OMEGA: ON-LINE MEMORY-BASED GENERAL PURPOSE SYSTEM CLASSIFIER

Kan Deng
CMU-RI-TR-98-33

November 1998

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis committee:

Andrew W. Moore (Chair)
Dean Pomerleau
Scott E. Fahlman
Christopher G. Atkeson, Georgia Institute of Technology

ABSTRACT

A *time series* is a sequence of data points in which the order of the data points is important. In many cases, each data point consists of both inputs and outputs. The reason that the time order of such a time series is important may be that at a certain time instant, the outputs are determined not only by the current inputs, but also by some of the more recent inputs and outputs. If we extend the input vector to include those previous inputs and outputs in addition to the current inputs, then the outputs are fully determined by the expanded input vector. Thus, we can transform a time series into a set of data points where the time order is no longer important.

Given a time series, a system classifier's purpose is to determine to which category the underlying system belongs, among a set of pre-defined candidate categories. To do so, our system classification algorithm transforms the time series into a set of expanded data points. It then employs a memory-based classifier to calculate a sequence of probabilities that measure how likely these expanded data points are to belong to each of the categories. Finally, it uses likelihood analysis and hypothesis testing to summarize these classification results. Our method can also handle the classification of non-time series.

Our contributions include: (1) the methodology that decomposes time series classification into the likelihood analysis of a sequence of classifications; (2) a new memory-based classifier that has many desirable properties; (3) re-organization of the memory in the form of a cached kd-tree that greatly improves the computational efficiency of information retrieval and memory-based learning algorithms; and (4) fast feature selection based on intensive cross-validation and greedy searching.

Compared with other methods, our new system classifier is simple to understand, easy to implement, robust for various types of systems, and adaptive to datasets with different densities and/or noise levels. It is capable of distinguishing the various categories of the underlying system without requiring any predefined thresholds. It is efficient not only because it can perform classification quickly, but also because it can focus on the promising categories while ignoring the others after only a few iterations. Based on our empirical evaluations, our method tends to be more accurate than other methods.

DEDICATION

To my family.

To my advisor.

ACKNOWLEDGMENTS

When I write this acknowledgments, I realize that my formal education will end very soon. Reviewing the six years and three months of my graduate study at the Robotics Institute of Carnegie Mellon University, not only has CMU given me a very solid education in computer science and robotics, but also it helped me to build my confidence to pursue career goals after my graduation. However, my experience in CMU is like a medicine: it is good for my health but sometimes it tastes bad.

First, I sincerely thank my advisor, Andrew Moore for his friendship and his patient and intellectual guidance. I feel very lucky to be his student; otherwise, I cannot imagine how I could have finished my Ph.D. study. My other three committee members, Christopher Atkeson, Dean Pomerleau and Scott Falman, were also very helpful. I especially appreciate their tough questions, which pushed me to make my work more solid.

My labmates and classmates, Leeman Baird, Justin Boyan, Steve Chen, Fabio Cozman, Scott Davis, Remi Munos, Michael Nechyba, Andrew Ng, and Jeff Schneider, contributed immeasurably to my research by arguing about the research ideas. I appreciate Michael Nechyba and Andrew Ng for their selfless help which introduced me to the culture of computer science research. Besides, I thank all my friends in CMU's machine learning community, especially: Rich Caruana, Frank Dellaert, Geoff Gordon, Andrew McCallum, Peter Stone, Astro Teller and Belinda Thom.

I am very happy to see that the number of Chinese students in the Robotics Institute has kept growing. A friendly social community is important to everybody, especially for foreign students. Fortunately, our RoboChinese, Heng Cao, Peng Chang, Mei Chen, Mei Han, Yingli Tian, Yanghai Tsin, Huadong Wu, Xingxing Yu, Dongmei Zhang, Li Zhang, Liang Zhao, as well as their families, is such a friendly community.

Almost all the CMU faculty to whom I have talked are very supportive. I want to thank following professors for their help these years: Avrim Blum, Christos Faloutsos, Tom Mitchell, Matt Mason, Roy Maxion, Sebastian Thrun, Manuela Veloso and Yangsheng Xu. Thanks also go to Suzanne Crow, Marie Elm, Carlyn Ludwig, Sandy Rocco, Ruth Wiehagen and Marce Zaragoza.

Since my undergraduate background is not in computer science, without the help of my folks: Yanbin Jia, Tang Lei, Harry Shum, Jiawen Su, Yalin Xiong and Dajun Zeng, I can not imagine how I could have survived in the Robotics Ph.D. program. I also appreciate my former advisor, Katia Sycara, as well as Dundee Navin-Chandra, the other principal researcher in our group, for their support and guidance.

Finally, my special thank goes to my dear wife, Xuemei Gu. A happy family life is the solid foundation of my pursuit of a career.

Contents

Chapter 1 13
Introduction 13

1.1 What is system classification? 13
1.2 The applications of system classification 15
1.3 The assumptions of OMEGA 16
1.4 Related fields 18
1.5 The system classification approaches 19
1.6 Thesis outline: 22
1.7 *: Hidden Markov Model (HMM) 23

Chapter 2 27
Memory-based System Classification 27

2.1 Likelihood Analysis 27
2.2 Hypothesis Testing 32
2.3 Efficiency Issues 35
2.4 Pre-processing 37
2.5 Memory-based learning 40
2.6 Summary 48

Chapter 3 49
Tennis Style Detection 49

3.1 Experimental Design 49
3.2 OMEGA Result Analysis 52
3.3 Comparison with Other Methods 55
3.4 Summary 58

Chapter 4 59
Logistic Regression as a Classifier 59

4.1 Classification methods 60
4.2 Global logistic regression 63
4.3 Locally Weighted Logistic Regression 66
4.4 Comparison Experiment 71
4.5 Summary 75

Chapter 5 77
Efficient Memory Information Retrieval 77

5.1 Efficient information retrieval 77
5.2 Kd-tree Construction and Information Retrieval 79
5.3 Cached Kd-tree for Memory-based Learning 82
5.4 Experiments and Results 88
5.5 Summary 92

Chapter 6 95
Using Kd-trees for Various Regressions 95

6.1 Locally Weighted Linear Regression 95
6.2 Efficient locally weighted linear regression 97
6.3 Technical details 101
6.4 Empirical Evaluation 102
6.5 Kd-tree for logistic regression 107
6.6 Empirical evaluation 111
6.7 Summary 114

Chapter 7 117
Feature Selection 117

7.1 Introduction 117
7.2 Cross Validation vs. Overfitting 119
7.3 Feature selection algorithms 120
7.4 Experiments 124
7.5 Summary 132

Chapter 8 133
Driving Simulation 133

8.1 Experimental data 134
8.2 Experimental results 136
8.3 Comparison with other methods 139
8.4 Summary 144

Chapter 9 147
Real World Driving 147

9.1 Data collection 147
9.2 OMEGA result 150
9.3 Comparison with other methods 153
9.4 Summary 155

Chapter 10 157
Conclusion 157
10.1 Discussion 157
10.2 Contributions 160
10.3 Future research 162
10.4 Applications 163

Appendix A 167
Chinese Handwriting Recognition 167
1.1 Feature selection for Chinese handwriting recognition 167
1.2 Future work 171
Bibliography 173

Chapter 1

Introduction

1.1 What is system classification?

With the dramatic development of computer science and technology, we are on the edge of making many machines intelligent by embedding computer systems in them. For example, people have known how to cook rice for thousands of years, but only in the last two decades was the neuro-controlled automatic rice cooker invented. In the near future, by embedding computer chips in other kitchen devices, people will be further liberated from the tedious and exhausting cooking tasks which their predecessors have suffered for many centuries. Similar things will also happen to vehicles. In next century, we expect cars will become autonomous. Once the passengers tell the vehicle where to go, they can go to sleep or watch television. In the short term, cars will become smarter, if not completely autonomous. The smart car's intelligence has many aspects, including the ability to tell if the human driver's sobriety level is good enough for further operation. If necessary, the monitoring system may warn the driver to stop for a break. This is important because inattention may lead to the fatal accidents. In the U.S. 1996, there were over 37,000 automobile accidents involving fatalities, in which 42,000 people were killed. Among these cases, over 21,000 were single vehicle accidents resulted in 22,500 fatalities [Batavia, 98].

The technique we explore in this thesis is useful for driving sobriety monitoring, as well as other applications. Let's imagine that we have a vehicle full of smart sensors which can tell the velocity of the vehicle, its orientation, its lateral distance to the center of road, and the distances to the other vehicles nearby, etc. If we regard a driver as a system, the above variables are the inputs to the system. Based on the inputs, the driver has to properly steer and control the gas and brake pedal. Thus the outputs of the system are the vehicle's steering angle and its acceleration. Suppose we also measure the outputs. Let's take a record of both the input and output values every time unit, say 0.1 second. We will get a multi-dimensional time series. The driving time series varies from case to case, even if the driver is the same person and his/her sobriety condition is identical. The reason is that road conditions and traffic may be different, and these differences will make the driver's response (system outputs) differ from case to case. However, we believe if the driver is sober, his driving behavior time series should be consistent with his historic "sober" driving time series. Otherwise, if the driver is intoxicated, his driving (system outputs) may differ from those normal cases in memory. In addition, an intoxicated driver may create some unusual input scenarios because of his careless behavior.

How can we formalize the informal discussion above into a useful and reliable algorithm? In statistical terms, to classify the driving style we want to calculate $Prob(S_{normal} \mid \boldsymbol{O}_q)$, which is the probability that a driver's sobriety is normal, as inferred from the observation of their driving behavior. $\boldsymbol{O}_q$ represents the current driver's driving behavior time series; q stands for *query*, implying that the underlying state of the driver's sobriety condition is unknown. S_{normal} is the event of the driver being sober. To calculate $Prob(S_{normal} \mid \boldsymbol{O}_q)$, we compare the unclassified time series $\boldsymbol{O}_q$ with those time series in memory generated by the same driver when he was sober. If $Prob(S_{normal} \mid \boldsymbol{O}_q)$ is higher than a certain threshold, the driver seems to be sober. Otherwise, they are intoxicated. Sometimes, the task can be more complicated. For example, the police department may want to distinguish drowsiness from drunkenness. In this case, we should calculate $Prob(S_{intoxicated} \mid \boldsymbol{O}_q)$ or $Prob(S_{drowsy} \mid \boldsymbol{O}_q)$, as well as $Prob(S_{normal} \mid \boldsymbol{O}_q)$, the largest value indicates the driver's most likely sobriety condition.

Generally speaking, we define the task of a *system classifier* as the following: given a set of observations of a system's inputs and outputs, a system classifier is to figure out the underlying mechanism which generates these observations.

1.2 The applications of system classification

- System diagnosis:

  No machine can work perfectly all the time. People need to know when to fix the machines and how to fix them. This is the purpose of system diagnosis. System diagnosis can be done by human experts. However, in some cases an on-line autonomous system diagnosis tool is preferred, because for some complicated machines, no single human expert can understand every detail. Also, it is hard to ask the human expert (or a group of them) to do the diagnosis job twenty-four hours a day, seven days a week, in all possible situations including dangerous environments.

- Surveillance

  With the progress of video tracking and speech signal processing, we are on the edge of implementing an autonomous system to liberate human operators from surveillance jobs which may be tedious and last long hours. We expect that these autonomous systems will have better performance than that of a sleepy human operator. Similarly, we expect to apply this technique to make some military surveillance devices more intelligent. For example, we can invent an automatic radar monitoring system so that the soldiers can be liberated from the radar desk, especially during the tedious period when nothing unusual happens.

- Human behavior monitoring

  Every year in the U.S., thousands of people die in traffic accidents. Some of these accidents are caused by the exhaustion of the drivers. It would be desirable to have a way to monitor the behavior of the human operators and give them warnings if necessary.

Another possible application is that with the booming of virtual reality stores on the Internet, more and more customers will go shopping via the Internet. Technically the e-stores' server is capable of tracking the behaviors of the visitors, to detect the customers' purpose and/or preference. This prospect does raise many moral, ethical, and social issues which are beyond the scope of this thesis.

- Human skill transition and evaluation

  Sometimes people want to learn physical skills from the masters. Some skills should be passed on before the old masters die. Some skills should also be transferred to robots, because robots can work in remote or inhospitable environments. Therefore, we need some ways to transfer skills and evaluate the learned performance.

- Financial monitoring

  We can apply the techniques of this thesis to keep an eye on the financial climate, which is useful and rewarding.

1.3 The assumptions of OMEGA

In this thesis, we investigate and extend memory-based learning for general propose on-line system classification. We name this new technique On-line MEmory-based GenerAl purpose system classifier, (OMEGA). OMEGA calculates $Prob(S_p \mid \boldsymbol{O}_q)$, which is the probability that the underlying mechanism of a set of observations $\boldsymbol{O}_q$ is system S_p. It has following the assumptions:

1. OMEGA does not approximate the closed-form mechanism of the underlying system. We also assume that the unknown underlying generator of $\boldsymbol{O}_q$ must be one of a *finite* set of candidate systems. This assumption is not so bad as it looks. For the example mentioned above, it is unnecessary to require every police officer to know the psychological and physiological processes underlying intoxication. Instead, if a traffic police officer can correctly

detect any unusual driving behavior, his job is well done.

2. For the same example, to calculate the probability $Prob(S_{normal} \mid \boldsymbol{O}_q)$, we compare the query driving time series $\boldsymbol{O}_q$ with those "sober" driving time series in memory. In other words, we assume that we have collected some training observations of each candidate system's behavior before the classification job for $\boldsymbol{O}_q$ comes. Notice that if there are only a few sober driving time series samples in memory, it is still possible to approximate $Prob(S_{normal} \mid \boldsymbol{O}_q)$. Of course, the fewer the sober samples in memory, the less reliable the approximated $Prob(S_{normal} \mid \boldsymbol{O}_q)$ is.

3. Originally motivated to classify time series, our research ends up with a general purpose technique which is also capable of general pattern classification. In other words, the observation $\boldsymbol{O}_q$ may be a time series, but this is not necessary. As defined, $\boldsymbol{O}_q$ is in fact a set of observation data points, while a data point consists of the inputs of the concerned system at a certain time instant and their corresponding outputs. When $\boldsymbol{O}_q$ is not a time series, we can shuffle its data points randomly.

4. OMEGA works best for those systems whose input and output are fully observable, and the output are fully determined by the input. Note that this assumption is often violated in practice. For example, in driving domain, a driver's control action may be influenced by some of his hidden psychological and physiological factors. However, like other machine learning methods, we assume a driver control action is somehow predictable by some observable input variables.

5. The inputs and outputs of any candidate systems can be of any type. They can be continuous or discrete, (including categorical), or even a mixture of the two. However we assume the types of the input and output of all candidate systems are the same.

6. We study stochastic systems; in other words, given a certain input, the corresponding output is stochastic. The conditional distribution of the output given a certain input can be of any type. For some systems, the outputs corresponding to an identical input may scatter

around a center, so that the conditional distribution can be roughly formed as Gaussian. However, as a general purpose approach, OMEGA does not require this uni-modal assumption.

1.4 Related fields

Conventionally, classification is to detect to which category a single data point belongs. However, since a time series consists of a sequence of data points, system classification involves a sequence of classifications, then summarize them so as to draw an overall conclusion.

System classification is different from *system identification*. The latter estimates the configuration and the parameters of an unknown system, but system classification's task is to recognize an unknown system, without necessarily estimating its parameters.

Another closely related field is *fault detection*, which is also referred to as *novelty detection*. The task of fault detection is to tell whether or not a system's current behavior is out of the tolerance of its normal performance. System classification is different from fault detection because system classification concerns multiple systems, and it assumes that every system always works normally. The difficulty of fault detection is that its training data is usually unbalanced; in other words, the majority of the training data is collected when the system works normally. However, it is still straightforward to apply OMEGA to solve the fault detection problem: we approximate $Prob(S_{normal} \mid \boldsymbol{O}_q)$, if this probability value is lower than a certain threshold, the system is abnormal; in another case, even if the value of $Prob(S_{normal} \mid \boldsymbol{O}_q)$ is higher than the threshold, but it is not reliable (its confidence interval is too large), the state of the system is uncertain. The threshold can be decided by hypothesis testing methods.

1.5 The system classification approaches

There are two approaches to system classification: comparing the system parameters, or comparing the predictions.

Comparing the system parameters

This approach is similar to system identification: we approximate the unknown system's parameters first, then classify the system based on the comparison of the system parameters. For example, suppose we have a collection of observations $(x_1, y_1), (x_2, y_2), ..., (x_T, y_T)$, where x's are the system's inputs, and y's are the outputs. Temporarily, let's assume based on prior knowledge that we know these signals were generated by a linear system:

$$y = \beta_o + \beta_1 x + \xi$$

If there are sufficient observations, we are able to approximate the system parameters, β_0 and β_1. To detect if the observation signals $(x_1, y_1), (x_2, y_2), ..., (x_T, y_T)$ were generated by a particular one-input-one-output linear system whose parameters are α_0 and α_1, we can straightforwardly check if the α's and β's are close to each other respectively.

This approach looks simple, but it has three problems: (1) We need the prior knowledge of the closed-form formula of the system. (2) Before we employ this approach, we should make sure that identical systems must have the same parameters. When the system is more complicated than a linear one, different sets of parameters may correspond to the same system. Section 1.7 gives an example.

In some circumstances like chemical manufacturing process, it is hard to get precise mathematical models of the systems. Therefore, to design a robust, general purpose system classification package, we will resort to the other approach.

Comparing the predictions

Given a set of observations whose underlying generator is unknown, the prediction approach temporarily assumes the unknown underlying system is a certain candidate one. Based on our knowledge of this assigned candidate system, we can predict the outputs corresponding to the inputs of the observations. If the candidate system is indeed the real underlying system, the predictions must be close to those observed outputs. Otherwise, the assumption is not correct.

In more details, let's suppose there is a collection of observations, $(x_1, y_1), (x_2, y_2), ..., (x_T, y_T)$. To figure out whether or not they were generated by a certain linear system,

$$y = \alpha_o + \alpha_1 x + \xi$$

with particular α_0 and α_1 values, we can use the above formula to predict the y value given a certain x. Therefore, we will get a sequence of predictions, $\hat{y}_1, \hat{y}_2, ..., \hat{y}_T$. The difference between them and the observed values $y_1, y_2, ..., y_T$ are the residuals. If the residuals are close to zero, the system with α_0 and α_1 as parameters is likely to be the underlying system which generated the observations.

Even with only one observation, the prediction-based approach can still start to work, though the result will be unreliable. With more observations, this approach can be expected to have improved performance. Therefore, the prediction-based approach is ideal for on-line applications.

Up to now, we have assumed the system is linear. The linear system model has been popular for several decades because it is simple and in many cases it is reasonable. For non-linear systems, we can apply non-linear function approximators such as neural network to do the prediction job, so that the prediction-based system classification approach still works [Petridis et al., 96].

The neural prediction approach uses neural networks to approximate every candidate system. If there are one hundred candidate systems, there will be one hundred neural networks. To calculate $Prob(S_p \mid \boldsymbol{O}_q)$, we compare the outputs of $\boldsymbol{O}_q$ with the predictions of the neural net, which represents S_p, given the corresponding inputs.

Although a neural classifier is capable of starting its job to detect the unknown underlying generator of $\boldsymbol{O}_q$ with very few data points in $\boldsymbol{O}_q$, we should clarify that it does need a large amount of training data to train the neural net to precisely represent the candidate system, say S_p. The training data are collections of observations similar to $\boldsymbol{O}_q$, but they are labeled by their underlying systems, say S_p.

There are three concerns with a neural prediction-based system classification approach. (1) It is computationally expensive to train a neural network. Things become worse when new training data is constantly becoming available. (2) Even if we can afford a supercomputer which is capable of updating the neural networks quickly, we will have another trouble: *interference*. The neural networks will evolve to fit the new data, and the old data will eventually lose their impact. (3) Every candidate system's neural network, should be included in the competition, until there is convincing evidence that a certain candidate's neural net is less competitive. Therefore, when there are a huge number of candidates, the computational cost becomes prohibitively expensive, especially in the early stage when all the candidates are involved in the process.

To overcome these problems, the memory-based learning approach is a good choice. A memory-based learning system stores all the training data in memory. When new data arrive, they will be stored into the memory together with the old data. All processing of the training data is deferred until a prediction query is made. Therefore, less interference happens. Second, as we will introduce in the later chapters, the memory-based learning methods do not require any parametric model of the system. Hence, there is no model which needs to be trained off-line. Third, by reorganizing the memory in kd-tree form and caching some information into the tree nodes, the memory-based learning process can be done very quickly. Fourth, also with the help of kd-tree, we can focus on the more promising candidates from the very beginning.

1.6 Thesis outline:

The thesis research consists of four parts: (1) The top-level principle of OMEGA, which is to combine a series of classifications in the context of likelihood analysis and hypothesis testing. (2) A new memory-based classifier, which has many improvements over existing classifiers. (3) Efficient memory information retrieval and regression using the cached kd-tree technique. (4) Cross-validation for feature selection and parameter tuning. Although (2) (3) (4) are three independent research topics, they act as components in the OMEGA approach.

Chapter 2 introduces the principle and framework of OMEGA to give the readers a birds-eye view of the whole approach and the relationship of the various components. As a demonstration, in Chapter 3 we use OMEGA to classify different styles of tennis playing, and compare OMEGA's performance with those of other methods. From Chapter 4 to Chapter 7, we discuss the components of OMEGA in details. Chapter 4 explores the new memory-based classifier, and compares it with other classification methods. In Chapter 5 and 6, we discuss a technique to re-organize the memory so as to improve the efficiency of information retrieval and regression. In Chapter 7, we talk about cross-validation, which is useful for feature selection and parameter tuning for the learning process. After that, we combine all the techniques into the OMEGA toolkit, and apply it to classify different driving styles, using both simulation data and real world data, referring to Chapter 8 and 9. Finally, Chapter 10 is a summary of all the research work, the contributions, and the open questions.

Figure 1-1 illustrates the structure of OMEGA system and the organization of the thesis.

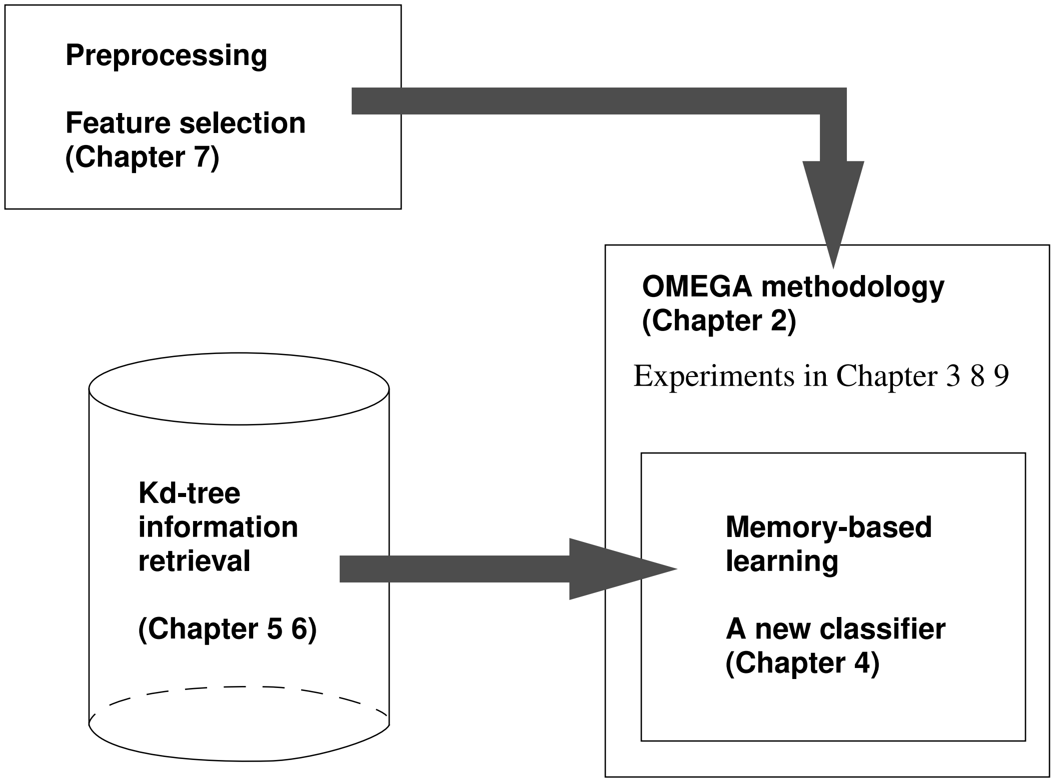


Figure 1-1: The structure of OMEGA system and the organization of the thesis.

1.7 *[1]: Hidden Markov Model (HMM)

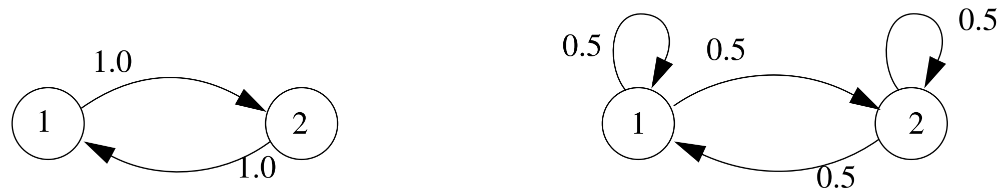
HMMs have been widely accepted as a time series analysis tool. They stand between the parameter comparison approach and the prediction approach. On one hand, it approximates the parameters of the hidden Markov model; on the other hand, it use a method similar to the prediction approach to evaluate whether or not two hidden Markov models with different parameters are in fact identical. There is no doubt HMM is an important and interesting technique, but it is questionable if it is a robust, general purpose system classification tool.

Before we argue the reasoning of our conclusion, let's give a brief introduction to HMM. HMM assume that a system has some internal hidden states. As time passes, the system jumps from

1. This section can be skipped if the reader does not have much interest in HMM.

Hidden State	Observation	
	A	B
1	0.5	0.5
2	0.5	0.5

Hidden State	Observation	
	A	B
1	1.0	0.0
2	0.0	1.0



An observation sequence: A A B A B A B B A B A B A A B B. The above two models have the same chance to be the generator of the observation sequence.

Figure 1-2: Two identical HMMs

one internal state to another. Each hidden state generates an observable signal, but it is possible that one state has several possible signals, and the same signal may be shared by several hidden states. The observation time series generated by a HMM is stochastic in two aspects: (1) The jumps are stochastically decided by the transition probabilities among the hidden states. (2) Even for the same hidden state, we may observe differing signals. Two two-state HMMs are illustrated in Figure 1-2. The numbers attached to the arc links are the transition probabilities. Since all the transition probabilities in Figure 1-2 (a) are 1.0, the system definitely switches its hidden state every time step. The system of Figure 1-2 (b) has a 50% chance to stay in the same hidden state, but has the other 50% chance to switch. The tables above the diagrams indicate the probabilities linking the hidden states to the observations, A and B.

If two time series are different, the underlying HMMs' parameters must be distinguishable. The HMM parameters include the transition probabilities and the probabilities linking the hidden states to the observations.

However, notice that an identical system may have a different structure and parameters. The system of Figure 1-2 (a) is in fact equivalent to that of Figure 1-2 (b), because both systems have exactly the same chance to generate the observation sequence written in Figure 1-2. Therefore, to detect if two HMMs are equivalent, we cannot simply compare their parameters. Instead, we should use the first HMM to generate a sample observation sequence, then find a way to measure how well the sample observation sequence fits the second HMM.

HMM were originally explored by the speech recognition community. For speech, there is no input, all the signals can be regarded as outputs. To extend HMM to systems which have both inputs and outputs, one solution is to enumerate every possible combination of input and output as a state. Thus, the number of states explodes as the number of possible input and output values increases. Therefore, in our opinion, HMMs did not easily fit our tastes.

Chapter 2

Memory-based System Classification

In this chapter, we study the methodology of the On-line MEmory-based GenerAl purpose system classification technique (OMEGA). OMEGA combines a series of classifications in the framework of likelihood analysis and hypothesis testing. In this chapter, we will introduce likelihood analysis and hypothesis testing first, then discuss efficiency issues. Afterwards, we will summarize pre-processing method and briefly discuss alternative memory-based classification and prediction methods.

2.1 Likelihood Analysis

As defined in the last chapter, a system classifier should estimate the underlying generator of a set of observation signals $\boldsymbol{O}_q$, under the assumption that the generator must be one of a finite number of candidate systems, $S_1, ..., S_n$. For example, given a time series of a vehicle's behavior in traffic, $\boldsymbol{O}_q$, the task of system classification is to tell the sobriety state of the driver, S_p, assuming that we have sufficient knowledge of the behavior of sober drivers, sleepy drivers and even intoxicated ones.

Average residuals

If we treat a driver as a system, the outputs are the control actions of the driver: the positions of the steering wheel and the gas and brake pedals. A driver chooses his control actions according to the state of the vehicle, the road condition and the traffic condition, as well as his previous actions, hence the inputs of the system are the speed of the vehicle, its orientation, its distance to the center of the road, the road curvature, the distances from the vehicle to the neighboring ones in traffic, as well as the driver's previous control of the steering angle and the gas/brake throttle. Usually an observation sequence consists of a series of input-output data points $\{x_{qi}, y_{qi}\}$. Temporarily let's assume that at any time instant, the output y_{qi} is fully controlled by the input x_{qi}. We will come back to this topic in Section 2.4.

We do not know which candidate system generated the observation sequence $\boldsymbol{O}_q$, but let's guess it is the first system, S_1. Assuming somehow we have sufficient knowledge about S_1, so that given a specific input x, we can predict the output $\hat{y}|S_1$. Since $\boldsymbol{O}_q$ consists of a series of data points $\{x_{qi}, y_{qi}\}$, $i = 1, ..., N_q$, if we pick up one input x_{qi} from them, we can predict the corresponding output, $\hat{y}_{qi}|S_1$. If S_1 is indeed the real underlying generator of $\boldsymbol{O}_q$, $\hat{y}_{qi}|S_1$ is expected to be close to the observed output, y_{qi}. In other words, the smaller the residual between $\hat{y}_{qi}|S_1$ and y_{qi}, the more likely the unlabeled data points $\{x_{qi}, y_{qi}\}$, $i = 1, ..., N_q$, were generated by S_1. If there are N_q data points in $\boldsymbol{O}_q$, we will get N_q such residuals. We can use the average of these residuals as a measure of the likelihood.

If there are n candidate systems, we can calculate n such averages of residuals. The smallest one indicates the particular candidate system which is most likely to be the generator of the unlabeled data points, $\{x_{qi}, y_{qi}\}$, $i = 1, ..., N_q$, or equivalently, the observation sequence $\boldsymbol{O}_q$.

The average residual is a useful metric, but it treats every residual equally. This is not desirable, because some $\hat{y}_{qi}|S_p$'s have better quality than the others, and they should therefore have stronger impact on the likelihood measurement. Hence, we explore the likelihood approach in next subsection.

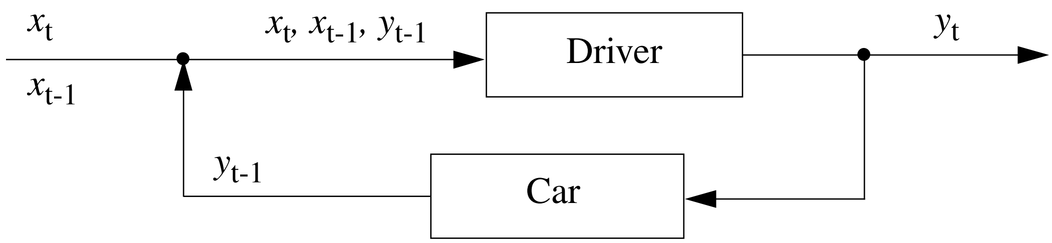


Figure 2-1: A driving system with one delay and feedback.

Likelihood

From the Bayesian point of view, the system classification problem can be structured as calculating $P(S_p \mid \boldsymbol{O}_q)$, $p = 1, ..., N_s$, which is the probability that given an observation sequence $\boldsymbol{O}_q$, the underlying generator is the p'th candidate system. The biggest $P(S_p \mid \boldsymbol{O}_q)$, $p \in \{1, \ldots, N_s\}$, indicates the most likely system which generated $\boldsymbol{O}_q$.

According to Bayes rule, $P(S_p \mid \boldsymbol{O}_q)$ is proportional to $P(\boldsymbol{O}_q \mid S_p)$, when the prior probability $P(S_p)$ is fixed. Let's assume $\boldsymbol{O}_q$ can be transformed into a set of data points, $\{x_{qi}, y_{qi}\}$, $i = 1, ..., N_q$, so that temporal order is not important. If so, the following equations hold:

$$P(\boldsymbol{O}_q|S_p) = \prod_{i=1}^{N_q} P(x_{qi}, y_{qi}|S_p) = \prod_{i=1}^{N_q} P(y_{qi}|S_p, x_{qi})P(x_{qi}|S_p) \qquad (2\text{-}1)$$

However, temporal order is important for most system because of the system's delays and feedback. Figure 2-1 illustrates a symple driving system with one delay and feedback. For such a system, $P(y_{qi} \mid S_p, x_{qi})$ in Equation 2-1 should be changed to $P(y_{qi} \mid S_p, x_{qi}, x_{q,i-1}, y_{q,i-1})$, because the current system output is not only determined by the input at the moment, but also the delay $x_{q,\, i-1}$ and the feedback $y_{q,\, i-1}$. To be convenient, we use X_{qi} to represent the conjunction of x_{qi}, $x_{q,i-1}$ and $y_{q,i-1}$. $P(x_{qi} \mid S_p)$ should be changed to $P(X_{qi} \mid S_p, X_{q,i-1})$. The reason for the appearance of $X_{q,i-1}$ is that the two components of X_{qi}: $x_{q,i-1}$ and $y_{q,i-1}$, may be partially dependent on their ancestors: $x_{q,i-2}$ and $y_{q,i-2}$. Theoretically, $P(X_{qi} \mid S_p)$ is no bigger than $P(X_{qi} \mid S_p, X_{q,i-1})$; however, in practice, we find that in many cases that we can substitute $P(X_{qi} \mid S_p)$ for $P(X_{qi} \mid S_p, X_{q,i-1})$, and the classification results are still satisfactory. Therefore, for a system with one delay and feedback, the following equations hold:

$$P(\boldsymbol{O}_q|S_p) = \prod_{i=1}^{N_q} P(y_{qi}|S_p, X_{qi})P(X_{qi}|S_p, X_{q,i-1}) \approx \prod_{i=1}^{N_q} P(y_{qi}|S_p, X_{qi})P(X_{qi}|S_p) \tag{2-2}$$

Therefore, to calculate $P(S_p \mid \boldsymbol{O}_q)$, an approach is to approximate $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$. To explain their physical meanings, let's study the driving domain again. Suppose we want to distinguish a certain driver's different driving behaviors under two sobriety conditions: sober and intoxicated. We notice that corresponding to the same scenario, X_{qi}, the driver's response when he is intoxicated tends to be different from that when he is sober; in other words, facing a certain situation X_{qi}, the probability that the driver gives a certain response y_{qi} while he is intoxicated, i.e. $P(y_{qi} \mid S_{intoxicated}, X_{qi})$, may be different from the probability when he is sober, i.e. $P(y_{qi} \mid S_{sober}, X_{qi})$. Therefore, we believe that the probability $P(y_{qi} \mid S_p, X_{qi})$ is a good metric of the driver's sobriety condition.

Besides, we also notice that an intoxicated driver may encounter situations which are not familiar to him when he is sober. For example, an intoxicated driver may let his car be very close to other vehicles in traffic, but when he is sober, the driver may realize that this situation is so dangerous that he would try to avoid it. In other words, the probability that a sober driver encounters a certain scenario X_{qi}, i.e. $P(X_{qi} \mid S_{sober})$, may be different from the probability that he faces the same situation when he is intoxicated, i.e. $P(X_{qi} \mid S_{intoxicated})$. Notice that if a sober driver intentionally does something new, our system classifier may misunderstand him as being drunk. But, we do not have to blame our system classifier for that. Tom Hanks' performance in Forrest Gump is highly appreciated. Why? Because Tom mimicked the dummy's behavior

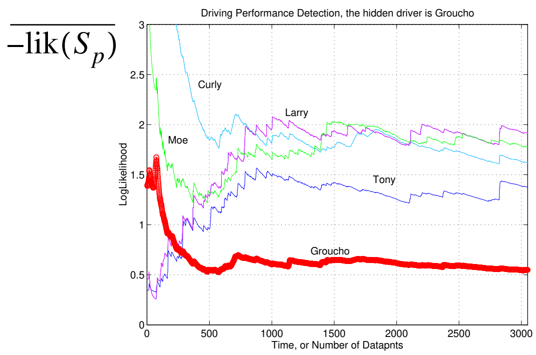


Figure 2-2: The X-axis is the number of observation data points. The Y-axis is the average of the negative log likelihood. To find the underlying system, one should compare the tails of the curves. Because Groucho's tail is closest to the X-axis, Groucho is most likely the underlying generator of the observation sequence.

seamlessly. Hence, we believe that by combining the two probabilities, $P(X_{qi} \mid S_p)$ and $P(Y_{qi} \mid S_p, X_{qi})$, we can have a good chance to distinguish the different underlying mechanisms, S_p, $p = 1, 2, ..., N_s$.

