LEARNING TO RANK

IMPROVING THE PERFORMANCE OF AN ENTROPY DRIVEN ADVISORY SYSTEM USING IMPLICIT USER FEEDBACK

By:
Seth Kingma
Studentnumber 0735272
s.kingma@aspin.nl

Supervisors:
Dr. M.A. Wiering (Artificial Intelligence, University of Groningen)
Prof. Dr. L.R.B. Schomaker (Artificial Intelligence, University of Groningen)

ARTIFICIAL INTELLIGENCE
UNIVERSITY OF GRONINGEN

CONTENTS

LIST OF TABLES

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

LIST OF FIGURES

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

LIST OF EQUATIONS

CHAPTER 2

CHAPTER 4

CHAPTER 5

CHAPTER 7

ABSTRACT

Koga-Miyata is a Dutch bicycle manufacturer specializing in the design and production of bicycles intended for the high-end segment. Yearly Koga markets about 60 to 70 different bicycles segmented into 6 to 8 partly overlapping segments. At times this overlap makes it relatively difficult for consumers to determine which Koga bicycles fit their needs best. To support their consumers Koga launched an online bicycle advisory system on their website on http://www.koga.com. By answering a number of simple use targeted questions, visitors of the Koga website can use this system to retrieve a list of possibly appropriate bicycles.

The advisory system was built using a number of techniques from the field of decision tree learning and information theory specifically. Borrowing from these concepts it proved to be possible to design a system in which the order of the questions is not provided beforehand. Given the answers provided by a user of the system to a series of questions, the system automatically selects the next best question to put to the user. In contrast to most other advisory systems where the order of the questions is programmed into the system, maintenance of this system is much easier. Changes do not need to be programmed, but can be configured using a simple maintenance tool. Given the yearly changing collection, this setup greatly reduced the total cost of ownership (TCO) for Koga.

The only information available to the advisory system is a set of questions, the possible answers to these questions, and a set of so-called activation values linking each answer to each bicycle. High activation values mean high relevancy, low values low relevancy. The questions, answers and activation values are provided by an expert using the maintenance tool mentioned above. Configuring the activation values can be somewhat tricky. Setting them incorrectly obviously leads the system to behave irrationally. To help reduce the load for an expert setting up these values, it would be interesting to extend the system with a learning component. By analyzing feedback from users, such a component might be able to adjust the activation values automatically. The basic idea behind this is that if users collectively agree on a certain bicycle being inappropriate given a certain line of questioning, its activation values are lowered to reflect this preference.

The goal of this thesis is to determine whether or not the activation values in the advisory system can be adjusted automatically using such a learning component. With the obvious need for qualitative and quantitative good feedback, the advisory system is first extended with a feedback mechanism. This mechanism records mouse clicks on bicycles as an indication for appropriateness and uses a variation on an algorithm by (Radlinski *et al.*, 2006) to obtain presentation bias free feedback. It is shown that click results can be interpreted both absolutely and relatively. In the absolute sense, the bicycle clicked most given a certain line of questioning is positioned absolutely top. In the relative sense, clicks are only interpreted as relative preferences of one bicycle over its immediate ranked predecessor or successor. Subsequently, various learning methods from the AI field of machine learning are explored for their usefulness in this context, the most promising being artificial neural networks (ANN) and Bayesian learning.

Based on this research a single layer feed-forward ANN is tested using a cost function optimized for ranking. Even though the cost function itself has no first order derivative, the ranking error it uses for each output unit proves quite useful. It turns out that using this error in a pragmatic Delta Rule approach enables a single layer ANN to learn the expert provided preferences almost perfectly. Results on the relative and absolute interpretation of feedback show learning convergence to satisfactory levels, albeit not altogether perfectly. For the relative interpretation this might however be due to a lack of insufficient feedback data gathered from the advisory system.

KEYWORDS

Ranking, machine learning, decision tree learning, entropy, artificial neural networks, Bayesian learning

PREFACE

Studying Cognitive Science and Engineering (nowadays Artificial Intelligence) since 1993, I finally reached a point where I had to think of a concrete graduation project. The main factor for the study delay I ran into was the start of my own company in 1996. Although initially started as a general purpose automation company, it evolved over time into a software house and internet company. By now Aspin internet solutions consists of four full-time employees, including me and my business partner, two part-time employees and one or two trainees on a regular base. Given my responsibilities for Aspin and limited time I sought for a graduation project combining both work and study.

Aspin specializes in internet software solutions for the Sports & Leisure industry. This specialization has led to a broad clientele among which several large bicycle manufacturers may be counted. One of these manufacturers is Koga-Miyata. Koga-Miyata designs and produces bicycles intended for the high-end market. Exclusive design and high quality are the absolute trademarks of this Dutch company. Aspin is responsible for the Koga websites and several important web-based back office applications.

Back in 2001 Koga received an e-mail from a website visitor, asking them for some sort of bicycle advisory system. With a collection of more than seventy bicycles at the time, this visitor found it quite hard to determine which Koga bicycle he should buy. The visitor stated that a system in which the answers to a couple of simple questions would lead to a selection of relevant bicycles would be really helpful. Bearing this in mind Koga approached Aspin for just such a system. Interviews with a domain expert were held, but until last year the project never really got off the ground.

Like almost any other bicycle manufacturer Koga launches a largely new collection each and every year. An advisory system of any kind would have to be adjusted every year to accommodate the new collection. This favoured the idea to allow Koga to perform this maintenance themselves. After all, who knows best which Koga bicycles are relevant for a given target use? It's obvious that such a maintenance system must be as simple and easy to use as possible. Being a bicycle manufacturer Koga should have no problem coming up with questions linking target use to relevant bicycles. Implementing them in a software system is a whole different matter altogether. Given the fact that most advisory systems use a hard-coded rule set, makes this requirement even harder and only feasible for software specialists.

In search for a minor project (the so-called *Klein Project*, a requirement prior to the graduation research) I took up this challenge and tried to come up with a workable solution. The first discussions with Koga reminded me somewhat of a classifying knowledge system in which bicycles are sorted on relevance. Relevant bicycles on top, less relevant ones lower and lower. Furthermore, I had the distinct feeling that certain forms of machine learning could prove useful, specifically decision tree learning and the use of information theory. Within the context of a minor project I studied the usefulness of these ideas. This led to the development of an algorithm able to determine the relevance of a series of questions semi-automatically. An advisory system using such an algorithm would not need a hard-coded rule set since it would be able to determine automatically what question to ask at any point in the process. As a proof of principle I build a simple prototype testing this algorithm on the 2002 collection of Koga-Miyata, for which Aspin still had questions from the domain expert (Kingma, 2008).

In the proposed design a domain expert enters all possible questions in advance. In doing so the expert enters a score for each and every answer and each and every bicycle. Scores range from totally irrelevant to highly relevant and everything in between. Using these scores it's possible to calculate a total score after a series of posed questions for each and every bicycle and sort the bicycles accordingly. The basic idea behind the algorithm is the improvement of the spreading of these scores. This is achieved by posing that question which spreads the scores of the relevant bicycles left at any given moment the most. As mentioned the algorithm was tested on the 2002 collection with a couple of example questions. Despite the limited number of available questions the system clearly showed intelligent behavior. Posing only those questions relevant to a given collection of relevant bicycles and aiming to further discriminate between them, without the need of a hard-coded preset rule set.

Although the minor project showed the potential use of the proposed mechanism, the approach also had some drawbacks. One of these was the initial load of setting the scores to each and every question and each and every bicycle. The system relies heavily on these scores

and setting them incorrectly or inaccurately lead to the posing of highly irrelevant questions at times. One way to overcome this problem, is allowing the system to somehow learn these scores and improve its performance over time. In doing so, the system should be able to adapt itself to user behavior and correct possible faulty scores automatically to accommodate this behavior. It is obvious that such a system could benefit heavily from AI research and machine learning specifically.

In need of an interesting graduation project combining both work and study, I set out to define a graduation project based on these ideas. The result is this thesis in which I investigate how machine learning can be used to improve the performance of the advisory system at hand.

ACKNOWLEDGEMENTS

In writing a thesis, one can use all the help and support one can get. Since I am certainly no exception to this rule, I want to show my sincere gratitude to a number of people who helped me during this project. I firstly want to thank Marco Wiering and Lambert Schomaker from the Artificial Intelligence department of the University of Groningen for their time and supervision of this project. The talks with Marco and Lambert were always inspiring and highly motivating. Koga-Miyata for giving me a platform for my research and allowing me to use the user feedback from their bicycle advisory system. I would especially like to thank managing director Wouter Jager, sales manager Benelux Hans Lammertsma and product manager Martin Schuttert for making time for me and this project in their very busy schedules. Former Koga public relations manager Jan de Jong, with whom the first ideas for the advisory system were discussed back in 2001. My colleague Jeen Helmantel at Aspin internet solutions for helping me out launching the advisory system for Koga, thereby kick-starting the gathering of data. Suzanne van Gelder, my Aspin business partner, for her unwaning support throughout my study and being a perfect sounding board for my ideas. And last but definitely not least, my girlfriend Fin Jilderda for her love, understanding and moral support, without which finishing this thesis would have been so much harder.

Seth Kingma
Groningen, August 2008

1. INTRODUCTION

Some things should be simple
Even an end has a start
Editors, from *An End Has A Start* (An End Has A Start, 2007)

Koga-Miyata is a Dutch bicycle manufacturer specializing in the design and production of high-end bicycles. Given the collection of 2008 with more than sixty bicycles, divided over various overlapping segments, their customers sometimes have a hard time selecting the right bicycle. To help their customers, Koga implemented a bicycle advisory system on their website. By answering a number of simple questions, customers can pick a bicycle from an on relevancy sorted list. The more appropriate a bicycle probably is, the higher it is displayed on the list.

The intelligence behind this advisory system is based on machine learning techniques, and in particular decision tree learning. Since Koga changes its collection every year, one of the key design issues was to not implement a hard-coded preset rule set. After all, reprogramming the rules each and every year would leave Koga with a relatively high total cost of ownership (TCO). Instead, the system is provided with a set of possible questions which can be put to a user of the system. These questions are presented to the system by a domain expert from Koga. The expert also links the answers to these questions to each and every bicycle by giving them activation values. These values indicate the relevancy of the bicycles when a certain answer is given to a question. The higher this value (i.e. the activation of the bicycle), the more relevant the bicycle will be considered by the system. The lower the value, the less relevant. Based on these values, a total activation value after a series of questions can be calculated. Sorting the bicycles accordingly provides users of the system with an actual advice.

As mentioned, the order in which the questions are actually put to a user has not been laid down beforehand. The expert only needs to provide the system with questions, answers and activation values. Using only this information, the system decides fully automatically which question to present a user with at any given moment. The selection mechanism is inspired by decision tree learning, and the information theory used there. The basic idea is that an improved spreading in total activation values, denotes a gain in ranking power. After all, bicycles sharing the same total activation value can not be ordered, whereas bicycles having distinct ones can. Just as is the case in decision tree learning, we can define an entropy measure indicating the gain we obtain by putting a certain question to a user. By choosing the question with the highest gain every time, the system should be able to display rational behaviour. The practical applicability of these ideas was investigated by Kingma and this led to an algorithm suitable for the advisory system to use (Kingma, 2008).

Although this so-called WIZARD algorithm was implemented successfully in the advisory system launched for Koga, a number of issues remained. One of which was its obvious dependence on expert-provided initial activation values, and the burden this entailed for a domain expert setting them up correctly. This dependence could be reduced by extending the system with a learning component. With such a component the system could for example improve on its advices by integrating user feedback on these same advices. This thesis explores the possibilities for extending the advisory system, and the WIZARD algorithm on which it is based, with just such a mechanism.

1.1. RESEARCH QUESTION

As indicated above, setting the activation values for each and every answer to the questions, linking them to the bicycles, is a manual process. Although only these values need to be modified for the advisory system to work, setting them properly can be a time consuming job for a domain expert. Since the intelligence of the system is completely based on these values, setting them improperly will cause the system to behave irrationally. Manual tuning can not be avoided altogether, since the system needs some basic intelligence to behave rational from the very first start. However, by integrating user feedback, it should be possible to modify the activation values subsequently if needed. If users unanimously disagree with a certain advice of the system, the system should ideally adjust its advice to cancel out the disagreement. The big advantage of such an approach is that the system will be able to adapt itself to user behaviour. In doing so it lessens its dependence on the initial activation values, and thereby the burden for

a domain expert setting these values very precisely. After all, the actual values will be modified based on actual user behaviour, so they do not have to be set that accurately.

Adjusting these activation values is of course actually a learning process in which the system learns to adapt its advices to the observed user behaviour. Based on this idea, the central research question in this thesis is therefore formulated as follows

> *How can the entropy-driven WIZARD algorithm improve on its advices by using feedback from its users, and in doing so, become less dependent on its initial expert-provided settings?*

In answering this question the following points of particular interest need to be addressed

- founded choice for the implementation of one or more feedback mechanisms
- founded choice for one or more learning algorithms
- application and implementation of the chosen learning algorithms to the context of the entropy-driven WIZARD algorithm
- demonstrate the chosen learning algorithms actually converge to a solution reflecting the witnessed feedback

1.2. METHODS

To be able to answer the research question, actual user feedback is needed. User feedback can be gathered in many ways. One could simply ask users for feedback (explicit feedback) or observe their behaviour and draw conclusions from that (implicit feedback). Literature shows that both types of feedback can be implemented in many ways. Based on this research, this thesis selects one or more feedback mechanisms. These are then actually implemented in the Koga advisory system, starting the gathering of serious real-life user feedback.

Given the actual set up of the WIZARD algorithm, the next step in answering the research question is exploring which learning algorithms from the field of machine learning are suitable to the current context. As is the case with feedback mechanisms, learning algorithms also comes in many flavours. Some more suitable to certain contexts than others and research is needed to explore which algorithms are probably most suited in this context and how to apply them. Drawing from this research one or more promising algorithms are selected and possibly modified for use in the current problem domain.

The final step in answering the research question is showing actual learning convergence of the choosen algorithms. This is achieved by designing a cost function measuring the total ranking error at each learning cycle, specializing it to the specific needs of the current problem domain. Reducing this cost function to satisfactory levels will show learning convergence, and hence improved advices reflecting the witnessed feedback.

1.3. SCIENTIFIC RELEVANCE FOR ARTIFICIAL INTELLIGENCE

A lot of AI research into inductive learning focuses on classification tasks. In a classification task the goal is to classify a collection of instances into a predefined set of categories. Such systems are trained on a set of labeled training instances (supervised learning) and performance is subsequently measured over a set of unlabeled instances. There are however more and more examples of real-life applications in which the actual order of a collection of instances is considered more important than their classification. For example, one need to think only of the research spent nowadays into search engine optimization, partly initiated by the popularity of the Google search engine and the PAGERANK algorithm behind it (Page *et al.*, 1998, 1999).

In the advisory system under current investigation in this thesis, the goal is also to rank a set of instances. In this setting, bicycles are sorted according to the answers to a series of questions provided by a user of the system. Relevant bikes are shown on top, less relevant ones lower and lower. The WIZARD algorithm on which the advisory system is based, is inspired heavily by ideas from the AI field of machine learning. In particular inductive learning and decision tree learning. Extending the system with a learning component will also lean heavily on these and other AI research fields.

With the proposed learning component a more general framework could be developed, which could be used as background intelligence for all sorts of wizard-like applications of the kind described in this thesis. Possibly such a framework could also prove useful in knowledge systems. If so, this would firstly reduce the development time needed to build such systems greatly. After all, one does not need to program or maintain the rule set, and with that the intelligence of the system. Secondly, solutions based on this technology would be highly

adaptive, as opposed to systems using a hard-coded preset rule set. The system simply adapts its rankings, if users start to think differently about the order of certain instances.

1.4. STRUCTURE OF THE THESIS

Addressing the points of interest mentioned in section 1.1, the structure of the rest of this thesis is as follows. Chapter two discusses the inner workings of the advisory system as developed for Koga-Miyata, and in particular the WIZARD algorithm on which it is based. Chapter three focuses on the use of user feedback and how it can be incorporated in systems for improving system performance. In Chapter four potential learning algorithms from the field of machine learning are investigated and applied to the current context. Chapter five works out the conceptual design for the learning algorithms found relevant in chapter five. This chapter also focuses on the actual implementation of these algorithms. Chapter six treats the results obtained and the thesis concludes with an evaluation and discussion in chapter seven.

2. THE BICYCLE ADVISORY SYSTEM

Advice, advice, advice me
This shroud will not suffice
Marillion, from *The Web* (Script For A Jester's Tear, 1983)

This chapter discusses the basic concepts behind the bicycle advisory system as built within the context of the minor project as mentioned in the introduction. The system is largely inspired by decision tree learning and therefore this chapter starts with a basic treatment of this type of learning. Having a clear understanding of decision tree learning and the concepts of entropy and information gain, the chapter continues with the conceptual design of the system. This part describes how the theory can be used to build an advisory system with no hard-coded preset rule set. The chapter concludes with a discussion of the issues encountered while testing the system in an actual implementation.

2.1. INSPIRATION

As mentioned Koga markets a new collection each and every year. Therefore any advisory system built will also have to be adjusted each and every year. New bicycles are introduced and existing bicycles may be no longer in production, both requiring modifications to the system. Most advisory systems use a hard-coded rule set. In situations where this rule set does not change that often (e.g. medical diagnosis), this is not really a problem. Using a hard-coded rule set in this context however, would require programming the modifications each and every year and would leave Koga with a relative high total cost of ownership (TCO).

This realization led to the idea of allowing Koga to perform the necessary yearly maintenance themselves. Since Koga specializes in producing bicycles and not software, one of the main challenges was to develop a system in which possible modifications would not have to be programmed. As noted earlier, Koga as domain expert should be perfectly able to come up with questions relating a certain target use to relevant bicycles. Programming the necessary modifications is indeed a whole different matter altogether. Ideally, the focus of Koga should only have to be on specifying the questions and their relation to the bicycles, leaving the rest up to the system. In such a system the possible questions are not known beforehand, which makes a hard-coded rule set impossible. The $64.000 Question is of course how to create a rational behaving advisory system considering these constraints. Using machine learning and decision tree learning specifically, a possible solution may lie within reach.

Decision tree learning provides a practical method for concept learning, i.e. learning a certain target function described by a collection of training examples. Decision tree learning is one of the most widely used methods in the field of inductive inference and is particularly useful in the construction of diagnostic and advisory systems. Not surprisingly, decision tree learning has been successfully applied to a broad range of tasks from medical diagnosis to the assessment of credit risk of loan applicants (Mitchell, 1997).

2.1.1. Decision Trees

Decision trees are learned from a collection of training examples describing a certain target concept. Training results in a tree-like structure describing the training examples perfectly. This structure also allows for the classification of unseen examples. Table 2.1 lists a collection of training examples for the target concept *BicycleRacingWeather*, describing the conditions for taking your racing bicycle out for a ride. Given these examples, D7 with snow, a slippery road and a light breeze is apparently no day to ride your racing bicycle.

The decision tree in Figure 2.1 describes the training examples from Table 2.1. Every node in the tree corresponds to one of the attributes *Sky*, *Temperature*, *Road* and *Wind*. Every branch beneath a node corresponds to one of the possible values for the attribute in that particular node. The attribute *Sky* has for example four branches, one for each of its four values *Sunny*, *Hail*, *Snow* and *Rainy*. Every leaf corresponds to the ultimate classification *Yes* or *No*. Examples are classified top-down starting from the root node with attribute *Sky*. Note that in this particular case the attribute *Road* is apparently irrelevant to the target concept as described by the training examples. It is not necessary to know the value of this attribute to

describe the training examples correctly. Also note the classifications for *Temperate Cold* and *Wind Calm* for which no physical evidence was presented in the training set.

Day	*Sky*	*Temperature*	*Road*	*Wind*	*BicycleRacingWeather*
D1	Rainy	Cold	Wet	Gale	Yes
D2	Rainy	Mild	Wet	Moderate	Yes
D3	Rainy	Cold	Wet	Light	Yes
D4	Snow	Cold	Slippery	Moderate	No
D5	Hail	Mild	Slippery	Moderate	No
D6	Sunny	Hot	Dry	Calm	No
D7	Snow	Cold	Slippery	Light	No
D8	Sunny	Warm	Dry	Light	Yes
D9	Sunny	Mild	Dry	Moderate	Yes
D10	Sunny	Warm	Dry	Calm	Yes
D11	Rainy	Mild	Wet	Storm	No
D12	Hail	Mild	Wet	Light	No

TABLE 2.1
Training examples for the target concept *BicycleRacingWeather*.

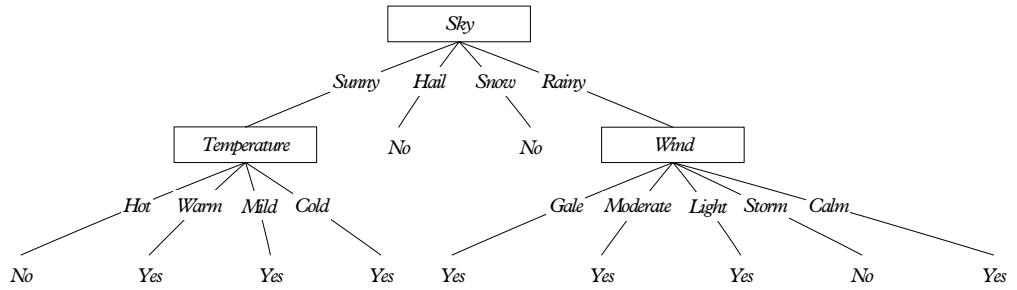


FIGURE 2.1
Decision tree for the target concept *BicycleRacingWeather* describing the training examples from Table 2.1. Note the classifications for *Temperature Cold* and *Wind Calm* for which no evidence was presented, and the absence of the attribute *Road*, which, apparently, is not needed for classifying the training data correctly.

