University of Alberta

APPLYING MACHINE LEARNING AND SELECTIVE SAMPLING TECHNIQUES TO GAME SOFTWARE TESTING

by

Gang Xiao

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2007

Your file Votre référence
ISBN: 978-0-494-30042-8
Our file Notre référence
ISBN: 978-0-494-30042-8

Abstract

Although commercial computer games usually undergo intensive testing before release, many bugs and sweet spots still exist and make games less attractive than expected. In this thesis, a Semi-Automated Gameplay Analysis (SAGA-ML) system is developed to summarize game behaviors as human readable rules, which can be presented to game designers to check if those behaviors are as intended. Unexpected game behaviors can be found this way. Machine learning and selective sampling techniques are incorporated into automated software testing. Machine learning is used to create a summary of the gameplay log that is comprehensible by humans. Selective sampling is used to sample instance space intelligently to build a good model. Four existing selective sampling algorithms (*Uncertainty Sampling, Bagging, Boosting* and *BootStrapLV*), and a new rule-based selective sampling method, are implemented and compared. SAGA-ML has been tested on Electronic Arts' FIFA99 soccer game and shown to be a practical game behavior testing solution.

Acknowledgements

I would first like to thank my supervisor, Dr. Robert Holte, for his guidance in this project. He is the designer of the whole system and all important components. He is also the one who kept this project on the right track. Whenever I ran out of ideas on how to proceed or was frustrated because of bad experimental results, Rob always helped me come up with new ways of solving the problem. Rob also gave me tremendous help with my career.

I would like to thank Finnegan Southey, who has also provided a great deal of insight into this research. Finnegan was involved in this project in many ways. He contributed many good ideas like *Blur*, gave many good suggestions about the experiments and he was the key person to put our system into publications.

I would also like to thank Mark Trommelen who finished SocerViz, a key component of our system. Whenever we proposed a new requirement, he would work it out quickly.

Thank you Mom, you are always encouraging me and trusting me. Thank you my sister, you take care of our mom for me. I also want to thank all my other family members for their support throughout my studies.

Finally, I would like to thank Electronic Art Inc.(especially John Buchanan) for providing us the source code of FIFA's 99. I would also like to thank IRIS/PRECARN, Alberta Ingenuity Center for Machine Learning (AICML), and the University of Alberta for their financial support.

Table of Contents

List of Figures

Chapter 1

Introduction

The gaming industry is a multi-billion dollar market, bigger than the traditional movie market. In the early days of the game industry, game design was a small-scale development process, relying on individual contributors. Now, gaming has become a large-scale industry. Current game design involves many different areas such as: *graphics, audio, AI* and *system* etc.

Graphics usually is one of the most important parts of game design. However, *gameplay* is also very important. According to www.webopedia.com's definition, *gameplay is a term most commonly used to ratè, or score the quality of the experience had by a gamer while playing a particular game.* The term gameplay is often found in game reviews where a score is given based on player experiences during the interaction with the game. Unlike the visible factors of a game (e.g. the graphics and frontend), gameplay usually refers to the game behaviour such as the game's logic, difficulty, goals and constraints. Gameplay contributes much to the enjoyability of a game. Like other game elements, gameplay will be designed to specifications.

As games get bigger and bigger, game testing becomes very challenging. Like other types of software, game testing also includes Unit Testing, Integration Testing, System Testing and Performance Testing. However, some testing areas are unique to game development. Gameplay testing is one of them.

1.1 Problem Description and Motivation

In the current game industry, the common method of doing gameplay evaluation is a process of play-and-feel, which totally relies on a human's sensitivity and experience. This play-and-feel method is not efficient, and sometimes not effective. That is why so many "sweet spots" exist, even when a game goes through intensive testing before it is released. In

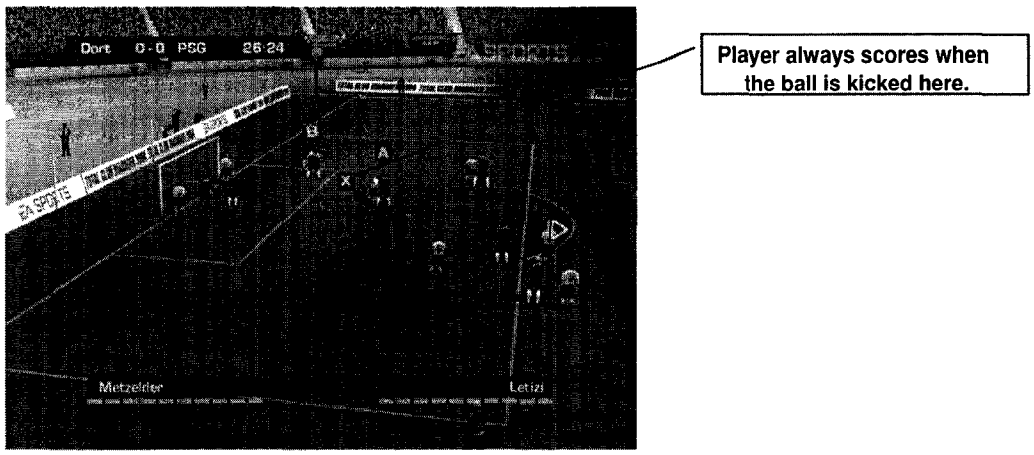


Figure 1.1: A Sweet Spot In FIFA99 Corner Kick Scenario

the context of a computer game, sweet spots are unexpected (from the view of the game designers) game behaviors. Many gameplay sweet spots are exploited by human players to decrease the difficulty level of the game, which will impact the playability of the game eventually. For instance, in the corner kick scenario of Electronic Arts (EA) FIFA'99, there are some spots where the attackers can score easily when the ball lands there.

Finding sweet spots is currently mainly done by human testers. In practice, game companies hire many QA testers or take clients' feedback (e.g. based on demo or Beta versions) to find bugs. However, due to the huge state space for a game, it is impossible to exhaustively check all the possible behaviors.

Previously, a Cheat Finding System (CFS)[8], developed at the University of Calgary, used genetic algorithms to find specific action sequences that lead to a goal in the FIFA99 soccer game. Firstly, CFS will define the set of control actions for the game (Figure1.2 is the example for FIFA99). Then, the game will be played by CFS. Every certain amount of time, a new action will be applied to the game from the action list and added to an action sequence. Genetic algorithms are used to evaluate and populate those action sequences until some interesting action sequences are generated. The CFS system as able to find some action sequences that lead to a goal in FIFA99.

However, reporting such action sequences is not all that game designers want. Firstly, some action sequences that lead to a goal found by CFS might be very hard to be reproduced by human players due to the complexity of the sequence. Secondly, game companies like general summarization of games' behaviors more than individual action sequences. Therefore, the research in this thesis aims to develop an automated software testing system

Action0 NO_OP

Action1 PASS

Action2 SHOOT

Action3 SWITCH

Action4 MOVE_UP

Action5 MOVE_DOWN

Action6 MOVE_RIGHT

Action7 MOVE_LEFT

Action8 MOVE_UP_LEFT

Action9 MOVE_UP_RIGHT

Action10 MOVE_DOWN_LEFT

Action11 MOVE_DOWN_RIGHT

Figure 1.2: CFS defined actions

to summarize games' behaviors. Game designers will check those summarized behaviors according to design specifications.

1.2 The Solution

Figure 1.3 is the basic architecture of our solution. The *Sampler* module generates initial conditions and action sequences defining samples of the game's behaviour. The *Game Engine* module takes the sample data, transforms the initial conditions into game variables and executes the given sequence of actions. The output of the *Game Engine* is labelled sample data, with the label depending on the testing purpose. The *Learner* module uses the labelled samples to create models of behaviour. The learned model can be used by the *Sampler* to generate more samples. The *Visualizer* module displays the learned models to the game designer and allows interaction with the Game Engine to help understand the learned behaviour model.

This system is a kind of automated software testing. Machine learning is used to create a summary of the gameplay log that is comprehensible by humans. Ideally, the behavior testing should exhaust all the possible behaviors, which is impossible for a big game. We therefore propose to apply our testing method to small, well-defined portions of the game,

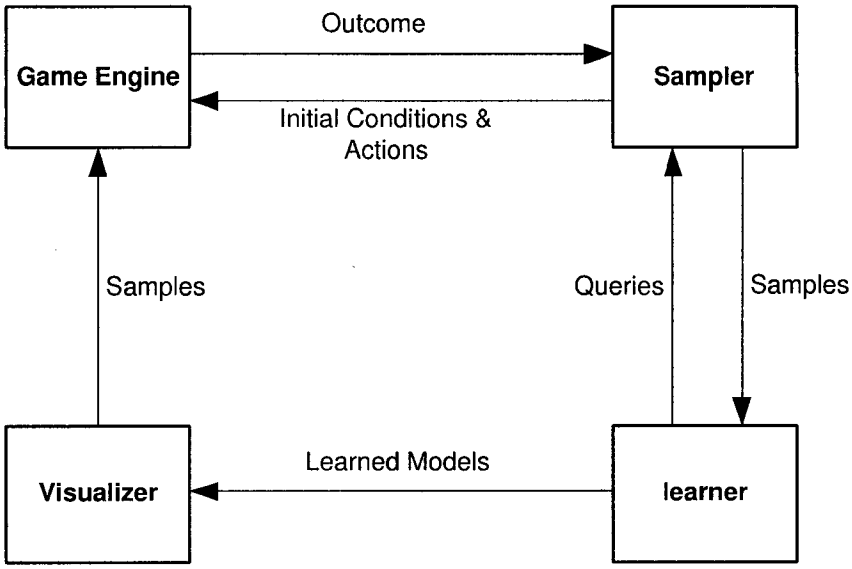


Figure 1.3: Brief Architecture

which we call *scenarios*. Usually, a game scenario is triggered under certain circumstances. For example, the *corner kick* scenario in a soccer game is triggered when the ball is kicked over the kicker's own end line. Thus, the huge sample space for the full game is broken down into relatively small game scenario spaces. From the game player's point of view, gameplay consists of many different game scenarios. Even restricted to scenarios, some types of behaviors cannot be enumerated exhaustively, e.g., behaviors involving continuous values. Therefore, in this thesis, selective sampling is used to sample instance space intelligently to build a good model. Four existing selective sampling algorithms (*uncertainty sampling, bagging, boosting* and *BootstrapLV*) are implemented, and a new rule-based selective sampling method is introduced.

1.3 Challenges and Contributions

There are quite a few challenges in this work. In order to summarize game scenarios' behaviors, we have to simulate those scenarios and automatically test them. Game environments are usually very complex and dynamic. The realistic automation of game scenarios is the first challenge. The state space of a game (or even a game scenario) can be huge and impossible to be exhaustively analyzed. Choosing efficient methods to sample such big state spaces is another challenge. How to evaluate the summarized game behaviors we got is the third challenge.

The first major contribution of this thesis is the development of a *Semi-Automated Gameplay Analysis (SAGA-ML)* system. This framework is a whole set of solutions that can be used to analyze game behaviors.

