In-Depth Loss Functions

Trainer: Dr. Darshan Ingle.

1. Mean Squared Error

Helps us for cross-entropy loss.

Note: Error = Cost = Loss = Objective

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \bar{y}_i)^2$$

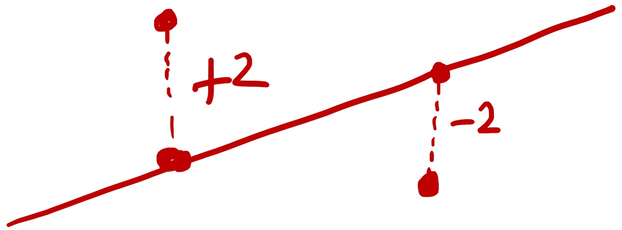
, where y_i = Actual values

$\bar{y}_i$ = Predicted values

MSE- Why is it squared?

$\because$ error to be +ve

eg: $+2 + (-2) = 0$

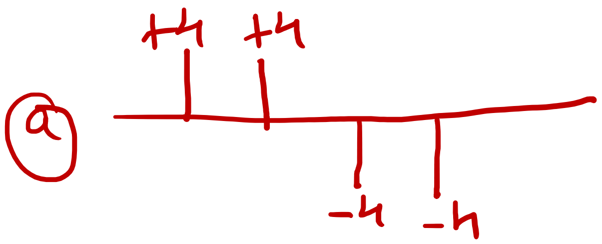


MSE- Footnote 1

Why squaring the differences?

Range: (−4) to (+4)

Spread less

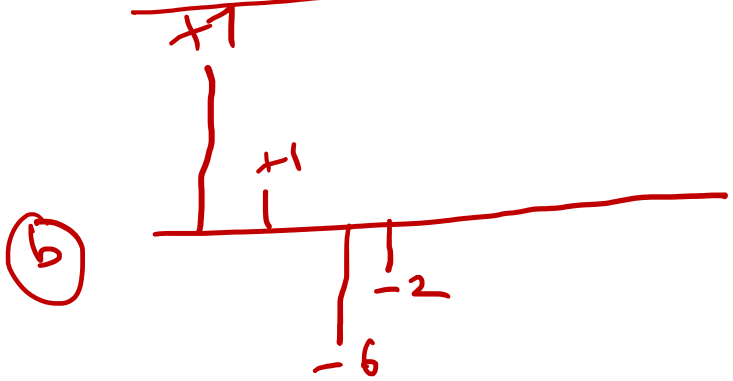


$$\frac{4+4+(-4)+(-4)}{4} = \frac{0}{4} = 0$$

This does not work.

Absolute Values:

$$\frac{|+4|+|+4|+|-4|+|-4|}{4} = \frac{16}{4} = 4 \text{ (Looks Good)}$$



$$\frac{|+7|+|+1|+|-6|+|-2|}{4} = \frac{7+1+6+2}{4} = \frac{16}{4} = 4 \ldots$$

Range: (+7) to (−6)

Spread more

MSE- Footnote 2

(a) $\sqrt{\dfrac{4^2+4^2+(-4)^2+(-4)^2}{4}} = \sqrt{\dfrac{64}{4}} = \sqrt{16} = \underline{\underline{4}}$

(b) $\sqrt{\dfrac{(7)^2+(1)^2+(-6)^2+(-2)^2}{4}} = \sqrt{\dfrac{90}{4}} = \underline{\underline{4.74}}$

2. Maximum Likelihood Estimation (MLE)

Probability

- https://www.youtube.com/watch?v=XepXtl9YKwc

MLE Formula

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$$

MLE vs MSE Conclusion

Max. the likelihood is same as minimizing the squared error.

$$l = -\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$ (-ve sign)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$ (does not have -ve sign)

3. Cross-Entropy / Log Loss

- If you are training a **binary classifier**, chances are you are using **binary cross-entropy** / **log loss** as your loss function.
- Have you ever thought about **what exactly does it mean** to use this loss function?
- The thing is, given the ease of use of today's libraries and frameworks, it is **very easy to overlook the true meaning of the loss function** used.

Cross-Entropy / Log Loss

→ Measure performance of Classification Model whose o/p is b/w 0 & 1.

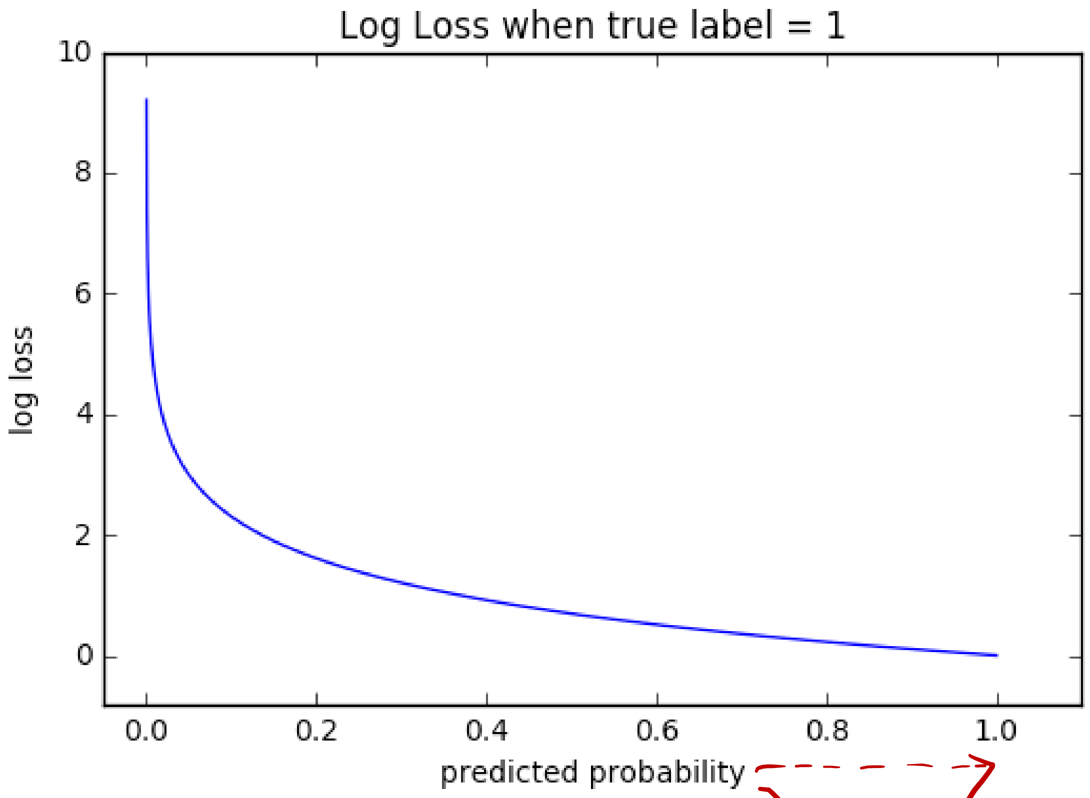
Pred $\frac{\text{huge}}{\text{diff}}$ Act ⟹ Cross_Entropy loss value ↑

