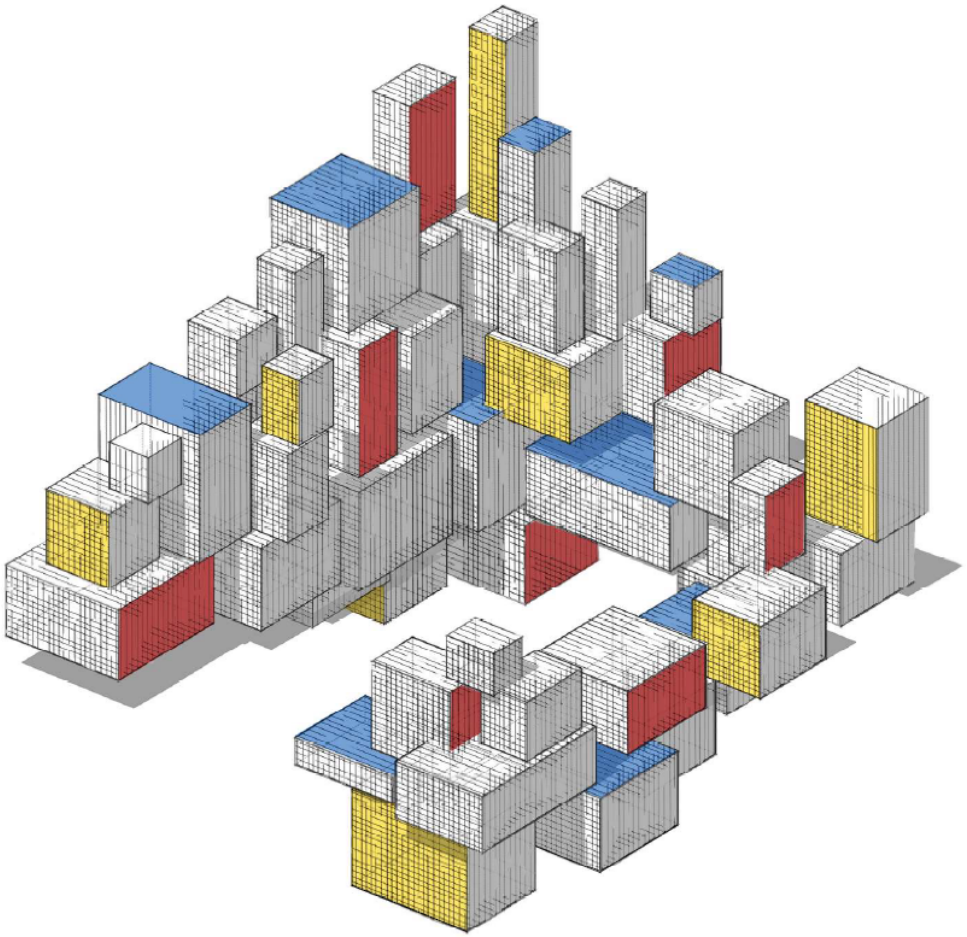
Boosting



Purpose

Boosting is a powerful idea that aims to improve the accuracy of any learning algorithm, especially when involving weak learners — simple prediction functions that exhibit performance slightly better than random guessing. Shallow decision trees typically yield weak learners.

In this lecture we discuss a number of boosting techniques:

- Boosting for regression
- Gradient boosting
- Adaboost

Boosting

Boosting can be used for both classification and regression problems.

Like the bagging method, boosting uses an ensemble of prediction functions.

However, prediction functions in boosting are learned sequentially. That is, each learner uses information from previous learners.

The idea is to start with a simple model (weak learner) g_0 for the data $\tau = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ and then to improve or "boost" this learner to a learner

$$g_1 := g_0 + h_1,$$

where the function h_1 is found by minimizing the training loss for $g_0 + h_1$ over all functions h in some class of functions $\mathcal{H}$.

