ML Paper Documentation

ML Paper Team

Jul 24, 2020

CONTENTS

Contents:

CHAPTER

ONE

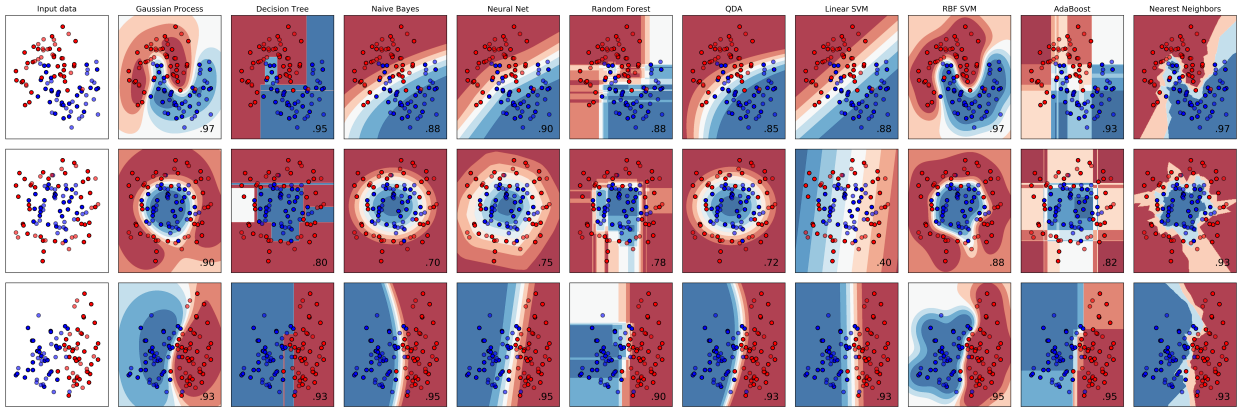
THE ML PAPER PACKAGE (MLPAPER)

Easy benchmarking of machine learning models with sklearn interface with statistical tests built-in.

1.1 Usage for classification problems

First, we consider the `plot_classifier_comparison.py` demo file. This extends the standard sklearn classifier comparison but also demos the ease of *mlpaper* to create a performance report.

In this demo, we use the example of the three toy data sets and ten classifiers from the sklearn example:



The *mlpaper* package can benchmark all of the of these methods and created a properly formatted LaTeX table (with error bars) in a few commands. This generates a results table for copy-and-paste into a ML paper *.tex* file in a few commands.

Pandas tables with the performance results of all the methods can be built by:

```
import mlpaper.classification as btc
from mlpaper.classification import STD_BINARY_CURVES, STD_CLASS_LOSS

performance_df, performance_curves_dict = btc.just_benchmark(
    X_train,
    y_train,
    X_test,
    y_test,
    2,
    classifiers,
    STD_CLASS_LOSS,
    STD_BINARY_CURVES,
```

(continues on next page)

(continued from previous page)

```
    ref_method,
)
```

This benchmarks all the models in classifiers on the data (`X_train`, `y_train`, `X_test`, `y_test`) for 2-class classification. It uses the loss function described in the dictionaries `STD_CLASS_LOSS`, and the curves (e.g., ROC, PR) in `STD_BINARY_CURVES`. The `ref_method` defines the model that is the reference to compare against for assessing statistically significant performance gains.

The *sciprint* module formats these tables for scientific presentation. The performance dictionaries can be converted to cleanly formatted tables: correct significant figures, shifting of exponent for compactness, thresholding huge/small (crap limit) results, and correct alignment of decimal points, units in headers, etc. Here we use:

```
import mlpaper.sciprint as sp

print(
    sp.just_format_it(
        performance_df,
        shift_mod=3,
        unit_dict={"NLL": "nats"},
        crap_limit_min={"AUPRG": -1},
        EB_limit={"AUPRG": -1},
        non_finite_fmt={sp.NAN_STR: "N/A"},
        use_tex=False,
    )
)
```

to export the results in plain text, or for LaTeX we use:

```
import mlpaper.sciprint as sp

print(
    sp.just_format_it(
        performance_df,
        shift_mod=3,
        unit_dict={"NLL": "nats"},
        crap_limit_min={"AUPRG": -1},
        EB_limit={"AUPRG": -1},
        non_finite_fmt={sp.NAN_STR: "{--}"},
        use_tex=True,
    )
)
```

1.1.1 Output

Dataset 0 (Moons)

```
                          AP         p         AUC         p    AUPRG         p␣
↪Brier        p NLL (nats)         p      sphere         p   zero one         p
AdaBoost            0.93(16)    <0.0001   0.950(96)   <0.0001   0.90464   <0.0001   0.42(14)␣
↪   <0.0001   0.368(80)   <0.0001   0.36(15)    <0.0001   0.075(86)   <0.0001
Decision Tree       0.95(13)    <0.0001   0.966(70)   <0.0001   0.93860   <0.0001   0.18(25)␣
↪   <0.0001   0.40(71)     0.4072   0.16(22)    <0.0001   0.050(71)   <0.0001
Gaussian Process    0.90(22)    <0.0001   0.95(12)    <0.0001   0.92081   <0.0001   0.27(17)␣
↪   <0.0001   0.27(11)    <0.0001   0.22(16)    <0.0001   0.025(51)   <0.0001
```

(continues on next page)

(continued from previous page)

```
Linear SVM          0.952(99)  <0.0001  0.950(77)  <0.0001  0.88705  <0.0001  0.34(24)␣
↪  <0.0001  0.29(16)    <0.0001  0.31(24)    <0.0001  0.15(12)     0.0006
Naive Bayes         0.957(97)  <0.0001  0.957(68)  <0.0001  0.89782  <0.0001  0.34(25)␣
↪  <0.0001  0.28(18)    <0.0001  0.31(24)    <0.0001  0.13(11)     0.0002
Nearest Neighbors   0.94(14)   <0.0001  0.969(69)  <0.0001  0.93498  <0.0001  0.18(21)␣
↪  <0.0001  0.42(70)     0.4241  0.15(18)    <0.0001  0.025(51)   <0.0001
Neural Net          0.957(91)  <0.0001  0.957(69)  <0.0001  0.89782  <0.0001  0.33(23)␣
↪  <0.0001  0.28(15)    <0.0001  0.30(22)    <0.0001  0.100(98)   <0.0001
QDA                 0.951(91)  <0.0001  0.950(80)  <0.0001  0.88517  <0.0001  0.34(27)␣
↪  <0.0001  0.29(21)     0.0003  0.31(25)    <0.0001  0.15(12)     0.0006
RBF SVM             0.93(18)   <0.0001  0.957(94)  <0.0001  0.92081  <0.0001  0.14(20)␣
↪  <0.0001  0.18(18)    <0.0001  0.12(17)    <0.0001  0.025(51)   <0.0001
Random Forest       0.965(82)  <0.0001  0.949(84)  <0.0001  0.92147  <0.0001  0.31(26)␣
↪  <0.0001  0.52(70)     0.6099  0.28(24)    <0.0001  0.100(98)   <0.0001
iid                 0.53(16)        N/A  0.5(0)          N/A  0(0)          N/A  1.
↪004(22)       N/A  0.695(11)       N/A  1.005(27)       N/A  0.53(17)        N/A
```

Dataset 0 (Moons) in LaTeX

```
\begin{tabular}{|l|Sr|Sr|Sr|Sr|Sr|Sr|}
\toprule
{}                &       {AP} &      {p} &      {AUC} &      {p} &  {AUPRG} &
↪{p} &    {Brier} &      {p} & {NLL (nats)} &      {p} &   {sphere} &      {p} &
↪{zero one} &      {p} \\
\midrule
AdaBoost          &  0.93(16)  &  <0.0001 &  0.950(96) &  <0.0001 &  0.90464 &  <0.
↪0001 &  0.42(14)  &  <0.0001 &    0.368(80) &  <0.0001 &  0.36(15)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
Decision Tree     &  0.95(13)  &  <0.0001 &  0.966(70) &  <0.0001 &  0.93860 &  <0.
↪0001 &  0.18(25)  &  <0.0001 &    0.40(71)  &   0.4072 &  0.16(22)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
Gaussian Process  &  0.90(22)  &  <0.0001 &  0.95(12)  &  <0.0001 &  0.92081 &  <0.
↪0001 &  0.27(17)  &  <0.0001 &    0.27(11)  &  <0.0001 &  0.22(16)  &  <0.0001 &  0.
↪025(51) &  <0.0001 \\
Linear SVM        &  0.952(99) &  <0.0001 &  0.950(77) &  <0.0001 &  0.88705 &  <0.
↪0001 &  0.34(24)  &  <0.0001 &    0.29(16)  &  <0.0001 &  0.31(24)  &  <0.0001 &  0.
↪15(12)  &   0.0006 \\
Naive Bayes       &  0.957(97) &  <0.0001 &  0.957(68) &  <0.0001 &  0.89782 &  <0.
↪0001 &  0.34(25)  &  <0.0001 &    0.28(18)  &  <0.0001 &  0.31(24)  &  <0.0001 &  0.
↪13(11)  &   0.0002 \\
Nearest Neighbors &  0.94(14)  &  <0.0001 &  0.969(69) &  <0.0001 &  0.93498 &  <0.
↪0001 &  0.18(21)  &  <0.0001 &    0.42(70)  &   0.4241 &  0.15(18)  &  <0.0001 &  0.
↪025(51) &  <0.0001 \\
Neural Net        &  0.957(91) &  <0.0001 &  0.957(69) &  <0.0001 &  0.89782 &  <0.
↪0001 &  0.33(23)  &  <0.0001 &    0.28(15)  &  <0.0001 &  0.30(22)  &  <0.0001 &  0.
↪100(98) &  <0.0001 \\
QDA               &  0.951(91) &  <0.0001 &  0.950(80) &  <0.0001 &  0.88517 &  <0.
↪0001 &  0.34(27)  &  <0.0001 &    0.29(21)  &   0.0003 &  0.31(25)  &  <0.0001 &  0.
↪15(12)  &   0.0006 \\
RBF SVM           &  0.93(18)  &  <0.0001 &  0.957(94) &  <0.0001 &  0.92081 &  <0.
↪0001 &  0.14(20)  &  <0.0001 &    0.18(18)  &  <0.0001 &  0.12(17)  &  <0.0001 &  0.
↪025(51) &  <0.0001 \\
Random Forest     &  0.965(82) &  <0.0001 &  0.949(84) &  <0.0001 &  0.92147 &  <0.
↪0001 &  0.31(26)  &  <0.0001 &    0.52(70)  &   0.6099 &  0.28(24)  &  <0.0001 &  0.
↪100(98) &  <0.0001 \\
```

(continues on next page)

(continued from previous page)

```
iid                &  0.53(16)  &      {--} &  0.5(0)    &      {--} &  0(0)    &      {--
↪} &  1.004(22) &      {--} &    0.695(11) &      {--} &  1.005(27) &      {--} &  0.
↪53(17)  &      {--} \\
\bottomrule
\end{tabular}
```

Dataset 1 (Circles)

```
                          AP         p         AUC         p      AUPRG         p
↪Brier        p NLL (nats)         p     sphere         p   zero one         p
AdaBoost            0.938(82)   <0.0001  0.89(12)    <0.0001  0.76091    <0.0001  0.
↪773(96)  <0.0001  0.576(50)   <0.0001  0.73(12)    <0.0001  0.17(13)    <0.0001
Decision Tree       0.86(16)    <0.0001  0.80(13)    <0.0001  0.76316    <0.0001  0.
↪80(52)    0.3009  2.8(18)      0.0270  0.68(45)     0.0792  0.20(13)     0.0003
Gaussian Process    0.977(47)   <0.0001  0.964(60)   <0.0001  0.93049    <0.0001  0.
↪39(23)   <0.0001  0.33(14)    <0.0001  0.36(23)    <0.0001  0.100(98)  <0.0001
Linear SVM          0.53(18)     0.1621  0.51(21)     0.8580  0.19756     0.3660  1.
↪066(80)   0.1521  0.726(41)    0.1514  1.079(96)    0.1531  0.60(16)    1.0000
Naive Bayes         0.9983(82)  <0.0001  0.997(13)   <0.0001  0.996(21)  <0.0001  0.
↪64(20)   <0.0001  0.48(12)    <0.0001  0.63(21)    <0.0001  0.30(15)    0.0003
Nearest Neighbors   0.996(15)   <0.0001  0.966(49)   <0.0001  0.991(47)  <0.0001  0.
↪30(16)   <0.0001  0.23(11)    <0.0001  0.28(16)    <0.0001  0.075(86)  <0.0001
Neural Net          0.993(23)   <0.0001  0.990(32)   <0.0001  0.982(79)  <0.0001  0.
↪69(14)   <0.0001  0.525(74)   <0.0001  0.65(16)    <0.0001  0.25(15)   <0.0001
QDA                 0.9983(83)  <0.0001  0.997(11)   <0.0001  0.996(32)  <0.0001  0.
↪63(19)   <0.0001  0.47(11)    <0.0001  0.61(20)    <0.0001  0.28(15)   <0.0001
RBF SVM             0.979(44)   <0.0001  0.966(63)   <0.0001  0.93680    <0.0001  0.
↪34(22)   <0.0001  0.29(14)    <0.0001  0.31(22)    <0.0001  0.100(98)  <0.0001
Random Forest       0.90(13)    <0.0001  0.85(16)    <0.0001  0.64512     0.0021  0.
↪65(30)    0.0070  0.48(19)     0.0094  0.62(31)     0.0047  0.23(14)    0.0006
iid                 0.60(16)       N/A   0.5(0)         N/A  0(0)           N/A  1.
↪071(85)       N/A  0.729(43)       N/A  1.08(11)       N/A  0.60(16)       N/A
```