The power of decision trees lies in their ability to classify unseen instances. For example, given the conditions for D20 in Table 2.2, classification will follow the branches *Sunny* for attribute *Sky* and *Warm* for attribute *Temperature*, ultimately labeling D20 as a good day to ride your racing bicycle.

Day	*Sky*	*Temperature*	*Road*	*Wind*	*BicycleRacingWeather*
D20	Sunny	Warm	Dry	Calm	?

TABLE 2.2
Is D20 a good day to take your racing bicycle out for a ride?

2.1.2. Constructing Decision Trees

Most algorithms used for constructing decision trees are variations on a core algorithm employing a top-down, greedy search through the hypothesis space of possible decision trees. The ID3 algorithm by Quinlan (Quinlan, 1986) and its successor C4.5 (Quinlan, 1993) are excellent examples of this approach. A simplified version of the standard ID3 algorithm, specialized to learning Boolean-valued functions, is described in Table 2.3 (Mitchell, 1997).

The central question in the algorithm is of course how to select the attribute to test for each and every node in the ultimate decision tree. To answer this question it is necessary to introduce a commonly used measure in information theory, called *entropy*. Using this measure it is possible to define a statistical property, called *information gain*. Information gain measures how well an attribute separates the training examples according to their target classification. ID3 uses this value to determine the most discriminating attribute for each node in the tree.

ID3(*Examples*, *TargetAttribute*, *Attributes*)

Examples *are the training examples*, TargetAttribute *is the attribute whose value is to be predicted by the tree.* Attributes *is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given* Examples.

- Create a *Root* for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = –
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *TargetAttribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have v_i for A
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *TargetAttribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *TargetAttribute*, *Attributes*–{A})
- End
- Return *Root*

* The best attribute is the one with highest *information gain*, as defined in Equation (2.3)

TABLE 2.3
Summary of the ID3 algorithm specialized for learning Boolean-valued functions.

2.1.2.1. Entropy

Commonly used in information theory, entropy characterizes the (im)purity of an arbitrary collection of examples. Given a collection S with positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is defined by

$$Entropy(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_- \tag{2.1}$$

where p_+ is the proportion of positive examples in S and p_- the proportion of negative examples. In all calculations involving entropy the value $0 \log_2 0$ is defined to be 0.

For example, take the collection training instances from Table 2.1 consisting of 6 positive and 6 negative examples (which can be formulated as [6+, 6–]). The entropy of this collection may then be calculated as

$$Entropy([6+,6-]) = -\left(\frac{6}{12}\right)^2 \log\left(\frac{6}{12}\right) - \left(\frac{6}{12}\right)^2 \log\left(\frac{6}{12}\right) = 1$$

As can be seen from this example, the entropy is 1 when there are just as many positive as there are negative examples in S. The entropy is 0 when all examples belong to the same class. In all other cases the entropy value will vary between 0 and 1.

The discussion above only handles the special case where the target classification is Boolean. More generally, if the target attribute can take on c different values, the entropy of S relative to this c-wise classification is given by

$$Entropy(S) \equiv \sum_{i=1}^{c} - p_i \log_2 p_i \tag{2.2}$$

where p_i represents the proportion of examples belonging to class i. Since the target attribute can now take on c possible values, the maximum entropy can now be as large as $\log_2 c$.

2.1.2.2. Information Gain

Given this concept of entropy, we can now define a measure of the effectiveness of an attribute in classifying the training examples. This can be done by determining the so-called *information gain* of an attribute. The information gain of an attribute is basically the expected reduction in entropy caused by partitioning the examples according to that attribute. The information gain $Gain(S, A)$ of an attribute A, relative to a collection S of examples, is formally defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.3}$$

where $Values(A)$ is the collection of all possible values for attribute A, and S_v is the subset examples of S for which attribute A has value v, i.e. $S_v = \{s \in S \mid A(s) = v\}$. The first term in Equation (2.3) is just the entropy of the original collection S. The expected entropy as defined by the second term, is the sum of the entropies of the subsets S_v, weighted by the fraction of examples $|S_v|/|S|$ that belong to S_v. Therefore, $Gain(S, A)$ measures the expected reduction in entropy caused by knowing the value of attribute A. The ID3 algorithm calculates the information gain of all the attributes and always selects that attribute that has the highest information gain.

Take for example the training instances from Table 2.1. To determine the root node of the decision tree, ID3 first calculates the information gain for each of the four candidate attributes *Sky*, *Temperature*, *Road* and *Wind*. The attribute *Sky* can take one of the values *Sunny*, *Hail*, *Snow* and *Rainy*. Of the 12 examples 4 take the value *Sunny* for *Sky*. Of these 4 examples 3 are positive and 1 is negative ([3+, 1–]). Likewise, setting the values *Hail*, *Snow* and *Rainy* for attribute *Sky* produces the subsets [0+, 2–], [0+, 2–] and [3+, 1–] respectively. The expected information gain by sorting on attribute *Sky* is then calculated as follows

$$\begin{aligned}
Values(Sky) &= Sunny, Hail, Snow, Rain \\
S &= [6+,6-] \\
S_{Sunny} &\leftarrow [3+,1-] \\
S_{Hail} &\leftarrow [0+,2-] \\
S_{Snow} &\leftarrow [0+,2-] \\
S_{Rainy} &\leftarrow [3+,1-]
\end{aligned}$$

$$\begin{aligned}
Gain(S, Sky) &= Entropy(S) - \sum_{v \in \{Sunny, Hail, Snow, Rainy\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - \left(\frac{4}{12}\right) Entropy(S_{Sunny}) - \left(\frac{4}{12}\right) Entropy(S_{Rainy}) \\
&= 1 + \left(\frac{1}{2}\right)^2 \log\left(\frac{3}{4}\right) + \left(\frac{1}{6}\right)^2 \log\left(\frac{1}{4}\right) \\
&\approx 0.459
\end{aligned}$$

The information gain values for the other three attributes are calculated along the same lines and amount to

$$\begin{aligned}
Gain(S, Temperature) &\approx 0.262 \\
Gain(S, Road) &\approx 0.325 \\
Gain(S, Wind) &\approx 0.270
\end{aligned}$$

As can be seen, the information gain of attribute *Sky* is the highest. Therefore ID3 selects this attribute for the root node of the decision tree. The attributes of the other nodes in the tree of Figure 2.1 are calculated the same way.

2.2. CONCEPTUAL DESIGN AND IMPLEMENTATION

As mentioned earlier, one of the key challenges in designing the advisory system is to make the subsequent maintenance of the system as easy as possible. Again, Koga should be able to perform the yearly needed maintenance themselves, just by specifying the questions and their relations to the bicycles. The biggest problem is of course determining which of the available questions to present a user of the system with and in what sequence.

One can understand intuitively that the answer to a certain question highly determines the next relevant question. For example, someone who is clearly interested in a racing bicycle, should not be bothered with questions whether he or she wants to go shopping with the bicycle. Furthermore, selecting which question to put to a user, reminds somewhat of selecting which attribute to use for a certain node in a decision tree as shown previously. In this section these ideas are further developed to finally arrive at an algorithm suited for building an advisory system with no hard-coded preset rule set.

2.2.1. Activation Values

Some bicycles are more suited for certain purposes than others. A good example of this is the concept 'holiday bicycle'. It is clear that the racing bicycles in the Koga 2008 segment *Race* are certainly not suited for this purpose. And although front luggage carriers can be mounted on most of the bicycle frames in the segment *Light Touring*, the bicycles in the segment *Trekking* are generally more suited for taking your bicycle on a biking holiday.

This natural arrangement can be expressed by assigning real-valued activation values in the range [–1, +1] to the bicycles. Bicycles with high activation values are well suited for the target purpose as intended by a user. Bicycles with low values less. The goal of the advisory system is then to modify the activation values in such a way that ultimately the most suited bicycles end up with the highest activation values. The use of these activation values is of course directly linked to the questions available to the advisory system. Bicycles are therefore assigned activation values for every answer to every question. These values, together with the questions, are provided by the domain expert. To make matters a little bit more complex, the advisory system assumes two types of questions. The first type consists of questions to which users can respond to with only one possible answer (e.g. *What is your gender?*, although even this question could prove problematic in some cases). The second type consists of questions to which multiple answers are possible (e.g. *What descriptions suit your bicycle best?*). The activation value for the first type is simply the activation of the selected answer. The value for the second type is calculated by averaging the activation values of the selected answers. Noting that the first type is actually a special case of the second type, we can now formally define the actual activation value a_b for a certain bicycle b to a question q by

$$[a_b]_q \equiv \frac{\sum_{r \in Responses(q)} [a_b]_{q_r}}{|\, Responses(q) \,|} \tag{2.4}$$

where $[a_b]_{q_r}$ denotes the activation value for bicycle b for answer r to question q as provided by the domain expert. The set *Responses*(q) is a subset of the possible answers to q, *Answers*(q), and consists of the answers actually provided by the user to q. Figure 2.2 displays the actual online management tool developed for Koga to configure the questions and the activation values. Domain experts from Koga only need to focus on these questions and activation values to provide the advisory system with its intelligence.

In the original prototype a Boolean setup was used, in which all questions were answered with a simple *Yes* or *No*. This constraint forced the creation of interdependent questions. Since the actual posing sequence of the questions is not preset, this led to irrational behaviour of the system. For example, in denying the question *Do you want suspension?*, the system would sometimes continue with the question *Do you want full suspension?*. This problem proved easy to solve by allowing multiple non-Boolean answers to the questions. For example, the two interdependent suspension questions could now be eliminated and combined into a single one with the three answers: *Yes, full suspension*, *Yes, but frontal suspension only* and *No*.

Also note in Equation (2.4) that it is absolutely necessary to define separate activation values for all the possible answers to a certain question. Consider for example again the question *What is your gender?* with possible answers *Male* and *Female*. Obviously, answering

Male should remove all female frames from consideration. However, answering *Female* shouldn't remove all male bicycles since women may want to ride a male frame for certain types of bicycles (e.g. the bicycles in the *Race* and *Trekking* segments).

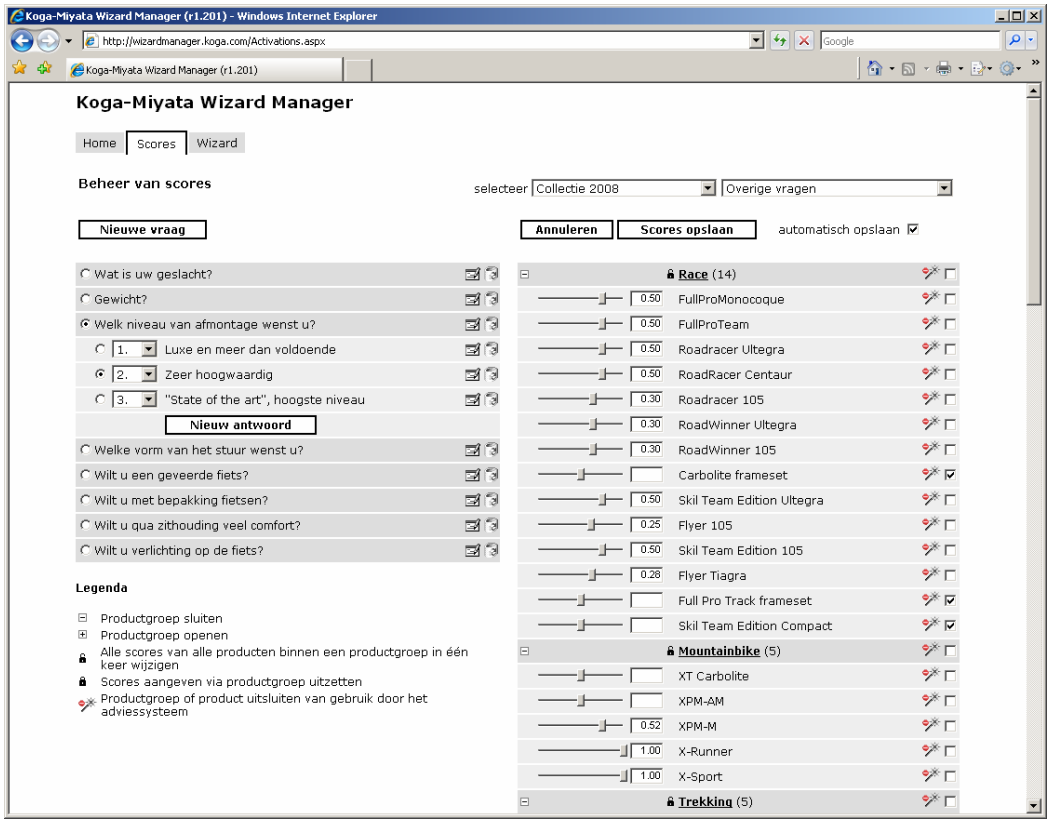


FIGURE 2.2
Managing the activation values. The possible questions are managed on the left side, the activation values of the bicycles to these questions on the right side. In this example the activation values for the question *What level of assembly would you like?* and its possible answer *Very high quality* are configured. Activation values are set by dragging a slider control or simply entering the activation value for the bicycle. Note that a domain expert does not need to set the order of the questions.

The use of activation values in this way calls for a function that can calculate the total activation value after a series of answered questions. A number of constraints need to be taken in consideration when selecting such a function. Firstly, the function must yield values in the range [–1, +1]. Secondly, it must increase the activation value when the answer to a question has a higher activation value than the calculated total activation value so far, and decrease it otherwise. Finally, is should decrease the influence of later questions compared to earlier ones. After all, one can assume that after a series of questions the number of relevant bicycles match the targeted purpose better and better. Subsequent questions should then aim at discriminating between these bicycles and not distort the basal arrangement (suitable or not). There are of course numerous functions conceivable that will satisfy the above requirements. In the advisory system the unweighted arithmetic average is used

$$[A_b]_n = \frac{1}{n}\sum_{q=1}^{n}[a_b]_q \qquad (2.5)$$

where $[A_b]_n$ represents the total and averaged activation value of bicycle b after n questions. Given computational considerations the average activation value for $n \geq 1$ is calculated using

$$[A_b]_{n+1} = \frac{1}{n+1}(n[A_b]_n + [a_b]_{n+1}) \tag{2.6}$$

2.2.2. Automatic Selection of the Most Relevant Question

After all the questions have been configured and all their activation values have been set, the system must be able to continually determine the next most relevant question. As indicated earlier the selection of this question reminds a bit of selecting an attribute for a particular node in a decision tree. Similar to a decision tree we want to select just that question that has the highest gain. However, gain in this context is not defined by classification to a preset collection of target categories, but to an enhanced spreading of total activation values. Consider for example a set of bicycles having the same activation value after a series of questions. We now want to put just that question to a user that will yield the most distinct total activation values. After all, these distinct values immediately yield a sorting of the bicycles, which was not present before the question was put to the user. It is clear that the use of entropy can prove helpful in selecting that question.

2.2.2.1. Calculation of the Entropy

In determining which question yields the most distinct activation values, we need a measure for the actual spreading of these values in the first place. Entropy as defined by Equations (2.1) and (2.2) could be such a measure. However, it can not be applied directly in this context since we have no predefined set of target categories to classify to. Remember that the activation values themselves can take on any real number in the range [–1, +1] and it is not at all clear to what category a certain activation value belongs to.

Again, we can look at decision trees for a solution to overcome this problem. Fayyad demonstrated an interesting extension to decision trees enabling them to incorporate real-valued attributes (Fayyad, 1991). In this extension a new Boolean attribute A_c is defined for each real-valued attribute A and training examples will be classified true if the value of $A<c$, and false otherwise. Consider for example a real valued attribute *Temperature* with real values ranging from –20° to 40° Celsius. One could now define a new attribute $Temperature_{30}$ which is true if the value of *Temperature* is lower than 30° Celsius, and false otherwise. Furthermore, Fayyad even showed how the value of c can be determined given a set of training examples. When considering the real attribute A, the ID3 algorithm can now consider the classification according to A_c and compare this with the other attributes without any further modification to its workings.

Borrowing from this idea, we could segmentize the range of total activation values and classify the values according to which segment they belong. To do this, the activation values are first normalized from the range [–1, 1] to the range [0, 1] according to

$$[A_b]'_n = \begin{cases} 0 & \text{if } |\max(A_n) - \min(A_n)| = 0 \\ \dfrac{|[A_b]_n - \min(A_n)|}{|\max(A_n) - \min(A_n)|} & \text{if } |\max(A_n) - \min(A_n)| \neq 0 \end{cases} \tag{2.7}$$

where $[A_b]'_n$ represents the normalized activation value for bicycle b after n questions. The operators $\max(A_n)$ and $\min(A_n)$ denote the maximum and minimum activation value after n questions and $|a - b|$ the absolute distance between the values a and b. The range [0, 1] is now divided into c segments and we count how many activation values are located in each of the c segments. Using the normalized activation values, the correct index i of the corresponding segment can easily be determined according to

$$i = \begin{cases} c & \text{if } [A_b]'_n \times c >= c \\ [A_b]'_n \times c & \text{if } [A_b]'_n \times c < c \end{cases} \tag{2.8}$$

Based on this segmentation S and Equation (2.2), we are now in a position to calculate an actual entropy for a given set of total activation values after a series of n questions

$$Entropy(A_n) \equiv \sum_{i=1}^{c} -\frac{S_i}{|A_n|}^2 \log \frac{S_i}{|A_n|} \tag{2.9}$$

where S_i and $|A_n|$ represent the number of activation values in segment i and the total number of activation values respectively.

2.2.2.2. Information Gain

Using Equation (2.9) to calculate the entropy of a set of activation values, we are now able to compare questions. However, since activation values are entered for each possible answer to a question, we need to calculate the entropy for each of these answers separately. The final entropy yielded by a certain question is then a combination of the entropies of the various answers to that question. As always, we can use different methods in doing so. We could for example just take the minimum (worst case) or maximum entropy (most optimistic). The advisory system chooses to use the average entropy over all the possible answers to a question. Given a set of total activation values A_n after a series of n questions, the expected entropy yielded by putting a new question q to a user is therefore defined by

$$Entropy(A_n, q) \equiv \frac{\sum_{r \in Answers(q)} Entropy(A_r)}{|Answers(q)|} \tag{2.10}$$

where $Answers(q)$ represents the collection of possible answers to question q, and A_r the collection activation values after answering question q with answer r in accordance with equation (2.5).

Similar to deciding what attribute to choose for a particular node in a decision tree, we want to select that question that yields the highest gain. Gain in this context can be defined as the improved spreading of activation values compared to the situation before a certain question was put to a user. A measure for the expected spreading after presenting a user with a certain question is given by Equation (2.11). Using this measure we can now define the information gain (or loss) for a set of total activation values A_n after a series of n questions and a new question q simply by

$$Gain(A_n, q) \equiv Entropy(A_n, q) - Entropy(A_n) \tag{2.11}$$

If $Entropy(A_n,q)$ is larger than $Entropy(A_n)$, the spreading of the activation values has actually improved. In this case we have more distinct activation values than we started with and therefore we gain ranking power. If it is smaller, the spreading worsened. In this case we have more activation values actually sharing the same value (i.e. more activation values sharing the same segment) and therefore we loose ranking power. From this discussion it is clear that the most relevant question q we want to select is the one with the highest $Gain(A_n,q) > 0$.

2.2.2.3. Determination of the Most Relevant Bicycles

Defining what makes a bicycle relevant is actually a fuzzy concept given the current use of activation values. It is obvious that bicycles with higher total activation values are more appropriate than bicycles with lower ones, but at what activation threshold does a bicycle stop being relevant? To be able to spread the most relevant bicycles up to a certain point, we need to define the concept of *relevant bicycles* precisely. Using the normalized total activation values from Equation (2.7) we can easily formalize this concept as the collection R of relevant bicycles after n questions by

$$R \equiv \{b \in B \mid [A_b]'_n \geq sp\} \tag{2.12}$$

where B represents the collection of all bicycles, $[A_b]'_n$ the normalized total activation value for bicycle b after n questions, and $sp \in [0, 1]$ a so-called significance parameter. With sp set to 0, the system will consider each and every bicycle relevant. Set to 1, the system will only consider

the bicycles sharing the same maximum activation value. For example, consider setting the value of *sp* to 0.60. In this case all bicycles with normalized activation values within 40% of the bicycle with the highest activation value are regarded as relevant. The higher the value of *sp* the stricter the system will thus behave in considering a bicycle relevant or not.