Boosting

For example, $\mathcal{H}$ could be the set of prediction functions that can be obtained via a decision tree of maximal depth 2.

Given a loss function Loss, the function h_1 is thus obtained as the solution to the optimization problem

$$h_1 = \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \text{Loss}\left(y_i, g_0(\boldsymbol{x}_i) + h\left(\boldsymbol{x}_i\right)\right).$$

This process can be repeated for g_1 to obtain $g_2 = g_1 + h_2$, and so on, yielding the boosted prediction function

$$g_B(\boldsymbol{x}) = g_0(\boldsymbol{x}) + \sum_{b=1}^{B} h_b(\boldsymbol{x}).$$

Boosting

Instead of using the updating step $g_b = g_{b-1} + h_b$, one prefers to use the smooth updating step $g_b = g_{b-1} + \gamma\, h_b$, for some suitably chosen step-size parameter γ. This helps reduce overfitting.

Consider a simple regression setting, using the squared-error loss; thus, $\text{Loss}(y, \widehat{y}) = (y - \widehat{y})^2$.

In this case, it is common to start with $g_0(\boldsymbol{x}) = n^{-1} \sum_{i=1}^{n} y_i$, and each h_b for $b = 1, \ldots, B$ is chosen as a learner for the data set τ_b of residuals corresponding to g_{b-1}. That is, $\tau_b := \left\{ \left(\boldsymbol{x}_i, e_i^{(b)} \right) \right\}_{i=1}^{n}$, with

$$e_i^{(b)} := y_i - g_{b-1}(\boldsymbol{x}_i). \tag{1}$$

This leads to the following boosting procedure for regression with squared-error loss.

Algorithm 1: Regression Boosting with Squared-Error Loss

Input: Training set $\tau = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, the number of boosting rounds B, and a shrinkage step-size parameter γ.

Output: Boosted prediction function.

1 Set $g_0(\boldsymbol{x}) \leftarrow n^{-1} \sum_{i=1}^n y_i$.

2 **for** $b = 1$ **to** B **do**

3 Set $e_i^{(b)} \leftarrow y_i - g_{b-1}(\boldsymbol{x}_i)$ for $i = 1, \ldots, n$, and let

$$\tau_b \leftarrow \left\{\left(\boldsymbol{x}_i, e_i^{(b)}\right)\right\}_{i=1}^n.$$

4 Fit a prediction function h_b on the training data τ_b.

5 Set $g_b(\boldsymbol{x}) \leftarrow g_{b-1}(\boldsymbol{x}) + \gamma\, h_b(\boldsymbol{x})$.

6 **return** g_B.

The step-size parameter γ controls the speed of the fitting process. Specifically, for small values of γ, boosting takes smaller steps towards the training loss minimization.

Step-size Parameter

The step-size γ is of great practical importance, since it helps the boosting algorithm to avoid overfitting, as illustrated below.