Dataset 1 (Circles) in LaTeX

```
\begin{tabular}{|l|Sr|Sr|Sr|Sr|Sr|Sr|}
\toprule
{}                 &        {AP} &      {p} &      {AUC} &      {p} &    {AUPRG} &
↪ {p} &    {Brier} &      {p} & {NLL (nats)} &      {p} &   {sphere} &      {p} &
↪{zero one} &      {p} \\
\midrule
AdaBoost           &  0.938(82)  &  <0.0001 &  0.89(12)  &  <0.0001 &  0.76091   &  <0.
↪0001 &  0.773(96) &  <0.0001 &    0.576(50) &  <0.0001 &  0.73(12)  &  <0.0001 &  0.
↪17(13)  &  <0.0001 \\
Decision Tree      &  0.86(16)   &  <0.0001 &  0.80(13)  &  <0.0001 &  0.76316   &  <0.
↪0001 &  0.80(52)  &   0.3009 &    2.8(18)   &   0.0270 &  0.68(45)  &   0.0792 &  0.
↪20(13)  &   0.0003 \\
Gaussian Process   &  0.977(47)  &  <0.0001 &  0.964(60) &  <0.0001 &  0.93049   &  <0.
↪0001 &  0.39(23)  &  <0.0001 &    0.33(14)  &  <0.0001 &  0.36(23)  &  <0.0001 &  0.
↪100(98) &  <0.0001 \\
Linear SVM         &  0.53(18)   &   0.1621 &  0.51(21)  &   0.8580 &  0.19756   &   0.
↪3660 &  1.066(80) &   0.1521 &    0.726(41) &   0.1514 &  1.079(96) &   0.1531 &  0.
↪60(16)  &   1.0000 \\
```

(continues on next page)

(continued from previous page)

```
Naive Bayes       &  0.9983(82) &  <0.0001 &  0.997(13) &  <0.0001 &  0.996(21) &  <0.
↪0001 &  0.64(20)  &  <0.0001 &    0.48(12)  &  <0.0001 &  0.63(21)  &  <0.0001 &  0.
↪30(15)  &   0.0003 \\
Nearest Neighbors &  0.996(15)  &  <0.0001 &  0.966(49) &  <0.0001 &  0.991(47) &  <0.
↪0001 &  0.30(16)  &  <0.0001 &    0.23(11)  &  <0.0001 &  0.28(16)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
Neural Net        &  0.993(23)  &  <0.0001 &  0.990(32) &  <0.0001 &  0.982(79) &  <0.
↪0001 &  0.69(14)  &  <0.0001 &    0.525(74) &  <0.0001 &  0.65(16)  &  <0.0001 &  0.
↪25(15)  &  <0.0001 \\
QDA               &  0.9983(83) &  <0.0001 &  0.997(11) &  <0.0001 &  0.996(32) &  <0.
↪0001 &  0.63(19)  &  <0.0001 &    0.47(11)  &  <0.0001 &  0.61(20)  &  <0.0001 &  0.
↪28(15)  &  <0.0001 \\
RBF SVM           &  0.979(44)  &  <0.0001 &  0.966(63) &  <0.0001 &  0.93680   &  <0.
↪0001 &  0.34(22)  &  <0.0001 &    0.29(14)  &  <0.0001 &  0.31(22)  &  <0.0001 &  0.
↪100(98) &  <0.0001 \\
Random Forest     &  0.90(13)   &  <0.0001 &  0.85(16)  &  <0.0001 &  0.64512   &   0.
↪0021 &  0.65(30)  &   0.0070 &    0.48(19)  &   0.0094 &  0.62(31)  &   0.0047 &  0.
↪23(14)  &   0.0006 \\
iid               &  0.60(16)   &     {--} &  0.5(0)    &     {--} &  0(0)      &
↪{--} &  1.071(85) &     {--} &    0.729(43) &     {--} &  1.08(11)  &     {--} &  0.
↪60(16)  &     {--} \\
\bottomrule
\end{tabular}
```

Dataset 2 (Linear)

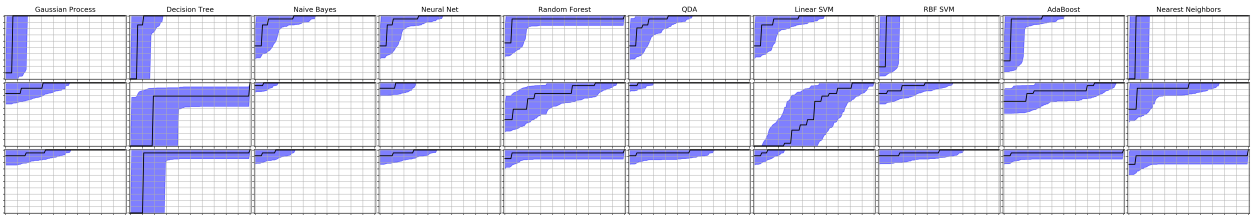
```
                        AP         p        AUC         p      AUPRG         p
↪Brier       p NLL (nats)         p     sphere         p   zero one         p
AdaBoost           0.984(43)  <0.0001  0.962(87)  <0.0001  0.96274    <0.0001  0.
↪21(23)   <0.0001  0.27(29)    0.0034  0.18(20)   <0.0001  0.050(71)  <0.0001
Decision Tree      0.91(14)   <0.0001  0.922(98)  <0.0001  0.88360    <0.0001  0.
↪30(35)    0.0002  1.0(12)     0.5706  0.26(30)   <0.0001  0.075(86)  <0.0001
Gaussian Process   0.984(38)  <0.0001  0.977(52)  <0.0001  0.96794    <0.0001  0.
↪25(24)   <0.0001  0.23(17)   <0.0001  0.23(23)   <0.0001  0.075(86)  <0.0001
Linear SVM         0.994(26)  <0.0001  0.992(23)  <0.0001  0.989(47)  <0.0001  0.
↪17(14)   <0.0001  0.163(86)  <0.0001  0.16(15)   <0.0001  0.050(71)  <0.0001
Naive Bayes        0.992(25)  <0.0001  0.990(32)  <0.0001  0.986(50)  <0.0001  0.
↪18(20)   <0.0001  0.15(15)   <0.0001  0.17(19)   <0.0001  0.050(71)  <0.0001
Nearest Neighbors  0.992(25)  <0.0001  0.946(78)  <0.0001  0.985(67)  <0.0001  0.
↪29(30)   <0.0001  0.76(98)    0.9063  0.25(26)   <0.0001  0.075(86)  <0.0001
Neural Net         0.987(35)  <0.0001  0.982(40)  <0.0001  0.975(83)  <0.0001  0.
↪24(19)   <0.0001  0.22(12)   <0.0001  0.21(19)   <0.0001  0.050(71)  <0.0001
QDA                0.984(42)  <0.0001  0.975(57)  <0.0001  0.96560    <0.0001  0.
↪21(24)   <0.0001  0.23(28)    0.0014  0.19(22)   <0.0001  0.075(86)  <0.0001
RBF SVM            0.980(45)  <0.0001  0.970(62)  <0.0001  0.95778    <0.0001  0.
↪21(25)   <0.0001  0.20(21)   <0.0001  0.18(23)   <0.0001  0.050(71)  <0.0001
Random Forest      0.990(25)  <0.0001  0.968(58)  <0.0001  0.981(73)  <0.0001  0.
↪25(25)   <0.0001  0.47(70)    0.5055  0.23(23)   <0.0001  0.075(86)  <0.0001
iid                0.55(16)       N/A  0.5(0)         N/A  0(0)           N/A  1.
↪018(43)      N/A  0.702(22)      N/A  1.021(52)      N/A  0.55(17)       N/A
```

Dataset 2 (Linear) in LaTeX

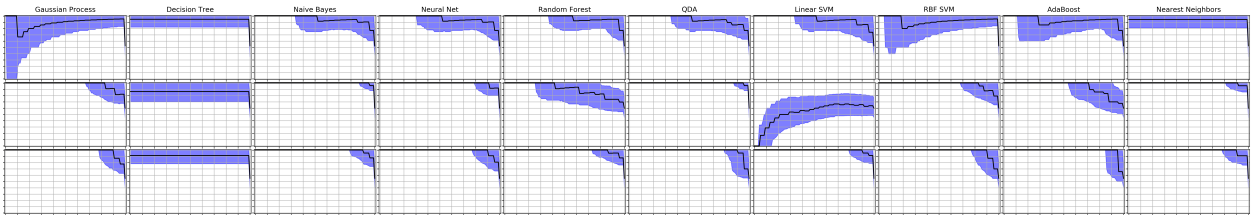
```
\begin{tabular}{|l|Sr|Sr|Sr|Sr|Sr|Sr|}
\toprule
{}                &       {AP} &      {p} &       {AUC} &      {p} &    {AUPRG} &
↪{p} &    {Brier} &      {p} & {NLL (nats)} &       {p} &   {sphere} &       {p} &
↪{zero one} &      {p} \\
\midrule
AdaBoost          &  0.984(43) &  <0.0001 &  0.962(87) &  <0.0001 &  0.96274   &  <0.
↪0001 &  0.21(23)  &  <0.0001 &    0.27(29)  &   0.0034 &  0.18(20)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
Decision Tree     &  0.91(14)  &  <0.0001 &  0.922(98) &  <0.0001 &  0.88360   &  <0.
↪0001 &  0.30(35)  &   0.0002 &    1.0(12)   &   0.5706 &  0.26(30)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
Gaussian Process  &  0.984(38) &  <0.0001 &  0.977(52) &  <0.0001 &  0.96794   &  <0.
↪0001 &  0.25(24)  &  <0.0001 &    0.23(17)  &  <0.0001 &  0.23(23)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
Linear SVM        &  0.994(26) &  <0.0001 &  0.992(23) &  <0.0001 &  0.989(47) &  <0.
↪0001 &  0.17(14)  &  <0.0001 &    0.163(86) &  <0.0001 &  0.16(15)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
Naive Bayes       &  0.992(25) &  <0.0001 &  0.990(32) &  <0.0001 &  0.986(50) &  <0.
↪0001 &  0.18(20)  &  <0.0001 &    0.15(15)  &  <0.0001 &  0.17(19)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
Nearest Neighbors &  0.992(25) &  <0.0001 &  0.946(78) &  <0.0001 &  0.985(67) &  <0.
↪0001 &  0.29(30)  &  <0.0001 &    0.76(98)  &   0.9063 &  0.25(26)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
Neural Net        &  0.987(35) &  <0.0001 &  0.982(40) &  <0.0001 &  0.975(83) &  <0.
↪0001 &  0.24(19)  &  <0.0001 &    0.22(12)  &  <0.0001 &  0.21(19)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
QDA               &  0.984(42) &  <0.0001 &  0.975(57) &  <0.0001 &  0.96560   &  <0.
↪0001 &  0.21(24)  &  <0.0001 &    0.23(28)  &   0.0014 &  0.19(22)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
RBF SVM           &  0.980(45) &  <0.0001 &  0.970(62) &  <0.0001 &  0.95778   &  <0.
↪0001 &  0.21(25)  &  <0.0001 &    0.20(21)  &  <0.0001 &  0.18(23)  &  <0.0001 &  0.
↪050(71) &  <0.0001 \\
Random Forest     &  0.990(25) &  <0.0001 &  0.968(58) &  <0.0001 &  0.981(73) &  <0.
↪0001 &  0.25(25)  &  <0.0001 &    0.47(70)  &   0.5055 &  0.23(23)  &  <0.0001 &  0.
↪075(86) &  <0.0001 \\
iid               &  0.55(16)  &     {--} &  0.5(0)    &     {--} &  0(0)      &
↪{--} &  1.018(43) &     {--} &    0.702(22) &     {--} &  1.021(52) &     {--} &  0.
↪55(17)  &     {--} \\
\bottomrule
\end{tabular}
```

ROC curves

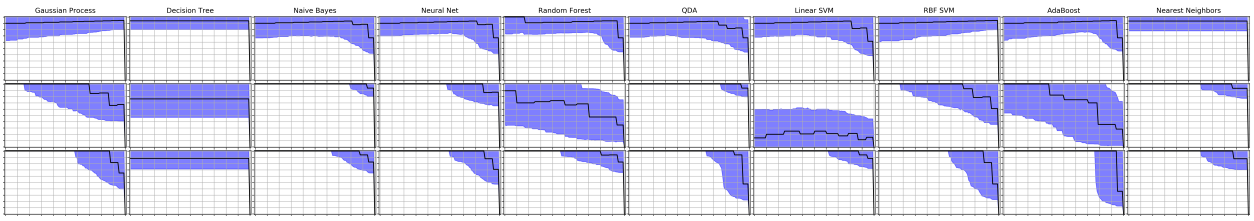
The *just_benchmark* routines also produces ROC curves with error bars from bootstrap analysis, which have been vectorized for speed:



Precision-recall curves



Precision-recall-gain curves



1.2 Usage for regression problems

The *mlpaper* package can also be applied to a regression problem with:

```
import mlpaper.regression as btr

full_tbl = btr.just_benchmark(X_train, y_train, X_test, y_test, regressors, STD_REGR_
↪LOSS, "iid", pairwise_CI=True)
```

Here we have used `pairwise_CI=True` which makes the confidence intervals based on the uncertainty of the loss *difference* to the reference method rather than a confidence interval on the actual loss.

