note

The ICI is calculated by integrating the absolute difference between the calibration function and the identity function, weighted by the density of the predicted probabilities.

calzone.metrics.**cal_ICI_cox**(*coef*, *intercept*, *y_proba*, *class_to_calculate=1*, *epsilon=1e-07*, ***kwargs*)

Calculate the Integrated Calibration Index (ICI) for a given Cox regression model.

The ICI measures the average absolute difference between the predicted probabilities and the probabilities transformed by the fitted Cox regression model.

Parameters

- **coef** (*float*) – The coefficient (slope) from the Cox regression.
- **intercept** (*float*) – The intercept from the Cox regression.
- **y_proba** (*array-like*) – Predicted probabilities. Should be of shape (n_samples, n_classes).
- **class_to_calculate** (*int*) – The class to calculate the ICI for in multi-class problems. Default is 1.
- **epsilon** (*float*) – Small value to avoid numerical instability when clipping probabilities. Default is 1e-7.

Returns

ICI (*float*) – The Integrated Calibration Index.

Note

- Lower ICI values indicate better calibration.
- The function applies the inverse logit transformation to the predicted probabilities using the coefficients from the Cox regression.

calzone.metrics.**calculate_ece_mce**(*reliability*, *confindence*, *bin_counts*)

Calculate Expected Calibration Error (ECE) and Maximum Calibration Error (MCE).

These metrics assess the calibration of a classification model by comparing predicted probabilities to observed frequencies.

Parameters

- **reliability** (*array-like*) – Array of observed frequencies for each bin.
- **confindence** (*array-like*) – Array of predicted probabilities for each bin.
- **bin_counts** (*array-like*) – Array of sample counts in each bin.

Returns

- **ece** (*float*) – The Expected Calibration Error.
- **mce** (*float*) – The Maximum Calibration Error.

Note

- ECE is a weighted average of the absolute differences between confidence and reliability.
- MCE is the maximum absolute difference between confidence and reliability across all bins.
- Lower values of both ECE and MCE indicate better calibration.

calzone.metrics.**cox_regression_analysis**(*y_true*, *y_proba*, *epsilon=1e-07*, *class_to_calculate=1*, *print_results=False*, *fix_intercept=False*, *fix_slope=False*, ***kwargs*)

Perform Cox regression analysis for classification calibration.

This function fits a logistic regression model to the logit of predicted probabilities to assess the calibration of classification predictions.

Parameters

- **y_true** (*array-like*) – True binary labels. If multi-class, will be converted to binary.
- **y_proba** (*array-like*) – Predicted probabilities. Should be of shape (n_samples, n_classes).
- **epsilon** (*float*) – Small value to avoid log(0) errors. Default is 1e-7.
- **class_to_calculate** (*int*) – The class to treat as the positive class in binary classification. Default is 1.
- **print_results** (*bool*) – If True, prints the summary of the logistic regression results. Default is False.
- **fix_intercept** (*bool*) – If True, fixes the intercept to 0. Can't be used with fix_slope. Default is False.
- **fix_slope** (*bool*) – If True, fixes the coefficient to 1. Can't be used with fix_intercept. Default is False.

Returns

- **coef** (*float*) – The coefficient (slope) of the Cox regression.

- **intercept** (*float*) – The intercept of the Cox regression.
- **coef_ci** (*tuple*) – The confidence interval for the coefficient.
- **intercept_ci** (*tuple*) – The confidence interval for the intercept.

Note

- A well-calibrated model should have a coefficient close to 1 and an intercept close to 0.
- The function clips probabilities to avoid numerical instability.
- For multi-class problems, the function converts the problem to binary classification based on the specified class_to_calculate.

`calzone.metrics.get_CI(result, alpha=0.05)`

Calculate confidence intervals for each field in the result.

Parameters

- **result** (*numpy.ndarray*) – Structured array containing the results for which to calculate confidence intervals.
- **alpha** (*float, optional*) – The significance level for the confidence interval calculation. Defaults to 0.05.

Returns

dict – A dictionary where keys are field names from the input array and values are tuples containing the lower and upper bounds of the confidence interval.

Note

This function calculates percentile-based confidence intervals for each field in the input structured array. It's useful for bootstrap or Monte Carlo simulations.