y_pred = 0.12
y_act = 1

} bad ⟹ High loss

A perfect Model = Cross_ent. value as 0

Cross-Entropy / Log Loss



(is Dog = 1)

Log Loss → penalizes both the type of errors.

- Cross-entropy and log loss are slightly different depending on context, but in machine learning when calculating error rates between 0 and 1 they resolve to the same thing.

Cross-Entropy / Log Loss

Math

- In binary classification, where the number of classes M equals 2, cross-entropy can be calculated as:

$$-(y \log(p) + (1-y) \log(1-p))$$

- If M>2 (i.e. multiclass classification), we calculate a separate loss for each class label per observation and sum the result.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

- **Note:**
- M - number of classes (dog, cat, fish)
- log - the natural log
- y - binary indicator (0 or 1) if class label c is the correct classification for observation o
- p - predicted probability observation o is of class c

Cross-Entropy / Log Loss

Code:

```
def CrossEntropy(yHat, y):
    if y == 1:
      return -log(yHat)
    else:
      return -log(1 - yHat)
```

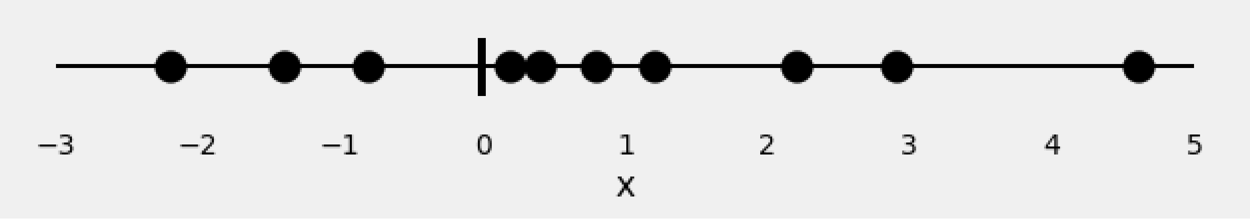
Cross-Entropy / Log Loss

A Simple Classification Problem

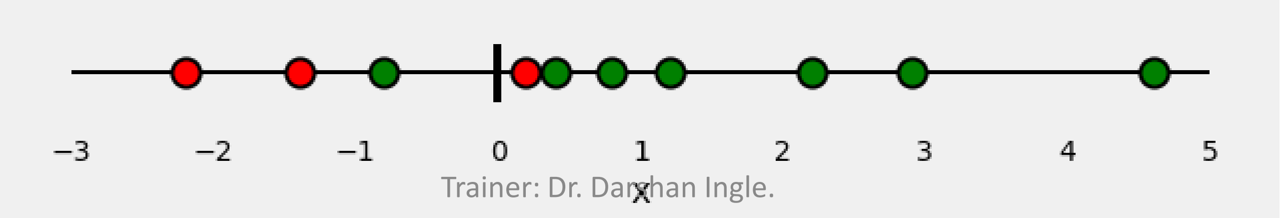
- Let’s start with 10 random points:

x = [-2.2, -1.4, -0.8, 0.2, 0.4, 0.8, 1.2, 2.2, 2.9, 4.6]

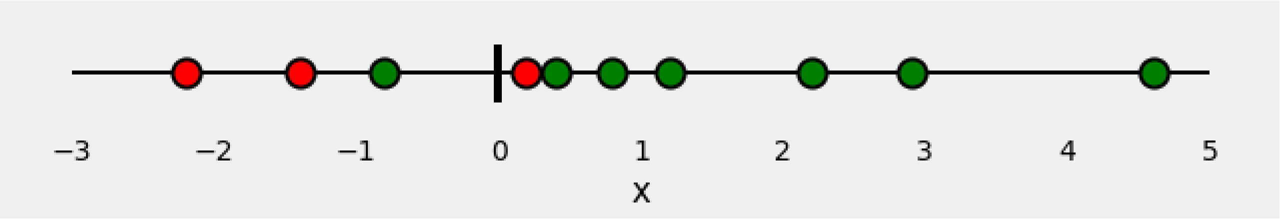
This is our only **feature**: ***x***.



Now, let’s assign some **colors** to our points: **red** and **green**. These are our **labels**.