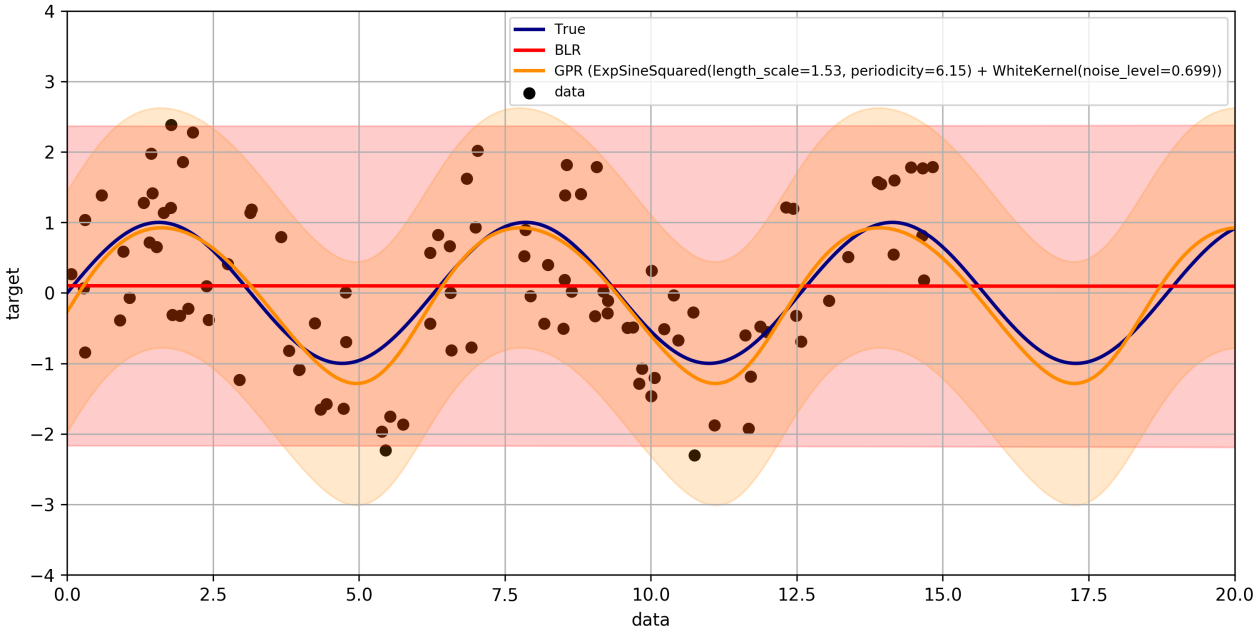
1.2.1 Output

By extending the sklearn regression demo we can make simple formatted tables:

```
              MAE        p           MSE         p   NLL (nats)         p
BLR  0.96933(30)   0.0979  1.39881(67)    0.0665  1.58842(57)    0.9828
GPR  0.75(13)      0.0009  0.75(28)      <0.0001  1.27(12)      <0.0001
iid  0.96908          N/A  1.3982            N/A  1.5884            N/A
```

or in LaTeX:

```
\begin{tabular}{|l|Sr|Sr|Sr|}
\toprule
{}  &        {MAE} &      {p} &         {MSE} &       {p} & {NLL (nats)} &       {p} \\
\midrule
BLR &  0.96933(30) &  0.0979 &  1.39881(67) &   0.0665 &  1.58842(57) &   0.9828 \\
GPR &  0.75(13)    &  0.0009 &  0.75(28)    &  <0.0001 &  1.27(12)    &  <0.0001 \\
iid &  0.96908     &     N/A &  1.3982      &      N/A &  1.5884      &      N/A \\
\bottomrule
\end{tabular}
```



1.3 Installation

Only `Python>=3.5` is officially supported, but older versions of Python likely work as well.

The core package itself can be installed with:

```
pip install mlpaper
```

To also get the dependencies for the demos in the README install with

```
pip install mlpaper[demo]
```

1.4 Contributing

The following instructions have been tested with Python 3.7.4 on Mac OS (10.14.6).

1.4.1 Install in editable mode

First, define the variables for the paths we will use:

```
GIT=/path/to/where/you/put/repos
ENVS=/path/to/where/you/put/virtualenvs
```

Then clone the repo in your git directory `$GIT`:

```
cd $GIT
git clone https://github.com/rdturnermtl/mlpaper.git
```

Inside your virtual environments folder `$ENVS`, make the environment:

```
cd $ENVS
virtualenv mlpaper --python=python3.7
source $ENVS/mlpaper/bin/activate
```

Now we can install the pip dependencies. Move back into your git directory and run

```
cd $GIT/mlpaper
pip install -r requirements/base.txt
pip install -e .  # Install the package itself
```

1.4.2 Contributor tools

First, we need to setup some needed tools:

```
cd $ENVS
virtualenv mlpaper_tools --python=python3.7
source $ENVS/mlpaper_tools/bin/activate
pip install -r $GIT/mlpaper/requirements/tools.txt
```

To install the pre-commit hooks for contributing run (in the `mlpaper_tools` environment):

```
cd $GIT/mlpaper
pre-commit install
```

To rebuild the requirements, we can run:

```
cd $GIT/mlpaper

# Check if there any discrepancies in the .in files
pipreqs mlpaper/ --diff requirements/base.in
pipreqs tests/ --diff requirements/test.in
pipreqs demos/ --diff requirements/demo.in
pipreqs docs/ --diff requirements/docs.in

# Regenerate the .txt files from .in files
pip-compile-multi --no-upgrade
```

1.4.3 Generating the documentation

First setup the environment for building with `Sphinx`:

```
cd $ENVS
virtualenv mlpaper_docs --python=python3.7
source $ENVS/mlpaper_docs/bin/activate
pip install -r $GIT/mlpaper/requirements/docs.txt
```

Then we can do the build:

```
cd $GIT/mlpaper/docs
make all
open _build/html/index.html
```

Documentation will be available in all formats in `Makefile`. Use `make html` to only generate the HTML documentation.

1.4.4 Running the tests

The tests for this package can be run with:

```
cd $GIT/mlpaper
./local_test.sh
```

The script creates an environment using the requirements found in `requirements/test.txt`. A code coverage report will also be produced in `$GIT/mlpaper/htmlcov/index.html`.

1.4.5 Deployment

The wheel (tar ball) for deployment as a pip installable package can be built using the script:

```
cd $GIT/mlpaper/
./build_wheel.sh
```

1.5 Links

The source is hosted on GitHub.

The documentation is hosted at Read the Docs.

Installable from PyPI.

1.6 License

This project is licensed under the Apache 2 License - see the LICENSE file for details.

CHAPTER

TWO

CODE OVERVIEW

2.1 Bootstrap Utilities

mlpaper.boot_util.**basic**(*boot_estimates*, *original_estimate*, *confidence=0.95*)

Build confidence interval using basic boostrap method.

Parameters

- **boot_estimates** (*ndarray, shape (n_boot, ..)*) – Estimated quantity across different bootstrap replications.
- **original_estimate** (*ndarray, shape (..)*) – Quantity estimated using original (non-bootstrap) data set.
- **confidence** (*float*) – Confidence level, use 0.95 for 95% interval. Must be in (0,1).

Returns

- **LB** (*ndarray, shape (...)*) – Lower end of confidence interval.
- **UB** (*ndarray, shape (...)*) – Upper end of confidence interval.

mlpaper.boot_util.**boot_weights**(*N*, *n_boot*, *epsilon=0*)

Sample weights for data points that makes it equivalent to bootstrap resampling of data points.

Parameters

- **N** (*int*) – Number of data points must be >= 1..
- **n_boot** (*int*) – Number of bootstrap replicates, must be >= 1.
- **epsilon** (*int or float*) – Minimum weight, typically 0 unless this creates numerical problems for a down stream algorithm in which case a value such as 1e-10 is used.

Returns **weight** – Weights equivalent to resampling for bootstrap algorithm.

Return type ndarray, shape (n_boot, N)

mlpaper.boot_util.**confidence_to_percentiles**(*confidence*)

Convert confidence level to percentiles in sampling distribution to build confidence interval.

Parameters **confidence** (*float*) – Confidence level, use 0.95 for 95% interval. Must be in (0,1).

Returns

- **LB** (*float*) – Lower end quantile in (0,1).
- **UB** (*float*) – Upper end quantile in (0,1).

Examples

```
>>> confidence_to_percentiles(0.95)
(2.5, 97.5)
```

mlpaper.boot_util.**error_bar**(*boot_estimates*, *original_estimate*, *confidence=0.95*)

Build error bar using boostrap method. The results is the same regardless of whether the percentile or basic boostrap is used for CIs.

Parameters

- **boot_estimates** (*ndarray, shape (n_boot,)*) – Estimated quantity across different bootstrap replications.
- **original_estimate** (*float*) – Quantity estimated using original (non-bootstrap) data set.
- **confidence** (*float*) – Confidence level, use 0.95 for 95% interval. Must be in (0,1).

Returns **EB** – Error bar around the original estimate.

Return type float

mlpaper.boot_util.**percentile**(*boot_estimates*, *confidence=0.95*)

Build confidence interval using percentile boostrap method.

Parameters

- **boot_estimates** (*ndarray, shape (n_boot, ..)*) – Estimated quantity across different bootstrap replications.
- **confidence** (*float*) – Confidence level, use 0.95 for 95% interval. Must be in (0,1).

Returns

- **LB** (*ndarray, shape (...)*) – Lower end of confidence interval.
- **UB** (*ndarray, shape (...)*) – Upper end of confidence interval.

mlpaper.boot_util.**significance**(*boot_estimates*, *ref*)

Perform a two-sided bootstrap based hypothesis test on whether the unknown quantity is equal to some reference.

Parameters

- **boot_estimates** (*ndarray, shape (n_boot,)*) – Estimated quantity across different bootstrap replications.
- **ref** (*float or ndarray of shape (n_boot,)*) – Reference value is in hypothesis test. Use a scalar value for a known reference value or a array of n_boot bootstraped value to perform a paired test against another unknown quantity.

Returns **pval** – Resulting p-value of hypothesis test in (0,1).

Return type float

2.2 Benchmarking for Classification

class mlpaper.classification.**JustNoise**(*n_labels=2*, *pseudo_count=0.0*)

Class version of iid predictor compatible with sklearn interface. Same as `sklearn.dummy.DummyClassifier(strategy='prior')`.

mlpaper.classification.**brier_loss**(*y*, *log_pred_prob*, *rescale=True*)
Compute (rescaled) Brier loss.

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point.
- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape (len(y), n_labels). Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.
- **rescale** (*bool*) – If True, linearly rescales lost so perfect (P=1) predictions give 0.0 loss and a uniform prediction gives loss of 1.0. False gives the standard Brier loss.

Returns **loss** – Array of the Brier loss for the predictions on each data point in *y*.

Return type ndarray, shape (n_samples,)

mlpaper.classification.**check_curve**(*result*, *x_grid=None*)
Check performance curve output matches expected format and return the curve after validation.

Parameters

- **curve** (*result of curve function, e.g., classification.roc_curve*) – Curves defined by a ROC or other curve estimation.
- **x_grid** (*None or ndarray of shape (n_grid,)*) – If provided, check that all the curves are defined over a wider range than the x_grid. So, when the functions are interpolated onto the range of x_grid no extrapolation is needed.

Returns **curve** – Returns same object passed in after some input checks. Each of the ndarrays have shape (n_boot, n_thresholds).

Return type tuple of (ndarray, ndarray, str)

mlpaper.classification.**curve_boot**(*y*, *log_pred_prob*, *ref*, *curve_f=<function roc_curve>*, *x_grid=None*, *n_boot=1000*, *pairwise_CI=False*, *confidence=0.95*)
Perform boot strap analysis of performance curve, e.g., ROC or prec-rec. For binary classification only.

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – Array containing true labels, must be *bool* or {0,1}.
- **log_pred_prob** (*ndarray, shape (n_samples, 2)*) – Array of shape (len(y), 2). Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. However, many curves (e.g., ROC) are invariant to monotonic transformation and hence linear scale could also be used.
- **ref** (*float or ndarray of shape (n_samples, 2)*) – If *ref* is an rray of shape (len(y), 2): Same as *log_pred_prob* except for the reference (baseline) method if a paired statistical test is desired on the area under the curve. If *ref* is a scalar float: *curve_boot* tests the statistical significance that the area under the curve differs from *ref* in a non-paired test. For ROC analysis, *ref* is typically 0.5.
- **curve_f** (*callable*) – Function to compute the performance curve. Standard choices are: *perf_curves.roc_curve* or *perf_curves.recall_precision_curve*.
- **x_grid** (*None or ndarray of shape (n_grid,)*) – Grid of points to evaluate curve in results. If *None*, defaults to linear grid on [0,1].
- **n_boot** (*int*) – Number of bootstrap iterations to perform.

- **pairwise_CI** (*bool*) – If True, compute error bars on `summary - summary_ref` instead of just the summary. This typically results in smaller error bars.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.