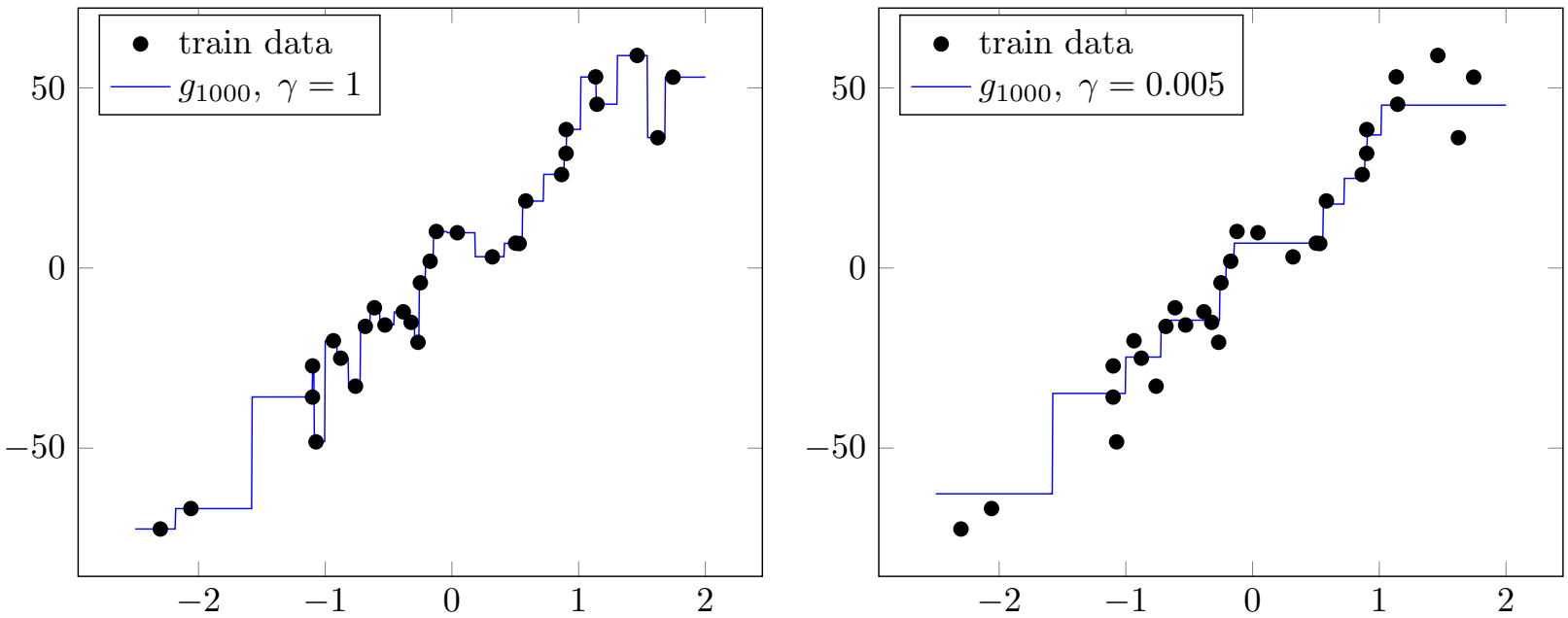


Figure: The left and the right panels show the fitted boosting regression model g_{1000} with $\gamma = 1.0$ and $\gamma = 0.005$, respectively. Note the overfitting on the left.

RegressionBoosting.py

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt


def TrainBoost(alpha,BoostingRounds,x,y):
    g_0 = np.mean(y)
    residuals  = y-alpha*g_0

    # list of basic regressor
    g_boost = []

    for i in range(BoostingRounds):
        h_i = DecisionTreeRegressor(max_depth=1)
        h_i.fit(x,residuals)
        residuals = residuals  - alpha*h_i.predict(x)
        g_boost.append(h_i)

    return g_0, g_boost
```

```
def Predict(g_0, g_boost,alpha, x):
    yhat = alpha*g_0*np.ones(len(x))
    for j in range(len(g_boost)):
        yhat = yhat+alpha*g_boost[j].predict(x)
    return yhat

np.random.seed(1)
sz = 30

x,y = make_regression(n_samples=sz, n_features=1,
n_informative=1,noise=10.0) # create data set

BoostingRounds = 1000
alphas = [1, 0.005]

for alpha in alphas: # boosting algorithm
    g_0, g_boost = TrainBoost(alpha,BoostingRounds,x,y)
    yhat = Predict(g_0, g_boost, alpha, x)