2.2.3. The WIZARD Algorithm

Based on the ideas presented above, we can now formulate the algorithm for the advisory system to use. A summary of this WIZARD algorithm is displayed in Table 2.3. All the algorithm needs are three sets of bicycles, questions and activation values linking the answers to the questions to the bicycles, and two parameters *c* (the number of segments) and *sp* (the significance parameter). Note how the algorithm decides fully automatically which question to put a user to at any given point. The order in which these questions are presented to the user has not been laid down beforehand. An implementation of the advisory system based on the WIZARD algorithm will therefore satisfy the basic requirement we set out to meet: easy maintenance by not using a hard-coded preset rule set.

Although not explicitly mentioned in the algorithm, all questions in the advisory system have a default answer indicating *Don't care*. When a user selects this answer, the corresponding question is removed from the pool of possible questions still left to present the user with (just as it would have been if an actual expert-provided answer had been given to it). However, when calculating the total activation value for a bicycle, this question is ignored altogether. The total activation is therefore based on actual responses to the questions, ignoring the *Don't care* responses. Consider for example a situation in which 6 questions have been put to a user, of which 2 were answered with *Don't care*. In this case 4 actual answers have been provided, setting n in Equation (2.5) to 4, averaging only the 4 questions for which these answers were given. In other words, *Don't care* responses do not decrease the total activation value for the bicycles under consideration.

Another important thing to note is, that during each iteration the total activation values for each and every bicycle is calculated, also for those considered irrelevant at a certain point. This allows bicycles not taken into consideration earlier to 'bubble up' when more evidence is gathered about the intended target use.

WIZARD(*Bicycles*, *Questions*, *Activations*, *c*, *sp*)

Bicycles *is a list of all the bicycles that the wizard must rank.* Questions *is the list of all questions available to the wizard as provided by a domain expert.* Activations *is the list of all activation values as provided by the domain expert, linking answers to questions to bicycles. The value of* c *determines the number of segments to use when calculating the entropy using Equation (2.9). The value of* sp *denotes the significance parameter used in determining relevant bicycles. WIZARD will determine the next relevant question from* Questions *to put to a user if any, and will rank the bicycles in* Bicycles *according to the answers provided by this user.*

- *QuestionsPut* ← ∅
- While *Questions* ≠ ∅, Do
  - Given *QuestionsPut*, *Activations*, and the provided answers to these questions, calculate for every bicycle *b* in *Bicycles* the total activation value according to Equation (2.5)
  - *R* ← the set of relevant bicycles according to Equation (2.12) ranked on total activation value
  - Present the user with the ranked list *R*
  - A_R ← the set of total activation values for the bicycles in *R*
  - For each question *q* in *Questions*, Do
    - Calculate the gain $g \leftarrow Gain(A_R, q)$ according to Equation (2.11)
  - g_{max} ← the highest attainable gain *g*
  - If $g_{max} > 0$
    - Present the user with question q_{max} ← the question *q* with the highest attainable gain g_{max}
    - Record the answer provided by the user to q_{max}
    - *Questions* ← *Questions* $- \{q_{max}\}$
    - *QuestionsPut* ← *QuestionsPut* $\cup \{q_{max}\}$
  - Otherwise Stop since no improvement can be achieved
- Stop since no questions are left to present the user with

TABLE 2.3
Summary of the WIZARD algorithm behind the advisory system.

2.2.4. Implementation

During the minor project mentioned in the preface, the WIZARD algorithm was put to a first test in a simple prototype advisory system (Kingma, 2008). The prototype was tested using parameter values c=100 and sp=0.80. Selecting the value for the sp-parameter took some tuning and depends strongly on the activation values provided by the domain expert. Although the test questions were limited, the system clearly showed rational behaviour aimed at discriminating between the most relevant bicycles. This prototype has by now evolved into a fully functional advisory system available on the website of Koga on http://www.koga.com. Figure 2.3 displays a typical session with the system.

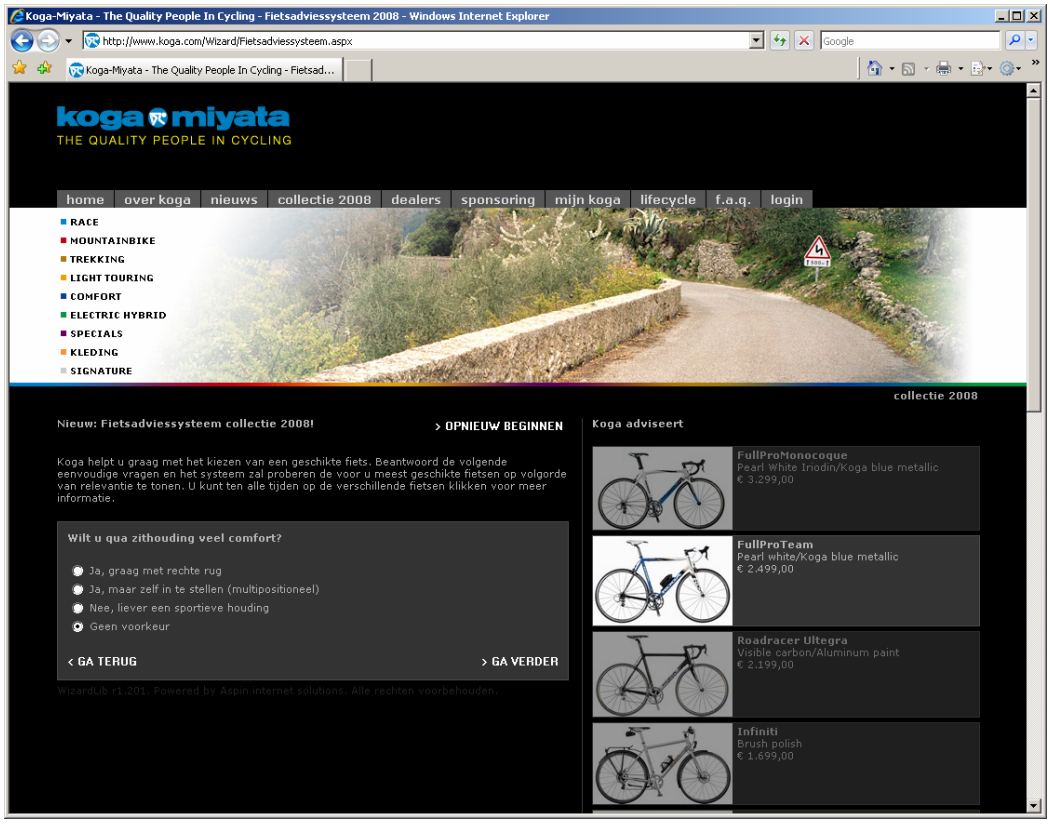


FIGURE 2.3
The advisory system at work as implemented on http://www.koga.com. Questions put to a user are displayed on the left side of the screen, the actual advice on the right side. The user in this session hovered with his mouse over the *FullProTeam*, highlighting it in the interface.

In this implementation a minor modification to the original algorithm was made. This modification entailed the use of obligatory questions in the first step of the advisory system. These questions are configured just as any other question, but users have to answer these questions first when starting a session with the advisory system. The system uses these answers to create a basic set of relevant bicycles to start with. This provides the system with some immediate guidance what to ask for next, thereby displaying rational behaviour from the start. The system currently uses two such questions: *What is your gender?* and *What descriptions suit your bicycle best?*. The activation values for these obligatory questions are calculated the same way as for all subsequent questions, leaving inner working of the algorithm fully intact.

2.3. FUTURE WORK

Although the advisory system has been launched successfully for Koga, certain points of particular interest remain. The most important one under current investigation in this thesis, is of course extending the system with a learning component. Learning observed user preferences over the bicycles, will lessen the burden for a domain expert setting the initial activation values.

After all, these values are subsequently changed by the learning algorithm to reflect the observed behaviour. In addition to this, several other points of interest validate future research.

2.3.1. Weighted Activation Values

In the current advisory system the total activation value for a bicycle is the unweighted average of all the separate activation values of the answers provided by the user. As seen in Equation (2.5) all the answers contribute their activation values evenly to the total activation value. One can however imagine a scenario in which a user considers the answers to certain questions more important than the answers to others. By weighing these answers more heavily, the system should be able to converge more quickly to the bicycles most probably relevant for the given user. This feature could be implemented by allowing users to indicate how important they consider certain questions on a scale from e.g. 1 (not important) to 5 (most important). This value can be incorporated in the advisory system relatively simple by redefining Equation (2.5) as follows

$$[A_b]_n = \frac{\sum_{q=1}^{n} w_q [a_b]_q}{\sum_{q=1}^{n} w_q} \tag{2.13}$$

where $w_q \in [0, 5]$ represents the user provided weighing factor for question q.

Using weighted activation values does however have two minor drawbacks. Firstly, it will need an implementation in the user interface, making the system slightly less easy to use. Secondly, since Equation (2.6) can no longer be used, the total activation for each and every bicycle has to be calculated from scratch every step along the way. This will make the system somewhat slower.

2.3.2. Weighted Expected Entropy

In calculating the expected entropy for a certain question according to Equation (2.10), each answer contributes evenly to the overall entropy. It is however possible that in practice certain answers are structurally given less frequently than others. By taking into account the a priori chances that a certain answer is selected or not, we can reflect this user behaviour and decrease the influence of less likely answers. Equation (2.10) should then be modified to

$$Entropy(A_n, q) \equiv \sum_{r \in Answers(q)} p_{q_r} Entropy(A_r) \tag{2.14}$$

where p_{q_r} represents the total proportion of answering question q with answer r over all users. For example, if we put a question q ten times to different users, and in four instances answer r was selected, the proportion p_{q_r} would be (4/10). Given enough user data these proportions will converge to the a priori chances as mentioned.

2.3.3. Entropy and Order

The WIZARD algorithm from Table 2.3 deploys an entropy measure to determine the spreading of a collection activation values. The basic idea behind this being that an improved spread in activation values is a gain in ranking power. Both the prototype and the actual advisory system launched for Koga, have shown that this method provides a workable solution given the constraints set at the start of this chapter. Using entropy in a ranking system has however one major drawback. The fact is that entropy as defined in Equation (2.9) does not state anything about the underlying order of the various activation values in the c segments. Put otherwise, two collections of activation values can have the same entropy value, but still a different underlying order. The prototype did not suffer noticeably from this problem, but the actual advisory system launched for Koga at times does. Although this certainly does not compromise system performance, the advices from the system can benefit from solving this issue.

Consider for example a simple scenario in which c=4 and we have 4 bicycles (b_1, b_2, b_3, b_4), bicycle b_1 in segment 1, b_2 in segment 2 and so on. Now suppose we have only

one question left to put to the user, and that it yields a slightly different segmentation in which bicycles b_1 and b_2 are now switched. Given the fact that the new situation yields the same entropy value, the question will not be presented to the user. It is clear that putting the question to the user, would have certainly provided the user with a better advice.

To overcome this problem the system should not only consider the entropy value, but also measure the change in the order of the bicycles after putting a question to the user. By combining these two values the system should be able to tackle this problem and improve on its ranking power.

2.4. SUMMARY

This chapter showed how to build a rational behaving advisory system with no hard-coded preset rule set. Using an entropy measure to define the amount of spread in a set of activation values of possibly relevant instances, such a system is able to fully automatically select the next relevant question to put to a user at any moment. The system has however some minor drawbacks. One of which is its strong dependence on its initial activation values as provided by a domain expert. One way to make the system less dependent on these values, is enabling it to adapt them, if users structurally disagree with the advices of the system. One of the key issues in developing such a mechanism, is of course how to actual gather feedback from users and when to decide whether a given advice disagrees with this feedback or not. The next chapter focuses on the gathering of user feedback and how such a mechanism could be implemented in the advisory system.

3. FEEDBACK

I can change, I can change, I can change
But who do you want me to be?
Foo Fighters, from *Stranger Things Have Happened* (Echoes, Silence, Patience & Grace, 2007)

This chapter focuses on the use of feedback to improve the performance of classification and ranking systems. The basic idea behind using feedback is to take the advices that are initially returned by the advisory system and to use information about whether or not those advices are sorted correctly to modify future advices. With the popularity rise of the Internet this field has received a lot of attention lately as a means to improve the performance and quality of search engine results. The chapter starts with a brief discussion of two basic types of feedback: explicit and implicit feedback. Explicit feedback is feedback gathered manually and although of high quality relatively hard to come by. Implicit feedback is gathered by just observing and recording the interaction of users with a system. Research shows that, given sufficient implicit feedback data, even this behavioural data can be used to stake classification or preference claims. The chapter concludes with a validation for the use of implicit feedback in the advisory systems and a way to implement this use to obtain bias free feedback.

3.1. EXPLICIT FEEDBACK

Explicit feedback is obtained by asking users of a system explicitly to assess the performance of the system they use. Thus in an explicit feedback setting users basically have two tasks. The first being the primary use of the system to satisfy their needs in using the system in the first place. The second to assess the system itself. Explicit feedback has been used extensively in information retrieval research and for ranking problems in particular. In such problem domains users are asked to provide judgments as to the relevance of particular documents to particular queries. These preferences can then be used to improve the performance of the system. Explicit feedback normally yields clean training data of high quality. However it is usually prohibitively expensive and time consuming to obtain due to the human effort involved.

Note that explicit feedback does not necessarily involve actually interviewing the user. In an online setting, such as the advisory system, we could build a mechanism which allowed users to provide feedback concerning the advices they obtained from the system. For example, they could rate how they felt about a certain advice, or which two or three bicycles are most interesting for them at any given point. Note that such a setup still requires a user to perform a second task next to the one the system is used primarily for. Certainly in the absence of some rewarding system, users can be expected not to be very inclined to provide this feedback. Research indeed confirms this feeling for these types of applications.

3.2. IMPLICIT FEEDBACK

In recent years another approach acquired more and more attention and that is the use of implicit feedback. As mentioned in the introduction of this chapter, the research community has taken a particular interest in search engine optimization using implicit feedback. For example, Fox *et al.* performed an interesting study to the relation between explicit and implicit feedback in this setting and what implicit measures were most strongly associated with user satisfaction (Fox *et al.*, 2005).

Implicit feedback is gathered by unobtrusively observing a user's interaction with a system. All the while recording what the user looks at, for how long, what is selected, saved, printed or clicked, etc. Users are not provided with a second task to assess the performance of the system. In most settings users are not even told that their behaviour is being logged. The primary advantage of using implicit feedback is that such techniques remove the human effort needed in providing the feedback. The cost of obtaining this type of feedback is therefore much lower than that of explicit feedback. Because of this implicit feedback can be gathered in much higher quantities than explicit feedback. This property is especially important given the fact

that implicit feedback is generally thought to be less accurate than explicit feedback. But by gathering sufficient data we could settle for the general consensus using the Law of Large Numbers[1]. The basic idea being that if a sufficient number of users agree on a certain change, it is probably safe to use this information as such and modify the system accordingly to improve its performance.

A study performed by White *et al.* investigated the use and effectiveness of using implicit relevance feedback in general and found that search task complexity, search experience of the user and the stage in the search all contribute to the utility of the technique (White *et al.*, 2005). Kelly et al. provide an excellent survey of the various techniques available for gathering this type of feedback (Kelly *et al.*, 2003).

3.3. USING FEEDBACK IN THE ADVISORY SYSTEM

Given the previous discussion of explicit and implicit feedback I opt for the use of implicit feedback in the advisory system. Especially the unobtrusive nature of this type of feedback gathering makes it quite easy to obtain. Furthermore, an implementation of explicit feedback would entail adding interface elements to the user interface, thereby making it less easy to use.

Turning to the context of the advisory system we can note that the problem domain is somewhat less complex than that of for example a search engine context. The possibilities a user has to interact with the system are constrained to the set of expert-provided questions. This narrows down the possibilities for implementing implicit feedback in the system. One obvious way to obtain implicit feedback in the system is to record the selection clicking behaviour of users in the system. If an user clicks to select a bicycle for more information, this is clearly an indication the user found the bicycle relevant in the given context. Chances are that bicycles deemed irrelevant are certainly not clicked for more information.

Research has however shown that clickthrough data is both noisy and biased. Noisy in the sense that different users will obviously have different concepts of relevance, even given the same query. Biased in the sense that the decision whether or not to click on a certain result depends on a combination of the relevance and the presentation position in the search results. That is to say, users tend to click search results top-down, disregarding to a certain extent the possible higher relevancy of documents lower in the results (Joachims *et al.*, 2005).

Given sufficient clickthrough data and the Law of Large Numbers, the noisiness of the data may prove not to be such a big problem. To overcome the presentation bias, Radlinski *et al.* proposed a mechanism for modifying the presentation of search results, called *FAIRPAIRS* (Radlinski *et al.*, 2006). FAIRPAIRS tackles the user inclination to click the results top-down by shuffling these results and recording the clicks accordingly. In doing so, more reliable relevance feedback can be gathered, i.e. feedback unaffected by presentation bias.

3.3.1. The FAIRPAIRS Algorithm

The basic idea behind the FAIRPAIRS algorithm is the partly randomization of the search results to eliminate the presentation bias, while at the same time making only minimal changes to the ranking. Consider for example some query returning the sorted result set $(d_1, d_2, \ldots, d_n)$. The order in which the results are presented to the user is now modified to elicit relevance judgments unaffected by presentation bias. We first pick $k \in \{0, 1\}$. If $k=0$, the result set is considered as the pairs $((d_1, d_2), (d_3, d_4), \ldots)$. Each pair is now independently flipped with 50% probability. The final ranking may thus end up as $((d_1, d_2), (d_4, d_3), (d_5, d_6), \ldots)$ in which only d_3 and d_4 are flipped. Similarly, if $k=1$, we do the same thing but now consider the result set as the pairs $(d_1, (d_2, d_3), (d_4, d_5), \ldots)$. The basics of this mechanism are formalized in Table 3.1.

[1] The Law of Large Numbers is a theorem in probability that describes the long-term stability of a random variable. Given a sample of independent and identically distributed random variables with a finite expected value, the average of these observations will eventually approach and stay close to the expected value. An example is the flip of a coin. Given repeated flips of a coin, the frequency of heads (or tails) will increasingly approach 50% over a large number of trials (http://www.wikipedia.org).

FAIRPAIRS(*ResultSet*)

ResultSet *is the sorted result set of n documents* (d_1, d_2, …, d_n) *of some query, where each document* d_i *is ranked strictly higher than document* d_{i+1}.

- Randomly choose $k \in \{0, 1\}$ with uniform probability
- If $k = 0$
  - For $i \in \{1, 3, 5, \ldots\}$, Do
    - Swap d_i and d_{i+1} in *ResultSet* with 50% probability
- Otherwise ($k = 1$)
  - For $i \in \{2, 4, 6, \ldots\}$, Do
    - Swap d_i and d_{i+1} in *ResultSet* with 50% probability
- Present *ResultSet* to the user, recording clicks on results
- Every time the lower result in a pair that was considered for flipping is clicked, record this as a preference for that result over the one above it

TABLE 3.1
Summary of the FAIRPAIRS algorithm for obtaining unbiased clickthrough data.

To interpret the clickthrough data gathered by FAIRPAIRS, consider again some query q resulting in the sorted result set (d_1, d_2, …, d_n). Let $d_j \triangleleft d_i$ denote that result d_j is ranked higher than d_i, and consider that the value of k is such that d_i and d_j are in the same pair. Let c_{ij} denote the number of clicks on d_i when $d_j \triangleleft d_i$, i.e. d_i is the bottom result in a pair. By randomizing the results with FAIRPAIRS c_{ij} can now be interpreted as the number of votes for $relevance(d_i) > relevance(d_j)$ and c_{ji} as the number of votes for $relevance(d_j) > relevance(d_i)$.

FAIRPAIRS makes a number of basic assumptions to stake its claim. Two of them relate to the assumed display of rational behaviour by users. Users are assumed to look for sufficient relevant results, and not skip ones they recognize and know to be relevant (the so-called assumptions of Document Identity and Relevance Score). FAIRPAIRS also assumes a strict order in the original search results, i.e. each and every result has a unique ranking value on some ranking function f_{rank}. As Radlinski *et al.* have shown that under these reasonable assumptions and given sufficient clickthrough data (for which a lower bound ε is provided), FAIRPAIRS provably gathers relevance feedback unaffected by presentation bias. Under these conditions training data gathered with FAIRPAIRS will allow a learning algorithm to converge to an ideal ranking, if one exists.

3.3.2. Application to the Advisory System: FAIRSHARES

Although FAIRPAIRS seems to provide a very sensible way for eliciting unbiased clickthrough data in the advisory system, application of the algorithm to the current context is not all that clear. The major problem is the assumption that all the results have a unique score on which they can be sorted. This assumption doesn't hold for the bicycles in the advisory system. More often than not, two or more different sets of bicycles end up sharing the same total activation value after a series of questions. The FAIRPAIRS algorithm will have to be adjusted to overcome this problem. Therefore, I propose a modified version of the algorithm, called *FAIRSHARES*.

