Boosting Notes

greeness

August 6, 2013

1 Background

Let $S = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$ be a set of training instances where each instance x_i belongs to a domain and each label $y_i \in \{-1, +1\}$. Assume we also we have a set of classifiers $k_1, \cdots, k_L$, each of them corresponds to an individual feature of x; in the boosting literature, these would be called weak or base hypotheses. We study the problem of approximate the y_i's using a linear combination of features. We are interested in finding a vector of parameters α such that

$$C(x_i) = \sum_{j=1}^{L} \alpha_j k_j(x_i)$$

is a "good approximate" of y_i. For classification problem, a natural goal to try to match the sign of $k_j(x_i)$ to to y_i, that is, to attempt to minimize

$$\sum_{i=1}^{N} I(y_i C(x_i) \leq 0),$$

where $I(\pi)$ is 1 if π is true and 0 otherwise.

However minimization of the number of classification errors is an intractable problem. It is therefore advantageous to instead minimize some relaxation.

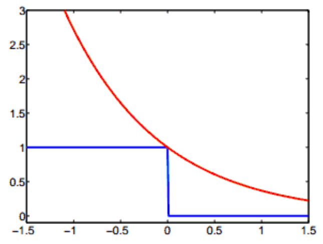

Figure 1: The figure shows 0-1 error versus the exponential loss (in red). It's easily seen that the 0-1 error is upper bounded by the exponential loss.

2 AdaBoost

AdaBoost or Adaptive Boosting, is an algorithm used for generating strong classifiers out of weak classifiers. For a given input x each expert classifier k_j can emit an opinion $k_j(x) \in \{-1, 1\}$, and the final decision of the committee K of experts is $\text{sign}(C(x))$, the sign the weighted sum of expert opinions, where

$$C(x) = \sum_{j=1}^{L} \alpha_j k_j(x),$$

where constants α_j is the weight we assign to the opinion of the jth expert in the committee.

The process of AdaBoost consists three steps: 1) scouting prospective experts, 2) drafting them, and 3) assigning a weight to their contribution to the team [1].

2.1 Notation

- x: input; $y \in \{-1, 1\}$: output.
- $C(x)$: the committee's output given input x.
- $k_j(x)$: the j-th classifier (expert) in the committee.
- α_j: the weight of the j-th classifier.
- i in $1 \cdots N$, the index of instances.
- j in $1 \cdots L$, the index of classifiers.
- m in $1 \cdots M$, the index of iterations.
- E, the total error defined as the exponential loss.
- $w_i^{(m)}$, the weight assigned to the i-th data instance at iteration m.

2.2 Scouting

Scouting is done by testing the classifiers in the pool using a training set T of N multidimensional data points x_i. For each point x_i we have a row i built in matrix with misses (with a 1) and hits (with a 0) of each classifier. Column j is reserved for the $j - th$ classifier in the pool:

The main idea of AdaBoost is to proceed systematically by extracting one classifier from the pool in each of M iterations. At beginning, all elements (α_j) are assigned the same weight (e.g., 1). As the drafting progresses, the more difficult examples, that is, those where the committee still perform badly, are assigned larger and larger weights. The best "team players" are those which can provide new insights to the committee. Classifiers being drafted should complement each other in an optimal way.

Table 1: Scout table of classifiers

	1	2	$\cdots$	L
x_1	0	1	$\cdots$	1
x_2	0	0	$\cdots$	1
x_3	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$		$\vdots$
x_N	0	0	$\cdots$	0

2.3 Drafting

In each iteration, we need to rank all classifiers, so that we can select the current best out of the pool. At the m-th iteration we have already included $m-1$ classifiers in the committee and we want to draft the next one. The current linear combination of classifiers is

$$C_{m-1}(x) = \sum_{j=1}^{m-1} \alpha_j k_j(x)$$

and we want to extend it to

$$C_m(x) = C_{m-1}(x) + \alpha_m k_m(x).$$

At the first iteration ($m = 1$, C_0 is the zero function. We define the total cost, or total error, of the extended classifier as the exponential loss

$$E = \sum_{i=1}^{N} e^{-y_i C_m(x_i)} = \sum_{i=1}^{N} e^{-y_i (C_{m-1}(x_i) + \alpha_m k_m(x_i))}$$

where $y_i \in \{-1, 1\}$.

