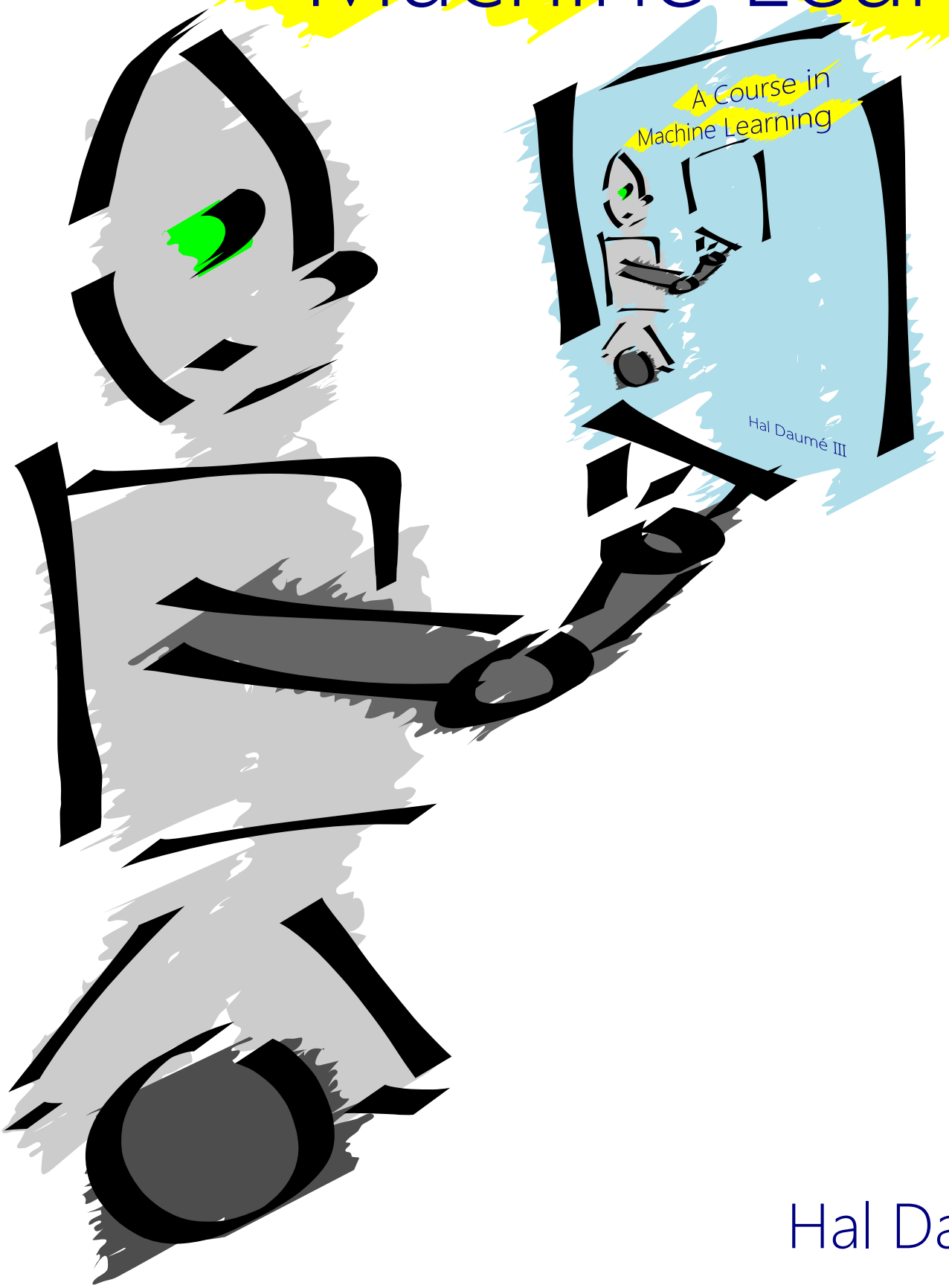
A Course in Machine Learning



Hal Daumé III

5 | Beyond Binary Classification

Learning Objectives:

- Represent complex prediction problems in a formal learning setting.
- Be able to artifically "balance" imbalanced data.
- Understand the positive and negative aspects of several reductions from multiclass classification to binary classification.
- Recognize the difference between regression and ordinal regression.
- Implement stacking as a method of collective classification.

Dependencies:

In the preceeding chapters, you have learned all about a very simple form of prediction: predicting bits. In the real world, however, we often need to predict much more complex objects. You may need to categorize a document into one of several categories: sports, entertainment, news, politics, etc. You may need to rank web pages or ads based on relevance to a query. You may need to simultaneously classify a collection of objects, such as web pages, that have important information in the links between them. These problems are all commonly encountered, yet fundamentally more complex than binary classification.

In this chapter, you will learn how to use everything you already know about binary classification to solve these more complicated problems. You will see that it's relatively easy to think of a binary classifier as a black box, which you can reuse for solving these more complex problems. This is a very useful abstraction, since it allows us to reuse knowledge, rather than having to build new learning models and algorithms from scratch.