    tmpX =  np.reshape(np.linspace(-2.5,2,1000),(1000,1))
    yhatX = Predict(g_0, g_boost, alpha, tmpX)
    f = plt.figure()
    plt.plot(x,y,'*')
    plt.plot(tmpX,yhatX)
    plt.show()     # plot
```

Gradient Boosting

The parameter γ can be viewed as a step size made in the direction of the negative gradient of the squared-error training loss, because

$$-\left.\frac{\partial \operatorname{Loss}(y_i, z)}{\partial z}\right|_{z=g_{b-1}(\boldsymbol{x}_i)} = -\left.\frac{\partial\ (y_i - z)^2}{\partial z}\right|_{z=g_{b-1}(\boldsymbol{x}_i)} = 2(y_i - g_{b-1}(\boldsymbol{x}_i))$$

is two times the residual $e_i^{(b)}$ given in (1) that is used in Algorithm 1 to fit the prediction function h_b.

One can use a similar gradient descent method for any differentiable loss function. The resulting algorithm is called gradient boosting.

The main idea is to mimic a gradient descent algorithm:

- Calculate a negative gradient on n training points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$.
- Fit a simple model (such as a shallow decision tree) to approximate the gradient.
- Make a γ-sized step in the direction of the negative gradient.

Algorithm 2: Gradient Boosting

Input: Training set $\tau = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$, the number of boosting rounds B, a differentiable loss function $\text{Loss}(y, \widehat{y})$, and a step-size parameter γ.

Output: Gradient boosted prediction function.

1 Set $g_0(\boldsymbol{x}) \leftarrow 0$.

2 **for** $b = 1$ **to** B **do**

3 **for** $i = 1$ **to** n **do**

4 Evaluate the negative gradient of the loss at $(\boldsymbol{x}_i, y_i)$ via

$$r_i^{(b)} \leftarrow - \left. \frac{\partial \, \text{Loss}\,(y_i, z)}{\partial \, z} \right|_{z = g_{b-1}(\boldsymbol{x}_i)} \qquad i = 1, \ldots, n.$$

5 Approximate the negative gradient by solving

$$h_b = \underset{h \in \mathcal{H}}{\text{argmin}} \, \frac{1}{n} \sum_{i=0}^{n} \left(r_i^{(b)} - \left[g_{b-1}(\boldsymbol{x}_i) + h\,(\boldsymbol{x}_i) \right] \right)^2.$$

6 Set $g_b(\boldsymbol{x}) \leftarrow g_{b-1}(\boldsymbol{x}) + \gamma \, h_b(\boldsymbol{x})$.

7 **return** g_B

Example: Gradient Boosting for a Regression Tree

We continue with the basic bagging and random forest example for a regression tree.

Now, we use the gradient boosting estimator from Algorithm 2, as implemented in **sklearn**.

We use $\gamma = 0.1$ and perform $B = 100$ boosting rounds. As a prediction function h_b for $b = 1, \ldots, B$ we use small regression trees of depth at most 3. Note that such individual trees do not usually give good performance; that is, they are weak prediction functions.

We see (next) that the resulting boosting prediction function gives the R^2 score equal to 0.899, which is better than R^2 scores of simple decision tree (0.5754), the bagged tree (0.761), and the random forest (0.8106).

GradientBoostingRegression.py

```
import numpy as np
from sklearn.datasets import make_friedman1
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

n_points = 1000 # create regression problem
x, y =  make_friedman1(n_samples=n_points, n_features=15,
                       noise=1.0, random_state=100)

# split to train/test set
x_train, x_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.33, random_state=100)

# boosting sklearn
from sklearn.ensemble import GradientBoostingRegressor

breg = GradientBoostingRegressor(learning_rate=0.1,
            n_estimators=100, max_depth =3, random_state=100)
breg.fit(x_train,y_train)
yhat = breg.predict(x_test)
print("Gradient Boosting R^2 score = ",r2_score(y_test, yhat))
```