FAIRSHARES basically works along the same lines as the original FAIRPAIRS algorithm. Again, consider a question returning the sorted result set (d_1, d_2, …, d_n) which contains multiple subsets of d_i's sharing the same value on some ranking function f_r. If, for example, $f_r(d_2)=f_r(d_3)=f_r(d_4)$, and $f_r(d_5)=f_r(d_6)$ and all other $f_r(d_i)$'s are unique otherwise, we can rewrite the original result set as (d_1, (d_2, d_3, d_4), (d_5, d_6), d_7, d_8, …, d_n). Note that the in-set sorting of the results (d_2, d_3, d_4) and (d_5, d_6) is arbitrary, e.g. sorting them as (d_4, d_2, d_3) and (d_6, d_5) would have been just as valid. This property is reflected by randomizing the order of the results in these subsets first. In doing so, each result has an even chance of being displayed on top of its subset siblings. This procedure avoids unfounded favouring of certain results over others with the same ranking score. Next, perform a FAIRPAIRS permutation to the result set, treating the subsets sharing ranking score values as separate entities. For example, for k=0, this could result in the set (((d_4, d_2, d_3), d_1), (d_7, (d_6, d_5)), (d_8, d_9), …) in which the two subsets have been randomized as above, and only the first two 'pairs' have been switched. The complete FAIRSHARES algorithm is formalized in Table 3.2.

FAIRSHARES(*ResultSet*, f_{rank})

ResultSet *is the sorted result set of n documents* (d_1, d_2, …, d_n) *of some query and* f_{rank} *is the ranking function used for sorting the documents in* ResultSet. *Documents in* ResultSet *may share the same ranking value* f_{rank}, *i.e.* $f_{rank}(d_i)=f_{rank}(d_j)$ *for some documents* d_i *and* d_j, *where* $i \neq j$.

- Let $S=(S_1, S_2, \ldots, S_m)$ be the sorted union of subsets S_v of *ResultSet* with $\{d_i \in S_v \mid f_{rank}(d_i)=v\}$ for $1 \leq i \leq n$
- Randomly reorder all d_i in the subsets S_v
- Randomly choose $k \in \{0, 1\}$ with uniform probability
- If $k = 0$
  - For $i \in \{1, 3, 5, \ldots\}$, Do
    - Swap S_i and S_{i+1} in S with 50% probability
- Otherwise ($k = 1$)
  - For $i \in \{2, 4, 6, \ldots\}$, Do
    - Swap S_i and S_{i+1} in S with 50% probability
- Present S to the user, recording clicks on results
- Every time a result in a subset S_v is clicked, record this is as a preference for that result over the other results in S_v
- Every time a result in the lower result set in a set pair that was considered for flipping is clicked, record this as a preference for that result over the ones in the set above it

TABLE 3.2
Summary of the FAIRSHARES algorithm.

As can be seen in Table 3.2 FAIRSHARES interprets clickthrough data basically the same as the original FAIRPAIRS algorithm. In contrast to FAIRPAIRS however a click on a result gives rise to a series of preference recordings. Clicking a result in a subset S_v which consists of more than one result, is regarded as a preference of this result over its sibling results. Furthermore, since FAIRSHARES swaps sets of results instead of single results, a click on a result in a lower result set considered for flipping, is regarded as a preference of this result over all the results in the set above it. In doing so it follows the original FAIRPAIRS algorithm using a minimally invasive tactic to undo the presentation bias while at the same time making only the slightest changes to the order.

There is also another possibility of interpreting the clickthrough data which is somewhat more absolute. One could claim that results clicked the most, should also be ranked highest. To undo the presentation bias a FAIRSHARES permutation is performed, but the clicks are not considered as relative preferences over result pairs, but as absolute preferences over *all* results. In such a scenario a result clicked the most would be ranked highest, and not just higher than its immediate neighbouring results. It could prove interesting to compare the FAIRPAIRS relative approach to this more absolute approach.

An interesting feature of the FAIRSHARES algorithm is the side-effect of ultimately being able to discriminate between results in a subset S_v sharing the same ranking value v. By recording the clicks on each and every result in S_v and sorting them accordingly, a natural user-preferred ranking arises automatically.

3.3.3. Implementation

The FAIRSHARES algorithm as presented in Table 3.2 has been implemented in the actual advisory system and the bicycles deemed relevant by the system are sorted accordingly. Figure 3.1 shows the implementation of anonymous implicit user feedback in the system.

The implementation showed a somewhat minor drawback of the FAIRSHARES algorithm. Many of the bicycles in the Koga advisory system were originally assigned the same activation value for many of the questions by the domain experts. This resulted in bicycles sharing the same total activation value after a series of questions. Switching subset pairs on such listings led to somewhat strange advices. Consider for example a bicycle (b_1) with top relevancy (i.e. the highest total activation value) and ten less relevant bicycles (b_2, …, b_{11}) sharing the same ranking value. If the FAIRSHARES conditions are such that these two sets are switched, the highly relevant bicycle b_1 will be listed only eleventh! Although this can be considered unwanted behaviour, it is obviously caused by less than optimal initial expert-provided intelligence. After all, if the expert distinguished the bicycles better in setting up the system, the FAIRSHARES permutation would not cause bicycles to make very large leaps in their listings. One of our initial requirements was however to just reduce the load of setting the initial activation values in the first place, not to enhance it.

This drawback should not prove to be that problematic however. After all one can assume that given enough time all bicycles will ultimately differ in their click results. These click results enable the system to display its advice using a now strict sorting. In this sorting the

most clicked and therefore most relevant bicycle is listed top indeed. It is even possible to apply subsequent FAIRSHARES permutations to this sorted list. Since all bicycles are now strictly sorted, subsequent FAIRSHARES permutations will not be as invasive as the first. This enables a continuous process of gathering clickthrough data and rearranging the advices accordingly, actually the same way Google for example continuously updates its rankings in a so-called *Google dance*.

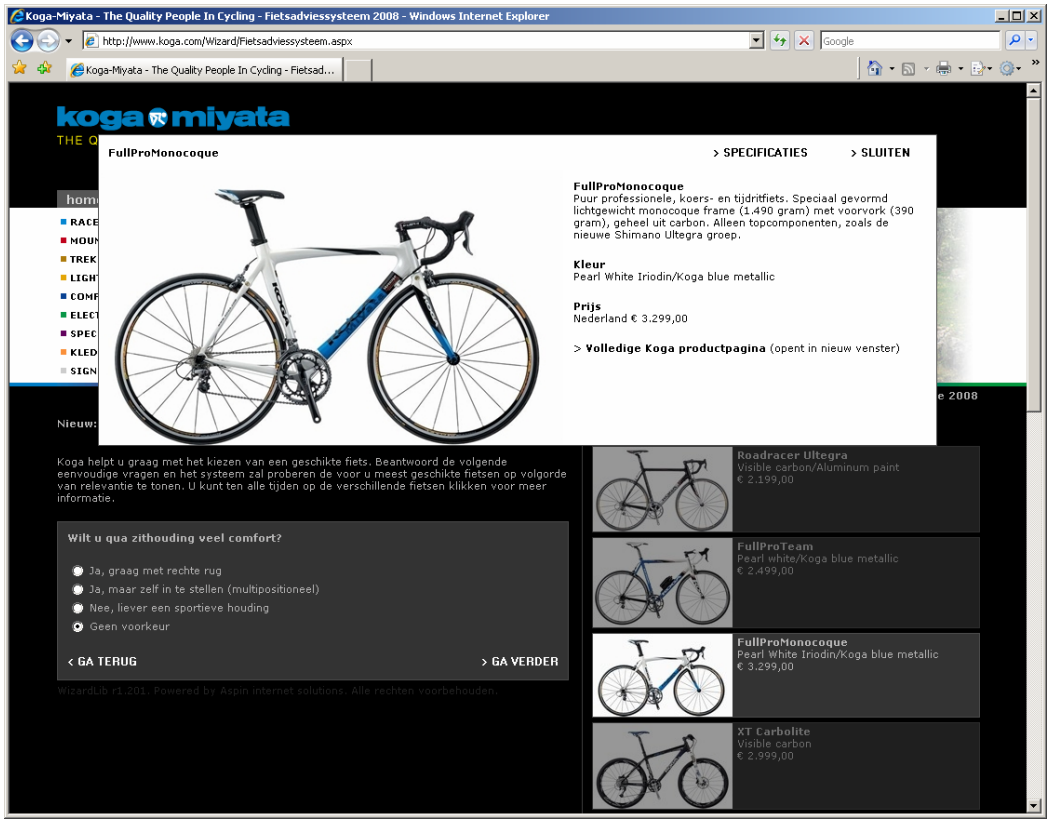


FIGURE 3.1
Using anonymous and implicit user feedback in the advisory system. In this particular instance a user has clicked on the model *FullProMonocoque* for more information. This click was recorded as an absolute preference of this model over all other models shown in the advice.

3.4. SUMMARY

This chapter discussed the use of user feedback in optimizing the performance of classification and ranking systems. Feedback can be explicit of implicit. Given its obtrusive nature, explicit feedback is relatively hard to come by. Implicit feedback however is much easier to obtain and can be just as useful given sufficient data. Implicit feedback can be gathered in many ways and the use of clickthrough data was chosen for the advisory system. To counter the presentation bias users often display, the advices of the system are permutated according to the proposed FAIRSHARES algorithm. With the clickthrough data gathered this way, the advisory system 'knows' what bicycles users are apparently most interested in after a series of questions. The bicycles clicked on most ranked top, and less clicked bicycles lower and lower. These gathered lists could be used in a supervised learning setting to adjust the activation values of the system, learning to advice the user-preferred lists over the indirect expert-provided ones. The next question is of course how to adjust these activation values exactly. Several machine learning algorithms could prove useful here and they are the topic of the next chapter.

4. LEARNING ALGORITHMS

You live, you learn
You love, you learn
Alanis Morissette, from *You Learn* (Jagged Little Pill, 1995)

The field of machine learning provides a broad range of learning algorithms. Obviously, some of these algorithms are more suitable than others in the current context. In this chapter two potentially suitable methods are discussed. The first method is the use of an artificial neural network (ANN). ANNs are partly inspired on biological learning systems and are among the most effective learning methods currently known. As will be shown in this chapter the advisory system at hand highly resembles an ANN itself. The second method is Bayesian learning. Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the dynamics of a problem domain are governed by underlying probability distributions and that optimal decisions can be made by reasoning about these probabilities and the observed data (Mitchell, 1997).

For each of the two methods a way of applying it to the advisory system at hand is provided. The chapter concludes with a discussion of the pros and cons of the algorithms. The result of this discussion is the selection of one of the algorithms for actual implementation in the system, which will be the focus of chapter 5

4.1. ARTIFICIAL NEURAL NETWORKS

The use of ANNs is partly inspired on the observation that biological learning systems are built of very complex webs of interconnected neurons. As is roughly the case in biological systems, ANNs are built out of a densely interconnected set of simple units. These artificial neurons take a number of real-valued inputs (possibly the outputs from other units) and produce a simple real-valued output (which may possibly be the input to many other units). Although it's practically impossible to simulate e.g. the complexity of the human brain with more than 10^{11} interconnected neurons, artificial systems based on such interconnectivity of even a small number of simple units prove surprisingly useful. Methods in neural network learning provide a robust approach for the approximation of real-valued, discrete-valued, or vector-valued target functions. For some problems ANNs are even among the most effective learning methods known. For example, ANNs have been successfully applied to speech, handwriting and face recognition.

ANNs are basically graphs for which many structures can be used—acyclic or cyclic, directed or undirected, single or multilayer to name but a few. The most commonly used topology is however a layered acyclic feed-forward structure. In a layered neural network the units are organized in the form of layers. The units in a layer channel their output to units in other layers by means of a weighted connection. Learning in an ANN setting corresponds to determining the right weight values for each of these connections, i.e. the edges in the graph.

This section describes single and multilayer ANNs. In a single layer setting only a layer of output neurons is used. The input layer is not counted as a separate layer in ANN terminology, since no actual calculation is performed at that level. In a multilayer setting the input layer connects to a layer of hidden neurons, which in turn connects to other hidden layers or directly to the output layer. To understand these types of networks we must first establish a basic understanding of the elementary building block of an ANN: the neuron.

4.1.1. The Artificial Neuron

A neuron is an information-processing unit fundamental to the operation of a neural network. Figure 4.1 displays the basic model of a neuron, which forms the basic building block for any ANN (Haykin, 1999).

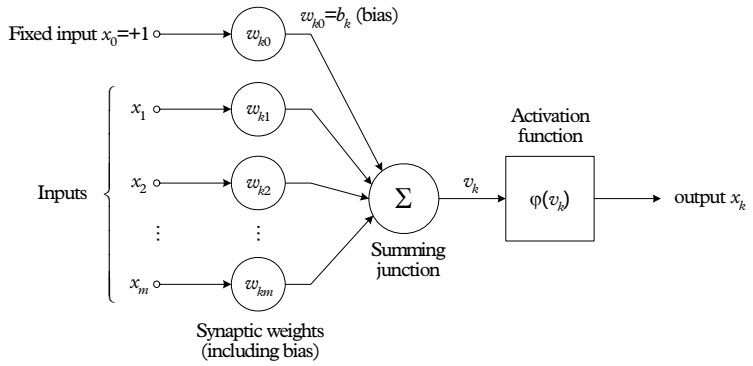
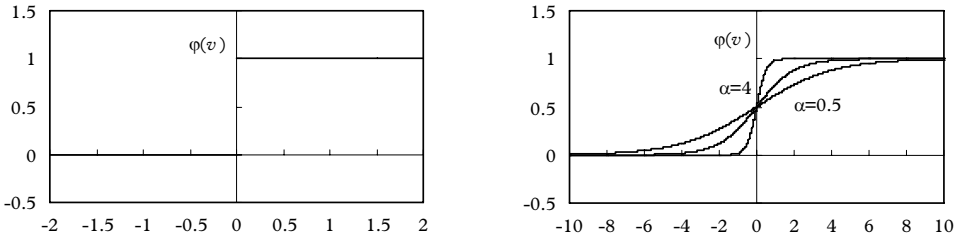


FIGURE 4.1
Nonlinear model of a neuron.

As Figure 4.1 shows three basic elements can be identified in the neuronal model. The first of which is a set of *synapses* of *connecting links*, each characterized by a weight or strength on its own. The second element is a *linear combiner* or *adder* for summing the input values weighted by their respective synapses. Given a neuron k, its input channels x_i and associated weights w_{ki}, the output of this linear combiner is defined by

$$v_k = \sum_{i=0}^{m} w_{ki} x_i \tag{4.1}$$

Note that the notation w_{ji} indicates a weight value from node i into node j. The third is an *activation function* for limiting the amplitude of the output of the neuron. This activation is sometimes also referred to as a *squashing function* in that it squashes the output signal to some finite value. There are several functions that could be used as such. Figure 4.2 displays two such functions.



Threshold function $\varphi(v)=\begin{cases}1 & \text{if } v\geq 0\\ 0 & \text{if } v<0\end{cases}$

Sigmoid logistic function $\varphi(v)=\sigma(v)=\frac{1}{1+e^{-\alpha v}}$

FIGURE 4.2
Threshold and sigmoid activation functions. Note that by setting a high enough value for α the logistic function will act as a continuous and thereby differentiable threshold function.

Neurons using a threshold function are also called *perceptrons*. The sigmoid function in Figure 4.2 defined by the logistic function

$$\varphi(v)= \frac{1}{1+e^{-\alpha v}} \tag{4.2}$$

is by far the most common form of activation function used in the construction of ANNs. Given a high enough value for α the sigmoid function behaves like the threshold function, but in contrast to it, it is differentiable. Differentiability is an important feature in training multilayer ANNs.

4.1.2. Single Layer Feed-Forward Networks

In its simplest form a layered network consists of an input layer of source nodes connecting to an output layer of output units. This topology is called a single layer network, referring to the output layer of (computational) units. The input layer is not counted as a separate layer since no actual computation is performed there.

Consider for example the training data from Table 2.1, providing a set of 12 examples of the target concept *BicycleRacingWeather*. We can construct a single layer neural network for this domain by using 2 sigmoid neurons, one for the classification *No* and one for the classification *Yes*. For each of the possible attribute values a separate input channel to these nodes is created. Each neuron therefore has 16 input channels (i.e. 4 for *Sky*, 4 for *Temperature*, 3 for *Road* and 5 for *Wind*). Figure 4.3 displays the topology of this network.

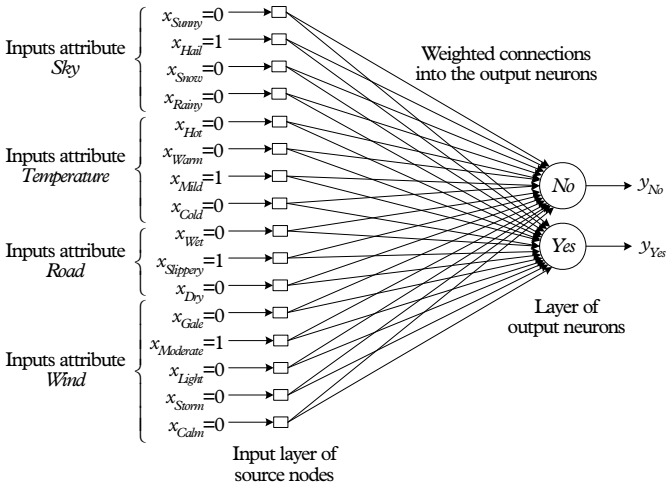


FIGURE 4.3
A feed-forward single layer neural network for the concept *BicycleRacingWeather* described by Table 2.1. In this example D5 (<*Sky=Hail*, *Temperature=Mild*, *Road=Slippery*, *Wind=Moderate*>) is fed into the input layer. All the respective input values for the observed attributes are set to 1, all other to 0.

An input value x_i is 1 if the corresponding attribute value is observed, and 0 otherwise. Omitting the attribute names for brevity, suppose these values are weighted as follows

$w_{No,Sunny} = -1.35$	$w_{No,Hail} = 2.84$	$w_{No,Snow} = 3.06$	$w_{No,Rainy} = -2.71$	
$w_{Yes,Sunny} = 1.91$	$w_{Yes,Hail} = -2.17$	$w_{Yes,Snow} = -3.03$	$w_{Yes,Rainy} = 2.30$	
$w_{No,Hot} = 4.30$	$w_{No,Warm} = -3.20$	$w_{No,Mild} = 1.13$	$w_{No,Cold} = -1.09$	
$w_{Yes,Hot} = -3.92$	$w_{Yes,Warm} = 3.57$	$w_{Yes,Mild} = -1.07$	$w_{Yes,Cold} = 1.57$	
$w_{No,Wet} = -0.25$	$w_{No,Slippery} = 3.84$	$w_{No,Dry} = -1.68$		
$w_{Yes,Wet} = 0.62$	$w_{Yes,Slippery} = -3.61$	$w_{Yes,Dry} = 1.81$		
$w_{No,Gale} = -1.11$	$w_{No,Moderate} = -2.70$	$w_{No,Light} = -0.49$	$w_{No,Storm} = 4.74$	$w_{No,Calm} = 1.67$
$w_{Yes,Gale} = 1.16$	$w_{Yes,Moderate} = 3.15$	$w_{Yes,Light} = 0.56$	$w_{Yes,Storm} = -4.26$	$w_{Yes,Calm} = -1.56$

For example $w_{No,Sunny}$ denotes the weight for the input channel for attribute *Sunny* (*Sky*) to the unit *No*. Doing the math for D5 in Table 2.1 and using Equations (4.1) and (4.2) with α=1, we can calculate the two output values y_{No} and y_{Yes} as follows

$$v_{No} = w_{No,Hail} + w_{No,Mild} + w_{No,Slippery} + w_{No,Moderate} = 2.84 + 1.13 + 3.84 - 2.70 = 5.11$$
$$v_{Yes} = w_{Yes,Hail} + w_{Yes,Mild} + w_{Yes,Slippery} + w_{Yes,Moderate} = -2.17 - 1.07 - 3.61 + 3.15 = -3.7$$
$$y_{No} = \frac{1}{1+e^{-v_{No}}} = \frac{1}{1+e^{-5.11}} \approx 0.994$$
$$y_{Yes} = \frac{1}{1+e^{-v_{Yes}}} = \frac{1}{1+e^{3.7}} \approx 0.024$$

thereby labeling D5 with *BicycleRacingWeather=No*. The labels for the other examples in Table 2.1 can be determined along the same lines. As was the case with the decision tree in section 2.1.1, this neural network can also be used to classify unseen examples. Reconsider for example the unseen instance D20 from Table 2.2 repeated here as the tuple

<*Sky=Sunny*, *Temperature=Warm*, *Road=Dry*, *Wind=Strong*>

Calculating the output values along exact the same lines we get

$$v_{No} = w_{No,Sunny} + w_{No,Warm} + w_{No,Dry} + w_{No,Calm} = -1.35 - 3.20 - 1.68 + 1.67 = -4.56$$
$$v_{Yes} = w_{Yes,Sunny} + w_{Yes,Warm} + w_{Yes,Dry} + w_{Yes,Calm} = 1.91 + 3.57 + 1.81 - 1.56 = 5.73$$
$$y_{No} = \frac{1}{1+e^{4.56}} \approx 0.010$$
$$y_{Yes} = \frac{1}{1+e^{-5.73}} \approx 0.997$$

thereby labeling D20 with *BicycleRacingWeather=Yes*, as did the decision tree.

4.1.2.1. Widrow-Hoff or Delta Rule Learning

Although the weight values in the previous example miraculously yielded the intended behaviour, one might of course wonder how these values were obtained in the first place. Several algorithms can be used to solve this learning problem. One of these approaches is based on iterative gradient descent and was invented by Widrow and Hoff in the 1960's. Their procedure constitutes a form of supervised learning, and is also known as the *Least Mean Square* (LMS) method or the *Delta Rule*. The method can be applied to any single layer feed-forward ANN using a differentiable activation function (Patterson, 1996).