5.1 *Learning with Imbalanced Data*

Your boss tells you to build a classifier that can identify fraudulent transactions in credit card histories. Fortunately, most transactions are legitimate, so perhaps only 0.1% of the data is a positive instance. The **imbalanced data** problem refers to the fact that for a large number of real world problems, the number of positive examples is dwarfed by the number of negative examples (or vice versa). This is actually something of a misnomer: it is not the *data* that is imbalanced, but the *distribution* from which the data is drawn. (And since the distribution is imbalanced, so must the data be.)

Imbalanced data is a problem because machine learning algorithms are too smart for your own good. For most learning algorithms, if you give them data that is 99.9% negative and 0.1% positive, they will simply learn to always predict negative. Why? Because

they are trying to minimize error, and they can achieve 0.1% error by doing nothing! If a teacher told you to study for an exam with 1000 true/false questions and only one of them is true, it is unlikely you will study very long.

Really, the problem is not with the data, but rather with the way that you have defined the learning problem. That is to say, what you care about is *not* accuracy: you care about something else. If you want a learning algorithm to do a reasonable job, you have to tell it what you want!

Most likely, what you want is *not* to optimize accuracy, but rather to optimize some other measure, like f-score or AUC. You want your algorithm to make *some* positive predictions, and simply prefer those to be "good." We will shortly discuss two heuristics for dealing with this problem: subsampling and weighting. In subsampling, you *throw out* some of you negative examples so that you are left with a balanced data set (50% positive, 50% negative). This might scare you a bit since throwing out data seems like a bad idea, but at least it makes learning much more efficient. In weighting, instead of throwing out positive examples, we just given them lower weight. If you assign an **importance weight** of 0.00101 to each of the positive examples, then there will be as much *weight* associated with positive examples as negative examples.

Before formally defining these heuristics, we need to have a mechanism for formally defining supervised learning problems. We will proceed by example, using binary classification as the canonical learning problem.

TASK: BINARY CLASSIFICATION

Given:

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \{-1, +1\}$

Compute: A function f minimizing: $\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[f(\boldsymbol{x}) \neq y\big]$

As in all the binary classification examples you've seen, you have some input space (which has always been $\mathbb{R}^D$). There is some distribution that produces labeled examples over the input space. You do not have access to that distribution, but can obtain samples from it. Your goal is to find a classifier that minimizes error on that distribution.

A small modification on this definition gives a α-weighted classification problem, where you believe that the positive class is α-times as

Algorithm 11 SUBSAMPLEMAP($\mathcal{D}^{weighted}$, α)

```
while true do
    (x, y) ~ D^weighted          // draw an example from the weighted distribution
    u ~ uniform random variable in [0, 1]
    if y = +1 or u < 1/α then
        return (x, y)
    end if
end while
```

important as the negative class.

TASK: α-WEIGHTED BINARY CLASSIFICATION

Given:

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \{-1, +1\}$

Compute: A function f minimizing: $\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\alpha^{y=1}\left[f(x) \neq y\right]\right]$

The objects given to you in weighted binary classification are identical to standard binary classification. The only difference is that the *cost* of misprediction for $y = +1$ is α, while the cost of misprediction for $y = -1$ is 1. In what follows, we assume that $\alpha > 1$. If it is not, you can simply swap the labels and use $1/\alpha$.

The question we will ask is: suppose that I have a good algorithm for solving the BINARY CLASSIFICATION problem. Can I turn that into a good algorithm for solving the α-WEIGHTED BINARY CLASSIFICATION problem?

In order to do this, you need to define a *transformation* that maps a concrete weighted problem into a concrete unweighted problem. This transformation needs to happen both at training time and at test time (though it need not be the same transformation!). Algorithm **??** sketches a training-time **sub-sampling** transformation and Algorithm **??** sketches a test-time transformation (which, in this case, is trivial). All the training algorithm is doing is retaining all positive examples and a $1/\alpha$ fraction of all negative examples. The algorithm is explicitly turning the distribution over weighted examples into a (different) distribution over binary examples. A vanilla binary classifier is trained on this **induced distribution**.

Aside from the fact that this algorithm throws out a lot of data (especially for large α), it does seem to be doing a reasonable thing. In fact, from a **reductions** perspective, it is an optimal algorithm. You can prove the following result:

Theorem 2 (Subsampling Optimality). *Suppose the binary classifier trained in Algorithm* **??** *achieves a binary error rate of ϵ. Then the error rate of the weighted predictor is equal to $\alpha\epsilon$.*

This theorem states that if your binary classifier does well (on the induced distribution), then the learned predictor will also do well (on the original distribution). Thus, we have successfully converted a weighted learning problem into a plain classification problem! The fact that the error rate of the weighted predictor is exactly α times more than that of the unweighted predictor is unavoidable: the error metric on which it is evaluated is α times bigger!

The proof of this theorem is so straightforward that we will prove it here. It simply involves some algebra on expected values.

? Why is it unreasonable to expect to be able to achieve, for instance, an error of $\sqrt{\alpha}\epsilon$, or anything that is sublinear in α?