```
Gradient Boosting R^2 score =  0.8993055635639531
```

AdaBoost for Binary ($\{-1, 1\}$) Classification

The idea of AdaBoost is to create a sequence of prediction functions $g_0, g_1 = g_0 + h_1, g_2 = g_0 + h_1 + h_2, \ldots$ with final prediction function

$$g_B(\boldsymbol{x}) = g_0(\boldsymbol{x}) + \sum_{b=1}^{B} h_b(\boldsymbol{x}),$$

where each h_b is of the form $h_b(\boldsymbol{x}) = \alpha_b \, c_b(\boldsymbol{x})$, with $\alpha_b \in \mathbb{R}_+$ and where $c_b(\boldsymbol{x}) \in \{-1, 1\}$ is a classifier in some class C.

At each boosting iteration we solve the optimization problem

$$(\alpha_b, c_b) = \underset{\alpha \geqslant 0,\, c \in C}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \operatorname{Loss}\left(y_i, g_{b-1}(\boldsymbol{x}_i) + \alpha \, c(\boldsymbol{x}_i)\right),$$

where the loss function is defined as $\operatorname{Loss}(y, \widehat{y}) = \mathrm{e}^{-y\,\widehat{y}}$.

AdaBoost

The algorithm starts with a simple model $g_0 := 0$. Thus,

$$(\alpha_b, c_b) = \underset{\alpha \geqslant 0,\, c \in C}{\operatorname{argmin}} \sum_{i=1}^{n} \underbrace{\mathrm{e}^{-y_i\, g_{b-1}(\boldsymbol{x}_i)}}_{w_i^{(b)}} \mathrm{e}^{-y_i\, \alpha\, c(\boldsymbol{x}_i)} = \underset{\alpha \geqslant 0,\, c \in C}{\operatorname{argmin}} \sum_{i=1}^{n} w_i^{(b)}\, \mathrm{e}^{-y_i \alpha\, c(\boldsymbol{x}_i)},$$

where $w_i^{(b)} := \exp\{-y_i\, g_{b-1}(\boldsymbol{x}_i)\}$ does not depend on α or c. Hence,

$$\begin{aligned}(\alpha_b, c_b) &= \underset{\alpha \geqslant 0,\, c \in C}{\operatorname{argmin}}\ \mathrm{e}^{-\alpha} \sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{c(\boldsymbol{x}_i) = y_i\} + \mathrm{e}^{\alpha} \sum_{i=1}^{n} w_i^{(b)} \mathbb{I}\{c(\boldsymbol{x}_i) \neq y_i\} \\ &= \underset{\alpha \geqslant 0,\, c \in C}{\operatorname{argmin}}\ (\mathrm{e}^{\alpha} - \mathrm{e}^{-\alpha})\, \ell_\tau^{(b)}(c) + \mathrm{e}^{-\alpha}, \qquad (2)\end{aligned}$$

where

$$\ell_\tau^{(b)}(c) := \frac{\sum_{i=1}^{n} w_i^{(b)}\, \mathbb{I}\{c(\boldsymbol{x}_i) \neq y_i\}}{\sum_{i=1}^{n} w_i^{(b)}}$$

can be interpreted as the weighted zero–one training loss at iteration b.

AdaBoost

For any $\alpha \geqslant 0$, the program (2) is minimized by a classifier $c \in C$ that minimizes this weighted zero–one training loss; that is,

$$c_b(\boldsymbol{x}) = \operatorname*{argmin}_{c \in C} \ell_\tau^{(b)}. \tag{3}$$

Substituting (3) into (2) and solving for the optimal α gives

$$\alpha_b = \frac{1}{2} \ln \left(\frac{1 - \ell_\tau^{(b)}(c_b)}{\ell_\tau^{(b)}(c_b)} \right).$$

As soon as the AdaBoost algorithm finds the prediction function g_B, the final classification is delivered via

$$\operatorname{sign} \left(\sum_{b=1}^{B} \alpha_b \, c_b(\boldsymbol{x}) \right).$$

Algorithm 3: AdaBoost

Input: Training set $\tau = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, and the number of boosting rounds B.
Output: AdaBoost prediction function.

Set $g_0(\boldsymbol{x}) \leftarrow 0$.
for $i = 1$ **to** n **do**
    $w_i^{(1)} \leftarrow 1/n$
for $b = 1$ **to** B **do**
    Fit a classifier c_b on the training set τ by solving

$$c_b = \operatorname*{argmin}_{c \in C} \ell_\tau^{(b)}(c) = \operatorname*{argmin}_{c \in C} \frac{\sum_{i=1}^n w_i^{(b)} \mathbb{I}\{c(\boldsymbol{x}_i) \neq y_i\}}{\sum_{i=1}^n w_i^{(b)}}.$$

    Set $\alpha_b \leftarrow \frac{1}{2} \ln\left(\frac{1 - \ell_\tau^{(b)}(c_b)}{\ell_\tau^{(b)}(c_b)}\right)$. `// Update weights`
    for $i = 1$ **to** n **do**
        $w_i^{(b+1)} \leftarrow w_i^{(b)} \exp\{-y_i\, \alpha_b\, c_b(\boldsymbol{x}_i)\}$.
return $g_B(\boldsymbol{x}) := \sum_{b=1}^B \alpha_b\, c_b(\boldsymbol{x})$.

AdaBoost

- At the first step ($b = 1$), AdaBoost assigns an equal weight $w_i^{(1)} = 1/n$ to each training sample $(\boldsymbol{x}_i, y_i)$ in the set $\tau = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$.
- At each subsequent step $b > 1$, the weights of observations that were incorrectly classified by the previous boosting prediction function g_b are increased, and the weights of correctly classified observations are decreased.

  Due to the use of the weighted zero–one loss, the set of incorrectly classified training samples will receive an extra weight and thus have a better chance of being classified correctly.
- The step-size parameter α_b found by the AdaBoost algorithm can be viewed as an optimal step-size in the sense of training loss minimization.

  One can slow down the AdaBoost algorithm by setting α_b to be a fixed (small) value $\alpha_b = \gamma$ to reduce overfitting.

AdaBoost.py