Let's assume we know how to approximate $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$. The details will be covered by Section 4. To make the computation more convenient, usually we calculate *the average of the negative log likelihood* instead of $P(\boldsymbol{O}_q \mid S_p)$. The average of the negative log likelihood is defined as:

$$\begin{aligned}\overline{-\text{lik}(S_p)} &= -\frac{1}{N_q}\log(P(\boldsymbol{O}_q|S_p)) \\ &= -\frac{1}{N_q}\log\prod_{i=1}^{N_q} P(y_{qi}|S_p, X_{qi})P(X_{qi}|S_p) \\ &= -\frac{1}{N_q}\sum_{i=1}^{N_q}\log P(X_{qi}|S_p) - \frac{1}{N_q}\sum_{i=1}^{N_q}\log P(y_{qi}|S_p, X_{qi})\end{aligned} \tag{2-3}$$

Notice $\overline{-\text{lik}(S_p)}$ is a positive real number, because both $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$ are between *0* and *1*.

For the example in Figure 2-2, we were given a sequence of unlabeled observations of the driving behavior. The driver is unknown, but he must be one of five candidates: Tony, Larry, Curly, Moe and Groucho. Using all the 3,150 unlabeled data points, we calculated five averages of the negative log likelihood $\overline{-\text{lik}(S_p)}$, $p = 1, ..., 5$. Since OMEGA is an on-line approach, the 3,150 data points were not available at the early stage, we also study the $\overline{-\text{lik}(S_p)}$ with fewer data points. Therefore, we have five curves in the picture, the *X*-axis is the number of data points involved in the calculation of $\overline{-\text{lik}(S_p)}$, the *Y*-axis is $\overline{-\text{lik}(S_p)}$. At the very beginning, the values of $\overline{-\text{lik}(S_p)}$ are not reliable, because they were calculated using only a few data points; but with more and more data points involved, the $\overline{-\text{lik}(S_p)}$ curves become more consistent. The tails of the curves (to the right extreme) are the $\overline{-\text{lik}(S_p)}$ based on all the 3,150 data points. Among the five tails, the one closest to the horizontal axis indicates the generator of the observation sequences. In Figure 2-2, Groucho's tail is closest to the *X*-axis, thus Groucho seems to be the unknown driver.

2.2 Hypothesis Testing

Closely looking at the picture, $\overline{-\text{lik}(S_p)}$ of Groucho at the tail is 0.53, while that of Tony is about 1.40. Since 0.53 looks significantly smaller than 1.40, we claim that Groucho, not any other operator, seems to be the unknown driver.

However, we are not always lucky enough to be able to assign a unique candidate system to be the generator of $\boldsymbol{O}_q$. It is possible that more than one candidate's curves so close to each other that it is hard to tell which one is more likely to be the underlying generator. In Figure 2-3, Larry and Tony's tails are very close to each other. Larry's $\overline{-\text{lik}(S_p)}$ is 1.19, while Tony's is 1.21. Although Larry is a bit closer to the horizontal axis than Tony, we do not want to stake too much on Larry to be the only probable operator. Instead we say that the observation sequence $\boldsymbol{O}_q$ is confusing. It is important to distinguish the confusing situation from the exclusive one; because if the situation is confusing and we appoint a unique operator, we may end up with a severe mistake.

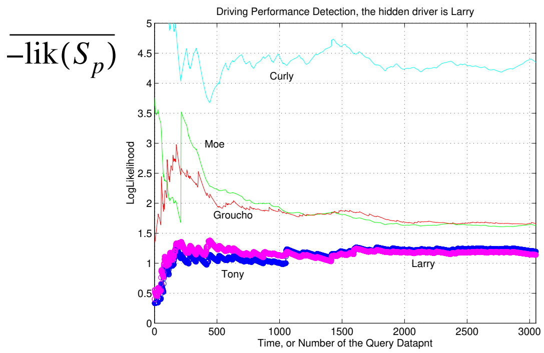


Figure 2-3: A confusing case. Since Larry and Tony's curves, especially their tails to the right extreme, are so close, that it is hard to tell which one is more likely**.**

To strictly define a confusing situation, we need a threshold. If the gap between the lowest tail and the second lowest one is beyond the threshold, the unique generator is easy to decide; otherwise, the situation is confusing. A difficulty arises in that there does not exist a fixed threshold applicable to any domain. For different domains, $\overline{-\mathrm{lik}(S_p)}$ are of differing scales, resulting in different thresholds. Therefore, we resort to the statistical *two sample hypothesis testing* method [Devore, 91]. For two candidate systems S_{p1} and S_{p2}:

1. We calculate the Z-test value from statistics,[1]

$$Z = \frac{(\overline{-\mathrm{lik}(S_{p1})}) - (\overline{-\mathrm{lik}(S_{p2})})}{\sqrt{\sigma_{p1}^2 / N_{p1} + \sigma_{p2}^2 / N_{p2}}}, \tag{2-4}$$

where $\sigma_{p1}{}^2$ and $\sigma_{p2}{}^2$ are the sample variance of $-lik(S_{p1})$ and $-lik(S_{p2})$ respectively, defined as,

$$\sigma_p^2 = \frac{1}{N_p} \sum_{i=1}^{N_p} \{[-\log P(X_{qi} | S_p) - \log P(y_{qi} | S_p, X_{qi})] - [\overline{-\mathrm{lik}(S_p)}]\}^2. \tag{2-5}$$

1. $P(y_{qi} \mid S_p, X_{qi})$ are independently identifically distributed (iid). Although theoretically $P(X_{qi} \mid S_p)$ is not iid, in practice, we roughly regard it as iid, and the hypothesis testing result is satisfactory.

N_{p1} and N_{p2} are the numbers of data points involved in the calculation of the likelihoods of system S_{p1} and S_{p2}. Sometimes N_{p1} and N_{p2} are equal. However, this is not a requirement. The bigger N_{p1} and/or N_{p2}, the larger the absolute value of the Z statistic tends to be.

2. The beauty of statistic Z is that its distribution is close to standard normal distribution if N_{p1} and N_{p2} are big enough, due to Central Limit Theorem. In this way, we can find a standard threshold for any domain and any observation sequence. We define this domain-independent threshold as Z_α. If $Z < -Z_\alpha$, S_{p1} has more potential than S_{p2} to be the generator of the unlabeled observation sequence $\boldsymbol{O}_q$. If $Z > Z_\alpha$, S_{p2} has more potential than S_{p1}. Otherwise, the observation sequence is confusing because S_{p1} and S_{p2} are closely likely to be its generator.

   The value of Z_α depends on the significance level α. Referring to Figure 2-4, the smaller the significance level α, the remoter the threshold Z_α deviates from zero, then it is harder for Z to be bigger than Z_α or smaller than $-Z_\alpha$, so that maybe no candidate system is found to be more competitive than all others to be the underlying generator of $\boldsymbol{O}_q$. Therefore, the smaller α is, the "pickier" we are.

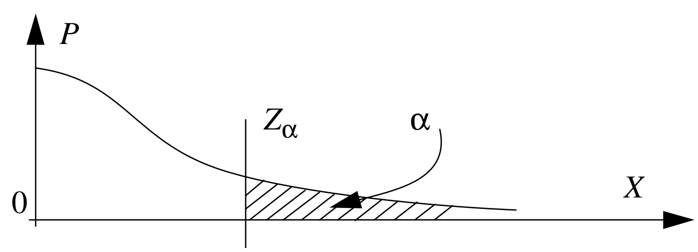


Figure 2-4: The physical meaning of Z_α.

   In practice, the significance level α is pre-defined by the user of OMEGA, and Z_α can be found by consulting the standard normal distribution table.

3. With more data points, the absolute value of the Z statistic tends to be bigger, and it is eas-

ier to distinguish the competitiveness of the various candidate systems. Therefore, in Figure 2-2 and 2-3, with more data points, the five curves become more separated. But the redundant data points do not help to distinguish $\overline{-\mathrm{lik}(S_p)}$, $p = 1, ..., 5,$ any further; hence, the curves become smooth and consistent afterwards.

2.3 Efficiency Issues

The efficiency of OMEGA is important for two reasons: (1) OMEGA is an on-line technique. (2) Because OMEGA calculates $\overline{-\mathrm{lik}(S_p)}$ for every possible candidate system, suppose there are one hundred candidate systems, S_1, S_2, ..., S_{100}, OMEGA will repeat the likelihood calculation for one hundred times to get $\overline{-\mathrm{lik}(S_1)}$, ..., $\overline{-\mathrm{lik}(S_{100})}$. When there are numerous candidate systems, the computational cost may be prohibitively high even if the task is off-line.

There are three ways to improve the efficiency,

1. Eliminate non-promising candidate systems from consideration:

Recall that the crucial steps of system classification are to calculate $\overline{-\mathrm{lik}(S_p)}$, then compare the $\overline{-\mathrm{lik}(S_p)}$ of the variant candidate systems to eliminate the non-promising candidates, and finally select the most likely one. The $\overline{-\mathrm{lik}(S_p)}$ is calculated according to the following equation:

$$\overline{-\mathrm{lik}(S_p)} = -\frac{1}{N_q}\sum_{i=1}^{N_q}\log P(X_{qi}|S_p) - \frac{1}{N_q}\sum_{i=1}^{N_q}\log P(y_{qi}|S_p, X_{qi})$$

In fact, it is unnecessary to consume all the N_q unlabeled observation data points to approximate $\overline{-\mathrm{lik}(S_p)}$. With fewer data points, even only a single data point, we can still do it. The problem is that with fewer data points, it is more difficult to distinguish a candidate from the others, referring to Section 2.2. But if some systems are far less promising than the others, even with a limited number of data points, its $\overline{-\mathrm{lik}(S_p)}$ value is significantly larger than the others', so that they can be neglected afterwards.

2. Speed up the calculation of the likelihoods:

Since $\overline{-\text{lik}(S_p)}$ is decided by $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$, a quick calculation or approximation for $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$ would improve the efficiency.

3. Focus on the promising candidates:

Even though we can eliminate unpromising candidate systems after a limited number of observations, at the early stage there may still be a large number of candidate systems involved in the processing. For example, if there are 10,000 candidate systems, perhaps after 100 observation data points, we can decide 9,999 candidates are irrelevant. Suppose that with fewer than 100 data points, no elimination can be performed and we have to calculate $\overline{-\text{lik}(S_p)}$ 10,000 times. To enhance the computational efficiency, it may be worthwhile to take a risk and focus on the more promising candidates from the beginning.

Compared with $P(y_{qi} \mid S_p, X_{qi})$, the computational cost of $P(X_{qi} \mid S_p)$ is much cheaper. Therefore, at the early stage with a limited number of (X_{qi}, y_{qi}), $i = 1, 2, \ldots,$ we can eliminate those candidate systems whose $P(X_{qi} \mid S_p)$'s are far lower than the others'. Of course this selection may make mistakes, but in case there are too many candidate systems, the risk is worthy of taking.

To implement the second and the third solutions, we need the *kd-tree* technique, which will be described in Chapter 5 and 6. A kd-tree re-organizes the memory of the training data points in a tree structure and caches some useful information in the nodes. A kd-tree is useful in two respects: (1) We can implement alternative memory-based learning methods with dramatically less cost. Thus we can greatly enhance the efficiency of calculating $P(y_{qi} \mid S_p, X_{qi})$. (2) When a specific query X_{qi} is given, we can quickly retrieve all its neighboring training data points, so as to approximate $P(X_{qi} \mid S_p)$ rapidly. Based on these two aspects, we can improve the efficiency of approximating $\overline{-\text{lik}(S_p)}$, as well as focus on the promising candidates from the beginning.

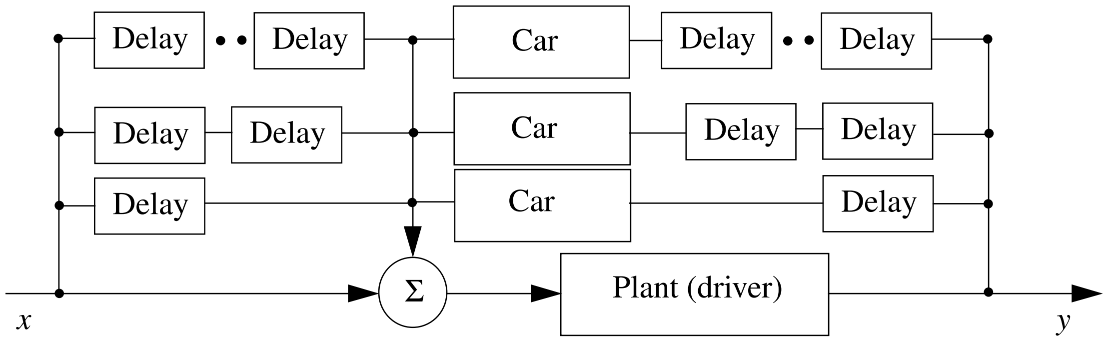


Figure 2-5: An one-input-one-output system with feedbacks and delays. The time order is important.

2.4 Pre-processing

In Subsection 2.1.2, we expand the input to include the delays and feedback so that the output at a certain moment is fully determined by the expanded input at that moment. More generally, for an one-input-one-output system illustrated in Figure 2-5, at time instant t, the output is simply y_t, while the input consists of x_t, x_{t-1}, ..., x_{t-p}, and y_{t-1}, ..., y_{t-q}. Thus, the input space dimensionality is $p + q + 1$.

If the time delays p and q are not known via prior knowledge; we have to figure them out based on empirical analysis of the observation data. Cross-validation, which is discussed in Chapter 7, is a useful technique to select the proper time delays.

It is straightforward to extend this method to transform time series with multiple input and/or output variables. It is not necessary for different input variables to have the same delay, nor likewise for the output variables. In the case where there are u input variables, whose time delays are p_1, ..., p_u, and there are v output variables with q_1, ..., q_v feedback variables, then the dimensionality of the transformed data point's input is $p_1 + ... + p_u + u + q_1 + ... + q_v$.

The transformation of the time series data is not always necessary. Imagining a set of data points { (x1, y1), (x2, y2), ..., (xT, yT) } are generated by a system which has no time delays and feedback, the output yt is fully determined by xt, and xt is independent from the previous ones, xt-1, xt-2, ... In this case, although the data points are collected as time passes, the order of time is not important and we can shuffle the data points randomly.

However, a high dimensionality of the data points is always a concern. Especially those transformed time series data points with expanded input tend to have a dimensionality which is prohibitively high for the further OMEGA steps. This motivates the pre-processing: decreasing the dimensionality of the input space.

Other alternatives may exist, but we choose two approaches: feature selection and Principal Component Analysis (PCA).

Feature selection

In the driving domain, many variables affect our driving performance. While the distance between our vehicle to the vehicle immediately in front of us is probably important, the distance from our vehicle to that one behind us may not be very important in most cases. Therefore, we should consider eliminating the latter distance from the input vector.

To perform feature selection, we follow *cross-validation* approach again. The biggest concern of cross-validation is the computational cost. Therefore, in Chapter 7, we explore ways to improve its efficiency. Feature selection may not be very crucial in the driving domain due to the large amount of prior knowledge. Feature selection is an important component of OMEGA, as a general purpose toolkit.

Principal component analysis

In the driving domain, although we have eliminated those irrelevant input variables based on prior knowledge, the input dimension of the transformed data points may still be as high as 50, (referring to Chapter 8 and Chapter 9). To reduce the dimensionality, we resort to Principal Component Analysis (PCA) [Jolliffe, 86].

Each data point consists of two parts: input and output. Assume the input vector X is d-dimensional. Without loss of generality, X can be represented as a linear combination of a set of d orthonormal vectors $\boldsymbol{U}_k$,

$$X = \sum_{k=1}^{d} z_k \boldsymbol{U}_k$$

With fixed orthonormal vectors $\boldsymbol{U}_k$, $k = 1, ..., d$, different data points' inputs have differing coefficients z_k, $k = 1 ,..., d$. If we carefully choose $\boldsymbol{U}_k$, it is sometimes possible that the first M coefficients contains the most information, i.e.

$$X = \sum_{k=1}^{d} z_k \boldsymbol{U}_k = \sum_{k=1}^{M} z_k \boldsymbol{U}_k + \sum_{k=M+1}^{d} z_k \boldsymbol{U}_k \approx \sum_{k=1}^{M} z_k \boldsymbol{U}_k$$

If so, we can shrink the dimensionality of X from d down to M. Notice that only when all the data points satisfy the above equation, is PCA useful for compressing the dimensionality, as illustrated in Figure 2-6 (a). In the two cases illustrated in Figure 2-6 (b) and (c), PCA does not help.

In one of our experiments, PCA compressed the input dimensionality of the independent data points from 50 dimensions to 3; and in another case, it helped to reduce from 36 dimensions to 8.[1]

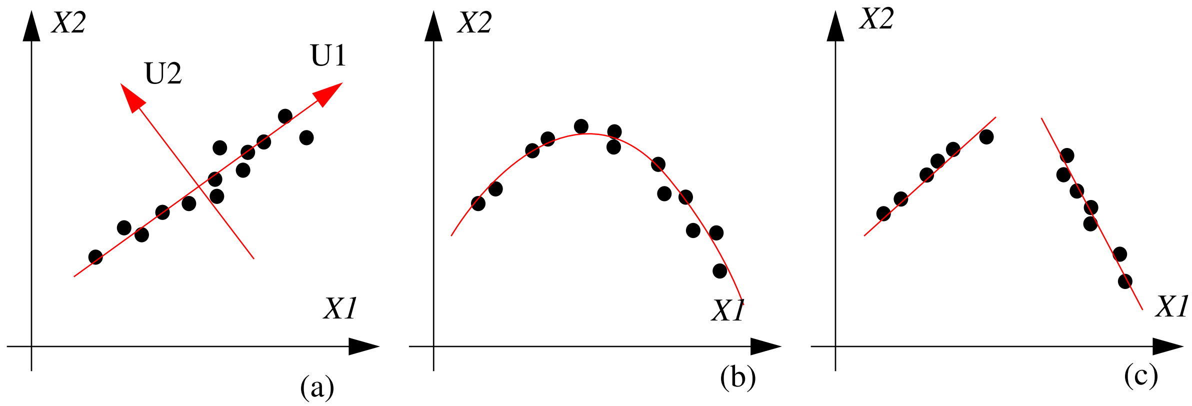


Figure 2-6: PCA can be used to compress the dimensionality of a set of data points. In (a), after the transformation of the coordinates, the information along U_2 axis is no more significant, so that the dimension is reduced from two to one. However, PCA may not be useful for all cases. Although there obviously exist submanifolds in (b) and (c), the conventional PCA does not help to reduce the dimensionality.

2.5 Memory-based learning

In this section, we discuss how to use memory-based learning methods to approximate $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$. To do so, we need some knowledge of system S_p. Memory-based learning methods assume that the knowledge about a system S_p comes from a memory which consists of the observation data points of this system's previous behavior, $\{(x_{p1}, y_{p1}), (x_{p2}, y_{p2}), ..., \}$. Again, these data points have been pre-processed so that temporal order is no longer important.

When there are n candidate systems, we will have at least n sets of observation data points. The memory contains all of them together. To distinguish the data points generated by different systems, each data is labeled by its generator. Suppose the p'th system has N_p memory data points and there are n candidate systems, the size of memory will be $N_1 + N_2 + ... + N_n$.

1. In first case, the loss of information is 14%. The second case loses 17%.

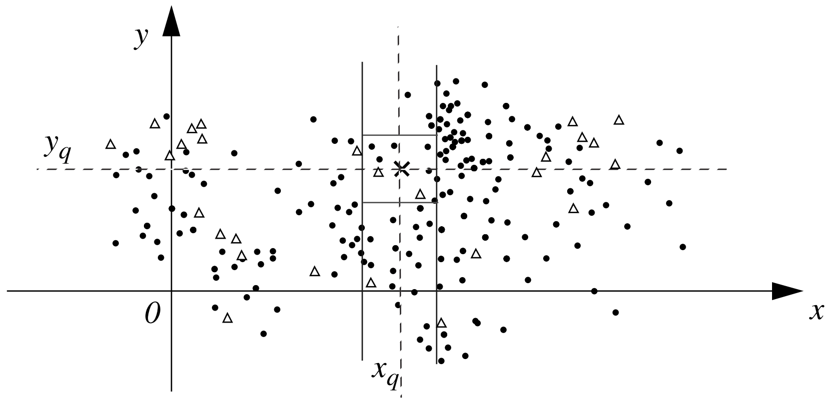


Figure 2-7: Memory-based learning methods to approximate $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$

A naive method

In Figure 2-7 the *X*-axis is the input of a system, the *Y*-axis is the output. Each dot represents a data point of system S_p. There should exist data points generated by other systems in the memory, too. For example, the triangles are the data points of another system. The cross represents the unlabeled data point (x_q, y_q), which is a component of the observation sequence $\boldsymbol{O}_q$ whose underlying generator is unknown.

To approximate $P(x_q \mid S_p)$, we can simply count the number of the memory data points of S_p (the dots) in the stripe shown in Figure 2-7. The stripe defines the neighboring region of x_q. It is a big concern to decide the boundaries of the stripe, but let's temporarily assume that the boundaries can be easily decided. Suppose the number of dots in the stripe is N_q ($N_q = 27$ in this case), while the total number of dots in the whole memory space is N_p, then $P(x_q \mid S_p)$ can be approximated as N_q / N_p.

To approximate $P(y_q \mid S_p, x_q)$, we can simply count the number of dots in the square around the unlabeled data point (x_q, y_q); in this case, the number is 6. $P(y_q \mid S_p, x_q)$ can be approximated as the ratio of *6* to N_q, the number of dots in the stripe.

There is one question here: why do not we simply approximate $P((x_q, y_q) \mid S_p)$, instead of approximating two probabilities $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$? In fact, $P((x_q, y_q) \mid S_p)$ can be approximated as the ratio of the number of dots in the square to the total number of dots in the whole memory space; in this case, the ratio is $6 / N_p$.

Recall Equation 2-2 and 2-3, which are

$$P(\boldsymbol{O}_q|S_p) \approx \prod_{i=1}^{N_q} P(y_{qi}|S_p, X_{qi})P(X_{qi}|S_p)$$

and

$$\overline{-\text{lik}(S_p)} = -\frac{1}{N_q}\sum_{i=1}^{N_q} \log P(x_{qi}|S_p) - \frac{1}{N_q}\sum_{i=1}^{N_q} \log P(y_{qi}|S_p, x_{qi}).$$

There are three advantages of decomposing $P((x_q, y_q) \mid S_p)$ into $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$. First of all, we can try any machine learning methods to approximate $P(y_q \mid S_p, x_q)$, for example neural networks and Bayes networks. Hence, $P(y_q \mid S_p, x_q)$ is a socket for alternative methods to plug in. Second, the approximation of $P((x_q, y_q) \mid S_p)$ is an interpolation problem, but the approximation of $P(y_q \mid S_p, x_q)$ can be extrapolation as well. Finally, the probability $P(y_q \mid S_p, x_q)$ is about the function relationship between the system input and output. If we have some domain knowledge of the system S_p, we can use them to improve the approximation of $P(y_q \mid S_p, x_q)$.

The goodness of the naive method is its simplicity. However, it is difficult to define the boundaries of the stripe and the square. If the stripe is too narrow and the square is too small, the approximation of the probabilities will be too sensitive to the noise of the limited number of the memory data points in the stripe and the square. Otherwise, if the stripe is too wide and the square is too big, it is hard to tell the difference between $P(x_q \mid S_p)$ and $P((x_q+ \delta) \mid S_p)$, as well as the difference between $P(y_q \mid S_p, x_q)$ and $P((y_q+\xi) \mid S_p, (x_q+\delta))$. Besides, the inconsistency of the distribution of the memory data points brings more troubles. In the case of Figure 2-7, if we expand the stripe to be wider, the value of $P(y_q \mid S_p, x_q)$ will chance greatly. In fact, it will become larger, because there are numerous memory data points residing just outside the boundaries.

Therefore, we consider Kernel density estimation, because it does not require any boundaries.

Kernel density estimation

Kernel density estimation does not neglect any data points in memory, so that every memory data point is involved in the approximation of $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$. However, higher weights are assigned to those memory data points neighboring to the unlabeled data point (x_q, y_q), so that the neighboring memory data points have stronger impact on the approximation of $P(x_q \mid S_p)$ and $P(y_q \mid S_p, x_q)$. Conversely, remote memory data points have smaller weights, therefore any single remote data points hardly has any influence on the approximation, but if many remote memory data points express the same preference, the approximation will be biased in their favor.

Using Kernel density estimation, $P(x_q \mid S_p)$ can be approximated as,

$$P(x_q | S_p) = \sum_{i=1}^{N_p} w(x_i, x_q) / N_q \tag{2-6}$$

in which N_p is the total number of data points in memory generated by S_p. w_i is the weight associated with the i'th one among them, usually defined as a Gaussian function of the Euclidean distance from x_q to the concerned memory data point,

$$w(x_i, x_q) = \text{Const} \times \exp\left(-\frac{\|x_i - x_q\|^2}{2{K_w}^2}\right). \tag{2-7}$$

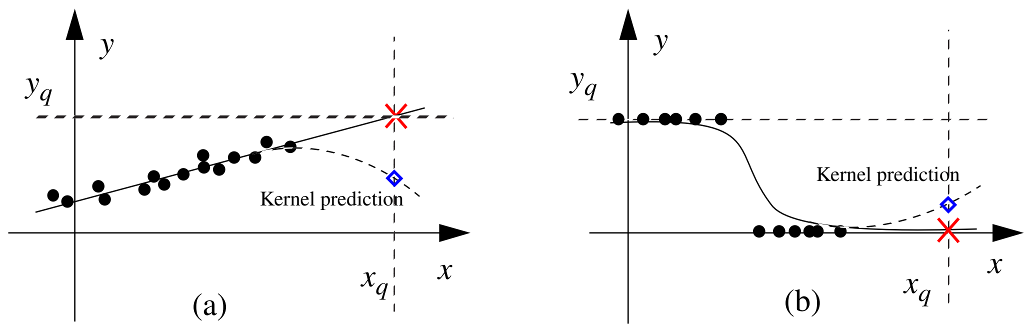


Figure 2-8: Kernel regression does not extrapolate.

Therefore, with respect to different x_q's, the weights associated with an identical memory data point may be different. The higher the Euclidean distance $||x_i - x_q||$, the smaller the weight. K_w is the *kernel width*. The higher the kernel width, the less the weights change with respect to different distances. There are many other possible definitions of the weight [Atkeson et al., 97].

$P(y_q \mid S_p, x_q)$ can be approximated as,

$$P(y_q|S_p, x_q) = \left(\sum_{i=1}^{N_p} w(x_i, x_q)v(y_i, y_q)\right) \Big/ \left(\sum_{i=1}^{N_q} w(x_i, x_q)\right). \tag{2-8}$$

$v(y_i, y_q)$ is also a weighting function but with respect to the Euclidean distance of $|| y_i - y_q||$. If y's value is continuous, it is fine to define $v(y_i, y_q)$ as a Gaussian function in a way similar to Equation 2-7. However, when y is discrete or categorical, we should be more careful. For example, when y is boolean, the weighting function $v(y_i, y_q)$ can be defined as,

$$v(y_i, y_q) = \begin{cases} 1 & \text{When } y_i = y_q \\ 0 & \text{Otherwise} \end{cases}.$$

Kernel density estimation is useful in many cases, its drawback is that it is only good for interpolation, it does not extrapolate. This is not desirable for the approximation of $P(y_q \mid S_p, x_q)$.

Referring to Figure 2-8(a), suppose we want to approximate $P(y_q \mid S_p, x_q)$, while (x_q, y_q) locates at the position of the cross, intuitively it should be fairly large because it is on the "trend" of the memory data points. However, Kernel density estimation's results will be smaller than they should be. Kernel density estimation does not extrapolate in both continuous case and categorical one. Figure 2-8(b) shows the similar problem in a categorical case.

Locally weighted linear and logistic regressions

Locally weighted linear regression is applicable for both interpolation and extrapolation. Although in many cases, the relationship between the input and the output is more complicated than linear, in any local region, sometimes the relationship can still be approximated as a linear one, illustrated in Figure 2-9. Locally weighted linear regression is a popular memory-based learning method. But it works only when the output y is continuous.

The counterpart of locally weighted linear regression for cases when the output y is discrete or categorical is locally weighted *logistic* regression. Logistic regression has been explored by the statistical community since 1970's. We improve this technique by following a locally weighted paradigm, so that in the toolkit of memory-based learning method, we have a more reliable classifier.

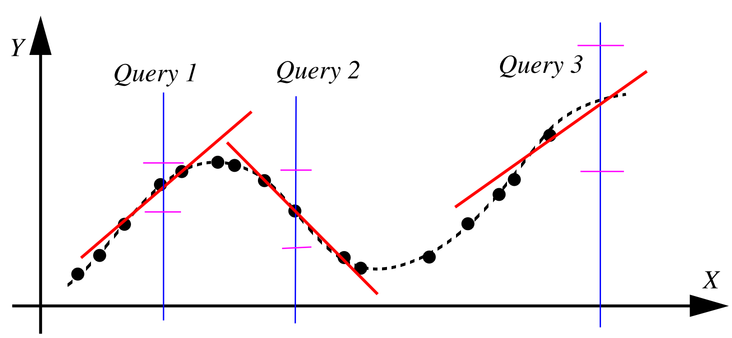


Figure 2-9: Locally weighted linear regression can approximate non-linear functional relationship. It works for both interpolation and extrapolation. The pairs of horizontal bars indicate the variance.

Similar to the principle of locally weighted linear regression, locally weighted *logistic* regression assumes the relationship between the input and output in any local region can be approximated in a form of a simple function. But unlike locally weighted linear regression, which assumes the local relationship is *linear*, locally weighted logistic regression approximates the local relationship in the form of a *logistic* function of a linear combination of inputs. Logistic functions are also referred to as *sigmoid* functions, which are monotonic continuous functions between zero and one. The details will be discussed in Chapter 4.

Approximate $P(y_q \mid S_p, x_q)$ using regression methods

Kernel regression is good enough to approximate $P(x_q \mid S_p)$. In this subsection, we focus on how to use the regression methods to approximate $P(y_q \mid S_p, x_q)$. We discuss this issue in three cases according to the different distribution types of y_q.

1. Suppose the conditional distribution of y_q given a specific x_q is Gaussian, i.e.,

$$P(y_q | S_p, x_q) = \frac{1}{\sqrt{2\pi}\sigma_q} \exp\left(-\frac{(y_q - E(y_q | S_p, x_q))^2}{2\sigma_q^2}\right),$$

   in which $E(y_q \mid S_p, x_q)$ can be predicted using locally weighted linear regression technique, the variance σ_q^2 can be approximated as,

$$\sigma_q^2 = \text{Var}\,(y_q | S_p, x_q) = E(y_q^2 | S_p, x_q) - E^2(y_q | S_p, x_q).$$

   When the conditional distribution of y_q is continuous and *uni-modal*, we will always treat it as a Gaussian distribution. Therefore, the above method is applicable for many cases.

2. When output y_q is discrete or categorical, we can approximate $P(y_q \mid S_p, x_q)$ using locally weighted logistic regression.

3. When output y_q is continuous, but with multiple modes, there are two approaches. First, we can use the techniques like [King et al., 96] to perform the distribution approximation. But this approach still relies on some prior knowledge of the distribution. Second, as a general purpose approach, we can discretizes the output so as to employ the logistic regression approach described in last paragraph.

   For example, suppose the output y_q is continuous within [0, 10). Regardless of whether y's distribution is uni-modal or multi-modal, we discretize it into five equal-sized bins; so that when y_q's value is between [0, 2), we transform it into a categorical value, $(1, 0, 0, 0, 0)^T$. While y is between [2, 4), the corresponding categorical value is $(0, 1, 0, 0, 0)^T$. Now we can use locally weighted logistic regression to approximate $P(y_q \mid S_p, x_q)$.

   However, the discretization approach has two problems. First, in the example above, $P(y_q = 2.5 \mid S_p, x_q)$ and $P(y_q = 3.5 \mid S_p, x_q)$ will be identical, because $y_q = 2.5$ and $y_q = 3.5$ are in the same bin. Therefore, the variance of $P(y_q \mid S_p, x_q)$ increases with fewer bins.

   Second, increasing the discretization resolution causes increased loss of information. For example, as categorical values, both (0, 1, 0, 0, 0) and (0, 0, 1, 0, 0) are differing from (1, 0, 0, 0, 0), but one cannot tell that (0, 1, 0, 0, 0) is closer to (1, 0, 0, 0, 0) than (0, 0, 1, 0, 0). Thus, we retain the information that $P(y_q = 1.0 \mid S_p, x_q)$ and $P(y_q = 3.9 \mid S_p, x_q)$ are both different from $P(y_q = 4.0 \mid S_p, x_q)$, but lose the information that $P(y_q = 3.9 \mid S_p, x_q)$ and $P(y_q = 4.0 \mid S_p, x_q)$ are closely related to each other.

Overall, we still suggest the discretizing method as a general purpose approach. In our experiments in Chapters 3, 8 and 9, we discretized the outputs into 8 or 10 categories, and found the results to be satisfactory.

2.6 Summary

In this chapter, we introduce the main steps for system classification: pre-processing, prediction, likelihood calculation, and hypothesis testing. In addition, we discuss three ways to improve the efficiency.

This chapter is the framework of OMEGA technique, although we mention other relevant topics, i.e. feature selection, logistic regression-based classifier and kd-tree technique. We will discuss these topics in depth in later chapters.

The next chapter discusses an experiment, demonstrating the usefulness of OMEGA system. More complicated experiments will be discussed in Chapter 8 and 9, after we have finished the discussion on feature selection, logistic regression, and kd-tree.

Chapter 3

Tennis Style Detection

3.1 Experimental Design

In this experiment, we designed a simple simulator of tennis, to study different people's playing styles. The ball is served automatically from a random position in the upper half field with a random speed within a certain range and a random direction towards the bottom line. A human player can control the racket by moving the mouse. The speed of the racket is proportional to the speed of the mouse, and its orientation is perpendicular to the recent trajectory of the mouse.

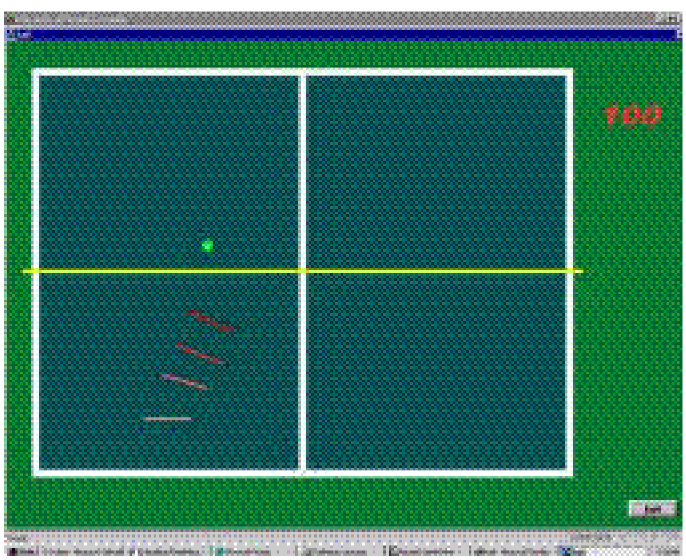


Figure 3-1: Tennis simulator interface.

Input: serve position (x_s, y_s), serve speed (v_s) and orientation (θ_s).
Output: contact position (x_r, y_r), the ball's speed and orientation after the contact, (v_r, θ_r).

Figure 3-2: The tennis simulation system is not dynamic because there is no feedback from the outputs and no delays for the inputs.

The short line segments in Figure 3-1 illustrate the recent movement of the racket. When the racket hits the ball, the ball is bounced back as a light beam is reflected by a mirror. Thus, the direction of the ball after contact is decided by both the orientation of the racket and the incident direction of the ball. Concerning the ball's emitted speed, it is decided by three factors: speed of the racket, the incident speed of the ball and the ball's incident angle with respect to the orientation of the racket.

This simulation system is not dynamic. Referring to Figure 3-2, if we regard the human player as a system, the input consists of four variables: the position where the ball is served by the computer, (x_s, y_s), the ball's speed (v_s) and orientation (θ_s) after the serve. The output includes the position where the contact between the racket and the ball happens, (x_r, y_r), the speed and orientation of the ball after the contact, (v_r and θ_r). We only took records of those shots when the racket hits the ball. If the player was so careless that the racket missed the ball, we did not record that shot. We did not consider the ball's movement after the contact, because we were only interested in distinguishing the different playing styles, instead of evaluating the goodness and drawback of each style. Illustrated in Figure 3-2, there is no time delay in the input, and there is no feedback from the outputs, hence the system is not dynamic. In other words, the time order of the sequence of the data points, $(x_s, y_s, v_s, \theta_s, x_r, y_r, v_r, \theta_r)_t$, $t = 1, ..., T$, is not important; we can shuffle the order of the data points randomly.