Proof of Theorem **??**. Let $\mathcal{D}^w$ be the original distribution and let $\mathcal{D}^b$ be the induced distribution. Let f be the binary classifier trained on data from $\mathcal{D}^b$ that achieves a binary error rate of ϵ^b on that distribution. We will compute the expected error ϵ^w of f on the weighted problem:

$$\begin{aligned}
\epsilon^w &= \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}^w}\Big[\alpha^{y=1}\big[f(\boldsymbol{x}) \neq y\big]\Big] && (5.1)\\
&= \sum_{\boldsymbol{x}\in\mathcal{X}}\sum_{y\in\pm 1} \mathcal{D}^w(\boldsymbol{x},y)\alpha^{y=1}\big[f(\boldsymbol{x}) \neq y\big] && (5.2)\\
&= \alpha \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(\mathcal{D}^w(\boldsymbol{x},+1)\big[f(\boldsymbol{x}) \neq +1\big] + \mathcal{D}^w(\boldsymbol{x},-1)\frac{1}{\alpha}\big[f(\boldsymbol{x}) \neq -1\big]\Big) && (5.3)\\
&= \alpha \sum_{\boldsymbol{x}\in\mathcal{X}} \Big(\mathcal{D}^b(\boldsymbol{x},+1)\big[f(\boldsymbol{x}) \neq +1\big] + \mathcal{D}^b(\boldsymbol{x},-1)\big[f(\boldsymbol{x}) \neq -1\big]\Big) && (5.4)\\
&= \alpha\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}^b}\big[f(\boldsymbol{x}) \neq y\big] && (5.5)\\
&= \alpha\epsilon^b && (5.6)
\end{aligned}$$

And we're done! (We implicitly assumed $\mathcal{X}$ is discrete. In the case of continuous data, you need to replace all the sums over $\boldsymbol{x}$ with integrals over $\boldsymbol{x}$, but the result still holds.) □

Instead of subsampling the low-cost class, you could alternatively **oversample** the high-cost class. The easiest case is when α is an integer, say 5. Now, whenever you get a positive point, you include 5 copies of it in the induced distribution. Whenever you get a negative point, you include a single copy.

? How can you handle non-integral α, for instance 5.5?

This oversampling algorithm achieves exactly the same theoretical result as the subsampling algorithm. The main advantage to the oversampling algorithm is that it does not throw out any data. The main advantage to the subsampling algorithm is that it is more computationally efficient.

? Modify the proof of optimality for the subsampling algorithm so that it applies to the oversampling algorithm.

You might be asking yourself: intuitively, the oversampling algorithm seems like a much better idea than the subsampling algorithm, at least if you don't care about computational efficiency. But the theory tells us that they are the same! What is going on? Of course the theory isn't wrong. It's just that the assumptions are effectively different in the two cases. Both theorems state that if you can get error of ϵ on the binary problem, you automatically get error of $\alpha\epsilon$ on the weighted problem. But they do not say anything about how possible it is to get error ϵ on the binary problem. Since the oversampling algorithm produces more data points than the subsampling algorithm it is very concievable that you could get lower binary error with oversampling than subsampling.

The primary drawback to oversampling is computational inefficiency. However, for many learning algorithms, it is straightforward to include *weighted* copies of data points at no cost. The idea is to store only the unique data points and maintain a counter saying how many times they are replicated. This is not easy to do for the perceptron (it can be done, but takes work), but it *is* easy for both decision trees and KNN. For example, for decision trees (recall Algorithm 1.3), the only changes are to: (1) ensure that line 1 computes the most frequent *weighted* answer, and (2) change lines 10 and 11 to compute weighted errors.

? Why is it hard to change the perceptron? (Hint: it has to do with the fact that perceptron is online.)

5.2 Multiclass Classification

Multiclass classification is a natural extension of binary classification. The goal is still to assign a discrete label to examples (for instance, is a document about entertainment, sports, finance or world news?). The difference is that you have $K > 2$ classes to choose from.

? How would you modify KNN to take into account weights?

TASK: MULTICLASS CLASSIFICATION

Given:

1. An input space $\mathcal{X}$ and number of classes K
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times [K]$

Compute: A function f minimizing: $\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[f(\boldsymbol{x}) \neq y\big]$

Note that this is *identical* to binary classification, except for the presence of K classes. (In the above, $[K] = \{1, 2, 3, \ldots, K\}$.) In fact, if you set $K = 2$ you exactly recover binary classification.

The game we play is the same: someone gives you a binary classifier and you have to use it to solve the multiclass classification prob-

Algorithm 12 OneVersusAllTrain($\mathbf{D}^{multiclass}$, BinaryTrain)

```
for i = 1 to K do
    D^bin ← relabel D^multiclass so class i is positive and ¬i is negative
    f_i ← BinaryTrain(D^bin)
end for
return f_1, ..., f_K
```

Algorithm 13 OneVersusAllTest($f_1, \ldots, f_K, \hat{\boldsymbol{x}}$)

```
score ← ⟨0, 0, ..., 0⟩                    // initialize K-many scores to zero
for i = 1 to K do
    y ← f_i(x̂)
    score_i ← score_i + y
end for
return argmax_k score_k
```

lem. A very common approach is the **one versus all** technique (also called **OVA** or **one versus rest**). To perform OVA, you train K-many binary classifiers, $f_1, \ldots, f_K$. Each classifier sees *all* of the training data. Classifier f_i receives all examples labeled class i as positives and all other examples as negatives. At test time, whichever classifier predicts "positive" wins, with ties broken randomly.