A second major contribution is that *machine learning* techniques are used to summarize game behaviors. Gaming testing is a quite new area to apply machine learning techniques. Experiments show that machine learning can help game designers by providing accurate and concise summaries of game behaviors.

A third major contribution is that *selective sampling* techniques are used for efficient sampling. Five selective sampling methods are implemented and compared with random sampling in both artificial environments and real games.

The fourth contribution of this thesis is a new, rule-based selective sampling method. This new sampling method analyzes the rules that make up the behaviour model. Experiments show that this sampling method has very good performance.

The last contribution is the introduction of an evaluation method for gameplay analysis based on "blurred" target concepts. Blurring generates larger regions in in its concept definitions by eliminating small regions and merging neighbouring regions. Evaluation based on blurred target concepts is more useful for game designers because only large regions are important to game developers and human players. Game developers can easily get a global view of their system instead of being distracted by many small summaries.

Four publications have been generated by this thesis work. "Machine Learning for Semi-Automated Gameplay Analysis"[7], "Semi-Automated Gameplay Analysis by Machine Learning"[37] and "Semi-Automated Gameplay Analysis"[36] focus on the use of the SAGA-ML system for gameplay analysis. "Software Testing by Active Learning for Commercial Games"[47] focuses on active learning and experimentally compares different sampling methods.

1.4 Organization

Chapter 2 overviews *selective sampling* and explains the 5 selective sampling methods that are used in this research. Chapter 3 presents the game application and the framework of our solution. Chapter 4 shows the experimental results when our solution is applied on a real commercial game: Electronic Arts's FIFA99. Conclusions are drawn in Chapter 5.

Chapter 2

Selective Sampling

In this chapter, selective sampling is reviewed and studied experimentally. Section 2.1 gives the explanations of *what selective sampling is* and *why selective sampling is expected to be superior to random sampling*. The four existing selective sampling studied in this thesis are described in Section 2.2.1: *uncertainty sampling, QBC: bagging, QBC: boosting,* and *BootstrapLV*. A new rule-based sampling method is presented in Section 2.2.2. In Section 2.3, these five selective sampling methods and random sampling are compared experimentally.

2.1 Selective Sampling – What and Why

2.1.1 What is Selective Sampling

Normally, the training data for learning is provided by an external source entirely independent of the learning algorithm. In selective sampling (also called *active learning*), the algorithm starts from an initial training set which is usually small in size and randomly generated. After a model is learned by the learner, a strategy is used to evaluate potential additional training data. The most informative examples (as defined by the selection criteria) are picked for labelling and added to the training set. Then, the model is updated based on the new training set. This process is continued until the stopping criteria are met (e.g., the model becomes stable). Selective sampling is a methodology and a framework. There are three key components in a selective sampling algorithm. The first one is the basic learning algorithm which is in charge of building models from training data. The second one is the strategy for picking new training data. The third one is the stopping criterion, which usually is defined by the convergence of certain criteria. Learning algorithms have been very well studied in machine learning community and there are many excellent learners available. Therefore, the biggest concern in selective sampling is the second issue – how

to effectively and efficiently select the most informative examples as new training data.

2.1.2 Why Selective Sampling

Selective sampling is attractive for the following types of applications:

- **Training data is expensive**

  Training data for some applications such as astronomy and biology can be very expensive. For example, the satellite images in [17] cost hundreds, sometimes thousands of dollars each.

- **Labelling is expensive**

  Sometimes, unlabelled data is abundant and cheap but labelling is very expensive. For example, manually classifying a single protein shape requires months of expensive analysis by expert crystallographers[48].

- **Too much training data**

  In some applications, large quantities of unlabelled examples are very cheap. For example, web pages are abundant[22]. The task of classifying large quantities of web pages is not possible for most memory-based classifiers.

2.1.3 Creating New Examples vs. Selecting From an Example Pool

New examples in a selective sampling algorithm can come from two sources: created in instance space or selected from an example pool. The algorithms based on an unlabelled example pool are called pool-based. Many database applications (e.g., [38]) fall into this category. On the other hand, some applications have continuous instance spaces and an oracle that can classify on the fly any new examples created.

2.2 Comparison of Different Selective Sampling Methods

This thesis focuses on the methods for selecting new training data in each iteration. Therefore, the comparison of different selective sampling methods in this thesis means the comparison of different new training example selection methods. Other topics about selective sampling such as the size of the initial training set, the number of new examples on each iteration, and the stopping criteria, are avoided by using fixed definitions, as follows.

- **The size of initial training set**

  Fixed size initial training set is given for each comparison.

- **The number of new examples on each iteration**

  Fixed number of new points on each iteration are taken for each comparison.

- **Stopping criteria**

  Fixed number of iterations is used for each comparison.

2.2.1 The existing selective sampling methods used in this thesis

Saar-Tsechansky and Provost[32] provide a good review on the area of selective sampling. *Region of Uncertainty* [9], *Query By Committee* (QBC)[34] and *Uncertainty Sampling*[21] are explained in that paper. They also presented a new selective sampling method *boosting with local weighting (BootstrapLV)* and compared it with uncertainty sampling. In this thesis, *uncertainty sampling*, *bagging-based QBC*, *boosting-based QBC* and *BootstrapLV* are implemented and are compared with random sampling and a new rule-based sampling method. They are all pool-based sampling methods.

- **Uncertainty Sampling**

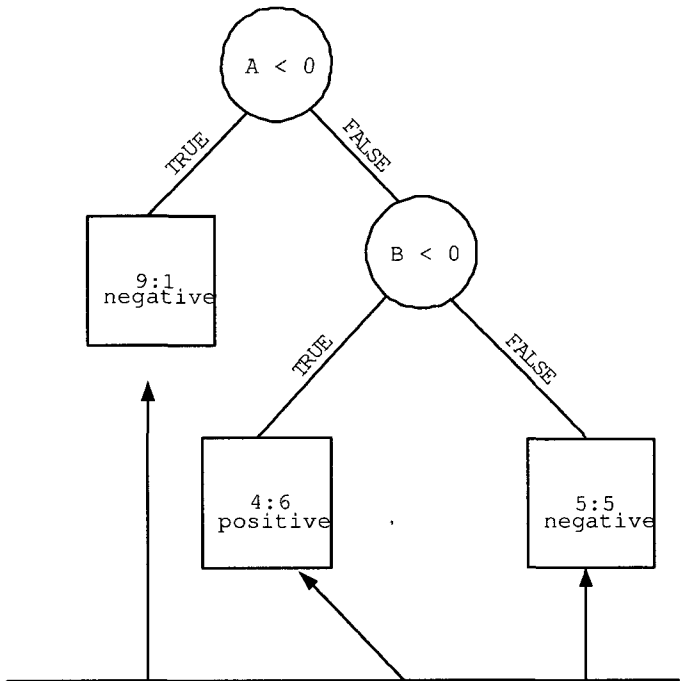
  Uncertainty Sampling[21] aims to select those examples whose classification is least certain. In this thesis, a C4.5[29] decision tree is used to measure the certainty of an example. After training, each leaf node has a label, which is determined by the majority class of all examples in this leaf. When a prediction query comes and reaches a leaf node in the tree, the probability of belonging to each class is calculated by the example class distribution of this leaf node. e.g., if a leaf node (labelled as *positive*) is supported by 12 examples and contradicted by 4 examples, the prediction class probability of an incoming query example is 75% *positive* and 25% *negative*.

  Pseudocode of the implementation of uncertainty sampling in this thesis is as follows:

  1. Classify each example, e_i, in the example pool using C4.5's tree, and record the probability, p_i, of it being in the *positive* class.
  2. Sort all examples in the pool by the absolute value of $(p_i - 0.5)$
  3. Pick the top n examples from the sorted list

  Figure 2.1 shows how uncertainty sampling works. The upper part of Figure 2.1 is a decision tree generated by C4.5. A and B are two features. The numbers inside each leaf node are the number of positive examples and negative examples respectively in the training set. The lower part of Figure 2.1 is a table which shows 3 examples (e_1, e_2, and e_3) from an example pool. e_1 goes to the leftmost leaf, whose positive

probability is 0.1; e_2 goes to the middle leaf, whose positive probability is 0.6; e_3 goes to the rightmost leaf, whose positive probability is 0.5. The ranking scores of e_1, e_2, and e_3 are $|0.1 - 0.5| = 0.4$, $|0.6 - 0.5| = 0.1$ and $|0.5 - 0.5| = 0$. Those scores are sorted in ascending order. Therefore, the selection order of those 3 examples is e_3, e_2 and e_1. The number of examples to be picked in each selective sampling iteration is a parameter of the implementation.



	e_1	e_2	e_3
A	-10	10	1
B	50	-5	2

Figure 2.1: Uncertainty sampling example

- **Query by Committee (QBC) methods**

  Query by Committee[34] originally was used to build a strong classifier from a group of very weak simple classifiers. This is not simply an algorithm, but a methodology. In selective sampling contexts, instead of using a single classifier (e.g., as uncertainty sampling does), a Query by Committee algorithm will construct a collection

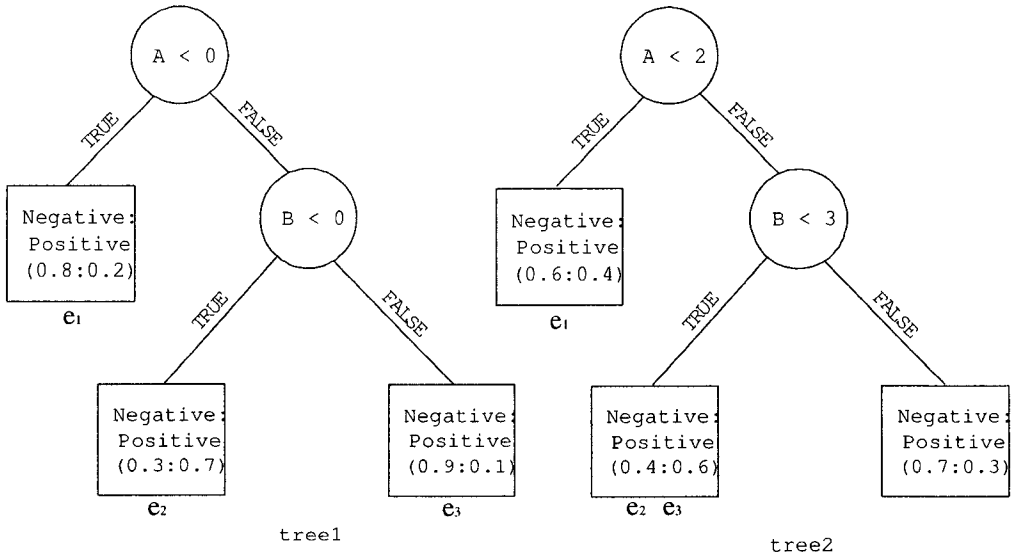
of individual classifiers that are diverse and yet accurate. After such a committee is generated, the potential training data is evaluated by the voting of this committee. The examples with the most classification disagreement among the committee are selected to be added to the training set. A new committee will then be built from the new training set, and so on. Because Query by Committee algorithms do not evaluate the uncertainty by directly checking the classification probability value for single examples (as uncertainty sampling does), they overcome the problem that uncertain sampling has: oversampling examples with a real classification probability close to 0.5. If most committee member agree that a region's real classification probability is 0.5, the examples there will not be chosen. In Figure 2.2, there is a 2-member committee, which are two similar C4.5 trees (tree1 and tree2), and 3 pool examples (e_1, e_2, and e_3) are sent to the committee to calculate their disagreement scores. For example e_i, the disagreement score of e_i is $S_i = \sum_{j=1}^{m} |p_{ij} - \overline{p_i}|$, where m is the number of members of the committee; p_{ij} is the probability of a positive label for example e_i by committee member j; and $\overline{p_i}$ is the average positive probability for e_i from all committee members. The disagreement score of e_1 is $0.2 = |0.2 - 0.3| + |0.4 - 0.3|$; the disagreement score of e_2 is $0.1 = |0.7 - 0.65| + |0.6 - 0.65|$; the disagreement score of e_3 is $0.5 = |0.1 - 0.35| + |0.6 - 0.35|$. Those scores are sorted in descending order. Therefore, the selection order of those 3 examples is e_3, e_1 and e_2. The number of examples to be picked in each selective sampling iteration is a parameter of the implementation. This sample selection strategy works for general Query by Committee (QBC) selective sampling algorithms. *Bagging*-[6] and *boosting*-[14] based selective sampling methods implemented in this thesis follow this sample selection strategy. In the actual implementation, there are 10 C4.5 trees in the committee. *Boosting With Local Variance* (BootstrapLV, a variant of QBC-boosting) is implemented as well but its sampling selection method, which will be described later, is slightly different from the general one described here.