If the committee's opinion (the sign) is different (a miss) with the label, e.g. $C_m(x_i) = \beta, y_i = -1$ where $\beta > 0$ or vise versa,

$$-y_i C_m = -(-1 \cdot \beta) = \beta,$$

the expert pool is charged by a cost e^{β}.

If the committee's opinion (the sign) is the same (a hit) as the label, e.g. $C_m(x_i) = -\beta, y_i = -1$ or vise versa,

$$-y_i C_m = -(-1 \cdot -\beta) = -\beta,$$

the expert pool is charged by a cost $e^{-\beta}$. Note that $e^{-\beta} < e^{\beta}$, so that misses are more heavily penalized than hits.

Since our intention is to draft k_m we rewrite the above expression as

$$E = \sum_{i=1}^{N} w_i^{(m)} e^{-y_i \alpha_m k_m(x_i)}, \tag{1}$$

where

$$w_i^{(m)} = e^{-y_i C_{m-1}(x_i)}, \tag{2}$$

If on instance i, the current committee performs well, $w_i^{(m)}$ goes to zero. Otherwise, $w_i^{(m)}$ will blow and therefore this instance will obtain a large importance weight. So after normalization, $w_i^{(m)}$ can be seen as the mistake probability [3].

In the first iteration $w_i^{(1)} = 1$ for $i = 1, 2, \cdots, N$. During later iterations, the vector $w^{(m)}$ represents the weight assigned to each data point in the training set at iteration m. We can split the sum in Eq. 1 into two sums

$$\begin{aligned} E &= \sum_{y_i = k_m(x_i)} w_i^{(m)} e^{-\alpha_m} + \sum_{y_i \neq k_m(x_i)} w_i^{(m)} e^{\alpha_m} & (3) \\ &= \mu^+ e^{-\alpha_m} + \mu^- e^{\alpha_m} & (4) \end{aligned}$$

which means that the total cost is the weighted cost of all hits (correct) plus the weighted cost of all misses (error). μ^+ also stands for the positive correlation between the feature and all instances; while μ^- stands for the negative correlation between the feature and all instances.

Minimizing E is equivalent to minimizing $e^{\alpha_m} E$ for a fixed α_m,

$$\begin{aligned} e^{\alpha_m} E &= \mu^+ + \mu^- e^{2\alpha_m} & (5) \\ &= (\mu^+ + \mu^-) + \mu^-(e^{2\alpha_m} - 1) & (6) \\ &= \mu + \mu^-(e^{2\alpha_m} - 1) & (7) \end{aligned}$$

$\mu = \mu^+ + \mu^-$ is the total sum of the weights of all data points, that is, a constant in the current iteration. The right hand side of the equation is minimized when at the m-th iteration we pick the classifier with the lowest total cost μ^- (we know $e^{2\alpha_m} > 1$). The next Draftee, k_m, should be the one with the lowest penalty given the current set of weights.

2.4 Weighting

Having picked the m-th member of the committee we need to determine its weight α_m. From Eq 4 we see that (note $de^x/dx = e^x$)

$$\frac{dE}{d\alpha_m} = -\mu^+ e^{-\alpha_m} + \mu^- e^{\alpha_m}$$

Setting the above equation to 0 and multiplying by e^{α_m} we obtain

$$-\mu^- + \mu^- e^{2\alpha_m} = 0$$

The optimal α_m^* is thus:

$$\alpha_m^* = \frac{1}{2} \ln\left(\frac{\mu^+}{\mu^-}\right) = \frac{1}{2} \ln\left(\frac{1 - e_m}{e_m}\right),$$

where $e_m = \mu^-/\mu$, is the percentage rate of error given the weights of the data points. Since the percentage is invariant if we normalize the weight, normalizing the weights of data at each iteration is irrelevant.

Also note that as long as error rate $e_m < 1/2$, $\frac{1-e_m}{em} > 1$, so that α_m^* is positive.