Six people were invited to do the experiment. Each of them played twenty runs; and during each run, they gave one hundred shots. We did not use the data sets of the first three runs because the human players needed some time to learn how to play this game. We did not use the data set of the twentieth run, because when the players realized that they were close to the end, they did not pay enough attention to their performance, instead, they only wanted to finish the experiments as soon as possible. Thus, for each player, we have sixteen valid data sets.

We did not evaluate the merit of the performance, we only want to distinguish the different styles. However, it is an interesting but open question that if we evaluate the performance, whether or not people will adjust their styles so as to pursue higher scores; also, after a long time, whether or not different people will converge to the same style which is preferred by the evaluator.

The style is relevant to the distribution of the eight variables. Some people tended to hit the ball when the ball was close to the bottom line; the others gave a quick response once the ball came across the net. Some people wanted the ball to go in a direction as far as possible from the serving direction; others preferred the ball going back along the way it came, because this action is safer and easier. However, we cannot distinguish the styles only relying on the distribution of any one variable, because it is influenced by the other variables. As a matter of fact, we found that the speed of racket, v_r, was the best single feature to distinguish different players. But comparing with OMEGA, the single-feature-based classifier's accuracy is very low (Section 3.3).

Since there are six players, and each player has sixteen data sets, totally there are ninety-six data sets. Randomly we picked out one from the ninety-six datasets, and asked OMEGA to detect who was the underlying player by using the other ninety-five datasets as the training datasets. By comparing OMEGA's result with the real underlying player, we could tell for this data set, whether or not OMEGA's detection is correct. Similarly, we selected another data set to do this test, thus, we repeated the experiment for ninety-six times. The number of times that OMEGA succeeded to detect the correct underlying players can be used as a measurement of

OMEGA's accuracy. In the same way, we can measure the accuracy of the other methods, like the single-feature-based classifier.

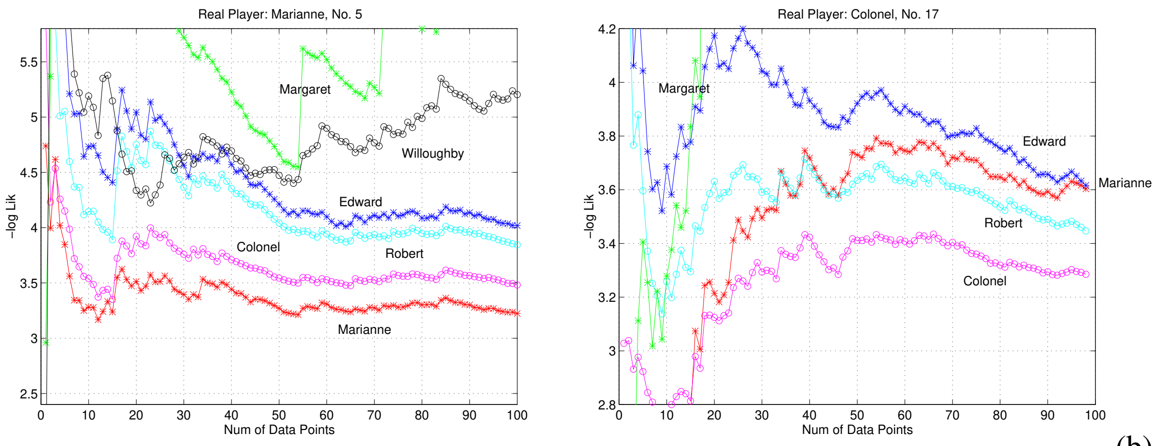
3.2 OMEGA Result Analysis

This subsection discussed the experiment, which was to test if OMEGA could detect the underlying player correctly. We picked out one data set from each player's sixteen data sets as the testing set, and used the other fifteen data sets as the memory data sets. To detect who was the underlying player of the testing data set, OMEGA compared the testing data set with the six players' memory data sets one by one. Hence, we got six average negative log likelihoods, $\overline{-\mathrm{lik}(S_p)}$'s. In Figure 3-3, 3-4, 3-5, the six curves correspond to the six players' $\overline{-\mathrm{lik}(S_p)}$'s with respect to different numbers of data points involved in the calculation. The horizontal axis is the number of data points in the unlabeled data set. Thus, the tails of the $\overline{-\mathrm{lik}(S_p)}$ curves tell who were most likely to be the underlying players.

Shown in Figure 3-3 (a) and (b), OMEGA detected Marianne and Colonel were the underlying players of the concerned data sets. These results are correct. For the ninety-six data sets, OMEGA did correct jobs for eighty-five times. It made mistakes for four times and was confused for seven times[1]. Figure 3-4 (a) shows a confused case, while Figure 3-4 (b) is a wrong one. Even in the wrong cases and the confused ones, OMEGA always found that the tails of the real players' likelihood curves were closer to the horizontal axis than most of the others.

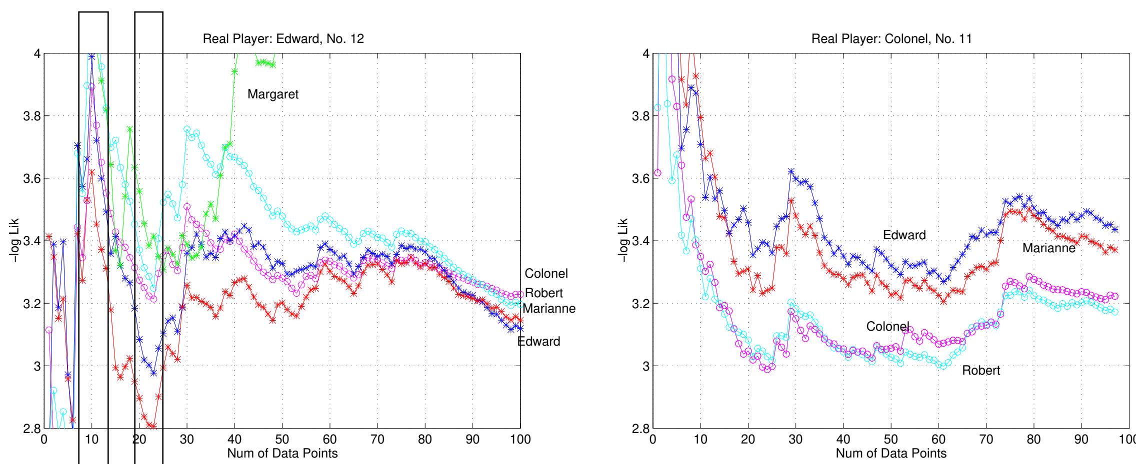
Sometimes the likelihood curves are bumpy. This is because the player performed in an unusual way that hasn't been observed in memory. If a performance was so strange that it rarely happened to all the players, including the underlying player himself, then all the likelihood curves are bumpy, and roughly paralleling each other. In the case illustrated by Figure 3-4 (a), the ninth ball was served from a position very close to the right edge and also close to the net, with a sharp angle towards the left edge of the opposite field. Although the speed was not too fast, it

1. The definition of confusion refers to Chapter 2.2., Hypothesis testing, with significance level $\alpha = 5\%$.



(a) (b)

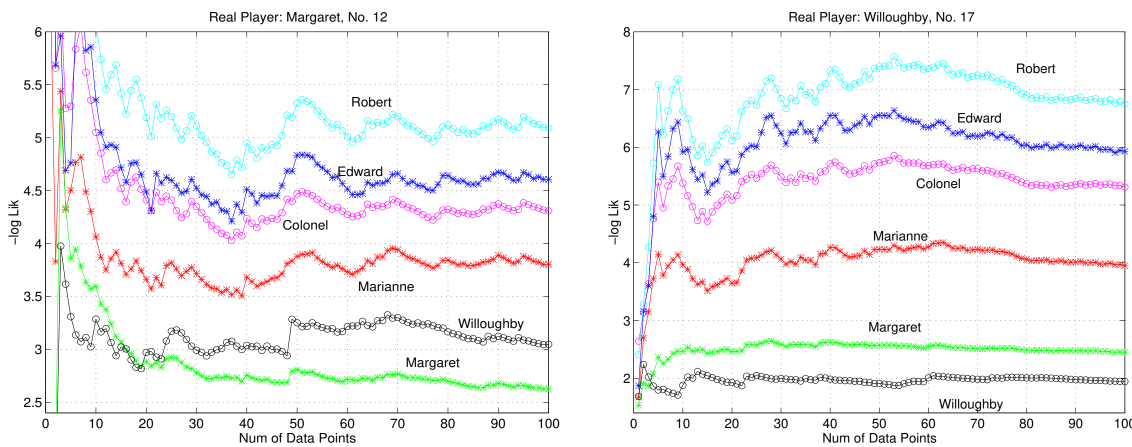
Figure 3-3: Likelihood curves of six human players. Two sample of the correct cases.



(a) (b)

Figure 3-4: A confused case and a wrong case. **(a)** Confused: OMEGA can hardly distinguish Edward from Marianne. **(b)** Wrong: The real play should be Colonel, but OMEGA decided it was Robert. However, OMEGA did figure out Colonel was also very likely to be the player.

left Edward little time to react. Because Edward is right-handed, any ball coming from the right made him uncomfortable. Therefore, Edward's action for the ninth hit was totally a failure: the ball did not go across the net before it went out of the tennis court. Not only that, it seemed Edward did not recover from this shock until the twelfth hit. In the eleventh hit, he hardly touched the ball, because the ball's direction did not change too much after the contact. Hence, the likelihood curves in Figure 3-4 (a) rose to a peak at the eleventh hit. Fortunately, the twelfth



(a) (b)

Figure 3-5: Willoughby and Margaret behaved similarly all the time. But they are different from others. Willoughby played more consistently, referring to his likelihood curve in (b) which is smoother than others.

ball was served in a manner Edward preferred: from the top left corner of the court towards the lower-right one, with a slow speed. This gave Edward a break to rebuild his confidence. He played in normal way again. Therefore, the likelihood curves start to go downhill. The twenty-second ball was another triumph. It started not far from center of the upper field, slowly and straightly downward. This was a great chance for Edward to exaggerate all his unique characteristics: he moved his racket rapidly to hit the ball when it arrived the center of the lower half field; after the contact, the ball rushed towards to the top right corner. Therefore, in Figure 3-4 (a), we see a great peak around the eleventh data point and a deep valley at the twenty-second.

The bumpiness implies the consistency of the players. Willoughby was the most consistent players among the six, because comparing Figure 3-5 (b) with other figures, Willoughby's curves are smoother than the others'.

The distances among the likelihood curves imply whose performances are similar. In this experiment, Margaret and Willoughby behaved similarly, referring to Figure 3-5 (a) and (b). But they are quite different from the others. As in Figure 3-3, 3-4, their curves were so much higher than the others that they are off the graphs.

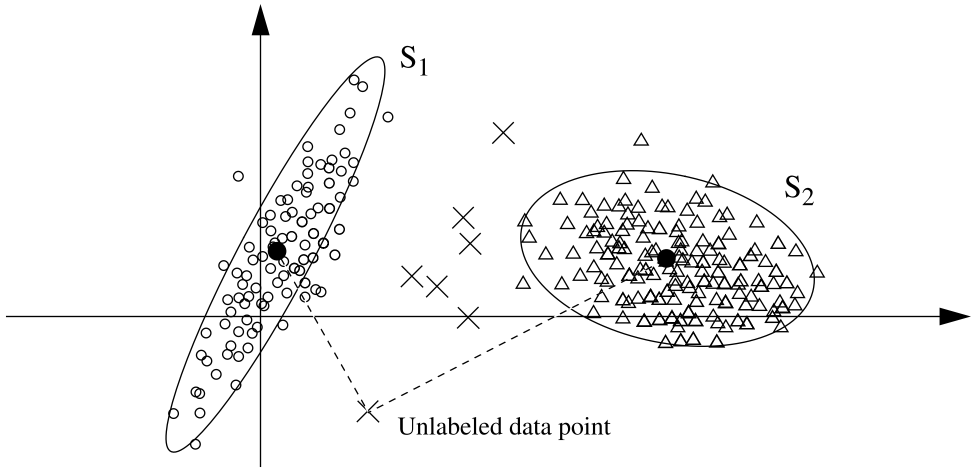


Figure 3-6: Bayes classifier assumes the distributions of the candidate system's data points are all of Gaussian distributions.

The likelihood curves tend to be more bumpy or chaotic at the beginning phase than afterwards. Recall that with limited testing data points, OMEGA is still able to start the classification job; but with more data points, OMEGA may improve its precision. Therefore, OMEGA is an ideal on-line classification technique.

3.3 Comparison with Other Methods

In this section we compare OMEGA's performance with those of other methods, like Bayes classifier and linear regression, because Bayes classifier is a popular statistical classifier while linear regression represents the linear control system approach. We also used the best single feature to do the classification. The purpose was to show that it is not easy to distinguish different tennis playing styles.

Bayes classifier

Bayes classifier assumes the memory data points of each candidate system are of Gaussian distribution, in plain words, each candidate system's memory data points cluster in a shape more or less like an ellipse. In Figure 3-6, there are two candidate systems, S_1 and S_2, whose data points are represented by the circles and the triangles respectively. The horizontal axis may be the input, the vertical one may be the output; but this is not a requirement. As a matter of fact, Bayes classifier does not distinguish the input and output, instead, it treats all the input and output as features. By adjusting the scales of the axes, Bayes classifier can discriminate the importance of different features. In Figure 3-6, if the scales of the axes are changed, the elliptical shape of the clusters will be different. To classify an unlabeled data point, like the cross in Figure 3-6, we can measure the distances from the unlabeled data point to the centroids of the elliptical clusters. The shortest distance indicates to which candidate system (represented by the ellipse) the unlabeled data point belong to. Given a set of unlabeled data points, we can do the classification one by one, then make an overall judgement.

The Gaussian assumption of Bayes classifier is too restrictive for the tennis style domain. Therefore, Bayes classifier's performance as shown in Table 3-1 is very poor compared with OMEGA.

Linear regression approach

Linear regression assumes the function relationship between the inputs and the outputs are linear. Furthermore, *global* linear regression assumes the function relationship (the parameters of the function) is fixed anywhere around the input space. If the function parameters of a certain system is distinguishable from the others, the classification job is feasible. In this experiment, we did the global linear regression of each candidate system based on its memory data points. In other words, we determined the parameters, β's, in the following linear equations for every candidate system,

$$\begin{cases} x_r = \beta_{10} + \beta_{11}x_s + \beta_{12}y_s + \beta_{13}v_s + \beta_{14}\theta_s + \xi_1 \\ y_r = \beta_{20} + \beta_{21}x_s + \beta_{22}y_s + \beta_{23}v_s + \beta_{24}\theta_s + \xi_2 \\ v_r = \beta_{30} + \beta_{31}x_s + \beta_{32}y_s + \beta_{33}v_s + \beta_{34}\theta_s + \xi_3 \\ \theta_r = \beta_{40} + \beta_{41}x_s + \beta_{42}y_s + \beta_{43}v_s + \beta_{44}\theta_s + \xi_4 \end{cases} \tag{3-1}$$

in which the definitions of the input variables, x_s, y_s, v_s, θ_s, and the output variables x_r, y_r, v_r, θ_r refer to Section 3.1. When a unlabeled data set came, to detect its underlying player, we temporarily assume the unlabeled set was generated by the first player. Since we have already estimated the first player's function parameters (the β's), we picked out a data point $(x_s, y_s, v_s, \theta_s, x_r, y_r, v_r, \theta_r)_t$ from the unlabeled data set, we could predict the outputs $(x_r, y_r, v_r, \theta_r)_t$ corresponding to the input $(x_s, y_s, v_s, \theta_s)_t$. If the residual between the predicted outputs and "real" observed output is small, the first player is likely to be the underlying player. We repeated this test with respect to all the six players, the smallest residual responds to the most likely player.

We used the estimated β's to predict the outputs, then compare the predicted outputs with the real outputs. Usually there is a residual between the predictions and the real outputs. The system with the least residuals is most likely to be the underlying system which generates the testing dataset.

Referring to Table 3-1, global linear regression can hardly distinguish the variant human players, because in most cases, global linear regression is "confused". To improve it, we can do two things: (1) We can extend the linear equations in Equation 3-1 to polynomials with higher degrees. In this way, the function is capable of describing more complicated relationship between the input and output. (2) Instead of assuming there is one fixed global linear function, we can assume in any local region, the input and output relationship is linear, but the linear parameters may vary with different inputs.

In Table 3-1, we notice that quadratic model does not do any better than the linear models, but local paradigm does help. However, the local approach, even the local models with quadratic items, is still worse than OMEGA by a large margin. The reason is that in this tennis playing style domain, even for the identical serves, the same player may react in different ways. That means, the conditional distribution of the output with respect to a certain input may be of multi-modal, instead of uni-modal as the linear model assumes. Therefore, the linear models are not proper for the tennis playing style domain, either.

Table 3-1: Comparison experiment for tennis domain

	Correct	*Wrong*	*Confused*
One Feature	21	57	18
Bayes	34	40	22
k-Nearest Neighbors[a]	17	14	67
Global Linear	9	12	75
Global Quadratic	9	12	75
Local Linear	17	8	71
Local Quadratic	20	5	71
OMEGA	85	4	7

a. We used 9 nearest neighbors here. Also, we tried 3 nearest neighbors as well as 6, the results do not deviate from those values in the table significantly.

3.4 Summary

In this chapter, we used OMEGA to classify different human operators' behavior in a game mimicking tennis. Although the simulation system is not dynamic, the classification job is not easy, especially because the distribution of the input and output is complicated. OMEGA performs very well in this domain, which demonstrates that OMEGA is a good classification technique. Although originally it was explored to classify time series, OMEGA is also a general purpose classification tool, which is capable of handling both time series and non-time series.

Experiments have been done to compare OMEGA with other methods. Although we have tuned up those methods to perform as well as possible, they still are not competitive with OMEGA.

Chapter 4

Logistic Regression as a Classifier

In this chapter, we discuss how to approximate the probability $P(y_q \mid S_p, x_q)$, i.e., the probability that if the underlying system is S_p, corresponding to a certain input x_q, the system's output is y_q. We explore a new memory-based method, *locally weighted logistic regression*, which aims at approximating $P(y_q \mid S_p, x_q)$ when the output y_q is categorical.

Figure 4-1 illustrates the task of this chapter. Suppose there is a system, S_p, whose input space is 2-dimensional, and the output is boolean. Suppose a unlabeled data point is (x_q, y_q), to approximate $P(y_q \mid S_p, x_q)$, we need some knowledge of system S_p. Memory-based methods assume that the knowledge comes from the previous observations of the system's behavior, i.e. the memory data points or the training data points, as the circles and crosses in Figure 4-1. The circles correspond to those memory data points of S_p with outputs equal to 0, the crosses correspond to the other memory data points with outputs equal to 1. Now, if there come two queries, residing at the positions of the dark triangles, if both of the queries' outputs are "cross", then intuitively $P((y_q = \text{"cross"}) \mid S_p, x_q = (2.0, 3.0)^T)$ should be close to 1.0 because the majority of its neighbors are crosses, while $P((y_q = \text{"cross"}) \mid S_p, x_q = (4.5, 1.0)^T)$ should be near 0.0, based on the similar reasoning.

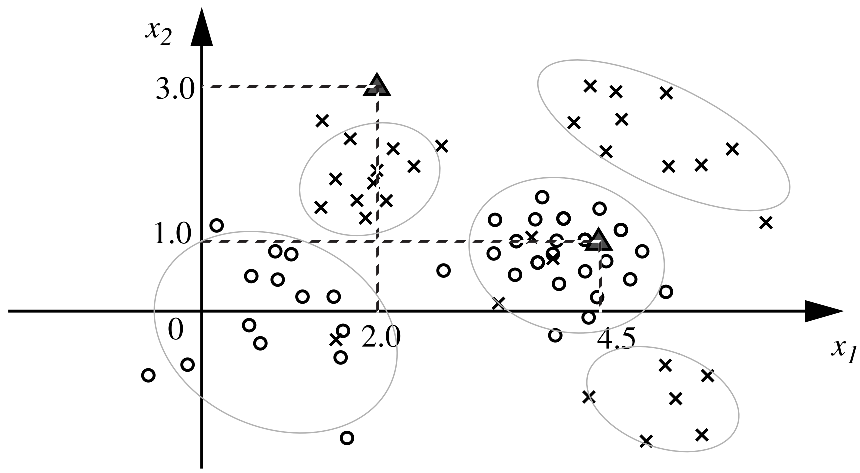


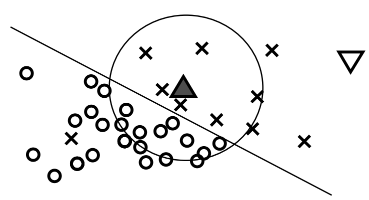
Figure 4-1: An illustration of the classification task.

4.1 Classification methods

Since the output y is categorical, the approximation of $P(y_q \mid S_p, x_q)$ is a classification problem by itself. System classification is to summarize a sequence of such classifications. There are many classification methods. The simplest one is nearest neighbor [Duda et al, 73; Aha et al, 89]. Its derivative, k-nearest neighbors, is more popular. Kernel regression, as mentioned in Chapter 2, is another important method. These methods are referred to as *memory-based* or *instance-based* classification methods [Atkeson et al, 97], while non-memory-based classification methods include neural network [Bishop, 95], decision-tree [Quinlan, 93], hierarchical mixtures of experts (HME) [Jordan, et al, 93], Bayes classifier [James, 85], etc. Both memory-based classifiers and non-memory-based ones assume the knowledge of the system S_p comes from the training data points. The distinguishing characteristic of memory-based classification methods is that they defer most of the processing of the training data points until after a query is made. This characteristic is desirable for processing continuous streams of training data and queries in real-time systems. In addition, the memory-based classifiers are capable of self-tuning according to the distribution and noise level of the training data points. Non-memory-based methods try to learn the underlying function model of the system S_p before any query comes. For example, neural networks have been proved capable of approximating any functions, if there is no restriction of the numbers of its hidden layers and its hidden nodes. Given a sufficient number of training data points, neural network uses them to approximate the underlying function relationship of the input and output. Once the training is done, the training data points are tossed away. Then, we wholly rely only on the trained neural network to process any queries.

Let's pick up some popular classification methods, and discuss them in a little depth.

1. Nearest neighborhood or *1*-nearest neighborhood doesn't perform satisfactorily in most cases, because it is too sensitive to the noise of the single nearest neighboring data point. *k*-nearest neighborhood performs quite well in many domains. But notice that it does not recognize the "boundary" of the different patterns. Besides, *k*-nearest neighborhood may be influenced by the density of the neighboring data points along the border. In the following diagram, intuitively the output of the query (the dark triangle) should be a cross,



because it is on the cross side. However, *k*-nearest neighborhood's conclusion tends to be a circle, because among the *k* nearest neighboring data points, the majority are circles.

2. Kernel regression is a good method for interpolation. However, it is not ideal for extrapolation. Suppose a query resides at a location remote from the centroid of other memory data points, like the reversed triangles in the above diagram, Kernel regression can not clearly decide if the category of the reversed triangle. Instead, it tends to assign 50% to the probability for the query's output to be "cross" (or "circle").

3. The simple Bayes classifier, referring to Section 3.3, puts too strong assumptions on the distribution of the data points. The conventional Bayes classifier assumes that if the out-

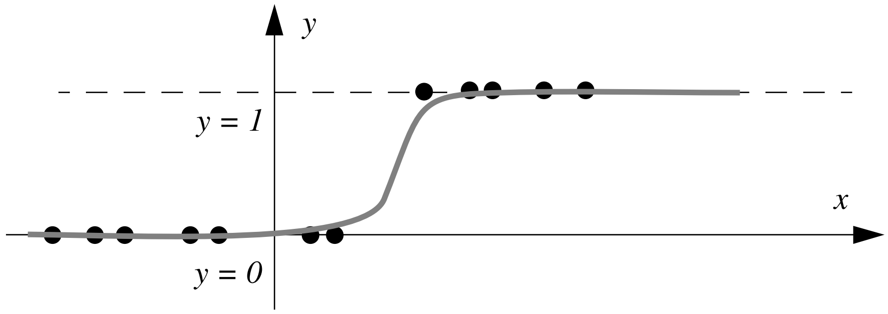
puts are boolean, the memory data points distribute in two clusters, one for the memory data points with output equal to 0, the other cluster for the data points with output equal to 1. Furthermore, the points are Gaussian-distributed in the input space so that the shapes of the clusters are ellipses. Referring to Figure 4-1, these restrictions are too strong for most classification problems. Even if we extend Bayes classifier to consider multiple clusters, it is still too hard to meet the requirement that the shapes of these clusters must be ellipses. Another concern about Bayes classifier is that it needs a large number of parameters to decide the centroids and the shapes of the Gaussian ellipses, this problem becomes more severe when we employ multiple ellipses.

4. The idea of a decision tree [Quinlan, 93] is to partition the input space into small segments, and label these small segments with one of the various output categories. However, conventional decision tree only does the partitioning to the coordinate axes. It is plausible that with the growth of the tree, the input space can be partitioned into tiny segments so as to recognize subtle patterns. However, overgrown trees lead to overfitting. More flexible than the conventional decision tree, CART [Breiman et al, 84] and Linear Machine Decision Tree [Utgoff et al, 91] can divide the input space using oblique lines. However, any nonlinear boundary may either make the tree overgrown or reduce the accuracy of the classification.

In this thesis, we explore a locally weighted version of logistic regression which can be used as a new memory-based classification method. Our method shares the properties of other memory-based classification methods. Besides, our method has some other good properties, including simplicity, capability of extrapolating, and a known confidence interval. Concerning the accuracy, our new method is competitive with others, supported by the experimental results.

4.2 Global logistic regression

Locally weighted logistic regression can be used to approximate $P(y_q \mid S_p, x_q)$. Let's begin with a very simple case with boolean output, shown in the following figure.



The straightforward way to approximate this function is to use two line segments to fit the dots, which are also referred to as training data points. However, to be learnable, we want to use a differentiable function to do the fitting instead of using two line segments. *Logistic* function, which is also referred to as *sigmoid* function, can be employed here. Logistic function is a monotonic, continuous function between *0* and *1*, whose shape is shown as the grey curve in the above figure. Mathematically, it is defined as,

$$\pi_q = \frac{1}{1 + \exp(-(1, \underset{\sim}{x}_q^T)\underset{\sim}{\beta})} \tag{4-1}$$

where $\underset{\sim}{x}_q$ is the input vector of the query, and $\underset{\sim}{\beta}$ is the parameter vector. π_q as the probability for y_q to be *1*, *i.e.*

$$\pi_q \equiv P(y_q = 1 \mid S_p, x_q) \qquad \text{Or, equivalently,}$$

$$y_q = \begin{cases} 1 & \text{with probability } \pi_q \\ 0 & \text{with probability } 1 - \pi_q \end{cases} .$$

Therefore, deciding the output of a query is now equivalent to finding the value of $\underset{\sim}{\beta}$.*Global* logistic regression assumes that all data points share the same parameter vector with the query, *i.e.*

$$\beta_1 = \beta_2 = \ldots = \beta_N = \beta$$

While *local* logistic regression allows $\underset{\sim}{\beta}_i$ vary cross the input space, but it changes smoothly. For example, if $\underset{\sim}{x}_1$ and $\underset{\sim}{x}_2$ are neighboring to each other, then we assume $\underset{\sim}{\beta}_1$ and $\underset{\sim}{\beta}_2$ must be close to each other, too. Back to global logistic regression, a good estimate of $\underset{\sim}{\beta}$ should fit, or in plain words, "go through", all the training data points as well as possible. Mathematically, the estimate of $\underset{\sim}{\beta}$ can be derived by maximizing the likelihood as following,

$$\begin{aligned} \mathrm{Lik}_G &= \prod_{i=1}^{N} P(y_i | S_p, x_i) = \prod_{i=1}^{N} \pi_i^{y_i}(1-\pi_i)^{1-y_i} \\ &= \prod_{i=1}^{N} \left[\frac{1}{1+\exp(-(1, \underset{\sim}{x}_i^T)\underset{\sim}{\beta})}\right]^{y_i} \left[\frac{\exp(-(1, \underset{\sim}{x}_i^T)\underset{\sim}{\beta})}{1+\exp(-(1, \underset{\sim}{x}_i^T)\underset{\sim}{\beta})}\right]^{1-y_i} \end{aligned} \tag{4-2}$$

Global logistic regression is a well-established algorithm in statistical literature [McCullagh et al, 89]. Although we discuss only the binary output case here, global logistic regression is ready to be extended to multiple categorical output cases. We will talk about this later.

The simplest classification problem is illustrated as Figure 4-2. The input is one-dimensional, which is represented by the horizontal axis; the output is boolean, represented by *0* or *1* on the vertical axis. The small circles in the pictures are the data points in memory. Global logistic regression works perfectly in the noise free case illustrated by Figure 4-2 (a), because the logistic function curve goes through most of the data points in memory. Global logistic regression also works in the noisy case shown as Figure 4-2 (b). Although the function curve moves midway between the data points, the curve is close to most of the data points. In summary, global logistic regression can be used as a noise tolerant classification method.

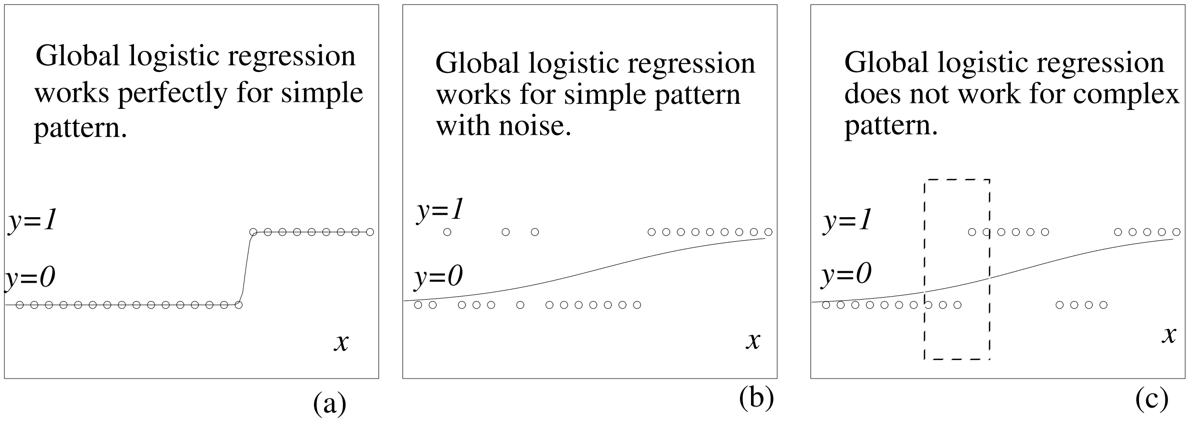


Figure 4-2: (Global) logistic regression for classification.

The fatal weakness of global logistic regression is shown in Figure 4-2 (c). Since it contains more than two segments, global logistic regression does not work. Recalling logistic function is a monotonic function, that is the reason global logistic regression fails whenever there are more than two segments. There are two approaches to solve this problem. One way to think is that although one logistic function does not work, we can combine several logistic functions. In fact, neural networks, especially feed-forward multi-layer perceptrons, can be regarded as an implementation of this idea.

The second approach resorts to the localization paradigm. The idea of local logistic regression is that although no single logistic function works well globally, in any local region a single function should be capable of doing the classification.

There are several versions of local logistic regression that can be investigated. *K*-nearest neighbor local regression would only select those neighboring data points, and ignores all others. Locally weighted version of logistic regression does not ignore any data points in memory, instead, it discriminates the data points by assigning weights to them.

4.3 Locally Weighted Logistic Regression

4.3.1 Maximum Likelihood Estimation

Locally weighted logistic regression is very similar to the global logistic regression, except that the locally weighted version assign a weight to $P(y_i|S_p, x_i)$. Differing from Equation 4-2, the locally weighted version of likelihood is,

$$\mathrm{Lik}_L = \prod_{i=1}^{N} P(y_i|S_p, x_i)^{w_i} \qquad \text{in which } w_i = \exp\left(-\frac{\mathrm{distance}(x_i, x_q)^2}{K_w^{\ 2}}\right) \tag{4-3}$$

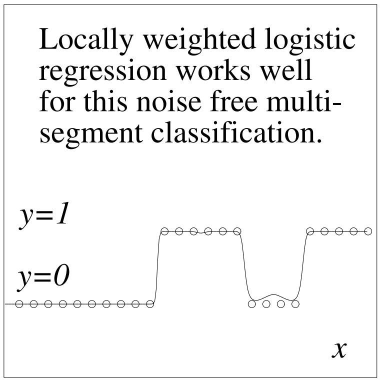
The weight is a function of the Euclidean distance from the *i*'th memory data point to the query. Other metrics of distance are also possible depending on the specific domains. The K_w in the weighting function definition is referred to as *Kernel width*. The influence of Kernel width will be discussed shortly. Due to the weights, those data points remote from the query have smaller weights, while the neighboring memory data points have bigger weights.

Using Newton-Raphson algorithm, and through some algebraic manipulations, the maximum likelihood estimate of $\underset{\sim}{\beta}$ can be simplified as,

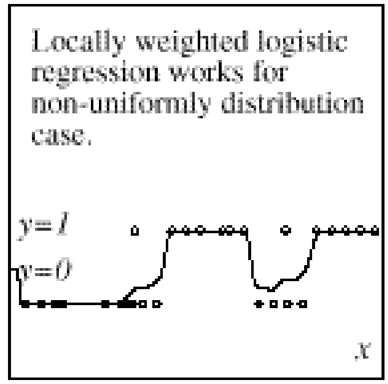
$$\hat{\beta}_{(r+1)} = \hat{\beta}_{(r)} + (X^T W X)^{-1} X^T W e \tag{4-4}$$

Suppose there are N data points in the memory, each data point consists of a d-dimensional input vector and a boolean output. X is then a $N \times (1 + d)$ matrix. The *i*'th row of X matrix is $(1, x_i^{\mathrm{T}})$. And W is a $N \times N$ diagonal matrix, whose *i*'th diagonals element is, $W_i = w_i \pi'_i$, where π'_i is the derivative of π_i with respect to $\underset{\sim}{\beta}$, i.e.,

$$\pi_i' \equiv \frac{\exp(-(1, x_i^T)\beta)}{(1 + \exp(-(1, x_i^T)\beta))^2} \begin{pmatrix} 1 \\ x_i \end{pmatrix}. \tag{4-5}$$



(a)



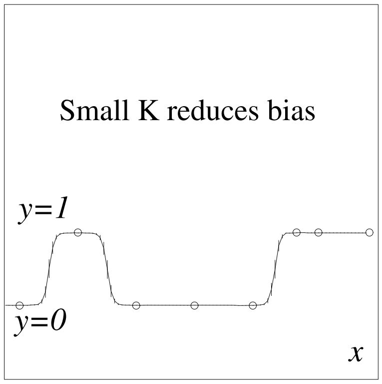
(b)

Figure 4-3: Locally weighted logistic regression as a classification technique works robustly.

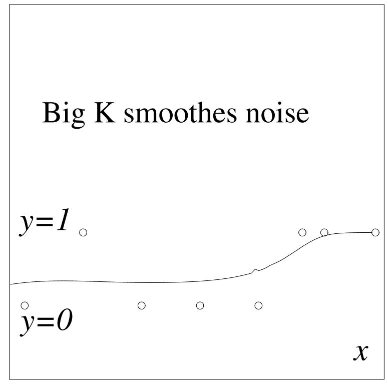
w_i is the weight defined in Equation 4-3. The last item, e is a ratio of $y_i - \pi_i$ to π'_i. Newton-Raphson algorithm starts from a random vector of $\underset{\sim}{\beta}$, usually we assign $\hat{\underset{\sim}{\beta}}_{(0)}$ to be zero vector. The recursive process converges very quickly, usually no more than 10 loops. Once we get the maximum likelihood estimate of $\underset{\sim}{\beta}$, we can estimate the query's π_q.

Also notice that, $(X^t WX)^{-1}$ is the asymptotic variance matrix of $\hat{\underset{\sim}{\beta}}$.

Now let us go back to the case of Figure 4-2 (c) and see if locally weighted logistic regression classifier is capable of solving the problem where global logic regression fails. The result is shown in Figure 4-3 (a). The circles are the memory data points. And each dot on the solid curve, which is π_q, is plotted by doing its own locally weighted regression at that local region. Locally weighted logistic regression works well in this case. Also, in the harder case of Figure 4-3 (b), it still works. Notice, π_q is influenced by the noise but not the distribution of the data points.