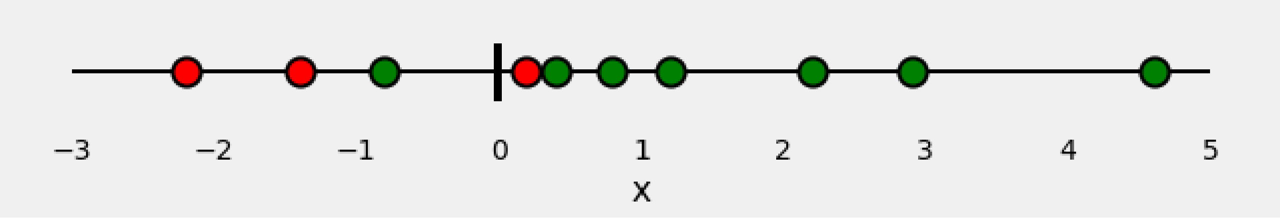
Cross-Entropy / Log Loss



So, our classification problem is quite straightforward: given our **feature *x***, we need to predict its **label**: **red** or **green**.

Since this is a **binary classification**, we can also pose this problem as: "**is the point green**" or, even better, "**what is the probability of the point being green**"? Ideally, **green points** would have a probability of **1.0** (of being green), while **red points** would have a probability of **0.0** (of being green).

Cross-Entropy / Log Loss



If we **fit a model** to perform this classification, it will **predict a probability of being green** to each one of our points. Given what we know about the color of the points, how can we **evaluate** how good (or bad) are the predicted probabilities? This is the whole purpose of the **loss function**! It should return **high values** for **bad predictions** and **low values** for **good predictions**.

For a **binary classification** like our example, the **typical loss function** is the **binary cross-entropy / log loss**.

Cross-Entropy / Log Loss

If you look this **loss function** up, this is what you'll find:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

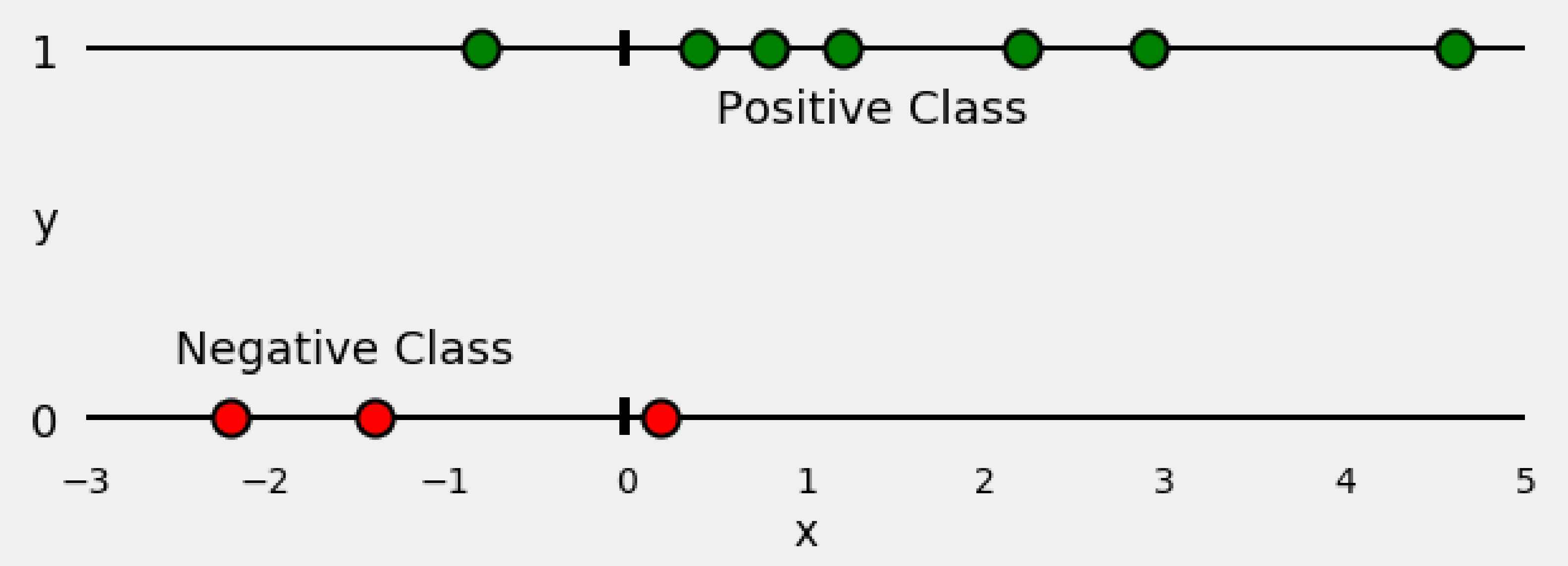
where **y** is the **label** (**1 for green** points and **0 for red** points) and **p(y)** is the predicted **probability of the point being green** for all **N** points.

Entropy ? Log of Prob?.

Cross-Entropy / Log Loss

- **Computing the Loss — the visual way**

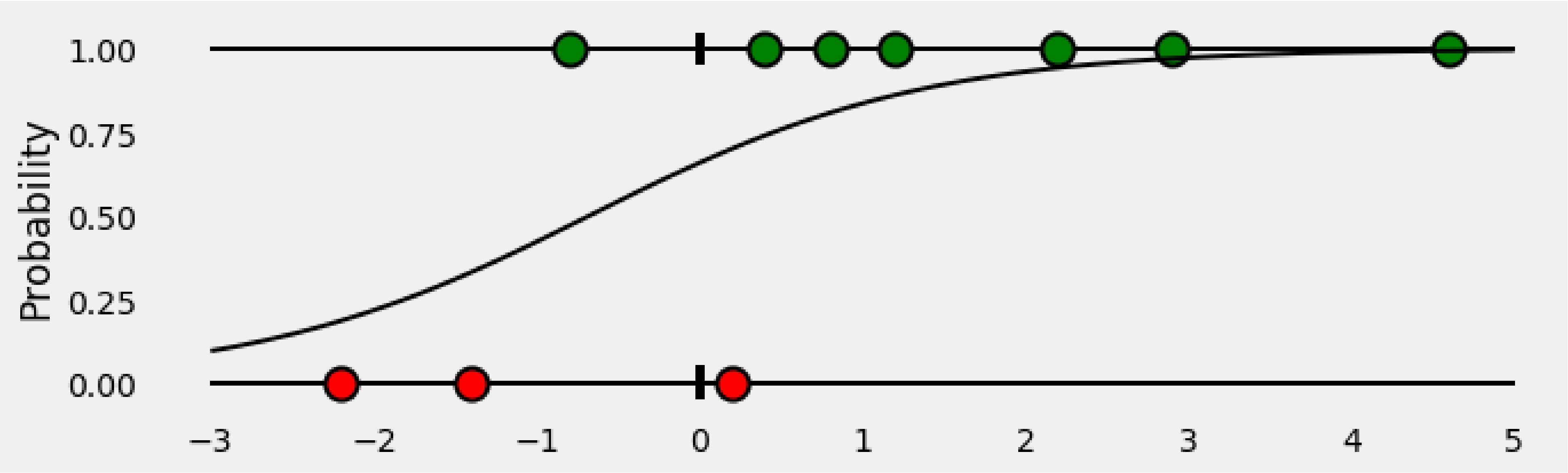
Split acc. to their classes : +ve or −ve