- **Bagging**

  In bagging, a classifier is built from a modified training set, which comes from the original training set by random selection with replacement. Pseudocode of the implementation is as follows:

  1. Build committee (m members).

     for each member $j, 1 \leq j \leq m$



	e_1	e_2	e_3
A	-10	3	3
B	50	-1	2
Average Positive Probability	0.3	0.65	0.35
Disagreement score	0. 2	0. 1	0.5

Figure 2.2: Query by Committee example

   * build a training set TS_j by randomly selecting n examples from the original training set with replacement
   * generate a decision tree DT_j by applying C4.5 to TS_j

2. calculate the disagreement score for each example in the pool.
   for each example e_i in the pool
   * for each committee member j, classify example e_i using DT_j and record the probability of the positive class p_{ij}
   * calculate the disagreement score $S_i = \sum_{j=1}^{m} |p_{ij} - \overline{p}|$, where $\overline{p}$ is the average positive probability for e_i from all committee members
3. Pick the k examples with the highest disagreement scores

- **Boosting**

  Boosting algorithms maintain a set of weights for the examples in the original training set. The weights will be adjusted after each classifier is learned. The weight for an example will be increased if it is misclassified by the classifier, and decreased on the other hand. The training set for each classifier of the committee is built based on the original training set and the current set of weights. The version implemented in our system is AdaBoost.M1[14]. Pseudocode of the implementation is as follows:

  1. create a weight vector W for all n examples in the pool ($E = \{e_i, 1 \leq i \leq n\}$) and initialize those weights as $W_i = 1/n, 1 \leq i \leq n$
  2. create a weight vector DTW for m committee members
  3. Build the committee.

     for each committee member (decision tree $DT_j, 1 \leq j \leq m$)

     * generate DT_j with C4.5 using Training Set TS and current W
     * let $\beta_j = \epsilon/(1-\epsilon)$, where ϵ is the error of DT_j
     * for each example $e_i, 1 \leq i \leq n$, $W_i = W_i * \beta_j$, if e_i is correctly classified by DT_j
     * normalize W so that $\sum_{1 \leq i \leq n} W_i = 1$
     * $DTW_j = \log(1/\beta_j)$
  4. normalize DTW so that $\sum_{1 \leq j \leq m} DTW_j = 1$
  5. calculate the disagreement score for each example in E

     for each example $e_i, 1 \leq i \leq n$ in the pool

     * for each committee member $j, 1 \leq j \leq m$, classify e_i using DT_j and record the probability of positive class p_{ij}
     * calculate the disagreement score $S_i = \sum_{j=1}^{m} (|p_{ij} - \overline{p}| * DTW_j)$, where $\overline{p}$ is the average positive probability for e_i from all committee members
  6. Pick the k examples with the highest disagreement scores

- **Boosting with local variance (BootstrapLV)**

  This sampling method is proposed in [32] as a selective sampling method that can be applied to applications requiring estimations of the probability of class membership, or scores that can be used to rank new cases. The paper claims BootstrapLV is more powerful because existing empirical selective sampling approaches have focused on learning classifiers (note: all the selective sampling

methods are implemented to be capable of estimating the probability of class membership). The sample selection method of BootstrapLV differs from other general QBC based algorithms (e.g., bagging- and boosting-based) in that after the disagreement scores of all examples in the pool are calculated, BootstrapLV does not select the examples with the highest disagreement scores. Instead, each example (except those whose disagreement score = 0) has a chance to be chosen, with a probability proportional to its disagreement score. Figure 2.3 shows 8 examples and their scores. The scores are calculated in the same way as bagging. Unlike bagging and boosting, which will pick the top examples, BootstrapLV can pick any example according to its probability (proportional to its score). In Figure 2.3, e_1 has the highest score, but it only has a 30% probability of being selected.

	score
e_1	0.3
e_2	0.2
e_3	0.1
e_4	0.1
e_5	0.1
e_6	0.1
e_7	0.05
e_8	0.05

Figure 2.3: BootstrapLV sample selection example

2.2.2 A new rule-based selective sampling method

A new selective rule based sampling method is proposed in this section. There are two motivations to introduce this new method:

- Specific to rule learning

  We use a rule-based classifier in our system. Existing selective sampling methods

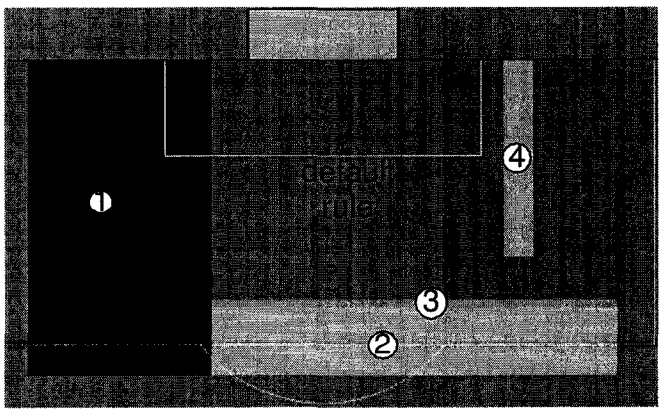
are general-purpose, not specific to a particular type of classifier. We hope the new method specific to rule learning will be more powerful.

- Specific to our summarization task

  Existing selective sampling methods are mainly score based, which places more attention on individual examples. The new rule-based sampling method is distinct from other selective sampling methods in that it works directly on the rules themselves, not on examples. Therefore, the new sampling method is expected to work better in terms of refining rules during the active learning process compared to other selective sampling methods.

The rule inducer used in this thesis is C4.5's rule learner. A rule is in conjunctive form, such as "if A and B then C", where A and B are the rule's antecedents and C is the rule's consequence. From a C4.5 decision tree, every path from the root to a leaf can be converted into a rule. From that initial set of rules will be removed unnecessary rule antecedents using a greedy algorithm; then the rules are grouped according to classes; the rules of each group are polished: some unnecessary rules are filtered; groups are sorted by their false positive error rates; finally, the default rule is generated.

The rules generated by C4.5 are ordered and can overlap. Our rule-based method requires unordered, non-overlapping rules, hence it starts by converting the rule set into an *exclusive rule set.* A rule set (generated by C4.5) for the FIFA99 corner kick scenario is shown on the right side of Figure 2.4. The corresponding exclusive rule set is visualized on the left side of Figure 2.4 (individual rules are marked as 1, 2, 3 and 4). The algorithm for converting an ordered and overlapping rule set into an exclusive rule set is as follows:



Rule1 Class: Not score
target.x <= -336
Rule2 Class: Not score
target.y <= 2082
Rule3 Class: Score
target.y <= 2105
Rule4 Class: Score
target.x > 552
target.x <= 641
target.y > 2234

Default class: Not score

Figure 2.4: An example rule set (FIFA99 corner kick scenario)

Algorithm: Convert An Ordered and Overlapping Rule Set into An Exclusive Rule Set

Notation:

Original rule set S_o

Exclusive rule set S_e

Sub-rule set R_s

Number of rules in original rule set N

a single rule r

Input: S_o

Output:S_e

Initialization: set S_e = empty

Algorithm:

For $i = 1, ..., N$

For $j = 1, ..., M$, M is the current size of S_e

1. calculate the intersection r_c of r_i and $S_e j$
2. If r_i intersects $S_e j$, r_i needs to be divided into a set (R_s) non-intersecting hyperrectangles whose union is all of r_i except for the part that intersects with Se_j (in Figure 2.5, r_i is split into R_{s1}, and R_{s2}).
3. $S_e = S_e \cup R_s$

This conversion process is not free, it can be very expensive. In terms of the number of rules, the exclusive rule set can be a hundred times larger than its ordered but overlapping counterpart. Imagining a worst case: in a two-dimensional space, with 10 distinct attribute values for each dimension, the total number of exclusive rules is 100 (10 by 10). Therefore, in the worst case, the number of exclusive rules in a n-dimensional space can be $N_{total} = v_1 * v_2 * ... * v_n$, where v_i is the number of attribute values of dimension i and $1 \leq i \leq n$. Usually, in the active learning process, the learned rule set will become more and more complex as the iteration number grows. Therefore, the process of converting the learned rule set into its exclusive counterpart will become correspondingly more and more expensive. This is a limitation of our current rule-based sampling method. Future improvements can be made by refining the conversion algorithm.