The rule can be derived analytically by defining the following measure for the training error of a possible weight vector $\vec{w}$

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{4.3}$$

in which D is the set of training examples, t_d the target output for example d, o_d the output of the unit for example d. To determine the update for weight w_i we want to descend the error landscape, i.e. we want to alter each w_i of $\vec{w}$ in proportion to $\partial E / \partial w_i$, the derivative of E with respect to w_i. Instantiating Equation (4.3) and deriving this with respect to w_i we get

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 = \sum_{d \in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\vec{x}_d) = \sum_{d \in D}(t_d - o_d)(-x_{id}) \tag{4.4}$$

Multiplying Equation (4.4) by $-\eta$ we get the weight update rule for each of the w_i's

$$\Delta w_i = \eta \sum_{d \in D}(t_d - o_d)x_{id} \tag{4.5}$$

The value of η is a positive constant called the learning rate. This value is used to moderate the degree to which weights are changed each step and is usually set to some small value (e.g. 0.05). The negative sign is present because we want to move the weight vector in the direction that *decreases E*.

Based on this weight update rule a gradient descent training algorithm can be specified. This algorithm basically works as follows. Pick an initial random weight vector. Apply all examples and compute Δw_i for each weight according to Equation (4.5). Update each weight w_i by adding Δw_i and repeat this process. Given the quadratic nature of the defined error measure, the error landscape will contain only a single global minimum. Therefore this algorithm is guaranteed to converge to a weight vector with minimal error. Care must however

be taken not to choose a too large learning rate η. If η is too large, the search runs the risk of overstepping the minimum in the error landscape, rather than settling in to it.

Although the algorithm is guaranteed to converge, there are a number of practical difficulties in applying gradient descent based algorithms in general. First of all converging to a local minimum can sometimes be quite slow and can require many thousands of gradient descent steps. Secondly, if there are more local minima in the error landscape, there is no guarantee that the method will find the global minimum. Of course this second issue is of no concern here. Given the quadratic cost function of Equation (4.3) we can be sure there is just one minimum in the error landscape.

To overcome these issues a variation called *incremental gradient descent*, or *stochastic gradient descent* is commonly used. Whereas the gradient descent training rule presented in Equation (4.5) updates the weights after summing over *all* the training examples, the idea behind stochastic gradient descent is to approximate this gradient descent. This is done by updating weights incrementally, following the calculation of the error for *each* individual example. Weights are thus updated after each example instead of after all examples. The weights are updated according to the so-called *Widrow-Hoff Rule* or *Delta Rule* which is defined as

$$\Delta w_i = \eta(t - o)x_i \tag{4.6}$$

where t, o, and x_i are the target value, unit output, and i-th input for the training example in question. If a sigmoid activation function is used we can rewrite the Delta Rule in Equation (4.6) more specifically. Noting that the first derivative of the sigmoid logistic function σ can easily be expressed as $\partial\sigma(v)/\partial v = \sigma(v)(1-\sigma(v))$, we can now calculate $\partial E/\partial w_i$ by

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \sigma(\vec{w}\vec{x}_d)) \\ &= \sum_{d\in D}(t_d - o_d)(\sigma(\vec{w}\vec{x}_d)(1-\sigma(\vec{w}\vec{x}_d))\left(-\frac{\partial \vec{w}\vec{x}_d}{\partial w_i}\right) \\ &= \sum_{d\in D}(t_d - o_d)(o_d(1-o_d))(-x_i)\end{aligned} \tag{4.7}$$

Again multiplying by a learning rate $-\eta$ and adopting a stochastic approach we get the following Delta Rule for sigmoid output units using the logistic function as activation function

$$\Delta w_i = \eta(t - o)(o(1-o))x_i \tag{4.8}$$

The complete STOCHASTIC-GRADIENT-DESCENT algorithm based on this specific Delta Rule is presented in Table 4.1. The weights in the example of the previous section were calculated using this algorithm.

STOCHASTIC-GRADIENT-DESCENT(*TrainingExamples*, η, n_{in}, n_{out})

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$ where $\vec{x}$ is the vector of input values, and $\vec{t}$ is the target vector of output values. η is the learning rate (e.g. 0.05). n_{in} is the number of network inputs, and n_{out} the number of output units. The weight from input i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs and n_{out} output units
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *TrainingExamples*, Do
    - Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network
    - Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \eta(t_j - o_j)(o_j(1 - o_j))x_i$$

TABLE 4.1
Summary of the STOCHASTIC-GRADIENT-DESCENT algorithm for single layer feed-forward networks using sigmoid units.

4.1.3. Multilayer Feed-Forward Networks

The previous section focused on single layer ANNs based on units using nonlinear sigmoid activation functions. Although other activation functions could also be used (e.g. the threshold function used in perceptron units), the problem with these networks is, that they are not able to express nonlinear decision surfaces. The most commonly used example in the literature is the simple XOR-function. This function outputs one if and only if exactly one of its two inputs is one. The weights in a single layer ANNs can not be set or trained to reflect this nonlinearity. This inability makes these networks quite uncapable of expressing the highly nonlinear nature of many interesting real-life problem domains.

Multilayer feed-forward ANNs on the other hand are capable of expressing a wide variety of nonlinear decision surfaces. A typical multilayer feed-forward ANN consists of a number of input nodes, one or more hidden layers consisting of sigmoid units, an output layer consisting of sigmoid units, and the interconnections between them. The weights of these interconnections are trained using a gradient descent algorithm similar to that discussed in the previous section. This algorithm goes by the name of *BACKPROPAGATION* or *Generalized Delta Rule* and is discussed in the next section.

4.1.3.1. The BACKPROPAGATION Algorithm

The BACKPROPAGATION algorithm learns the weights for a multilayer feed-forward network, given a network with a fixed number of units and interconnections. Just as the STOCHASTIC-GRADIENT-DESCENT algorithm in section 4.1.2.1 it employs gradient descent to attempt to minimize the squared error between the network output and the target values for these outputs. The name BACKPROPAGATION arises from the method in which the correction to the weights are made. In contrast to the single layer network case, multilayer networks can have multiple local minima. Gradient descent is therefore guaranteed only to converge toward some local minimum, and not the global minimum error. Despite this, BACKPROPAGATION yields excellent results in many real-world applications including for example the recognition of speech and visual information to name but a few.

The general layout of the algorithm is the same as that of the STOCHASTIC-GRADIENT-DESCENT algorithm presented earlier. Table 4.2 presents the algorithm for the specialized case of a two-layer network with one hidden layer and one output layer (Mitchell, 1997).

BACKPROPAGATION(*TrainingExamples*, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$ where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target output values. η is the learning rate (e.g. 0.05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units. The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units
- Initialize all network weights to small random numbers (e.g. between –0.05 and +0.05)
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *TrainingExamples*, Do

Propagate the input forward through the network

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network

Propagate the errors backward through the network

2. For each network output unit k, calculate its error term δ_k, Do

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h, calculate its error term δ_h, Do

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

TABLE 4.2
Summary of the stochastic gradient descent version of the BACKPROPAGATION algorithm for feed-forward networks containing two layers of sigmoid units.

The key points to note in the algorithm are the calculations of the error terms δ_k and δ_h. These terms can again be derived analytically. Recall that stochastic gradient descent iterating through the training examples one by one, for each training example d descending the error gradient E_d with respect to this single example. Each training example d causes the weight w_{ji} to be updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{4.9}$$

where E_d is the total error on training example d, summed over all output units

$$E_d \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2 \tag{4.10}$$

The error terms δ_k and δ_h are obtained by deriving an expression for $\partial E_d / \partial w_{ji}$. Noting that w_{ji} can only influence the network through net_j, the weighted sum of inputs for unit j, this expression can be rewritten as

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji} \end{aligned} \tag{4.11}$$

where x_{ji} is the i-th input to unit j. Given Equation (4.11) the goal is to find convenient expressions for the term $\partial E_d / \partial net_j$, one where unit j is an output unit and one where unit j is a hidden unit.

For output units net_j only influences the network only through o_j, the output computed by unit j. Using the chain rule and instantiating Equation (4.10) we get

$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \\
&= \left(\frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2\right)\left(\frac{\partial \sigma(net_j)}{\partial net_j}\right) \\
&= \left(\sum_{k\in outputs}(t_k - o_k)\frac{\partial}{\partial o_j}(t_k - o_k)\right)\left(o_j(1-o_j)\right) \\
&= -(t_j - o_j)o_j(1-o_j)
\end{aligned}
\tag{4.12}
$$

where σ again represents the sigmoid function. Using the notation δ_k to denote the quantity $-\partial E_d / \partial net_k$ we get the error term for an output unit as used in the algorithm

$$
\delta_k = o_k(1-o_k)(t_k - o_k) \tag{4.13}
$$

For units in a hidden layer, the weight update rule must take into account the indirect ways a weight w_{ji} can influence the network outputs and hence E_d. Therefore the notation *Downstream*(*j*) is used to denote the set of units which inputs are directly connected to the outputs of unit *j*. By noting that the output of a hidden unit *j*, i.e. net_j, can only influence the network outputs through the units in *Downstream*(*j*) we can write

$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k\in Downstream(j)}\frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k\left(\frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j}\right) \\
&= \sum_{k\in Downstream(j)} -\delta_k w_{kj}\frac{\partial \sigma(net_j)}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k w_{kj} o_j(1-o_j)
\end{aligned}
\tag{4.14}
$$

Again using the notation δ_h to denote the quantity $-\partial E_d / \partial net_h$ we get the error term for a hidden unit in the algorithm

$$
\delta_h = o_h(1-o_h)\sum_{k\in Downstream(h)} w_{kh}\delta_k \tag{4.15}
$$

Combining the two results in Equations (4.9) and (4.11) we find the exact weight updates Δw_{ji} for both hidden and output units as used by the BACKPROPAGATION algorithm.

4.1.4. Neural Networks applied to the Advisory System

The previous sections introduced the concepts of single and multilayer feed-forward ANNs and how they are trained. This section explores how these types of networks could be applied to the advisory system.

4.1.4.1. Single Layer Feed-Forward Network

Applying a single layer linear neural network to learn the activation values of the advisory system comes actually quite naturally. The advisory system is after all a linear network itself as can be seen in Figure 4.4.

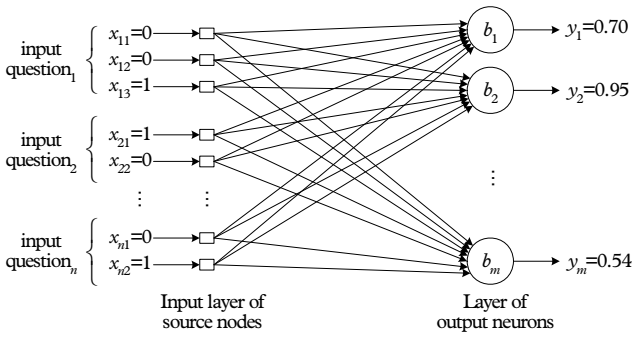


FIGURE 4.4
The advisory system as a single layer linear network. The input layer consists of nodes for each and every answer to each and every question. The output layer consists of nodes for each and every bicycle. The weight values from the source nodes into the output neurons are the activation values used in the advisory system. In this particular example the user responded to question 1 with its third answer, to question 2 with its first answer and so on. The output values of the output neurons represent the total activation value after a series of question. Sorting the bikes accordingly will yield the advice provided by the system.

The input layer consists of nodes for each and every answer to each and every question. Answering a question with a certain answer will set the input value of the corresponding input node to 1 and 0 for all its sibling nodes, i.e. the nodes for the other answers to the question. The output layer consists of nodes for each and every bicycle. Each input node is connected to each output node by a weight factor. This weight factor is the very same activation value as used in the advisory system linking the answers to the bicycles. By setting up the linear network this way, the output value for a certain output node is actually the total score of a bicycle after a series of questions as defined by Equation (2.5). Suppose for example that Figure 4.4 represents the situation after a series of questions. The first question was answered with its third possible answer, the second with its first, and the *n*-th with its second, setting the corresponding input nodes to 1 and all other nodes to 0. Crunching the numbers will lead the output layer to reproduce the ranking of the bicycles relative to the answered questions. Sorting the bicycles on the values at the output layer will reflect this ranking.

Learning in this setting simply breaks down to some sort of Delta Rule learning, modifying the original activation values directly to reflect the witnessed user preferences over the data. There are however a number of issues that need to be addressed should a single layer ANN be implemented in the advisory system.

The first is determining the target output values for the output units. The output units should now produce values that when sorted accordingly reproduce the lists as observed in the clickthrough data. In a classification task where each output unit represents just one target class determining the output value is much easier (i.e. a simple 0 or 1). In ranking mode the target output value must be considered relative with respect to the values of the other output units. Remember that we are not interested in the actual value of the output node, just in its relative position with respect to the other units.

This relates to the second issue of cutting off irrelevant bicycles. Following Equation (2.12) in section 2.2.2.3, only relevant bicycles are displayed in the advisory system. Training an ANN to reflect the clickthrough data must take into account the number of relevant bicycles shown to the users after a selected series of questions. In other words, the bicycles found relevant after a series of questions after learning, are the same bicycles found relevant before learning. It is therefore not enough to just learn the observed ranking. The weights learned must also reflect the relevance of the bicycles.

Third, the advisory system uses activation values in the range [–1, +1], whereas this condition does not necessarily hold for the single layer ANN. Therefore the learned weight can not be directly used as activation values in the advisory system. If they are to be used as such they need to be converted to the range [–1, +1]. In doing so, care must be taken to keep the number of relevant bicycles constant.

In addition to these issues, the question is whether a single layer ANN network offers sufficient complexity to learn the observed clickthrough preferences. Even using a STOCHASTIC-GRADIENT-DESCENT approach with output units using a sigmoid activation function. After all, adjusting the weights for a certain answer in a series of questions, may prove counterproductive for the weights of the same answer in other series of questions. Activation values in the advisory system are entered for answers to questions separately, not for the

combination of them. Whether or not a single layer ANN proves sufficient thus remains to be seen.

4.1.4.2. Multilayer Feed-Forward Network

From the discussion above it is clear that the proposed advisory system actually is a linear network. By adding a hidden layer between the input and output layer, the network can easily be extended to a multilayer topology as can be seen in Figure 4.5.

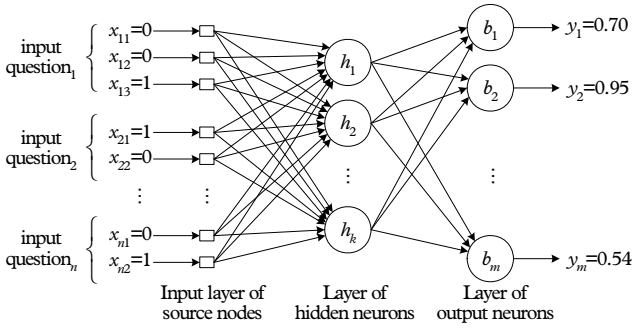


FIGURE 4.5
The advisory system as a feed-forward multilayer network. The source nodes now connect into a layer of k hidden neurons, which in turn connect to the m output neurons.

Although with this a multilayer network setting is certainly conceivable, it is quite clear that the weight values learned, bear no direct relation to the activation values used in the advisory system. After all, output values are now determined by the combined weights of two or more layers. Obviously, there is no way to reduce this complexity back to the single layer weight values as used in the advisory system.

Application of this type of neural network would then mean a two-stage design concept. The first stage uses the basic weight values provided by a domain expert and sorts the bicycles accordingly. This provides the advisory system with some basic intelligence, so it will behave rationally when launched for the first time. Using the manually provided weight values, the system is then allowed to collect implicit user feedback over a period of time. After collecting sufficient data, a multilayer network can be trained to reflect the initial ranking based on the original weights and the witnessed user preferences. Having done so, the advisory system can then enter the second stage and completely switch to the trained neural network for its intelligence.

4.2. BAYESIAN LEARNING

As already mentioned in the introduction of this chapter, Bayesian reasoning provides a probabilistic approach to learning. It is based on the assumption that underlying probability distributions govern the dynamics of a problem, and that decisions can be made just by reasoning about these probabilities in conjunction with the observed data. Bayesian learning methods that calculate explicit probabilities for hypotheses, such as the naïve Bayes classifier, are among the most practical approaches to certain types of learning problems.

Some notation must be introduced to be able to formally define Bayesian learning. In machine learning we are mostly interested in determining the best hypothesis h from some hypothesis space H, given the observed training data D. We write $P(h)$ to denote the initial probability that hypothesis h holds, regardless of the observed training data D. $P(h)$ can reflect any prior knowledge available about the probabilities of the various hypotheses. If no such knowledge is available, all these probabilities are the same. $P(h)$ is often called the *prior probability* that h holds. Along the same lines we write $P(D)$ to denote the prior probability that training data D is observed. Next we write $P(D|h)$ to denote the probability of observing data D given that hypothesis h holds. Generally $P(x|y)$ denotes the probability that x holds given y. In machine learning we are interested in the probability $P(h|D)$, i.e. the probability that hypothesis h holds given the observation of training data D. $P(h|D)$ is called the *posterior probability* of h. Using this notation we can now define Bayes theorem.

4.2.1. Bayes Theorem

Bayes theorem is the cornerstone of all Bayesian learning methods. Bayes theorem provides a means to determine the posterior probability $P(h|D)$, from the prior probability $P(h)$, together with $P(D)$ and $P(D|h)$

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)} \tag{4.16}$$

As can be intuitively expected, Bayes theorem states that $P(h|D)$ increases with $P(h)$ and $P(D|h)$. Equally, $P(h|D)$ decreases if $P(D)$ increases, because the more probable it is that D is observed independent of h, the less evidence D provides in support of h.

In many learning scenarios the main point of interest is finding the most probable hypothesis h from a set of hypotheses H given observed data D. Such a hypothesis is called a *maximum a posteriori* (MAP) hypothesis. Using Bayes theorem the MAP hypothesis can be determined by calculating the posterior probability of each of the candidate hypotheses

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h \mid D) = \underset{h \in H}{\operatorname{argmax}} \frac{P(D \mid h)P(h)}{P(D)} = \underset{h \in H}{\operatorname{argmax}} P(D \mid h)P(h) \tag{4.17}$$

By assuming that the prior probability $P(h)$ of each of the hypotheses is equal (i.e. $P(h_i)=P(h_j)$ for every i and j), h_{MAP} can be further simplified by only considering the term $P(D|h)$ exclusively. Since $P(D|h)$ is often called the *likelihood* of the data D given h, the hypothesis that maximizes $P(D|h)$ is called the *maximum likelihood* (ML) hypothesis

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D \mid h) \tag{4.18}$$

4.2.2. Naïve Bayes Classifiers

One of the most widely used Bayesian learning methods is the naïve Bayes classifier. The naïve Bayes classifier can be applied to learning tasks where instances x are described by conjunctions of attribute values a_i and where the target function $f(x)$ can take on any value from some finite set V. In these tasks a set of training examples is provided, and the goal is to predict the target value, or classification, for a new instance described by the tuple of attributes $<a_1, a_2, \ldots, a_n>$. The Bayesian approach in classifying is to assign the most probable target value v_{MAP} given the attribute values $<a_1, a_2, \ldots, a_n>$ that describe the new instance. Using Bayes theorem we can formulate v_{MAP} as follows

$$\begin{aligned} v_{MAP} &= \underset{v_j \in V}{\operatorname{argmax}} P(v_j \mid a_1, a_2, \ldots, a_n) \\ &= \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2, \ldots, a_n \mid v_j)P(v_j)}{P(a_1, a_2, \ldots, a_n)} \\ &= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2, \ldots, a_n \mid v_j)P(v_j) \end{aligned} \tag{4.19}$$

Calculating $P(a_1, a_2, \ldots, a_n|v_j)$ can however practically prove problematic. In order to obtain reliable estimates every instance in the instance space needs to be observed many, many times, requiring a very, very large training set. The naïve Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. The assumption is that given the target value v_j, the probability of observing the conjunction $a_1, a_2, \ldots, a_n$ is just the product of the probabilities for the individual attributes, i.e. $P(a_1, a_2, \ldots, a_n)=\prod_i P(a_i|v_j)$. Substituting this in Equation (4.19) the naïve Bayes classifier is formulated as follows

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_{i=1}^{n} P(a_i \mid v_j) \tag{4.20}$$

where v_{NB} denotes the target value output by the naïve Bayes classifier. In some contexts the performance of the naïve Bayes classifier has been shown to be comparable to that of neural networks and decision tree learning.