Cross-Entropy / Log Loss

Logistic Regression to classify our points.

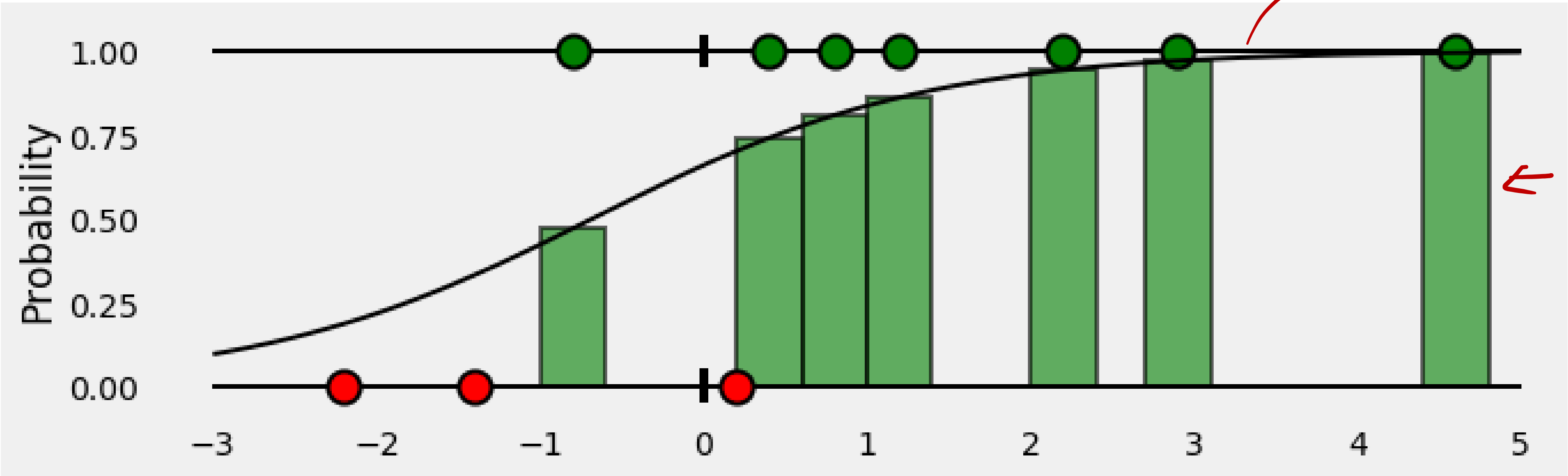
Sigmoid curve



Cross-Entropy / Log Loss

Predicted prob. by classifier?

Sigmoid Curve



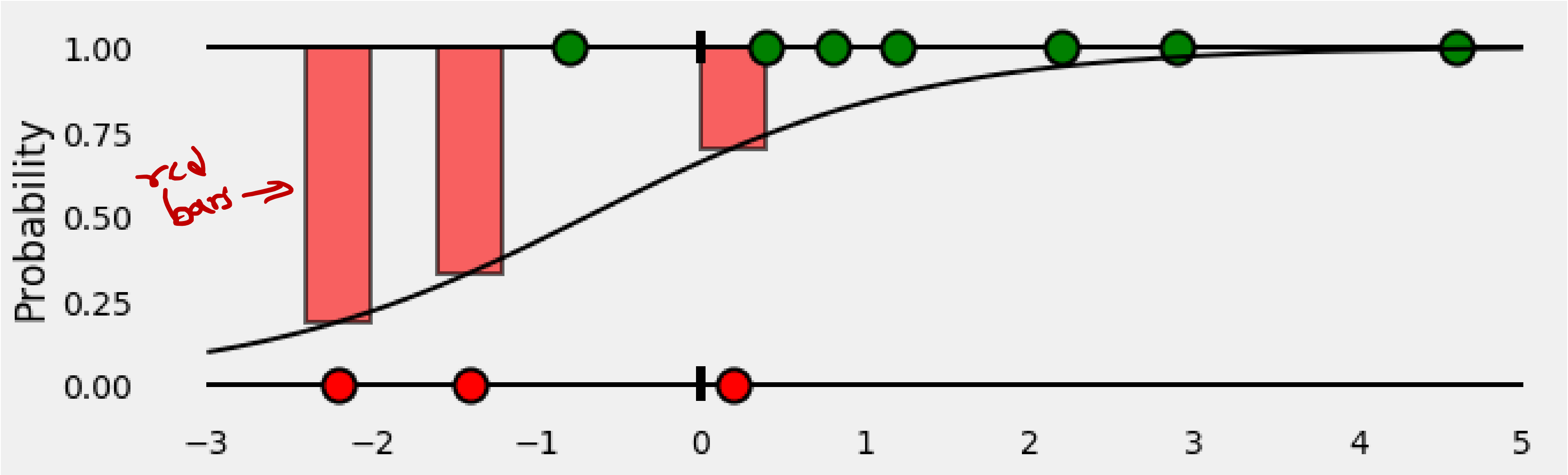
← Green bars

Cross-Entropy / Log Loss

-ve class?