`calzone.metrics.hosmer_lemeshow_test(reliability, confidence, bin_count, df=None, **kwargs)`

Compute the Hosmer-Lemeshow test for goodness of fit.

This test is used to assess the calibration of binary classification models with full probability outputs. It compares observed and expected frequencies of events in groups of the data.

Parameters

- **reliability** (*array-like*) – Observed proportion of positive samples in each bin.
- **confidence** (*array-like*) – Predicted probabilities for each bin.
- **bin_count** (*array-like*) – Number of samples in each bin.
- **df** (*int, optional*) – Degrees of freedom for the test. Defaults is nbins - 2.

Returns

- **chi_squared** (*float*) – The chi-squared statistic of the Hosmer-Lemeshow test.
- **p_value** (*float*) – The p-value associated with the chi-squared statistic.
- **df** (*int*) – The degrees of freedom for the test.

> **Note**
>
> - The Hosmer-Lemeshow test is widely used for assessing calibration in probabiliticst models.
> - A small p-value (typically < 0.05) suggests that the model is a poor fit to the data.
> - This test can be sensitive to the number of groups and sample size.
> - It is recommended to use the Hosmer-Lemeshow test in conjunction with other metrics.

calzone.metrics.**logit_func**(*coef*, *intercept*)

Create a logistic function with given coefficient and intercept.

Parameters

- **coef** (*float*) – The coefficient (slope) of the logistic function.
- **intercept** (*float*) – The intercept of the logistic function.

Returns

callable – A function that takes an input x and returns the logistic function value.

> **Note**
>
> The returned function applies the logistic transformation: f(x) = 1 / (1 + exp(-(coef * log(x / (1 - x)) + intercept)))

calzone.metrics.**lowess_regression_analysis**(*y_true*, *y_proba*, *epsilon=1e-07*, *class_to_calculate=1*, *span=0.5*, *delta=0.001*, *it=0*, ***kwargs*)

Perform Lowess regression analysis for classification calibration.

This function applies Locally Weighted Scatterplot Smoothing (LOWESS) to assess the calibration of classification predictions.

Parameters

- **y_true** (*array-like*) – True binary labels. If multi-class, will be converted to binary.
- **y_proba** (*array-like*) – Predicted probabilities. Should be of shape (n_samples, n_classes).
- **epsilon** (*float, optional*) – Small value to avoid numerical instability when clipping probabilities. Defaults to 1e-10.
- **class_to_calculate** (*int, optional*) – The class to treat as the positive class in binary classification. Defaults to 1.
- **span** (*float, optional*) – The fraction of the data used when estimating each y-value. Defaults to 0.5.
- **delta** (*float, optional*) – Distance within which to use linear-interpolation instead of weighted regression. Defaults to 0.001.
- **it** (*int, optional*) – The number of residual-based reweightings to perform. Defaults to 0.

Returns

- **ICI** (*float*) – The Integrated Calibration Index.

- **sorted_proba** (*array-like*) – Sorted predicted probabilities.
- **smoothed_proba** (*array-like*) – Corresponding LOWESS-smoothed actual probabilities.

Note

- The function clips probabilities to avoid numerical instability.
- For multi-class problems, the function converts the problem to binary classification based on the specified class_to_calculate.
- The Integrated Calibration Index (ICI) provides a measure of calibration error, with lower values indicating better calibration.

`calzone.metrics.`**`spiegelhalter_z_test`**(*y_true*, *y_proba*, *class_to_calculate=1*)

Perform Spiegelhalter's Z-test for calibration of probabilistic predictions.

This test assesses whether predicted probabilities are well-calibrated by comparing them to observed outcomes.

Parameters

- **y_true** (*array-like*) – True labels of the samples.
- **y_proba** (*array-like*) – Predicted probabilities for each class. Shape should be (n_samples, n_classes).
- **class_to_calculate** (*int*) – Index of the class to calculate the test for. Default is 1.

Returns

- **z_score** (*float*) – The z-score of the Spiegelhalter's Z-test.
- **p_value** (*float*) – The p-value associated with the z-score.