Returns

- **summary** (*tuple of floats, shape (3,)*) – Tuple containing (mu, EB, pval), where mu is the best estimate on the summary statistic of the curve, EB is the error bar, and pval is the p-value from the two-sided boot strap significance test that its value is the same as the reference summary value (from either *log_pred_prob_ref* or *default_summary_ref*).
- **curve** (*DataFrame, shape (n_grid, 4)*) – DataFrame containing four columns: *x_grid*, the curve value, the lower end of confidence envelope, and the upper end of the confidence envelope.

mlpaper.classification.**curve_summary_table**(*log_pred_prob_table*, *y*, *curve_dict*, *ref_method*, *x_grid=None*, *n_boot=1000*, *pairwise_CI=False*, *confidence=0.95*)

Build table with mean and error bars of curve summaries from a table of probalistic predictions.

Parameters

- **log_pred_prob_table** (*DataFrame, shape (n_samples, n_methods * n_labels)*) – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x labels. For exampe, `log_pred_prob_table.loc[5, 'foo']` is the categorical distribution (in log scale) prediction that method foo places on `y[5]`.
- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point. Must be of same length as DataFrame *log_pred_prob_table*.
- **curve_dict** (*dict of str to callable*) – Dictionary mapping curve name to performance curve. Standard choices: *perf_curves.roc_curve* or *perf_curves.recall_precision_curve*.
- **ref_method** (*str*) – Name of method that is used as reference point in paired statistical tests. This is usually some some of baseline method. *ref_method* must be found in the 1st level of the columns of *log_pred_prob_table*.
- **x_grid** (*None or ndarray of shape (n_grid,)*) – Grid of points to evaluate curve in results. If *None*, defaults to linear grid on [0,1].
- **n_boot** (*int*) – Number of bootstrap iterations to perform.
- **pairwise_CI** (*bool*) – If True, compute error bars on `summary - summary_ref` instead of just the summary. This typically results in smaller error bars.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.

Returns

- **curve_tbl** (*DataFrame, shape (n_methods, n_metrics * 3)*) – DataFrame with curve summary of each method according to each curve. The rows are the methods. The columns are a hierarchical index that is the cartesian product of curve x (summary, error bar, p-value). That is, `curve_tbl.loc['foo', 'bar']` is a pandas series with (summary of bar curve on foo, corresponding error bar, statistical sig) The statistical significance is a p-value from a two-sided hypothesis test on the hypothesis H0 that foo has the same curve summary as the reference method *ref_method*.
- **curve_dump** (*dict of (str, str) to DataFrame of shape (n_grid, 4)*) – Each key is a pair of (method name, curve name) with the value being a pandas dataframe with the performance

curve, which has four columns: *x_grid*, the curve value, the lower end of confidence envelope, and the upper end of the confidence envelope.

mlpaper.classification.**get_pred_log_prob**(*X_train*, *y_train*, *X_test*, *n_labels*, *methods*, *min_log_prob=-inf*, *verbose=False*, *checkpointdir=None*)

Get the predictive probability tables for each test point on a collection of classification methods.

Parameters

- **X_train** (*ndarray, shape (n_train, n_features)*) – Training set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_train** (*ndarray of type int or bool, shape (n_train,)*) – Training set 1d array of truth labels for classifiers. Must be of same length as *X_train*. Values must be in range [0, *n_labels*) or *bool*.
- **X_test** (*ndarray, shape (n_test, n_features)*) – Test set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **n_labels** (*int*) – Number of labels, must be >= 1. This is not infered from *y* because some labels may not be found in small data chunks.
- **methods** (*dict of str to sklearn estimator*) – Dictionary mapping method name (*str*) to object that performs training and test. Object must follow the interface of sklearn estimators, that is it has a `fit()` method and either a `predict_log_proba()` or `predict_proba()` method.
- **min_log_prob** (*float*) – Minimum value to floor the predictive log probabilities (while still normalizing). Must be < 0. Useful to prevent inf log loss penalties.
- **verbose** (*bool*) – If True, display which method being trained.
- **checkpointdir** (*str (directory)*) – If provided, stores checkpoint results using joblib for the train/test in case process interrupted. If None, no checkpointing is done.

Returns **log_pred_prob_table** – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x labels. For exampe, `log_pred_prob_table.loc[5, 'foo']` is the categorical distribution (in log scale) prediction that method foo places on `y[5]`.

Return type DataFrame, shape (n_samples, n_methods * n_labels)

Notes

If a train/test operation is loaded from a checkpoint file, the estimator object in methods will not be in a fit state.

mlpaper.classification.**hard_loss**(*y*, *log_pred_prob*, *loss_mat=None*)

Loss function for making classification decisions from a loss matrix.

This function both computes the optimal action under the predictive distribution and the loss matrix, and then scores that decision using the loss matrix.

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point.
- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape `(len(y), n_labels)`. Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.

- **loss_mat** (*None or ndarray of shape (n_labels, n_actions)*) – Loss matrix to use for making decisions of size `(n_labels, n_actions)`. The loss of taking action a when the true outcome (label) is y is found in `loss_mat[y, a]`. If None, 1 - identity matrix is used to obtain the 0-1 loss function.

Returns **loss** – Array of the resulting loss for the predictions on each point in *y*.

Return type ndarray, shape (n_samples,)

`mlpaper.classification.`**hard_loss_decision**(*log_pred_prob*, *loss_mat*)

Make Bayes' optimal action according to predictive probability distribution and loss matrix.

Parameters

- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape `(len(y), n_labels)`. Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.
- **loss_mat** (*ndarray, shape (n_labels, n_actions)*) – Loss matrix to use for making decisions of size `(n_labels, n_actions)`. The loss of taking action a when the true outcome (label) is y is found in `loss_mat[y, a]`.

Returns **action** – Array of resulting Bayes' optimal action for each data point.

Return type ndarray of type int, shape (n_samples,)

`mlpaper.classification.`**just_benchmark**(*X_train*, *y_train*, *X_test*, *y_test*, *n_labels*, *methods*, *loss_dict*, *curve_dict*, *ref_method*, *min_pred_log_prob=-inf*, *pairwise_CI=False*, *method_EB='t'*, *limits={}*)

Simplest one-call interface to this package. Just pass it data and method objects and a performance summary DataFrame is returned.

Parameters

- **X_train** (*ndarray, shape (n_train, n_features)*) – Training set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_train** (*ndarray of type int or bool, shape (n_train,)*) – Training set 1d array of truth labels for classifiers. Must be of same length as *X_train*. Values must be in range [0, *n_labels*) or *bool*.
- **X_test** (*ndarray, shape (n_test, n_features)*) – Test set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_test** (*ndarray of type int or bool, shape (n_test,)*) – Test set 1d array of truth labels for classifiers. Must be of same length as *X_test*. Values must be in range [0, *n_labels*) or *bool*.
- **n_labels** (*int*) – Number of labels, must be >= 1. This is not infered from *y* because some labels may not be found in small data chunks.
- **methods** (*dict of str to sklearn estimator*) – Dictionary mapping method name (*str*) to object that performs training and test. Object must follow the interface of sklearn estimators, that is it has a `fit()` method and either a `predict_log_proba()` or `predict_proba()` method.
- **loss_dict** (*dict of str to callable*) – Dictionary mapping loss function name to function that computes loss, e.g., *log_loss*, *brier_loss*, ...
- **curve_dict** (*dict of str to callable*) – Dictionary mapping curve name to performance curve. Standard choices: *perf_curves.roc_curve* or *perf_curves.recall_precision_curve*.

- **ref_method** (*str*) – Name of method that is used as reference point in paired statistical tests. This is usually some some of baseline method. *ref_method* must be found in *methods* dictionary.
- **min_log_prob** (*float*) – Minimum value to floor the predictive log probabilities (while still normalizing). Must be < 0. Useful to prevent inf log loss penalties.
- **pairwise_CI** (*bool*) – If True, compute error bars on the mean of `loss - loss_ref` instead of just the mean of *loss*. This typically gives smaller error bars.
- **method_EB** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.
- **limits** (*dict of str to (float, float)*) – Dictionary mapping metric name to tuple with (lower, upper) which are the theoretical limits on the mean loss. For instance, zero-one loss should be `(0.0, 1.0)`. If entry missing, (-inf, inf) is used.

Returns

- **full_tbl** (*DataFrame, shape (n_methods, (n_loss + n_curve) * 3)*) – DataFrame with curve/loss summary of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (summary, error bar, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo, corresponding error bar, statistical sig) The statistical significance is a p-value from a two-sided hypothesis test on the hypothesis H0 that foo has the same metric as the reference method *ref_method*.
- **curve_dump** (*dict of (str, str) to DataFrame of shape (n_grid, 4)*) – Each key is a pair of (method name, curve name) with the value being a pandas dataframe with the performance curve, which has four columns: *x_grid*, the curve value, the lower end of confidence envelope, and the upper end of the confidence envelope. Only metrics from *curve_dict* and *not* from *loss_dict* are found here.

mlpaper.classification.**log_loss**(*y*, *log_pred_prob*)

Compute log loss (e.g, negative log likelihood or cross-entropy).

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point.
- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape `(len(y), n_labels)`. Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.

Returns **loss** – Array of the log loss for the predictions on each data point in *y*.

Return type ndarray, shape (n_samples,)

mlpaper.classification.**loss_table**(*log_pred_prob_table*, *y*, *metrics_dict*, *assume_normalized=False*)

Compute loss table from table of probalistic predictions.

Parameters

- **log_pred_prob_table** (*DataFrame, shape (n_samples, n_methods * n_labels)*) – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x labels. For exampe, `log_pred_prob_table.loc[5, 'foo']` is the categorical distribution (in log scale) prediction that method foo places on `y[5]`.
- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point. Must be of same length as DataFrame *log_pred_prob_table*.

- **metrics_dict** (*dict of str to callable*) – Dictionary mapping loss function name to function that computes loss, e.g., *log_loss*, *brier_loss*, ...
- **assume_normalized** (*bool*) – If False, renormalize the predictive distributions to ensure there is no cheating. If True, skips this step for speed.

Returns **loss_tbl** – DataFrame with loss of each method according to each loss function on each data point. The rows are the data points in *y* (that is the index matches *log_pred_prob_table*). The columns are a hierarchical index that is the cartesian product of loss x method. That is, the loss of method foo's prediction of `y[5]` according to loss function bar is stored in `loss_tbl.loc[5, ('bar', 'foo')]`.

Return type DataFrame, shape (n_samples, n_metrics * n_methods)

mlpaper.classification.**shape_and_validate**(*y*, *log_pred_prob*)

Validate shapes and types of predictive distribution against data and return the shape information.

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point.
- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape `(len(y), n_labels)`. Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.

Returns

- **n_samples** (*int*) – Number of data points (length of *y*)
- **n_labels** (*int*) – The number of possible labels in *y*. Inferred from size of *log_pred_prob* and *not* from *y*.

Notes

This does *not* check normalization.

mlpaper.classification.**spherical_loss**(*y*, *log_pred_prob*, *rescale=True*)

Compute (rescaled) spherical loss.

Parameters

- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point.
- **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Array of shape `(len(y), n_labels)`. Each row corresponds to a categorical distribution with *normalized* probabilities in log scale. Therefore, the number of columns must be at least 1.
- **rescale** (*bool*) – If True, linearly rescales lost so perfect (P=1) predictions give 0.0 loss and a uniform prediction gives loss of 1.0. False gives the standard spherical loss, which is the negative spherical *score*.

Returns **loss** – Array of the spherical loss for the predictions on each point in *y*.

Return type ndarray, shape (n_samples,)

mlpaper.classification.**summary_table**(*log_pred_prob_table*, *y*, *loss_dict*, *curve_dict*, *ref_method*, *x_grid=None*, *n_boot=1000*, *pairwise_CI=False*, *confidence=0.95*, *method_EB='t'*, *limits={}*)

Build table with mean and error bars of both loss and curve summaries from a table of probalistic predictions.

Parameters

- **log_pred_prob_table** (*DataFrame, shape (n_samples, n_methods * n_labels)*) – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x labels. For exampe, `log_pred_prob_table.loc[5, 'foo']` is the categorical distribution (in log scale) prediction that method foo places on `y[5]`.
- **y** (*ndarray of type int or bool, shape (n_samples,)*) – True labels for each classication data point. Must be of same length as DataFrame *log_pred_prob_table*.
- **loss_dict** (*dict of str to callable*) – Dictionary mapping loss function name to function that computes loss, e.g., *log_loss*, *brier_loss*, ...
- **curve_dict** (*dict of str to callable*) – Dictionary mapping curve name to performance curve. Standard choices: *perf_curves.roc_curve* or *perf_curves.recall_precision_curve*.
- **ref_method** (*str*) – Name of method that is used as reference point in paired statistical tests. This is usually some some of baseline method. *ref_method* must be found in the 1st level of the columns of *log_pred_prob_table*.
- **x_grid** (*None or ndarray of shape (n_grid,)*) – Grid of points to evaluate curve in results. If *None*, defaults to linear grid on [0,1].
- **n_boot** (*int*) – Number of bootstrap iterations to perform for performance curves.
- **pairwise_CI** (*bool*) – If True, compute error bars on `summary - summary_ref` instead of just the summary. This typically results in smaller error bars.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.
- **method_EB** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.
- **limits** (*dict of str to (float, float)*) – Dictionary mapping metric name to tuple with (lower, upper) which are the theoretical limits on the mean loss. For instance, zero-one loss should be `(0.0, 1.0)`. If entry missing, (-inf, inf) is used.