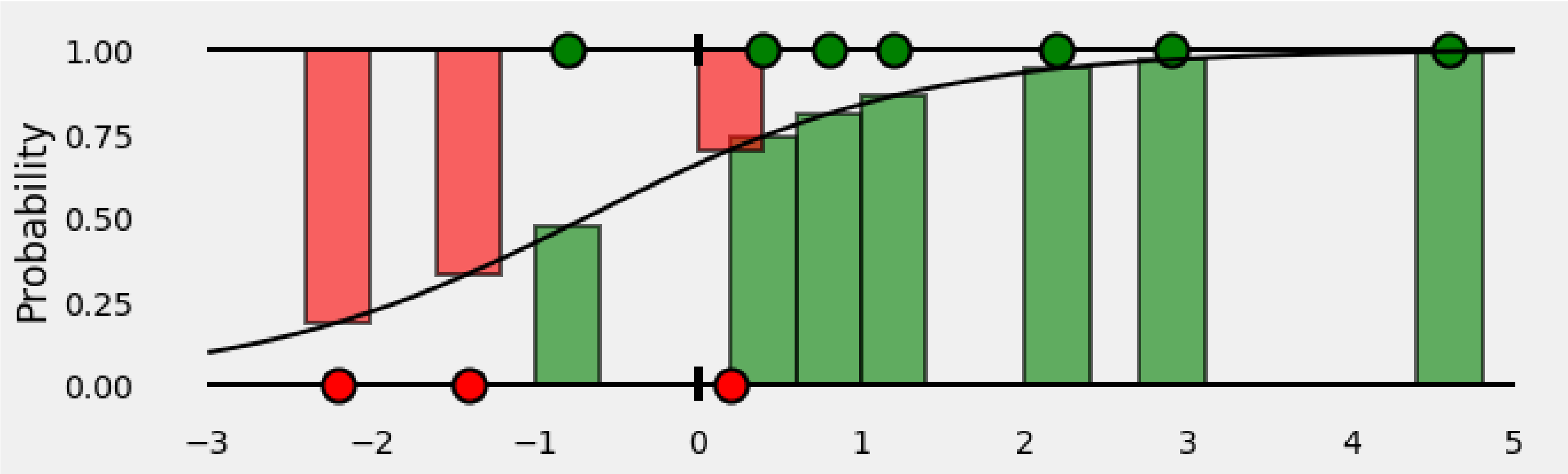
(y) = Green

(1-y) = Red



Cross-Entropy / Log Loss

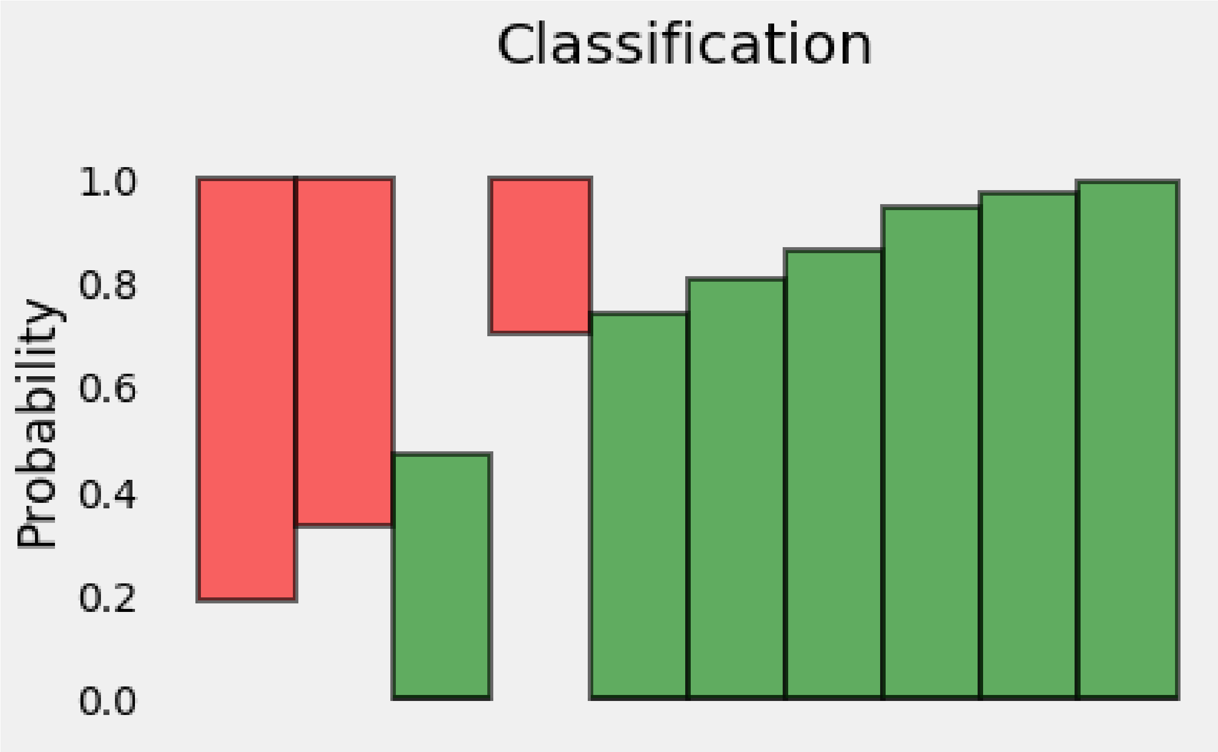
Look at it all once.



Evaluate Binary Cross-entropy / log loss?