(a)



(b)

Figure 4-4: Kernel width adjusts the weighting function.

4.3.2 Weighting Function and Kernel Width

Referring to Equation 4-3, w_i, the weight can be adjusted by the Kernel width.When the Kernel width is big, more data points have high weights. Therefore, a big K_w is usually preferred when the noise level in memory is high. Extremely, when K_w goes to infinity, locally weighted logistic regression is equivalent to the global one. When K_w is small, only those close neighbors can effect the regression. Hence, a small K_w is good at recognizing the details of the memory. The influence of K_w is demonstrated by Figure 4-4.

4.3.3 Confidence Interval

Our estimate $\hat{\pi}_q$ is a point estimate, which is our best guess for the true value of π_q. Reporting only the point estimate is often unsatisfactory. Some measure of how close the point estimate is likely to be the true value is required. The *confidence interval* is such a metric.

The confidence interval of π_q is an interval of plausible values for π_q, $[\pi_L, \pi_U]$; the probability or the confidence for the true value of π_q falling into this interval is $100(1 - \alpha)\%$, in which α is the *confidence level*. Usually we pre-define a confidence level, then decide the lower and upper bounds, π_L and π_U, which are also effected by the density and consistency (noise level) of the data points in the neighborhood of x_q.

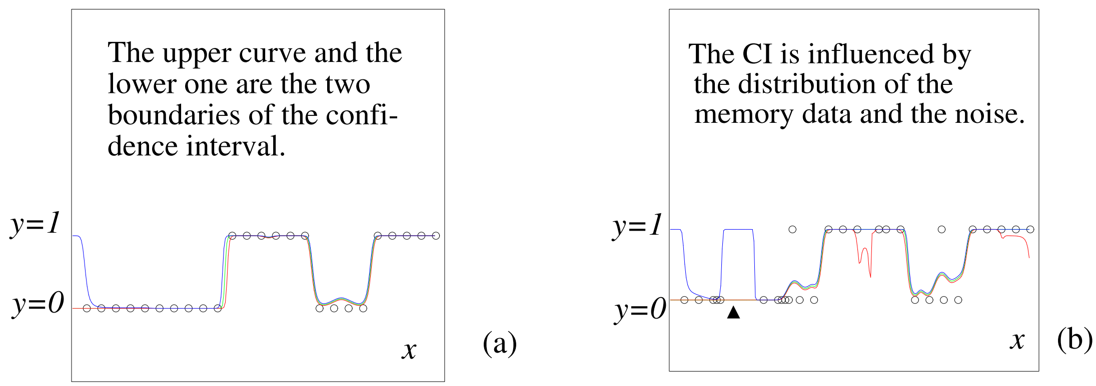


Figure 4-5: Confidence intervals for classification.

Referring to Equation 4-1, π is a monotonic function of $(1, x_q^T)\underset{\sim}{\beta}$; hence, to calculate the lower and upper bounds, we need know the lower and upper bounds of $(1, x_q^T)\underset{\sim}{\beta}$. Referring to Subsection 4.3.1, we can calculate the asymptotic variance of $\hat{\beta}_q$ which is $(X^T WX)^{-1}$, where X is decided by the memory data points and W is effected by the distances from the query to the memory data points. Notice that the asymptotic variance of $\hat{\beta}_q$ is likely to be small when there are more data points in the memory, especially in the neighborhood of the query. It is straightforward to calculate the confidence interval of $\hat{\pi}_q$ based on the upper and lower bounds of $x_q\hat{\beta}_q$.

The confidence intervals of the cases of Figure 4-3 (a) and (b) are plotted in Figure 4-5 (a) and (b). When the data points distribute uniformly as Figure 4-5 (a), the confidence interval is quite consistent. Otherwise, the confidence interval varies cross the input space.

According to $\pi_q{}'$'s definition, referring to Equation 4-5, when $\hat{\pi}_q$ is close to *1.0*, we tend to predict that the query's output y_q is likely to be *1*. However, if at the same time $\hat{\pi}_q$'s confidence interval is too big, we should be conservative about our prediction. Figure 4-5 (b) shows such a situation: $\hat{\pi}_q$ is almost zero, therefore, if we only rely on $\hat{\pi}_q$, we should predict the output will be *0*. But since the confidence interval is very wide, we should be aware that there is still a lot of chance for the output y_q to be *1*.

Confidence interval is helpful for active learning and/or experimental designs. Wherever the confidence interval is wide, we need more data points in that region.

4.3.4 Multi-categorical classification inference

Up to now, we focus on boolean classification. In case the output has more than two output categories, locally weighted logistic regression method is still useful. But we should do some modifications.

1. Suppose there are m output categories, we can represent the output by a m-dimensional vector. If a data point falls into the first category, its output, $\underset{\sim}{y}$, is $[1, 0, \ldots, 0]^T$; if it is in the second category, $\underset{\sim}{y}$ is $[0, 1, \ldots, 0]^T$. In general, the distribution of output is multinomial, in the form of,

$$P(\underset{\sim}{y}_q | S_p, \underset{\sim}{x}_q) = \pi_1^{\underset{\sim}{y}_q} \pi_2^{\underset{\sim}{y}_q} \ldots \pi_m^{\underset{\sim}{y}_q} = \prod_{j=1}^{m} \pi_j^{\underset{\sim}{y}_q}$$

where π_j the probability for the data point falling into the j'th category.

2. We assume π_j is decided by a function similar to logistic function,

$$\pi_j = (\exp((1, \underset{\sim}{x}_q^T) \underset{\sim}{\beta}_j)) / \sum_{j=1}^{m} \exp((1, \underset{\sim}{x}_q^T) \underset{\sim}{\beta}_j)$$

Notice that the sum of π_j, $j = 1, \ldots, m$, is 1.0. And for each output category, there is a unifying $\underset{\sim}{\beta}_j$; totally, there are m of them.

3. The likelihood can be constructed following the descriptions in Section 4.2 and section 4.3. For example, the global likelihood, which assumes all data points share the same $\underset{\sim}{\beta}$, is defined by Equation 4-6,

$$\mathrm{Lik}_G(\underset{\sim}{\beta}) = \prod_{i=1}^{N} P(\underset{\sim}{y}_i | S_p, \underset{\sim}{x}_i) = \prod_{i=1}^{N} \prod_{j=1}^{m} \pi_j^{y_i} \tag{4-6}$$

Now it is straightforward to follow the same inferences described in Sections 4.3 to figure out the locally weighted regression of $\underset{\sim}{\beta}_q$ and the confidence interval of π_q.

4.4 Comparison Experiment

Artificial Experiments

We artificially generate three data sets, each data consists of two input attributes (*2-d* input) and a boolean output. In Figure 4-6, we represent those data points with output values equal to *0* by circles, and represent the other data points, whose outputs are *1*, by crosses.

Figure 4-6 (a-c) are the contours of the $\hat{\pi}_q$ values corresponding to three different memory data sets. Figure 4-6 (a) shows a simple case, in which locally weighted logistic regression does a perfect job. Figure 4-6 (b) is similar to Figure 4-6 (a) except that, the “boundary” of the two regions is messier, and there is noise involved as well. In this case, $\hat{\pi}_q$ value increases from *0* to *1*, starting from the bottom left corner to the top right one; hence, locally weighted logistic regression works well, too. The small gradient of the contour of $\hat{\pi}_q$ shows the influence of the inconsistency (noise) of the data points in memory. Figure 4-6 (c) is the hardest case, in which locally weighted logistic regression still works well. Figure 4-6 (d) is the contour of *confidence interval* for the same memory as Figure 4-6 (c). It is apparent that the memory data points’ noise level, as well as their distribution and density, influence the confidence interval.

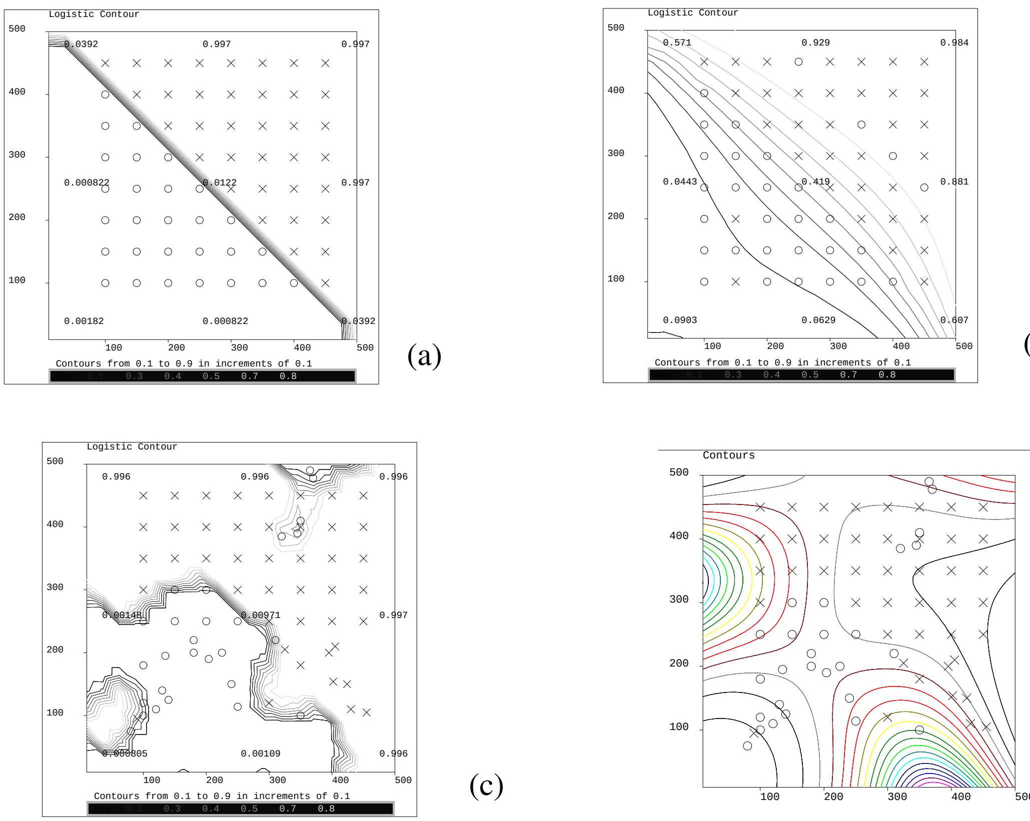


Figure 4-6: Three artificially generated data sets as the testbeds of locally weighted logistic regression classifier.

Real World Datasets

We use four binary output data sets from UCI's machine learning dataset repository, Ionos., Pima., Breast., and Bupa. We try six different classifiers, including nearest neighbor method (*1-Nearest*), *k*-nearest neighbors (*k-Nearest*), Kernel regression (Kernel), conventional Bayes classifier with two clusters (Bayes), C4.5 decision tree (*Decision*), feedforward perceptron (*Neural*), global logistic regression (*Global Logistic*) and our locally weighted logistic regression method (*Local Logistic*). The dimensionalities of the inputs vary from *6-d* to *34-d*.

We split each data set into two parts, the first part contains two thirds of the data points, which are used as the memory or the training dataset. The remaining one third of the data points are used as the test set. We can approximate the accuracy of a certain method for a certain dataset by the *error rate*, which is the ratio of the number of the failures to the number of the testing data points. For the same dataset, the lower the error rate, the better the classification method performs.

With different Kernel width, locally weighted logistic regression may have different accuracies. We split the range of the Kernel width into ten equal-length steps, and tried the logistic regressions using these ten different Kernel widths, so as to find the optimal Kernel width. Similarly, for k-nearest neighbor method and kernel regression, we enumerated parameter k from *10* to *100* with step *10*; for perceptron, we tried one-hidden layer feedforward perceptron with *1* to *10* hidden nodes. In this way, we found the best parameters for the various machine learning methods.

For each dataset, we shuffled it five times; each time we split it into training set and testing set. Hence, for each dataset by each method, we got five error-rates which were the best performances of the method with the tuned-up parameter(s). We recorded the mean values of these error-rates in Table 4-1, along with the standard deviations in parentheses.

Table 4-1: Comparison of logistic classifier with other methods

Error rate (%)	Ionos. (34-d)	Pima (8-d)	Breast (9-d)	Bupa (6-d)
1-Nearest	12.7 (2.5)	33.9 (1.8)	4.9 (0.6)	40.0 (2.4)
k-Nearnest	13.9 (2.9)	31.5 (4.7)	3.3 (0.5)	37.5 (5.8)
Kernel	12.7 (3.3)	30.9 (3.2)	3.3 (0.6)	37.3 (2.0)
Bayes	12.9 (1.2)	25.3 (2.3)	3.4 (1.2)	34.2 (3.6)
Decision	9.2 (2.1)	28.6 (3.0)	4.2 (1.1)	35.8 (3.2)
Neural	10.5 (3.2)	33.4 (2.0)	3.2 (0.6)	32.1 (4.5)
Global Logistic	12.4 (0.7)	24.9 (3.0)	3.9 (1.4)	34.4 (3.4)
Local Logistic	13.0 (0.4)	22.5 (2.8)	3.1 (0.7)	31.0 (2.7)

The experiments show that the accuracy of the locally weighted logistic regression method (Local Logistic) is competitive compared with other classification method. Some remarks are listed as following,

1. It is not surprising that locally weighted logistic regression is more accurate in most cases than *1*-nearest neighborhood, *k*-nearest neighborhood, Kernel regression, conventional Bayes classifier, C4.5 decision tree, and global logistic regression according to our discussion in Section 4.1.

2. Global logistic regression's performance is similar to that of the conventional Bayes classifier with two clusters. But global logistic regression is computationally cheaper than the conventional Bayes classifier. Suppose the input space's dimensionality is d and the memory size is N, the computation cost of locally weighted logistic regression is $O(d^3 + d \times N)$, while that of the conventional Bayes classifier with improved efficiency by some tricks is $O(d^3 \times N + d \times N \times k)$, where k is the number of clusters.

3. Concerning neural networks, locally weighted logistic regression does not outperform it in accuracy. Instead, an advantage comes from the general good properties of the memory-based approach over non-memory-based ones. As mentioned in the beginning of this chapter, Section 4.1, as well as [Atkeson et al., 97], because memory-based learning does not process data until the query arrives, the parameters of the logistic regression are not fixed in advance. When we update the memory, unlike neural network, less interference will happen, because the previous arrived memory data points are treated equally as the new comers. And by adjusting the parameters, we can shift the logistic regression continuously along the global-local spectrum.

4. Locally weighted logistic regression performs poorly on the Ionos dataset. The reason is that the dimensionality of the input is very high (34-d). Maybe many input attributes are irrelevant to the classification but only confuse the classifiers. When we selected the first,

the fourth and the fifth attributes to be input, the mean value of error-rate of the local logistic classifier dropped from *13.0%* to *10.7%*, with standard deviation *0.7%*.

To eliminate those less important input variables, recall that locally weighted logistic regression estimates the parameter vector β. In fact, each element of β indicates the significance of the corresponding input attribute for classification. If one element of β is close to zero, it implies that the corresponding input attribute is not very relevant to the classification job. We can get rid of the irrelevant input attributes using this heuristic. Some preliminary experiments showed that the selection result was quite consistent with the nodes of decision tree.

4.5 Summary

In this thesis, we explore a locally weighted version of logistic regression which can be used as a new memory-based classification method. Our method shares the properties of other memory-based classification methods. Besides, our method has some other desirable properties, including simplicity, competitive accuracy, capability of extrapolating, and confidence interval.

In Chapter 5 and Chapter 6, we will discuss the issue about how to improve the efficiency of locally weighted logistic regression as well as other memory-based methods.

Chapter 5

Efficient Memory Information Retrieval

In this chapter, we will talk about two topics: (1) What is a kd-tree? (2) How can we use kd-trees to speed up the memory-based learning algorithms? Since there are many details in the second topic, we only discuss how to improve the efficiency of Kernel regression in this chapter, to demonstrate the approach in principle. In next chapter, we will explain the details of applying kd-tree techniques to improve the efficiency of locally linear regression and locally weighted logistic regression.

5.1 Efficient information retrieval

Suppose there are a set of memory data points whose input space is 2-dimensional, shown in Figure 5-1. Given a query (x_q, y_q), a task of information retrieval is to find this query's neighboring memory data points. The brute force approach is to measure the distances from this query to each of the memory data points. Then based on these distances, it is straightforward to decide which memory data points are the query's neighbors. The distance may be Euclidean or another metric depending on the specific domain. The drawback of the brute force method is obvious: since its computational cost is $O(N \times d)$, where N is the memory size and d is the dimensionality of the input space. When the memory size N becomes very large, its costs will increase, too.

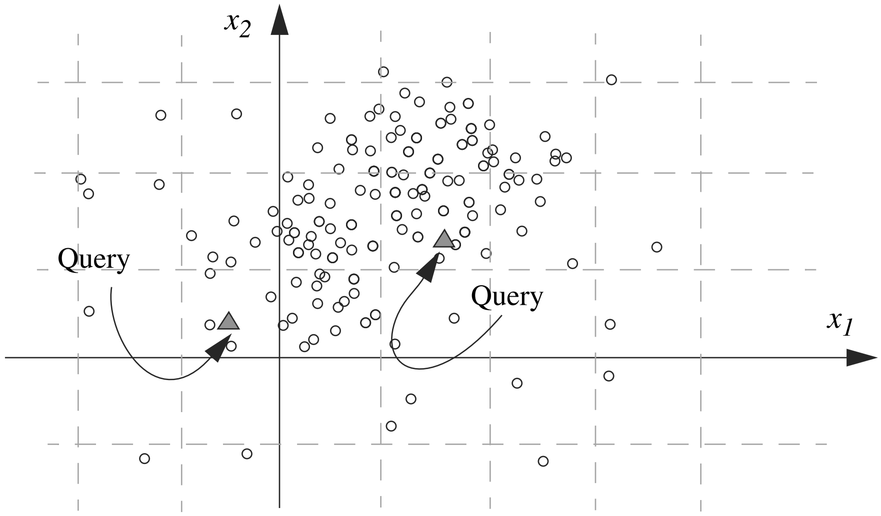


Figure 5-1: Grid for efficiency information retrieval.

To improve the efficiency of finding the neighbors, we can partition the input space of the memory data points into many cells by means of a grid. When a query arrives, we can consult the cell where the query locates and its neighboring cells, instead of visiting all the memory data points individually. In this way, the computational cost shrink from $O(N \times d)$ to $O(n \times d^2)$, where n is the number of memory data points in the concerned cell(s). (If we neglect the cost of finding the cell where the query resides.) The grid method performs the best when the memory data points distribute uniformly, so that n tends to be N / G, in which G is the number of grids in the whole input space. However, there is no guarantee that the memory data points distribute uniformly forever and wherever. Sometimes most of the memory data points are packed in only a limited number of cells, while the other cells are almost vacant. Therefore, the contribution of the grid method to the efficiency is not reliable.

The kd-tree technique [Preparata et al, 85] is similar to the grid method in the sense that it also partitions the input space into many cells. However, the partition is flexible with respect to the density of the data points in the input space. Wherever, the density is high in the input space, the resolution of the kd-tree's partition at that region is also high, so that the cells tend to be small. Otherwise, for those regions where there are only a limited number of memory data points, the partition resolutions are low, and the cells are large.

5.2 Kd-tree Construction and Information Retrieval

A kd-tree is a binary tree that recursively splits the whole input space into partitions, in a manner similar to a decision tree [Quinlan, 93] acting on real-valued inputs. Each node in the kd-tree represents a certain hyper-rectangular partition of the input space; the children of this node denote subsets of the partition. Hence, the root of the kd-tree is the whole input space, while the leaves are the smallest possible partitions this kd-tree offers. And each leaf explicitly records the data points that reside in the leaf. The tree is built in a manner that adapts to the local density of input points and so the sizes of partitions at the same level are not necessarily equal to each other.

In our formulation of the kd-tree structure, each node records the hyper-rectangle covered by it. This is defined as the smallest bounding box that contains all the data points owned by this node of the tree. Each non-leaf node has two children representing two disjoint subregions of the parent node. The break between the children is defined by two values: **`split_d`** is the splitting dimension, which determines which component of input space the children will be split upon; **`split_v`** determines the numerical value at which each split occurs. The data points owned by the left child of a node are those data points owned by the node which are less than value **`split_v`** in input component **`split_d`**. The right child contains the other data points. A sample kd-tree is shown in Figure 5-2.

To construct a tree from a batch of training data points in memory, we use a top-down recursive procedure. This is the most standard way of constructing kd-trees, described, for example, in [Preparata et al., 85] [Omohundro, 91]. In our work, we use the common variation of splitting a hypercube in the center of the widest dimension instead of at the median point. This method of splitting does not guarantee a balanced tree, but leads to generally more cubic hyper-rectan-

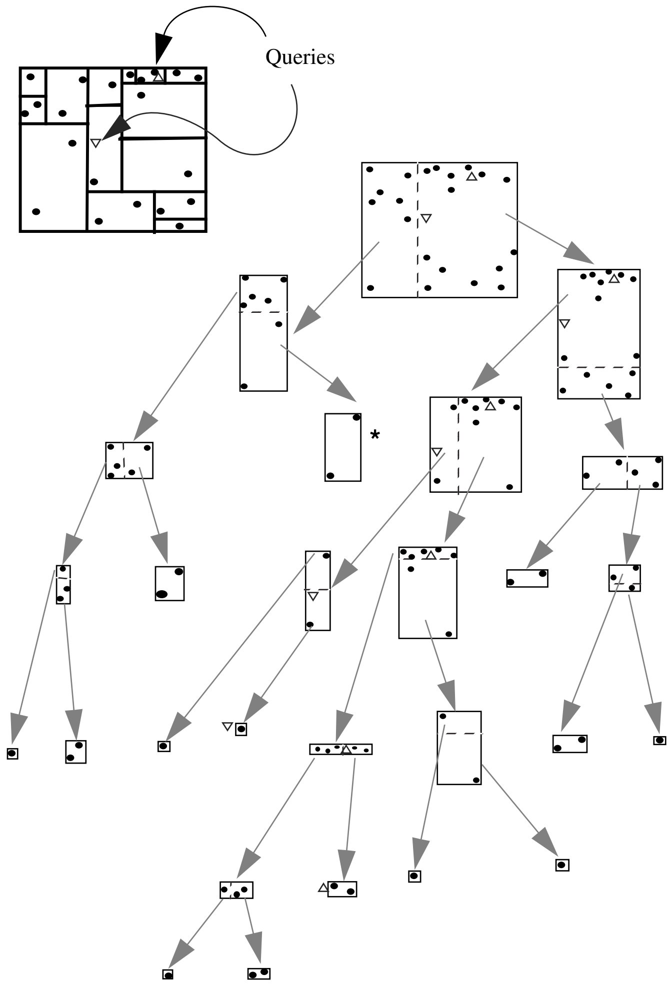


Figure 5-2: To implement the grouping idea, we use hyper-rectangles with *k*d-tree. To find the neighborhood of a certain query (triangle), we can recursively search the tree from the root towards to the leave where the query resides. For different query (reversed triangle), we can use the same kd-tree but choose different nodes.

gles, which has empirically proved better than other schemes (pathologically imbalances are conceivable, but trivial modifications to the algorithm prevent that.) The cost of making a tree from N data points is $O(Nd \log N)$.

The base case of the recursion occurs when a node is created with N_{min} or fewer data points. Then those data points are explicitly stored in the leaf node. In our experiments, $N_{min} = 2$.

To incrementally add a new data point to the tree, the leaf node containing the point is determined (*O(logN)* cost). The data point is inserted there (and a new subtree is recursively built if the number of nodes exceeds N_{min}).

Given a query (x_q, y_q), to find those memory data points whose input vectors are close to x_q, we can recursively search the tree from the root towards to leaves, referring to Figure 5-2, with the triangle query. According to the pre-defined range of "neighborhood", it is straightforward to find those branches of the kd-tree, which are close to the branches where the query resides. Two issues to be noticed:

1. With different ranges of the "neighborhood", the "neighboring" branches can be different. The neighboring branches with respect to a strict defined neighborhood is a subset of those neighboring branches corresponding to a loose definition. This characteristic is desirable, because it allows us to find those neighboring data points corresponding to any definition of the neighborhood along the local-global spectrum.

2. Although we will use the kd-tree to find a *set* of neighboring data points, it is also possible to find the "exact" nearest neighboring data point. For the example in Figure 5-2 with the reversed triangle query, to find its nearest neighbor data point, we wish we could search from the root of the tree down towards to the leaf where the query locates, so that the cost is $O(\log N)$, where N is the memory size. Unfortunately, it is possible that its nearest neighboring data point is in another leaf of a remote branch of the kd-tree, marked with "*" in the diagram. More theoretical analysis refers to [Kleinberg, 97]. The standard nearest neighbor algorithm, [Preparata et al, 85] [Moore, 90], avoids this problem while still only requiring $O(\log N)$ time.

5.3 Cached Kd-tree for Memory-based Learning

The goal of our exploring kd-trees is not to find the nearest neighbor, and not only to find a set of nearest neighbors, but mainly to enhance the efficiency of the memory-based learning methods. The basic principle is to cache useful statistical information into the kd-tree nodes, so that when we do the memory-based learning process, instead of visiting every relevant memory data point, we mainly rely on the statistical information in the tree nodes. In this chapter, we focus on using this cached kd-tree to speed up Kernel regression, to demonstrate the approach in general.

Kernel regression

In Chapter 2, we discussed using Kernel regression's idea to approximate $P(y_q \mid S_p, x_q)$, i.e. the probability that a given query data point (x_q, y_q) belongs to a system S_p, where the knowledge of S_p comes from a set of memory data point, (x_1, y_1) ..., (x_N, y_N), which is the observations of S_p's previous behavior. Cached kd-trees can improve the efficiency of Kernel regression (for example, [Franke, 82]), not only for the approximation of $P(y_q \mid S_p, x_q)$, but also for the general purpose use. As a popular machine learning method, Kernel regression is often used to do prediction: given an input vector x_q, which is called query, Kernel regression predicts its output, $\hat{y}_q(x_q)$, based on the memory data points $(x_1, y_1), ..., (x_N, y_N)$. We assume all the memory data points were generated by an identical system.

Kernel regression use the weighted average of the outputs of all the memory data points to predict $\hat{y}_q(x_q)$:

$$\hat{y}_q(x_q) = \left(\sum_{i=1}^{N} w_i y_i\right) \Big/ \left(\sum_{i=1}^{N} w_i\right) \tag{5-1}$$

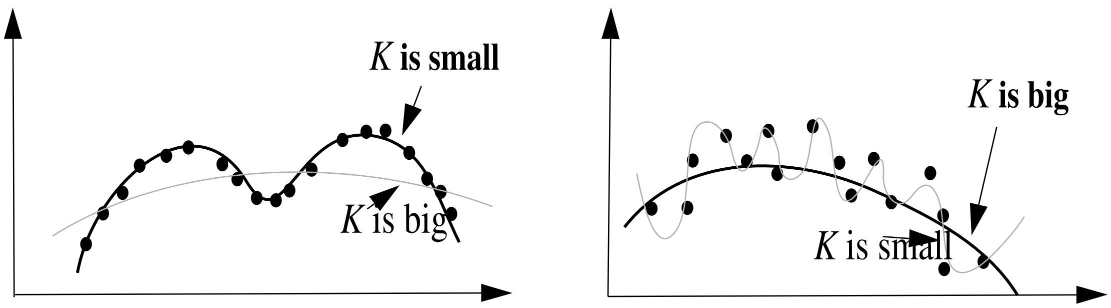


Figure 5-3: For the noiseless data in the top example, a small K gives the best regression (in terms of future predictive accuracy). For the noisy data in the bottom example, a large K is preferable.

where w_i is the weight assigned to the i'th datapoint in our memory, and is large for points close to the query and almost zero for points far from the query. It is usually calculated as a decreasing function of Euclidean distance, for example by Gaussian:

$$w_i = \text{Const} \times \exp\left(-\frac{\|x_q, x_i\|^2}{2K_w^2}\right)$$

As we have mentioned previously, K_w is the Kernel width. The bigger the parameter K_w is, the flatter the weight function curve is, which means that many memory points contribute quite evenly to the regression. As K_w tends to infinity the predictions approach the global average of all points in the database. If the K_w is very small, only closely neighboring data points make a significant contribution. K_w is an important smoothing parameter for kernel regression. If the data is noise free then a small K_w will avoid smearing away fine details in the function. If the data is relatively noisy, we expect to obtain smaller prediction errors with a relatively large K_w. This is illustrated in Figure 5-3.

The drawback of kernel regression is the expense of enumerating all the distances and weights from the memory points to the query. This expense is incurred every time a prediction is required. Several methods have been proposed to address this problem, reviewed as following:

1.[Preparata *et al*, 85] proposed a range-search solution. Similar to our cached kd-tree method, the range-search solution finds all points in the kd-tree that have significant weights, and then only sum together the weighted components of those points. This is only practical if the kernel width K_w is small. If it is large, all the memory data points may have significant weights, but with only small local variations, thus range searching would sum all the points individually. Even in cases of small kernel widths, but if there are many data points in the neighborhood, the range search method will need to search all the data points individually and may still end up with a large computational cost.

2.Another solution to the cost of conventional Kernel regression is *editing* (or *prototypes*): most data points are forgotten and only particularly representative ones are used (e.g. [Kibler and Aha, 88] [Skalak, 94]). Kibler and Aha extended this idea further by allowing data points to represent local averages of sets of previously-observed data points. This can be effective, and unlike range-searching can be applicable even for wide kernel widths. However, the degree of local averaging must be decided in advance, and queries cannot occur with different kernel widths without rebuilding the prototypes. A second occasional problem is that if we require very local predictions, the prototypes must either lose local details by averaging, or else all the data points are stored as prototypes.

3.Decision trees and *k*d-trees have been previously used to cache local mappings in the tree leaves [Grosse, 89], [Moore, 90], [Omohundro, 91], [Quinlan, 93]. These algorithms provide fast access once the tree is built, but a new structure needs to be built each time new learning parameters, such as Kernel width, are required. Furthermore, the resulting predictions from the tree have substantial discontinuities between boundaries. Only in [Grosse, 89] is continuity enforced, but at the cost of tree-size, tree-building-cost and prediction-cost all being exponential in the number of input variables.

Computing the kernel regression sums

Now it is time for us to use the cached kd-tree to improve the efficiency of Kernel regression, and at the same time avoid the drawbacks of the other competing methods.

Recall that each kd-tree node represents a hyper-rectangle sub-region of the input space, which covers a set of memory data points. Assume in one node there are n data points, and corresponding to a certain query, these n data points' weights are all close to a value w; in other words, the weight of the i'th data point in this node is $w_i = w + \xi_i$, where all ξ_i's are small. Referring to Equation 5-1, when performing Kernel regression, we need to accumulate two sums over all data points in memory, including these n data points in this node,

$$\sum w_i y_i \quad \text{and} \quad \sum w_i$$

Restricting our attention to summations over the n data points in the concerned kd-tree node, we have,

$$\sum w_i y_i = \sum (\bar{w} + \varepsilon_i) y_i = \bar{w} \sum y_i + \sum \varepsilon_i y_i \quad \text{and}$$
$$\sum w_i = \sum (\bar{w} + \varepsilon_i) = n\bar{w} + \sum \varepsilon_i$$

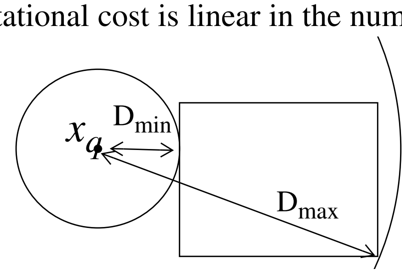
Providing we know n, w and Σy_i for the current node, we can therefore compute an approximation to $\Sigma w_i y_i$ and Σw_i in constant time without needing to sum individual data points contained in the node. This approximation to the partial sums is good to the extent that $\Sigma \varepsilon_i y_i$ is small with respect to $w \Sigma y_i$ and $\Sigma \varepsilon_i$ is small with respect to nw.

Therefore, we should cache two other pieces of information into each kd-tree node in conjunction with **`split_v`** and **`split_d`**: the number of data points below the current node, **`n_below`**, and the sum Σy_i of all output values of the data points contained in the node, **`sum`**. These are two of the three values needed to compute the contribution of a kd-tree node to the partial sums in Kernel regression. The third component, w, depends upon the location of the query and is determined dynamically in a manner described shortly.

With such cached information in each kd-tree node, we can efficiently approximate $\Sigma w_i y_i$ and Σw_i, summed over all data points in the kd-tree, so as to speed up the process of Kernel regression. This is performed by a top-down search over the tree. At each node we make a decision between:

1.(Cutoff) Treat all the points in this node as one group (a cheap operation) or
2.(Recurse) search the children.

We will use the cutoff option if we are confident that all weights inside the node are similar. Given the current query x_q and the hyper-rectangle of the current node it is an easy matter to compute $D_{\min}$ and $D_{\max}$: the minimum and maximum possible distances of any datapoint in this node to the query (computational cost is linear in the number of dimensions). From these



values one can then compute the maximum and minimum possible weights $w_{\max}$ and $w_{\min}$ of any data points owned by this node, since the weight of a point is a decreasing function of distance to the query. We thus decide if $w_{\max}$ and $w_{\min}$ are close enough to warrant the cut-off option.

The search is thus a recursive procedure which returns two values: *sum-weights* and *sum-wy*. If the cutoff option is taken, then estimate the weight of all data points as $\overline{w} = (w_{min} + w_{max})/2$ and return:

$$\begin{aligned} \textit{sum-weights} &= \texttt{n\_below} \times \overline{w} \\ \textit{sum-wy} &= \texttt{sum} \times \overline{w} \end{aligned}$$

If the cutoff option is not taken, recursively compute *sum-weights* and *sum-wy* for the left and right children, and then return:

$$\textit{sum-weights} = \textit{sum-weights(left)} + \textit{sum-weights(right)}$$

$$\textit{sum-wy} = \textit{sum-wy(left)} + \textit{sum-wy(right)}$$

Search cutoffs

Last section described how we can make our approximation arbitrarily accurate by bounding the maximum deviation we will permit from the true weight estimate with a value $\varepsilon_{max} > 0$ and then making ε_{max} arbitrarily small. Thus the simplest cutoff rule in the kd-tree search would be to cutoff if $w_{max} - w_{min} < \varepsilon_{max}$. It is easy to show that this guarantees that the total sum of absolute deviations $|\Sigma\varepsilon_i|$ is less than $N_T\varepsilon_{max} / 2$ where N_T is the number of points in the tree. There are, however, other possible cutoff criteria which provide arbitrary accuracy in the limit, but which, when used as an approximation, have more satisfactory properties.

The simple cutoff rule does not take into account that a larger total error will occur if the node contains very many points than if the node contains only a few points. It does also not account for the fact that in a practical case we are less concerned about the absolute value of the sum of deviations $|\Sigma\varepsilon_i|$ but rather the size of $|\Sigma\varepsilon_i|$ relative to the sum of the weights Σw_i. Some simple analysis reveals a cutoff criterion to satisfy both of these intuitions. Cutoff only if

$$(w_{max} - w_{min})\, N_B < \tau\, \Sigma \mathrm{w}_i$$

where N_B is the number of data points below the current node. Simple algebra reveals that this guarantees

$$|\, \Sigma\varepsilon_i \,| < 0.5\ \mathrm{G}\ \tau\ \Sigma \mathrm{w}_i$$

where G is the number of groups finally used in the search (and thus G < N_T, hopefully considerably less). Notice that this cutoff rule requires us to know $\Sigma \mathrm{w}_i$ in advance, which of course we do not. Fortunately the sum of weights obtained so far in the search can be used as a valid lower bound, and so the real algorithm makes a cutoff if

$$\frac{(w_{\max} - w_{\min})N_B}{\text{weight so far in search}} < \tau$$

where τ is a system constant.

5.4 Experiments and Results

Let us review the performance of the Kernel regression with the help of cached kd-tree in comparison to the conventional Kernel regression. In the first experiment we use a trigonometric function of two inputs with added noise: x_i = uniformly generated random vector with all components between 0 and 100 and y_i = a function of x_i (which ranges between 0 and 100 in height), with gaussian noise of standard deviation 10.

10,000 data points were generated. Experiments were run for different values of kernel width K_w. In all experiments, the cutoff threshold τ was 0.005. Figure 5-4 (a1) shows the test-set error on 1000 test points for both regular kernel regression ("Regular KR") and cached kd-tree's kernel regression ("Tree KR") graphed for different values of K_w. The values are very close, indicating that Tree KR is providing, for a wide range of kernel widths, a very close approximation to Regular KR. Figure 5-4 (a2) shows the computational cost (in terms of the summations that dominate the cost of KR) of the two methods. Regular KR sums all points, and so is a constant 10,000 in cost. Tree KR is substantially cheaper for all values of K_w, but particularly so for very small and very large values.