For example consider again the training data from Table 2.1, providing the set of 12 examples for the target concept *BicycleRacingWeather*. Sections 2.1.1 and 4.1.2 showed how a decision tree and a single layer ANN could be used to classify the new instance from Table 2.2. We are now going to use a naïve Bayes classifier to classify this instance. Instantiating Equation (4.20) using the attribute values for *Sky*, *Temperature*, *Road* and *Warm*, and omitting these attribute names for brevity, we get

$$\begin{aligned} v_{NB} &= \underset{v_j \in \{Yes,No\}}{\operatorname{argmax}} P(v_j) \prod_{i=1}^{n} P(a_i \mid v_j) \\ &= \underset{v_j \in \{Yes,No\}}{\operatorname{argmax}} P(v_j) P(Sunny \mid v_j) P(Warm \mid v_j) P(Dry \mid v_j) P(Strong \mid v_j) \end{aligned}$$

In order to calculate v_{NB} we need to estimate 10 probabilities. The probabilities of the target values *Yes* and *No* are easy to estimate based on their frequencies in the 12 training examples

$$P(BicycleRacingWeather = Yes) = 6/12 = 0.5$$
$$P(BicycleRacingWeather = No) = 6/12 = 0.5$$

The remaining conditional probabilities are estimated along the same lines

$$P(Sunny \mid Yes) = 3/6 = 0.5 \quad P(Warm \mid Yes) \approx 0.33 \quad P(Dry \mid Yes) = 0.5 \quad P(Calm \mid Yes) \approx 0.17$$
$$P(Sunny \mid No) = 1/6 \approx 0.17 \quad P(Warm \mid No) = 0 \quad P(Dry \mid No) \approx 0.17 \quad P(Calm \mid No) \approx 0.17$$

Doing the math we can now calculate the value of v_{NB} using

$$P(Yes)P(Sunny \mid Yes)P(Warm \mid Yes)P(Dry \mid Yes)P(Calm \mid Yes) \approx 0.0070$$
$$P(No)P(Sunny \mid No)P(Warm \mid No)P(Dry \mid No)P(Calm \mid No) = 0$$

The naïve Bayes classifier will therefore assign the target value *BicycleRacingWeather*=*Yes* to the new instance, as did the decision tree and neural network earlier.

An interesting thing to note is that there is actually no explicit search through the space of possible hypothesis. In a decision tree setting, the decision tree has to be constructed first. In a neural network setting the weights first have to be learned. In using a naïve Bayes classifier, we only need to count the frequencies of the various data combinations within the training examples.

4.2.3. Bayesian Learning applied to the Advisory System

The example in the previous section showed how a naïve Bayes classifier could be used to classify instances. In doing so, the probabilities for each of the hypotheses were calculated and the one with the highest probability was ultimately selected. By considering all the probabilities calculated and sorting them on these probabilities this Bayesian approach could also be used for ranking problems.

If we were to apply a Bayesian learning method to learn the activation values in the advisory system, the first thing that needed to be done is determining what probability we want the bicycles to be sorted on. Recall from chapter 3 that user feedback in the system is gathered by clickthrough data. This chapter discussed the use of absolute click results for extracting user-preferred rankings, in which bicycles that were clicked most were considered most relevant. Given this implementation we are therefore interested in the probability that a certain

bicycle is clicked after a certain series of answers to a certain series of questions. Using Bayes theorem this probability can be formulated as follows

$$P(click_b \mid a_1,\ldots,a_n) = \frac{P(a_1,\ldots,a_n \mid click_b)P(click_b)}{P(a_1,\ldots,a_n)} \tag{4.21}$$

where $click_b$ denotes the event in which bicycle b was clicked and a_1, …, a_n the event in which the first question was answered with answer a_1, the second with a_2, and so on. Note that in contrast to a classification task with attributes (e.g. the one in the previous section), the number of questions put to a user can vary. The series a_1, …, a_n therefore actually defines a context in which n questions were put to the user, and answered accordingly. Given an advisory system consisting of a total of n possible questions, each having m possible disjunctive answers, the total number of different contexts c is defined by

$$c = \sum_{i=1}^{n} m^i \binom{n}{i} = \sum_{i=1}^{n} m^i \frac{n!}{i!(n-i)!} \tag{4.22}$$

where $\binom{n}{k}$ denotes the binomial coefficient. Given a classification problem with n attributes, each having one of m possible values the number of different contexts would be m^n, i.e. the last term in the summation in Equation (4.22). Obviously, with a rising number of questions, the number of contexts to be considered in the advisory system will rapidly outgrow the number of contexts in a typical classification task. To worsen matters the advisory system also allows questions with conjunctive answers. The possibilities for such a question having m answers are not m, but $2^m - 1$ (note that this value is not 2^m, because of the possible *Don't care* option a user has in the advisory system. If a user chooses this option, the question will not be considered to be a part of the context (see also section 2.2.3)).

To be able to rank the bicycles according to their click probabilities, we need to calculate Equation (4.21) for each and every bicycle. In doing so, the ultimate ranking position is independent of the term $P(a_1, \ldots, a_n)$, which therefore can be omitted. Combining this with a naïve Bayes approach to counter the large number of contexts to otherwise gather data for, Equation (4.21) can be rewritten as

$$P_{NB}(click_b \mid a_1,\ldots,a_n) = P(click_b)\prod_{i=1}^{n} P(a_i \mid click_b) \tag{4.23}$$

Using Equation (4.23) the advisory system could rank bicycles for contexts not observed during the training phase.

There is however a problem when using a naïve Bayes approach in ranking. Although naïve Bayes is one of the most effective and efficient classification algorithms, research shows that it produces poor class probability estimates, often unreliable for ranking. In the advisory system the naïve Bayes' assumption that attributes are independent given the class, could prove to be particularly problematic. Although questions are entered *in*dependently, the system behaviour will obviously put them to users *inter*dependently. For example, a user highly interested in racing bicycles will not be presented with a question like *Do you want to do groceries with your bicycle?*. To tackle such problems numerous techniques have been proposed to extend naïve Bayes for better classification and ranking results. One of these is called *Hidden Naïve Bayes* (HNB) and might prove useful in this context. In HNB, a hidden parent is created for each attribute to represent the influences from all other attributes. In doing so the interdependent character of certain attributes can be expressed. HNB inherits the structural simplicity of naïve Bayes and can be learned almost just as easily. Research shows that HNB outperforms other forms of Bayesian learning on ranking problems (Zhang, 2005).

Another more fundamental problem in applying a Bayesian learning method in the advisory system, is the actual modification of activation values based on such an approach. The probabilities calculated using Equation (4.23) can obviously not be used as activation values for the corresponding contexts. Although the probability $P(a_i \mid click_b)$ could perhaps be used as an activation value relating answer a_i to bicycle b, it is not altogether clear whether such an

approach would be valid. It would require a radical modification to the heart of the advisory system at the very least.

4.3. CONCLUSION

This chapter discussed the use of two potentially useful learning paradigms: artificial neural networks and Bayesian learning. As section 4.1.4 showed, the advisory system highly resembles an ANN, thereby validating the further exploration of such an approach. Bayesian learning on the other hand was thought to be useful given the nature of the problem domain at hand. After all, underlying probabilities are governed by witnessed click frequencies which can be gathered relatively easily. Of course many, many more methods could have been explored. One potentially sticking out is the use of Support Vector Machines (SVM). SVMs have been given a lot of attention lately and have been successfully applied in for example ranking web pages (Joachims, 2002).

Adopting an Occam's Razor[1] approach these methods were however not taken into serious consideration. Firstly because some of them are highly complex and therefore relatively hard to understand and implement (even in spite of standard libraries as is the case for SVMs for example). Secondly, and much more importantly, they would not relate to the central research question of this thesis, namely the adjustment of the underlying activation values. It is not enough to just learn the observed clickthrough preferences. Addressing the research question we want to be able to adjust the activation values to reflect the observed clickthrough data.

Returning to the discussed learning paradigms it is altogether clear that the use of a single layer feed-forward ANN is definitely the most promising implementation candidate. By highly resembling an ANN itself, it can be applied to the advisory system highly elegantly. If we were able to train a single layer feed-forward ANN to reflect the witnessed clickthrough data, the weights learned could be directly related to the activation values used in the advisory system. This would not be possible with a multilayer ANN, Bayesian learning or other methods.

4.4. SUMMARY

This chapter explored different learning approaches for their possible use in the advisory system. Of course many more learning methods could have been applied, but obviously some are more useful than others in the current context. Given this context the use of ANNs and Bayesian learning seemed particularly useful and were explored further. Given the nature of the current problem domain and the central research question in this thesis, a single layer feed-forward ANN was selected as implementation candidate. As section 4.1.4.1 showed, there are however a number of issues that need to be resolved when applying this method to the system. These issues are the topic of the next chapter.

[1] Occam's razor is a principle attributed to the 14th-century English logician and Franciscan friar, William of Ockham. The principle is often expressed in Latin as: "entia non sunt multiplicanda praeter necessitatem", roughly translated as "entities must not be multiplied beyond necessity". It is often paraphrased as "All other things being equal, the simplest solution is the best." In other words, when multiple competing theories are equal in other respects, the principle recommends selecting the theory that introduces the fewest assumptions and postulates the fewest entities. It is in this sense that Occam's razor is usually understood (http://www.wikipedia.org).

5. CONCEPTUAL DESIGN & IMPLEMENTATION

Ferocious designs
Connected and ready to play
Marillion, from *Cover My Eyes* (Holidays In Eden, 1991)

In selecting a single layer feed-forward ANN to learn the activation values in the advisory system, a number of issues still need serious addressing. Although ANNs have been applied successfully to ranking problems, the nature of the current problem domain denies direct application of such approaches. It is therefore not altogether obvious how to train a single layer feed-forward ANN in this context. Remembering the discussion in section 4.1.4.1 it is even uncertain if it is even possible to do so.

To be able to test its usefulness in the current context, the first part of this chapter focuses on the conceptual design of a single layer feed-forward ranking ANN. This part addresses the issues that need to be resolved for an actual implementation of such an ANN. The chapter concludes with a brief overview of the implementation of this design.

5.1. CONCEPTUAL DESIGN

ANNs have been successfully applied to ranking problems in the past. For example, Caruana *et al.* introduced RANKPROP to improve on standard BACKPROPAGATION in ranking patients by risk. Their so-called *Medis Pneumonia Database* consisted of more than 14.000 pneumonia cases, each consisting of 65 basic measurements and lab results. For each case it was known whether the patient lived or died. The goal was to rank the pneumonia cases according to their mortality probability. In doing so they used a network setup with just one single output node, and ranked the cases according to the output value of that node, the mortality probability (Caruana *et al.*, 1996). Another more recent example is the RANKNET algorithm of Burges *et al.* which uses a differently defined cost function, called *cross entropy*, instead of the sum of squared errors to improve on ANN ranking. In their approach they also use a single output node on which output value the original instances are sorted (Burges *et al.*, 2005). Basically all these approaches share a basic network setup in which the data offered to the inputs of the network is sorted on the output of a single output node. Another common property is that the number of instances for which a ranking is needed, can be endless. Consider for example the ranking of web pages (Page *et al.*, 1998, 1999).

The setup of the current problem domain is however totally different. Remember that the inputs here are the answers provided to a series of questions. Although there can be many such series, the number is certainly not endless. Furthermore, it is obvious that we do not want to sort this set of inputs, but another set, the bicycles. Combining this with the ultimate need to be able to adjust the activation values, we are left with a totally different network setup. As can be seen in Figures 4.4 and 4.5 from section 4.1.4 the output layer therefore does not consist of a single output node, but of nodes for each and every bicycle. The output of the network is the whole set of output nodes, sorted on the observed output values. Bicycle nodes with high output values are ranked high, ones with low values lower and lower. Note that in doing so the actual observed output values are of no concern at all, it is the order in which they appear that is relevant. Worse still, we do not even have target values to learn for each and every output node. Applying a Delta Rule based algorithm is therefore not directly possible.

Fortunately the feedback gathered allows us to define a set of strict preference pairs reflecting the user preferred ranking for a given lines of questioning. Using these pairs it is possible to formulate a measure for the ranking error made at each output unit, and hence a cost function for the total rank error over all units. Although this cost function has no first order derivative, the ranking error measure it uses, nevertheless enables us to define a weight update rule for the ranking ANN.

The rest of this section first discusses the use and gathering of preference pairs. Using these pairs the ranking error for an output unit is subsequently formalized, as is the cost function based on these ranking errors. After settling for a useful unit activation function for the output units given the context of the advisory system, the weight update rule for the ranking ANN is introduced.

5.1.1. Preference Pairs

Using the feedback gathered by the FAIRSHARES algorithm it is possible to define a set of strict preference pairs for a given line of questioning. As discussed in section 3.3.2 we have two options for interpreting the clickthrough data. The first being relative, and the second absolute. In relative interpretation mode clickthrough data is interpreted as preference statements about pairs of bicycles. This kind of interpretation adjusts rankings on a somewhat more local level, leaving the global ranking largely intact. In absolute interpretation mode the data is interpreted in an absolute sense. The bicycle clicked most for a given line of questioning will be ranked highest globally. Therefore absolute ranking can be expected to be much more invasive than relative ranking. For both ranking modes we need a mechanism to translate the feedback for a given line of questioning to a series of preference pairs.

In relative ranking mode we firstly need a set of base preferences reflecting the original judgment of the advisory system for the line of questioning. This is necessary because we want to impose the original expert-provided ranking if our feedback concerning a pair of bicycles is not conclusive. In doing so we ensure that the global ranking stays intact as much as possible. This set of base preferences is then updated to reflect the relative preferences as witnessed in the feedback. Each relative preference pair thus causes a minor change in the set with all preferences.

In absolute ranking mode on the other hand, the original expert-provided ranking is of no concern. Bicycles are rearranged according to their click counts, regardless of the previously existing arrangement. However, in doing so care must be taken to enforce a preference of the relevant bicycles for a given context over the bicycles considered irrelevant. Remember that bicycles considered irrelevant by the system, are not shown to the user. We surely do not want a situation where bicycles previously considered irrelevant end up between the relevant ones.

5.1.1.1. Obtaining Preference Pairs

Both ranking modes need to convert a list of real or integer values to a set of preferences. For relative ranking we need to convert a list of real activation values. For absolute ranking a list of integer click counts. In doing so we must bear in mind that two or more bicycles can actually have the same total activation value after a series of questions or the same number of clicks. It is obvious that we can not make valid claims about the arrangement of such bicycles.

Minding this issue, converting a list of values to preference pairs is actually quite straightforward and somewhat similar to the FAIRSHARES algorithm presented in section 3.3.2. As was the case for this algorithm, we start by creating a sorted union of subsets, each containing objects sharing the same value. The objects with the highest values can thus be found in the first subset, and the ones with the lowest in the last. The procedure then walks through these subsets, preferencing each element in the current subset over all the elements of the next subset. Table 5.1 summarizes the procedure.

OBTAIN-PREFERENCES(*Values*)

Values *is a top-down sorted set of n values* ($f(o_1)$, $f(o_2)$, …, $f(o_n)$) *for a set of objects* o_1, …, o_n *for a given line of questioning. Objects in* Values *may share the same value, i.e.* $f(o_i)=f(o_j)$ *for some objects* o_i *and* o_j*, where* $i \neq j$.

- Let $S=(S_1, S_2, \ldots, S_m)$ be the sorted union of subsets S_v of *Values* with $\{o_i \in S_v \mid f(o_i)=v\}$ for $1 \leq i \leq n$
- *PreferencePairs* $\leftarrow \varnothing$
- For $i \in \{1, \ldots, m-1\}$, Do
  - For each o_j in S_i, Do
    - For each o_k in S_{i+1}, Do
      - *PreferencePairs* $\leftarrow$ *PreferencePairs* $\cup \{ o_j \rhd o_k \}$
- Return *PreferencePairs*

TABLE 5.1
Summary of the OBTAIN-PREFERENCES procedure for converting a list of real or integer values for a given set of objects to a set of preference pairs.

5.1.1.2. Obtaining Training Data for Relative Ranking

As discussed above the training set for relative ranking is based on the set of base preference pairs from the advisory system. Relative interpretation of the clickthrough data will yield another set of preference pairs, which may or may not conflict with the set of base preferences.

In accordance with the FAIRSHARES algorithm, this set not only expresses preferences about bicycles originally having distinct activation values, but also about bicycles originally sharing the same value.

To obtain the necessary training preferences, the basic idea is to correct the set of base preferences, making it consistent with the relative preferences witnessed in the clickthrough data. First of all, the set with base preferences for a certain line of questioning is obtained by using the OBTAIN-PREFERENCES-procedure from the previous paragraph, using the actual activation values of the advisory system. Again, this set may now conflict with the set of relative preferences we obtained from the clickthrough data. We therefore check each preference in the set of relative preferences. First we check whether there is a conflict. If we have a relative preference $o_i \rhd o_j$ and the set of base preferences contains a preference $o_j \rhd o_i$, this preference obviously conflicts and has to be removed. Subsequently, we check whether the set of preferences even contains $o_i \rhd o_j$ at all. If not, it is added to the list. The complete procedure for obtaining training data for relative ranking is summarized in Table 5.2.

OBTAIN-RELATIVE-TRAINING-DATA(*ActivationValues*, *RelativePreferences*)

ActivationValues *is a top-down sorted set of n activation values* ($f(o_1)$, $f(o_2)$, …, $f(o_n)$) *for a set of objects* o_1, …, o_n *for a given line of questioning.* RelativePreferences *is the set of preference pairs expressing the relative preferences as witnessed in the clickthrough data.*

- *TrainingData* ← OBTAIN-PREFERENCES(*ActivationValues*)
- For each preference $o_i \rhd o_j$ in *RelativePreferences*, Do
  - If $\{ o_j \rhd o_i \} \in$ *TrainingData*
    - *TrainingData* ← *TrainingData* – $\{ o_j \rhd o_i \}$
  - If $\{ o_i \rhd o_j \} \notin$ *TrainingData*
    - *TrainingData* ← *TrainingData* $\cup \{ o_j \rhd o_i \}$
- Return *TrainingData*

TABLE 5.2
Summary of the OBTAIN-RELATIVE-TRAINING-DATA procedure for retrieving relative training data needed for a given line of questioning.

5.1.1.3. Obtaining Training Data for Absolute Ranking

Training data for absolute ranking is actually based on the combination of two sets. The first set reflects the absolute preferences gathered from the clickthrough data for a given line of questioning. The second set the bicycles deemed irrelevant for that context. As already mentioned, we want to ensure that every bicycle used in an absolute preference pair is to be strictly sorted over bicycles deemed irrelevant. Even if that bicycle was not clicked at all, since it is still to be preferred over bicycles deemed irrelevant by the system. Remember that these bicycles were not even shown by the system.

In gathering the training data, the first step is therefore to determine the irrelevant bicycles. These can be gathered using the original activation values of the system for the context and the absolute click count values for the bicycles. The list of irrelevant bicycles consists of the subset of bicycles for which an activation value is available, but no absolute click count (even if that click count happens to be zero). The next thing we need is a set with the bicycles clicked fewest in the absolute values. We want these bicyles to take preference over the irrelevant bicycles. With these two helper sets obtaining the training data is now straightforward. We first call the OBTAIN-PREFERENCES-procedure using the set of absolute click counts to create a set of absolute preferences. We only need to extend this set with preferences for the least clicked bicycles over the irrelevant ones. Table 5.3 summarizes the complete procedure.

OBTAIN-ABSOLUTE-TRAINING-DATA(*ActivationValues*, *AbsoluteValues*)

ActivationValues *is a top-down sorted set of n activation values* ($f(o_1)$, $f(o_2)$, …, $f(o_n)$) *for a set of objects* o_1, …, o_n *for a given line of questioning.* AbsoluteValues *is a top-down sorted set of m click counts* ($c(o_1)$, $c(o_2)$, …, $c(o_m)$) *for a set of objects* o_1, …, o_m *deemed relevant by the advisory system for the same line of questioning as witnessed in the clickthrough data.*

- Let *IrrelevantObjects* be the set of irrelevant objects not contained in *AbsoluteValues*, i.e. $\{o_i \in IrrelevantObjects \mid f(o_i) \in ActivationValues \wedge c(o_i) \notin AbsoluteValues\}$ for $1 \le i \le n$
- Let *LowestObjects* be the set of objects with the lowest click count in *AbsoluteValues*
- *TrainingData* ← OBTAIN-PREFERENCES(*AbsoluteValues*)
- For each object o_i in *LowestObjects*, Do
  - For each object o_j in *IrrelevantObjects*, Do
    - *TrainingData* ← *TrainingData* ∪ $\{ o_i \rhd o_j \}$
- Return *TrainingData*

TABLE 5.3
Summary of the OBTAIN-ABSOLUTE-TRAINING-DATA procedure for retrieving absolute training data needed for a given line of questioning.

5.1.2. Ranking Cost Function

The gathered preference pairs enable the definition of a ranking error term for each and every output unit and hence a cost function. I therefore propose a mechanism of rewards and punishments, instantiated when a preference pair is not consistent with the actual observed output values of the units in the pair. If this is the case for a preference pair $o_i \rhd o_j$, then the output unit o_i is rewarded one point and o_j punished one point. By parsing all preference pairs and summing all the rewards and punishments, we end up with a ranking error measure for the output units. Obviously, if the output of the ranking ANN is fully consistent with the preference pairs, the ranking error terms for each and every unit will be zero. If not, then the output value has to be changed in the direction given by its corresponding ranking error term. The higher this ranking term (or lower if it is negative), the higher the change should be. Table 5.4 shows an example of the calculation of the ranking error terms for a set of six preference pairs.

	o_1 =0.70	o_2 =0.85	o_3 =0.80	o_4 =0.70	o_5 =0.75	o_6 =0.90
$o_1 \rhd o_2$	+1	–1	0	0	0	0
$o_1 \rhd o_3$	+1	0	–1	0	0	0
$o_2 \rhd o_4$	0	0	0	0	0	0
$o_3 \rhd o_4$	0	0	0	0	0	0
$o_4 \rhd o_5$	0	0	0	+1	–1	0
$o_5 \rhd o_6$	0	0	0	0	+1	–1
δ	+2	–1	–1	+1	0	–1

TABLE 5.4
Calculating the error ranking term δ for each and every output unit for a given set of six preference pairs

The top row in this example shows the actual observed output values at the output units. The bottom row shows the ranking error terms for each output node, obtained by parsing all preference pairs and summing the values. Note that the ranking error term for the first unit is +2, indicating that a relatively large change is needed to agree with all preference pairs. Also note the term for the fifth unit for which the rewards and punishments sum up to zero. Evidently this unit must be left in peace, and it is better to tweak the values of other units.