We proceed to get the recursive update equations:

$$\begin{aligned} e^{-\alpha_m^*} &= e^{-\frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right)} \\ &= (e^{\ln\left(\frac{1-e_m}{e_m}\right)})^{-\frac{1}{2}} \\ &= \sqrt{\frac{e_m}{1-e_m}} \end{aligned}$$

$$\begin{aligned} e^{\alpha_m^*} &= e^{\frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right)} \\ &= (e^{\ln\left(\frac{1-e_m}{e_m}\right)})^{\frac{1}{2}} \\ &= \sqrt{\frac{1-e_m}{e_m}} \end{aligned}$$

$$\begin{aligned} w_i^{(m+1)} &= e^{-y_i C_m(x_i)} \\ &= e^{-y_i(C_{m-1}(x_i)+\alpha_m^* k_m(x_i))} \\ &= e^{-y_i C_{m-1}(x_i)} \cdot e^{-y_i \alpha_m^* k_m(x_i)} \\ &= w_i^{(m)} e^{-y_i \alpha_m^* k_m(x_i)} \end{aligned}$$

$$w_i^{(m+1)} = \begin{cases} w_i^{(m)} e^{-\alpha_m^*} = w_i^{(m)} \sqrt{\frac{e_m}{1-e_m}}, & y_i = k_m(x_i) \quad \textit{hit} \\ w_i^{(m)} e^{\alpha_m^*} = w_i^{(m)} \sqrt{\frac{1-e_m}{e_m}}, & y_i \neq k_m(x_i) \quad \textit{miss} \end{cases} \tag{8}$$

If we take the 2nd order derivative with respective to α_m,

$$\begin{aligned} \frac{d^2E}{d\alpha_m^2} &= \frac{d(-\mu^+ e^{-\alpha_m} + \mu^- e^{\alpha_m})}{d\alpha_m} \\ &= -\mu^+(-1)e^{-\alpha_m} + \mu^- e^{\alpha_m} \\ &= \mu^+ e^{-\alpha_m} + \mu^- e^{\alpha_m} > 0 \ \forall \alpha_m \in R \end{aligned}$$

The 2nd order derivative is strictly positive, which means that there is an unique minimum and the function is completely smooth.

2.5 Further Discussions on the implications on the loss

Let's find out the value of the loss when we plug the solution of α_m in. The loss after a single iteration:

$$E = \mu^+ e^{-\alpha_m} + \mu^- e^{\alpha_m}$$

$$\begin{aligned}
&= \mu^+ e^{-\frac{1}{2}\ln\left(\frac{\mu^+}{\mu^-}\right)} + \mu^- e^{\frac{1}{2}\ln\left(\frac{\mu^+}{\mu^-}\right)} \\
&= \mu^+ e^{\ln\sqrt{\frac{\mu^-}{\mu^+}}} + \mu^- e^{\ln\sqrt{\frac{\mu^+}{\mu^-}}} \\
&= \mu^+ \sqrt{\frac{\mu^-}{\mu^+}} + \mu^- \sqrt{\frac{\mu^+}{\mu^-}} \\
&= \sqrt{\mu^+\mu^-} + \sqrt{\mu^-\mu^+} \\
&= 2\sqrt{\mu^+\mu^-}
\end{aligned}$$

Before the iteration, the objective function is simply $\mu^+ + \mu^-$ (by plugging $\alpha_m = 0$). The decrease in loss is

$$\Delta E = \mu^+ + \mu^- - 2\sqrt{\mu^+\mu^-} = (\sqrt{\mu^+} - \sqrt{\mu^-})^2 \tag{9}$$

This is actually a good measure of how a specific feature is doing. Suppose we have two features, we can calculate their corresponding ΔE. Since ΔE does not rely on the scale of μ^+, μ^-, we don't need to normalize and we can compare them directly.

2.6 Pseudocode

Given a training set T of data points x_i and their labels y_i in a two class problem, we assign initial weights $w_j^{(1)} = 1$ to all data points x_i. We perform M iterations.

For $m = 1$ to M

1. Select and extract from the pool of classifiers the classifier k_m which minimizes
$$\mu^- = \sum_{y_i \neq k_m(x_i)} w_i^{(m)}$$

2. Set the weight α_m of the classifier to
$$\alpha_m = \frac{1}{2}\ln\left(\frac{\mu^+}{\mu^-}\right)$$

3. Update the weights of the data points for the next iteration. If $k_m(x_i)$ is a miss, set
$$w_i^{(m+1)} = w_i^{(m)} e^{\alpha_m} = w_i^{(m)} \sqrt{\frac{1-e_m}{e_m}}$$
otherwise
$$w_i^{(m+1)} = w_i^{(m)} e^{-\alpha_m} = w_i^{(m)} \sqrt{\frac{e_m}{1-e_m}}$$

3 Example

Define a two-class problem in the plane:

- class 1: points in circle $x^2 + y^2 < 1$
- class 2: points not in circle and in box
  - $-2 \leq x \leq 2$ and
  - $-2 \leq y \leq 2$ and
  - $x^2 + y^2 > 1$

which includes a circle of points inside a square. We will build a strong classifier out of a pool of L (10 or 100) randomly generated linear discriminates of the type $\text{sign}(ax_1 + bx_2 + c)$.