Figures 5-4 (b1) and (b2) show corresponding figures for a similar trigonometric function of five inputs. This still shows similar prediction performance as Regular KR. The cost of kd-tree's Kernel regression is still always less than Regular KR, but in the worst case the computational saving is only a factor of three (when K_w = 40, Tree KR cost = 3,200). This is not an

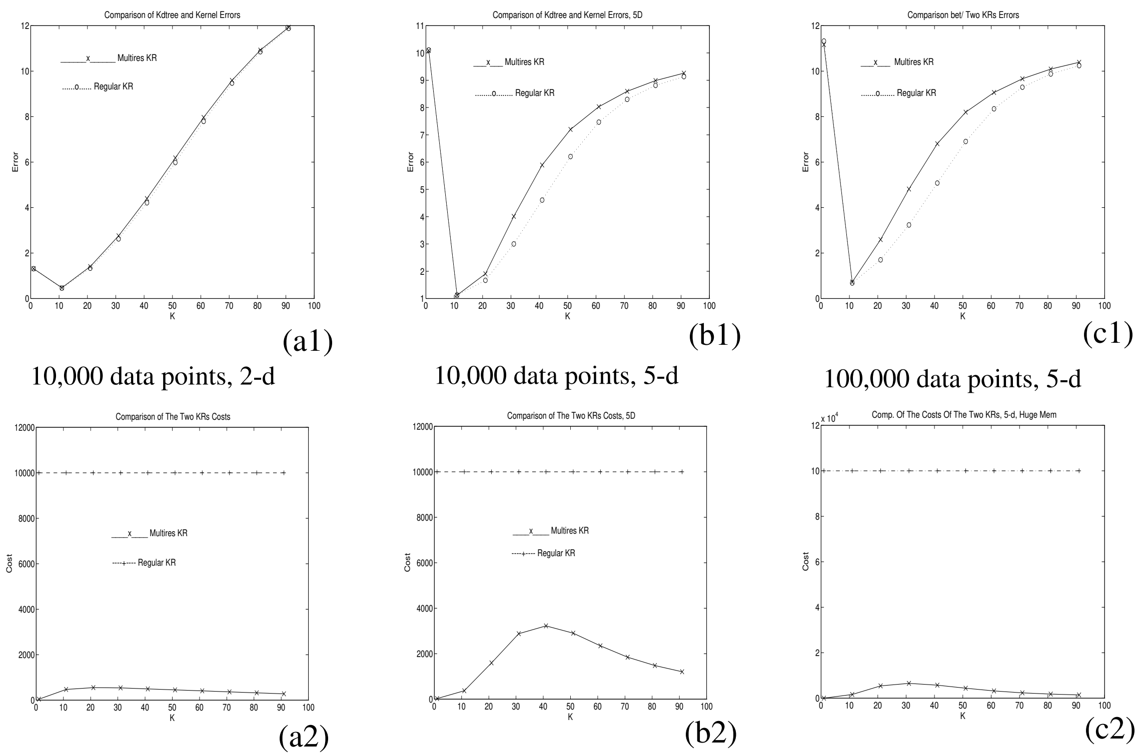


Figure 5-4: Comparison between the errors (*1) and the costs (*2) between regular kernel regression versus cached kd-tree's one. In the cases of (a*), the dataset is of 2-d inputs, of size 10,000. In (b*), 5-d inputs, dataset size 10,000. In (*c), 5-d inputs, 100,000 data points.

especially impressive result. However, for any fixed dimensionality and kernel width, costs rise sub-linearly (in principle logarithmically) with the number of data points. To check this, we ran the same set of experiments for a dataset of ten times the size: 100,000 points. The results, in Figure 5-4 (c1) and (c2), show that with this large increase in data, the effectiveness of cached kd-tree's KR becomes more apparent. For example, consider the $K_w = 40$ case. With 100,000 data points instead of 10,000, the cost is only increased from 3,200 to 5,700 while the cost of Regular KR (of course) increased from 10,000 to 100,000.

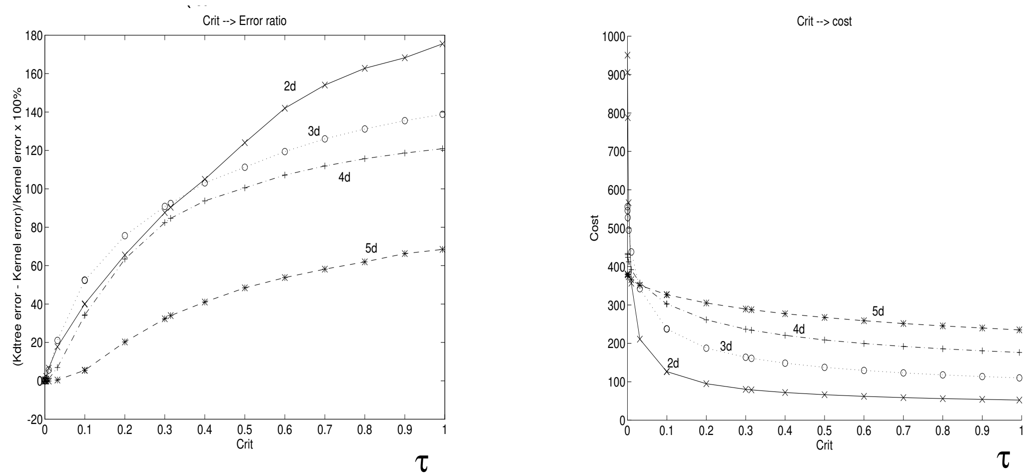


Figure 5-5: (Upper) the relative accuracy and (lower) the computational cost of kd-tree's KR against τ --- the cutoff threshold.

Investigating the τ threshold parameter

Next, we will examine the effect of the τ parameter on the behavior of the algorithm. As τ is increased we expect the computational cost to be reduced, but at the expense of the accuracy of the predictions in comparison to the regular KR. The results in Figure 5-5 agree with this expectation: the left hand graph shows that for 2-d, 3-d, 4-d and 5-d datasets (each with 10,000 points) the proportional error between cached kd-tree's and regular regression increases with τ. The right hand graph shows a corresponding decrease in computational cost.

Real datasets

In another experiment, we ran cached kd-tree's KR on data from several real-world and robot-learning datasets. Further details of the datasets can be found in [Maron et al, 94]. They include an industrial packaging process for which the slowness of prediction had been a reasonable cause for concern. Encouragingly, cached kd-tree's KR speeds up prediction by a factor of 100 with no discernible difference in prediction quality between cached kd-tree's and regular KR. This and other results are tabulated below. The costs and error values given are averages taken over an independent test set. Notably, the datasets with the least savings were **pool**, which had few data points, and **robot**, which was high dimensional.

Table 5-1: Real dataset test of cached kd-tree's kernel regression

Domain	*Dataset Size*	*Dim of Input*	*Regular KR Cost*	*Tree's KR Cost*	*Regular KR Err.*	*Tree's KR Error*
Energy	2144	5-d	2144	232.9	1.687	1.690
Package	32000	3-d	32000	289.0	6.07	6.09
Pool	213	3-d	213	50.7	2.125	2.123
Protein	4664	3-d	4664	383.8	1.036	1.106
Robot	871	14-d	871	225	6.354	6.976

High dimensional, non-uniform data

Our final experiment concerned the question of how well the method performs if the number of input variables is relatively large, but if the attributes are not independent. For example, a common scenario in robot learning is for the input vectors to be embedded on a lower-dimensional manifold. We performed two experiments, each with 9 inputs and 10,000 data points. In the first experiment, the components of the input vectors were distributed uniformly randomly. In the second experiment the input vectors were distributed on a non-linear 2-d manifold of the 9-d input space. The results were:

Table 5-2: Cached kd-tree's kernel regression for sub-manifold cases

	9-d uniform	*9-d inputs on 2-d manifold*
Regular KR cost	10,000	10,000
Cached kd-tree's KR cost	3,100	430
Regular KR mean testset error	13.07	1.06
Cached kd-tree's KR mean testset error	13.08	1.15

The results indicate that, as would be expected, the cost advantage of cached kd-tree's KR is not large (a factor of 3) for 9-d uniform inputs, but is far better if the inputs are distributed within a lower-dimensional space.

5.5 Summary

Kernel regression with the help of the cached kd-tree is preferable in case the application needs the following properties:

- Flexibility to work throughout the local/global spectrum.
- The ability to make predictions with different parameters without needing a retraining phase.

In addition, cached kd-tree's Kernel regression has a number of additional flexibilities. Once the kd-tree structure is built, it is possible to make different queries with not only different kernel widths K_w, but also different Euclidean distance metrics, with subsets of attributes ignored, or with some other distance metrics such as Manhattan. It is also possible to apply the same technique with different weight functions and for classification instead of regression.

Dimensionality is a weakness of cached kd-tree's Kernel regression. Diminishing returns set in above approximately 10 dimensions if the data points are distributed uniformly. This is an inherent problem for which no solution seems likely because uniform data points in high dimensions will have almost all data points almost exactly the same distance apart, and a useful notion of locality breaks down.

This chapter discussed an efficient implementation of kernel regression. In next chapter, we will apply exactly the same algorithm to locally weighted linear regression and locally weighted logistic regression, in which a prediction fits a local polynomial or a local logistic function to minimize the locally weighted sum squared error. The only difference is that each node of the kd-tree stores the regression design matrices of all points below it in the tree. This permits fast prediction and also fast computation of confidence intervals and analysis of variance information.

Chapter 6

Using Kd-trees for Various Regressions

In last chapter, we discussed how to use kd-tree to make kernel regression more efficient. In fact, kd-tree can be used for other regressions, too. In this chapter, we will introduce how to apply it to speed up locally weighted linear regression and locally weighted logistic regression.

6.1 Locally Weighted Linear Regression

Linear regression can be used as a function approximator. Given a set of memory data points, known as *training* data points, a *global* linear regression finds a line with parameters such that the sum of the residual squares from the training data points to the line is minimized. In the example of Figure 6-1(a), each data point has one input and one output. A global linear regression finds a line,

$$\hat{y}(x) = \beta_0 + \beta_1 x$$

with β_0 and β_1, so that the sum of the residual squares is minimized, i.e.,

$$(\beta_o, \beta_1) = \arg\min \sum_{i=1}^{N} (y_i - \hat{y}(x_i))^2 = \arg\min \sum_{i=1}^{N} (y_i - \beta_o - \beta_1 x_i)^2$$

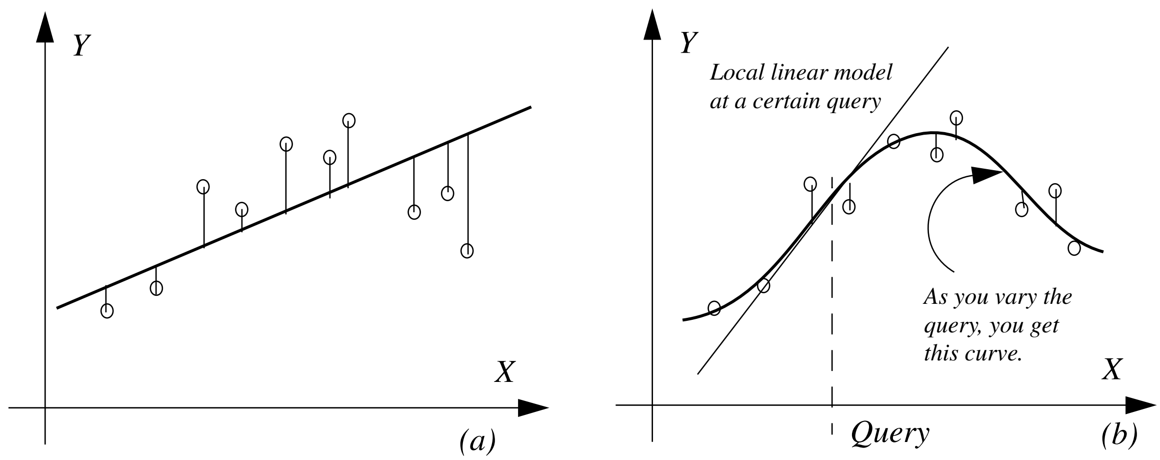


Figure 6-1: (a) A global linear regression (b) A locally weighted linear regression.

By global, we mean β_0 and β_1 are fixed for any possible x. Obviously, this linear function approximator would not work for any non-linear functions. That is the reason we have more interest in *locally weighted* linear regression.

Locally weighted regression assumes for any local region around a query point, x_q, the relationship between the input and output is linear. To construct the local function approximator, the local linear parameters can be approximated by minimizing the *weighted* sum of residual squares. For the example as shown in Figure 6-1(b), the weighted sum of residual squares is

$$\sum_{i=1}^{N} w_i^2 (y_i - \beta_o - \beta_1 x_i)^2$$

The weight, w_i, is usually a function of the Euclidean distance from the i'th training data points to the query, $\| x_q - x_i \|$. A popular form of the function is Gaussian.

After some algebra which requires no gradient descent, the linear parameters can be obtained directly by,

$$\hat{\beta} = (X^T W X)^{-1} (X^T W Y) \tag{6-1}$$

where X is a N-row M-column matrix, N is the number of the training data points in memory, M is the dimensionality of the input space plus 1. If the input vector of the k'th data point in memory is x_k, the k'th row of X is $(1, x_k^T)$. Y is a vector consisting of the training data points' outputs. W is a diagonal matrix, whose k'th element is the square of the weight of the k'th training data point, w_k^2.

6.2 Efficient locally weighted linear regression

As we have known in last section, the crucial thing to improve the efficiency of locally weighted linear regression is to speed up the calculation of $X^T WX$ and $X^T WY$. Since W is a diagonal matrix, $X^T WX$ and $X^T WY$ can be transformed as,

$$X^T W X = \sum_{i=1}^{N} w_i^2 x_i x_i^T \quad \text{and} \quad X^T W Y = \sum_{i=1}^{N} w_i^2 x_i y_i$$

in which vector x_i corresponds to the i'th row of X, and y_i is the i'th element of Y vector.

Recall that the kd-tree is a binary tree, the root of the tree covers the whole input space, hence contains all the training data points in memory. The root can be split into two nodes: the left node and the right node, each of them covers a partition of the input space. Furthermore, the left node can be split into another pair of nodes, so does the right node. Hence, in the second layer there are four nodes at most. Therefore, to calculate $X^T WX$ of all the memory data points, we can follow a recursion process,

$$(X^T WX)_{Root} = \sum_{i=1}^{N} w_i^2 x_i x_i^T = \sum_{i=1}^{N_{Left}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{Right}} w_i^2 x_i x_i^T$$

$$= (X^T WX)_{Left} + (X^T WX)_{Right}$$

$$= \sum_{i=1}^{N_{LeftLeft}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{LeftRight}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{RightLeft}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{RightRight}} w_i^2 x_i x_i^T$$

$$= (X^T WX)_{LeftLeft} + (X^T WX)_{LeftRight} + (X^T WX)_{RightLeft} + (X^T WX)_{RightRight}$$

in which N is the total number of training data points in memory, the sum of N_{Left} and N_{Right}, as well as the sum of $N_{LeftLeft}$, $N_{LeftRight}$, $N_{RightLeft}$, and $N_{RightRight}$, are equal to N.

Hence, to calculate $X^T WX$ of the root, or any other node of the kd-tree, we can recursively sum its two children's $X^T WX$'s. A leaf's $X^T WX$ can be calculated according to the definition:

$$(X^T WX)_{Leaf} = \sum_{i=1}^{N_{Leaf}} w_i^2 x_i x_i^T$$

However, this recursion process does not bring us any gain in computational efficiency, because it still visits every training data point in memory. But sometimes we may be able to cutoff the computation at a node, if all the memory data points within this node have near-identical weights. In other words,

$$X^T WX)_{Node} = \sum_{i=1}^{N_{Node}} w_i^2 x_i x_i^T \approx \bar{w}_{Node}^2 \left(\sum_{i=1}^{N_{Node}} x_i x_i^T \right) = \bar{w}_{Node}^2 (X^T X)_{Node} \qquad (6\text{-}2)$$

If w_i, $i = 1, \ldots, N_{Node}$, are near identical.

This scenario happens for three reasons:

- All data points within the node are so far from the query vector, x_q, that their weights are near zeroes.
- All the data points are close together, providing no room for weight variation.
- The weight function varies negligibly over the partition of the input space covered by the current node.

Given a certain query, x_q, and a certain node, to judge if any of these situations happens, we can rely on the comparison of the lower bound and the upper bound of the weights of the memory data points within this node. Roughly speaking, if the difference between the upper bound and the lower bound is smaller than a threshold, then Equation 6-2 holds and the cutoff is permitted. Further discussion on the threshold will come latter in this section. To calculate the lower bound and the upper bound of the weights, recall that each node of the kd-tree corresponds to a hyper-rectangular partition of the input space, thus, given a query, x_q, it is straightforward to calculate the longest and the shortest distances from the query to the concerned hyper-rectangle. Because the weight function is a monotonic function of the distance, it is not difficult to calculate the lower bound and the upper bound of the weights based on the range of the distance.

Therefore, to calculate X^TWX for all the data points in memory, we can follow the recursive algorithm listed in Figure 6-2.

Similarly, we can efficiently calculate X^TWY. But be aware that we need to cache X^TX and X^TY into each node of kd-tree. When we build a kd-tree, we calculate X^TX and X^TY for each node, from the leaves in the bottom, upward to the root. Once this is done, the kd-tree is ready to handle any queries. When a query occurs, we follow the recursion algorithm in Figure 6-2, from

```
  calc_linear_XtWX(Node, Query)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. If ( Wmax - Wmin ) < Threshold
     Then
         Node->XtWX = 0.25 * (Wmax + Wmin)^2 * Node->XtX;
     Else
        (Node->Left)->XtWX = calc_XtWX(Node->left, Query);
        (Node->Right)->XtWX = calc_XtWX(Node->right, Query);
        Node->XtWX = (Node->Left)->XtWX + (Node->Right)->XtWX;
  3. Return result;
}
```

Figure 6-2: Using divide-and-conquer algorithm to calculate XtWX of a node.

the root downward to the leaves, to calculate $(X^TWX)_{Root}$, as well as $(X^TWY)_{Root}$, then we can do the locally weighted linear regression.

Concerning the threshold in Figure 6-2, a simple way is to assign a fixed one, ε, and see if W_{max} - W_{min} < ε. However, this is dangerous. Suppose a query is far away from all the memory data points, then even the root node of the kd-tree may satisfy $w_{max} - w_{min} < \varepsilon$, so that all the memory data points have the same weight, $0.5 \times (w_{max} + w_{min})$. This means that the prediction of the output of the query will be equal to the mean value of all the memory data points' outputs, i.e.,

$$\hat{y}_q = \left(\sum_{i=1}^{N} y_i\right) / N .$$

This may be wildly different from the non-approximate linear regression without kd-tree, which takes the prediction as an extrapolation of the linear function fitting those memory data points, referring to Figure 6-3,

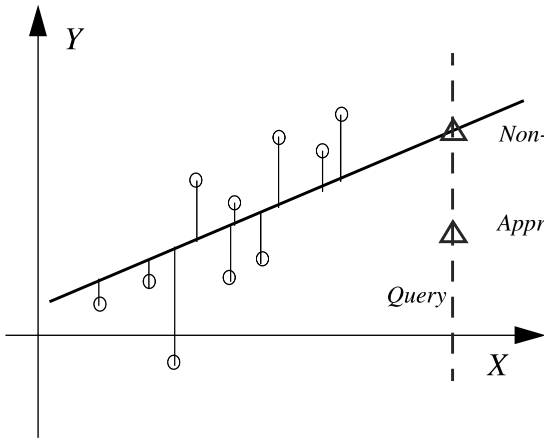


Figure 6-3: The danger of a wrong threshold of the cutoff condition.

This problem can be solved by setting ε to be a fraction of the total sum of weights involved in the regression: $\varepsilon = \tau \times \sum_{k=1}^{N} w_k$ for some small fraction τ. So we would then like to cutoff if and only if, $w_{max} - w_{min} < \tau \times \sum_{k=1}^{N} w_k$. But we do not know the value of $\sum_{k=1}^{N} w_k$ before we begin the prediction, and computing it would not be desirable (cost *O(N)*). Instead, we estimate a lower bound on $\sum_{k=1}^{N} w_k$. If, during the computation so far, we have accumulated sum-of-weights, w_{sofar}, and if currently we are visiting the *Node*'th node in the kd-tree and there are N_{Node} within this node, then,

$$w_{SoFar} + N_{Node} w_{min} \le \sum_{k=1}^{N} w_k .$$

Therefore, the improved cutoff condition is to judge if,

$$w_{max} - w_{min} < \tau(w_{SoFar} + N_{Node} w_{min}) . \qquad (6\text{-}3)$$

6.3 Technical details

There are several details which we summarize briefly here,

- To ensure numerical stability of this algorithm, all attributes must be pre-scaled to a hypercube centered around the origin.

- The cost of building the tree is $O(M^2N + N\log N)$, where M is the input space's dimensionality plus 1, and N is the number of data points in memory. It can be built lazily, (growing on-demand as queries occur) and data points can be added in $O(M^2 \times \text{Tree depth})$ time, though occasional rebalancing may be needed. The tree occupies $O(M^2N)$ space. Huge memory savings are possible if nodes with fewer than M data points are not split, but instead retain the data points in a linked list.

- Instead of always searching the left child first it is advantageous to search the node closest to x_q first. This strengthens the w_{SoFar} bound.

- Ball trees [Omohundro, 91] plays a similar role to kd-trees used for range searching, but it is possible that a hierarchy of balls, each containing the sufficient statistics of data points they contain, could be used beneficially in place of the bounding boxes we used.

- The algorithms can be modified to permit the k nearest neighbors of x_q to receive a weight of 1 each no matter how far they are from the query. This can make the regression more robust.

6.4 Empirical Evaluation

We evaluated five algorithms for comparison.

First of all, we examined prediction on a dataset ABALONE from UCI repository, with 10 inputs and 4177 data points; the task was to predict the number of rings in a shellfish. In these experiments we removed a hundred data points at random as a testset, and examined each algorithm performing a hundred predictions; all variables were scaled to [0..1], and a kernel width of 0.03 was used. As Table 6-2 shows, the **Regular** method took almost a second per prediction.

Table 6-1: Five linear regression algorithms

Regular	Direct computation of X^TWX as $\sum_{k=1}^{N} w_k^2 x_k x_k^T$.
Regzero	Direct computation of X^TX with an obvious and useful tweak. Whenever $w_k = 0$, do not bother with $O(M^2)$ operation of adding $w_k x_k x_k^T$.
Tree	The near-exact tree based algorithm. (we set $\tau = 10^{-7}$).
Approx.	The approximate tree-based algorithm with $\tau = 0.05$.
Fast	A wildly approximate tree-based algorithm with $\tau = 0.5$. This gives an extremely rough approximation to the weight function.

Regzero saved 20% of that. **Tree** reduced **Regular**'s time by 50%, producing identical predictions (shown by the identical mean absolute errors of **Regular**, **Regzero**, and **Tree**). The **Approx.** algorithm gives an eighty-fold saving compared with **Tree**, and the **Fast** algorithm is about three times faster still. What price do **Approx.** and **Fast** pay in terms of predictive accuracy? Compare the standard error of the dataset (2.65 if the mean value of the training data points' outputs was always given as the predicted value) against **Tree**'s error of 1.65, **Approx.**'s error of 1.67, and **Fast**'s error of 1.71. We notice a small but not insignificant penalty relative to the percentage variance explained.

Table 6-2: Costs and errors predicting the ABALONE dataset

	Regular	**Regzero**	**Tree**	**Approx.**	**Fast**
Millisecs per prediction	980	800	460	5.7	1.7
Mean absolute error	1.65	1.65	1.65	1.67	1.71

The above results are from one run on a testset of size 100. Are they representative? Table 6-3 should reassure that reader, containing averages and confidence intervals from 20 runs with different randomly chosen testsets. The bottom row shows that the error of **Approx.** and **Fast** relative to the **Regular** algorithm is confidently estimated as being small.

Table 6-3: Millisecs to do the predictions, errors of the predictions, and errors relative to Regular.

Algorithms:	Regular	Regzero	Tree	Approx.	Fast
Millisecs	982.0±2.5	814.±3.3	468.±0.8	6.00±0.2	1.70±0.04
Abs. Error Mean	1.534±0.062	1.534±0.062	1.534±0.062	1.536±0.061	1.556±0.063
Excess error compared w/ **Regular**	0±0	0±0	0±0	0.023±0.034	0.032±0.032

Table 6-4: Performance on 5 UCI datasets and one robot dataset. All use locally weighted linear regression with kernel width 0.03

		Regular	Regzero	Tree	Approx.	Fast
Heart, 3-d, 170 datapnts. StdErr. 0.43	Cost	42.16	32.95	21.23	18.93	14.12
	Error	0.27	0.28	0.28	0.28	0.28
Pool, 3-d, 153 datapnts. StdErr. 2.21	Cost	34.65	33.45	22.33	4.41	0.80
	Error	0.63	0.63	0.63	0.63	0.62
Energy, 5-d, 2344 datapnts. StdErr. 286.07	Cost	535.87	484.30	323.37	5.11	1.10
	Error	11.93	11.93	11.93	15.15	21.60
Abalone, 10-d, 4077 datapnts. StdErr. 2.66	Cost	964.00	806.00	469.00	5.80	1.70
	Error	1.65	1.65	1.65	1.67	1.71
MPG, 9-d, 292 datapnts. StdErr. 6.82	Cost	70.10	55.18	34.35	11.61	2.00
	Error	1.92	1.92	1.92	1.92	1.93
Breast, 9-d, 599 datapnts. StdErr. 0.3	Cost	143.40	126.18	59.88	13.82	6.21
	Error	0.03	0.03	0.03	0.03	0.02

We also examined the algorithms applied to a collection of five UCI-repository datasets and one robot dataset (described in [Atkeson et al., 97]). Table 6-4 shows results in which all datasets had the same local model: locally weighted linear regression with a kernel width of 0.03 on the unit-scaled input attributes. Table 6-5 shows the results on a variety of different local polyno-

Table 6-5: Same experiments, but with a variety of models. The models were selected by cross-validation depending on the specific domains.

		Regular	Regzero	Tree	Approx.	Fast
Heart, Kernel regress. kw = 0.015	Cost	37.86	25.84	14.32	13.42	0.50
	Error	0.22	0.22	0.22	0.22	0.24
Pool, Loc. wgted quad. regress., kw = 0.06	Cost	36.05	35.95	25.43	8.12	1.20
	Error	0.63	0.63	0.63	0.63	0.62
Energy, LW Quad. regress. without cross terms	Cost	546.48	356.12	202.29	25.53	1.60
	Error	6.12	6.12	6.12	6.03	7.50
Abalone, LW Linear regress. ignore 1 input.	Cost	958.90	717.34	203.91	2.35	1.40
	Error	1.33	1.33	1.33	1.33	1.34
MPG, Using all inputs but only has three in the dist. metrics	Cost	66.79	54.18	8.41	1.70	1.20
	Error	1.95	1.95	1.95	1.94	1.92
Breast, Only use five out of ten inputs.	Cost	44.06	43.96	2.20	2.20	0.50
	Error	0.01	0.01	0.01	0.01	0.02

mial models. The pattern of computational savings without serious accuracy penalties is consistent with our earlier experiment.

The above examples all have fixed kernel widths. There are datasets for which an adaptive kernel-width (dependent on the current x_q) are desirable. At this point, two issues arise: the statistical issue of how to evaluate different kernel widths (for example, by the confidence interval width on the resulting prediction, or by an estimate of local variance, or by an estimate of local data density) and the computational cost of searching for the best kernel width for our chosen criterion. Here we are interested in the computational issue and so we resort to a very simple criterion: the local weight, $\sum w_i$.

Table 6-6: Prediction-time optimization of kernel width.

Using fixed Kernel width		Using variable Kernel width		Using variable Kernel width Goal weight is 8.0		
Kernel width	Mean error	Goal weight	Mean error	Algorithm	Mean error	Millisecs per prediction
0.25000	0.41	64	0.19	**Regular**	0.104	2000
0.12500	0.24	32	0.13	**Regzero**	0.104	1400
0.06250	0.24	16	0.11	**Tree**	0.104	395
0.03125	0.22	8	0.10	**Approx.**	0.103	181
0.01562	0.29	4	0.10	**Fast**	0.107	165
0.00781	0.37	2	0.11			
0.00391	0.41	1	0.15			
0.00195	0.51	0.5	0.51			

We artificially generated a dataset with 2-dimensional inputs, for which a variable kernel width is desirable. When evaluated on a testset of 100 data points we saw that no fixed kernel width did better than a mean error of 0.20 (Table 6-6, first two columns). We chose the simplest imaginable adaptive kernel-width prediction algorithm: on each top level prediction make eight inner-loop predictions make eight inner-loop predictions, with the kernel widths $\{2^{-2}, 2^{-3}, ..., 2^{-9}\}$; then choose to predict with the kernel width that produces a local weight $\sum w_i$ closest to some fixed goal weight. For dense data a small kernel width will thus be chosen, and for sparse data the kernel will be wide. The results are striking: The middle two columns of Table 6-6 reveal that for a wide range of goal-weights a testset error of 0.10 is achieved. At the same time, as the rightmost three columns show, the approximate methods continue to win computationally.

6.5 Kd-tree for logistic regression

Recall in Chapter 5, locally weighted logistic regression is to approximate the parameter vector β in the following formula,

$$P(y_q = 1 | S_p, x_q) = \pi_q = \frac{1}{1 + \exp(-[1, x_q^T]\beta)} \tag{6-4}$$

To do so, we should follow the Newton-Raphson recursion:

$$\hat{\beta}_{(r+1)} = \hat{\beta}_{(r)} + (X^T W X)^{-1} X^T W e \tag{6-5}$$

Suppose there are N training data points in memory, each training data point consists of a d-dimensional input vector and a boolean output. X is a $N \times (1 + d)$ matrix. The i'th row of X matrix is $(1, x_i^T)$. And W is a $N \times N$ diagonal matrix, whose i'th element is $W_i = w_i^2 \pi'_i$, where π'_i is a scalar, which is the derivative value of π_i with respect to the current estimate of β at the query x_q:

$$\pi_i' = \left. \frac{\exp(-[1, x_i^T]\beta)}{\{1 + \exp(-[1, x_i^T]\beta)\}^2} \right|_{\beta = \hat{\beta}_{(r)}} \tag{6-6}$$

For example, when a training data point's input is $x_i = [2]$, while the current estimate of β is $[0.5, 1]^T$, then π_i' is equal to *0.07*. As mentioned above, the i'th element of W diagonal matrix is also decided by the weight, w_i, which is a function of the distance from the i'th training data point to the query x_q. The last item, e is the ratio of $y_i - \pi_i$ to π'_i, i.e. $e = (y_i - \pi_i)/\pi_i'$. Newton-Raphson starts from a random estimate of β, usually we assign $\hat{\beta}_{(0)}$ to be zero vector. Although it is not strictly proved, usually with no more than 10 loops, the recursive process comes to a satisfactory estimate of β.

Now, our task is that what information we should cache into the nodes of kd-tree, so that we can approximate X^TWX and X^TWe quickly without any significant loss of the accuracy. The most important characteristic of the cached information is that it must be independent from any specific query, because we want to exploit the same cached information to handle various queries.

Our solution is to cache $\sum_{i \in \text{Node}} x_i$, $\sum_{i \in \text{Node}} x_i x_i^T$ and $\sum_{i \in \text{Node}} y_i x_i$, which are expressed as $\mathbf{1}^T X$, $X^T X$, and $X^T Y$, too.

To calculate $(X^TWX)_{Node}$ of a particular kd-tree node, we can either do it precisely following its definition:

$$(X^T WX)_{Node} = \sum_{i \in \text{Node}} w_i^2 \pi_i' x_i x_i^T \tag{6-7}$$

where π_i' is the derivative value of the logistic function defined in Equation 6-6, w_i is the weight of the i'th data point with respect to the query.

When all the weights, w_i, $i \in Node$, are near identical, and so are the derivative values, π_i', $i \in Node$, we can approximate $(X^TWX)_{Node}$ as,

$$(X^T WX)_{Node} \approx \overline{w^2}\bar{\pi}' \sum_{i \in Node} x_i x_i^T = \overline{w^2}\bar{\pi}' X^T X \tag{6-8}$$

There are three scenarios that the weights, w_i, within a kd-tree node, are near identical, referring to Section 6-2. Hence, the cutoff condition and the threshold discussed in Section 6-2 should be employed for logistic regression, too. In other words, to make Equation 6-8 hold, the concerned kd-tree node should satisfy:

$$w_{max} - w_{min} < \tau(w_{SoFar} + N_{Node} w_{min}) \tag{6-9}$$

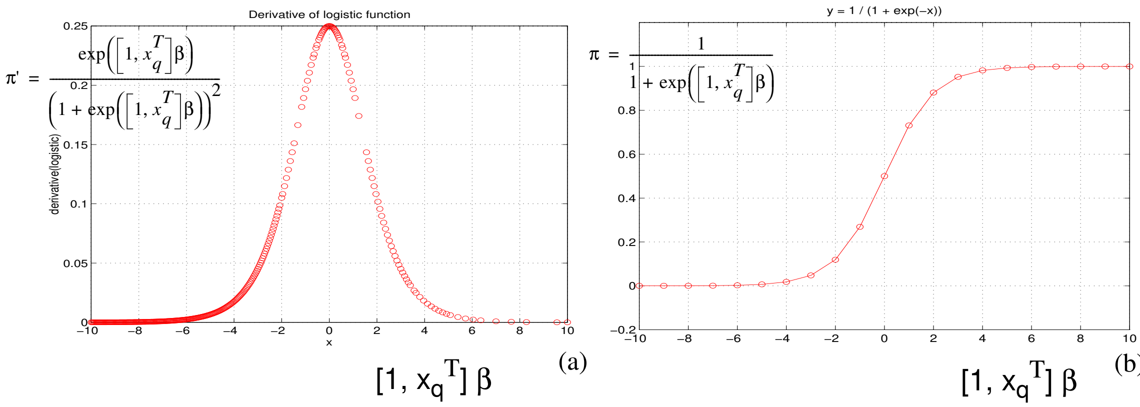


Figure 6-4: (a) The derivative function of logistic, which has symmetric two tails close to zero and a peak in the center. (b) The logistic function which is monotonic between 0 and 1.

To tell if the derivative values, π_i', $i \in Node$, do not differ too much, it looks that we can use a simple fixed threshold, ε_1:

$$\pi'_{max} - \pi'_{min} < \varepsilon_1$$

However, it is not easy to find the upper bound and lower bound of π_i'. Referring to Figure 6-4 (a) and (b), if Equation 6-10 holds,

$$\pi_{max} - \pi_{min} < \varepsilon_1 \tag{6-10}$$

the gap between π'_{max} and π'_{min} must be small, too. Since logistic function is monotonic, usually we can rely on the calculation of the logistic function values at the corners of the hyper-rectangle region in the input space represented by the kd-tree node, to find π_{max} and π_{min}.

Therefore, given a specific query x_q in conjunction with a certain estimate of β, to calculate $(X^TWX)_{Root}$ efficiently, we can recursively sum the two X^TWX's of the child nodes from the root on the top of the kd-tree downward to the leaves, in a way similar to that of locally weighted linear regression described in Figure 6-2. Sometimes the recursion can be cut off if both the two conditions in Equation 6-9 and 6-10 are satisfied, then the X^TWX of that node can be approximated as $\overline{w^2}\bar{\pi}'X^TX$. Thus, we need to cache X^TX into each node of the kd-tree before any query occurs.

More interestingly, notice that in Figure 6-4(a), the derivative of logistic function with respect to the scalar, $[1, x_q^T]\beta$, has a pair of long tails close to zero. That means, when the scalar $[1, x_q^T]\beta$ deviates from the origin, the derivative value, π', approaches zero quickly; and when the π'_{max} value of a kd-tree's node is near zero, it is unnecessary to calculate the X^TWX matrix of that node, because it must be a zero matrix according to $(X^TWX)_{Node} \le \overline{w^2}\bar{\pi}'_{max}(X^TX)_{Node}$.

