

## 6 | BEYOND BINARY CLASSIFICATION

Different general classification methods can give different, but equally plausible, classifications, so you need an application context to choose among them.

– Karen Spärck-Jones

### Learning Objectives:

- Represent complex prediction problems in a formal learning setting.
- Be able to artificially “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.

IN THE PRECEDING 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. 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.

Dependencies:

### 6.1 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 your 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 give 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  $\mathcal{X} \times \{-1, +1\}$
3. A training set  $D$  sampled from  $\mathcal{D}$

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

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  $\alpha$ -weighted classification problem, where you believe that the positive class is  $\alpha$ -times as

**Algorithm 11** SUBSAMPLEMAP( $\mathcal{D}^{\text{weighted}}, \alpha$ )

---

```

1: while true do
2:    $(x, y) \sim \mathcal{D}^{\text{weighted}}$  // draw an example from the weighted distribution
3:    $u \sim$  uniform random variable in  $[0, 1]$ 
4:   if  $y = +1$  or  $u < \frac{1}{\alpha}$  then
5:     return  $(x, y)$ 
6:   end if
7: end while

```

---

**Algorithm 12** SUBSAMPLETEST( $f^{\text{BINARY}}, \hat{x}$ )

---

```

1: return  $f^{\text{BINARY}}(\hat{x})$ 

```

---

important as the negative class.

**TASK:  $\alpha$ -WEIGHTED BINARY CLASSIFICATION**

Given:

1. An input space  $\mathcal{X}$
2. An unknown distribution  $\mathcal{D}$  over  $\mathcal{X} \times \{-1, +1\}$
3. A training set  $D$  sampled from  $\mathcal{D}$

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

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  $\alpha$ , 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  $\alpha$ -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 6.1 sketches a training-time **sub-sampling** transformation and Algorithm 6.1 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  $\alpha$ ), 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 3** (Subsampling Optimality). *Suppose the binary classifier trained in Algorithm 6.1 achieves a binary error rate of  $\epsilon$ . 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  $\alpha$  times more than that of the unweighted predictor is unavoidable: the error metric on which it is evaluated is  $\alpha$  times bigger!

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

*Proof of Theorem 3.* 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  $\epsilon^b$  on that distribution. We will compute the expected error  $\epsilon^w$  of  $f$  on the weighted problem:

$$\epsilon^w = \mathbb{E}_{(x,y) \sim \mathcal{D}^w} [\alpha^{y=1} [f(x) \neq y]] \quad (6.1)$$

$$= \sum_{x \in \mathcal{X}} \sum_{y \in \pm 1} \mathcal{D}^w(x, y) \alpha^{y=1} [f(x) \neq y] \quad (6.2)$$

$$= \alpha \sum_{x \in \mathcal{X}} \left( \mathcal{D}^w(x, +1) [f(x) \neq +1] + \mathcal{D}^w(x, -1) \frac{1}{\alpha} [f(x) \neq -1] \right) \quad (6.3)$$

$$= \alpha \sum_{x \in \mathcal{X}} \left( \mathcal{D}^b(x, +1) [f(x) \neq +1] + \mathcal{D}^b(x, -1) [f(x) \neq -1] \right) \quad (6.4)$$

$$= \alpha \mathbb{E}_{(x,y) \sim \mathcal{D}^b} [f(x) \neq y] \quad (6.5)$$

$$= \alpha \epsilon^b \quad (6.6)$$

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  $x$  with integrals over  $x$ , but the result still holds.)  $\square$

Instead of subsampling the low-cost class, you could alternatively **oversample** the high-cost class. The easiest case is when  $\alpha$  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.

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  $\alpha$ ?

How can you handle non-integral  $\alpha$ , 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.

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  $\epsilon$  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  $\epsilon$  on the binary problem. Since the oversampling algorithm produces more data points than the subsampling algorithm it is very conceivable 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.



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



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



How would you modify KNN to take into account weights?