```
from sklearn.datasets import make_blobs
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import zero_one_loss
import numpy as np

def ExponentialLoss(y,yhat):
    n = len(y)
    loss = 0
    for i in range(n):
        loss = loss+np.exp(-y[i]*yhat[i])
    loss = loss/n
    return loss

# create binary classification problem
np.random.seed(100)

n_points = 100 # points
x, y =  make_blobs(n_samples=n_points, n_features=5,  centers=2,
                       cluster_std=20.0, random_state=100)
y[y==0]=-1
```

```
# AdaBoost implementation
BoostingRounds = 1000
n = len(x)
W = 1/n*np.ones(n)

Learner = []
alpha_b_arr = []

for i in range(BoostingRounds):
    clf = DecisionTreeClassifier(max_depth=1)
    clf.fit(x,y, sample_weight=W)
    Learner.append(clf)

    train_pred = clf.predict(x)
    err_b = 0
    for i in range(n):
        if(train_pred[i]!=y[i]):
            err_b = err_b+W[i]

    err_b = err_b/np.sum(W)
    alpha_b = 0.5*np.log((1-err_b)/err_b)
    alpha_b_arr.append(alpha_b)
    for i in range(n):
        W[i] = W[i]*np.exp(-y[i]*alpha_b*train_pred[i])
```

```
yhat_boost = np.zeros(len(y))

for j in range(BoostingRounds):
    yhat_boost = yhat_boost+alpha_b_arr[j]*Learner[j].predict(x)


yhat = np.zeros(n)
yhat[yhat_boost>=0] = 1
yhat[yhat_boost<0] = -1
print("AdaBoost Classifier exponential loss = ", ExponentialLoss(y,
     yhat_boost))
print("AdaBoost Classifier zero--one loss = ",zero_one_loss(y,yhat))
```