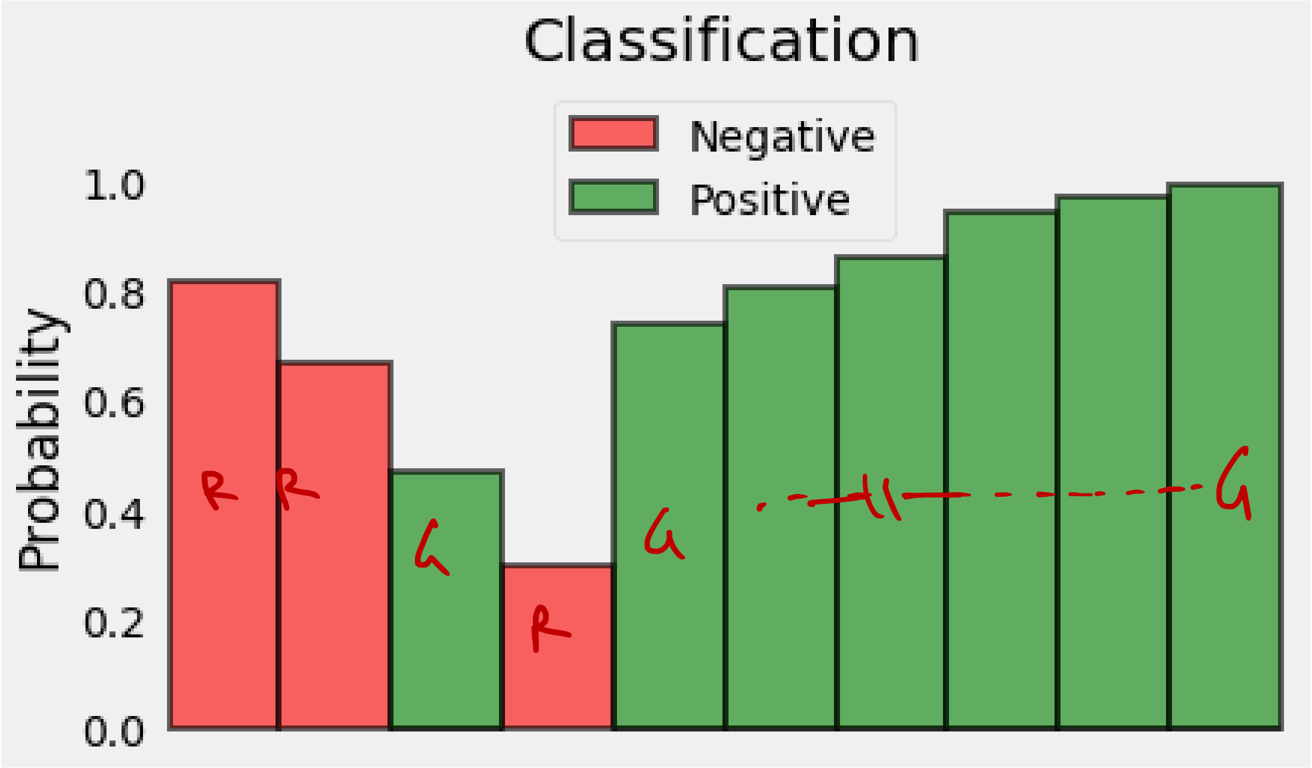
Cross-Entropy / Log Loss

ignored my X axis



Cross-Entropy / Log Loss

Repositioning the bars



Cross-Entropy / Log Loss

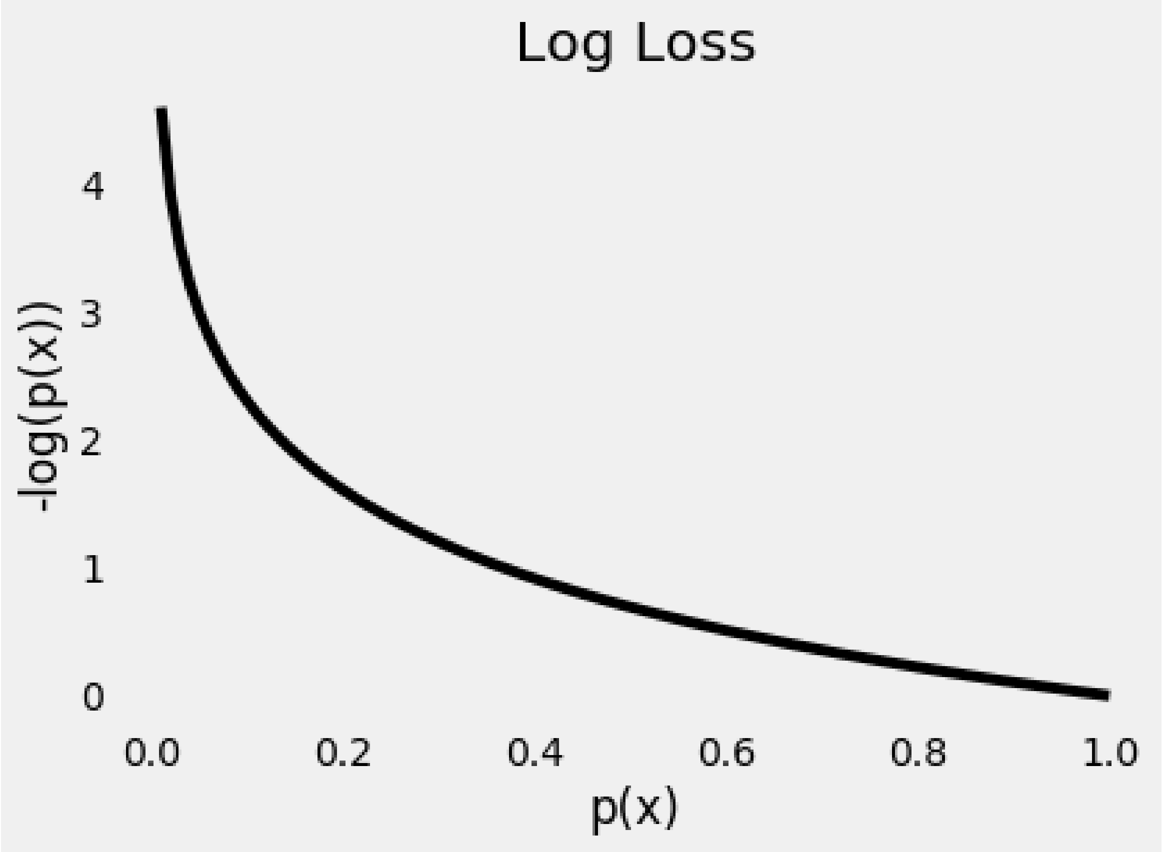
- Since we're trying to compute a **loss**, we need to penalize bad predictions, right? If the **probability** associated with the **true class** is **1.0**, we need its **loss** to be **zero**. Conversely, if that **probability is low**, say, **0.01**, we need its **loss** to be **HUGE**!