3.1 Data Points

Below python code generates 1000 data points for each class.

```
N = 1000
def unif(n):
    vec = []
    while len(vec) < n:
        vec.append(uniform(-1,1))
    return vec

def generate_class(label=1):
    X = []
    while len(X) < N:
        x,y = unif(2)
        if label ==1 and x**2 + y**2 >= 1: continue
        if label ==-1:
            x *= 2; y*= 2
            if x**2 + y**2 < 1: continue
        X.append([x,y])
    return X
```

3.2 Weak Classfiers

We randomly generate L linear discriminants using below code:

```
L = 10
def generate_classifiers():
    C = []
    while len(C) < L:
        a,b,c = unif(3)
```

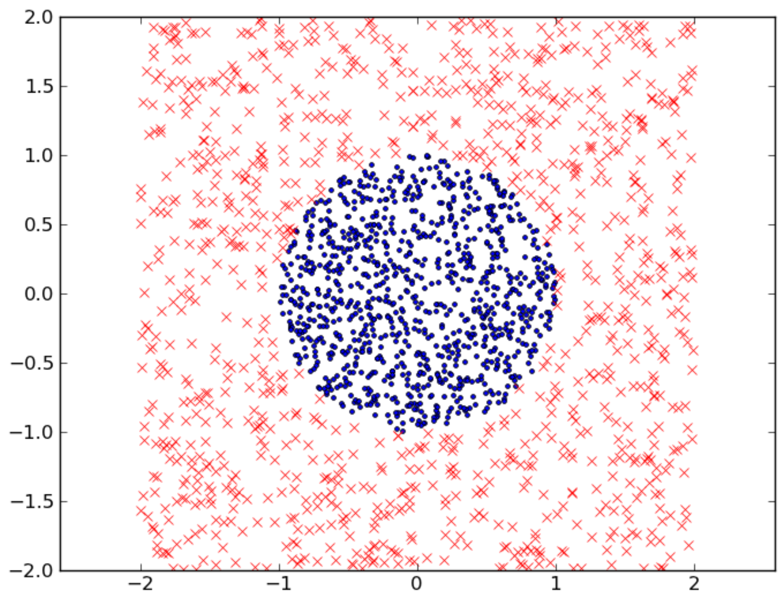

Figure 2: The data plot of the two-class problem. Blue dots are in class 1 and red crosses are in class 2.

```
        C.append([a,b,c])
    return C
```

To get predictions from the linear classifiers:

```
def is_prediction_hit(classifier, data, label):
    a,b,c = classifier
    x, y = data
    if linear(a,b,c,x,y) > 0:
        pred = 1
    else:
        pred = -1
    return pred == label
```

3.3 Scout

Assuming class 1 are labeled with 'True' and class 2 are labeled with 'False' in our data. We generate our scout table with below codes:

```
X1 = generate_class(1)
```