More formally we can define this mechanism by

$$\delta_k = \sum_{\langle k,j \rangle \in P} \mathbf{1}[o_k < o_j] - \sum_{\langle j,k \rangle \in P} \mathbf{1}[o_k > o_j] \tag{5.1}$$

where δ_k denotes the ranking error term for output unit k and $\mathbf{1}$ the indicator function. As the example in Table 5.4 shows, δ_k can be calculated with just one run over the preference pairs.

Using this ranking error we can now formulate a ranking cost function, indicating the total ranking error E over all output units by

$$E \equiv \frac{1}{2} \sum_{k \in outputs} (\delta_k)^2 \tag{5.2}$$

The biggest disadvantage of this cost function is that it is not derivable. Descending the error landscape by taking the derivative of E with respect to w_i is therefore not possible and hence the derivation of a formal weight update rule. Although a formal derivation is not possible, the ranking error terms can still be expected to be highly useful in descending the error landscape, as will be shown later.

5.1.3. The Bipolar Sigmoid Activation Function

As discussed in section 2.2.1 activation values in the advisory system always take on real values in the range [–1, +1]. Consequently the total average activation of a bicycle after a series of values also lies in the range [–1, +1]. To reflect this property, I choose to use a bipolar sigmoid activation function instead of the sigmoid logistic function used in chapter 4. This function looks very similar to the logistic function of Equation (4.2) and is defined by

$$\sigma_2(v) = \frac{2}{1+e^{-\alpha v}} - 1 \tag{5.3}$$

where α again denotes the slope of the function. As was the case with the logistic function, the bipolar logistic function also has a first derivative which can be easily calculated using $\partial\sigma_2(v)/\partial v = (1-\sigma_2^2(v))/2$.

The α parameter can be expected to influence learning. Remember that if this value is set too high, the function will behave more and more like a threshold function. This will make it much harder to learn the rankings, because all the output values are either really close to plus or minus one with hardly any room to differentiate between the values. This could be tackled using a small enough learning rate η, but this would decrease the learning speed considerably. Too low an α value and the same problem arises.

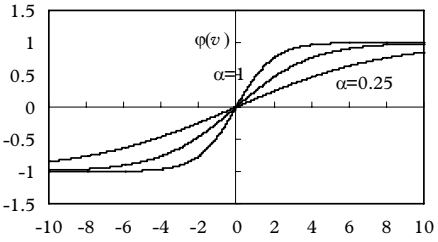


FIGURE 5.1
Bipolar sigmoid activation function defined by Equation (5.3) for α=1, α=0.5 and α=0.25. The values of the activation are in the range [–1, +1].

Just as the activation values in the advisory system, this bipolar logistic function now spans the range [–1, +1]. Another nice property of these logistic functions is that they are monotonically ascending, i.e. $\sigma(v) > \sigma(w) \Leftrightarrow v > w$. Noting that the ranking based on the total average activation value in the advisory system after a series of question is independent of the actual number of questions makes it even possible, though not necessary, to use the bipolar sigmoid as a replacement for Equation (2.5) to calculate the total activation value for a bicycle.

5.1.4. Learning to Rank

Putting all the above together, a weight update rule for the ranking ANN can now be formulated. Since the cost function defined by Equation (5.2) has no first order derivative, I basically propose a pragmatic standard Delta Rule approach. Chapter 4 showed that such an approach uses a cost function as defined by Equation (4.3), here repeated as (5.4)

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \tag{5.4}$$

To determine the update for weight w_i we again take the derivative $\partial E / \partial w_i$, but now taking into account that we use a bipolar logistic activation function

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\ &= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \sigma_2(\vec{w}\vec{x}_d)) \\ &= \sum_{d\in D}(t_d - o_d)\left(\frac{1-\sigma_2^2(\vec{w}\vec{x}_d)}{2}\right)\left(-\frac{\partial \vec{w}\vec{x}_d}{\partial w_i}\right) \\ &= \sum_{d\in D}(t_d - o_d)\left(\frac{1-o_d^2}{2}\right)(-x_i)\end{aligned} \tag{5.5}$$

Again multiplying by a learning rate $-\eta$ and adopting a stochastic approach we get the following Delta Rule for a network with output units using a bipolar logistic function as activation function

$$\Delta w_i = \eta(t-o)\left(\frac{1-o^2}{2}\right)x_i \tag{5.6}$$

The only problem in calculating this weight update in the current setting is that the value of the term $(t - o)$ is unknown. We do not have target values for each and every output unit, only preference pairs. Note that normally the term $(t - o)$ represents the error made at the output unit. Although we do not have a target value, we do have a measure for the ranking error made at the output unit *k*, namely δ_k as defined by Equation (5.1).

Now suppose we have a preference pair for nodes i and j denoting $o_i \rhd o_j$, i.e. the value of node o_i is to be higher then that of node o_j. According to Equation (5.1), if the actual values are such that $o_i \lhd o_j$, the error for node i is raised by one and the error for node j lowered by one. In such a situation we basically want to increase the value at node *i*, and lower that of node *j*. This completely agrees with our defined ranking error measure δ_k, the same way the term $(t - o)$ does when a target value t is available. This example justifies the pragmatic approach of using δ_k as a substitute for the term $(t - o)$ in Equation (5.6). Instantiating we now get the weight update rule for the ranking ANN

$$\Delta w_i = \eta\delta\left(\frac{1-o^2}{2}\right)x_i \tag{5.7}$$

where δ thus denotes the ranking error made at node o. Based on this update rule we can formulate a STOCHASTIC-GRADIENT-DESCENT-RANKING algorithm which could be used to train the weights in the ranking ANN. This algorithm is summarized in Table 5.5

STOCHASTIC-GRADIENT-DESCENT-RANKING(*TrainingExamples*, η, n_{in}, n_{out})

Each training example is a pair of the form $\langle \vec{x}, P \rangle$ where $\vec{x}$ is the vector of input values, and P is the set of strict preference pairs $\langle i, j \rangle$ denoting that the output at node i is to be strictly higher than the output at node j. η is the learning rate (e.g. 0.1). n_{in} is the number of network inputs, and n_{out} the number of output units. The weight from input i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs and n_{out} output units using a bipolar sigmoid logistic activation function
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, P \rangle$ in *TrainingExamples*, Do
    - Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network
    - For each network output unit k, calculate its error term δ_k according to Equation (5.1)
    - Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j \left(\frac{1 - o_j^2}{2} \right) x_i$$

TABLE 5.5
Summary of the STOCHASTIC-GRADIENT-DESCENT-RANKING algorithm for a single layer feed-forward ANN using a bipolar logistic activation function

5.2. IMPLEMENTATION

The advisory system itself is implemented in Microsoft.NET 2.0 using C# as core programming language. Obviously, the extension with a learning component was to be preferable implemented within the same context. Luckily many well-documented implementations of ANNs are available for Microsoft.NET, rendering a build from scratch superfluous. The conceptual design of the previous section was implemented using just such an implementation: AForge.NET.

AForge.NET is an open source C# framework [1] designed for developers and researchers in the AI field. It was originally developed by Andrew Kirillov and consists of libraries specialized for a broad range of AI related methods and algorithms. One of the libraries, AForge.Neuro, is especially targeted at the use and implementation of neural networks of all kinds.

This framework was very helpful in implementing the ranking ANN. The framework provides a standard *ISupervisedLearning* interface for this purpose which forces the implementations of the most important *RunEpoch()* and *Run()* member functions. The first function is used to perform a complete run over all training data, calling the second function for each and every training sample to actually update the network weights. This interface basically forms the learning heart of any ANN built with the framework. The framework furthermore provides some basic examples of various learning paradigms implemented in accordance with the *ISupervisedLearning* interface. One of these is a single layer activation network using a sigmoid activation function. Clearly this example was extremely helpful in setting up the ranking ANN, extending it with the ideas explored in this section.

The implementation makes extensive use of hash tables [2] to speed up the search for preference pairs and the like. Basically every search is performed using previously built hash tables throughout the complete implementation. To test the ranking ANN a simple command line utility was written. This application created the necessary training and tests sets and performed a specified number of epochs. All the while recording the various errors and results to text files for later analysis.

[1] More information about AForge.NET can be found on http://www.codeproject.com/KB/recipes/aforge.aspx and http://code.google.com/p/aforge. The latest binaries for the AForge.NET framework can be found at the second address.

[2] In computer science, a hash table, or a hash map, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a hash function into a hash, a number that is used as an index in an array to locate the desired location ("bucket") where the values should be. (http://www.wikipedia.org)

5.3. SUMMARY

This chapter dealt with the conceptual design for the ranking ANN. Given the absence of obvious target values for the output units to converge to, adopting a standard Delta Rule approach required some modifications in this context. The key insight of this chapter is using the defined ranking error δ_k of Equation (5.1) as an indication for the necessary change to the weights of an output unit to improve on ranking. Although the cost function based on this error measure has no first order derivative, the method can intuitively be expected to be useful in learning the ranking problems posed by the advisory system. The next chapter explores and tests the performance of the proposed ranking ANN on relative and absolute ranking.

6. RESULTS

Have we learnt
What we set out to learn?
Editors, from *Push Your Head Towards The Air* (An End Has A Start, 2007)

Using the single layer feed-forward ANN setup proposed in the previous chapter, this chapter discusses the results obtained in using such a setup to learn to rank. After the necessary data preparation, three experiments are conducted. The first experiment entails a proof of principle test that the network is at least able to learn the linear activation values as provided by the domain expert. If the network is not able to even learn this linear case, we can expect that it will most certainly not be able to learn the harder relative and absolute cases. Having shown that the network indeed is able to learn the linear case, the second and third experiments relate to the performance on relative and absolute ranking respectively.

6.1. DATA PREPARATION

The advisory system was launched on the website of Koga on July 22, 2008 using a set of 10 expert-provided questions with an average of 3 to 4 answers per question. The clickthrough data used to test the approach was gathered during the period from July 22 until August 22, 2008. During this period the advisory system served 7,722 unique sessions, delivering 20,174 advices, permutated by the FAIRSHARES algorithm. These advices were given for 4,449 different lines of questioning and answering, and were clicked 8,904 times.

Some lines of questioning obviously occur more often than others, the lowest being just one occurrence and the highest 882 occurrences. Obviously we want to reflect this Bayesian property in our training data, i.e. less frequently occurring events should be considered less important. Therefore the set of 20,174 advices is divided into 4,449 subsets, one for each observed line of questioning or context. Each of these context sets contains as many copies of the advice for the given context as it was witnessed in the total of 20,174 advices. Using an odd/even scheme these 4,449 subsets are randomly divided in a set of 2,225 contexts and a set of 2,224 contexts. Training and test sets can now be created from these two context sets by randomly drawing and deleting advices from the sets until there are no more advices left to be drawn. This procedure ensures we have a randomly distributed training set, roughly containing 10,000 advices and reflecting the number of occurrences of each and every context in that set. This procedure also ensures a test set of about 10,000 samples not present in the training set. Note that due to the occurrences of the contexts the actual number of samples in the training and test sets can vary.

6.2. EXPERIMENTS

This section describes the experiments conducted to test the ideas from the previous chapter. The first experiment served as a test case to determine whether the approach is at least able to learn the linear situation as provided by the expert. The second and third entail testing the ranking ANN on relative and absolute ranking.

6.2.1. Learning the Expert's Opinion

Before actually being applied to the cases of relative and absolute ranking, the ranking ANN was put to a first test learning the preferences as provided by the domain expert. Remember from section 2.2.1 that these preferences are basically governed by the activation values provided by the expert for each and every answer and bicycle. As Equation (2.5) showed, the total activation of a bicycle after a series of questions is just a linear combination of the activation values of the given answers. Given this linearity, the ranking ANN should be able to learn these preferences. If not, the approach can be expected to certainly not to work for the harder cases of relative and absolute ranking.

To test the approach the 20,174 samples were distributed across training and test sets as described above. For each of the samples (actually each line of questioning) a set of strict set preference pairs was obtained using the OBTAIN-PREFERENCES-procedure described in Table 5.1. This set of preference pairs reflected the advice of the advisory system for the given

context. In doing so a total of 704,219 preference pairs were collected for the 4,449 contexts, averaging to about 158 pairs per context. Taking the total number of occurrences of each context into account, this sums to a total of 4,445,081 preference pairs for the 20,174 samples, or about 220 pairs per sample. The slope parameter α of the bipolar sigmoid was set to 1.0 and the learning speed parameter η to 0.1. The number of epochs (learning cycles) was 50 and the experiment was repeated 10 times, each time randomly redistributing training and test sets. The averaged results are presented in Figure 6.1.

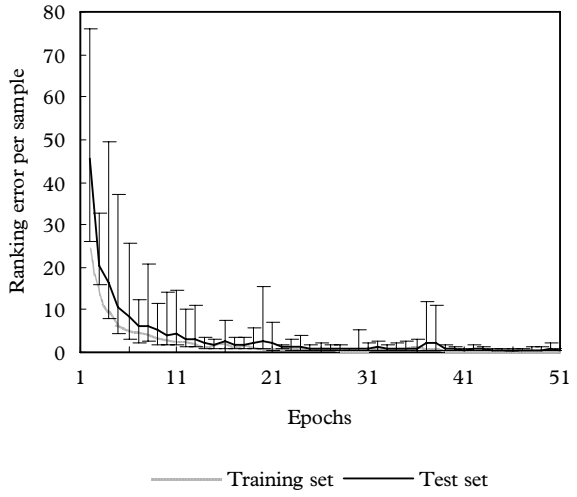


FIGURE 6.1
Testing the ranking network on learning the expert-provided rankings using α=1 and η=0.1. The figure displays the total ranking error as defined by Equation (5.2) divided by the number of samples used in training or testing the network. The confidence levels for the test set are based on 10 trials.

The figure displays the total ranking error according to Equation (5.2) divided by the total number of used samples, giving the ranking error per sample. The confidence levels over 10 trials are shown as error bars in the figure. The figure clearly shows a dramatic drop in the ranking error per sample, both for training and test set, indicating that the network can clearly learn the strict preferences. In Figure 6.2 the same data and conditions are used, but now the total number of conflicting preference pairs per sample is displayed.

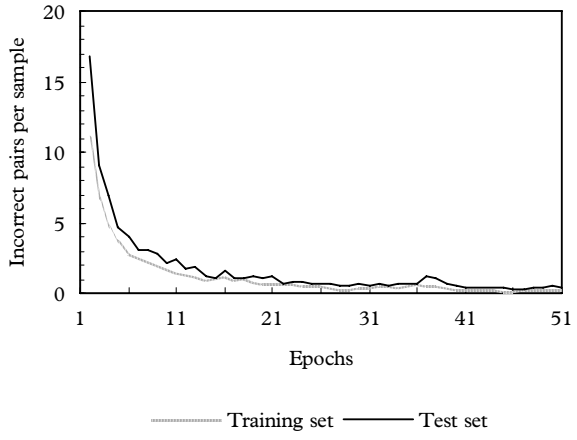


FIGURE 6.2
The total number of conflicting preference pairs in training and testing the network using the same 10 trials as used in Figure 6.1

As can be seen in the figure, the number of inconsistent pairs drops significantly after just a couple of epochs. Worth noting is how the test follows the training set, steadily improving on subsequent epochs and not showing any signs that the network is over fitting the training data. After about 50 epochs both training and test have less than one inconsistent preference pair per sample. Clearly, the network is able to learn the expert's opinion!

6.2.2. Relative Ranking

Having shown that the approach can actually work in this context, it is time to turn to the problem of relative ranking. In relative ranking we can expect a rise in the number of preference pairs the network has to learn. This is due to the FAIRSHARES interpretation of click results for objects originally sharing the same ranking value. The preference pairs for the various samples are gathered by the OBTAIN-RELATIVE-TRAINING-DATA-procedure described in section 5.1.1.2. This yields 710,852 strict preference pairs for the 4,449 contexts, or about 160 per context. Again taking the number of occurrences into account, this sums to a total of 5,006,155 pairs for the 20,174 samples, or about 248 pairs per sample. Indeed a rise of about 28 preference pairs per sample.

6.2.2.1. Setting the Network Parameters

The α=1.0 and η=0.1 parameters in the previous example were just picked with the need of showing learning convergence first. As was noted in section 5.1.3, especially the α parameter can be expected to influence learning convergence. Before seriously applying the paradigm to relative ranking different combinations of α and η were therefore tested on just 20 epochs, again averaging the results over 10 trials. These tests were performed to obtain a feel for the optimal parameter settings suitable for the current problem domain problem. Table 6.1 displays the results of these trials for the training set along with the corresponding confidence levels.

	η=0.15	η=0.10	η=0.05
α=0.25	22.41±11.61	22.99±9.94	32.94±13.27
α=0.50	20.94±9.89	**21.71±8.41**	22.78±10.56
α=1.00	22.63±11.61	24.72±7.80	**21.39±8.69**

TABLE 6.1
Testing different combinations of the α and η parameters. Each combination shows the total ranking error per training sample at epoch 20 averaged over 10 trials.

As the experiments show, both the parameter combinations α=0.50 and η=0.10, and α=1.00 and η=0.05 perform almost equally well for the combinations tested. Experiments are therefore performed using both settings alternately.

6.2.2.2. Learning Relative Ranking

Comparing the results of this test with the results for the training set of the previous experiment after 20 epochs in Figure 6.1, this problem indeed seems quite harder to learn. Various experiments over 100 epochs were conducted using various sizes for the training and test sets. The results are presented in Table 6.2.

	Train error	*Pairs incorrect*	*% Incorrect*	*Test error*	*Pairs incorrect*	*% Incorrect*
50/50	16.53±8.92	7.68±3.66	3.15±1.36	99.68±45.20	23.08±7.02	9.04±1.88
75/25	22.91±5.40	9.92±1.89	4.08±0.95	114.82±82.61	23.34±12.58	8.94±2.76
90/10	23.84±3.44	10.06±1.32	4.04±0.47	97.45±136.22	24.22±17.75	9.67±5.15

TABLE 6.2
Testing various sizes for the training and test sets. Experiments were conducted over 100 epochs using α=0.5 and η=0.1 for three cases averaged over 10 trials. The first case used a 50% sized training set, the second 75% and the third 90%. The remaining 50, 25 and 10% were used for testing. The first three columns relate to the results for the training set. The first column represents the total ranking error per sample, the second the number of pairs found inconsistent per sample and the third the total percentage of inconsistent pairs. The three remaining columns denote the same measures, but for the test set.

As these results show, the performance of the training set degrades with more training samples, while that of the test set remains roughly constant. Although the network is able to significantly drop the inconsistent pairs from 50% to about 4% and 10% for the training and test sets, it clearly has great trouble with the apparent nonlinearity reflected in these sets. These results further validate the subsequent use of the 50/50 distributed training and test sets.

One of the factors possibly contributing to this problem is the earlier mentioned FAIRSHARES interpretation of click results for objects sharing the same ranking value. To investigate this claim, another experiment was conducted in which the clickthrough data was

interpreted according to the original FAIRPAIRS interpretation. Remember that in this interpretation, preference statements are only laid down for strictly sorted objects. In dropping the preference statements for units originally sharing the same value, 704,609 preference pairs remained for the 4,449 contexts. This entailed a total of 4,447,655 preference pairs for the 20,174 samples, or about 220 pairs per sample. The results of this experiment are shown in Table 6.3.

	Train error	*Pairs incorrect*	*% Incorrect*	*Test error*	*Pairs incorrect*	*% Incorrect*
50/50	10.50±5.72	5.25±2.67	2.35±1.09	35.22±18.17	10.65±4.01	4.89±1.86

TABLE 6.3
Simplifying the FAIRSHARES interpretation. Experiment was conducted over 100 epochs using α=0.5 and η=0.1 and averaged over 10 trials.

The results clearly show that learning improves significantly if the FAIRSHARES interpretation is simplified. In doing so we basically tell the network to only update its weights for objects that were strictly sorted previously. Such an approach could still prove useful, because we might expect that objects sharing the same value previously will stay relatively close to one another. The FAIRSHARES algorithm might very well see them differently in future runs, allowing the items to be strictly sorted eventually anyway.