The training and test algorithms for OVA are sketched in Algorithms 5.2 and 5.2. In the testing procedure, the prediction of the ith classifier is added to the overall score for class i. Thus, if the prediction is positive, class i gets a vote; if the prdiction is negative, everyone else (implicitly) gets a vote. (In fact, if your learning algorithm can output a confidence, as discussed in Section **??**, you can often do better by using the confidence as y, rather than a simple ± 1.)

? Suppose that you have N data points in K classes, evenly divided. How long does it take to train an OVA classifier, if the base binary classifier takes $\mathcal{O}(N)$ time to train? What if the base classifier takes $\mathcal{O}(N^2)$ time?

OVA is very natural, easy to implement, and quite natural. It also works very well in practice, so long as you do a good job choosing a good binary classification algorithm *tuning* its hyperparameters well. Its weakness is that it can be somewhat brittle. Intuitively, it is not particularly robust to errors in the underlying classifiers. If *one* classifier makes a mistake, it eis possible that the entire prediction is erroneous. In fact, it is entirely possible that *none* of the K classifiers predicts positive (which is actually the worst-case scenario from a theoretical perspective)! This is made explicit in the OVA error bound below.

? Why would using a confidence help.

Theorem 3 (OVA Error Bound)**.** *Suppose the average binary error of the K binary classifiers is ϵ. Then the error rate of the OVA multiclass predictor is* at most $(K-1)\epsilon$.

Proof of Theorem 3. The key question is erroneous predictions from the binary classifiers lead to multiclass errors. We break it down into false negatives (predicting -1 when the truth is +1) and false positives

(predicting +1 when the truth is -1).

When a false negative occurs, then the testing procedure chooses randomly between available options, which is all labels. This gives a $(K-1)/K$ probability of multiclass error. Since only *one* binary error is necessary to make this happen, the *efficiency* of this error mode is $[(K-1)/K]/1 = (K-1)/K$.

Multiple false positives can occur simultaneously. Suppose there are m false positives. If there is simultaneously a false negative, the error is 1. In order for this to happen, there have to be $m+1$ errors, so the efficiency is $1/(M+1)$. In the case that there is not a simultaneous false negative, the error probability is $m/(m+1)$. This requires m errors, leading to an efficiency of $1/(m+1)$.

The worse case, therefore, is the false negative case, which gives an efficiency of $(K-1)/K$. Since we have K-many opportunities to err, we multiply this by K and get a bound of $(K-1)\epsilon$. □

The constants in this are relatively unimportant: the aspect that matters is that this scales *linearly* in K. That is, as the number of classes grows, so does your expected error.

To develop alternative approaches, a useful way to think about turning multiclass classification problems into binary classification problems is to think of them like tournaments (football, soccer–aka football, cricket, tennis, or whatever appeals to you). You have K teams entering a tournament, but unfortunately the sport they are playing only allows two to compete at a time. You want to set up a way of pairing the teams and having them compete so that you can figure out which team is best. In learning, the teams are now the classes and you're trying to figure out which class is best.[1]

[1] The sporting analogy breaks down a bit for OVA: K games are played, wherein each team will play simultaneously against all other teams.

One natural approach is to have every team compete against every other team. The team that wins the majority of its matches is declared the winner. This is the **all versus all** (or **AVA**) approach (sometimes called **all pairs**). The most natural way to think about it is as training $\binom{K}{2}$ classifiers. Say f_{ij} for $1 \leq i < j \leq k$ is the classifier that pits class i against class j. This classifier receives all of the class i examples as "positive" and all of the class j examples as "negative." When a test point arrives, it is run through all f_{ij} classifiers. Every time f_{ij} predicts positive, class i gets a point; otherwise, class j gets a point. After running all $\binom{K}{2}$ classifiers, the class with the most votes wins.

? Suppose that you have N data points in K classes, evenly divided. How long does it take to train an AVA classifier, if the base binary classifier takes $\mathcal{O}(N)$ time to train? What if the base classifier takes $\mathcal{O}(N^2)$ time? How does this compare to OVA?

The training and test algorithms for AVA are sketched in Algorithms 5.2 and 5.2. In theory, the AVA mapping is more complicated than the weighted binary case. The result is stated below, but the proof is omitted.

Theorem 4 (AVA Error Bound). *Suppose the average binary error of*

Algorithm 14 AllVersusAllTrain($\mathbf{D}^{multiclass}$, BinaryTrain)

```
f_ij ← ∅, ∀1 ≤ i < j ≤ K
for i = 1 to K-1 do
    D^pos ← all x ∈ D^multiclass labeled i
    for j = i+1 to K do
        D^neg ← all x ∈ D^multiclass labeled j
        D^bin ← {(x, +1) : x ∈ D^pos} ∪ {(x, −1) : x ∈ D^neg}
        f_ij ← BinaryTrain(D^bin)
    end for
end for
return all f_ij s
```

Algorithm 15 AllVersusAllTest(all f_{ij}, $\hat{\boldsymbol{x}}$)

```
score ← ⟨0, 0, ..., 0⟩          // initialize K-many scores to zero
for i = 1 to K-1 do
    for j = i+1 to K do
        y ← f_ij(x̂)
        score_i ← score_i + y
        score_j ← score_j - y
    end for
end for
return argmax_k score_k
```

the $\binom{K}{2}$ *binary classifiers is* ϵ*. Then the error rate of the AVA multiclass predictor is* at most $2(K-1)\epsilon$.