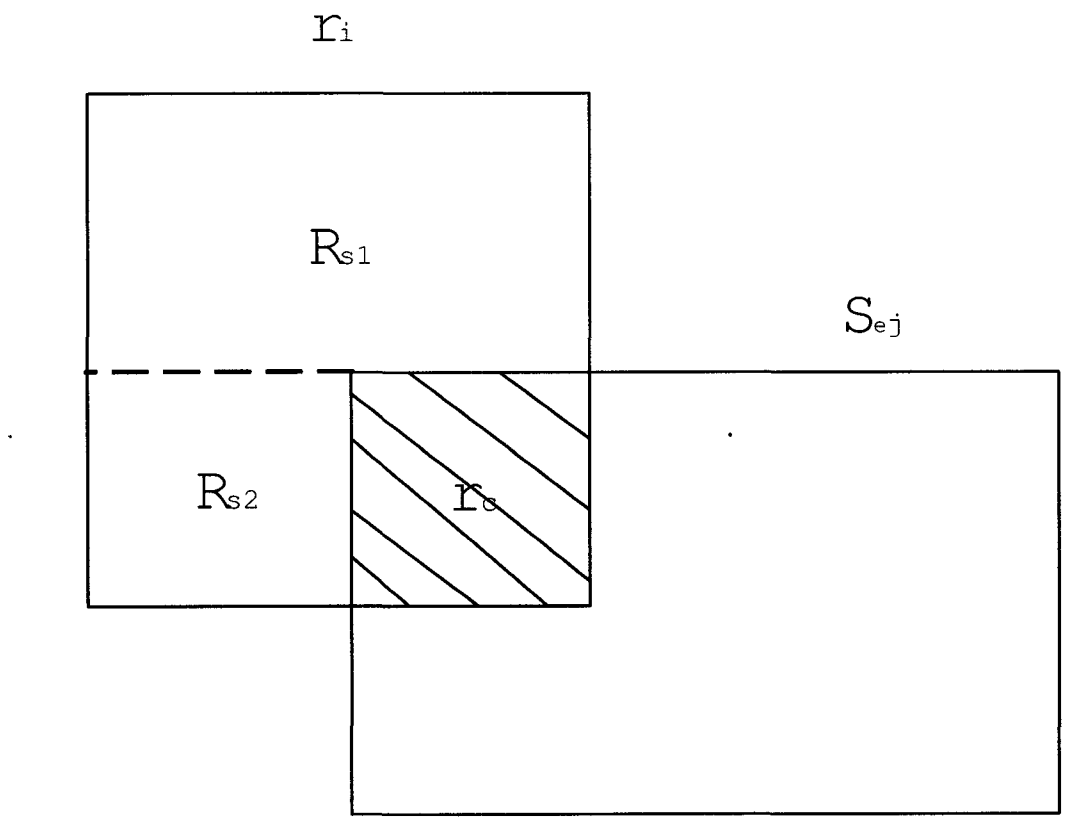
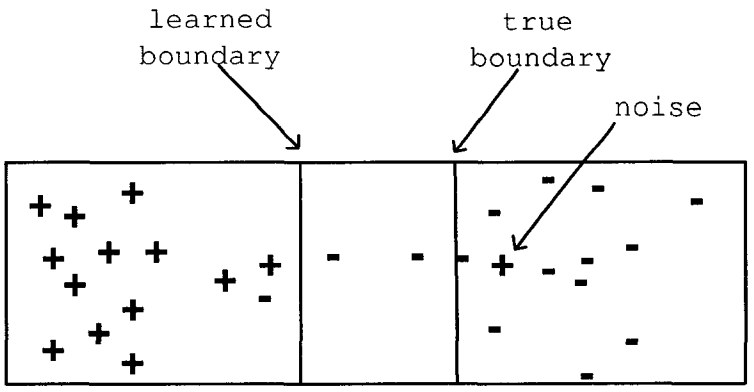


Figure 2.5: Rule split

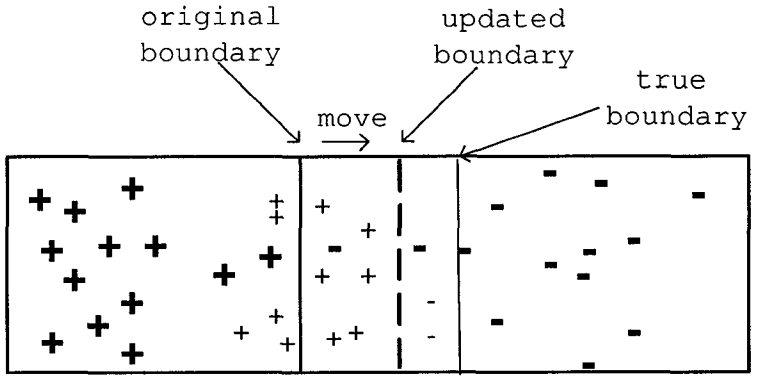
Based on exclusive rule sets, rule-based selective sampling consists of four methods: sampling rule boundaries, sampling rules with low support, sampling the neighbourhood of counterexamples, and sampling the default rule. The weights (in proportion to the number of new generated examples) of each method are input parameters of SAGA-ML system. These 4 methods are described in the following subsections.

- **Sampling rule boundaries**

  Given enough training data, the rule boundaries could be very accurate. Selective



(a) Two rules (+ represents a positive example, - represents a negative example) sharing a rule boundary



(b) After adding more examples (small + and small -) in the rule boundary area, the rule boundary moves to be more accurate

Figure 2.6: Sampling Rule Boundaries

sampling algorithms usually start from a small set training data, which means the rule boundaries at the beginning could be (very) inaccurate. Figure 2.6(a) shows a rule that roughly separates two groups of examples. The idea is to sample the boundary area of the rules so that the rule boundaries move towards the "correct" location.

Figure 2.6(b) shows that after sampling the rule boundary area (non-bold examples), the rule boundary moves in the correct direction. One potential problem is that the same (or similar) rule boundaries could be sampled again and again during the active learning process, even though some rule boundaries might already be very accurate. In our system, we keep track of which rule boundaries have been sampled and avoid repeatedly sampling the same boundary areas.

Algorithm: Sampling rule boundaries

Notation:

A set of mutually exclusive rules for an active learning iteration: R

Rule boundaries generated by R: B

Rule boundaries that have been sampled in the past: B_{done}

Rule boundaries to be sampled: B_o

Examples to be labelled: E

Input: R

Output: E

Initialization: set B = empty, set B_o = empty, set E = empty .

Algorithm:

1. for $i = 1, ..., n_r$, where n_r is the number of rules in R

   $B = B \cup B_i$, where B_i is the rule boundaries generated by rule i

2. for $i = 1, ..., n_b$, where n_b is the number of boundaries in B

   if the boundary $b_i \notin B_{done}$ and b_i is the shared boundary of multiple (> 1) classes

   then $B_o = B_o \cup \{b_i\}$

3. $B_{done} = B_{done} \cup B_o$

4. for $i = 1, ..., n_o$, where n_o is the number of boundaries in B_o

   $E = E \cup E_i$,

   where E_i is the example set generated from boundary b_i. In our implementation, a fixed number of examples (3) are sampled randomly from the boundary space.

- **Sampling rules with low support**

  Some rules might have low support – very low example density compared with the

average example density of the whole instance space. In an extreme case, a rule might have 100% accuracy supported by only a few examples. Such rules have high accuracy but low confidence. In our system, we put more new training data into the areas covered by such low support rules. A threshold is used to determined if a rule has low support or not. For each low support rule, a fixed number of samples are randomly generated from the space covered by the rule.

- **Sampling the neighborhood of counterexamples**

  Each learned rule will be supported by some training examples, but also could cover counterexamples – examples in the opposite class. The technique of sampling the neighborhood of counterexamples is not new, many selective sampling methods (e.g., the Windowing technique for C4.5) have already used this idea. Counterexamples represent two possible situations: positively, new concepts could be found around such data points; negatively, such data points could be just noise. In our implementation, the 3 nearest neighbors of a counterexample are selected and 3 new samples are generated by calculating the middle points between the counterexample and these 3 neighbors. Figure 2.7 shows two counterexamples and their neighbor area.

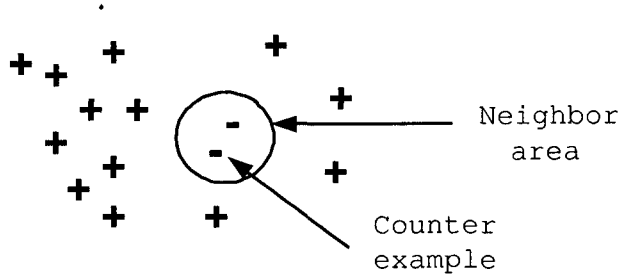


Figure 2.7: Sampling the neighborhood of counterexamples

- **Sampling Default Rule areas**

  The C4.5 rule learner will include all accurate enough (the default minimum accuracy is 50%) rules sorted by class. There is always a default rule which covers the rest of instance space. We found those default areas are just the *Uncertainty Regions*, which can have high information value.

2.3 Experiments

Experiments have been done to answer the following questions:

1. is selective sampling better than random sampling, as is often claimed?

2. which selective sampling method is best?

An artificial test environment is built for initially investigating these questions. The key benefit of an artificial environment is that the real target concepts are known in advance because we have generated them. This enables accurate evaluations. Artificial test environments are not real domains, but are designed to resemble real problems. Ideally, artificial environments will share as many properties as possible with their real counterparts, e.g., the nature and amount of noise. Artificial environments have some advantages over real ones:

- it is impossible to know exactly the true concepts of a real problem, but it is possible for an artificial environment because we generate those target concepts.
- the number and size of dimensions can be controlled.
- arbitrary noise can be added.

2.3.1 Classification Thresholds

To avoid the effects of randomness and noise, the same instance will be labelled multiple times (10 in this experiment). The final class for the example will be decided by a threshold on the ratio of *positive vs negative* examples. Suppose, for example, the threshold is 4 out of 10; this means an instance will be classified as positive if the number of positive labels it receives is greater than or equal to 4 (out of 10).

2.3.2 Two-dimensional experiments

Two dimensional artificial environment setup

To allow results to be visualized easily, we begin with an artificial environment with two continuous dimensions. 9 rectangles are put into the two dimensional space. Examples outside those rectangles will always be classified as *negative*. Examples falling into those rectangles will be classified as *positive* with a probability. Those probabilities vary from 0.1 to 0.9. Figure 2.8 is the visualized scenario. To simulate real applications, small transition areas between each *positive* box and the *negative* background are set up. For example, the positive class probability of the transition areas surrounding the 0.9 box will decrease gradually from 0.9 down to 0. All selective sampling methods start with 675 uniformly sampled initial examples. In each iteration, 50 more examples are added to the training set.