Returns

- **full_tbl** (*DataFrame, shape (n_methods, (n_loss + n_curve) * 3)*) – DataFrame with curve/loss summary of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (summary, error bar, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo, corresponding error bar, statistical sig) The statistical significance is a p-value from a two-sided hypothesis test on the hypothesis H0 that foo has the same metric as the reference method *ref_method*.
- **curve_dump** (*dict of (str, str) to DataFrame of shape (n_grid, 4)*) – Each key is a pair of (method name, curve name) with the value being a pandas dataframe with the performance curve, which has four columns: *x_grid*, the curve value, the lower end of confidence envelope, and the upper end of the confidence envelope. Only metrics from *curve_dict* and *not* from *loss_dict* are found here.

2.3 Data Splitting Tools

`mlpaper.data_splitter.`**build_lag_df**(*df*, *n_lags*, *stride=1*, *features=None*)

Build a lad dataframe from dataframe where the rows are ordered time indices for a time series data set. This is

useful for autoregressive models.

Parameters

- **df** (*DataFrame, shape (n_samples, n_cols)*) – Orginal dataset we want to build lag data set from.
- **n_lags** (*int*) – Number of lags. `n_lags=1` means only the original data set. Must be >= 1.
- **stride** (*int*) – Stride of the lags. For instance, `stride=2` means only even lags.
- **features** (*array-like, shape (n_features,)*) – Subset of columns in *df* to include in the lags data. All columns are retained for lag 0. For data frames containing features and targets, the features (inputs) can be placed in *features* so the targets (outputs) are only present for lag 0. If None, use all columns.

Returns **df** – New data frame where lags data frames have been concat'ed together. The columns are a new hierarchical index with the lag at the lowest level.

Return type DataFrame, shape (n_samples, n_cols + (n_lags - 1) * n_features)

Examples

```
>>> data=np.random.choice(10,size=(4,3))
>>> df=pd.DataFrame(data=data,columns=['a','b','c'])
>>> ds.build_lag_df(df,3,features=['a','b'])
      a  b  c   a   b   a   b
  lag L0 L0 L0  L1  L1  L2  L2
  0    2  2  2 NaN NaN NaN NaN
  1    2  9  4   2   2 NaN NaN
  2    8  4  0   2   9   2   2
  3    3  5  6   8   4   2   9
```

mlpaper.data_splitter.**index_to_series**(*index*)

Make a pandas series from a pandas index with the value equal to index.

Parameters **index** (*Index*) – Pandas Index to make series from.

Returns **S** – Pandas series where `s[idx] = idx`.

Return type Series

Examples

```
>>> index_to_series(pd.Index([1,5,7]))
1    1
5    5
7    7
dtype: int64
```

mlpaper.data_splitter.**linear_split_series**(*S*, *frac*, *assume_sorted=False*, *assume_unique=False*)

Create a binary mask to split a series into training/test based on a linear split based on values of series. That is, the train/test divide is based on a point that is a linear interpolation between lowest value and highest value in the series.

Parameters

- **S** (*Series, shape (n_samples,)*) – Pandas Series whose index will be used for binary mask. The linear split is based on the series *values*.
- **frac** (*float*) – Fraction of region be between series min and series max we want to be True. Must be in [0,1].
- **assume_sorted** (*bool*) – If True, assume series is already sorted based on values. This can be used for computational speedups.
- **assume_unique** (*bool*) – If True, assume all values in series are unique. This can be used for computational speedups.

Returns **train_curr** – Binary mask with index matching *S*.

Return type Series with values of type bool, shape (n_samples,)

mlpaper.data_splitter.**ordered_split_series**(*S*, *frac*, *assume_sorted=False*, *assume_unique=False*)

Create a binary mask to split a series into training/test based on a ordered split based on values of series. That is, indices with a lower value get put in train and the rest go in test.

Parameters

- **S** (*Series, shape (n_samples,)*) – Pandas Series whose index will be used for binary mask. The ordered split is based on the series *values*.
- **frac** (*float*) – Fraction of elements we want to be True. Must be in [0,1].
- **assume_sorted** (*bool*) – If True, assume series is already sorted based on values. This can be used for computational speedups.
- **assume_unique** (*bool*) – If True, assume all values in series are unique. This can be used for computational speedups.

Returns **train_curr** – Binary mask with index matching *S*.

Return type Series with values of type bool, shape (n_samples,)

mlpaper.data_splitter.**rand_mask**(*n_samples*, *frac*)

Make a random binary mask with a certain fraction. Rounds number of elements up to next integer when exact fraction is not possible.

Parameters

- **n_samples** (*int*) – Length of mask.
- **frac** (*float*) – Fraction of elements we want to be True. Must be in [0,1].

Returns **L** – Random binary mask.

Return type ndarray of type bool, shape (n_samples,)

mlpaper.data_splitter.**rand_subset**(*x*, *frac*)

Take random subset of array *x* with a certain fraction. Rounds number of elements up to next integer when exact fraction is not possible.

Parameters

- **x** (*array-like, shape (n_samples,)*) – List that we want a subset of.
- **frac** (*float*) – Fraction of *x* elements we want to keep in subset. Must be in [0,1].

Returns **L** – Array that is subset with m_samples = ceil(frac * n_samples) samples.

Return type ndarray, shape (m_samples,)

mlpaper.data_splitter.**random_split_series**(*S*, *frac*, *assume_sorted=False*, *assume_unique=False*)

Create a binary mask to split a series into training/test based on a random split based on values of series. That is, elements with the same value in the series always get grouped into both train or both test.

Parameters

- **S** (*Series, shape (n_samples,)*) – Pandas Series whose index will be used for binary mask. Random splitting is based on a random parititioning of the series *values*.
- **frac** (*float*) – Fraction of elements we want to be True. Must be in [0,1].
- **assume_sorted** (*bool*) – If True, assume series is already sorted based on values. This can be used for computational speedups.
- **assume_unique** (*bool*) – If True, assume all values in series are unique. This can be used for computational speedups.

Returns **train_curr** – Random binary mask with index matching *S*.

Return type Series with values of type bool, shape (n_samples,)

mlpaper.data_splitter.**split_df**(*df*, *splits={None: ('random', 0.8)}*, *assume_unique=()*, *assume_sorted=()*)

Split a pandas data frame based on criteria across multiple columns.

A seperate train test split is done for each column specified as a split column in *splits*. A row is added to the final training set, only if it is placed in training by every column splits. Likewise, A row is added to the final test set, only if it is placed in test by every column splits. All other rows are placed in the unused data points DataFrame.

Parameters

- **df** (*DataFrame, shape (n_samples, n_features)*) – DataFrame we wish to split into training and test chunks
- **splits** (*dict of object to ({RANDOM, ORDRED, LINEAR}, float)*) – Dictionary explaining how to do the split. The keys of the *splits* are the columns in *df* we will base the split on. The constant INDEX can be used to symbolize that the index is the desired column. Each value is a tuple with (split type, fraction for training). The split type can be either: random, ordered, or linear. The fraction for training must be in [0,1]. Fraction of region be between series min and series max we want to be True. The Fraction must be in [0,1]. If *splits* is omitted, the default is to perform a 80-20 random split based on the index.
- **assume_sorted** (*array-like of str*) – Columns that we can assume are alreay sorted by value. This can be used for computational speedups.
- **assume_unique** (*array-like of str*) – Columns that we can assume have unique values. This can be used for computational speedups.

Returns

- **df_train** (*DataFrame, shape (n_train, n_features)*) – Subset of *df* placed in training set.
- **df_test** (*DataFrame, shape (n_test, n_features)*) – Subset of *df* placed in test set.
- **df_unused** (*DataFrame, shape (n_unused, n_features)*) – Subset of *df* not in training or test. This will be empty if only a single column is ued in *splits*.

2.4 Core Routines

mlpaper.mlpaper.**bernstein_EB**(*x*, *lower*, *upper*, *confidence=0.95*)

Get Bernstein bound based error bars on mean of *x*. This error bar makes no distributional or central limit theorem assumption on *x*.

Parameters

- **x** (*array-like, shape (n_samples,)*) – Data points to estimate mean. Must not be empty or contain NaNs.
- **lower** (*float*) – A priori known theoretical lower limit on unknown mean. For instance, for mean zero-one loss, `lower=0`.
- **upper** (*float*) – A priori known theoretical upper limit on unknown mean. For instance, for mean zero-one loss, `upper=1`.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct confidence interval from t statistic.

Returns **EB** – Size of error bar on mean (>= 0). The confidence interval is `[mean(x) - EB, mean(x) + EB]`. `EB = upper - lower` is inf when `len(x) = 0`.

Return type float

Notes

This does not do clipping of to trivial error bars, i.e., *EB* could be larger than `upper - lower`. However, *clip_EB* can be called to enforce trivial error bar limits.

References

Audibert, Jean-Yves, Remi Munos, and Csaba Szepesvari. "Exploration-exploitation tradeoff using variance estimates in multi-armed bandits." Theoretical Computer Science 410.19 (2009): 1876-1902.

mlpaper.mlpaper.**bernstein_test**(*x*, *lower*, *upper*)

Perform Bernstein bound-based test to test if the values in *x* are sampled from a distribution with a zero mean. This test makes no distributional or central limit theorem assumption on *x*.

As a result the bound may be loose and the p-value will not be sampled from a uniform distribution under H0 (E[x] = 0), but rather be skewed larger than uniform.

Parameters

- **x** (*array-like, shape (n_samples,)*) – array of data points to test.
- **lower** (*float*) – A priori known theoretical lower limit on unknown mean. For instance, for mean zero-one loss, `lower=0`.
- **upper** (*float*) – A priori known theoretical upper limit on unknown mean. For instance, for mean zero-one loss, `upper=1`.

Returns **pval** – p-value (in [0,1]) from t-test on *x*.

Return type float

mlpaper.mlpaper.**boot_EB**(*x*, *confidence=0.95*, *n_boot=1000*)

Get bootstrap bound based error bars on mean of *x*.

Parameters

- **x** (*array-like, shape (n_samples,)*) – Data points to estimate mean. Must not be empty or contain NaNs.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct confidence interval from t statistic.
- **n_boot** (*int*) – Number of bootstrap iterations to perform.

Returns **EB** – Size of error bar on mean (>= 0). The confidence interval is `[mean(x) - EB, mean(x) + EB]`. *EB* is inf when `len(x) <= 1`.

Return type float

mlpaper.mlpaper.**boot_test**(*x*, *n_boot=1000*)

Perform a bootstrap-based test to test if the values in *x* are sampled from a distribution with a zero mean.

Parameters

- **x** (*array-like, shape (n_samples,)*) – array of data points to test.
- **n_boot** (*int*) – Number of bootstrap iterations to perform.

Returns **pval** – p-value (in [0,1]) from t-test on *x*.

Return type float

mlpaper.mlpaper.**clip_EB**(*mu*, *EB*, *lower=-inf*, *upper=inf*, *min_EB=0.0*)

Clip error bars to both a minimum uncertainty level and a maximum level determined by trivial error bars from the a prior known limits of the unknown parameter *theta*. Similar to *np.clip*, but for error bars.

Parameters

- **mu** (*float*) – Point estimate of unknown parameter *theta* around which error bars are based.
- **EB** (*float*) – Size of error bar around *mu* (`EB > 0`). The confidence interval on *theta* is `[mu - EB, mu + EB]`.
- **lower** (*float*) – A priori known theoretical lower limit on unknown parameter *theta*. For instance, for mean zero-one loss, `lower=0`.
- **upper** (*float*) – A priori known theoretical upper limit on unknown parameter *theta*. For instance, for mean zero-one loss, `upper=1`.
- **min_EB** (*float*) – Minimum size beleivable size of error bar. Typically, leave `min_EB=0` for simplicity.

Returns **EB** – Error bar after possible clipping.

Return type float

mlpaper.mlpaper.**get_mean_EB_test**(*x*, *confidence=0.95*, *min_EB=0.0*, *lower=-inf*, *upper=inf*, *method='t'*)

Get mean loss and estimated error bar. Also, perform a statistical test to determine if the values in *x* are sampled from a distribution with a zero mean.

Parameters

- **x** (*ndarray, shape (n_samples,)*) – Array of independent observations.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.
- **min_EB** (*float*) – Minimum size of resulting error bar regardless of the data in *x*.
- **lower** (*float*) – A priori known theoretical lower limit on unknown mean of *x*. For instance, for mean zero-one loss, `lower=0`.

- **upper** (*float*) – A priori known theoretical upper limit on unknown mean of *x*. For instance, for mean zero-one loss, `upper=1`.
- **method** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.

Returns

- **mu** (*float*) – Estimated mean of *x*.
- **EB** (*float*) – Size of error bar on mean of *x* (`EB > 0`). The confidence interval is `[mu - EB, mu + EB]`.
- **pval** (*float*) – p-value (in [0,1]) from statistical test on *x*.

`mlpaper.mlpaper.`**get_mean_and_EB**(*x*, *confidence=0.95*, *min_EB=0.0*, *lower=-inf*, *upper=inf*, *method='t'*)

Get mean loss and estimated error bar.

Parameters

- **x** (*ndarray, shape (n_samples,)*) – Array of independent observations.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.
- **min_EB** (*float*) – Minimum size of resulting error bar regardless of the data in *x*.
- **lower** (*float*) – A priori known theoretical lower limit on unknown mean of *x*. For instance, for mean zero-one loss, `lower=0`.
- **upper** (*float*) – A priori known theoretical upper limit on unknown mean of *x*. For instance, for mean zero-one loss, `upper=1`.
- **method** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.