? The bound for AVA is $2(K-1)\epsilon$; the bound for OVA is $(K-1)\epsilon$. Does this mean that OVA is necessarily better than AVA? Why or why not?

At this point, you might be wondering if it's possible to do better than something linear in K. Fortunately, the answer is yes! The solution, like so much in computer science, is divide and conquer. The idea is to construct a *binary tree* of classifiers. The leaves of this tree correspond to the K labels. Since there are only $\log_2 K$ decisions made to get from the root to a leaf, then there are only $\log_2 K$ chances to make an error.

An example of a classification tree for $K = 8$ classes is shown in Figure 5.2. At the root, you distinguish between classes $\{1,2,3,4\}$ and classes $\{5,6,7,8\}$. This means that you will train a binary classifier whose positive examples are all data points with multiclass label $\{1,2,3,4\}$ and whose negative examples are all data points with multiclass label $\{5,6,7,8\}$. Based on what decision is made by this classifier, you can walk down the appropriate path in the tree. When K is not a powwr of 2, the tree will not be full. This classification tree algorithm achieves the following bound.

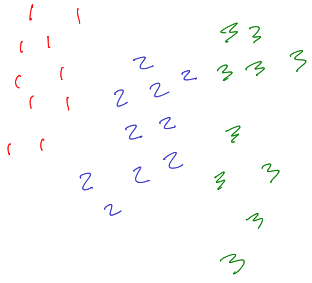


Figure 5.1: data set on which OVA will do terribly with linear classifiers

? Consider the data in Figure 5.1 and assume that you are using a perceptron as the base classifier. How well will OVA do on this data? What about AVA?

Theorem 5 (Tree Error Bound)**.** *Suppose the average binary classifiers error is* ϵ*. Then the error rate of the tree classifier is* at most $\lceil \log_2 K \rceil \epsilon$.

Proof of Theorem 5. A multiclass error is made if any classifier on

the path from the root to the correct leaf makes an error. Each has probability ϵ of making an error and the path consists of at most $\lceil \log_2 K \rceil$ binary decisions. □

One think to keep in mind with tree classifiers is that you have control over how the tree is defined. In OVA and AVA you have no say in what classification problems are created. In tree classifiers, the only thing that matters is that, at the root, half of the classes are considered positive and half are considered negative. You want to split the classes in such a way that this classification decision is as easy as possible. You can use whatever you happen to know about your classification problem to try to separate the classes out in a reasonable way.

Can you do better than $\lceil \log_2 K \rceil \epsilon$? It turns out the answer is yes, but the algorithms to do so are relatively complicated. You can actually do as well as 2ϵ using the idea of error-correcting tournaments. Moreover, you can prove a *lower bound* that states that the best you could possible do is $\epsilon/2$. This means that error-correcting tournaments are at most a factor of four worse than optimal.

5.3 Ranking

You start a new web search company called Goohooing. Like other search engines, a user inputs a query and a set of documents is retrieved. Your goal is to rank the resulting documents based on relevance to the query. The ranking problem is to take a collection of items and sort them according to some notion of preference. One of the trickiest parts of doing ranking through learning is to properly define the loss function. Toward the end of this section you will see a very general loss function, but before that let's consider a few special cases.

Continuing the web search example, you are given a collection of queries. For each query, you are also given a collection of documents, together with a desired ranking over those documents. In the following, we'll assume that you have N-many queries and for each query you have M-many documents. (In practice, M will probably vary by query, but for ease we'll consider the simplified case.) The goal is to train a binary classifier to predict a **preference function**. Given a query q and two documents d_i and d_j, the classifier should predict whether d_i should be preferred to d_j with respect to the query q.

As in all the previous examples, there are two things we have to take care of: (1) how to train the classifier that predicts preferences; (2) how to turn the predicted preferences into a ranking. Unlike the previous examples, the second step is somewhat complicated in the

Algorithm 16 NAIVERANKTRAIN(*RankingData*, BINARYTRAIN)

```
D ← [ ]
for n = 1 to N do
  for all i, j = 1 to M and i ≠ j do
    if i is prefered to j on query n then
      D ← D ⊕ (x_nij, +1)
    else if j is prefered to i on query n then
      D ← D ⊕ (x_nij, −1)
    end if
  end for
end for
return BINARYTRAIN(D)
```

Algorithm 17 NAIVERANKTEST(f, $\hat{\boldsymbol{x}}$)

```
score ← ⟨0, 0, ..., 0⟩          // initialize M-many scores to zero
for all i, j = 1 to M and i ≠ j do
  y ← f(x̂_ij)                    // get predicted ranking of i and j
  score_i ← score_i + y
  score_j ← score_j - y
end for
return ARGSORT(score)            // return queries sorted by score
```

ranking case. This is because we need to predict an entire ranking of a large number of documents, somehow assimilating the preference function into an overall permutation.

For notationally simplicity, let $\boldsymbol{x}_{nij}$ denote the features associated with comparing document i to document j on query n. Training is fairly straightforward. For every n and every pair $i \neq j$, we will create a binary classification example based on features $\boldsymbol{x}_{nij}$. This example is positive if i is preferred to j in the true ranking. It is negative if j is preferred to i. (In some cases the true ranking will not express a preference between two objects, in which case we exclude the i, j and j, i pair from training.)