## 6.2 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.

**Algorithm 13** ONEVERSUSALLTRAIN( $\mathbf{D}^{\text{multiclass}}$ , BINARYTRAIN)

---

```

1: for  $i = 1$  to  $K$  do
2:    $\mathbf{D}^{\text{bin}} \leftarrow$  relabel  $\mathbf{D}^{\text{multiclass}}$  so class  $i$  is positive and  $\neg i$  is negative
3:    $f_i \leftarrow$  BINARYTRAIN( $\mathbf{D}^{\text{bin}}$ )
4: end for
5: return  $f_1, \dots, f_K$ 

```

---

**Algorithm 14** ONEVERSUSALLTEST( $f_1, \dots, f_K, \hat{x}$ )

---

```

1:  $\text{score} \leftarrow \langle 0, 0, \dots, 0 \rangle$  // initialize  $K$ -many scores to zero
2: for  $i = 1$  to  $K$  do
3:    $y \leftarrow f_i(\hat{x})$ 
4:    $\text{score}_i \leftarrow \text{score}_i + y$ 
5: end for
6: return  $\text{argmax}_k \text{score}_k$ 

```

---

**TASK: MULTICLASS CLASSIFICATION**

Given:

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

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

Note that this is *identical* to binary classification, except for the presence of  $K$  classes. (In the above,  $[K] = \{1, 2, 3, \dots, K\}$ .) 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 problem. 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,  $f_1, \dots, f_K$ . 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 6.2 and 6.2. In the testing procedure, the prediction of the  $i$ th classifier is added to the overall score for class  $i$ . Thus, if the prediction is positive, class  $i$  gets a vote; if the prediction 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  $\pm 1$ .)

OVA is quite natural and easy to implement. It also works very

?

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?

?

Why would using a confidence help?

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 is 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.

**Theorem 4** (OVA Error Bound). *Suppose the average binary error of the  $K$  binary classifiers is  $\epsilon$ . Then the error rate of the OVA multiclass predictor is at most  $(K - 1)\epsilon$ .*

*Proof of Theorem 4.* The key question is how 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  $(K - 1)\epsilon$ .  $\square$

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.<sup>1</sup>

One natural approach is to have every team compete against every other team. The team that wins the majority of its matches is

<sup>1</sup> The sporting analogy breaks down a bit for OVA:  $K$  games are played, wherein each team will play simultaneously against all other teams.

**Algorithm 15** ALLVERSUSALLTRAIN( $\mathbf{D}^{\text{multiclass}}$ , BINARYTRAIN)

---

```

1:  $f_{ij} \leftarrow \emptyset, \forall 1 \leq i < j \leq K$ 
2: for  $i = 1$  to  $K-1$  do
3:    $\mathbf{D}^{\text{pos}} \leftarrow$  all  $x \in \mathbf{D}^{\text{multiclass}}$  labeled  $i$ 
4:   for  $j = i+1$  to  $K$  do
5:      $\mathbf{D}^{\text{neg}} \leftarrow$  all  $x \in \mathbf{D}^{\text{multiclass}}$  labeled  $j$ 
6:      $\mathbf{D}^{\text{bin}} \leftarrow \{(x, +1) : x \in \mathbf{D}^{\text{pos}}\} \cup \{(x, -1) : x \in \mathbf{D}^{\text{neg}}\}$ 
7:      $f_{ij} \leftarrow$  BINARYTRAIN( $\mathbf{D}^{\text{bin}}$ )
8:   end for
9: end for
10: return all  $f_{ij}$ s

```

---

**Algorithm 16** ALLVERSUSALLTEST(all  $f_{ij}$ ,  $\hat{x}$ )

---

```

1:  $\text{score} \leftarrow \langle 0, 0, \dots, 0 \rangle$  // initialize  $K$ -many scores to zero
2: for  $i = 1$  to  $K-1$  do
3:   for  $j = i+1$  to  $K$  do
4:      $y \leftarrow f_{ij}(\hat{x})$ 
5:      $\text{score}_i \leftarrow \text{score}_i + y$ 
6:      $\text{score}_j \leftarrow \text{score}_j - y$ 
7:   end for
8: end for
9: return  $\text{argmax}_k \text{score}_k$ 

```

---

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.