Note

- This test is used to assess the calibration of a classification model.
- A small p-value (typically < 0.05) suggests that the model is poorly calibrated.
- The test assumes that predictions are independent and identically distributed.

13.1.3 calzone.utils module

Uitlity functions for the Calibration Measure package.

`calzone.utils.`**`apply_prevalence_adjustment`**(*adjusted_prevalence*, *y_true*, *y_proba*, *class_to_calculate=1*)

Apply the prevalence adjustment method.

Parameters

- **adjusted_prevalence** (*float*) – The adjusted prevalence to test.
- **y_true** (*array-like*) – True labels.
- **y_proba** (*array-like*) – Predicted probabilities.
- **class_to_calculate** (*int*) – The class index to adjust. Default is 1.

Returns
numpy.ndarray – Adjusted probabilities.

class calzone.utils.**data_loader**(*data_path*)

Bases: object

A class for loading and preprocessing data from a CSV file.

This class handles various data formats, including those with or without subgroup information and headers.

data_path

Path to the CSV file containing the data.

Type
str

Header

Array of column headers from the CSV file.

Type
numpy.ndarray

subgroups

List of subgroup column names, if present.

Type
list

subgroup_indices

List of indices for subgroup columns, if present.

Type
list

have_subgroup

Flag indicating whether subgroup information is present.

Type
bool

data

Raw data loaded from the CSV file.

Type
numpy.ndarray

probs

Probability values extracted from the data.

Type
numpy.ndarray

labels

Label values extracted from the data.

Type
numpy.ndarray

subgroups_class

List of unique subgroup classes for each subgroup, if present.

Type
list

subgroups_index

List of indices for each subgroup class, if present.

Type
list

__init__(*self*, *data_path*)

Initializes the data_loader object and loads data from a CSV file.

transform_topclass(*self*)

Transforms the data to top class binary problem.

__init__(*data_path*)

Initializes the data_loader object and loads data from the specified file.

Parameters
data_path (*str*) – Path to the CSV file containing the data.

The method performs the following steps: 1. Loads the header from the CSV file. 2. Checks for the presence of subgroup information. 3. Loads the data based on the presence or absence of subgroup information. 4. Extracts probability values and labels from the loaded data. 5. If subgroups are present, extracts subgroup classes and their indices.

Note: - If there is a header, it must be in the format: proba_0,proba_1,...,subgroup_1(optional),subgroup_2(optional),...,label - If there is no header, the columns must be in the order of proba_0,proba_1,...,label

transform_topclass()

Transforms the data to top class binary problem

Returns
data_loader – A new data_loader object with transformed data

class calzone.utils.**fake_binary_data_generator**(*alpha_val*, *beta_val*)

Bases: object

A class for generating fake binary data and applying miscalibration.

This class provides methods to generate binary classification data and apply different types of miscalibration to the probabilities.

alpha_val

Alpha parameter for the beta distribution.

Type
float

beta_val

Beta parameter for the beta distribution.

Type
float

__init__(*alpha_val*, *beta_val*)

Initialize the fake binary data generator.

Parameters

- **alpha_val** (*float*) – Alpha parameter for the beta distribution.

- **beta_val** (*float*) – Beta parameter for the beta distribution.

abraitary_miscal(*logits*, *miscal_function*)

Apply arbitrary miscalibration to the input logits.

This function allows for the application of any custom miscalibration function to the input logits.

Parameters

- **logits** (*numpy.ndarray*) – Input logits of shape (n_samples, 2).
- **miscal_function** (*callable*) – Function to apply miscalibration to the logits.

Returns

numpy.ndarray – Miscalibrated probabilities of shape (n_samples, 2).

generate_data(*sample_size*)

Generate fake binary classification data.

Parameters

sample_size (*int*) – Number of samples to generate.

Returns

- **X** (*numpy.ndarray*) – Array of shape (sample_size, 2) containing probabilities for each class.
- **y_true** (*numpy.ndarray*) – Array of shape (sample_size,) containing true binary labels.

linear_miscal(*X*, *miscal_scale*)

Apply linear miscalibration to the input probabilities.

This function transforms the input probabilities to logits, applies a linear scaling, and then converts back to probabilities.