Now, let's consider X^TWe of the training data points within a kd-tree's node, according to its definition,

$$\begin{aligned}(X^TWe)_{Node} &= \left(X^TW\left(\frac{Y-\pi}{\pi'}\right)\right)_{Node} = \sum_{i \in Node} x_i w_i^2 \pi_i' \left(\frac{y_i - \pi_i}{\pi_i'}\right) \\ &= \sum_{i \in Node} w_i^2 y_i x_i - \sum_{i \in Node} w_i^2 \pi_i x_i\end{aligned} \tag{6-11}$$

In case the following two conditions are satisfied: (1) all the individual weights, w_i, $i \in Node$, are near identical, (2) all the predictions, π_i, $i \in Node$, are near identical, X^TWe can be approximated as,

$$\begin{aligned}(X^TWe)_{Node} &\approx \bar{w}_{Node}^2 \sum_{i \in Node} y_i x_i - \bar{w}_{Node}^2 \bar{\pi}_{Node} \sum_{i \in Node} x_i \\ &= \bar{w}_{Node}^2 XY - \bar{w}_{Node}^2 \bar{\pi}_{Node} \mathbf{1}^T X\end{aligned} \tag{6-12}$$

Concerning the first cutoff condition related to the weights, we can use Equation 6-9 again to tell if the situation happens. Concerning the second cutoff condition about the prediction π_i, we can pre-define a fixed threshold, ε_2, to see if the following relationship is satisfied,

$$\pi_{max} - \pi_{min} < \varepsilon_2 \tag{6-13}$$

This cutoff condition is the same as Equation 6-10; furthermore, usually threshold ε_2 can be assigned to be equal to threshold ε_1. Referring to Figure 6-4(b), the function curve of π_i becomes flat when $[1, x_q^T]\beta$ deviates from the origin. Hence, there should be many chances for Equation 6-13 to hold. To find π_{max} and π_{min}, we can calculate the π values at the corners of the hyper-rectangular partition of the input space which the kd-tree node corresponds to.

In summary, to quickly approximate X^TWX and X^TWe, first of all, we should calculate $\mathbf{1}^TX$, X^TX, and X^TY for each kd-tree node respectively, and cache them into each node in conjunction with the number of data points within the node, **`num`**, **`split_d`** and **`split_v`**. When a query occurs, we follow a recursive algorithm similar to that of Figure 6-2, except that the cutoff conditions are different. The pseudo-code of the recursive algorithm for logistic regression is listed as Figure 6-5.

6.6 Empirical evaluation

In this section, we want to evaluate the performance of cached kd-tree's locally weighted logistic regression in two aspects: (1) how fast is it in comparison with the non-approximate locally weighted logistic regression? (2) how much does it lose in the accuracy?

We used again the four datasets from the UCI data repository which have been used in Section 4.4. Similar to the experiments we have done in Section 4.4, we shuffled the datasets five times each. Every time, we selected one third of the data points as the testing dataset, used the remaining two-thirds of the dataset as the training dataset. For every data point in the testing dataset, we assigned the input as a query, used locally weighted logistic regression based on the training dataset to predict its output, and compared the prediction with the real output of the data point to see if locally weighted logistic regression did correct job. We defined the error rate as the ratio of the number of wrong predictions to the number of total testing data points. Hence, the

```
  calc_logistic_XtWX(Node, Query, est_Beta, W_SoFar)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. Computer dev_Pi_min(Node, est_Beta), dev_Pi_max(Node, est_Beta);
  3. If ( Wmax - Wmin ) < τ * (W_SoFar + Node->num * Wmin)
         and ( Pi_max - Pi_min ) < ε
     Then    Node->XtWX = 0.125 * (Wmax + Wmin)^2
                     * (dev_Pi_max + dev_Pi_min) * Node->XtX;
     Else
        (Node->Left)->XtWX =
            calc_logistic_XtWX(Node->left, Query, est_Beta, W_SoFar);
        (Node->Right)->XtWX = calc_logistic_XtWX(Node->right, ...);
        Node->XtWX = (Node->Left)->XtWX + (Node->Right)->XtWX;
        Update W_SoFar to include 0.25 * (Wmax + Wmin)^2;
  4. Return Node->XtWX;
}
```

```
  calc_logistic_XtWe(Node, Query, est_Beta, W_SoFar)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. Computer Pi_min(Node, est_Beta), Pi_max(Node, est_Beta);
  3. If ( Wmax - Wmin ) < τ * (W_SoFar + Node->num * Wmin)
         and ( Pi_max - Pi_min ) < ε
     Then    Node->XtWe = 0.25 * (Wmax + Wmin)^2 * ( Node->XtY
                     - 0.5 * (Pi_max + Pi_min) * Node->1tX );
     Else
        (Node->Left)->XtWe =
             calc_logistic_XtWe(Node->left, Query, est_Beta, W_SoFar);
        (Node->Right)->XtWe = calc_logistic_XtWX(Node->right, ...);
        Node->XtWX = (Node->Left)->XtWe + (Node->Right)->XtWe;
        Update W_SoFar to include 0.25 * (Wmax + Wmin)^2;
  4. Return Node->XtWe;
}
```

Figure 6-5: Using the cached information of kd-tree to quickly approximate the XtWX and XtWe for locally weighted logistic regression.

lower the error rate, the more accurate the locally weighted logistic regression algorithms are. Since for every raw UCI dataset, we shuffled it for five times, thus we got five error rates. In Table 6-7, we listed the mean values of the error rates in conjunction with their standard deviations. In this way, we want to reassure the readers the representativeness of our results.

The first two rows of Table 6-7 are the performance of the regular locally weighted logistic regression without the help of cached kd-tree. As we expected, the error rates (in the second row) are exactly the same as those in Table 4-1. The first row recorded the milliseconds it took the regular locally weighted logistic regression to do one prediction for each datasets. As we have noticed, the computational cost varies a lot from 119.20 to 880.20. That is because the datasets have various dimensionalities of the input space which range from 6 to 34, also because the sizes of the training datasets differ a lot from 230 to 512.

Table 6-7: Performance on 4 UCI datasets

		Ionos. 234 datapnts 34 dim	**Pima** 512 datapnts 8 dim	**Breast** 191 datapnts 9 dim	**Bupa** 230 datapnts 6 dim
Non-approx.	Cost	880.20±5.63	263.20±1.48	119.20±7.85	548.10±4.47
	Error (%)	13.0±0.4	22.5±2.8	3.1±0.7	31.0±2.7
Kd-tree	Cost	906.20±11.19	5.40±0.13	8.28±1.09	5.03±0.04
	Error (%)	7.9±2.8	23.4±3.0	3.1±1.2	31.7±2.2
Cost gain		0.971	48.75	36.36	103.22
Accuracy loss		-38.93%	4.00%	0.0%	2.26%

The third and the fourth rows show the performance of the cached kd-tree's locally weighted logistic regression[1]. We expected that the improved logistic regression was much faster than the regular one while it did not lose too much in the accuracy. To make the comparison easier to follow, in the fifth row we calculated the multiplications of the costs of the regular logistic regression to those of the kd-tree's. As we see in the table, "Bupa" dataset, which is of low dimensionality with fairly small number of data points, benefited the most from the cached kd-tree: the efficiency improved more than 100 times. "Breast" dataset has a medium dimensionality and the number of data points is small. But still, the cached kd-tree improved the efficiency of locally weighted logistic regression 36 times. "Pima" consists of more data points, so it is not surprising that its multiplication is higher than that of "Breast"'s. "Ionos." is a special dataset because its dimensionality is high. In this case, cached kd-tree does not help to save the computational cost, instead it slightly enlarges the cost.

However, an interesting thing is that cached kd-tree improved the accuracy of locally weighted logistic regression applied to the "Ionos." dataset: the error rate dropped from 13.0% to 7.9%, in other words, the accuracy improved 38.93%, as shown in the last row in the table. Other datasets like "Pima" and "Bupa" did lose some accuracy, but not significantly.

6.7 Summary

In Chapter 5, we explored the use of kd-trees with some cached information, and we found improvements in the efficiency of kernel regression. In this chapter, we discussed how to cache different information into the kd-tree's node so as to improve the efficiency of locally weighted linear regression and locally weighted logistic regression. We found that for different memory-based learning, the cached information is different. Consequently, the cutoff thresholds should also be modified. Cached kd-trees can help both locally weighted linear regression and locally weighted logistic regression improve their computational efficiency, and at the same time not sacrifice their accuracy too much. This contribution is more significant when the size of the training dataset becomes larger. The limitation of cached kd-tree is that when the input space's

1. There are several control knobs for cached kd-tree's locally weighted logistic regression: Kernel width (*kw*), the fraction parameter for the weight's cutoff (τ), the fixed thresholds for the derivative and the prediction ($\varepsilon 1$ and $\varepsilon 2$). We found that the prediction accuracy is not very sensitive to $\varepsilon 1$ and $\varepsilon 2$, so we set both of them as 0.01.τ is also assigned to be 0.01. But Kernel width (*kw*) varies from dataset to dataset, tuned up by cross-validation.

dimensionality is higher than 10, a kd-tree cannot help to improve the efficiency too much. Further research needs to be done combat the curse of dimensionality.

Chapter 7

Feature Selection

Feature selection is not used in the system classification experiments, which will be discussed in Chapter 8 and 9. However, as an autonomous system, OMEGA includes feature selection as an important module.

7.1 Introduction

A fundamental problem of machine learning is to approximate the functional relationship *f( )* between an input $X = \{x_1, x_2, \dots, x_M\}$ and an output *Y*, based on a memory of data points, $\{X_i, Y_i\}$, $i = 1, \dots, N$, usually the X_i's are vectors of reals and the Y_i's are real numbers. Sometimes the output *Y* is not determined by the complete set of the input features $\{x_1, x_2, \dots, x_M\}$, instead, it is decided only by a subset of them $\{x_{(1)}, x_{(2)}, \dots, x_{(m)}\}$, where $m < M$. With sufficient data and time, it is fine to use all the input features, including those irrelevant features, to approximate the underlying function between the input and the output. But in practice, there are two problems which may be evoked by the irrelevant features involved in the learning process.

1. The irrelevant input features will induce greater computational cost. For example, using cached *k*d-trees as we discussed in last chapter, locally weighted linear regression's computational expense is $O(m^3 + m^2 \log N)$ for doing a single prediction, where *N* is the num-

ber of data points in memory and *m* is the number of features used. Apparently, with more features, the computational cost for predictions will increase polynomially; especially when there are a large number of such predictions, the computational cost will increase immensely.

2. The irrelevant input features may lead to overfitting. For example, in the domain of medical diagnosis, our purpose is to infer the relationship between the symptoms and their corresponding diagnosis. If by mistake we include the patient ID number as one input feature, an over-tuned machine learning process may come to the conclusion that the illness is determined by the ID number.

Another motivation for feature selection is that, since our goal is to approximate the underlying function between the input and the output, it is reasonable and important to ignore those input features with little effect on the output, so as to keep the size of the approximator model small. For example, [Akaike, 73] proposed several versions of model selection criteria, which basically are the trade-offs between high accuracy and small model size.

The feature selection problem has been studied by the statistics and machine learning communities for many years. It has received more attention recently because of enthusiastic research in data mining. According to [John et al., 94]'s definition, [Kira et al, 92] [Almuallim et al., 91] [Moore et al, 94] [Skalak, 94] [Koller et al, 96] can be labelled as "filter" models, while [Caruana et al., 94] [Langley et al, 94]'s research is classified as "wrapped around" methods. In the statistics community, feature selection is also known as "subset selection", which is surveyed thoroughly in [Miller, 90].

The brute-force feature selection method is to exhaustively evaluate all possible combinations of the input features, and then find the best subset. Obviously, the exhaustive search's computational cost is prohibitively high, with considerable danger of overfitting. Hence, people resort

1. Shuffle the data set and split into a training set of 70% of the data and a testset of the remaining 30%.
2. Let j vary among feature-set sizes: $j = (0 , 1 , 2 , ... , m)$
   a. Let fs_j = best feature set of size j, where "best" is measured as the minimizer of the leave-one-out cross-validation error over the training set.
   b. Let $Testscore_j$ = the RMS prediction error of feature set fs_j on the test set.

   End of loop of (j).
3. Select the feature set fs_j for which the test-set score is minimized.

Figure 7-1: Cascaded cross-validation procedure for finding the best set of up to *m* features.

to greedy methods, such as forward selection. In this paper, we propose three greedier selection algorithms in order to further enhance the efficiency. We use real-world data sets from over ten different domains to compare the accuracy and efficiency of the various algorithms.

7.2 Cross Validation vs. Overfitting

The goal of feature selection is to choose a subset X_s of the complete set of input features $X = \{x_1, x_2, ..., x_M\}$ so that the subset X_s can predict the output Y with accuracy comparable to the performance of the complete input set X, and with great reduction of the computational cost.

First, let us clarify how to evaluate the performance of a set of input features. In this chapter we use a very conservative form of feature set evaluation in order to avoid overfitting. This is important. Even if feature sets are evaluated by testset cross-validation or leave-one-out cross validation, an exhaustive search of possible feature-sets is likely to find a misleadingly well-scoring feature-set by chance. To prevent this, we use the *cascaded cross-validation* procedure in Figure 7-1, which selects from increasingly large sets of features (and thus from increasingly large model classes). The score for the best feature set of a given size is computed by an independent cross-validation from the score for the best size of feature set.

Two notes about the procedure in Figure 7-1: First, the choice of 70/30 split for training and testing is somewhat arbitrary, but is empirically a good practical ratio according to more detailed experiments. Second, note that Figure 7-1 does not describe how we search for the best feature set of size j in Step 2a. This is the subject of Section 7-3.

To evaluate the performance a feature selection algorithm is more complicated than to evaluate a feature set. This is because in order to evaluate an algorithm, we must first ask the algorithm to find the best feature subset. Second, to give a fair estimate of how well the feature selection algorithm performs, we should try the first step on different datasets. Therefore, the full procedure of evaluating the performance of a feature selection algorithm, which is described in Figure 7-2, has two layers of loops. The inner loop is to use an algorithm to find the best subset of features. The outer loop is to evaluate the performance of the algorithm using different datasets.

7.3 Feature selection algorithms

In this section, we introduce the conventional feature selection algorithm: forward feature selection algorithm; then we explore three greedy variants of the forward algorithm, in order to improve the computational efficiency without sacrificing too much accuracy.

7.3.1 Forward feature selection

The forward feature selection procedure begins by evaluating all feature subsets which consist of only one input attribute. In other words, we start by measuring the Leave-One-Out Cross Validation (LOOCV) error of the one-component subsets, $\{X_1\}, \{X_2\}, ..., \{X_M\}$, where M is the input dimensionality; so that we can find the best individual feature, $X_{(1)}$.

1. Collect a training data set from the specific domain.
2. Shuffle the data set.
3. Break it into *P* partitions, (say $P = 20$)
4. For each partition ($i = 0, 1, ..., P-1$)
    a. Let *OuterTrainset(i)* = all partitions except *i*.
    b. Let *OuterTestset(i)* = the *i*'th partition
    c. Let *InnerTrain(i)* = randomly chosen 70% of the *OuterTrainset(i).*
    d. Let *InnerTest(i)* = the remaining 30% of the *OuterTrainset(i)*.
    e. For $j = 0, 1, ..., m$
        Search for the best feature set with *j* components, fs_{ij}.using leave-one-out on *InnerTrain(i)*
        Let $InnerTestScore_{ij}$ = RMS score of fs_{ij} on *InnerTest(i).*
       End loop of (*j*).
    f. Select the fs_{ij} with the best inner test score.
    g. Let $OuterScore_i$ = RMS score of the selected feature set on *OuterTestset(i)*
   End of loop of (*i*).
5. Return the mean Outer Score.

Figure 7-2: Full procedure for evaluating feature selection of up to *m* attributes.

Next, forward selection finds the best subset consisting of two components, $X_{(1)}$ and one other feature from the remaining $M - 1$ input attributes. Hence, there are a total of $M - 1$ pairs. Let's assume $X_{(2)}$ is the other attribute in the best pair besides $X_{(1)}$.

Afterwards, the input subsets with three, four, and more features are evaluated. According to forward selection, the best subset with *m* features is the *m*-tuple consisting of $X_{(1)}, X_{(2)}, ..., X_{(m)}$, while overall the best feature set is the winner out of all the *M* steps. Assuming the cost of a LOOCV evaluation with *i* features is $C(i)$, then the computational cost of forward selection searching for a feature subset of size *m* out of *M* total input attributes will be

$$MC(1) + (M-1)C(2) + ... + (M-m+1)C(m) .$$

For example, the cost of one prediction with one-nearest-neighbor as the function approximator, using a kd-tree with *j* inputs, is $O(j \log N)$ where *N* is the number of datapoints. Thus, the cost of computing the mean leave-one-out error, which involves N predictions, is $O(j\ N \log N)$. And so the full cost of feature selection using the above formula is $O(m^2\ M\ N \log N)$.

To find the overall best input feature set, we can also employ exhaustive search. Exhaustive search begins with searching the best one-component subset of the input features, which is the same in the forward selection algorithm; then it goes to find the best two-component feature subset which may consist of *any* pairs of the input features. Afterwards, it moves to find the best triple out of all the combinations of any three input features, etc. It is straightforward to see that the cost of exhaustive search is the following:

$$MC(1) + \binom{M}{2}C(2) + \dots + \binom{M}{m}C(m)$$

Compared with the exhaustive search, forward selection is much cheaper.

However, forward selection may suffer because of its greediness. For example, if $X_{(1)}$ is the best individual feature, it does not guarantee that either $\{X_{(1)}, X_{(2)}\}$ or $\{X_{(1)}, X_{(3)}\}$ must be better than $\{X_{(2)}, X_{(3)}\}$. Therefore, a forward selection algorithm may select a feature set different from that selected by exhaustive searching. With a bad selection of the input features, the prediction $\hat{Y}_q$ of a query $X_q = \{x_1, x_2, \dots, x_M\}$ may be significantly different from the true Y_q.

7.3.2 Three Variants of Forward Selection

In this subsection, we will investigate the following two questions based on empirical analysis using real world datasets mixed with artificially designed features.

1. How severely does the greediness of forward selection lead to a bad selection of the input features?

2. If the greediness of forward selection does not have a significantly negative effect on accuracy, how can we modify forward selection algorithm to be greedier in order to

improve the efficiency even further?

We postpone the first question until the next section. In this chapter, we propose three greedier feature selection algorithms whose goal is to select no more than m features from a total of M input attributes, and with tolerable loss of prediction accuracy.

Super Greedy Algorithm

Do all the 1-attribute LOOCV calculations, sort the individual features according to their LOOCV mean error, then take the m best features as the selected subset. We thus do M computations involving one feature and one computation involving m features. If nearest neighbor is the function approximator, the cost of super greedy algorithm is $O((M + m) N \log N)$.

Greedy Algorithm

Do all the 1-attribute LOOCVs and sort them, take the best two individual features and evaluate their LOOCV error, then take the best three individual features, and so on, until m features have been evaluated. Compared with the super greedy algorithm, this algorithm may conclude at a subset whose size is smaller than m but whose inner testset error is smaller than that of the m-component feature set. Hence, the greedy algorithm may end up with a better feature set than the super-greedy one does. The cost of the greedy algorithm for nearest neighbor is $O((M + m^2) N \log N)$.

Restricted Forward Selection (RFS)

1. Calculate all the 1-feature set LOOCV errors, and sort the features according to the corresponding LOOCV errors. Suppose the features ranking from the most important to the least important are $x_{(1)}, x_{(2)}, \ldots, x_{(M)}$.
2. Do the LOOCVs of 2-feature subsets which consist of the winner of the first round, $X_{(1)}$, along with another feature, either $X_{(2)}$, or $X_{(3)}$, or any other one until $X_{(M / 2)}$. There are

$M/2$ of these pairs. The winner of this round will be the best 2-component feature subset chosen by RFS.

3. Calculate the LOOCV errors of $M/3$ subsets which consist of the winner of the second round, along with the other $M/3$ features at the top of the remaining rank. In this way, RFS will select its best feature triple.
4. Continue this procedure, until RFS has found the best *m*-component feature set.

5. From Step 1 to Step 4, RFS has found *m* feature sets whose sizes range from *1* to *m*. By comparing their LOOCV errors, RFS can find the best overall feature set.

The difference between RFS and conventional Forward Selection (FS) is that at each step to insert an additional feature into the subset, FS considers all the remaining features, while RFS only tries a part of them which seem more promising. The cost of RFS for nearest neighbor is *O(M m N log N)*.

For all these varieties of forward selection, we want to know how cheap and how accurate they are compared with the conventional forward selection method. To answer these questions, we resort to experiments using real world datasets.

7.4 Experiments

In this section, we compare the greedy algorithms with the conventional methods empirically. We run ten experiments; for each experiment, we try two datasets with different input dimensionalities; and for each dataset, we use three different function approximators.

To evaluate the influence of the greediness on the accuracy and efficiency of the feature selection process, we use twelve real world datasets from StatLib/CMU and UCI's machine learning data repository. These datasets come from different domains, such as biology, sociology, robotics, etc. The datasets each contain 62 to 1601 points, and each point consists of an input vector and a scalar output. The dimensionality of the input varies from 3 to 13. In all of these examples we set *m* (the maximum feature set size) to be 10.

Table 7-1: Preliminary comparison of ES vs. FS

Domain (dim)	*20Fold Mean Errors*			*Time Cost*			*Selected Features*	
	ES	*FS*	*ES / FS*	*ES*	*FS*	*ES / FS*	*ES*	*FS*
Crab (7)	0.415	0.469	0.885	35644	522	68.28	A,F,G	A,E
Halibut (7)	57.972	52.267	1.109	61759	713	86.62	B,C,G	A,D,E,G
Irish (5)	0.863	0.905	0.954	138088	1142	120.91	A,C,E	A,D
Litter (3)	0.780	0.868	0.899	4982	117	42.58	A,B,C	A,B,C

Our first experiment demonstrates that Exhaustive Search (ES) is prohibitively time-consuming. We choose four domains with not-too-large datasets and limited input dimensionality for this test. Referring to Table 7-1, even for these easy cases, ES is far more expensive than the Forward Selection algorithm (FS), while it is not significantly more accurate than FS. However, the features selected by FS may differ from the result of ES. That is because some of the input features are not mutually independent.

Our second experiment investigates the influence of greediness. We compare the three greedier algorithms, Super Greedy, Greedy and Restricted Forward Selection (RFS), with the conventional FS in three aspects:(1) The probabilities for these algorithms to select any useless features, (2) The prediction errors using the feature set selected by these algorithms, and (3) The time cost for these algorithms to find their feature sets.

For example, if a raw data file consists of three input attributes, *U*, *V*, *W* and an output *Y*, we generate a new dataset consisting of more input features, *U*, *V*, *W*, *cU, cV*, *cW*, R_1, R_2,..., R_{10}, and the output *Y*, in which *cU*, *cV* and *cW* are copies of *U, V* and *W* but corrupted with 20%

noise, while R_1 to R_{10} are independent random numbers. The chance that any of these useless features is selected can be treated as an estimation of the probability for the certain feature selection algorithm to make a mistake.

Table 7-2: Greediness comparison

Domain (dim)	*Funct. Apprx.*	*# Corrupt / Total Corrupts*				*# Noise / Total Noise*			
		Super	*Greedy*	*RFS*	*FS*	*Super*	*Greedy*	*RFS*	*FS*
Bodyfat (13)	Nearest	0.23	0.12	0.10	0.12	0.10	0.05	0.05	0.06
	LocLin	0.31	0.08	0.17	0.18	0.00	0.00	0.05	0.20
	GlbLin	0.31	0.23	0.15	0.00	0.00	0.00	0.00	0.40
Boston (13)	Nearest	0.23	0.19	0.21	0.17	0.20	0.20	0.23	0.35
	LocLin	0.15	0.15	0.12	0.15	0.30	0.30	0.30	0.33
	GlbLin	0.15	0.12	0.15	0.23	0.40	0.30	0.30	0.40
Crab (7)	Nearest	0.29	0.29	0.29	0.29	0.30	0.13	0.17	0.20
	LocLin	0.29	0.14‘	0.21	0.21	0.40	0.40	0.20	0.15
	GlbLin	0.29	0.14	0.29	0.24	0.40	0.30	0.15	0.17
Halibut (7)	Nearest	0.57	0.57	0.14	0.43	0.10	0.10	0.10	0.10
	LocLin	0.43	0.21	0.04	0.24	0.20	0.10	0.10	0.20
	GlbLin	0.36	0.29	0.00	0.14	0.25	0.10	0.20	0.10
Irish (5)	Nearest	0.60	0.60	0.00	0.00	0.20	0.20	0.10	0.10
	LocLin	0.40	0.40	0.38	0.38	0.30	0.30	0.15	0.25
	GlbLin	0.60	0.60	0.30	0.40	0.30	0.30	0.40	0.25
Litter (3)	Nearest	0.67	0.33	0.33	0.33	0.30	0.00	0.05	0.07
	LocLin	0.67	0.33	0.33	0.33	0.30	0.00	0.05	0.07
	GlbLin	0.33	0.33	0.00	0.43	0.50	0.20	0.35	0.50

Table 7-2: Greediness comparison

Domain (dim)	*Funct. Apprx.*	*# Corrupt / Total Corrupts*				*# Noise / Total Noise*			
		Super	*Greedy*	*RFS*	*FS*	*Super*	*Greedy*	*RFS*	*FS*
Mpg (9)	Nearest	0.44	0.44	0.41	0.44	0.00	0.00	0.07	0.05
	LocLin	0.44	0.33	0.22	0.30	0.00	0.00	0.10	0.23
	GlbLin	0.33	0.28	0.22	0.17	0.00	0.00	0.20	0.20
Nursing (6)	Nearest	0.33	0.00	0.25	0.25	0.30	0.10	0.15	0.15
	LocLin	0.33	0.08	0.33	0.22	0.40	0.25	0.20	0.20
	GlbLin	0.33	0.25	0.33	0.25	0.40	0.35	0.20	0.30
Places (8)	Nearest	0.31	0.00	0.00	0.00	0.15	0.00	0.00	0.00
	LocLin	0.38	0.24	0.16	0.40	0.20	0.10	0.00	0.10
	GlbLin	0.25	0.25	0.23	0.31	0.35	0.15	0.15	0.25
Sleep (7)	Nearest	0.29	0.00	0.04	0.04	0.25	0.10	0.13	0.17
	LocLin	0.43	0.11	0.03	0.00	0.20	0.03	0.08	0.10
	GlbLin	0.26	0.21	0.26	0.29	0.40	0.15	0.18	0.40
Strike (6)	Nearest	0.33	0.17	0.17	0.17	0.30	0.00	0.03	0.03
	LocLin	0.58	0.00	0.00	0.00	0.15	0.00	0.00	0.05
	GlbLin	0.50	0.33	0.22	0.33	0.15	0.00	0.08	0.18
White-cell (13)	Nearest	0.15	0.15	0.08	0.23	0.40	0.20	0.15	0.25
	LocLin	0.15	0.04	0.02	0.02	0.04	0.10	0.27	0.27
	GlbLin	0.12	0.14	0.08	0.04	0.40	0.35	0.25	0.25
Mean over all twelve datasets	Nearest	0.37	0.27	0.17	0.21	0.23	0.10	0.11	0.13
	LocLin	0.38	0.18	0.17	0.20	0.24	0.13	0.13	0.18
	GlbLin	0.30	0.26	0.19	0.23	0.29	0.18	0.21	0.28
TOTAL	-	0.35	0.24	0.18	0.21	0.25	0.14	0.15	0.20

As we observe in Table 7-2, FS does not eliminate more useless features than the greedier competitors except the Super Greedy one. However, the greedier an algorithm is, the more easily it is confused by the relevant but corrupted features.

Since the input features may be mutually dependent, the different algorithms may find different feature sets. To measure the goodness of these selected feature sets, we calculate the mean 20-fold score. As described in Section 7-2, our scoring is carefully designed to avoid overfitting, so that the smaller the score, the better the corresponding feature set is. To confirm the consistency, we test the four algorithms in all the twelve domains from StatLib and UCI. For each domain, we apply the algorithms to two datasets. Both of the datasets are generated based on the same raw data file, but with different numbers of corrupted features and independent noise. And for each dataset, we try three function approximators, nearest neighbor (Nearest), locally weighted linear regression (LocLin) and global linear regression (GlbLin). For the sake of conciseness, we only list the ratios. If a ratio is close to 1.0, the corresponding algorithm's performance is not significantly different from that of FS. The experimental results are shown in Table 7-3. In addition, we also list the ratios of the number of seconds consumed by the greedier algorithms to that of FS.

First, we observe in Table 7-3 that the three greedier feature selection algorithms do not suffer great loss in accuracy, since the average ratios of the 20-fold scores to those of FS are very close to 1.0. In fact, RFS performs almost as well as FS. Second, as we expected, the greedier algorithms improve the efficiency. Super greedy algorithm (Super) is ten times faster than forward selection (FS), while greedy algorithm (Greedy) seven times, and the restricted forward selection (RFS) three times. Finally, restricted forward selection (RFS) performs better than the conventional FS in all aspects.

To further confirm our conclusion, we do the third experiment. This time, we insert more independent random noise and corrupted features to the datasets. For example, if the original data

Table 7-3: Greediness comparison

Domain (dim)	*Funct. Apprx.*	*20Fold() / 20Fold(FS)*			*Cost() / Cost(FS)*		
		Super	*Greedy*	*RFS*	*Super*	*Greedy*	*RFS*
Bodyfat (13)	Nearest	0.975	0.969	0.915	0.095	0.126	0.330
	LocLin	1.080	1.015	0.973	0.062	0.092	0.287
	GlbLin	0.984	0.981	0.966	0.084	0.109	0.247
Boston (13)	Nearest	0.876	0.872	0.881	0.105	0.145	0.389
	LocLin	1.091	1.091	0.969	0.058	0.080	0.270
	GlbLin	1.059	1.052	1.068	0.084	0.127	0.287
Crab (7)	Nearest	1.107	1.039	0.973	0.123	0.149	0.358
	LocLin	1.121	1.093	1.024	0.095	0.128	0.349
	GlbLin	1.123	1.101	0.957	0.079	0.116	0.319
Halibut (7)	Nearest	1.089	1.108	1.051	0.133	0.163	0.376
	LocLin	1.395	1.322	1.198	0.079	0.130	0.312
	GlbLin	1.073	1.018	1.022	0.079	0.137	0.273
Irish (5)	Nearest	1.132	1.072	0.954	0.127	0.171	0.343
	LocLin	1.039	0.979	0.984	0.086	0.137	0.316
	GlbLin	0.981	0.981	0.992	0.096	0.180	0.373
Litter (3)	Nearest	1.370	1.014	1.000	0.145	0.222	0.419
	LocLin	1.301	0.960	0.989	0.099	0.179	0.361
	GlbLin	0.886	0.902	0.930	0.111	0.179	0.410
Mpg (9)	Nearest	1.384	1.250	1.084	0.112	0.165	0.398
	LocLin	1.550	1.524	1.081	0.074	0.093	0.271
	GlbLin	1.295	1.317	1.014	0.086	0.142	0.298
Nursing (6)	Nearest	1.315	1.128	0.998	0.102	0.172	0.327
	LocLin	1.171	1.106	1.063	0.072	0.121	0.260
	GlbLin	1.044	1.043	1.002	0.092	0.137	0.267

Table 7-3: Greediness comparison

Domain (dim)	*Funct. Apprx.*	*20Fold() / 20Fold(FS)*			*Cost() / Cost(FS)*		
		Super	*Greedy*	*RFS*	*Super*	*Greedy*	*RFS*
Places (8)	Nearest	1.367	1.000	1.000	0.118	0.154	0.364
	LocLin	0.998	1.017	0.993	0.071	0.112	0.316
	GlbLin	1.041	1.044	1.064	0.091	0.130	0.265
Sleep (7)	Nearest	1.098	0.883	0.981	0.143	0.165	0.361
	LocLin	1.170	0.852	0.922	0.090	0.113	0.273
	GlbLin	0.918	0.925	1.026	0.096	0.122	0.276
Strike (6)	Nearest	1.142	0.952	1.000	0.161	0.178	0.424
	LocLin	1.172	0.987	1.003	0.068	0.108	0.293
	GlbLin	1.004	0.992	0.993	0.093	0.166	0.310
White-cell (13)	Nearest	0.854	0.718	0.906	0.100	0.138	0.288
	LocLin	1.259	0.821	0.931	0.077	0.088	0.254
	GlbLin	0.940	0.942	0.910	0.098	0.109	0.291
Mean over all twelve datasets	Nearest	1.142	1.001	0.978	0.122	0.163	0.365
	LocLin	1.196	1.064	1.011	0.077	0.115	0.296
	GlbLin	1.029	1.025	0.995	0.091	0.138	0.301
TOTAL	-	1.122	1.030	0.995	0.097	0.138	0.321

set consists of three input features, *{U,V,W}*, the new artificial data file contains *{U, cU, V, cV, cU * cV, W, cW, cV * cW,* R_1*,...,* R_{40}*}*. The results are listed in Table 7-4 and Table 7-5.

Comparing Table 7-2 with Table 7-4, we notice that with more input features, the probability for any corrupted feature to be selected remains almost the same, while that of independent noise reduces greatly. Comparing Table 7-3 with Table 7-5, with more input features, (1) the prediction accuracies of the feature sets selected by the variety of the algorithms are roughly

Table 7-4: Greediness comparison with more inputs

	Funct. Apprx.	# Corrupt / Total Corrupts				# Noise / Total Noise			
		Super	*Greedy*	*RFS*	*FS*	*Super*	*Greed y*	*RFS*	*FS*
Mean Values	Nearest	0.29	0.33	0.30	0.38	0.04	0.04	0.03	0.04
	LocLin	0.38	0.38	0.25	0.41	0.05	0.03	0.02	0.03
	GlbLin	0.38	0.25	0.29	0.16	0.05	0.05	0.08	0.07
TOTAL	-	0.35	0.32	0.28	0.32	0.05	0.04	0.04	0.05

Table 7-5: Greediness comparison with more inputs

	Funct. Apprx.	20Fold( ) / 20Fold(FS)			Cost( ) / Cost(FS)		
		Super	*Greedy*	*RFS*	*Super*	*Greedy*	*RFS*
Mean Values	Nearest	1.197	1.056	1.001	0.080	0.080	0.282
	LocLin	1.202	1.059	1.040	0.071	0.084	0.281
	GlbLin	1.032	1.026	0.998	0.079	0.104	0.294
TOTAL	-	1.144	1.047	1.013	0.077	0.088	0.286

consistent, because the 20fold scores in the two tables are almost the same; (2) the efficiency ratio of the greedier alternatives to FS is a little higher.

In summary, in theory the greediness of feature selection algorithms may lead to great reduction in the accuracy of function approximating, but in practice it does not happen quite often. The three greedier algorithms we propose in this paper improve the efficiency of the forward selection algorithm, especially for larger datasets with high input dimensionalities, without significant loss in accuracy. Even in the case the accuracy is more crucial than the efficiency, restricted forward selection is more competitive than the conventional forward selection.

7.5 Summary

In this chapter, we explore three greedier variants of the forward selection method. Our investigation shows that the greediness of the feature selection algorithms greatly improves the efficiency, while does not corrupt the correctness of the selected feature set so that the prediction accuracy using the selected features remains satisfactory. As an application, we apply feature selection to a prototype system of Chinese and Japanese handwriting recognition.

Chapter 8

Driving Simulation

The goal of this experiment is to distinguish different people's driving styles. The data was collected from five people using a simulator. The simulator, shown in Figure 8-1, was designed by M.C.Nechyba.

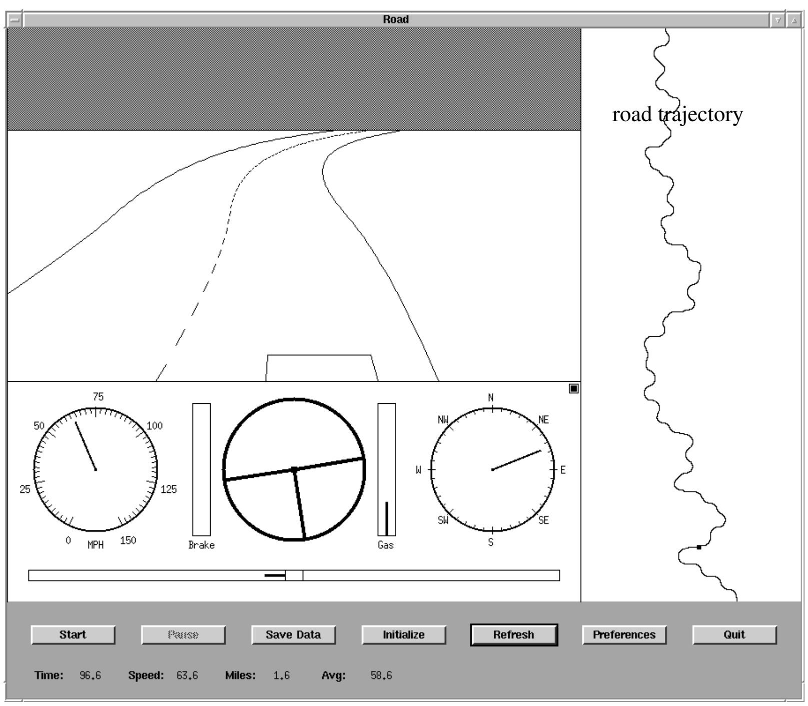


Figure 8-1: Driving simulator interface. (Courtesy M.C.Nechyba)

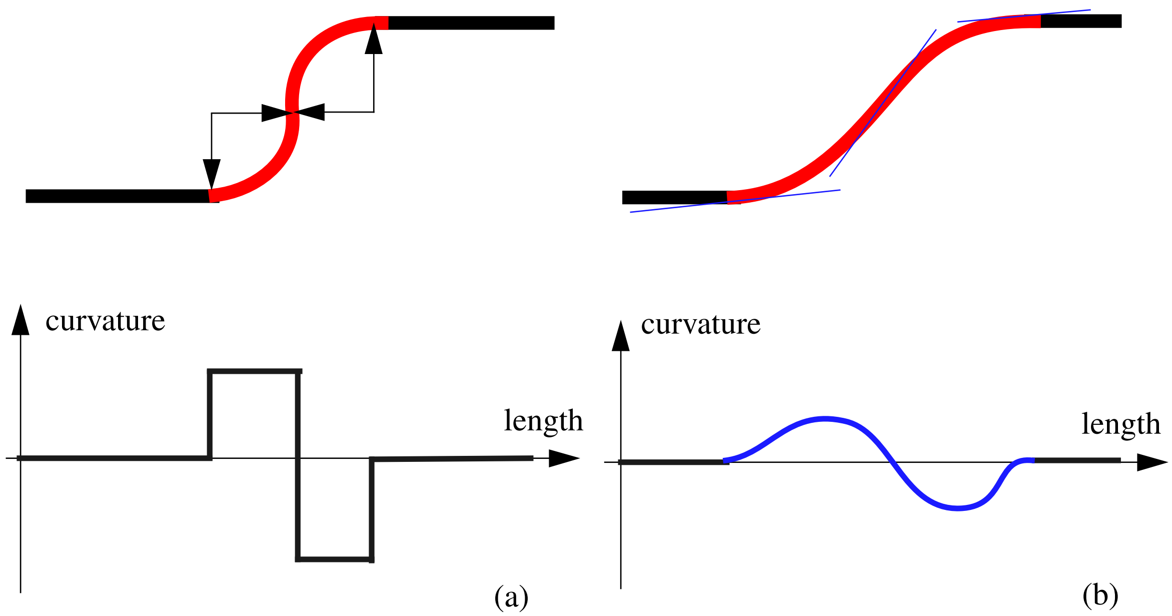


Figure 8-2: The simulator's road trajectory is generated in a way illustrated by (a), in which the curvature of the road changes abruptly. However, a high way in the real world is actually designed in the style of (b), in which the curvature changes smoothly.