The training and test algorithms for AVA are sketched in Algorithms 6.2 and 6.2. In theory, the AVA mapping is more complicated than the weighted binary case. The result is stated below, but the proof is omitted.

**Theorem 5** (AVA Error Bound). *Suppose the average binary error of the  $\binom{K}{2}$  binary classifiers is  $\epsilon$ . Then the error rate of the AVA multiclass predictor is at most  $2(K-1)\epsilon$ .*

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

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 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?



to make an error.

An example of a classification tree for  $K = 8$  classes is shown in Figure 6.2. 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 power of 2, the tree will not be full. This classification tree algorithm achieves the following bound.

**Theorem 6** (Tree Error Bound). *Suppose the average binary classifiers error is  $\epsilon$ . Then the error rate of the tree classifier is at most  $\lceil \log_2 K \rceil \epsilon$ .*

*Proof of Theorem 6.* A multiclass error is made if any classifier on the path from the root to the correct leaf makes an error. Each has probability  $\epsilon$  of making an error and the path consists of at most  $\lceil \log_2 K \rceil$  binary decisions.  $\square$

One thing 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  $2\epsilon$  using the idea of error-correcting tournaments. Moreover, you can prove a *lower bound* that states that the best you could possibly do is  $\epsilon/2$ . This means that error-correcting tournaments are at most a factor of four worse than optimal.

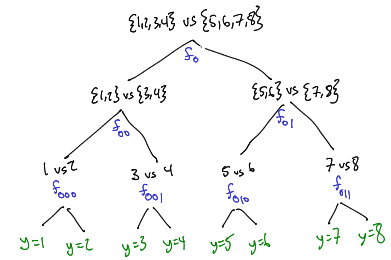


Figure 6.2: example classification tree for  $K = 8$

### 6.3 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.

**Algorithm 17** NAIVERANKTRAIN(*RankingData*, BINARYTRAIN)

---

```

1:  $\mathbf{D} \leftarrow []$ 
2: for  $n = 1$  to  $N$  do
3:   for all  $i, j = 1$  to  $M$  and  $i \neq j$  do
4:     if  $i$  is preferred to  $j$  on query  $n$  then
5:        $\mathbf{D} \leftarrow \mathbf{D} \oplus (x_{nij}, +1)$ 
6:     else if  $j$  is preferred to  $i$  on query  $n$  then
7:        $\mathbf{D} \leftarrow \mathbf{D} \oplus (x_{nij}, -1)$ 
8:     end if
9:   end for
10: end for
11: return BINARYTRAIN( $\mathbf{D}$ )

```

---

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 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 notational simplicity, let  $x_{nij}$  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  $x_{nij}$ . 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 6.3 and 6.3 show naive algorithms for training

**Algorithm 18** NAIVERANKTEST( $f, \hat{x}$ )

---

```

1:  $score \leftarrow \langle 0, 0, \dots, 0 \rangle$  // initialize  $M$ -many scores to zero
2: for all  $i, j = 1$  to  $M$  and  $i \neq j$  do
3:    $y \leftarrow f(\hat{x}_{ij})$  // get predicted ranking of  $i$  and  $j$ 
4:    $score_i \leftarrow score_i + y$ 
5:    $score_j \leftarrow score_j - y$ 
6: end for
7: return ARGSORT( $score$ ) // return queries sorted by score

```

---

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 is 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 preference 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  $\sigma$  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  $\sigma$ , our goal is to learn to predict rankings for new objects,  $\hat{\sigma}$ . We define  $\Sigma_M$  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  $\omega$  (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,  $\omega$  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:  $\omega$ -RANKING**