Returns

- **mu** (*float*) – Estimated mean of *x*.
- **EB** (*float*) – Size of error bar on mean of *x* (`EB > 0`). The confidence interval is `[mu - EB, mu + EB]`.

`mlpaper.mlpaper.`**get_test**(*x*, *lower=-inf*, *upper=inf*, *method='t'*)

Perform a statistical test to determine if the values in *x* are sampled from a distribution with a zero mean.

Parameters

- **x** (*ndarray, shape (n_samples,)*) – Array of independent observations.
- **lower** (*float*) – A priori known theoretical lower limit on unknown mean of *x*. For instance, for mean zero-one loss, `lower=0`.
- **upper** (*float*) – A priori known theoretical upper limit on unknown mean of *x*. For instance, for mean zero-one loss, `upper=1`.
- **method** (*{'t', 'bernstein', 'boot'}*) – Method to use statistical test.

Returns **pval** – p-value (in [0,1]) from statistical test on *x*.

Return type float

`mlpaper.mlpaper.`**loss_summary_table**(*loss_table*, *ref_method*, *pairwise_CI=False*, *confidence=0.95*, *method_EB='t'*, *limits={}*)

Build table with mean and error bar summaries from a loss table that contains losses on a per data point basis.

Parameters

- **loss_tbl** (*DataFrame, shape (n_samples, n_metrics * n_methods)*) – DataFrame with loss of each method according to each loss function on each data point. The rows are the data points in *y* (that is the index matches *log_pred_prob_table*). The columns are a hierarchical index that is the cartesian product of loss x method. That is, the loss of method foo's prediction of `y[5]` according to loss function bar is stored in `loss_tbl.loc[5, ('bar', 'foo')]`.
- **ref_method** (*str*) – Name of method that is used as reference point in paired statistical tests. This is usually some some of baseline method. *ref_method* must be found in the 2nd level of the columns of *loss_tbl*.
- **pairwise_CI** (*bool*) – If True, compute error bars on the mean of `loss - loss_ref` instead of just the mean of *loss*. This typically gives smaller error bars.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct error bar.
- **method_EB** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.
- **limits** (*dict of str to (float, float)*) – Dictionary mapping metric name to tuple with (lower, upper) which are the theoretical limits on the mean loss. For instance, zero-one loss should be `(0.0, 1.0)`. If entry missing, (-inf, inf) is used.

Returns **perf_tbl** – DataFrame with mean loss of each method according to each loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of loss x (mean, error bar, p-value). That is, `perf_tbl.loc['foo', 'bar']` is a pandas series with (mean loss of foo on bar, corresponding error bar, statistical sig) The statistical significance is a p-value from a two-sided hypothesis test on the hypothesis H0 that foo has the same mean loss as the reference method *ref_method*.

Return type DataFrame, shape (n_methods, n_metrics * 3)

`mlpaper.mlpaper.`**t_EB**(*x*, *confidence=0.95*)

Get t statistic based error bars on mean of *x*.

Parameters

- **x** (*array-like, shape (n_samples,)*) – Data points to estimate mean. Must not be empty or contain NaNs.
- **confidence** (*float*) – Confidence probability (in (0, 1)) to construct confidence interval from t statistic.

Returns **EB** – Size of error bar on mean (>= 0). The confidence interval is `[mean(x) - EB, mean(x) + EB]`. *EB* is inf when `len(x) <= 1`.

Return type float

`mlpaper.mlpaper.`**t_test**(*x*)

Perform a standard t-test to test if the values in *x* are sampled from a distribution with a zero mean.

Parameters **x** (*array-like, shape (n_samples,)*) – array of data points to test.

Returns **pval** – p-value (in [0,1]) from t-test on *x*.

Return type float

2.5 Performance Curves

`mlpaper.perf_curves.`**prg_curve**(*y_true*, *y_score*, *sample_weight=None*)

Compute precision recall gain curve with optional sample weight matrix. Similar to *recall_precision_curve*.

Parameters

- **y_true** (*ndarray of type bool, shape (n_samples,)*) – True targets of binary classification. Cannot be empty.
- **y_score** (*ndarray, shape (n_samples,)*) – Estimated probabilities or decision function. Must be finite.
- **sample_weight** (*None or ndarray of shape (n_samples, n_boot)*) – Sample weights. If *None*, all weights are one.

Returns

- **recall_gain** (*ndarray, shape (n_boot, n_thresholds)*) – The recall_gain. Each column is computed indepently by each column in *sample_weight*.
- **prec_gain** (*ndarray, shape (n_boot, n_thresholds)*) – The precision gain. Each column is computed indepently by each column in *sample_weight*.
- **thresholds** (*ndarray, shape (n_thresholds,)*) – Decreasing score values.

mlpaper.perf_curves.**recall_precision_curve**(*y_true*, *y_score*, *sample_weight=None*)

Compute recall precision curve with optional sample weight matrix. This has intentionally been named recall-precision rather than the traditional precision-recall.

Based on *sklearn.metrics.ranking.precision_recall_curve* except that it supports a matrix a different sample weights *sample_weight*. The name order has been switched to *recall_precision_curve* to be consistent with *roc_curve* because recall is typically placed on the x-axis. It computes the results indenpedently for each column of *sample_weight* in a vectorized way. This is useful when doing a fast boot strap analysis. It is also more robust to corner cases such as when only a single class is present in *y_true*.

Parameters

- **y_true** (*ndarray of type bool, shape (n_samples,)*) – True targets of binary classification. Cannot be empty.
- **y_score** (*ndarray, shape (n_samples,)*) – Estimated probabilities or decision function. Must be finite.
- **sample_weight** (*None or ndarray of shape (n_samples, n_boot)*) – Sample weights. If *None*, all weights are one.

Returns

- **recall** (*ndarray, shape (n_boot, n_thresholds)*) – The recall. Each column is computed indepently by each column in *sample_weight*.
- **precision** (*ndarray, shape (n_boot, n_thresholds)*) – The precision. Each column is computed indepently by each column in *sample_weight*.
- **thresholds** (*ndarray, shape (n_thresholds,)*) – Decreasing score values.

mlpaper.perf_curves.**roc_curve**(*y_true*, *y_score*, *sample_weight=None*)

Compute ROC curve with optional sample weight matrix.

Based on *sklearn.metrics.ranking.roc_curve* except that it supports a matrix a different sample weights *sample_weight*. It computes the results indenpedently for each column of *sample_weight* in a vectorized way. This is useful when doing a fast boot strap analysis. It is also more robust to corner cases such as when only a single class is present in *y_true*.

Parameters

- **y_true** (*ndarray of type bool, shape (n_samples,)*) – True targets of binary classification. Cannot be empty.

- **y_score** (*ndarray, shape (n_samples,)*) – Estimated probabilities or decision function. Must be finite.
- **sample_weight** (*None or ndarray of shape (n_samples, n_boot)*) – Sample weights. If *None*, all weights are one.

Returns

- **fpr** (*ndarray, shape (n_boot, n_thresholds)*) – The false positive rates. Each column is computed indepently by each column in *sample_weight*.
- **tpr** (*ndarray, shape (n_boot, n_thresholds)*) – The false positive rates. Each column is computed indepently by each column in *sample_weight*.
- **thresholds** (*ndarray, shape (n_thresholds,)*) – Decreasing score values.

2.6 Benchmarking for Regression

class mlpaper.regression.**JustNoise**

Class version of iid predictor compatible with sklearn interface. Same as `sklearn.dummy.DummyRegressor(strategy='mean')` but also keeps track of std to be able to accept `return_std=True`.

mlpaper.regression.**abs_loss**(*y*, *mu*, *std*)

Compute MAE of predictions vs true targets.

Parameters

- **y** (*ndarray, shape (n_samples,)*) – True targets for each regression data point. Typically of type *float*.
- **mu** (*ndarray, shape (n_samples,)*) – Predictive mean for each regression data point. Typically of type *float*. Must be of same shape as *y*.
- **std** (*ndarray, shape (n_samples,)*) – Predictive standard deviation for each regression data point. Typically of type *float*. Must be positive and of same shape as *y*. Ignored in this function.

Returns **loss** – Absolute error of target vs prediction. Same shape as *y*.

Return type ndarray, shape (n_samples,)

mlpaper.regression.**get_gauss_pred**(*X_train*, *y_train*, *X_test*, *methods*, *min_std=0.0*, *verbose=False*, *checkpointdir=None*)

Get the Gaussian prediction tables for each test point on a collection of regression methods.

Parameters

- **X_train** (*ndarray, shape (n_train, n_features)*) – Training set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_train** (*ndarray, shape (n_train,)*) – True training targets for each regression data point. Typically of type *float*. Must be of same length as *X_train*.
- **X_test** (*ndarray, shape (n_test, n_features)*) – Test set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **methods** (*dict of str to sklearn estimator*) – Dictionary mapping method name (*str*) to object that performs training and test. Object must follow the interface of sklearn estimators, that is, it has a `fit()` method and a `predict()` method that accepts the argument `return_std=True`.

- **min_std** (*float*) – Minimum value to floor the predictive standard deviation. Must be >= 0. Useful to prevent inf log loss penalties.
- **verbose** (*bool*) – If True, display which method being trained.
- **checkpointdir** (*str (directory)*) – If provided, stores checkpoint results using joblib for the train/test in case process interrupted. If None, no checkpointing is done.

Returns **pred_tbl** – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x moments. For exampe, `log_pred_prob_table.loc[5, 'foo']` is a pandas series with (mean, std deviation) prediction that method foo places on `y[5]`.

Return type DataFrame, shape (n_samples, n_methods * 2)

Notes

If a train/test operation is loaded from a checkpoint file, the estimator object in methods will not be in a fit state.

mlpaper.regression.**just_benchmark**(*X_train*, *y_train*, *X_test*, *y_test*, *methods*, *loss_dict*, *ref_method*, *min_std=0.0*, *pairwise_CI=False*, *method_EB='t'*, *limits={}*)

Simplest one-call interface to this package. Just pass it data and method objects and a performance summary DataFrame is returned.

Parameters

- **X_train** (*ndarray, shape (n_train, n_features)*) – Training set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_train** (*ndarray, shape (n_train,)*) – True training targets for each regression data point. Typically of type *float*. Must be of same length as *X_train*.
- **X_test** (*ndarray, shape (n_test, n_features)*) – Test set 2d feature array for classifiers. Each row is an indepedent data point and each column is a feature.
- **y_test** (*ndarray, shape (n_test,)*) – True test targets for each regression data point. Typically of type *float*. Cannot be empty. Must be of same length as *X_test*.
- **methods** (*dict of str to sklearn estimator*) – Dictionary mapping method name (*str*) to object that performs training and test. Object must follow the interface of sklearn estimators, that is, it has a `fit()` method and a `predict()` method that accepts the argument `return_std=True`.
- **loss_dict** (*dict of str to callable*) – Dictionary mapping loss function name to function that computes loss, e.g., *log_loss*, *square_loss*, ...
- **ref_method** (*str*) – Name of method that is used as reference point in paired statistical tests. This is usually some some of baseline method. *ref_method* must be found in *methods* dictionary.
- **min_std** (*float*) – Minimum value to floor the predictive standard deviation. Must be >= 0. Useful to prevent inf log loss penalties.
- **pairwise_CI** (*bool*) – If True, compute error bars on the mean of `loss - loss_ref` instead of just the mean of *loss*. This typically gives smaller error bars.
- **method_EB** (*{'t', 'bernstein', 'boot'}*) – Method to use for building error bar.
- **limits** (*dict of str to (float, float)*) – Dictionary mapping metric name to tuple with (lower, upper) which are the theoretical limits on the mean loss. For instance,

square loss on a bounded y domain of `(-1.0,1.0)` would give limits of `(0.0, 4.0)`. If entry missing, (-inf, inf) is used.

Returns **loss_summary** – DataFrame with mean loss of each method according to each loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of loss x (mean, error bar, p-value). That is, `perf_tbl.loc['foo', 'bar']` is a pandas series with (mean loss of foo on bar, corresponding error bar, statistical sig) The statistical significance is a p-value from a two-sided hypothesis test on the hypothesis H0 that foo has the same mean loss as the reference method *ref_method*.

Return type DataFrame, shape (n_methods, n_metrics * 3)

`mlpaper.regression.`**`log_loss`**(*y*, *mu*, *std*)

Compute log loss of Gaussian predictive distribution on target *y*.

Parameters

- **y** (*ndarray, shape (n_samples,)*) – True targets for each regression data point. Typically of type *float*.
- **mu** (*ndarray, shape (n_samples,)*) – Predictive mean for each regression data point. Typically of type *float*. Must be of same shape as *y*.
- **std** (*ndarray, shape (n_samples,)*) – Predictive standard deviation for each regression data point. Typically of type *float*. Must be positive and of same shape as *y*.

Returns **loss** – Log loss of Gaussian predictive distribution on target *y*. Same shape as *y*.

Return type ndarray, shape (n_samples,)

`mlpaper.regression.`**`loss_table`**(*pred_tbl*, *y*, *metrics_dict*)

Compute loss table from table of Gaussian predictions.