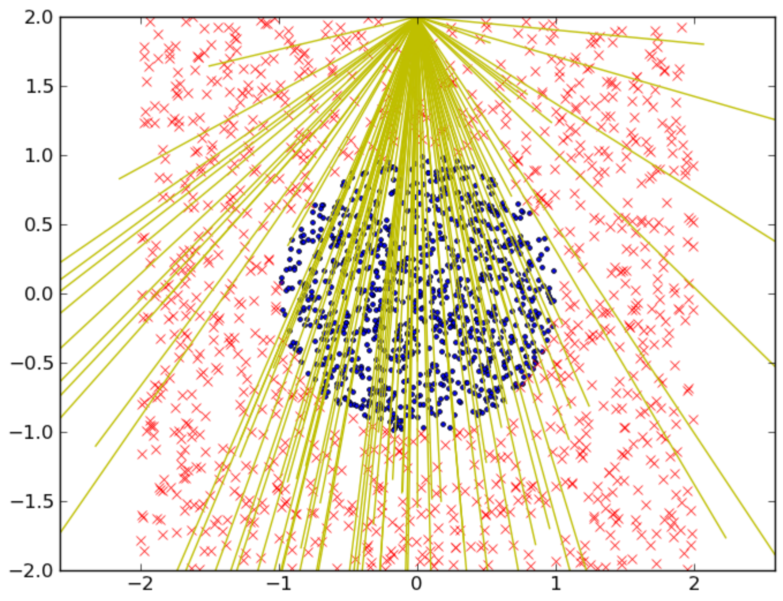

Figure 3: The data plot of the two-class problem. Blue dots are in class 1 and red crosses are in class 2. The yellow lines are linear classifiers.

```
X2 = generate_class(-1)
C = generate_classifiers()
pred = fill_scout()

def fill_scout():
    pred = {}
    for j, classifier in enumerate(C):
        pred[j] = []
        for i,data in enumerate(X1):
            pred[j].append(is_prediction_hit(classifier, data, 1))
        for i,data in enumerate(X2):
            pred[j].append(is_prediction_hit(classifier, data, -1))
    return pred
```

3.4 Initialization

```
w = [1.0]*(2*N)
alpha = [0]*L
```

```
in_committee = set()
```

3.5 Training

```
def get_draftee(pred):
    min_We = 1e100
    min_W = 1e100
    min_classifier = -1
    for j in range(L):
        if j in in_committee: continue
        We = 0
        W = 0
        for i,prediction in enumerate(pred[j]):
            W += w[i]
            if not prediction:
                We += w[i]
        if We < min_We:
            min_classifier = j
            min_We = We
            min_W = W

    em = float(min_We)/min_W
    print min_We, min_W
    alpha[min_classifier] = .5*log((1-em)/em)
    in_committee.add(min_classifier)

    for i in range(2*N):
        if not pred[min_classifier]:
            w[i] *= ((1-em)/em)**.5
        else:
            w[i] *= (em/(1-em))**.5
```

3.6 Results

Test code:

```
def test(X, y):
    cnt = 0
    for i,data in enumerate(X):
        pred = 0
        for j, classifier in enumerate(C):
            if is_prediction_hit(classifier, data, y):
                pred += alpha[j]
            else:
                pred += -alpha[j]
        if (pred > 0 and y > 0) or (pred < 0 and y < 0): cnt += 1
    return cnt

def testAll(X1, X2):
    cnt = 0
    cnt += test(X1, 1)
    cnt += test(X2, -1)
    print float(cnt)/2/N

T1 = generate_class(1)
T2 = generate_class(-1)

testAll(X1,X2)
testAll(T1,T2)
```

Training/Testing error : percentage of wrongly classified points

Table 2: Training and Testing error

number of classifiers	10	20	50	100
Train Error %	13.7	4.6	1.45	0
Test Error%	13.85	5.0	2.3	0

3.7 Regularization on boosting

Without regularization, our objective function is:

$$\min_{\alpha} \mu^+ e^{-\alpha} + \mu^- e^{\alpha}$$

Note we dropped the subscript m so α_m becomes α for simplicity. Adding L_1 regularization, we have:

$$\min_{\alpha} \mu^+ e^{-\alpha} + \mu^- e^{\alpha} + \lambda|\alpha|$$

Derivation with L_1: Assume: $\alpha_m > 0$, we can take out the absolute function,

$$\min_{\alpha} \mu^+ e^{-\alpha} + \mu^- e^{\alpha} + \lambda\alpha$$

Taking derivative with respective to α and setting it to zero,

$$\frac{d}{d\alpha}(\mu^+ e^{-\alpha} + \mu^- e^{\alpha} + \lambda\alpha) = -\mu^+ e^{-\alpha} + \mu^- e^{\alpha} + \lambda = 0$$