Given:

1. An input space  $\mathcal{X}$
2. An unknown distribution  $\mathcal{D}$  over  $\mathcal{X} \times \Sigma_M$
3. A training set  $D$  sampled from  $\mathcal{D}$

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

$$\mathbb{E}_{(x, \sigma) \sim \mathcal{D}} \left[ \sum_{u \neq v} [\sigma_u < \sigma_v] [\hat{\sigma}_v < \hat{\sigma}_u] \omega(\sigma_u, \sigma_v) \right] \quad (6.7)$$

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

In this definition, the only complex aspect is the loss function 6.7. This loss sums over all pairs of objects  $u$  and  $v$ . If the true ranking ( $\sigma$ ) 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  $\omega$  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:

$$\omega(i, j) = \begin{cases} 1 & \text{if } \min\{i, j\} \leq K \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases} \quad (6.8)$$

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:

$$\omega(i, j) = \frac{\binom{M}{2}}{M^+(M - M^+)} \times \begin{cases} 1 & \text{if } i \leq M^+ \text{ and } j > M^+ \\ 1 & \text{if } j \leq M^+ \text{ and } i > M^+ \\ 0 & \text{otherwise} \end{cases} \quad (6.9)$$

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

**Algorithm 19** RANKTRAIN( $\mathbf{D}^{rank}$ ,  $\omega$ , BINARYTRAIN)

---

```

1:  $\mathbf{D}^{bin} \leftarrow []$ 
2: for all  $(x, \sigma) \in \mathbf{D}^{rank}$  do
3:   for all  $u \neq v$  do
4:      $y \leftarrow \text{SIGN}(\sigma_v - \sigma_u)$            //  $y$  is +1 if  $u$  is preferred to  $v$ 
5:      $w \leftarrow \omega(\sigma_u, \sigma_v)$        //  $w$  is the cost of misclassification
6:      $\mathbf{D}^{bin} \leftarrow \mathbf{D}^{bin} \oplus (y, w, x_{uv})$ 
7:   end for
8: end for
9: return BINARYTRAIN( $\mathbf{D}^{bin}$ )

```

---

**Algorithm 20** RANKTEST( $f$ ,  $\hat{x}$ ,  $obj$ )

---

```

1: if  $obj$  contains 0 or 1 elements then
2:   return  $obj$ 
3: else
4:    $p \leftarrow$  randomly chosen object in  $obj$            // pick pivot
5:    $left \leftarrow []$                                    // elements that seem smaller than  $p$ 
6:    $right \leftarrow []$                                 // elements that seem larger than  $p$ 
7:   for all  $u \in obj \setminus \{p\}$  do
8:      $\hat{y} \leftarrow f(x_{up})$                            // what is the probability that  $u$  precedes  $p$ 
9:     if uniform random variable  $< \hat{y}$  then
10:       $left \leftarrow left \oplus u$ 
11:     else
12:       $right \leftarrow right \oplus u$ 
13:     end if
14:   end for
15:    $left \leftarrow$  RANKTEST( $f$ ,  $\hat{x}$ ,  $left$ )               // sort earlier elements
16:    $right \leftarrow$  RANKTEST( $f$ ,  $\hat{x}$ ,  $right$ )           // sort later elements
17:   return  $left \oplus \langle p \rangle \oplus right$ 
18: end if

```

---

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 6.3, where the binary classification data has *weights*  $w$  provided for saying how important a given example is. These weights are derived from the cost function  $\omega$ .

At test time, instead of predicting scores and then sorting the list, you essentially run the quicksort algorithm, using  $f$  as a comparison function. At each step in Algorithm 6.3, 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 algorithm and 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 7** (Rank Error Bound). *Suppose the average binary error of  $f$  is  $\epsilon$ . Then the ranking algorithm achieves a test error of at most  $2\epsilon$  in the general case, and  $\epsilon$  in the bipartite case.*

## 6.4 Further Reading

TODO further reading