Parameters

- **X** (*numpy.ndarray*) – Input probabilities of shape (n_samples, 2).
- **miscal_scale** (*float*) – Scale factor for miscalibration.

Returns

numpy.ndarray – Miscalibrated probabilities of shape (n_samples, 2).

calzone.utils.**find_optimal_prevalence**(*y_true*, *y_proba*, *class_to_calculate=1*, *epsilon=1e-07*)

Find the optimal adjustment prevalence using scipy.optimize.

Parameters

- **y_true** (*array-like*) – True labels.
- **y_proba** (*array-like*) – Predicted probabilities.
- **class_to_calculate** (*int*) – The class index to optimize for. Default is 1.
- **epsilon** (*float*) – Small value to avoid numerical instability. Default is 1e-7.

Returns

- **optimal_prevalence** (*float*) – The optimal prevalence.
- **adjusted_probabilities** (*numpy.ndarray*) – The adjusted probabilities using the optimal prevalence.

calzone.utils.**loss**(*adjusted_prevalence*, *y_true*, *y_proba*, *class_to_calculate=1*)

Calculate the loss function for prevalence adjustment.

Parameters

- **adjusted_prevalence** (*float*) – The adjusted prevalence.
- **y_true** (*array-like*) – True labels.
- **y_proba** (*array-like*) – Predicted probabilities.
- **class_to_calculate** (*int*) – The class index to calculate loss for. Default is 1.

Returns

float – Calculated loss value.

calzone.utils.**make_roc_curve**(*y_true*, *y_proba*, *class_to_plot=None*)

Compute the Receiver Operating Characteristic (ROC) curve for binary or multiclass classification.

Parameters

- **y_true** (*array-like*) – True labels of the data. Shape (n_samples,).
- **y_proba** (*array-like*) – Predicted probabilities of the positive class. Shape (n_samples, n_classes).
- **class_to_plot** (*int, optional*) – The class to plot the ROC curve for. If None, plots the ROC curve for each class. Default is None.

Returns

- **fpr** (*array*) – False Positive Rate for the selected class or each class. Shape (n_points,).
- **tpr** (*array*) – True Positive Rate for the selected class or each class. Shape (n_points,).
- **roc_auc** (*float or array*) – Area Under the ROC Curve (AUC) for the selected class or each class. If class_to_plot is not None, returns a float. If class_to_plot is None, returns an array of shape (n_classes,).

Note

- The input arrays y_true and y_proba must have the same number of samples.
- The input array y_proba must have probabilities for each class in a multiclass problem.
- The input array y_proba must not contain any NaN values.

Example

```
>>> y_true = [0, 1, 1, 0, 1]
>>> y_proba = [[0.2, 0.8], [0.6, 0.4], [0.3, 0.7], [0.9, 0.1], [0.4, 0.6]]
>>> fpr, tpr, roc_auc = roc_curve(y_true, y_proba, class_to_plot=1)
```

calzone.utils.**reliability_diagram**(*y_true*, *y_proba*, *num_bins=10*, *class_to_plot=None*, *is_equal_freq=False*, *save_path=None*)

Compute the reliability diagram for a binary or multi-class classification model.

Parameters

- **y_true** (*array-like*) – True labels of the samples. Can be a binary array or a one-hot encoded array.
- **y_proba** (*array-like*) – Predicted probabilities for each class. Shape should be (n_samples, n_classes).
- **num_bins** (*int*) – Number of bins to divide the predicted probabilities into. Default is 10.
- **class_to_plot** (*int or None*) – Index of the class to plot the reliability diagram for. If None, the diagram will be computed for all classes. Default is None.
- **is_equal_freq** (*bool*) – If True, the bins will be equally frequent. If False, the bins will be equally spaced. Default is False.

Returns

- **reliabilities** (*array-like*) – Array of accuracies for each bin. Shape depends on the value of class_to_plot.
- **confidences** (*array-like*) – Array of average confidences for each bin. Shape depends on the value of class_to_plot.
- **bin_edges** (*array-like*) – Array of bin edges.
- **bin_counts** (*array-like*) – Array of counts for each bin.