Claimed by researchers

- It turns out, taking the **(negative) log of the probability** suits us well enough for this purpose (*since the log of values between 0.0 and 1.0 is negative, we take the negative log to obtain a positive value for the loss*).

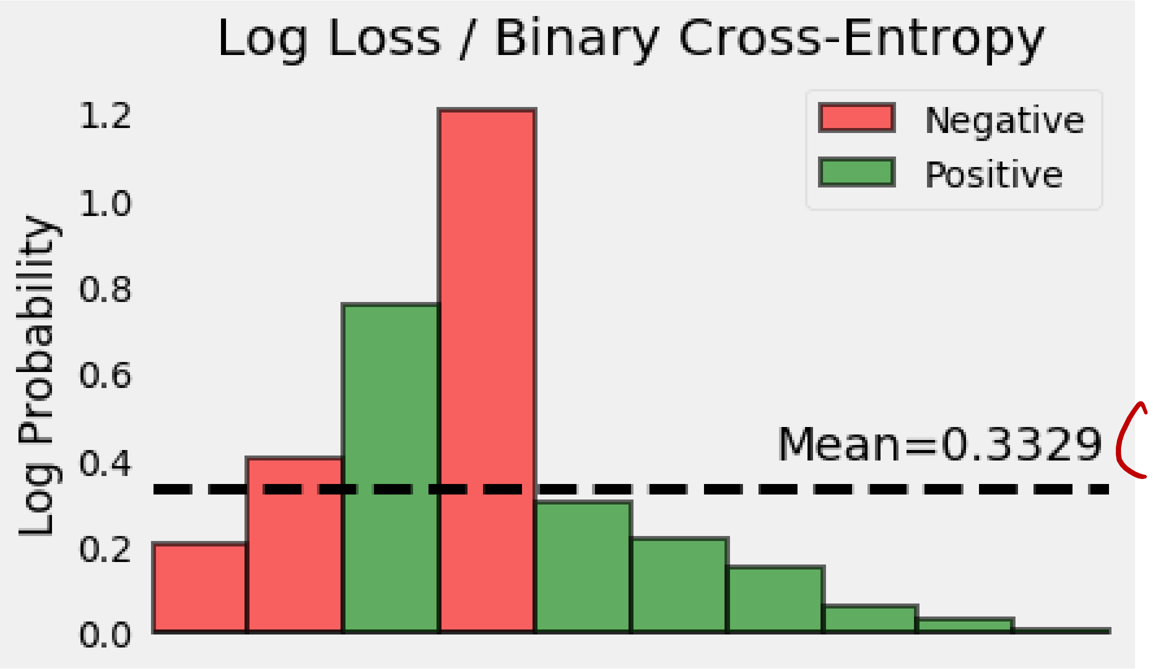
Cross-Entropy / Log Loss

- The plot below gives us a clear picture —as the **predicted probability** of the **true class** gets **closer to zero**, the **loss increases exponentially**:



Cross-Entropy / Log Loss

Take negative log of all probabilities



(Binary cross entropy / log loss for this example).

Cross-Entropy / Log Loss Code

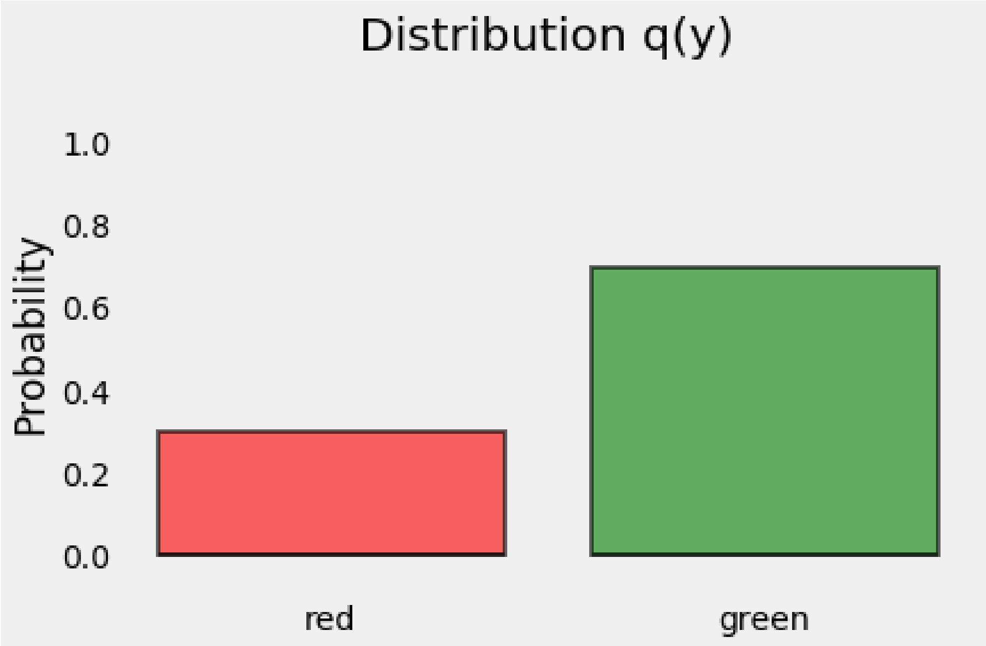
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
import numpy as np