8.1 Experimental data

The human operator has the full control over steering (horizontal mouse position), the brake (left mouse button) and the accelerator (right mouse button). Although the dynamics of the simulator strictly follows the form of some real vehicles [Nechyba et al, 98, (a) and (b)], the human drivers' behavior is quite different from the real one on the real roads. One reason is that the road trajectory of the simulator is generated as a sequence of straight-line segments and circular arcs, which differs from the real roads in the real world, illustrated by Figure 8-2.

We generated three road trajectories, each of them is around 20km. Five people were invited to operate on these three different roads after they had warmed up. The simulator took the record of the state of the vehicle and the environmental variables (described in details later) five times per second, while the simulator itself runs 50 Hz. Thus, we collected fifteen datasets, O_{ij}, i =

1, 2, ..., 5, j = 1, 2, 3, i represents the operators, and *j* corresponds to the different road trajectories.

The state and environmental variables are listed in the following table:

Table 8-1: State of vehicle and the environmental variables

	Description	*Time Delay (0.42 Seconds)*
v_ξ	The lateral velocity	6
v_η	The longitudinal velocity	6
ω	The angular velocity	6
(x, y)	The car-body-relative coordinates of the road median	10
δ	The user-applied steering angle	6
α	The user-applied longitudinal force on the front tires	6

If a human driver is viewed as a system, the input consists of the following information: (1) the current and recent vehicle states, $\{v_\xi(t-n_\xi), ..., v_\xi(t-1), v_\xi(t)\}$, $\{v_\eta(t-n_\eta), ..., v_\eta(t-1), v_\eta(t)\}$, $\{\omega(t-n_\omega), ..., \omega(t-1), \omega(t)\}$, where n_ξ, n_η, n_ω are the time delays. (2) previous control actions, $\{\alpha(t-n_\alpha), ..., \alpha(t-1), \alpha(t)\}$, $\{\delta(t-n_\delta), ..., \delta(t-1), \delta(t)\}$. (3) The visible view of the road ahead, $\{x(t+1), y(t+1), ..., x(t+n_r), y(t+n_r)\}$. The outputs should be $\delta(t+1)$ and $\alpha(t+1)$.

Notice that even for the same human driver, very similar inputs may lead to radically different outputs $\delta(t+1)$ and $\alpha(t+1)$, referring to [Nechyba, 98 (b)].

The time delays of the inputs (including n_r of the road median ahead) were decided based on our empirical experiments. Because of the time delays, the input dimensionality of a dynamic system tends to be very high, in this case, it is 50. The high dimensionality may have strong negative impact on the efficiency of both the information retrieval from memory and the classification process afterwards. For kernel regression, the computational cost is *O(Nd)*, where *N* is the memory size and *d* is the input space dimensionality. Even though we used kd-trees to re-organize the memory in order to speed up the information retrieval process, kd-tree performance is not satisfactory when the input dimensionality is too high.

Principal Component Analysis (PCA) [Jolliffe, 86] can be used to compress the input space if some of the inputs are linearly correlated. Notice that, theoretically there is no guarantee that PCA can shrink the dimensionality of the dataset in all cases especially when the input attributes are not linearly correlated; however in practice, PCA is a very popular method. In the simulation driving experiment, we used PCA to compress the input space from 50 dimensions to 3 dimensions, with only 7.2% loss of information.

8.2 Experimental results

As mentioned above, we collected fifteen datasets from five people driving on three road trajectories. We assigned one dataset to be a testing dataset; say, O_{21}, which is actually the dataset generated by the second driver along the first road. We did not tell OMEGA who was the real driver, and asked OMEGA to figure it out. To do so, OMEGA needed some *labeled* training datasets. In our experiments, we let those datasets collected from the other roads be the training datasets, i.e. O_{ik}, $i = 1, ..., 5$, $k = 2, 3$. By “labeled” we mean for each training dataset, OMEGA knew exactly who was the operator.

Using the OMEGA technique described in Chapter 2, we calculated the average of the negative log likelihood of each testing dataset with respect to all five human operators. Hence, for each testing dataset, we got five likelihood curves corresponding to the five possible drivers. OMEGA detected the hidden driver according to the tails of the likelihood curves: the lowest one indicates the most likely operator.

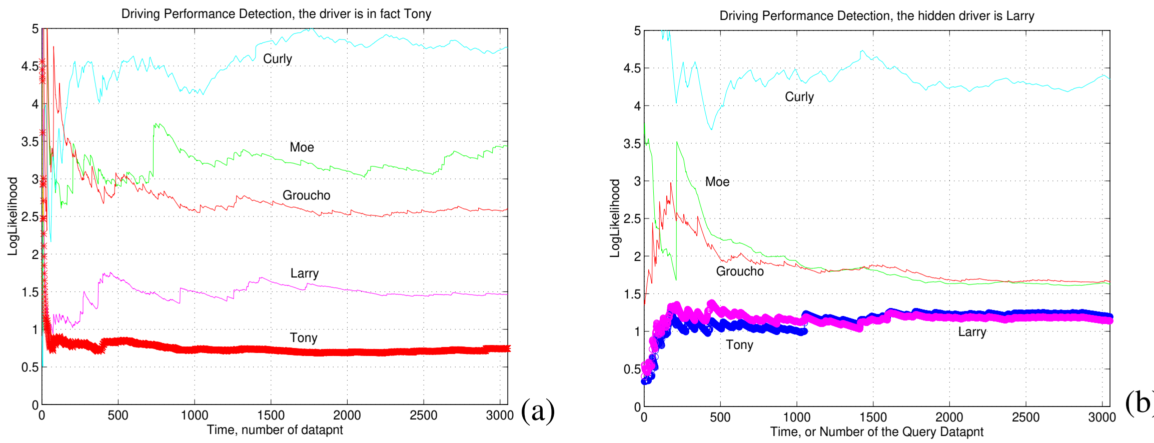


Figure 8-3: Simulation driving style OMEGA detection. (a) A correct case. (b) A sample of the confused cases. There are two confused cases out of the fifteen experiments, all others are correct.

There are in total fifteen testing datasets, OMEGA succeeded in detecting the hidden drivers correctly thirteen times. A typical correct case is demonstrated in Figure 8-3(a), which shows how OMEGA detected the underlying operator of a testing dataset, O_{11}. The horizontal axis is the number of data points in the testing dataset OMEGA has processed. The vertical axis is the average of the negative log likelihood. Tony's negative log likelihood curve is closest to the horizon, and it is remote from all other drivers' curves. Hence, Tony is the most likely operator of the testing dataset, O_{11}. At the early stage when only a few testing data points have been processed, the curves are not stable, but afterwards they become smoother and more stable.

Although OMEGA did not make any mistakes in the fifteen experiments, it was confused in two cases[1]. One of them is shown in Figure 8-3(b), in which the lowest curve does correspond the real driver, Larry; however, Tony's curve is too close to Larry's, so that OMEGA can hardly tell who is more likely to be the hidden driver between Larry and Tony.

1. To distinguish the confusing cases, we assign the significance level α to be 5%, referring to Chapter 2.

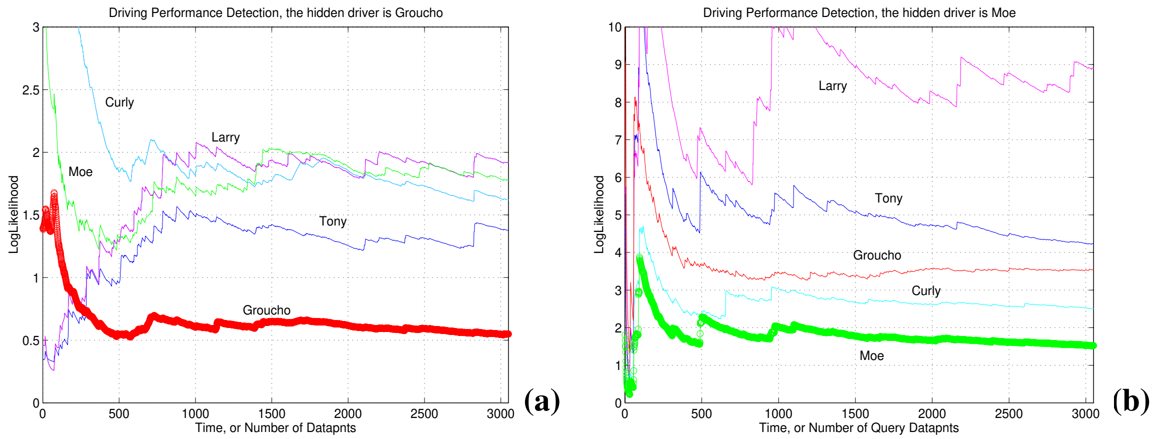


Figure 8-4: When some data points in the testing dataset are not consistent with a certain training dataset, the corresponding likelihood curve may look bumpy. If the data points are so unusual that there is no similar scenario in all the training datasets, then all the curves are bumpy, and roughly paralleling to each other, referring to (b).

As an on-line detection tool, OMEGA is capable of starting its job with very few data points. As expected, the precision is very bad. Thus, the likelihood curves look chaotic at first. But with more and more data come, the curves converge to be stable.

Sometimes the likelihood curves are bumpy, because the driver did something unusual compared with his behavior in the training datasets. After studying the datasets carefully, we notice that the abnormal behavior usually occurs when the curvatures of the road change rapidly, referring to Figure 8-2(a). If the human operator does not pay sufficient attention, he may drive off the road when the abrupt change of the curvature happens. Therefore, a careful driver's curve is smoother and more stable than others, illustrated by Figure 8-4(a). However, sometime the curvature changes so much and so suddenly that no one was able to keep his operation in a consistent manner. In those cases, all the curves are bumpy and roughly parallel to each other, referring to Figure 8-4(b).

Another interesting observation is that some people's curves tend to be close to each other, for example, Moe's and Groucho's. The short distances between their curves implies that their driving behaviors are close to each other in the experiments. But does it give any hint to the similarity of their personalities? This is an open question, but it is interesting to observe that Moe and Groucho do spend a lot of time together during weekends.

8.3 Comparison with other methods

Although OMEGA works well for detecting the hidden drivers in these simulation experiments, some legitimate questions are still opened, such as: is there any simpler method which can work as well or better?

8.3.1 Bayes classifier

Bayes classifier is a simple method which compares the features. Referring to Table 8-1, the state of the vehicle and the driver's action are the instantaneous velocity (including v_ξ and v_η), angular velocity ω, user applied steering angle δ and acceleration or brake force α. We treated the vehicle's state variables, the environmental variables, in conjunction with the control actions as the feature and applied Bayes classifier, with tuned-up parameters, to distinguish the five human operators. The result is shown in the first row of Table 8-2 :

Table 8-2: Comparison of OMEGA with other alternatives

	Correct	*Wrong*	*Confused*
Bayes classifier	6	6	10
HMM	13	0	2
Global linear	12	1	2
OMEGA	13	0	2

Obviously, feature-based Bayes classifier did not perform well. The reason are that: (1) features-based approach does not consider the mapping between the inputs and the outputs. (2)

some features are also influenced by the road conditions, besides the different human driving styles. (3) different human operators' feature values have a large overlapping region, exhibited in Table 8-3.

Table 8-3: Aggregate features of human simulation data (based on Nechyba's data)

	Velocity (v)	*Angular velocity* (ω)	*Steering angle* (δ)	*Longitudinal force* (α)
Tony	67.2 (12.6)	(0.205)	(0.097)	2.03 (3.86)
Larry	72.2 (7.8)	(0.193)	(0.072)	1.85 (2.37)
Moe	70.5 (7.9)	(0.198)	(0.074)	1.91 (3.25)
Curly	63.3 (10.1)	(0.175)	(0.056)	1.33 (1.88)
Groucho	73.2 (9.3)	(0.259)	(0.100)	2.33 (2.68)

The numbers in parentheses are the standard deviations. Since the mean values of angular velocities and steering angles depend on the specific road trajectories, only their standard deviations are listed in the table.

8.3.2 Hidden Markov Model

With rich mathematical fundamentals, the Hidden Markov Model [Rabiner, 89] is very useful in speech recognition. When we hear the sentence "I love you", in fact, our perception system recognizes the states [ai] [la] [v] [ju:] in sequence. The order is also important. However, due to the difference in emphasis, skipping, and pausing, the transitions among the states are not deterministic. Some states may last longer, others may be skipped. For the same example, it can be expressed in a different way: [ai] [pause] [la] [la] [v] [ju:], or "I, lo-ve you", which sounds more romantic than the plain tone. Therefore, HMM assumes the transitions among the different states are probabilistic instead of deterministic. To recognize a piece of speech, HMM relies on the approximation of those state transition probabilities.

Due to accents and/or personal styles, few people can precisely pronounce every word. Thus, the states (i.e. [ai], [la], [v], and [ju:]) are hidden underneath the stream of the sound signals. The mapping between the sound signals and the hidden states is not so simple as one-to-one; instead their relationship is also probabilistic. HMM is capable of approximating the probabilistic mapping between the sound signal and the states, as well as the transition probabilities.

Although HMM is very successful for speech recognition, one should be careful before using HMM as a general purpose time series recognizer. The reason is that HMM assumes the state transition probabilities are the most fundamental characteristic of a time series. And usually, the transition probabilities are assumed to be time-invariant.

[Nechyba, 98 (a)] applied HMM to distinguish different simulation driving styles. He did not separate the inputs and outputs, instead, he treated the states of the vehicle and the environmental variables equally as parts of observations. He assumed that the observations were stochastically decided by some hidden states. Although the physical meanings of those states were not clear, he conjectured that their transitions probabilities differed with different drivers. Therefore, given a unlabeled driving time series, Nechyba approximated a HMM which fit the time series well. Then he compared the new HMM with those in memory whose underlying drivers were known. Usually one HMM in memory is closer to the new one than the others are. The closest HMM in memory indicated the driver who is most likely to be generator of the unlabeled driving time series.

As Table 8-2 shows, the experimental performance of HMM is as good as that of OMEGA.

Why does HMM approach work in this domain? In our point of view, a hidden state is an abstract scenario of the state of the vehicle in conjunction with the environmental situation, and the human driver's control action. Facing a certain scenario, different drivers may give dissimilar control responses which lead to different new scenarios at the next time step. Thus, different drivers' diverse responses make the transition probabilities of his HMM distinguishable from those of others.

Therefore, we think the fundamental methodology of [Nechyba, 98(a)] is similar to that of OMEGA. There is no surprise that the accuracies of HMM and OMEGA are close to each other. While Table 8-2 gives a top-level comparison, Table 8-4 and Table 8-5 view the precision in depth. Each number in the tables is a probability of a testing dataset being generated by a certain operator. Each row corresponds to a specific testing data set, and the real operator is in the leftmost column. The other columns represent the five candidate drivers. The number in the (2,3)'th cell is the probability that a testing dataset, which was secretly generated by Larry, would be detected as the performance of Moe. Thus, the sum of the five probability values in each row is always 1.0. The number on the shaded diagonal is expected to be bigger than the others. And the bigger the diagonal number is, the better the detection system performs. Otherwise, the detection fails.

Comparing Table 8-4 and Table 8-5, we claim that HMM and OMEGA have similar accuracy in this simulation domain. No one is significant better than the other.

Table 8-4: Cross validation of OMEGA

	Tony	*Larry*	*Moe*	*Curly*	*Groucho*
Tony	0.677	0.139	0.020	0.031	0.133
Larry	0.243	0.441	0.014	0.129	0.173
Moe	0.037	0.001	0.836	0.114	0.012
Curly	0.060	0.030	0.272	0.570	0.068
Groucho	0.130	0.070	0.199	0.156	0.445

However, OMEGA outperforms HMM in other aspects, such as efficiency, data consumption, flexibility, robustness, etc., referring to Chapter 2.

Table 8-5: Cross validation of HMM (based on Nechyba's data)

	Tony	*Larry*	*Moe*	*Curly*	*Groucho*
Tony	0.425	0.157	0.217	0.154	0.047
Larry	0.202	0.538	0.116	0.101	0.043
Moe	0.212	0.077	0.429	0.172	0.110
Curly	0.154	0.073	0.180	0.413	0.180
Groucho	0.066	0.040	0.163	0.237	0.494

8.3.3 Global linear model

OMEGA is a non-parametric method, which means it does not need any assumption about the function relationship between the input and output. However, if we do know the function form, we have more options to detect the system. For example, linear system is simple and very popular in practice, which assumes the output is a linear function of the inputs. To detect a linear system, we can either follow the residual approach or compare the parameters of the linear functions.

- **Residual approach:** For each training dataset, we approximate the parameters of the linear function between the inputs and the outputs. Then, given a unlabeled testing dataset, we temporarily suppose it was generated by the first system. Through the first system's linear function, we predict the outputs corresponding to the inputs of the testing data points. There usually exist some residuals between the predicted outputs and the real outputs in the testing dataset. The smaller the residuals, the more likely the first system is the underlying system of the testing datasets. We enumerate all the candidate systems, the one with the smallest residuals is most likely to be the underlying system.

- **Parameter approach:** We can approximate the linear function's parameters of the testing dataset, as well as those of each training dataset. By comparing the parameters of the testing dataset with those of each training dataset, one by one, we can tell which training dataset is most similar to the testing dataset, hence, we detect the underlying operator of the unlabeled testing dataset.

It is interesting to find that the simulation driving domain *happens* to be linear. Referring to Table 8-2, the global linear approach performed satisfactorily compared with OMEGA and HMM. It did the correct detection job in most cases.

In our previous work [Deng et al, 97], we compared the driving behaviors of an identical human operator, but under two conditions: sober and intoxicated. We found that ARMA(4,4)[1] was a good model for the behaviors under both conditions. We approximated the ARMA parameters of the datasets under different sobriety conditions, and found the parameters of the intoxicated driving behavior deviated from the sober ones, shown in Figure 8-5. The drunken parameters were more widely scattered due to the fact that the human operator experienced the varying levels of intoxication.

8.4 Summary

In this chapter, we applied OMEGA to detect the driving style using simulation datasets. This domain is more complicated than the tennis one because driving is dynamic with feedback, and there are a large number of variables effecting the driver's control action. Hence, the pre-processing of the datasets is important. We used PCA technique to compress the input space.

OMEGA does very job in this domain, but is not significantly better than the other methods. However, OMEGA has other good properties: it is simple, it is easy to update the memory, it is

1. Auto Regression Moving Average (ARMA(p,q)) model [Brockwell et al, 91] is a popular linear time series model. (p,q) refers to the window sizes of its AR part and MA part.

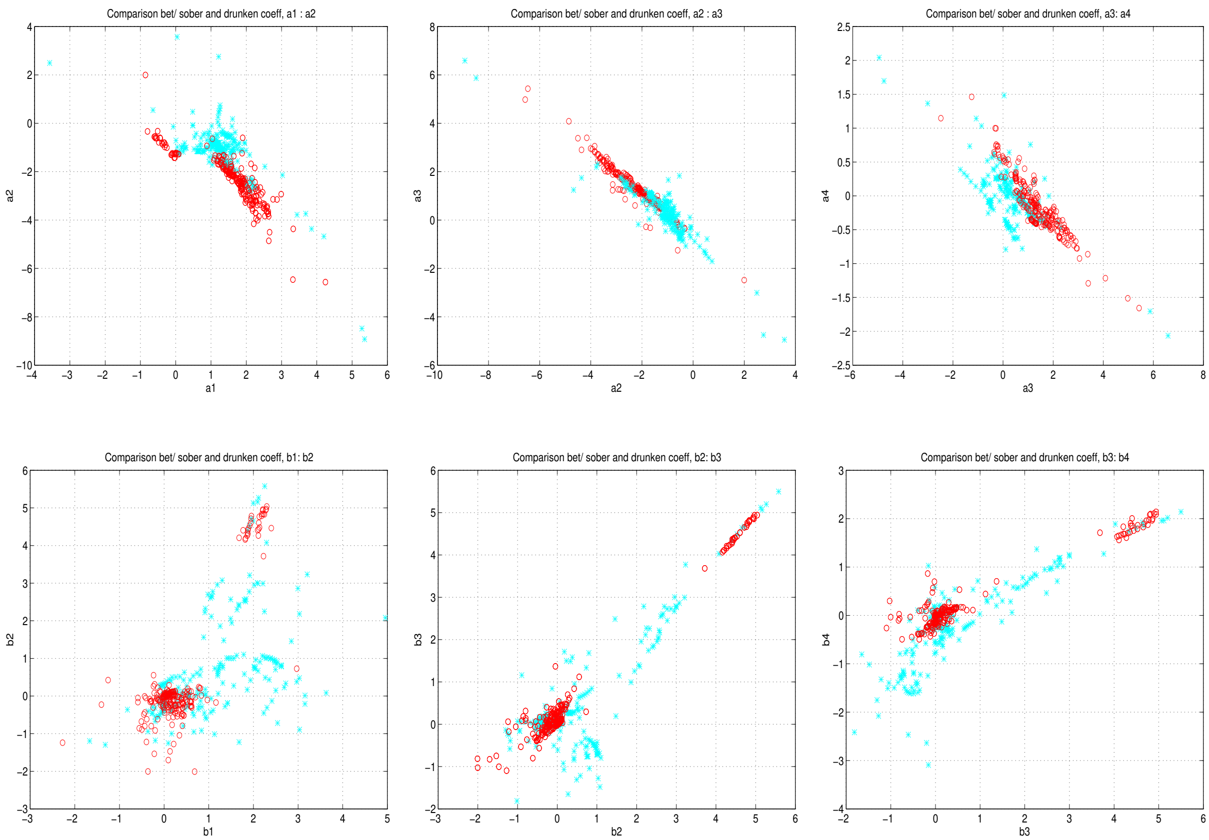


Figure 8-5: ARMA(4,4) parameters of the sober driving behavior are deviated from those of the intoxicated ones.

computational efficient, it consumes fewer data, and finally it is an on-line system, with more data involved in, it becomes more precise.

In next chapter, we will ask OMEGA to handle an even harder problem. We will see OMEGA performs more accurately than the other competing methods.

Chapter 9

Real World Driving

9.1 Data collection

The real world driving data were collected using the CMU Navlab 8 test vehicle, shown in Figure 9-1 [Pomerleau et al, 96]. A CCD camera is mounted on the windshield, underneath the rear-view mirror. This camera is used for lane tracking and vision based obstacle detection. A radar obstacle sensor is mounted behind the front license plate, and is used for detecting vehicles directly ahead and to the front-left/right. Two side sensors are mounted on the sides of the vehicles, near the rear. A single line laser range finder is mounted behind the rear bumper. It also has a Differential Global Positioning System receiver, which has a resolution of +/- 3-5m. Finally, a yaw-rate gyro is mounted in the rear, along with a tilt sensor. Hence, this vehicle allows us to take the time series records of the vehicle's states, the environmental situation, as well as the control actions. Notice that currently there is no sensor to measure the throttle of the engine in NavLab 8.

After eliminating not-important ones using our prior domain knowledge, the variables listed in Table 9-1 were used for the detection experiments. The shaded variable in the table, steering angle (ϖ), is the only output variable; the other output variable, the throttle of the gas in con-

Figure 9-1: NavLab's smart van. (Courtesy of Navlab, CMU).

junction with the brake force, is absent. All others, including the previous records of ϖ, were used as inputs. All the variables were taken record at a frequency between 14 Hz and 18 Hz.

Seven people were invited to drive the vehicle. They were selected from both genders and over a range of ages from twenty to fifty. All of them have valid U.S. driver's licenses, and have at least four years driving experience in the U.S., with no major traffic violations, accidents, or DUIs. The subjects were told only that we were interested in learning driving behaviors. Details were kept sketchy, to help avoid biasing the drivers' behaviors. They were not told how to drive, but the only instruction was to drive safely.

The operators were asked to drive from CMU to Grove City, a small town about 50 miles north of Pittsburgh, then back. "The route is primarily two lane (in each direction) highway driving, with short stretches of three lanes.” Each operator drove for over two hours round trip. “One concern is that the subjects most likely have never driven a Silhouette, or even a mini-van. A mini-van is large enough that it is hard to get a good feel for the boundaries and available space, particularly on the right hand side. Due to this, most drivers initially tended to hug the left side of the road. However, this effect seems to subside within a half hour or so of driving.”

Table 9-1: Real world driving variables

Variables	*Description*	*Variables*	*Description*
x_ξ	The lateral position	$1/s_F$	Inv. distance to the front obstacle
v_ξ	The lateral velocity	$1/s_{FL}$	Inv. dist. to the front-left obstacle
v_η	The longitudinal velocity	$1/s_{FR}$	Inv. dist. to the front-right obstacle
θ	Road Curvature	$1/s_B$	Inv. distance to the back obstacle
ϕ	Vehicle yaw	$1/s_{BL}$	Inv. dist. to the back-left obstacle
ϖ	Steering angle	$1/s_{BR}$	Inv. dist. to the back-right obstacle

Unlike the simulation cases discussed in last chapter, it seems to us that linear models are not appropriate for describing the real world driving behavior, because of the existence of traffic. For example, most drivers tend to take cut to the inside on a curvy road if there is no traffic, as illustrated by the dash curve in Figure 9-2. However, in case there is traffic, especially if there are other vehicles in the shortcut route, the drivers are more likely to stay in the middle of the lane. We can measure the distance from our vehicle to other vehicles in the curve, such as “*d*” in Figure 9-2. If there is no traffic in the curve, *d* goes to infinity or *1/d* is equal to zero. To decide to take the shortcut, the crucial issue is that *1/d* should be zero, however, it does not matter that *1/d* is equal to 0.25 or 0.32. Therefore, it is not proper to model the relationship among the vehicle’s lateral position, the road curvature and *1/d* as a linear function.

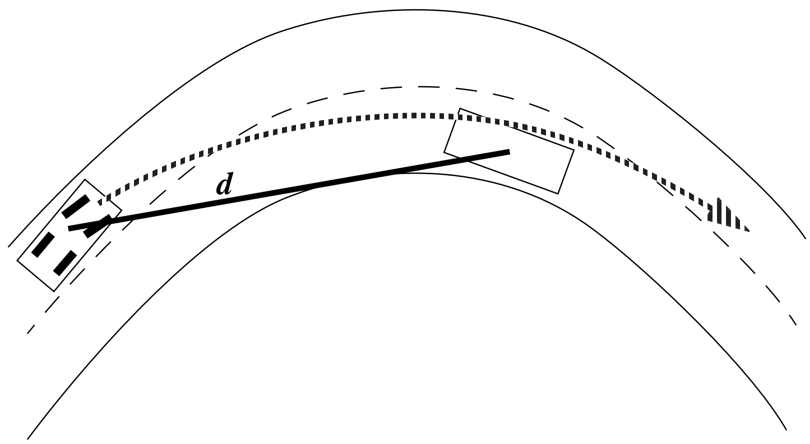


Figure 9-2: Driving in traffic may be non-linear. If there is no traffic, a driver tends to take a shortcut. Otherwise, he may stick to the same lane.

Based on empirical analysis, we found three seconds' time delay was sufficient for OMEGA to work properly. Hence, for each variable, we took its previous forty-eight $(3 \times 16(\mathrm{Hz}))$ records into account, except that for the road curvature, we took its forty-eight records ahead. For each variable, we used PCA to compress its dimensionality from forty-eight to three. Then we combined the twelve variables together, and used PCA again to reduce the dimensionality from thirty-six (12×3) to eight. The compression of the dimensionality is to make the further classification process feasible; however, as the price, we lost 17.8% information.

9.2 OMEGA result

Since there were seven drivers, and each one had two datasets, from Pittsburgh to Grove City and back, so that there were totally fourteen datasets: O_{ij}, $i = 1, ..., 7, j = 1, 2$. We can randomly select one dataset as a testing dataset, hide the real driver to OMEGA, and ask it to detect the driver to see if OMEGA is capable of detecting correctly.

To do so, OMEGA needs some training datasets. Define the datasets such that O_{ij} corresponds to journey *j* by driver *i*. If the testing dataset corresponds to a trip from Pittsburgh to Grove City, say O_{31}, where *1* refers to the route, *3* indicates the real driver. We assign the datasets collected

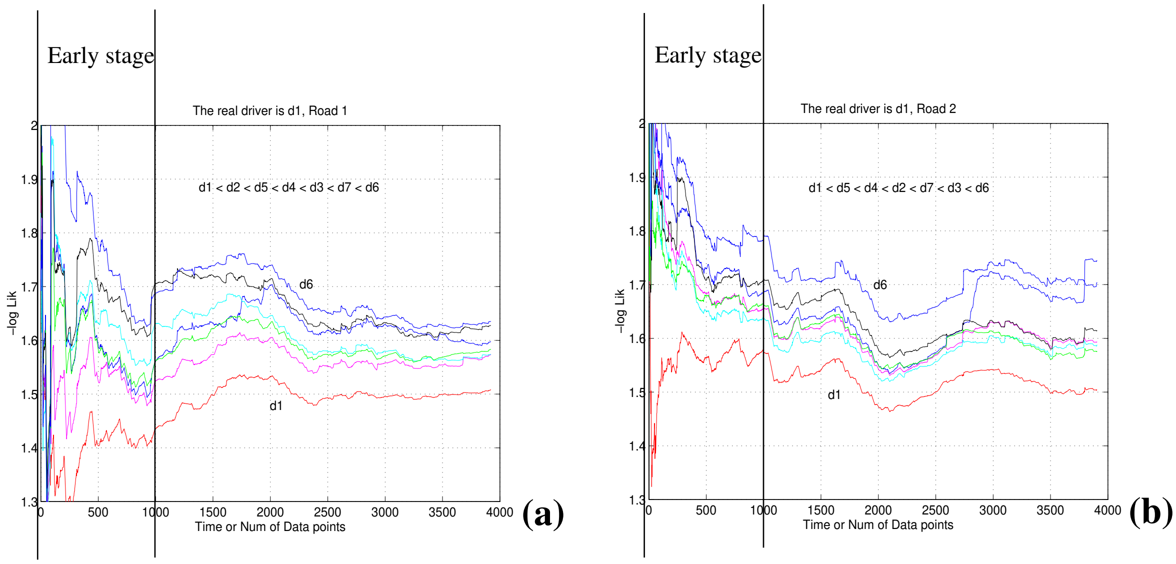


Figure 9-3: OMEGA detects the real world driving style. Two correct cases. (a) From Pittsburgh to Grove City, (b) From Grove City back to Pittsburgh.

on the way back from Grove City to Pittsburgh, as the training datasets. Thus, for each testing dataset, we have seven training datasets. For example, if the testing dataset is O_{31}, the training datasets will be O_{k2}, $k = 1, ..., 7$.

Since we can assign any dataset to be the testing dataset, totally we can do fourteen detection experiments. OMEGA succeeded in ten cases, failed three times and was confused once[1].

Referring to Figure 9-3, at the early stage of the detections, due to the insufficient number of data points involved in the analysis, the likelihood curves are unstable. With more and more data, the curves converge eventually. However, overall the curves look bumpier than those of the simulation experiments discussed in Chapter 8, referring to Figure 8-3. There are four possible reasons: (1) The real world datasets may be noisier than the simulation datasets because of the resolutions of the sensors. (2) We lost 17.8% information when we did the PCA pre-processing. (3) One of the two output variables, the throttle of the gas/brake is absent. (4) Although

1. Again, we assigned the significance level α to be 5%, referring to Chapter 2.

the real world driving datasets are large in size, the majority of their contents consist of nothing but very routine operations which are not helpful for distinguishing different people's driving styles.

Both Figure 9-3 (a) and (b) were generated by the same driver, "d1". Figure 9-3 (a) corresponds to the trip from Pittsburgh to Grove City, and Figure 9-3(b) corresponds to the way back. Comparing the early stages of Figure 9-3 (a) and (b), we notice that the curves in (a) were more chaotic than (b)'s. As a matter of fact, we observed the same phenomena happened to almost all the drivers, in other words, all drivers' initial performance were not so well-controlled as afterwards. In Table 9-2, we compare the standard deviations of the log likelihood of each driver's performance at the early stages of the trips from Pittsburgh to Grove City, with their counterparts on the ways back. Obviously, most operator's initial performance was significantly more disordered than the latter one, except that "d5" seems more ready to drive from the very beginning. These phenomena are supported by the observation mentioned in Section 9-1: "One concern is that the subjects most likely have never driven a Silhouette, or even a mini-van. ... However, this effect seems to subside within a half hour or so of driving."

Table 9-2: Standard deviations of the likelihood at the early stages.

	D1	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
Pgh - Grove	0.162	0.232	0.215	0.161	0.175	0.182	0.192
Grove - Pgh	0.120	0.095	0.159	0.118	0.163	0.053	0.047

As usual, the curve whose tail is the lowest indicates who is the real driver. In the correct cases, as in the examples of Figure 9-3, the lowest curves are underneath the others by large margins. Figure 9-4 (a) and (b) are examples of the confused cases and the incorrect ones. In fact, all the wrong cases are similar to Figure 9-4 (b): Although the real driver's curve is not the lowest one, it is lower than most others. That is to say, although OMEGA may make mistakes, the correct one is usually within the attention scope.

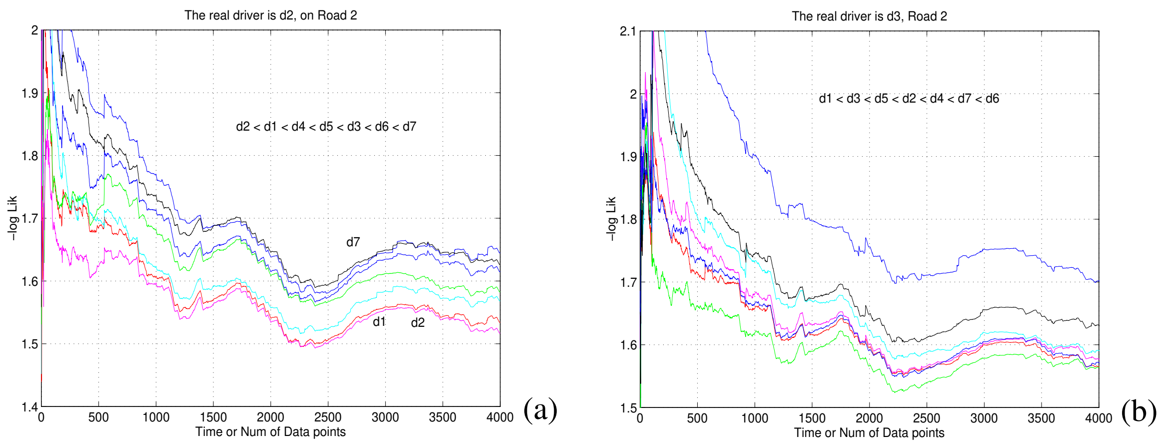


Figure 9-4: (a) A confused case. (b) A wrong case. Even as a wrong case, the real driver's curve is close to the lowest one.

9.3 Comparison with other methods

Table 9-3 is the comparison of OMEGA with other methods.