6.2.2.3. Why Try to Learn the Obvious?

As section 6.2.1 showed, the network is clearly able to learn the weight values reflecting the advices as given by the advisory system. Since the relative preference pairs are based on these advices, we actually have to learn two problems at the same time. The first being the original advisory system's behaviour and the second the changes needed to reflect the witnessed relative preferences in the clickthrough data. Although we can learn the weight values for the first, we could also use them directly from the advisory system. As was shown in section 5.1.3 for the bipolar sigmoid unit, we could basically use the original advisory activation values as inputs to the sigmoid and still retain the arrangement entailed by them. If we used these values the network would only have to focus on one problem and one problem only, i.e. changing these values to make them as consistent as possible with the relative preferences obtained from the clickthrough data. This method is somewhat unorthodox however, since in a traditional ANN setting all initial weight values are randomly set to some small value. Given the specific dynamics of the current problem domain, the optimization was tested nonetheless and the results are shown in Table 6.4. Though not conclusive enough the experiment indeed shows a slight improvement.

	Train error	*Pairs incorrect*	*% Incorrect*	*Test error*	*Pairs incorrect*	*% Incorrect*
50/50	9.55±6.59	4.11±2.36	1.81±0.96	31.94±12.89	10.29±2.82	4.81±1.18

TABLE 6.4
Using the activation values of the advisory system as initial weight values. Experiment was conducted over 100 epochs using α=0.5 and η=0.1 and averaged over 10 trials.

6.2.2.4. Comparing the Approaches

In a last experiment the three approaches were again tested on the same training and test set sizes, but now over 150 epochs and using the parameters α=1.0 and η=0.05. Table 6.5 shows the results for the various measures. Figures 6.3 en 6.4 display the corresponding total percentage of correctly arranged pairs in the test sets and the total number of inconsistent pairs per sample in these sets.

	Train error	*Pairs incorrect*	*% Incorrect*	*Test error*	*Pairs incorrect*	*% Incorrect*
Strict interpretation	16.99±4.96	8.17±2.22	3.25±0.97	90.35±38.55	22.08±6.47	9.04±2.57
FAIRPAIRS	9.61±4.10	5.12±2.28	2.34±1.13	39.03±12.11	11.29±2.04	5.15±1.51
Use advisory values	10.38±6.19	4.67±2.10	2.10±1.02	27.55±7.51	8.02±2.23	3.77±1.20

TABLE 6.5
Testing the three approaches for 150 epochs using α=1.0 and η=0.05 averaging them over 10 trials. In strict interpretation mode the training data was interpreted strictly according to the original FAIRSHARES interpretation. The second experiment simplified the FAIRSHARES interpretation to that of the original FAIRPAIRS interpretation. The last experiment optimized this condition even further by using the advisory system's activation values as initial weight values instead of randomizing them.

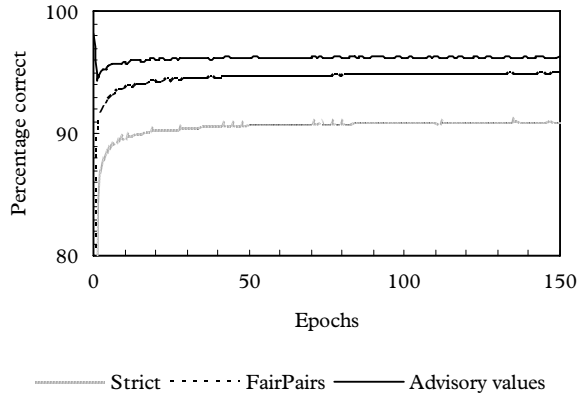


FIGURE 6.3
Comparing the three approaches over 150 epochs using α=1.0 and η=0.05 averaged over 10 trials. The figure displays the total percentage of correctly arranged pairs in the test sets. As Table 6.5 shows performance over the corresponding training sets is 2 to 3% better.

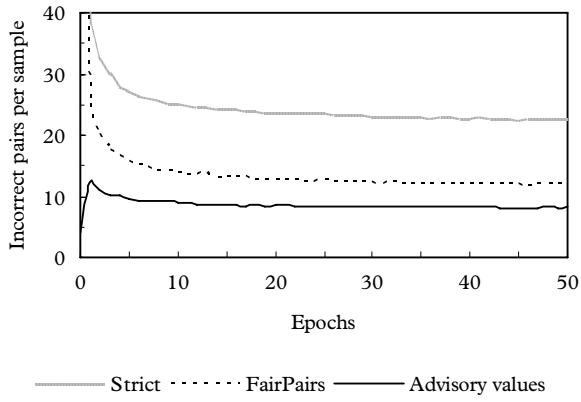


FIGURE 6.4
The same experiment as in Figure 6.3, but now displaying the actual number of inconsistent pairs per test set sample for the first 50 epochs. Note the rise in the number of inconsistent pairs for the third condition, to drop and settle after about 10 epochs.

Under these parameter conditions the optimization of the previous section actually outperforms the simplified FAIRSHARES interpretation. Note in Figure 6.3, that the initial percentage of consistent pairs is about 98% instead of the 50% for the other conditions. Since under these conditions the network scores a full 100% on the base preferences from the advisory system, the net error of 2% was induced by adding the new relative preferences. The 98% value drops when the network starts to adjust its weight values to reflect all preferences in the training data, to rise again after a number of epochs. Starting out with 98% and noting the drop to about 96%, one could argue that it would be best not to learn at all. However, in using the original activation values, we are only helping the network figuring out the base preferences. Our goal is still to learn the new relative preferences, retaining the previous base preferences as much as possible. Although performance drops from 98% to 96%, ultimate performance is still better than for the other cases. Furthermore, we can be sure the network has tried to adjust its weight values to cover both the new relative and the previous base preferences the best it could. This best of both worlds approach ensures the system actually changes it rankings. Subsequent cycles of gathering clickthrough data and learning can then further improve on these rankings, as was indicated in section 3.3.3.

The optimization issue set aside, the results clearly show the network's ability to significantly drop the ranking error, but also its inability to fully learn the preferences as generated by the FAIRSHARES and FAIRPAIRS algorithms.

This can be due to two possible issues. First the possibility exists that the relative ranking domain is just too nonlinear for the single layer network to learn. Second, the actual FAIRPAIRS algorithm sets a formal lower bound ε on the amount of necessary clickthrough data (Joachims *et al.*, 2005). This thesis did not take this lower bound into further consideration. Therefore it might very well be possible that the training data is not consistent due to insufficient clickthrough data. Supporting this hypothesis is the observation that many lines of questioning have not occurred that many times in the clickthrough data.

6.2.3. Absolute Ranking

As was indicated with the treatment of the FAIRSHARES algorithm, we could also treat the click results in a more absolute way. In this learning scenario we simply want the bicycles clicked most to seize the highest ranking positions, at the expense of bicycles clicked fewest. As mentioned earlier, the ultimate rankings can be expected to be perturbed much more in this scenario than they were for relative ranking.

The preference pairs for the various samples are now gathered using the OBTAIN-ABSOLUTE-TRAINING-DATA-procedure described in section 5.1.1.3. This yields 1,789,983 strict preference pairs for the 4,449 contexts, or about 402 per context. Taking the number of occurrences into account, this sums to a total of 6,824,506 pairs for the 20,174 samples, or about 338 pairs per sample. This number is substantial higher than the sample sizes for relative ranking. This is due to the addition of preferences over all bicycles deemed irrelevant. For example, given absolute click results for a context in which 5 relevant bicycles were not clicked at all and 40 bicycles were deemed irrelevant by the system, $5 \times 40 = 200$ preference pairs are added anyhow. Noting that the bicycles have only been clicked on 8,904 times, the number is fairly high accordingly. With the availability of more and more click results we can expect this number to drop. Although the number of preference pairs is much higher than for the relative case, most preference pairs can be considered quite trivial however and to hold almost for certain.

To gain some insight in the problem domain of absolute ranking in this context, a first experiment with different training and test set sizes was conducted. This experiment used 50/50, 75/25 and 90/10 distributions of samples over training and test sets and was allowed to learn for 150 epochs using parameters α=1.0 and η=0.05. The results are presented in Table 6.6.

	Train error	*Pairs incorrect*	*% Incorrect*	*Test error*	*Pairs incorrect*	*% Incorrect*
50/50	12.24±3.25	4.07±1.01	1.21±0.32	56.77±24.42	13.91±3.89	4.11±1.38
75/25	17.46±2.36	5.51±0.58	1.64±0.17	57.24±20.02	13.49±3.73	3.94±1.16
90/10	19.48±1.87	5.95±0.58	1.76±0.19	55.04±20.42	13.89±7.20	4.24±2.71

TABLE 6.6
Testing various sizes for the training and test sets for absolute ranking. Experiments were conducted over 150 epochs using α=1.0 and η=0.05 and averaged over 10 trials.

Again, the number of inconsistent pairs seems to rise with the size of the training set. As was the case for relative ranking, learning apparently becomes harder when more training data is available. Noting that the network performance does not actually improve in the wake of more training data, also validates the use of a 50/50 distribution of training and test samples in absolute ranking. Figure 6.5 displays the total percentage of consistent pairs after each epoch for this condition.

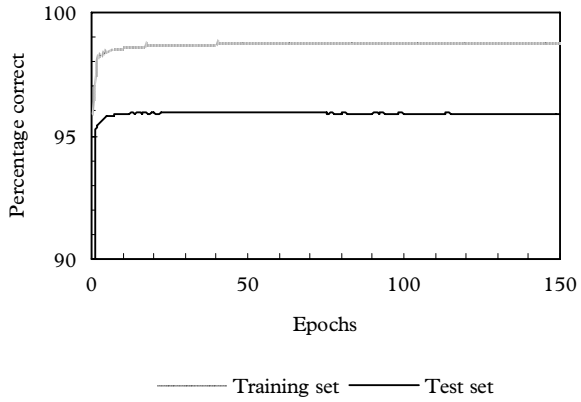


FIGURE 6.5
Experimental results for absolute ranking. The figure shows the total percentage of consistent pairs over 150 epochs using α=1.0 and η=0.05 averaged over 10 trials for both training and test sets.

Even though the number of preference pairs is considerably higher the results show an accuracy of about 96% over the test and almost 99% over the training set.

6.3. SUMMARY

In this chapter the proposed ranking ANN was put to the test. As the results showed the network was able to learn the original linear arrangements of the advisory system almost perfectly. Learning the relative preferences proved to be quite harder though. Lifting the FAIRSHARES interpretation of click results for objects originally sharing the same ranking value improved performance significantly. By using an unorthodox optimization the results could be improved even further. For absolute ranking the results proved quite satisfactory right from the start. All in all, the proposed ranking ANN is clearly able to learn the preferences as obtained from the clickthrough data to accuracy levels of at least 90% to 96% over unseen test samples and even higher for the training samples. Based on the results in this chapter, one might now be tempted to conclude that absolute ranking is to be preferred over relative ranking in this context. However, as the evaluation in the next and final chapter will show, there is a catch.

7. DISCUSSION

Questions of science
Science and progress
Coldplay, from *The Scientist* (A Rush Of Blood To The Head, 2002)

With the established results from the previous chapter it is time to return to the central research question of this thesis put forward in section 1.1. The first part of this final chapter therefore evaluates the results with an eye to this research question. Having established that the approach could actually be useful, a number of issues still remain. The second part of this chapter explores these issues, some of which need addressing before an effective application to the advisory system can be undertaken.

7.1. EVALUATION

Repeating the central research question of this thesis from chapter 1,

> *How can the entropy-driven WIZARD algorithm improve on its advices by using feedback from its users, and in doing so, become less dependent on its initial expert-provided settings?*

we can first of all state that the approach described in this thesis allows the underlying activation values of the system to be modified in such a way to reasonably reflect the preferences witnessed in the clickthrough data.

Noting that the advisory system highly resembles a single layer feed-forward ANN itself, this thesis justified the use of methods from the realm of neural networks. Other paradigms could have been used also, but translating them back to the actual modification of activation values in the system would not have been as straightforward. Answering the how question, a pragmatic Delta Rule approach was proposed which proved reasonably effective in learning both the relative and absolute interpretation of clickthrough data. Even though the proposed ranking cost function has no first order derivative, it turned out that the ranking error it uses for each output unit was quite useful as an indicator for the necessary weight changes.

The method was tested on both the relative and absolute interpretation of clickthrough data. The results show that learning relative preferences turns out to be somewhat harder than learning absolute preferences. Setting the possible issues with data sufficiency aside, this might also very well be due to the way training data is gathered for both interpretations. Note that in gathering training data for the relative case, a set of base preferences is obtained from the advisory system. This set also reflects preferences for bicycles deemed irrelevant by the advisory system and hence were not shown. In this scenario the network tries to not only reflect the relative preferences for the relevant bicycles, but also the original preferences for the ones deemed irrelevant. In gathering training data for the absolute case on the other hand, the preferences for bicycles deemed irrelevant are ignored completely. The procedure treats the group of irrelevant bicycles for a given context as a whole, only ensuring that the bicycles clicked fewest are at least preferred over the bicycles deemed irrelevant.

It would therefore be very interesting to explore the performance for both relative training data disregarding preferences over bicycles deemed irrelevant, and absolute training data maintaining these preferences. Intuitively, the performance for the first should rise, and drop for the latter. Such a study would further validate the use of one type of ranking over the other in this context. Based on the research in this thesis no conclusive statements can be made regarding this issue yet.

Although the performance of both interpretations possibly degrades when taking the preferences over irrelevant bicycles into account, it should be preferred nonetheless. If we do not, we actually loose information the system previously did possess. In maintaining this information as much as possible, the system will always have a justified top-down sorted list over *all* bicycles, relevant or not. If this information is not available only the order of the bicycles deemed relevant is justified. Since no restrictions have been laid down for the rest of the bicycles, the arrangement of these bicycles ought to be considered arbitrary. Given the fact that the advisory system uses a significance parameter to cut off irrelevant bicycles, we could end up showing bicycles totally irrelevant for a given context. Remember that the advisory system does not record which bicycles were deemed relevant earlier for a given context. The

total activation values are calculated, normalized and bicycles are cut off by the significance parameter accordingly. Transforming the weight values to activation values preserves the intended arrangement, but not the distances between the values. Normalizing and cutting them off thus might lead to a different number of bicycles deemed relevant, and hence the possible display of highly irrelevant bicycles.

Taking this issue into account, both interpretations can be expected to yield accuracy levels of about 95% over unseen test samples, performing even better on actual training samples. Although future research has to confirm this, we can expect that inconsistent pairs will only influence the arrangement on a somewhat more local level. Given the preference dynamics it is highly unlikely that for a given context a totally irrelevant bicycle ends up first in the final ranking. If this hypothesis can indeed be confirmed, the approach could perfectly well be used to improve the advices of the advisory system by learning from implicit user feedback.

In an actual implementation of the approach, the original advisory system would not even have to be modified that much. Noting that the activation values in the advisory system take on values within the range [–1, +1], we could just normalize the weight values of the ANN to this range and use them as such. Being a linear transformation, this will not perturb the arrangement of bicycles entailed by these values. Therefore, the rest of the system needs no further modification at all to reflect the learned rankings.

As was noted in this thesis, such an implementation would entail a continuous cycle of gathering clickthrough data and periodically adjusting the activation values by learning from this data. Again, much the way Google updates its rankings in periodic Google dances. This setup allows for an adaptive system in which possible future user preference shifts are accommodated quite elegantly. When the user base of the advisory systems starts to think structurally differently about certain advices, the system simply adjusts its weight values to reflect the data containing this shift as much as possible.

7.2. FUTURE WORK

In showing the approach could actually be useful, a number of interesting issues of course remain. Some of these are rather practical, other more general and academic.

7.2.1. Learning to Cluster

Although the ANN is able to learn the strict preferences to an acceptable extent, it has the side effect of strictly ranking bicycles for which no preference pairs were specified. It would be interesting to try to extend the ranking error for an output unit defined by Equation (5.1) to tackle this problem. We might want to impose a certain positive distance τ between strictly sorted pairs. Witnessing $|o_i - o_j| \geq \tau$ for two bicycles i and j, the system could then derive that the two bicycles are to be strictly sorted, and randomly otherwise.

Enforcing distance might be possible by not only modifying the ranking error when a conflicting preference pair is encountered, but moreover when the distance between the concerning output units is not sufficient enough. More formally

$$\delta_k = \sum_{\langle k,j \rangle \in P} \mathbf{1}[o_k - o_j < \tau] - \sum_{\langle j,k \rangle \in P} \mathbf{1}[o_j - o_k < \tau] \tag{7.1}$$

for $\tau > 0$. Defining the ranking error this way will ensure it produces an error term for unit k as long as there are strictly k paired output units (top or bottom) with a distance larger than τ. Care must be taken not to use a too large value for τ. If τ is too large we might run the risk that the values need to overshoot the bipolar sigmoid range [–1, +1], which is not possible. Provisional results on linear ranking, i.e. the expert's opinion, show that the network is indeed still able to learn the ranks using this measure.

7.2.2. Consumer Price

Consumers are normally heavily influenced by the price of a product and are therefore perhaps biased to click cheaper products first. Since the implemented advisory system also displays the bicycle price, it would be interesting to see if this also influences the clicking behaviour of users. If this is the case, care must be taken to rule out this bias. Since Koga designs and produces bicycles for the high-end segment, its customers are generally not that interested in price

though. Therefore, it might very well be possible that in this specific application the influence of price is less than in other contexts.

7.2.3. Multilayer Networks

From a more academic point of view it would be interesting to see whether a multilayer ANN using the ranking error term defined in this thesis, would outperform the proposed single layer variant. In principle, a multilayer ANN should be able to learn nonlinear problems much better than a single layer one. Observing the single layer ANN's inability to completely learn the relative and absolute ranking problems, adds to the hypothesis that apparently some nonlinearity is involved. Of course noting that a multilayer ANN possibly outperforms the single layer one, would be of relatively minor concern in this context. Again, there is no way of applying these results directly to the activation values as used by the advisory system. But in other contexts where one is not confined to a single layer case, this knowledge might prove very useful, and hence the applicability of the approach in general.

7.3. CONCLUSION

Addressing the central research question posed in section 1.1, the research in this thesis showed that the activation values in the advisory system can indeed be modified automatically to reflect the feedback gathered. Since the advisory system itself resembled a single layer feed-forward ANN highly, the usefulness of this type of network was further explored in this thesis. Using a set of strict preference pairs obtained from the feedback, it proved to be possible to define a weight update rule for this network, albeit a pragmatic one. Although the locality of inconsistent pairs needs to be confirmed by future research as to ensure globally rational rankings, the approach certainly looks very promising.

This thesis focused mainly on the ultimate actual application to the advisory system. However, given the definition of the ranking error term, the approach is expected to yield good results for any such domain in which the goal is to train a single layer feed-forward ANN to rank a set of output units rather than input samples. The one constraint being that at least a set of strict preference pairs for these units can be defined. Future research has to establish the performance of the method for multilayer ANNs and hence its applicability in a more general sense.

REFERENCES

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., & Hullender, G. (2005). *Learning to Rank using Gradient Descent.* Proceedings of the 22nd International Conference on Machine Learning, Bonn, Germany, 2005, 89–96.

Caruana, R., Baluja, S., & Mitchell, T. (1996). *Using the future to "sort out" the present: Rankprop and multitask learning for medical risk evaluation.* Advances in Neural Information Processing (NIPS) 8, 959–965.

Fayyad, U.M. (1991). *On the induction of decision trees for multiple concept learning*, (Ph.D. dissertation). EECS Department, University of Michigan.

Fox, S., Karnawat, K., Mydland, M., Dumais, S., & White, T. (2005). *Evaluating measures to improve websearch.* ACM Transactions on Information Systems. Vol. 23, No. 2, April 2005, 147–168.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* Prentice-Hall, Inc.

Joachims, T., (2002). *Optimizing search engines using clickthrough data.* In Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD), 2002.

Joachims, T., (2005). *Accurately interpreting clickthrough data as implicit feedback.* In Annual ACM Conference on Research and Development in Informational Retrieval (SIGIR), 154–161.

Kelly, D., & Teevan, J. (2003). *Implicit feedback for inferring user preference: A bibliography.* SIGIR Forum, 37(2), 18–28.

Kingma, S. (2008). *Onderzoek naar de Toepasbaarheid van Entropie in een Online Fietsadviessysteem.* Klein Project (TCPROJ), University of Groningen.

Mitchell, T.M. (1997). *Machine Learning.* McGraw-Hill, Singapore.

Page, L., & Brin, S. (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* Computer Science Department, Stanford University.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). *The PageRank Citation Ranking: Bringing Order to the Web.* Computer Science Department, Stanford University.

Patterson, D.W. (1996). *Artificial Neural Networks: Theory And Applications.* Prentice-Hall, Inc.

Quinlan, J.R. (1986). *Induction of decision trees.* Machine Learning, 1(1), 81–106.

Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Radlinski, F., & Joachims, T. (2006). *Minimally invasive randomization for collecting unbiased preferences from clickthrough logs.* Proceedings of the 21st National Conference on Artificial Intelligence (AAAI).

White, R.W., Ruthven, I., & Jose, J.M. (2005). *A Study of Factors Affecting the Utility of Implicit Relevance Feedback.* In Annual ACM Conference on Research and Development in Informational Retrieval (SIGIR), 35–42.

Zhang, H., Jiang, L., & Su, J. (2005). *Augmenting Naive Bayes for Ranking.* Proceedings of the 22nd International Conference on Machine Learning, Bonn, Germany, 2005, 1020–1027