```
AdaBoost Classifier exponential loss =  0.0042240136637771 42
AdaBoost Classifier zero--one loss =  0.0
```

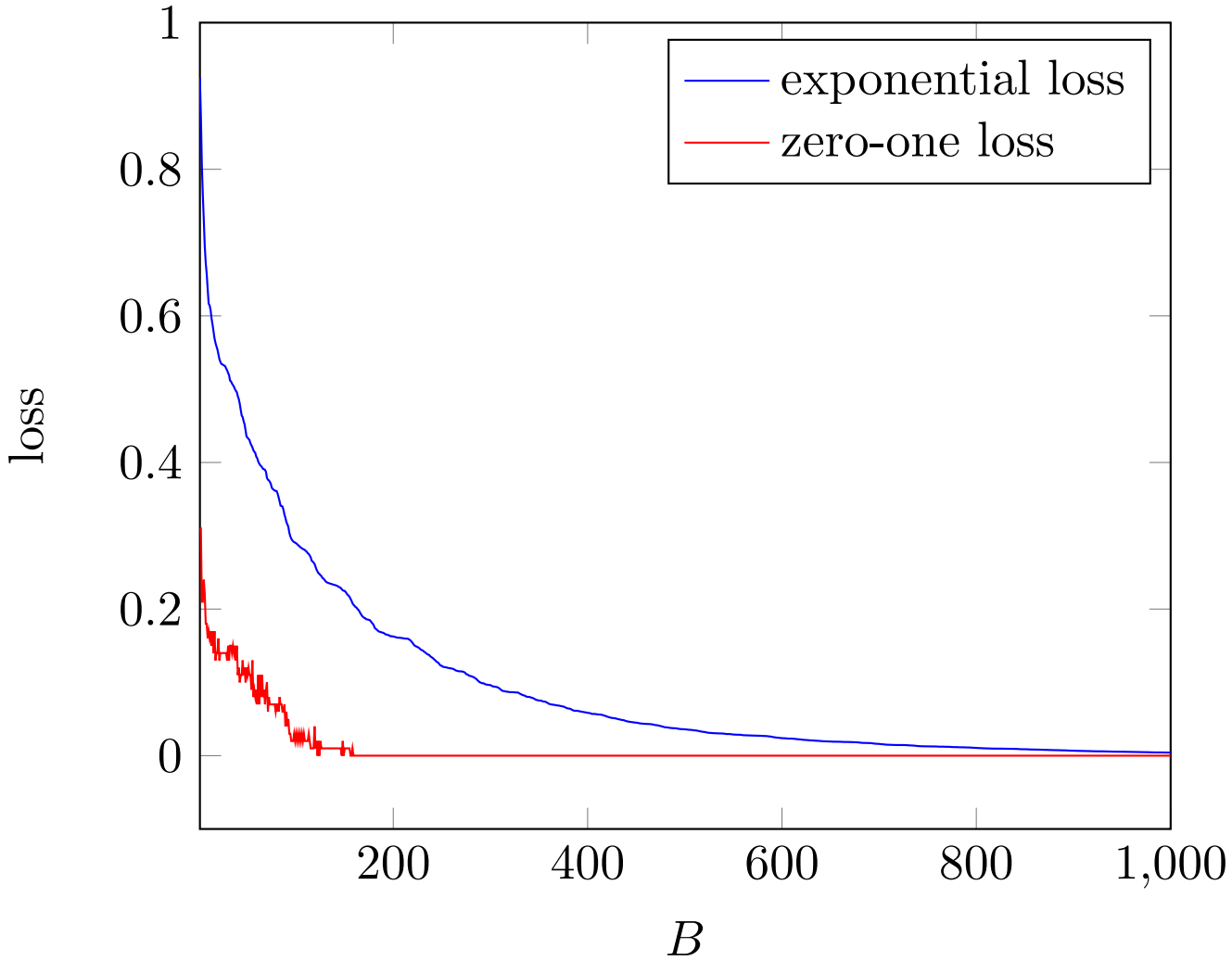


Figure: Exponential and zero–one training loss as a function of the number of boosting rounds B for a binary classification problem.