Parameters

- **pred_tbl** (*DataFrame, shape (n_samples, n_methods * 2)*) – DataFrame with predictive distributions. Each row is a data point. The columns should be hierarchical index that is the cartesian product of methods x moments. For exampe, `log_pred_prob_table.loc[5, 'foo']` is a pandas series with (mean, std deviation) prediction that method foo places on `y[5]`. Cannot be empty.
- **y** (*ndarray, shape (n_samples,)*) – True targets for each regression data point. Typically of type *float*.
- **metrics_dict** (*dict of str to callable*) – Dictionary mapping loss function name to function that computes loss, e.g., *log_loss*, *square_loss*, ...

Returns **loss_tbl** – DataFrame with loss of each method according to each loss function on each data point. The rows are the data points in *y* (that is the index matches *pred_tbl*). The columns are a hierarchical index that is the cartesian product of loss x method. That is, the loss of method foo's prediction of `y[5]` according to loss function bar is stored in `loss_tbl.loc[5, ('bar', 'foo')]`.

Return type DataFrame, shape (n_samples, n_metrics * n_methods)

`mlpaper.regression.`**`shape_and_validate`**(*y*, *mu*, *std*)

Validate shapes and types of predictive distribution against data and return the shape information.

Parameters

- **y** (*ndarray, shape (n_samples,)*) – True targets for each regression data point. Typically of type *float*.

- **mu** (*ndarray, shape (n_samples,)*) – Predictive mean for each regression data point. Typically of type *float*. Must be of same shape as *y*.
- **std** (*ndarray, shape (n_samples,)*) – Predictive standard deviation for each regression data point. Typically of type *float*. Must be positive and of same shape as *y*.

Returns **n_samples** – Number of data points (length of *y*)

Return type int

mlpaper.regression.**square_loss**(*y*, *mu*, *std*)
Compute MSE of predictions vs true targets.

Parameters

- **y** (*ndarray, shape (n_samples,)*) – True targets for each regression data point. Typically of type *float*.
- **mu** (*ndarray, shape (n_samples,)*) – Predictive mean for each regression data point. Typically of type *float*. Must be of same shape as *y*.
- **std** (*ndarray, shape (n_samples,)*) – Predictive standard deviation for each regression data point. Typically of type *float*. Must be positive and of same shape as *y*. Ignored in this function.

Returns **loss** – Square error of target vs prediction. Same shape as *y*.

Return type ndarray, shape (n_samples,)

2.7 Print with Advanced Scientific Formatting Tools

mlpaper.sciprint.**adjust_headers**(*headers*, *shifts*, *unit_dict*, *use_prefix=True*, *use_tex=False*)
Adjust the headers of a table generated by format_table to reflect the shift.

Parameters

- **headers** (*array-like of str, shape (n_metrics,)*) – List of metrics to adjust
- **shifts** (*dict of str to int*) – The used shift in log10 scale for each metric.
- **unit_dict** (*dict or str to str*) – Dictionary from metric name to associated unit symbol. Treat as unitless if entry is missing for a metric.
- **use_prefix** (*bool*) – If True, attempt to apply SI prefix to unit symbol for shift.
- **use_tex** (*bool*) – If True, adjust headers with TeX based formatting.

Returns **headers** – New header strings in same order as headers.

Return type list of str, shape (n_metrics,)

Notes

Requiring list *headers* is not redundant with dictionary *shifts* which contains the same entries as keys because we care about the order. Standard dictionaries in Python do not guarantee order.

mlpaper.sciprint.**all_same**(*L*)
Check if all elements in list are equal.

Parameters **L** (*array-like, shape (n,)*) – List of objects of any type.

Returns **y** – True if all elements are equal.

Return type bool

mlpaper.sciprint.**as_tuple_chk**(*x_dec*)
Convert *Decimal* to *DecimalTuple* and check finite.

Parameters **x_dec** (*Decimal*) – Input value in decimal.

Returns **x_tup** – Input converted to *DecimalTuple*.

Return type DecimalTuple

mlpaper.sciprint.**ceil_mod**(*x*, *mod*)
Do ceil in base mod instead of to nearest integer.

Parameters

- **x** (*int*) – Number to ceil.
- **mod** (*int*) – Positive number (*x* >= 1) to use as modulus.

Returns **y** – Smallest number `y >= x` such that `y % mod = 0`.

Return type int

mlpaper.sciprint.**create_decimal**(*x*, *digits*, *rounding='ROUND_HALF_UP'*)
Create *Decimal* object from *float* with desired significant figures.

Parameters

- **x** (*float*) – Value to convert to decimal.
- **digits** (*int*) – Number of signficant figures to keep in *x*, must be >= 1.
- **rounding** (*str*) – Rounding mode, must be one of the rounding modes accepted as in *decimal.Context.rounding*.

Returns **y** – Conversion of *x* to *Decimal*.

Return type Decimal

mlpaper.sciprint.**decimal_1ek**(*k*, *signed=False*)
Returns `10 ** k` or `-1 * 10 ** k` in *Decimal*.

Parameters

- **k** (*int*) – exponent for value.
- **signed** (*bool*) – If True, return negative.

Returns **y** – `10 ** k` or `-1 * 10 ** k` in *Decimal*.

Return type Decimal

mlpaper.sciprint.**decimal_all_finite**(*x_dec_list*)
Check if all elements in list of decimals are finite.

Parameters **x_dec_list** (*iterable of Decimal*) – List of decimal objects.

Returns **y** – True if all elements are finite.

Return type bool

mlpaper.sciprint.**decimal_eps**(*x_dec*)
Analog of eps (*np.spacing*) for *Decimal* objects.

Parameters **x_dec** (*Decimal*) – Input value in decimal.

Returns **y** – Smallest value that can be added to *x_dec*.

Return type Decimal

mlpaper.sciprint.**decimal_from_tuple**(*signed*, *digits*, *expo*)
Build *Decimal* objects from components of decimal tuple.

Parameters

- **signed** (*bool*) – True for negative values.
- **digits** (*iterable of ints*) – digits of value each in [0,10).
- **expo** (*int or {'F', 'n', 'N'}*) – exponent of decimal.

Returns **y** – corresponding decimal object.

Return type Decimal

mlpaper.sciprint.**decimal_to_dot**(*x_dec*)
Test if *Decimal* value has enough precision that it is defined to dot, i.e., its eps is <= 1.

Parameters **x_dec** (*Decimal*) – Input value in decimal.

Returns **y** – True if *x_dec* defined to dot.

Return type bool

Examples

```
>>> decimal_to_dot(Decimal('1.23E+1'))
True
>>> decimal_to_dot(Decimal('1.23E+2'))
True
>>> decimal_to_dot(Decimal('1.23E+3'))
False
```

mlpaper.sciprint.**decimalize**(*perf_tbl*, *err_digits=2*, *pval_digits=4*, *default_digits=5*, *EB_limit={}*)
Convert a performance table from *float* entries to *Decimal*.

Parameters

- **perf_tbl** (*DataFrame, shape (n_methods, n_metrics * 3)*) – DataFrame with curve/loss summary of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (summary, error bar, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo, corresponding error bar, statistical sig).
- **err_digits** (*int*) – Number of digits of error to keep for rounding in *Decimal* conversion: 1.2345 +/- 0.0671 is rounded to 1.235 +/- 0.068 when `err_digits=2`. The error is always rounded up, and the summary is rounded up on half. Must be >= 1.
- **pval_digits** (*int*) – Precision to keep in p-value when rounding to decimal: 0.001234 is rounded to 0.0013 when `pval_digits=4`. The p-value is always rounded up. Must be >= 1
- **default_digits** (*int*) – Number of digits to keep in estimate when error bar is 0, inf, nan, or beyond the error bar limit. Must be >= 1.

- **EB_limit** (*dict of str to int*) – Error bar limit in log10 scale for each column. If the `error > 10 ** EB_limit` then the error is treated as if `error = inf` since it is too large to be useful. This dictionary is optional. Can be positive or negative integer since in log10 scale.

Returns **perf_tbl_dec** – DataFrame with same rows and columns as *perf_tbl*, however the entires are now Decimal objects that have been rounded in accordance with the input options.

Return type DataFrame, shape (n_methods, n_metrics * 3)

`mlpaper.sciprint.`**digit_str**(*x_dec*)

Decimal to string with only digits (no decimal point, exponent, sign).

Parameters **x_dec** (*Decimal*) – Input value in *Decimal*.

Returns **y** – String of digits in *x_dec*.

Return type str

`mlpaper.sciprint.`**ensure_tuple_of_ints**(*L*)

This could possibly be done more efficiently with *tolist* if L is np or pd array, but will stick with this simple solution for now.

`mlpaper.sciprint.`**find_last_dig**(*num_str*)

Find index in string of number (possibly) with error bars immediately before the decimal point.

Parameters **num_str** (*str*) – String representation of a float, possibly with error bars in parens.

Returns **pos** – String index of digit before decimal point.

Return type int

Examples

```
>>> find_last_dig('5.555')
0
>>> find_last_dig('-5.555')
1
>>> find_last_dig('-567.555')
3
>>> find_last_dig('-567.555(45)')
3
>>> find_last_dig('-567(45)')
3
```

`mlpaper.sciprint.`**find_shift**(*mean_list*, *err_list*, *shift_mod=1*)

Find optimal decimal point shift to display the numbers in *mean_list* for display compactness.

Finds optimal shift of Decimal numbers with potentially varying significant figures and varying magnitudes to limit the length of the longest resulting string of all the numbers. This is to limit the length of the resulting column which is determined by the longest number. This function assumes the number will *not* be displayed in a fixed width font and hence the decimal point only adds a neglible width. Assumes all clipped and non-finite values have been removed from list.

Attempts to fulful three constraints: 1) All estimates displayed to dot after shifting 2) At least one estimate is >= 1 after shift to avoid space waste with 0s. 3) `shift % shift_mod == 0` If not all 3 are possible then requirement 2 is violated.

Parameters

- **mean_list** (*array-like of Decimal, shape (n,)*) – List of *Decimal* estimates to format. Assumes all non-finite and clipped values are already removed.
- **err_list** (*array-like of Decimal, shape (n,)*) – List of *Decimal* error bars. Must be of same length as *mean_list*.
- **shift_mod** (*int*) – Required modulus for output. This is usually 1 or 3. When an SI prefix is desired on the shift then a modulus of 3 is used. Must be >= 1.

Returns **best_shift** – Best shift of mean_list for compactness. This is number of digits to move point to right, e.g. `shift=3` => change 1.2345 to 1234.5

Return type int

Notes

This function is fairly inefficient and could be done implicitly, but it shouldn't be the bottleneck anyway for most usages.

mlpaper.sciprint.**floor_mod**(*x*, *mod*)

Do floor in base mod instead of to nearest integer.

Parameters

- **x** (*int*) – Number to floor.
- **mod** (*int*) – Positive number (*x* >= 1) to use as modulus.

Returns **y** – Largest number `y <= x` such that `y % mod = 0`.

Return type int

mlpaper.sciprint.**format_table**(*perf_tbl_dec*, *shift_mod=None*, *pad=True*, *crap_limit_max={}*, *crap_limit_min={}*, *non_finite_fmt={}*)

Format a performance table that is already in decimal form to one that is formatted with entries in string type.

Parameters

- **perf_tbl_dec** (*DataFrame, shape (n_methods, n_metrics * 3)*) – DataFrame with curve/loss summary of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (summary, error bar, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo, corresponding error bar, statistical sig). All entries *must* be of type *Decimal*.
- **shift_mod** (*int*) – Required modulus for output. This is usually 1 or 3. When an SI prefix is desired on the shift then a modulus of 3 is used. Must be >= 1. Use None for no shifting at all.
- **pad** (*bool*) – If True, pad resulting strings with spaces to make the decimal points align. If the resulting strings are TeX source, this will make the source more readable but not effect the appearence of the compiled TeX.
- **crap_limit_max** (*dict of str to int*) – Dictionary with the log10 max_clip for each column. This is optional.
- **crap_limit_min** (*dict of str to int*) – Dictionary with the log10 min_clip for each column. This is optional.
- **non_finite_fmt** (*dict of str to str*) – Display format when estimate is non-finite. For example, for latex looking output, one could use: `{'inf': r'\infty', '-inf': r'-\infty', 'nan': '--'}`.

Returns

- **perf_tbl_str** (*DataFrame, shape (n_methods, n_metrics * 2)*) – DataFrame with summary string of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (estimate with error, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo with error bar, statistical sig). All entries are of type string.
- **shifts** (*dict of str to int*) – The used shift in log10 scale for each metric.

`mlpaper.sciprint.`**`get_shift_range`**(*x_dec_list*, *shift_mod=1*)

Helper function to *find_shift* that find upper and lower limits to shift the estimates based on the constraints. This bounds the search space for the optimal shift.

Attempts to fulful three constraints: 1) All estimates displayed to dot after shifting 2) At least one estimate is >= 1 after shift to avoid space waste with 0s. 3) `shift % shift_mod == 0` If not all 3 are possible then requirement 2 is violated.

Parameters

- **x_dec_list** (*array-like of Decimal*) – List of *Decimal* estimates to format. Assumes all non-finite and clipped values are already removed.
- **shift_mod** (*int*) – Required modulus for output. This is usually 1 or 3. When an SI prefix is desired on the shift then a modulus of 3 is used. Must be >= 1.

Returns

- **min_shift** (*int*) – Minimum shift (inclusive) to consider to satisfy contraints.
- **max_shift** (*int*) – Maximum shift (inclusive) to consider to satisfy contraints.
- **all_small** (*bool*) – If True, it means constraint 2 needed to be violated. This could be used to flag warning.