x = np.array([-2.2, -1.4, -.8, .2, .4, .8, 1.2, 2.2, 2.9, 4.6])
y = np.array([0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0])

logr = LogisticRegression(solver='lbfgs')
logr.fit(x.reshape(-1, 1), y)

y_pred = logr.predict_proba(x.reshape(-1, 1))[:, 1].ravel()
loss = log_loss(y, y_pred)

print('x = {}'.format(x))
print('y = {}'.format(y))
print('p(y) = {}'.format(np.round(y_pred, 2)))
print('Log Loss / Cross Entropy = {:.4f}'.format(loss))
```

https://www.geeksforgeeks.org/differences-flatten-ravel-numpy/

Cross-Entropy / Log Loss

Distribution

Let's start with the distribution of our points. Since y represents the classes of our points (we have 3 red points and 7 green points), this is what its distribution, let's call it q(y), looks like:



Cross-Entropy / Log Loss

Entropy

- **Entropy** is a **measure of the uncertainty** associated with a given distribution **q(y)**.
- What if **all our points** were **green**? What would be the **uncertainty** of **that** distribution? **ZERO**, right? After all, there would be **no doubt about the color** of a point: it is **always** green! So, **entropy is zero**!

Cross-Entropy / Log Loss

Entropy

On the other hand, what if we knew exactly **half of the points** were **green** and the **other half**, **red**? That's the **worst case** scenario, right? We would have absolutely **no edge on guessing the color** of a point: it is totally **random**! For that case, entropy is given by the formula below (*we have two classes (colors)— red or green — hence, **2***):

$$H(q) = log(2)$$

Cross-Entropy / Log Loss

Entropy

For **every other case in between**, we can compute the **entropy of a distribution**, like our **q(y)**, using the formula below, where *C* is the number of classes:

$$H(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(q(y_c))$$

- So, if we *know* the **true distribution** of a random variable, we can compute its **entropy**. But, if that's the case, *why bother training a classifier* in the first place? After all, we **KNOW** the true distribution…
- But, what if we **DON'T**? Can we try to **approximate the true distribution** with some **other distribution**, say, **p(y)**? Sure we can! :-)

Cross-Entropy / Log Loss

Cross-Entropy

- Let's assume our **points follow** this **other** distribution **p(y)**. But we know they are **actually coming** from the **true** (*unknown*) distribution **q(y)**, right?
- If we compute **entropy** like this, we are actually computing the **cross-entropy** between both distributions:

$$H_p(q) = -\sum_{c=1}^{C} q(y_c) \cdot log(p(y_c))$$

- If we, somewhat miraculously, *match* **p(y)** *to* **q(y)** *perfectly*, the computed values for both **cross-entropy** *and* **entropy** *will match* as well.

Cross-Entropy / Log Loss

Cross-Entropy

Since this is likely never happening, **cross-entropy will have a BIGGER value than the entropy** computed on the true distribution.

$$H_p(q) - H(q) >= 0$$

It turns out, this difference between **cross-entropy** and **entropy** has a name…

Cross-Entropy / Log Loss

Kullback-Leibler Divergence

- The **Kullback-Leibler Divergence**,or "***KL Divergence***" for short, is a measure of **dissimilarity** between two distributions:

$$D_{KL}(q||p) = H_p(q) - H(q) = \sum_{c=1}^{C} q(y_c) \cdot [log(q(y_c)) - log(p(y_c))]$$

- This means that, the **closer p(y) gets to q(y)**,
  the **lower** the **divergence** and, consequently, the **cross-entropy**, will be.
- So, we need to find a good **p(y)** to use… but, this is what
  our **classifier** should do, isn't it?! **And indeed it does**! It looks for the **best possible p(y)**, which is the one that **minimizes the cross-entropy**.

Cross-Entropy / Log Loss

Loss Function

- During its training, the **classifier** uses each of the **N points** in its training set to compute the **cross-entropy** loss, effectively **fitting the distribution p(y)**! Since the probability of each point is 1/N, cross-entropy is given by:

$$q(y_i) = \frac{1}{N} \Rightarrow H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} log(p(y_i))$$

Cross-Entropy / Log Loss

Remember Figures above? We need to compute the **cross-entropy** on top of the *probabilities associated with the true class* of each point. It means using the **green bars** for the points in the **positive class** (*y=1*) and the **red *hanging* bars** for the points in the **negative class** (*y=0*) or, mathematically speaking:

$$y_i = 1 \Rightarrow log(p(y_i))$$
$$y_i = 0 \Rightarrow log(1 - p(y_i))$$

Cross-Entropy / Log Loss

The final step is to compute the **average** of all points in both classes, **positive** and **negative**:

$$H_p(q) = -\frac{1}{(N_{pos} + N_{neg})}\left[\sum_{i=1}^{N_{pos}} log(p(y_i)) + \sum_{i=1}^{N_{neg}} log(1 - p(y_i))\right]$$

Finally, with a little bit of manipulation, we can take any point, **either from the positive or negative classes**, under the same formula:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Voilà! We got back to the **original formula** for **binary cross-entropy / log loss** :-)

4. Hinge Loss

- Used for classification.

- **Code**

```
def Hinge(yHat, y):
    return np.max(0, 1 - yHat * y)
```

5. Huber Loss

- Typically used for regression.
- It’s less sensitive to outliers than the MSE as it treats error as square only inside an interval.

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\,|(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

Code

```
def Huber(yHat, y, delta=1.):
    return np.where(np.abs(y-yHat) < delta,.5*(y-yHat)**2 , delta*(np.abs(y-yHat)-0.5*delta))
```