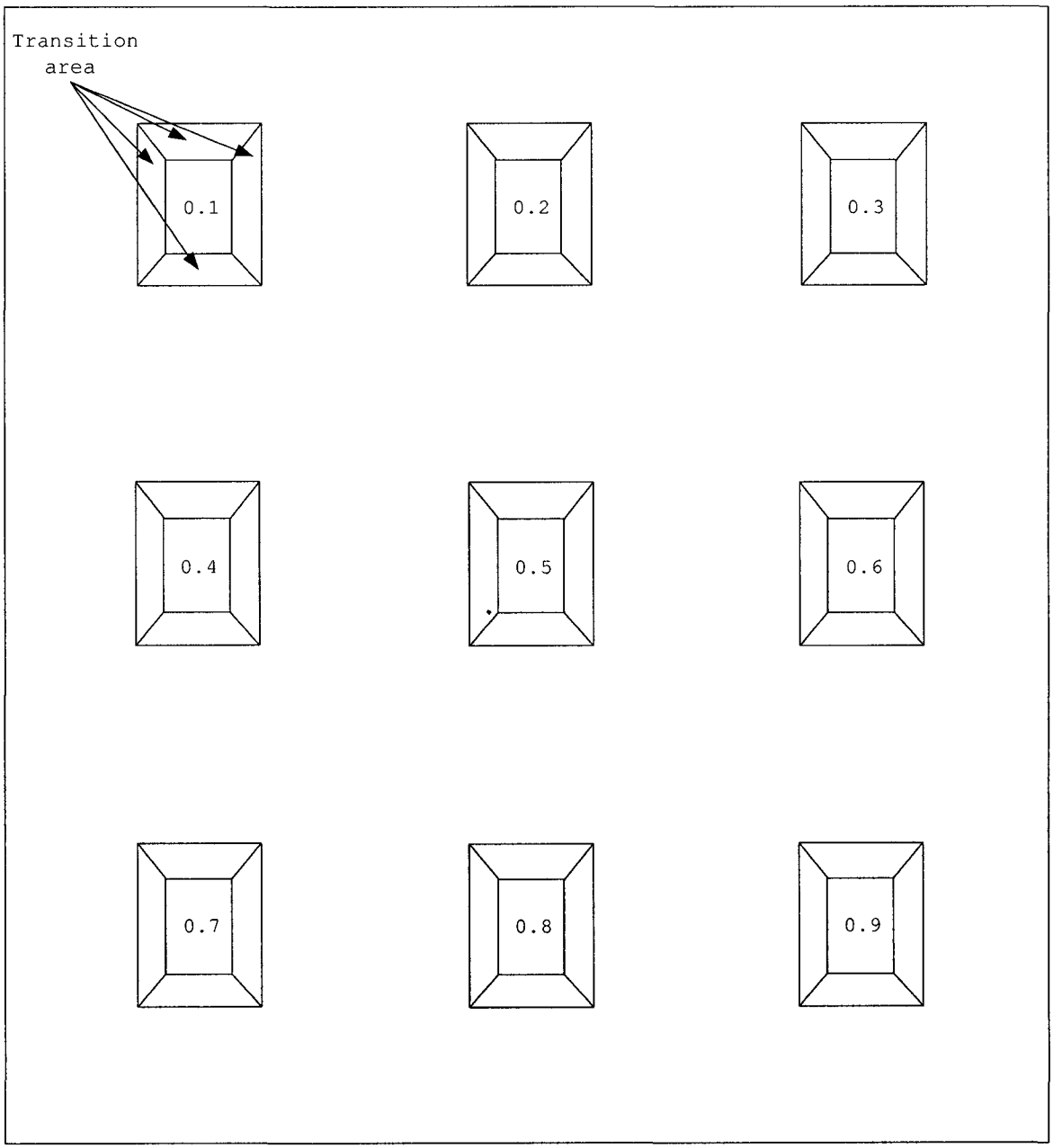
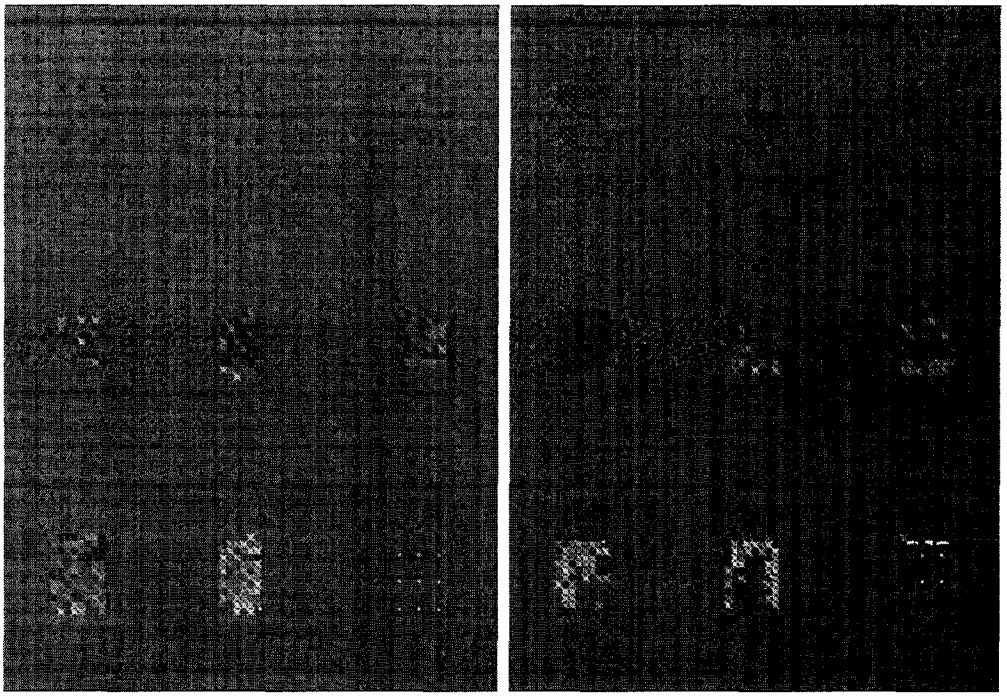
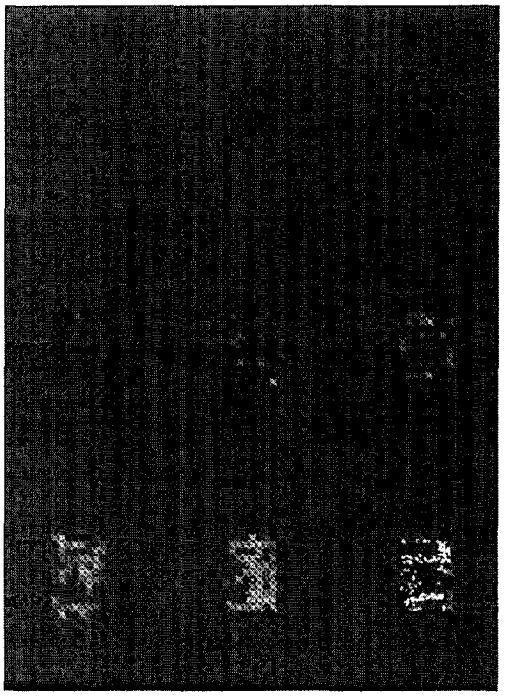


Figure 2.8: 2-D artificial scenario for sampling preference comparison



(a) Uncertainty (b) Bagging



(c) Boosting

Figure 2.9: Sampling preference comparison: data points — part 1

Sampling Preference of different methods

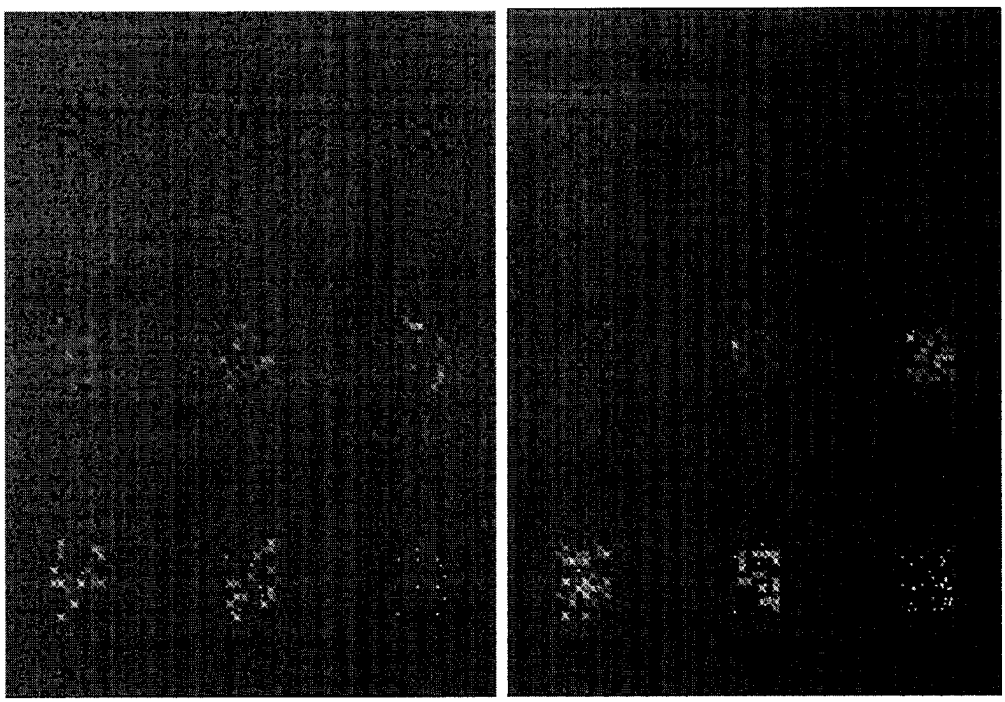
Figure 2.9 and Figure 2.10 show the visualized raw data distribution for the 5 selective sampling algorithms after 100 iterations. Each dot in these figures (and later similar figures) is a data point selected by the selective sampling algorithm; a white dot indicates a *positive* example; a black dot indicates a *negative* example. In these figures, some areas are marked by a colored "X", which means those area are impure regions whose shade indicates the ratio of positive to negative examples. Uncertainty sampling (Figure 2.9(a)) shows a preference for sampling areas in which the number of positive examples is similar to the number of negative examples, especially the box with probability 0.5. It prefers sampling inside of those boxes. Bagging (Figure 2.9(b)) and boosting (Figure 2.9(c)) prefer sampling the boundaries of boxes, rather than the interior of the boxes (in Figure 2.9(b), boxes 0.5, 0.6, 0.8 and 0.9 are sampled mainly around their boundaries; in Figure 2.9(c), boxes 0.4, 0.5, 0.6, 0.7, and 0.9 are sampled mainly around their boundaries). BootstrapLV (Figure 2.10(a)) looks just like random sampling. Rule-based sampling (Figure 2.10(b)) has a very good sampling behavior, concentrating on the boundaries of boxes. Figures 2.11 and 2.12 show the visualized rules (solid white boxes) that were learned with the data points in Figures 2.9 and 2.10. Almost all boxes whose probability is greater than or equal to 0.5 are discovered by the learning methods. One noticeable thing is that *uncertainty sampling* (Figure 2.11(a)) does not mark the box 0.5 as positive because *negative* and *positive* data points inside this box have equal representation.

Evaluation

Classification accuracy is the most commonly used criterion to evaluate a learning algorithm: induce a model using training data, and measure the classification accuracy on test data. Accuracy is computed as follows: $accuracy = correct/total$, where *correct* is the number of correctly classified examples and *total* is the number of all examples.