Now, you might be tempted to evaluate the classification performance of this binary classifier on its own. The problem with this approach is that it's impossible to tell—just by looking at its output on one i, j pair—how good the overall ranking is. This is because there is the intermediate step of turning these pairwise predictions into a coherent ranking. What you need to do is measure how well the ranking based on your predicted preferences compares to the true ordering. Algorithms 5.3 and 5.3 show naive algorithms for training and testing a ranking function.

These algorithms actually work quite well in the case of **bipartite ranking problems**. A bipartite ranking problem is one in which you are only ever trying to predict a binary response, for instance "is this

document relevant or not?" but are being evaluated according to a metric like **AUC**. This is essentially because the only goal in bipartite problems to to ensure that all the relevant documents are ahead of all the irrelevant documents. There is no notion that one relevant document is *more relevant* than another.

For non-bipartite ranking problems, you can do better. First, when the preferences that you get at training time are more nuanced than "relevant or not," you can incorporate these preferences at training time. Effectively, you want to give a higher weight to binary problems that are very different in terms of perference than others. Second, rather than producing a list of scores and then calling an arbitrary sorting algorithm, you can actually use the preference function as the sorting function inside your own implementation of quicksort.

We can now formalize the problem. Define a ranking as a function σ that maps the objects we are ranking (documents) to the desired position in the list, $1, 2, \dots M$. If $\sigma_u < \sigma_v$ then u is preferred to v (i.e., appears earlier on the ranked document list). Given data with observed rankings σ, our goal is to learn to predict rankings for new objects, $\hat{\sigma}$. We define Σ_M as the set of all ranking functions over M objects. We also wish to express the fact that making a mistake on some pairs is worse than making a mistake on others. This will be encoded in a cost function ω (omega), where $\omega(i, j)$ is the cost for accidentally putting something in position j when it should have gone in position i. To be a valid cost function valid, ω must be (1) symmetric, (2) monotonic and (3) satisfy the triangle inequality. Namely: (1) $\omega(i, j) = \omega(j, i)$; (2) if $i < j < k$ or $i > j > k$ then $\omega(i, j) \leq \omega(i, k)$; (3) $\omega(i, j) + \omega(j, k) \geq \omega(i, k)$. With these definitions, we can properly define the ranking problem.

TASK: ω-RANKING

Given:

1. An input space $\mathcal{X}$
2. An unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \Sigma_M$

Compute: A function $f : \mathcal{X} \to \Sigma_M$ minimizing:

$$\mathbb{E}_{(\boldsymbol{x},\sigma)\sim\mathcal{D}} \left[\sum_{u \neq v} [\sigma_u < \sigma_v] \, [\hat{\sigma}_v < \hat{\sigma}_u] \, \omega(\sigma_u, \sigma_v) \right] \tag{5.7}$$

where $\hat{\sigma} = f(\boldsymbol{x})$

In this definition, the only complex aspect is the loss function 5.7. This loss sums over all pairs of objects u and v. If the true ranking (σ)

Algorithm 18 RANKTRAIN($\mathbf{D}^{rank}$, ω, BINARYTRAIN)

```
1: D^bin ← [ ]
2: for all (x, σ) ∈ D^rank do
3:     for all u ≠ v do
4:         y ← sign(σ_v - σ_u)                    // y is +1 if u is prefered to v
5:         w ← ω(σ_u, σ_v)                        // w is the cost of misclassification
6:         D^bin ← D^bin ⊕ (y, w, x_uv)
7:     end for
8: end for
9: return BinaryTrain(D^bin)
```

prefers u to v, but the predicted ranking ($\hat{\sigma}$) prefers v to u, then you incur a cost of $\omega(\sigma_u, \sigma_v)$.

Depending on the problem you care about, you can set ω to many "standard" options. If $\omega(i,j) = 1$ whenever $i \neq j$, then you achieve the Kemeny distance measure, which simply counts the number of pairwise misordered items. In many applications, you may only care about getting the top K predictions correct. For instance, your web search algorithm may only display $K = 10$ results to a user. In this case, you can define:

$$\omega(i,j) = \begin{cases} 1 & \text{if } \min\{i,j\} \leq K \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

In this case, only errors in the top K elements are penalized. Swapping items 55 and 56 is irrelevant (for $K < 55$).

Finally, in the bipartite ranking case, you can express the **area under the curve** (**AUC**) metric as:

$$\omega(i,j) = \frac{\binom{M}{2}}{M^+(M-M^+)} \times \begin{cases} 1 & \text{if } i \leq M^+ \text{ and } j > M^+ \\ 1 & \text{if } j \leq M^+ \text{ and } i > M^+ \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

Here, M is the total number of objects to be ranked and M^+ is the number that are actually "good." (Hence, $M - M^+$ is the number that are actually "bad," since this is a bipartite problem.) You are only penalized if you rank a good item in position greater than M^+ or if you rank a bad item in a position less than or equal to M^+.

In order to *solve* this problem, you can follow a recipe similar to the naive approach sketched earlier. At training time, the biggest change is that you can *weight* each training example by how bad it would be to mess it up. This change is depicted in Algorithm 5.3, where the binary classiciation data has *weights* w provided for saying how important a given example is. These weights are derived from the cost function ω.