`mlpaper.sciprint.`**`just_format_it`**(*perf_tbl_fp*, *unit_dict={}*, *shift_mod=None*, *crap_limit_max={}*, *crap_limit_min={}*, *EB_limit={}*, *non_finite_fmt={}*, *use_tex=False*, *use_prefix=True*)

One stop function call to format a results table and get the output as a string in readable human plain text or as LaTeX source.

Parameters

- **perf_tbl_fp** (*DataFrame, shape (n_methods, n_metrics * 3)*) – DataFrame with curve/loss summary of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (summary, error bar, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo, corresponding error bar, statistical sig). The entries should all be *float*.
- **unit_dict** (*dict or str to str*) – Dictionary from metric name to associated unit symbol. Treat as unitless if entry is missing for a metric.
- **shift_mod** (*int*) – Required modulus for output. This is usually 1 or 3. When an SI prefix is desired on the shift then a modulus of 3 is used. Must be >= 1. Use None for no shifting at all.
- **crap_limit_max** (*dict of str to int*) – Dictionary with the log10 max_clip for each column. This is optional.
- **crap_limit_min** (*dict of str to int*) – Dictionary with the log10 min_clip for each column. This is optional.

- **EB_limit** (*dict of str to int*) – Error bar limit in log10 scale for each column. If the `error > 10 ** EB_limit` then the error is treated as if `error = inf` since it is too large to be useful. This dictionary is optional. Can be positive or negative integer since in log10 scale.
- **non_finite_fmt** (*dict of str to str*) – Display format when estimate is non-finite. For example, for latex looking output, one could use: `{'inf': r'\infty', '-inf': r'-\infty', 'nan': '--'}`.
- **use_tex** (*bool*) – If True, adjust headers with TeX based formatting.
- **use_prefix** (*bool*) – If True, attempt to apply SI prefix to unit symbol for shift.

Returns **str_out** – String containing formatted table in plain text or LaTeX.

Return type str

Notes

For Pandas `use_tex=True`, LaTeX export requires `\usepackage{booktabs}` and proper aligning of the decimal point requires `\usepackage{siunitx}`.

`mlpaper.sciprint.`**pad_num_str**(*num_str_list*, *pad=' '*)

Pad strings of formatted numbers so they are aligned at the decimal point when displayed in a right aligned manner (which is typical for numeric data).

Parameters

- **num_str_list** (*array-like of str, shape (n,)*) – List of numbers already formatted as strings.
- **pad** (*str*) – Padding character, typically space. Must be length 1.

Returns **L** – List of padded strings.

Return type list of str, shape (n,)

Examples

```
>>> sp.pad_num_str(['-55.5', '1.12(34)', '0'], pad='~')
['-55.5~~~~~', '1.12(34)', '0~~~~~~~']
```

`mlpaper.sciprint.`**print_estimate**(*mu*, *EB*, *shift=0*, *min_clip=Decimal('-Infinity')*, *max_clip=Decimal('Infinity')*, *below_fmt='<{0:, f}'*, *above_fmt='>{0:, f}'*, *non_finite_fmt={}*)

Convert a mean and error bar pair in *Decimal* to a string.

Parameters

- **mu** (*Decimal*) – Value of estimate in *Decimal*. Mu must have enough precision to be defined to dot after shifting. Can be inf or nan.
- **EB** (*Decimal*) – Error bar on estimate in *Decimal*. Must be non-negative. It must be defined to same precision (quantum) as *mu* if *EB* is finite positive and *mu* is positive.
- **shift** (*int*) – How many decimal points to shift *mu* for display purposes. If *mu* is in meters and shift=3 than we display the result in mm, i.e., x1e3.
- **min_clip** (*Decimal*) – Lower limit clip value on estimate. If `mu < min_clip` then simply return `< min_clip` for string. This is used for score metric where a lower metric is simply on another order of magnitude to other methods.

- **max_clip** (*Decimal*) – Upper limit clip value on estimate. If `mu > max_clip` then simply return `> max_clip` for string. This is used for loss metric where a high metric is simply on another order of magnitude to other methods.
- **below_fmt** (*str (format string)*) – Format string to display when estimate is lower limit clipped, often: '<{0:,f}'.
- **above_fmt** (*str (format string)*) – Format string to display when estimate is upper limit clipped, often: '>{0:,f}'.
- **non_finite_fmt** (*dict of str to str*) – Display format when estimate is non-finite. For example, for latex looking output, one could use: `{'inf': r'\infty', '-inf': r'-\infty', 'nan': '--'}`.

Returns **std_str** – String representation of *mu* and *EB*. This is in format 1.234(56) for `mu=1.234` and `EB=0.056` unless there are non-finite values or a value has been clipped.

Return type str

mlpaper.sciprint.**print_pval**(*pval*, *below_fmt='<{0:,f}'*, *non_finite_fmt={}*)

Convert decimal p-value into string representation.

Parameters

- **pval** (*Decimal*) – Decimal p-value to represent as string. Must be in [0,1] or nan.
- **below_fmt** (*str (format string)*) – Format string to display when p-value is lower limit clipped, often: `'<{0:,f}'`.
- **non_finite_fmt** (*dict of str to str*) – Display format when estimate is non-finite. For example, for latex looking output, one could use: `{'nan': '--'}`.

Returns **pval_str** – String representation of p-value. If p-value is zero or minimum Decimal value allowable in precision of pval. We simply return clipped string, e.g. `'<0.0001'`, as value.

Return type str

mlpaper.sciprint.**str_print_len**(*x_str*)

Estimated width of formatted number of string when *not* displayed using a fixed width font. This is the number of characters not including `.` and `,` because they are assumed to be of negligible width.

Parameters **x_str** (*str*) – Already formatted number string.

Returns **str_len** – Length of string without negligible width characters `.` and `,`.

Return type int

mlpaper.sciprint.**table_to_latex**(*perf_tbl_str*, *shifts*, *unit_dict*, *use_prefix=True*)

Export performance table already converted to string entries to a single string of LaTeX source.

This function includes adjustement of headers to reflect shift and display units.

Parameters

- **perf_tbl_str** (*DataFrame, shape (n_methods, n_metrics * 2)*) – DataFrame with summary string of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (estimate with error, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo with error bar, statistical sig). All entries must be of type string.
- **shifts** (*dict of str to int*) – The used shift in log10 scale for each metric.
- **unit_dict** (*dict or str to str*) – Dictionary from metric name to associated unit symbol. Treat as unitless if entry is missing for a metric.

- **use_prefix** (*bool*) – If True, attempt to apply SI prefix to unit symbol for shift.

Returns **latex_str** – String containing LaTeX export of perf_tbl_str.

Return type str

Notes

Pandas LaTeX export requires `\usepackage{booktabs}` and proper aligning of the decimal point requires `\usepackage{siunitx}`.

mlpaper.sciprint.**table_to_string**(*perf_tbl_str*, *shifts*, *unit_dict*, *use_prefix=True*)

Export performance table already converted to string entries to a single string of nicely formatted output in human readable form.

This function includes adjustement of headers to reflect shift and display units.

Parameters

- **perf_tbl_str** (*DataFrame, shape (n_methods, n_metrics * 2)*) – DataFrame with summary string of each method according to each curve or loss function. The rows are the methods. The columns are a hierarchical index that is the cartesian product of metric x (estimate with error, p-value), where metric can be a loss or a curve summary: `full_tbl.loc['foo', 'bar']` is a pandas series with (metric bar on foo with error bar, statistical sig). All entries must be of type string.
- **shifts** (*dict of str to int*) – The used shift in log10 scale for each metric.
- **unit_dict** (*dict or str to str*) – Dictionary from metric name to associated unit symbol. Treat as unitless if entry is missing for a metric.
- **use_prefix** (*bool*) – If True, attempt to apply SI prefix to unit symbol for shift.

Returns **latex_str** – String containing nicely formatted output in human readable form.

Return type str

2.8 Utilities

mlpaper.util.**area**(*x_curve*, *y_curve*, *kind*)

Compute area under function in vectorized way.

Parameters

- **x_curve** (*ndarray, shape (n_boot, n_thresholds)*) – The sample points corresponding to the y values. Must be sorted.
- **y_curve** (*ndarray, shape (n_boot, n_thresholds)*) – Input array to integrate. Must be same size as *x_curve*. Operation performed independently for each column.
- **kind** (*{'linear', 'kind'}*) – Type of interpolation scheme to turn points into lines.

Returns **auc** – Area under curve. Has same length as *x_curve* has columns.

Return type ndarray, shape (n_boot,)

mlpaper.util.**cummax_strict**(*x*, *copy=True*)

Minimally increase array elements to make the array strictly increasing.

Parameters

- **x** (*ndarray, shape (n_samples,)*) – A list of points.

- **copy** (*bool*) – If False, modify x in place.

Returns **x** – A list of points that are now *strictly* sorted. If *x* was already sorted then the new points will be as miniminally changed as the floating point representation allows.

Return type ndarray, shape (n_samples,)

mlpaper.util.**epsilon_noise**(*x*, *default_epsilon=1e-10*, *max_epsilon=1.0*)

Add a small amount of noise to a vector such that the output vector has all unique values. The ordering of the resutiling vector remains the same: `argsort(output) = argsort(input)` if input values are unique.

Parameters

- **x** (*ndarray, shape (n_samples,)*) – Input vector to be noise corrupted. Must have all finite values.
- **default_epsilon** (*float*) – Default noise to add for singleton lists, musts be > 0.0.
- **max_epsilon** (*float*) – Maximum amount of noise corruption regardless of scale found in *x*.

Returns **x** – Noise correupted version of input. All values are unique with probability 1. The ordering is the same as the input if the inputs values are all unique.

Return type ndarray, shape (n_samples,)

mlpaper.util.**eval_step_func**(*x_grid*, *xp*, *yp*, *ival=None*, *assume_sorted=False*, *skip_unique_chk=False*)

Evaluate a stepwise function. Based on the ECDF class in statsmodels. The function is assumed to cadlag (like a CDF function).

This is a non-OOP equivalent to class: *statsmodels.distributions.empirical_distribution.StepFunction* with `side='right'` option to be like a CDF.

Parameters

- **x_grid** (*ndarray, shape (n_grid,)*) – Values to evaluate the stepwise function at.
- **xp** (*ndarray, shape (n_samples,)*) – Points at which the step function changes. Typically of type float.
- **yp** (*ndarray, shape (n_samples,)*) – The new values at each of the steps
- **ival** (*scalar or None*) – Initial value for step function, e.g., the value of the step function at -inf. If None, we just require that all *x_grid* values are after the first step.
- **assume_sorted** (*bool*) – Set to True is *xp* is alreaded sorted in increasing order. This skips sorting for computational speed.
- **skip_unique_chk** (*bool*) – Assume all values in *xp* are sorted and unique. Setting to True skips checking this condition for speed.

Returns **y_grid** – Step function defined by *xp* and *yp* evaluated at the points in *x_grid*.

Return type ndarray, shape (n_grid,)

mlpaper.util.**normalize**(*log_pred_prob*)

Normalize log probability distributions for classification.

Parameters **log_pred_prob** (*ndarray, shape (n_samples, n_labels)*) – Each row corresponds to a categorical distribution with unnormalized probabilities in log scale. Therefore, the number of columns must be at least 1.

Returns **log_pred_prob** – A row-wise normalized (`exp(log_pred_prob)` sums to 1 on each row) version of the input.

Return type ndarray, shape (n_samples, n_labels)

mlpaper.util.**one_hot**(*y*, *n_labels*)

Same functionality *sklearn.preprocessing.OneHotEncoder* but avoids extra dependency.

Parameters

- **y** (*ndarray of type int, shape (n_samples,)*) – Integers in range `[0, n_labels)` to be one-hot encoded.
- **n_labels** (*int*) – Number of labels, must be >= 1. This is not infered from *y* because some labels may not be found in small data chunks.

Returns **y_bin** – One hot encoding of *y*, with size `(len(y), n_labels)`

Return type ndarray of type bool, shape (n_samples, n_labels)

mlpaper.util.**remove_chars**(*x_str*, *del_chars*)

Utility to remove specified characters from string.

Parameters

- **x_str** (*str*) – Generic input string.
- **del_chars** (*str*) – String containing characters we would like to remove.

Returns **x_str** – Generic input string after removing characters in *del_chars*.

Return type str

mlpaper.util.**unique_take_last**(*xp*, *yp=None*)

Take unique points in a sorted list *xp*. When duplicates occur take the last element and its corresponding element in an auxilary list *yp*.

This function is useful for taking a set of points and making a proper step function from them. A step function is ambiguous when there are multiple points at the same x coordinate. Similar functionality can be obtained from *np.unique* but it takes the first rather than last element when duplicates occur.

Parameters

- **xp** (*ndarray, shape (n_samples,)*) – A sorted list of points.
- **yp** (*None or ndarray of shape (n_samples,)*) – Optional points that must be kept allong with the x points. If *xp* are points on the x-axis, then yp are the y coordinate points.

Returns

- **xp** (*ndarray, shape (m_samples,)*) – Input *xp* after removing extra points. m_samples <= n_samples.
- **yp** (*ndarray, shape (m_samples,)*) – Input *yp* after removing extra points. m_samples <= n_samples.

CHAPTER

THREE

CREDITS

3.1 Development lead

Ryan Turner (rdturnermtl)

3.2 Contributors

- Zafarali Ahmed (zafarali)

PYTHON MODULE INDEX

m

INDEX