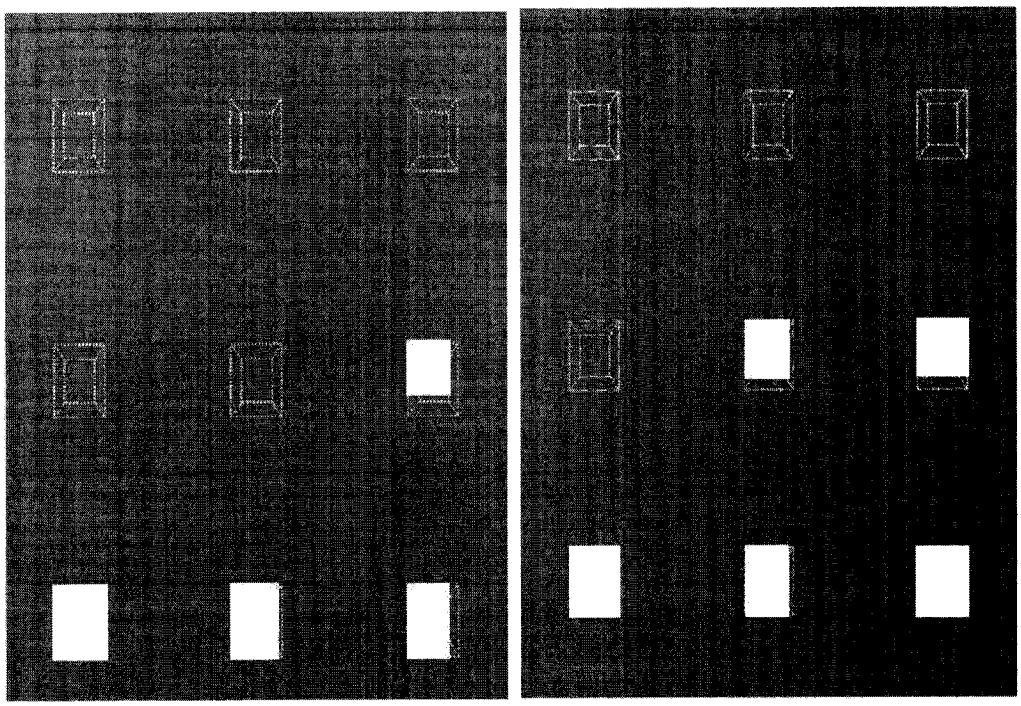
Accuracy has been proved to be unacceptable for many real applications[11]. Therefore, in our studies we will use TP rate = TP / (TP + FN) and FP rate = FP / (FP + TN), where TP is the number of true positives (examples classified correctly as positive), FP is the number of false positives (examples misclassified as positive), TN is the number of true negatives (examples classified correctly as negative), and FN is the number of false negatives (examples misclassified as negative). For the 2-D artificial experiments, the TP rates and FP rates are calculated by testing on a large test set.

To present clear results, a *Bezier* function is used to smooth all FP rate and TP rate plots



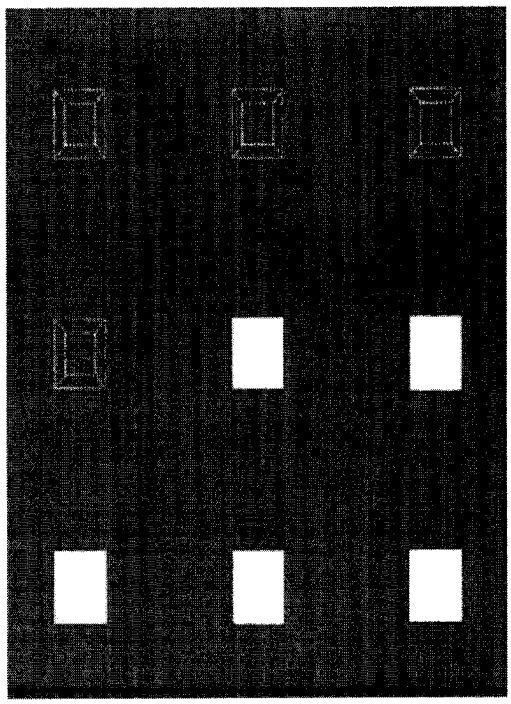
(a) BootstrapLV (b) Rule-based

Figure 2.10: Sampling preference comparison: data points — part 2



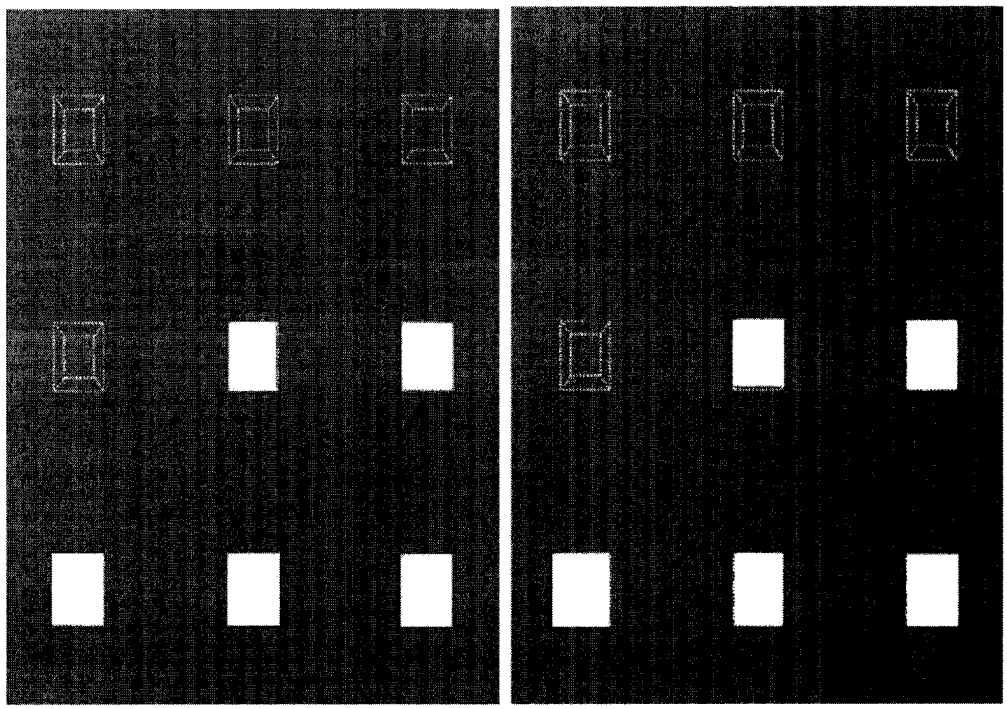
(a) Uncertainty

(b) Bagging



(c) Boosting

Figure 2.11: Sampling preference comparison: generated rules — part 1



(a) BootstrapLV (b) Rule-based

Figure 2.12: Sampling preference comparison: generated rules — part 2

in this thesis. According to *Paul Bourke*'s review([4]), the formula for a *Bezier* function is:

$$B(u) = \sum_{K=0}^{N} P_k \frac{N!}{k!(N-k)!} u^k (1-u)^{N-k} \quad for \ 0 \leq u \leq 1$$

where u is a continuous variable ranging from 0 to 1. A curve $B(u)$ is built based on the *Bezier* function with B(0) = P_0, B(1) = P_N, where $N+1$ is the number of total examples, and P_k is the data point with index k. Figure 2.13 (by Paul Bourke[4]) is an example of Bezier function: the original input points are P_0, P_1, P_2, P_3 and P_4; a continuous Bezier curve smooths the connections of those points.

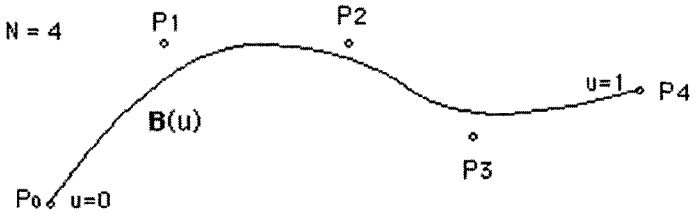


Figure 2.13: Example of a Bezier function

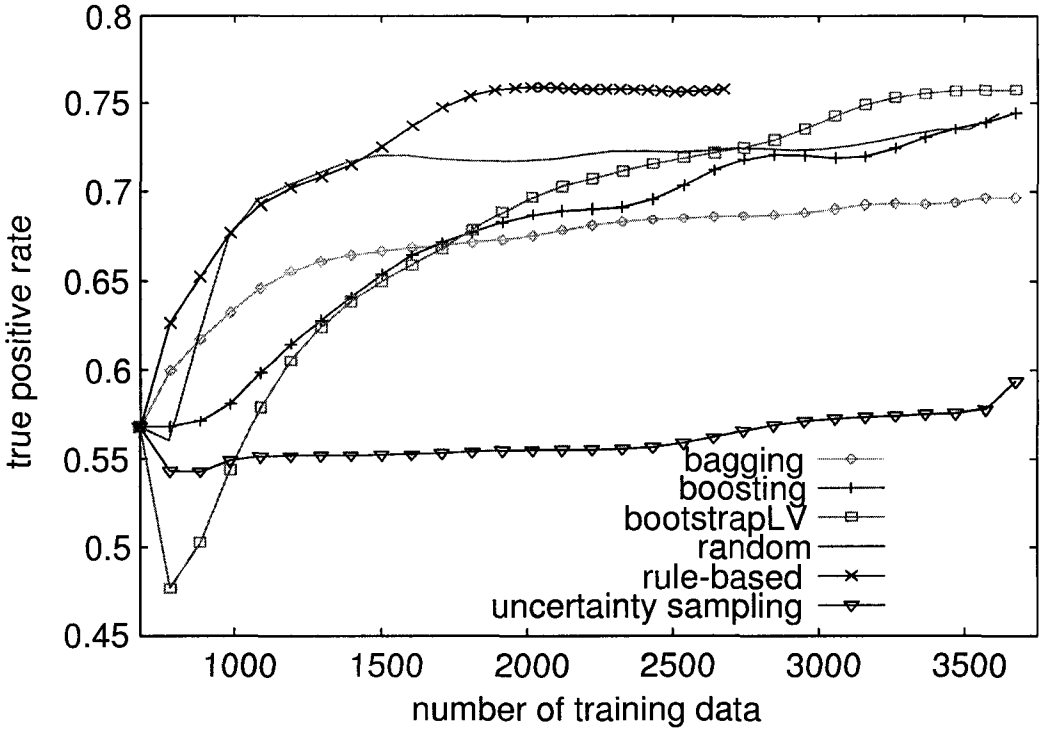
A one-sided t-test will be used to test the *statistical significance test* of the sampling methods at specific training set sizes. This is done as follows. At training set size x, methods A has 5 experimental numbers $\{a_i, 1 \leq i \leq 5\}$ representing 5 runs. Methods B also has 5 numbers $\{b_i, 1 \leq i \leq 5\}$. The 5 paired numbers $\{c_i = a_i - b_i, 1 \leq i \leq 5\}$ represent the difference of methods A and B at training set size x. A one-sided t-test ($\alpha = 0.1$) is applied to $\{c_i, 1 \leq i \leq 5\}$ to test the null hypothesis that the difference is equal to 0. If the statistical value is greater than 1.53 ($t_{.90}$, *degrees of freedom* = 4), the null hypothesis will be rejected and we can claim that methods A and B have a significant ($\alpha = 0.1$) performance difference at training set size x. In this thesis, if we claim that Method A is significantly ($\alpha = 0.1$) better than all other methods at training set size x, we actually mean in individual pairwise comparisons with the other 5 sampling methods in these experiments, Methods A is significantly better than the others at training set size x. If we claim that Method A and B are significantly ($\alpha = 0.1$) better than Method C and D at training set size x, we actually mean that all four pairwise tests indicate a significiant difference (Method A is significantly($\alpha = 0.1$) better than Method C *and* Method A is significantly ($\alpha = 0.1$) better than Method D *and* Method B is significantly ($\alpha = 0.1$) better than Method C *and* Method B is significantly ($\alpha = 0.1$) better than Method D) at training set size x.