Note

- The reliability diagram is a graphical tool to assess the calibration of a classification model. It plots the average predicted probability against the observed accuracy for each bin of predicted probabilities.
- If y_true is a binary array, it will be converted to a one-hot encoded array internally.
- If class_to_plot is not None, the reliability diagram will be computed only for the specified class. Otherwise, it will be computed for all classes.
- The number of bins determines the granularity of the reliability diagram. Higher values result in more bins and a more detailed diagram.

`calzone.utils.removing_nan(y_true, y_predict, y_proba)`

Remove rows containing NaN values from input arrays.

Parameters

- **y_true** (*array-like*) – True labels.
- **y_predict** (*array-like*) – Predicted labels.
- **y_proba** (*array-like*) – Predicted probabilities.

Returns

- **y_true** (*array-like*) – Cleaned version of y_true with NaN rows removed.
- **y_predict** (*array-like*) – Cleaned version of y_predict with NaN rows removed.
- **y_proba** (*array-like*) – Cleaned version of y_proba with NaN rows removed.

`calzone.utils.softmax_to_logits(probabilities, epsilon=1e-07)`

Convert softmax probabilities to logits.

Parameters

- **probabilities** (*array-like*) – Input probabilities.
- **epsilon** (*float*) – Small value to avoid log(0). Default is 1e-7.

Returns

numpy.ndarray – Computed logits.

13.1.4 calzone.vis module

Visualization functions for the Calibration Measure package.

calzone.vis.**plot_reliability_diagram**(*reliabilities*, *confidences*, *bin_counts*, *bin_edges=None*, *line=True*, *error_bar=False*, *z=1.96*, *title='Reliability Diagram'*, *save_path=None*, *return_fig=False*, *custom_colors=None*, *dpi=150*)

Plot a reliability diagram to visualize the calibration of a model.

Parameters

- **reliabilities** (*array-like*) – Empirical frequencies for each bin.
- **confidences** (*array-like*) – Mean predicted probabilities for each bin.
- **bin_counts** (*array-like*) – Number of samples in each bin.
- **bin_edges** (*array-like, optional*) – Edges of the bins. If None, assumes equal-spaced bins.
- **line** (*bool, optional*) – If True, plot lines connecting points. If False, plot as a bar chart. Defaults to True.
- **error_bar** (*bool, optional*) – If True, add error bars to the plot. Defaults to False.
- **z** (*float, optional*) – Z-score for calculating Wilson score interval. Defaults to 1.96.
- **title** (*str, optional*) – Title of the plot. Defaults to 'Reliability Diagram'.
- **save_path** (*str, optional*) – Path to save the figure. If None, figure is not saved. Defaults to None.
- **return_fig** (*bool, optional*) – If True, return the figure object. Defaults to False.
- **custom_colors** (*list, optional*) – List of custom colors for multi-class plots. Defaults to None.
- **dpi** (*int, optional*) – DPI for saving the figure. Defaults to 150.

Returns

matplotlib.figure.Figure, optional – The figure object if return_fig is True.

calzone.vis.**plot_roc_curve**(*fpr*, *tpr*, *roc_auc*, *class_to_plot=None*, *title='ROC Curve'*, *save_path=None*, *dpi=150*, *return_fig=False*)

Plots the Receiver Operating Characteristic (ROC) curve.

Parameters

- **fpr** (*array-like*) – False Positive Rate values.
- **tpr** (*array-like*) – True Positive Rate values.
- **roc_auc** (*float or array-like*) – Area Under the ROC Curve (AUC) value(s).
- **class_to_plot** (*int, optional*) – The class to plot. If None, plots all classes. Defaults to None.

- **title** (*str, optional*) – Title of the plot. Defaults to 'ROC Curve'.
- **save_path** (*str, optional*) – Path to save the figure. If None, the figure is not saved. Defaults to None.
- **dpi** (*int, optional*) – The resolution in dots per inch for saving the figure. Defaults to 150.
- **return_fig** (*bool, optional*) – If True, returns the figure object instead of displaying it. Defaults to False.

Returns

matplotlib.figure.Figure or None – The figure object if return_fig is True, otherwise None.

This function creates a matplotlib figure showing the ROC curve(s).

13.1.5 Module contents

PYTHON MODULE INDEX

C

INDEX

O

P

R

S

T