At test time, instead of predicting scores and then sorting the list, you essentially run the quicksort algorith, using f as a comparison

Algorithm 19 RANKTEST(f, $\hat{\boldsymbol{x}}$, obj)

```
if obj contains 0 or 1 elements then
    return obj
else
    p ← randomly chosen object in obj                          // pick pivot
    left ← [ ]                                  // elements that seem smaller than p
    right ← [ ]                                 // elements that seem larger than p
    for all u ∈ obj \{p} do
        ŷ ← f(x_up)                             // what is the probability that u precedes p
        if uniform random variable < ŷ then
            left ← left ⊕ u
        else
            right ← right ⊕ u
        end if
    end for
    left ← RANKTEST(f, x̂, left)                                // sort earlier elements
    right ← RANKTEST(f, x̂, right)                              // sort later elements
    return left ⊕ ⟨p⟩ ⊕ right
end if
```

function. At each step in Algorithm 5.3, a pivot p is chosen. Every other object u is compared to p using f. If f thinks u is better, then it is sorted on the left; otherwise it is sorted on the right. There is one major difference between this algorithmand quicksort: the comparison function is allowed to be *probabilistic*. If f outputs probabilities, for instance it predicts that u has an 80% probability of being better than p, then it puts it on the left with 80% probability and on the right with 20% probability. (The pseudocode is written in such a way that even if f just predicts $-1, +1$, the algorithm still works.)

This algorithm is better than the naive algorithm in at least two ways. First, it only makes $\mathcal{O}(M \log_2 M)$ calls to f (in expectation), rather than $\mathcal{O}(M^2)$ calls in the naive case. Second, it achieves a better error bound, shown below:

Theorem 6 (Rank Error Bound)**.** *Suppose the average binary error of f is ϵ. Then the ranking algorithm achieves a test error of at most 2ϵ in the general case, and ϵ in the bipartite case.*

Figure 5.3: example face finding image and pixel mask

5.4 Collective Classification

You are writing new software for a digital camera that does face identification. However, instead of simply finding a bounding box around faces in an image, you must predict where a face is *at the pixel level*. So your input is an image (say, 100×100 pixels: this is a really low resolution camera!) and your output is a set of 100×100 binary predictions about each pixel. You are given a large collection

of training examples. An example input/output pair is shown in Figure 5.3.

Your first attempt might be to train a binary classifier to predict whether pixel (i, j) is part of a face or not. You might feed in features to this classifier about the RGB values of pixel (i, j) as well as pixels in a window arround that. For instance, pixels in the region $\{(i + k, j + l) : k \in [-5, 5], l \in [-5, 5]\}$.

You run your classifier and notice that it predicts weird things, like what you see in Figure 5.4. You then realize that predicting each pixel independently is a bad idea! If pixel (i, j) is part of a face, then this significantly increases the chances that pixel $(i + 1, j)$ is also part of a face. (And similarly for other pixels.) This is a **collective classification** problem because you are trying to predict multiple, correlated objects at the same time.

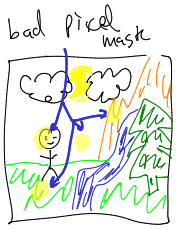


Figure 5.4: bad pixel mask for previous image

The most general way to formulate these problems is as (undirected) **graph** prediction problems. Our input now takes the form of a graph, where the vertices are input/output pairs and the edges represent the correlations among the putputs. (Note that edges do not need to express correlations among the inputs: these can simply be encoded on the nodes themselves.) For example, in the face identification case, each pixel would correspond to an vertex in the graph. For the vertex that corresponds to pixel $(5, 10)$, the input would be whatever set of features we want about that pixel (including features about neighboring pixels). There would be edges between that vertex and (for instance) vertices $(4, 10)$, $(6, 10)$, $(5, 9)$ and $(5, 11)$. If we are predicting one of K classes at each vertex, then we are given a graph whose vertices are labeled by pairs $(x, k) \in \mathcal{X} \times [K]$. We will write $\mathcal{G}(\mathcal{X} \times [K])$ to denote the set of all such graphs. A graph in this set is denoted as $G = (V, E)$ with vertices V and edges E. Our goal is a function f that takes as input a graph from $\mathcal{G}(\mathcal{X})$ and predicts a label from $[K]$ for each of its vertices.

? Similar problems come up all the time. Cast the following as collective classification problems: web page categorization; labeling words in a sentence as noun, verb, adjective, etc.; finding genes in DNA sequences; predicting the stock market.

? Formulate the example problems above as graph prediction problems.

TASK: COLLECTIVE CLASSIFICATION

Given:

1. An input space $\mathcal{X}$ and number of classes K
2. An unknown distribution $\mathcal{D}$ over $\mathcal{G}(\mathcal{X} \times [K])$

Compute: A function $f : \mathcal{G}(\mathcal{X}) \to \mathcal{G}([K])$ minimizing: $\mathbb{E}_{(V,E)\sim\mathcal{D}} \left[\sum_{v \in V} \left[\hat{y}_v \neq y_v \right] \right]$, where y_v is the label associated with vertex v in G and $\hat{y}_v$ is the label predicted by $f(G)$.