Figure 2.14 compares the TP rates of the 5 selective sampling methods in this artificial test. Random sampling is also included for comparison. The TP rates and FP rates shown are the average of 5 repetitions of the experiment. Figure 2.15 and figure 2.16 show the comparison of smoothed curve(average of 5 runs) and unsmoothed curve(average of 5 runs) for every sampling method. Throughout the whole thesis, each TP/FP rate figures will be followed by such a figure comparing the smoothed curve(average of 5 runs) and unsmoothed curve(average of 5 runs) for every sampling method.

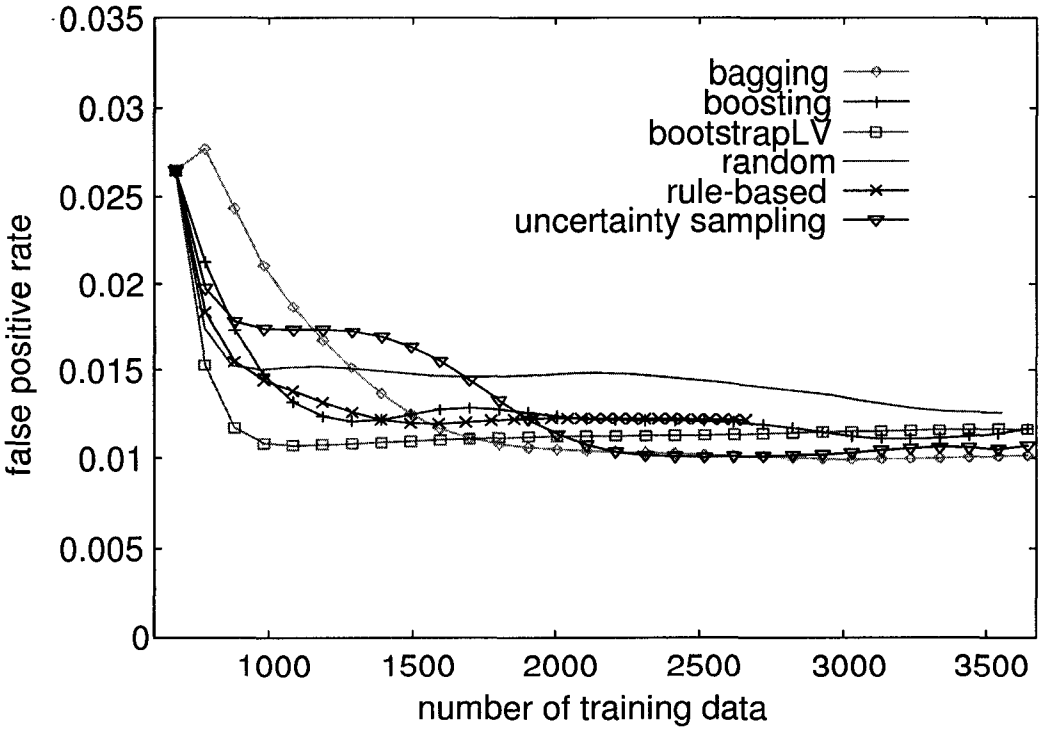
Uncertainty sampling has the worst performance in terms of TP rate (the performance difference is significant [$\alpha = 0.10$] at training set size 2500), mostly because it concentrates on the region where the real positive class probability is equal to 0.5. Bagging and boosting are better than uncertainty sampling (the performance difference is significant [$\alpha = 0.10$] at training set size 2500, in individual pairwise comparisons). Bagging and boosting both are concentrating on boundary sampling. Random sampling has a good TP rate curve (e.g, it significantly [$\alpha = 0.10$] beats all other methods except for rule-based sampling at training set size 1500), but its overall FP rate is the highest of all the 6 sampling methods (the performance difference is significant [$\alpha = 0.10$] at training set size 2500). The rule-based sampling method performs best in this artificial test environment. It is the fastest sampling method to reach a high and stable TP rate (e.g., it significantly [$\alpha = 0.10$] beats all other methods at training set size 2000), and it keeps a relatively low FP rate (but the difference is not shown to be significant [$\alpha = 0.10$] at training set size 2500 given our sample size). In Figure 2.14, the rule-based sampling method stopped before other sampling methods because the exclusive rule conversion algorithm cannot handle such a large number of rules. This is a limitation of the current implementation of the rule-based sampling method. In summary, in terms of TP rates, rule-based sampling is the best (e.g., it significantly [$\alpha = 0.10$] beats all other methods at training set size 2000), uncertainty is the worst (the performance difference is significant [$\alpha = 0.10$] at training set size 2500). In terms of FP rates, random is the worst (the performance difference is significant [$\alpha = 0.10$] at training set size 2500), the FP rates difference of all other sampling methods are not shown to be significant ($\alpha = 0.10$) at training set size 2500 given our sample size.

2.3.3 High-dimensional artificial tests

Figure 2.14 shows that random sampling is not bad compared with selective sampling methods in terms of TP rates in the 2-D artificial environment. We hypothesize that selective sampling methods will be better than random when the number of dimensions increases