Table 9-3: Comparison of OMEGA with other methods

	Correct	*Wrong*	*Confused*
Global linear	4	7	3
HMM[a]	4	0	3
OMEGA	10	3	1

a. Nechyba has done only half of the experiments.

As we expected, the linear approach does not work properly due to the reason we discussed at the end of Section 9.1.

[Nechyba, 98, (a)]'s method did work in this domain. However, unlike OMEGA which separated the real driver from the others with a salient margin in log likelihood, [Nechyba, 98, (a)] could not make a decisive detection between two or more candidates.

Table 9-4: Cross-validation of OMEGA.[a]

	D1	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
D1	0.346	0.142	0.031	0.150	0.169	0.047	0.115
D2	0.213	0.247	0.096	0.155	0.129	0.074	0.086
D3	0.182	0.167	0.176	0.144	0.182	0.048	0.102
D4	0.159	0.119	0.059	0.337	0.126	0.087	0.113
D5	0.156	0.105	0.077	0.098	0.435	0.034	0.094
D6	0.124	0.161	0.123	0.124	0.185	0.174	0.108
D7	0.154	0.120	0.065	0.100	0.199	0.050	0.312

a. Using the datasets collected on the way from Pgh to Grove city as the training dataset, and using the datasets collected on the way back as the testing datasets.

While Table 9-3 gives a top-level comparison, Table 9-4 and 9-5 view the precision in depth. Each number in the tables is a probability of a testing dataset being generated by a certain operator. Each row corresponds to a specific testing data set, and the real operator is in the leftmost column. The other columns represent the seven candidate drivers. The testing datasets of Table 9-5 were collected on the way from Pittsburgh to Grove City, and the training datasets were collected on the way back. To be fair, so did those for Table 9-4. The number in the (2,3)'th cell is the probability that a testing dataset, which was secretly generated by the second driver, would be detected as the performance of the third driver. Thus, the sum of the seven probability values in each row is always 1.0. The number on the shaded diagonal is expected to be bigger than the others. And the bigger the diagonal number is, the better the detection system performs. Otherwise, the detection fails. We used 0.030 as a threshold to judge if the probabilities on the diagonal are significantly bigger than all the other six probabilities in the row. We notice in Table 9-4, OMEGA made wrong decisions twice. But when OMEGA made correct decisions, it was quite decisive. Conversely, HMM did not make any wrong decision, but when it came to the correct conclusion, for three times, the numbers on the diagonal could not be separated from the other six numbers in the rows by the 0.030 threshold.

Table 9-5: Cross-validation of HMM, [a]

	D1	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
D1	0.359	0.309	0.066	0.113	0.040	0.037	0.076
D2	0.108	0.226	0.123	0.193	0.098	0.090	0.162
D3	0.055	0.159	0.243	0.126	0.202	0.124	0.092
D4	0.106	0.196	0.102	0.216	0.097	0.123	0.160
D5	0.180	0.164	0.174	0.089	0.207	0.134	0.052
D6	0.053	0.127	0.087	0.208	0.105	0.232	0.188
D7	0.041	0.149	0.056	0.244	0.058	0.161	0.291

a. The same as the footnote of Table 9-4.

9.4 Summary

This chapter demonstrated that OMEGA is capable of detecting different systems accurately even in a complicated domain, where the conventional linear system identification approach, is not functional any more.

Chapter 10

Conclusion

10.1 Discussion

Question 1: *Usually a dynamic system has delays and feedback. Can OMEGA handle systems with infinite delays, and with elastic delays?*

OMEGA handles those systems with finite orders of delays. A system with elastic delays means that the order of delay varies from time to time. OMEGA is applicable to systems with elastic delays, if we know the range of the delays. We can assign the maximum order of the elastic delays to be the order for OMEGA. However, notice that with redundant order of delays, OMEGA may perform inefficiently.

Question 2: *Both OMEGA and Hidden Markov Models can handle time series. When should we use OMEGA instead of HMM?*

Some systems have hidden states, and the observable input and/or output of the systems are the manifestation of the hidden states. Hidden Markov Models are good at modeling the hidden states and their transition relationships. Conversely, OMEGA analyzes the complicated distribution of the input and output. To some extent, OMEGA may be capable of handling systems with hidden states. However, in case there are no hidden states, OMEGA will perform better.

For example, in the driving domain, because of different road conditions and traffic conditions, the distribution of the input and output tends to be very complicated. However, it seems to us that probably there are not too many hidden states standing between the input and output. Therefore, OMEGA may be better for the driving domain.

Question 3: *Neural Networks, especially Recurrent Networks, are often used for forecasting. Can Neural Networks be used to do system classification? If so, what is the advantage of OMEGA compared with Neural Networks?*

As we mention in Section 2.5., we decompose $P((x_i, y_i) \mid S_p)$ into the product of $P(x_i \mid S_p)$ and $P(y_i \mid S_p, x_i)$. We can use any machine learning method to approximate $P(y_i \mid S_p, x_i)$. Hence, Neural Networks can also be used to do system classification.

However, for each candidate system S_p, we should prepare a Neural Network. If we have 10,000 candidate systems, and the time series to be classified is 40,000 units long ($i = 1, ..., 40,000$), then we will have to try all of the candidate system at every time step. The computational cost will be 4 million units, which is not desirable. However, as a memory-based learning method, OMEGA can focus on the promising candidate systems from the very beginning. In this sense, OMEGA is cheaper than any parametric machine learning methods, such as Neural Networks.

Question 4: *What about the computational efficiency of OMEGA compared with HMM, global linear model ARMA, as well as Neural Networks?*

Concerning computational complexity, the training cost of the global linear model, ARMA, is $O(M^3 + T)$, where T is the total length of training time series samples, while M is decided by the model size of ARMA[1]. The training cost of HMM is $O(N^2 \times T^2)$, where N is the number of hidden states in the HMM. Typically, the training process of a Neural Network is divided into several epochs. If there are W weights in a Neural Network, each epoch takes $O(W \times T)$. However, the worst-case number of epochs can be exponential in d, which is the number of input attributes. OMEGA does not need any training process, but it re-organizes the memory of training time series data points in the form of a kd-tree, which takes $O(d^2 \times T + T \times \log T)$.

To evaluate a time series query, the ARMA approach is to estimate the parameters of the ARMA model. Hence, the computational cost of evaluating a time series query is similar to the training cost. If the length of a time series query is t, the computation cost of evaluating is $O(M^3 + t)$. The evaluating cost in HMM model is $O(N \times t)$, while that of a Neural Network is $O(W \times t)$. The order of computation complexity in OMEGA is also proportional to t, but in addition depends on what machine learning method is used to approximate $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$. For example, if OMEGA uses locally weighted logistic regression as the approximator, the computational cost is $O((d^3 + d \times T) \times t)$, where T is the number of training data points. However, with the help of cached kd-tree, the cost can be greatly reduced if the dimensionality, d, is not too large.

In summary, unlike HMMs and Neural Networks, OMEGA is not expensive to train. However, evaluating a time series query in OMEGA is not trivial in computation. Based-on our empirical knowledge, OMEGA[2] is still fast enough to be an on-line system classifier. Also notice that ARMA, HMMs and Neural Networks have to try all candidate systems at every time step of a time series query, hence if there are S candidate systems, their computational cost of evaluating a time series query are $O((M^3 + t) \times S)$, $O(N \times t \times S)$ and $O(W \times t \times S)$ respectively. On the

1. An ARMA model consists of two parts: AutoRegression (AR) and Moving Average (MA). If the window size of AutoRegression is p, and the window size of Moving Average is q, then $M = max(p, q + 1)$.

2. In our experiments, we used locally weighted logistic regression as the approximator of $P(X_{qi} \mid S_p)$ and $P(y_{qi} \mid S_p, X_{qi})$.

other hand, OMEGA can quickly focus on the promising candidate systems at the beginning of the query, so that its cost is $O((d^3 + d \times T) \times t \times s)$, sometimes $s << S$.

Question 5: *Ideally OMEGA assumes the input and output are fully observable and the output is fully determined by the input. However, in practice, this assumption is often violated. How badly will OMEGA perform when the assumption is violated?*

OMEGA studies the mapping between the input distribution and the output distribution. If there are some patterns in the mapping which can be used to distinguish different systems, OMEGA will work well, no matter whether or not the input and output are fully observable.

Question 6: *The principle of OMEGA is to calculate the residuals between the predictions and the observed results, then summarize the residuals in the form of likelihood. This is similar to Kalman filter. What is the difference between OMEGA and Kalman filters?*

Kalman filters assume that we know the closed-form formula for a system, and its goal is to estimate the parameters of the formula by minimizing the residuals between the predictions and the observations. The Kalman filter approach can be modified to do system classification, if we know the closed-form formula of the system. However in many cases we do not explicitly know the mechanism of the system, so we cannot go through the mathematical process of Kalman filters. OMEGA is a non-parametric method, which regards the system as a black box. This is the main difference between OMEGA and Kalman filters.

Question 7: *What makes some people's tennis styles similar? Is there any way to learn the similarity of individual styles directly?*

For the tennis experiment, the only instruction that we gave to the participants was: "hit the ball to make it move across the net." Based on our observation of the tennis experiment, the right-handed people are more likely to hit the ball toward the top-left corner of the court, while some left-handed people tend to make the ball move to the top-middle or top-right. Some people hit the ball harder than others, some people hit the ball once it comes across the net, etc. All the above are relevant to individual tennis styles.

Can we use some simple statistical features to do the system classification, such as the mean values of the contact angle, speed, the position of the contact? It is possible that the simple features work in some cases. However, in those cases, the input variables, i.e. the serving variables, must be uniformly distributed, because the contact angle, speed and the position of the contact, are also dependent on the input variables. OMEGA is more powerful than the feature approach since OMEGA studies the mapping between the input distribution and output distribution.

Question 8: *Suppose OMEGA is employed to detect several drivers' sobriety conditions. Each driver has both "sober" training data sets and "drunk" training data sets. Certainly we can use each driver's two kinds of training data sets to detect his sobriety. But is it helpful to put every driver's sober training data sets together as a mega sober training data set?*

Since all alert drivers share some common behavior, it is helpful to collect all "sober" training datasets into a big pool. However, for different drivers, the definition of being alert may be different. A cowboy's alert action may look very wild to a conservative person. Thus, if possible, a better idea is to put the training dataset generated by the *same type* of people together.

Question 9: *Is OMEGA good for speech?*

Because of accent and emphasis, the same sentence may be pronounced in different ways. In other words, for the same sentence, the distribution of the signal may be different, but the hidden states are always the same. Referring to the answer to Question 2, OMEGA is not good at approximating the relationship among the hidden states, but focuses on the distribution pattern of the signals. Therefore, in our point of view, OMEGA is not good for speech recognition.

10.2 Contributions

In this thesis, we explore a coherent framework to detect the underlying system that produced a given sequence of data points. This set of data points can be a time series in which the order of the sequence is important, or it can be a non-time series as well. Our approach is to transform the time series or non-time series into a set of data points with low input dimensionality, then use efficient memory information retrieval techniques and machine learning methods to do a series of classifications, and employ likelihood analysis and hypothesis testing to summarize the classification results as the final detection conclusion. The framework of our system is illustrated in Figure 10-1. The original contributions of this work are:

1. To our best knowledge, our work, for the first time in the literature, employs state-of-art data mining techniques in conjunction with memory-based learning methods to approach time series detection problem. Compared with other alternative methods, our method is simple to understand and easy to implement, it is robust for different types of systems with noisy training data points, it is adaptive when the density and the noise level of the training data points vary in different regions, it is flexible because it does not request fixed thresholds to distinguish various categories, it is efficient not only because it is capable of processing the classification quickly but also can it focus on the promising categories from the very beginning, and based on our empirical evaluation, it is more accurate than other methods.

2. We combined the locally weighted paradigm with logistic regression to be a new memory-based classification methods. Unlike the other memory-based classifiers, it is capable of extrapolating as well as interpolating. It is competent in accuracy, and with some extra useful features, especially, confidence interval. With the help of cached kd-tree, it is a very efficient classification method.

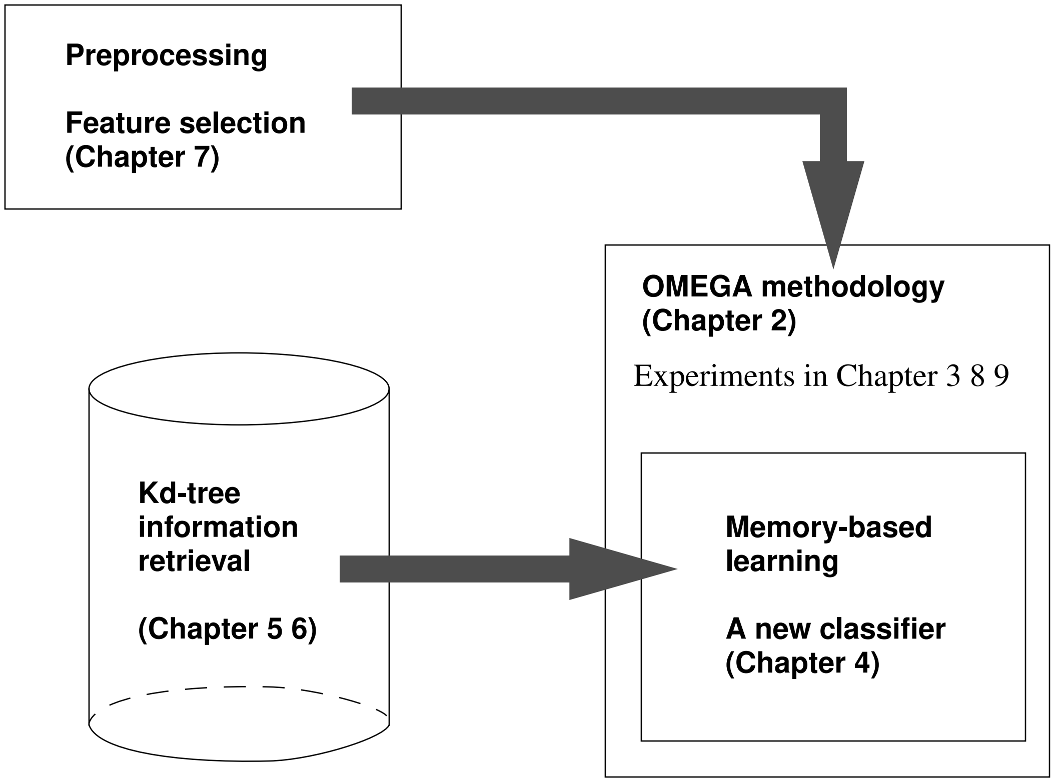


Figure 10-1: The structure of OMEGA system and the organization of the thesis.

3. As known for many years, kd-tree can be used to re-organize the memory so as to retrieve the useful information efficiently. By caching well-selected information into the kd-tree's node, we found a way to dramatically improve the efficiency of memory-based learning methods, including Kernel regression, locally weighted linear regression, and locally weighted logistic regression. Recently, cached kd-trees have also been applied to improve the efficiency of EM clustering [Moore, 98].

4. Due to the progress in improving the efficiency of the variety of learning methods, intensive cross-validation becomes feasible. We used intensive cross-validation to do feature selection, especially we explored several greedy algorithms to perform the selecting even faster while without severe loss in precision. We tried applying these algorithms to select the useful features so as to recognize Chinese handwriting off-line. Our prototype showed the accu-

racy could be over 95%.

10.3 Future research

1. Referring to Figure 10-1, the pre-processing module is to transform a time series into a set of data points in which the time order is no longer important, and to reduce the dimensionality of the dataset. Although in our system, we employ Principal Component Analysis and Feature Selection to reduce the dimensionality, there is no guarantee that we achieve our goal in any domain. In case the memory data points distribute in clusters, [Agrawal et al, 98] may be worth trying. More research should be done to attack the curse of dimensionality.

   One promising solution is that we can approximate the relationship among the input attribues using Bayesian network [Pearl, 88] or dynamic Bayesian network [Dean et al, 88]. By learning the configuration and the transition probabilities of the Bayes net [Heckerman et al, 95], we can compress each data points from high-dimensional space into a lower one, even a scalar [Frey, 98] [Davis, 98].

2. In this thesis research, we treated the system as a blackbox, we only study its inputs and outputs. This is desirable for many domains, because sometimes we do not have the precise domain knowledge. However, sometimes we *do* know somethings about the internal structure of the system, then we should exploit this knowledge because it is helpful to enhance the detection accuracy. [Heckerman, 96] and many other papers suggest that Bayesian network is capable of being a good system approximator with many advantageous properties.

   We propose that by using a same Bayes network, we can get double benefits: improving the accuracy, as well as reducing the dimensionality so as to improve the computational efficiency.

10.4 Applications

There are many possible applications, listed in Chapter 1. In this section, we discuss three applications in further depth.

Financial modeling

The importance of financial modeling is obvious: it helps to gain profit from the stock market, and avoid bad investments, such as the recent failure of Long Term Capital Management (LTCM) hedge fund. There are many researchers doing financial modeling, including some Nobel Prize winners. Why should we compete with them?

Most financial models assume the behavior of the financial market is controlled by a unique mechanism. Most Wall Street researchers want to make this unique model more complicated in order to fit all possible scenarios in the financial world. In contrary, we believe that although the stock index, like S&P index, is only an one-dimensional time series, the underlying mechanism of the financial market is not unique, instead, there are several different underlying control systems either working at the same time or switching from time to time. Suppose given the recent behavior of the stock market, including the various influencing factors like Fed's interests, we can use OMEGA to retrieve the similar historic data clips from the database, figure out which underlying mechanism is working nowadays. And based on that, we can predict what will happen to the stock market, with a certain confidence measurement.

Web server monitoring

The rapid growth of internet has greatly increased the pressure on administrators to quickly detect and resolve service problems. Typically, the detection job is done either by some ad hoc models to estimate weekly patterns [Maxion, 90], or by specifying threshold testing [Hellerstein et al, 98].

Using the techniques explored in this thesis, we are capable of detecting more complicated patterns efficiently, and distinguishing the patterns by specifying thresholds which are adaptable to datasets with different distribution densities and different noise levels. In other words, our technique may be more robust and accurate than the previous approaches.

Embedded detection device

To monitor if an engine works normally, we can embed a chip into the engine so that whenever it runs, the chip takes records of the engine's signals. If one day, the operator finds "sometimes" the engine did not work normally, he can pull out the chip from the engine, insert it into a device hooked to his home PC. His home PC is linked to a super server somewhere else through the internet. By comparing this engine's signal time series with those in the super server's database, the server can tell the operator when and how his engine went wrong. Thus, it is more convenient for the operator to decide if the engine needs repairing.

Compared with the conventional methods, which are based on the domain knowledge, our approach has more advantages: (1) Since there are so many engine nowadays in the world, and they are updated so quickly, it is not very convenient to update the conventional diagnosis system, because usually they are installed in the engines. For our distributed system, we can simply update the knowledge in our central super server, we do not need to modify the product we have sold to our customers one by one. (2) The conventional methods are of "we design, you use" style, our approach can interactively collect new data from the customers, then learn from them. Hence, with more and more experience, our system can automatically become more intelligent.

Appendix A

Chinese Handwriting Recognition

As a side experiment, we used the feature selection techniques discussed in Chapter 7 to recognize Chinese handwriting. Our goals are: (1) to demonstrate feature selection is important because it is the crucial part for the recognition job. (2) to compare the feature set found by the feature selection algorithms with a human expert's selection.

1.1 Feature selection for Chinese handwriting recognition

Although most of the research in handwriting recognition is for on-line systems [Singer et al, 94], there is no doubt that off-line systems are also very important especially in domains such as automatic tax form processing.

To date, research for Chinese and Japanese character recognition is still preliminary[1]. Because the number of Kanji, i.e. Chinese characters, is over fifty thousand, it is hard to rely on any general-purpose global model to recognize all Chinese characters. Alternatively, a promising approach is to separate the Chinese characters into several groups. For each group, a local model is developed to distinguish the different characters.

1. There are some Chinese and Japanese recognition products on the market. The product introductions claim that their accuracy is over 90%. However, we do not know what kind of principles they apply. And we notice some of those products can only recognize rigidly written characters.

Although it may be possible to build the local models off-line, manually, it is better if we have an on-line automatic configuration mechanism. Not only does this automatic system save software developers from tedious and time-consuming work, but also it is adaptive and can learn different personal handwriting styles.

In this section, we propose an idea to recognize Chinese and Japanese handwriting off-line, with automatically configured adaptive local models. We also give a prototype of this system.

Chinese characters are constructed by ten fundamental strokes.

The different combinations with different relative positioning determine different characters. For example, there are eight different Chinese characters plus “F” and the Japanese character “ki” containing two horizontal lines and one vertical line, illustrated in Figure A-1.

In this prototype system, some features are useful for recognition, while others may not be so significant, or, can be substituted, referring to Figure A-2. Notice: (1) The human expert’s selection, as shown in Figure A-2(a), is not the only functional set, there exist multiple options. (2) Among the multiple functional feature sets, some of them may lead to more accurate recognition than the others.

To find the features including those not-so-significant, we can follow these three steps:

- Figure out the horizontal lines, vertical lines, and other strokes, respectively.
- Sort the lines from top to bottom, or from left to right.
- Calculate all the possible features according to prior knowledge. In the case of Figure A-1, each stroke has two ends. The features can be the distances from the ends of each stroke to

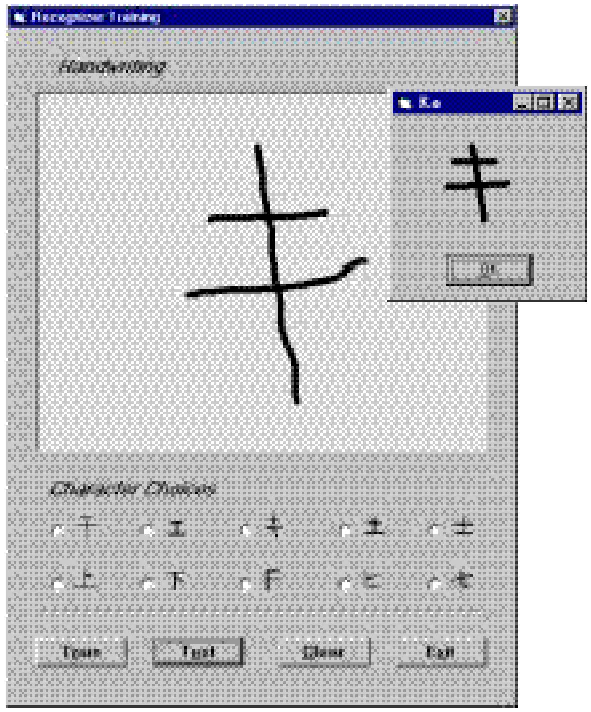
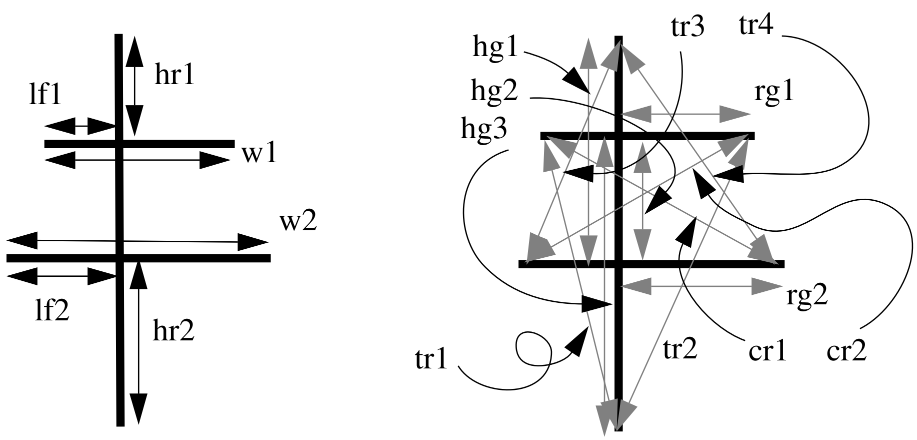


Figure A-1: A prototype of Chinese handwriting recognition system.



(a) Features selected by human expert

(b) Other candidate features, including those not-important ones.

Figure A-2: The features used for the Chinese handwriting recognition prototype system.

those of others, as well as the distances to all the intersections, illustrated in Figure A-2.

After we have found the candidate features, we can apply the various feature selection algorithms to select the proper features for the recognition job. In the experiment, we try four feature selection algorithms: Super-greedy (Super), Greedy (Greedy), Restricted Forward Selection (RFS) and conventional Forward Selection (FS). We request that any selected feature sets contain no more than eight components. To evaluate the goodness of the selected feature sets, we calculate their 20-fold scores. Since our procedure is carefully designed to avoid overfitting, the smaller a feature set's score is, the more accurately this feature set is able to recognize any one out of the ten characters. We also count the numbers of seconds consumed by the four algorithms so as to compare their computational costs.

Table A-1: Chinese character feature selection

Selection Methods	*m = 8 = Max Number of features*			*m = 12*	
	Selected feature set	*20fold score*	*Cost*	*20fold score*	*Cost*
Super	w2, lf1, lf2, hg1, tr1, tr2, tr3, tr4	0.038	532	0.018	529
Greedy	w2, lf1, lf2, hg1, tr2, tr3, tr4	0.041	767	0.022	916
RFS	hr1, w1, lf1, lf2, hg1, hg3, cr2, tr1	0.018	1414	0.016	1570
FS	hr1, hr2, w1, lf1, lf2, hg1, tr3	0.016	3586	0.018	4829
Human	hr1, hr2, w1, w2, lf1, lf2	0.016	--	0.016	--

In Table A-1, we observe that different selection algorithms may find different sets of features. When we carefully study these various sets with respect to Figure A-2, we find all of them are functional. Second, we find that the feature sets selected by RFS and FS are very similar to the human expert's preference, but different from the sets found by Super and Greedy. Third, although all of these feature sets have satisfactory accuracy, those found by the greedier algorithms lead to less accurate recognition performance. However, if we allow more components to enter the feature sets, even the greedier algorithms' selections become more powerful. Finally, the greedier algorithms are cheaper than the others.

1.2 Future work

The prototype system is sufficient to demonstrate the importance and capability of the feature selection algorithms. But to pursue a good Chinese handwriting recognition system, some further work has to be done. Since this topic is a digression from the discussion of feature selection, we only give a brief introduction.

For more complicated Chinese character, for example 藏 which means "hide" and "Tibet", the number of possible features will explode. Fortunately, every Chinese character can be split into some standard particles, and the number of these standard particles is no more than one hundred. Indexed by these particles and their relative positioning, any Chinese character can be represented by no more than five digits. One example is illustrated in Figure A-3. This technique is called Wang-coding or Five-stroke coding, which has become one of the national standard typing methods in China.

叭	23 (□)	24 (八)	1 (left to right)
只	23 (□)	24 (八)	2 (up and down)

Figure A-3: An illustration of Wang-coding of a Chinese character.

Now the remaining difficulty is how to find those standard particles from any Chinese characters. One promising approach is $A*$ search.

Bibliography

[Agrawal et al, 98] R.Agrawal, J.Gehrke, D.Gunopulos, and P.Raghavan. *Automatic subspace clustering of high dimensional data for data mining applications,* Proc. of the ACM SIGMOD Int'l Conference on Management of Data, Seattle, Washington, 1998.

[Aha, 89] D.W.Aha, *Incremental, instance-based learning of independent and graded concept descriptions*. In Proc. ICML-89, pp. 387-391, Morgan Kaufmann, 1989.

[Aha et al, 89] D.W.Aha and D.M.McNulty, *Learning relative attribute weights for instance-based concept descriptions*. In 11'th Annual conference of the cognitive science society, pp. 530-537. Lawrence Erlbaum Assoc. Hillsdale, 1989.

[Akaike, 73] Akaike, *Information theory and an extension of maximum likelihood principle*. 2'nd International Symposium on Information Theory, B.N.Petrov and F.Csaki (eds.), Akademiai Kiado, Budapest, pp 267-281, 1973.

[Almuallim et al, 91] H.Almuallim, and T.G.Dietterich. *Learning with many irrelevant features*. In Porc. AAAI-91, pp 547-552. MIT Press, 1991.

[Atkeson et al., 97] C.G.Atkeson, A.W.Moore and S.Schaal, *Locally weighted learning*, Artificial Intelligence Review, Vol. 11, No. 1-5, pp. 11-73, 1997.

[Batavia, 98] P.H.Batavia, *Driver adaptive warning system*. Thesis proposal, CMU-RI-TR-98-07, 1998.

[Bishop, 95] C.M.Bishop, *Neural networks for pattern recognition.* Oxford University Press Inc., 1995.

[Breiman et al, 84] L.Breiman, J.H.Friedman, R.A.Olshen, and C.J.Stone, *Classification and regression trees.* Belmont. CA:Wadsworth, 1984.

[Brockwell et al, 91] P.J.Brockwell, R.A.Davis. *Time series: theory and methods*, second edition, Springer series in statistics. Published by Springer-Verlag New York Inc. ISBN 3-540-96406-1, 1991.

[Caruana et al, 94] R. Caruana, and D. Freitag. *Greedy attribute selection.* In Proc. ML-94, Morgan Kaufmann, 1994.

[Davis, 98] S.Davis. *Compression, machine learning, and condensed representation*, CMU Ph.D thesis proposal, http://www.cs.cmu.edu/~scottd.

[Dean et al, 88] T.Dean and K.Kanazawa. *Probabilistic temporal reasoning.* In Proc. AAAI-88, pp 524-528, 1988.

[Deng et al, 97] K.Deng, A.W.Moore, and M.C.Nechyba, *Learning to Recognize Time Series: Combining ARMA models with Memory-based Learning,* Proc. IEEE Int. Symp. on Computational Intelligence in Robotics and Automation, vol. 1, pp. 246-50, 1997.

[Devore, 91] J.L.Devore, *Probability and statistics for engineering and the sciences*, 3'rd edition. Published by Brooks/Cole Publishing Co. ISBN 0-534-14352-0.

[Duda et al, 73] R.O.Duda, and P.E.Hart, *Pattern classification and scene analysis*, Published by John Wiley & Sons, New York, 1973.

[Franke, 82] R.Franke, *Scattered data interpolation: Tests of some methods*. Mathematics of computation. Vol. 38, No. 157. 1982.

[Frey, 98] B.J.Frey, *Graphical models for machine learning and digital communication.* MIT Press, 1998.

[Friedman et al, 98] N.Friedman,M.Goldszmidt, and T.Lee, *Bayesian network classification with continous attributes: getting the best of both discretization and parametric fitting,* In Proc. ICML'98, 1998.

[Grosse, 89] E.Grosse, *LOESS: Multivariate smoothing by moving least squares.* In C.K.Chul, L.L.S. and J.D.Ward editors, Approximation Theory VI. Academic Press, 1989.

[Heckerman et al, 95] D.Heckerman, D.Geiger, and D.M.chickering. *Learning Bayesian networks: the combination of knowledge and statistical data.* Machine Learning, 20: pp. 197-243, 1995.

[Heckerman, 96] D. Heckerman. *A tutorial on learning with Bayesian networks.* http://www.research.microsoft.com/~heckerman. Technical Report MSR-TR-95-06, Microsoft Research, March, 1995 (revised November, 1996).

[Hellerstein et al, 98] J.L.Hellerstein, F.Zhang, and P.Shahabuddin, *An approach to predictive detection*, IBM Research Report, limited distributed. 1998.

[James, 85] M.James, *Classification algorithms.* John Wiley & Sons, Inc. 1985.

[Jones et al, 94] M.C.Jones, S.J.Davies, and B.U.Park, *Versions of kernel-type regression estimators,* Jornal of the American Statistical Association, 89(427): pp 825-832, 1994.

[Jolliffe, 86] I.T.Jolliffe, *Principal components analysis*. Springer series in statistics. Published by Springer -Verlag New York Inc. ISBN 0-387-96269-7, 1986.

[Jordan et al, 93] M.I.Jordan, and R.A.Jacobs, *Hierarchical mixtures of experts and the EM algorithm*. MIT technical report. AI. Memo No. 1440, C.B.C.L. Memo No. 83, 1993.

[Kibler and Aha, 88] D.Kibler, and D.W.Aha, *Comparing instance averaging and instance filtering learning algorithms.* Proc. 3'rd European working session on leanring, Pitman, 1988.

[King et al., 96] R.A.R.King, H.L.MacGillivray, *Approximating disbutions using the generalised lambda distribution*, http://www.ens.gu.edu.au/robertk/publ/sisc.html, 1996.

[Kira et al, 92] K.Kira, and L.A.Rendell, *The feature selection problem: traditional methods and a new algorithms*. In Proc. AAAI-92, pp 129-134. MIT Press. 1992.

[Kleinberg, 97] J.M.Kleinberg. *Two Algorithms for nearest-neighbor search in high dimensions*. http://simon.cs.cornell.edu/home/kleinber.html, 1997.

[Koller et al, 96] D. Koller, and M. Shami, *Toward optimal feature selection*, In Proc. ML-96, Morgan Kaufmann, 1996.

[Langley et al, 94] P.Langley, and S.Sage, *Induction of selective Bayesian classifiers*. In Proc. UAI-94, pp 399-406. Seattle, WA. Morgan Kaufmann, 1994.

[Maron et al, 94] O.Maron, and A.W.Moore, *Hoeffding races: accelerating model selection search* , Proc. NIPS-94. Morgan Kaufmann, 1994.

[Maxion, 90] R.A.Maxion, *Anomaly detection for diagnosis*, Proc. of the 20'th Annual Iinternational Symposium on Fault Tolerant Computing (FTCS) 20, June 1990. pp 20-27.

[McCullagh et al, 89] P.McCullagh and J.A.Nelder, *Generalized linear models*, second edition, Monographs on statistics and applied probability 37, Chapman & Hall, 1989.

[Miller, 90] A.J.Miller, *Subset selection in regression*. Chapman & Hall, 1990.

[Moore, 90] A.W.Moore, *Efficient memory-based learning for robot control*. Ph.D. thesis, Technical Report, No. 209, Computer Laboratory, University of Cambridge, 1990.

[Moore, 90] A.W.Moore, *Acquisition of dynamic control knowledge for a robotic manipulator,* Proc. ICML-90. Morgan Kaufmann, 1990.

[Moore et al, 94] A.W.Moore, and M.S.Lee, *Efficient algorithms for minimizing cross-validation error*. In Proc. ML-94, Morgan Kaufmann, 1994.

[Moore, 98] A.W.Moore, *Very fast EM-based mixture model clustering using multiresolution kd-trees*. To appear in Neural Information Systems Processing, December 1998

[Nechyba, 98, (a)] M.C.Nechyba, and Y.Xu, *Stochastic similarity for validating human control strategy models*. Proc. IEEE trans. on Robotics and Automation, Vol. 14, No. 3, pp 437-51, 1998.

[Nechyba, 98, (b)] M.C.Nechyba, and Y.Xu, *On discontinuous human control strategies,* Proc. IEEE International conference on Robotics and Automation, Vol. 3, pp 2237-2243, 1998.

[Omohundro, 91] S.M.Omohundro. *Bumptrees for efficient function, constraint, and classification learning.* In R.P.Lippmann, J.E.Moody, D.S.Touretzky editors, Proc. NIPS-91, Morgan Kaufmann. 1991.

[Pearl, 88] J.Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan-Kaufmann, 1988.

[Petridis et al., 96] V.Petridis, A.Kehagias. *Modular neural networks for MAP classification of time series and the partition algorithm*. IEEE Transactions on Neural Networks. Vol. 7, No. 1, January 1996.

[Pomerleau et al, 96] D.Pomerleau. *RALPH: Rapidly adapting lateral position handler*. In IEEE Symposium on Intelligent Vehicle, Detroit, Michigan, 1995.

[Preparata et al., 85] P.F.Preparata, M.Shamos. *Computational geometry.* Springer-Verlag. 1985.

[Puskorius et al, 94] G.V.Puskorius, and L.A.Feldkamp, *Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks*. In IEEE Trans. on Neural Networks, Vol. 5, No. 2, March 1994.

[Quinlan, 93] J.R.Quinlan, *C4.5: Programs for machine learning,* Published by Morgan Kaufmann Publishers, London, England. 1993.

[Rabiner, 89] L.R.Rabiner, *A tutorial on Hidden Markov Models and selected applications in speech recognition*, in Proc. of the IEEE, Vol. 77, No. 2, Feb. 1989.

[Singer et al, 94] Y. Singer, and N. Tishby, *Dynamic encoding of cursive handwriting*. Biological Cybernetics, 71 (3), 1994. Springer-Verlag.

[Skalak, 94] D.B.Skalak, *Prototype and feature selection by sampling and random mutation hill climbing algorithms*. In Proc. ML-94, Morgan Kaufmann, 1994.

[Utgoff et al, 91] P.E.Tgoff, and C.E.Brodley, *Linear machine decision trees,* COINS Technical Report 91-10, University of Massachusetts, Amherst, MA.