In collective classification, you would like to be able to use the

labels of neighboring vertices to help predict the label of a given vertex. For instance, you might want to add features to the predict of a given vertex based on the labels of each neighbor. At training time, this is easy: you get to see the true labels of each neighbor. However, at test time, it is much more difficult: you are, yourself, predicting the labels of each neighbor.

This presents a chicken and egg problem. You are trying to predict a collection of labels. But the prediction of each label depends on the prediction of other labels. If you remember from before, a general solution to this problem is iteration: you can begin with some guesses, and then try to improve these guesses over time. [2]

This is the idea of **stacking** for solving collective classification (see Figure 5.5. You can train 5 classifiers. The first classifier *just* predicts the value of each pixel independently, like in Figure 5.4. This doesn't use any of the graph structure at all. In the second level, you can repeat the classification. However, you can use the outputs from the first level as initial guesses of labels. In general, for the Kth level in the stack, you can use the inputs (pixel values) as well as the predictions for all of the $K-1$ previous levels of the stack. This means training K-many binary classifiers based on different feature sets.

The prediction technique for stacking is sketched in Algorithm 5.4. This takes a list of K classifiers, corresponding to each level in the stack, and an input graph G. The variable $\hat{Y}_{k,v}$ stores the prediction of classifier k on vertex v in the graph. You first predict every node in the vertex using the first layer in the stack, and no neighboring information. For the rest of the layers, you add on features to each node based on the predictions made by lower levels in the stack for neighboring nodes ($\mathcal{N}(u)$ denotes the neighbors of u).

The training procedure follows a similar scheme, sketched in Algorithm 5.4. It largely follows the same schematic as the prediction algorithm, but with training fed in. After the classifier for the k level has been trained, it is used to predict labels on every node in the graph. These labels are used by later levels in the stack, as features.

One thing to be aware of is that MULTICLASSTRAIN could conceivably overfit its training data. For example, it is possible that the first layer might actually achieve 0% error, in which case there is no reason to iterate. But at test time, it will probably *not* get 0% error, so this is misleading. There are (at least) two ways to address this issue. The first is to use cross-validation during training, and to use the predictions obtained during cross-validation as the predictions from **StackTest**. This is typically very safe, but somewhat expensive. The alternative is to simply *over-regularize* your training algorithm. In particular, instead of trying to find hyperparameters that get the

[2] Alternatively, the fact that we're using a graph might scream to you "dynamic programming." Rest assured that you can do this too: skip forward to Chapter 18 for lots more detail here!

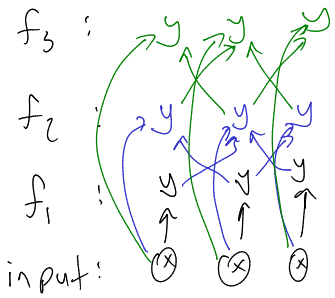


Figure 5.5: a charicature of how stacking works

Algorithm 20 StackTrain($\mathcal{D}^{cc}$, K, MulticlassTrain)

$\mathbf{D}^{mc} \leftarrow [\,]$ // our generated multiclass data
$\hat{Y}_{k,n,v} \leftarrow o, \forall k \in [K], n \in [N], v \in G_n$ // initialize predictions for all levels
for $k = 1$ **to** K **do**
  for $n = 1$ **to** N **do**
    for all $v \in G_n$ **do**
      $(\boldsymbol{x}, y) \leftarrow$ features and label for node v
      $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \hat{Y}_{l,n,u}, \forall u \in \mathcal{N}(u), \forall l \in [k-1]$ // add on features for
        // neighboring nodes from lower levels in the stack
      $\mathbf{D}^{mc} \leftarrow \mathbf{D}^{mc} \oplus (y, \boldsymbol{x})$ // add to multiclass data
    end for
  end for
  $f_k \leftarrow$ MulticlassTrain($\mathbf{D}^{bin}$) // train kth level classifier
  for $n = 1$ **to** N **do**
    $\hat{Y}_{k,n,v} \leftarrow$ StackTest($f_1, \ldots, f_k, G_n$) // predict using kth level classifier
  end for
end for
return $f_1, \ldots, f_K$ // return all classifiers

Algorithm 21 StackTest($f_1, \ldots, f_K, G$)

$\hat{Y}_{k,v} \leftarrow o, \forall k \in [K], v \in G$ // initialize predictions for all levels
for $k = 1$ **to** K **do**
  for all $v \in G$ **do**
    $\boldsymbol{x} \leftarrow$ features for node v
    $\boldsymbol{x} \leftarrow \boldsymbol{x} \oplus \hat{Y}_{l,u}, \forall u \in \mathcal{N}(u), \forall l \in [k-1]$ // add on features for
      // neighboring nodes from lower levels in the stack
    $\hat{Y}_{k,v} \leftarrow f_k(\boldsymbol{x})$ // predict according to kth level
  end for
end for
return $\{\hat{Y}_{K,v} : v \in G\}$ // return predictions for every node from the last layer

best development data performance, try to find hyperparameters that make your *training* performance approximately equal to your *development* performance. This will ensure that your predictions at the kth layer are indicative of how well the algorithm will actually do at test time.

TODO: finish this discussion

5.5 Exercises

Exercise 5.1. **TODO...**