Define: $e^{\alpha} = z$, we have:

$$-\frac{\mu^+}{z} + \mu^- z + \lambda = 0$$

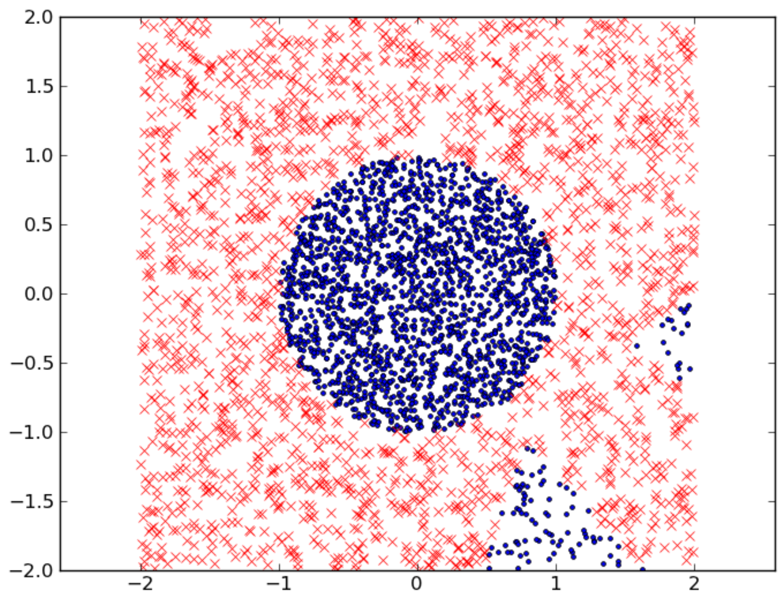

Figure 4: Predicted class labels in testing with 10 classifiers with some wrong labels. It is very clear that those error appears where we lack training examples.

i.e.,

$$\mu^- z^2 + \lambda z - \mu^+ = 0$$

Solving the quadratic equation, we get:

$$\alpha^* = \ln\left(\frac{-\lambda + \sqrt{\lambda^2 + 4\mu^+\mu^-}}{2\mu^-}\right)$$

The solution is good as long as $\mu^- > 0$, which means the feature is not perfect.

Similarly, if $\alpha < 0$, we can get:

$$\alpha^* = \ln\left(\frac{\lambda + \sqrt{\lambda^2 + 4\mu^+\mu^-}}{2\mu^-}\right)$$

First, if $\lambda = 0$, α^* is reduced to the same form as $\ln\sqrt{\frac{\mu^+}{\mu^-}} = \frac{1}{2}\ln\left(\frac{\mu^+}{\mu^-}\right)$.

Why this regularization term prevents over-fitting? Suppose now we choose a feature that is not that informative. With regularization, it should actually be pruned. If $\alpha_m \leq 0$, the term inside the ln in the above equation should be less than 1,

$$\begin{aligned}
\frac{\lambda + \sqrt{\lambda^2 + 4\mu^+\mu^-}}{2\mu^-} &< 1 \\
\sqrt{\lambda^2 + 4\mu^+\mu^-} &< 2\mu^- - \lambda \\
\lambda^2 + 4\mu^+\mu^- &< (2\mu^- - \lambda)^2 \\
\mu^+\mu^- &< {\mu^-}^2 - \lambda\mu^-
\end{aligned}$$

Since $\mu^- > 0$, we can cancel it out and do not change the inequality.

$$\mu^+ - \mu^- < -\lambda$$

If $\alpha_m \geq 0$, we have

$$\mu^+ - \mu^- > \lambda$$

So for $\alpha_m = 0$, we must have both $\alpha_m \leq 0$ and $\alpha_m \geq 0$, so that

$$|\mu^+ - \mu^-| \geq \lambda$$

If the difference between the correlation exceeds λ, the feature is redundant.

4 Statistical View and LogitBoost

References

[1] Baul Rojas, *AdaBoost and the Super Bowl of Classifiers: A tutorial introduction to Adaptive Boosting*, `http://www.inf.fu-berlin.de/inst/ag-ki/adaboost4.pdf`

[2] Yoav Freund and Robert Schapire, *A Short Introduction to Boosting*, Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, September, 1999., `http://www.yorku.ca/gisweb/eats4400/boost.pdf`

[3] Kevin Canini, et.al. *Sibyl: A system for large scale supervised machine learning.* `http://users.soe.ucsc.edu/~niejiazhong/slides/chandra.pdf`

[4] Jerome Friedman, Trevor Hastie and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. Annals of Statistics 28(2), 2000. 337-407. `http://www.stanford.edu/~hastie/Papers/AdditiveLogisticRegression/alr.pdf` or `http://people.csail.mit.edu/torralba/courses/6.869/lectures/lecture6/boosting.pdf`