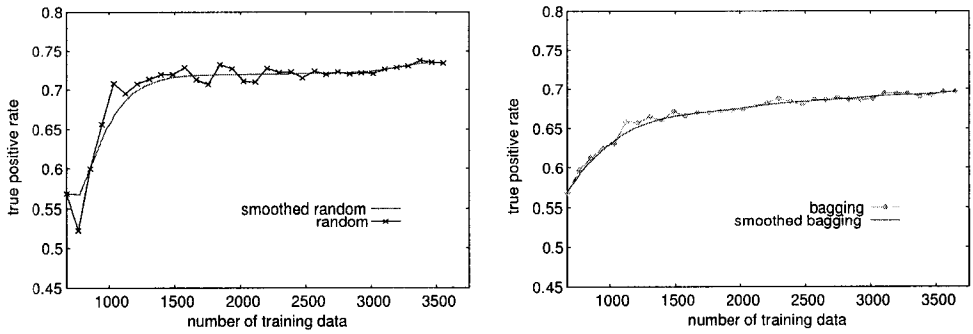


(a) TP rate



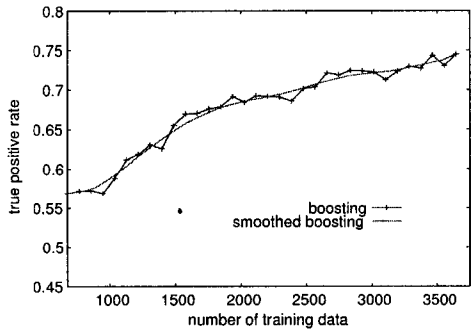
(b) FP rate

Figure 2.14: Sampling methods: TP and FP rates comparison

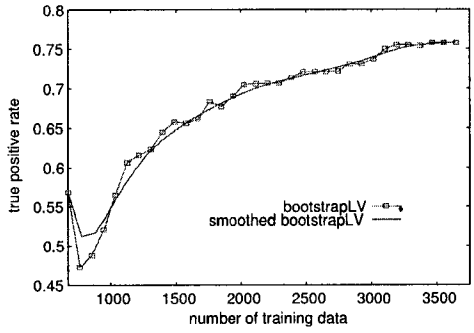


(a) Random

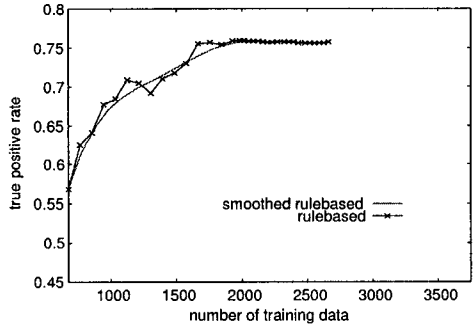
(b) Bagging



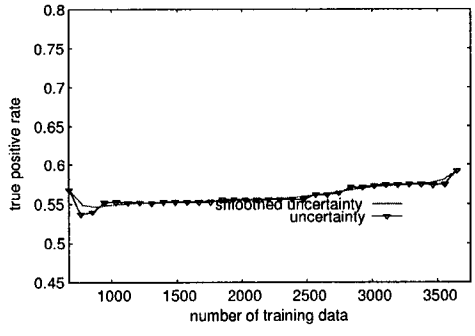
(c) Boosting



(d) BootstrapLV

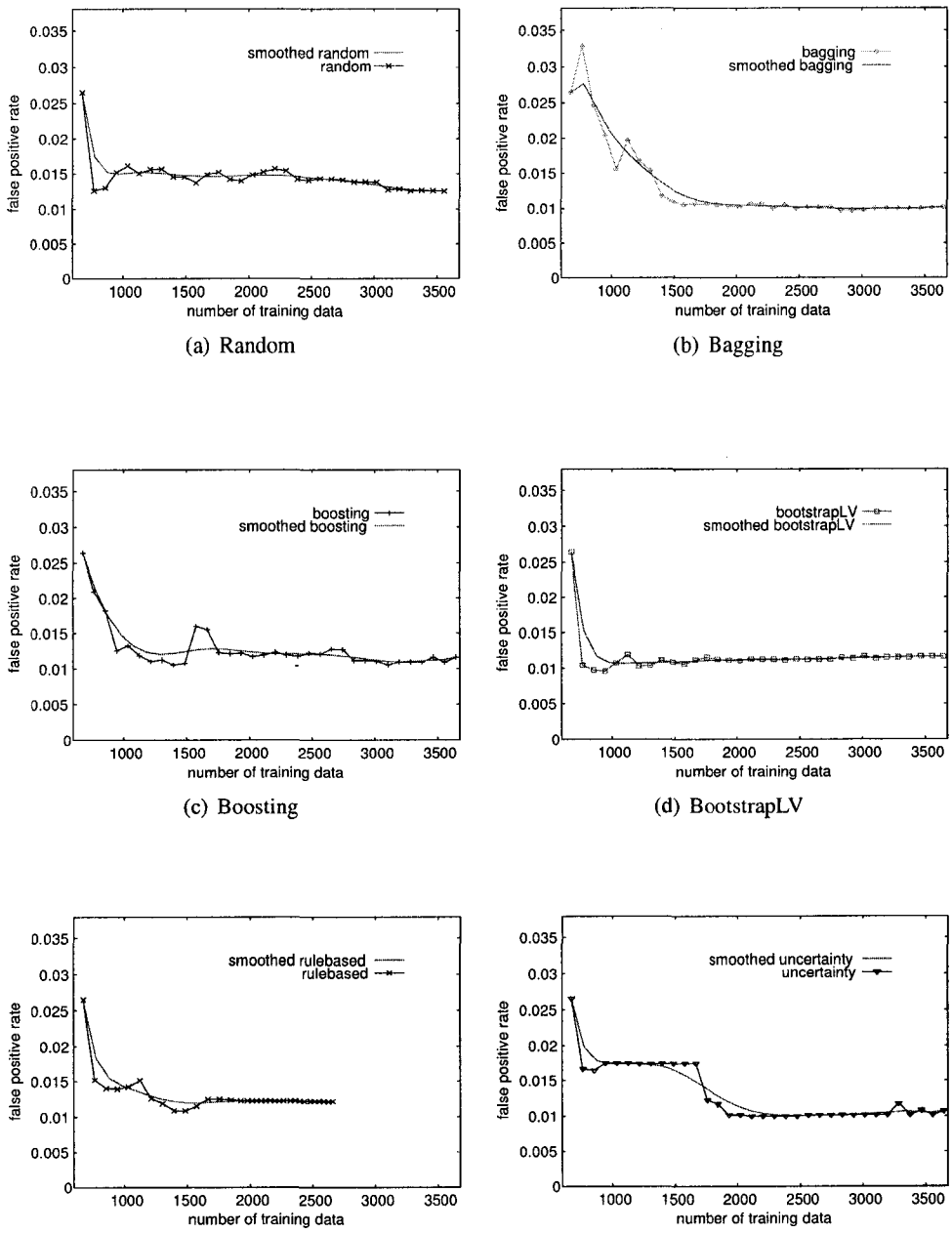


(e) Rulebased



(f) Uncertainty

Figure 2.15: 2-D TP rate: smoothed *vs.* unsmoothed



(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 2.16: 2-D FP rate: smoothed *vs.* unsmoothed

because random sampling must sample sparsely. For each higher dimensional space, target concepts will be generated, each of which is a region with a certain width in each dimension; the total volume of all disjoint regions is fixed to be 40% of the whole instance space. The positive probability of each target concept region is equal to 1.0 and each region is surrounded by a very thin transition area.

Evaluation

TP and FP rates will be calculated exactly by comparing the geometry of the learned rules with the *target concepts*. Figure 2.17 shows how to get TP, TN, FP and FN by comparing a learned rule with the target concept.

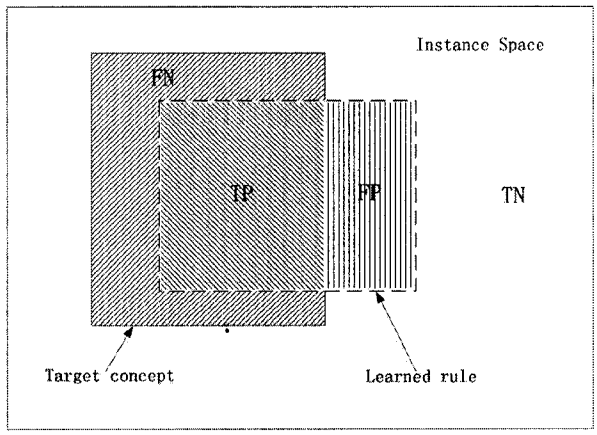


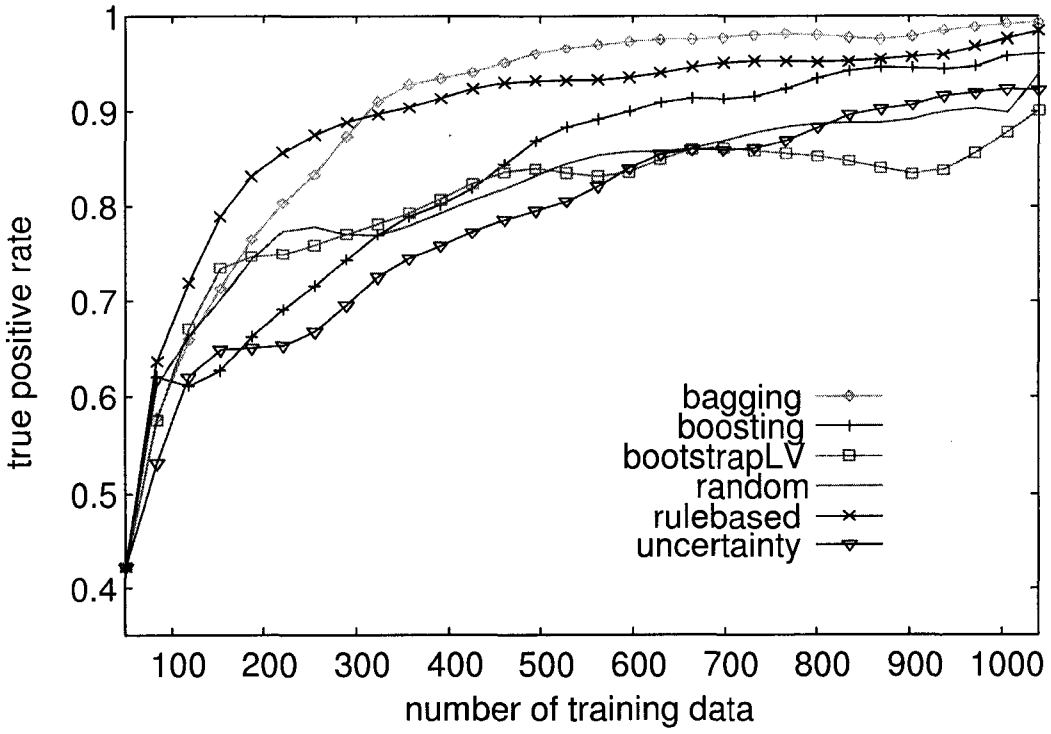
Figure 2.17: Evaluation by comparing the learned rule with the target concept

Experiment setup and results

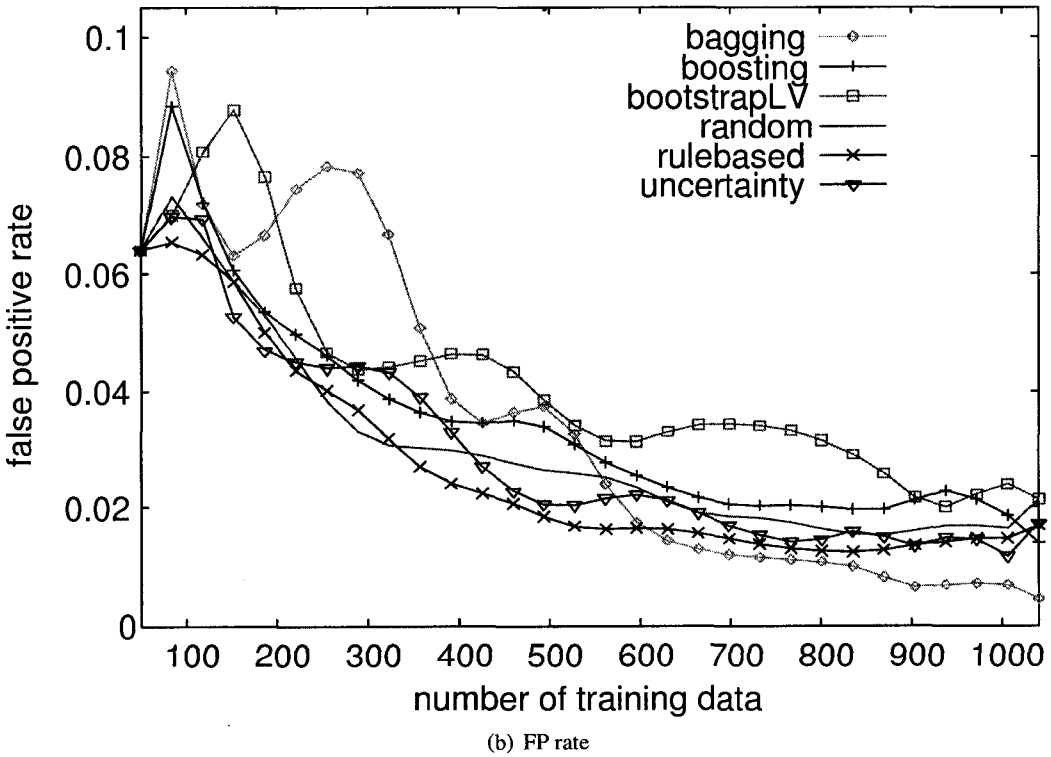
The initial training set is 50 examples and 10 more examples are added to the training set on each iteration.

- 4-D
  - TP rate (Figure 2.18(a))

    Bagging and rule-based sampling are the best two methods (e.g., bagging significantly beats all others [$\alpha = 0.10$] except for boosting at training set size 400; boosting significantly beats all others [$\alpha = 0.10$] except for bagging at training set size 400 as well). The TP rates difference of all other sampling methods are not shown to be significant ($\alpha = 0.10$) at training set size 400 given our sample size.

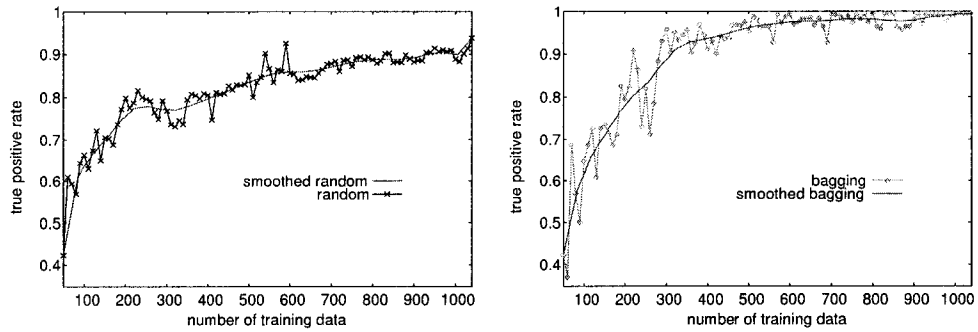


(a) TP rate



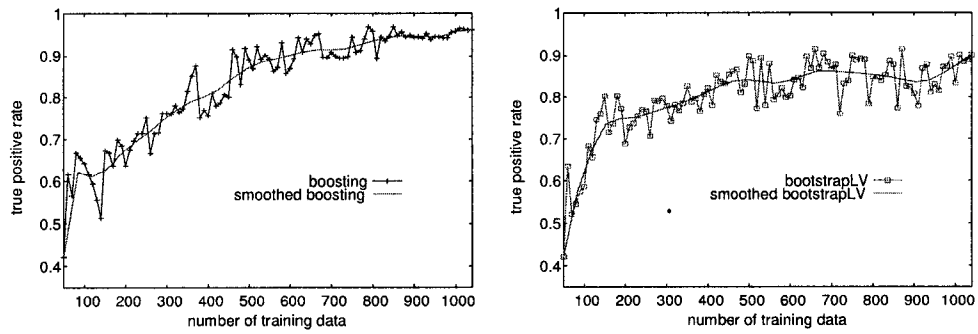
(b) FP rate

Figure 2.18: TP and FP rates comparison in a 4-D space



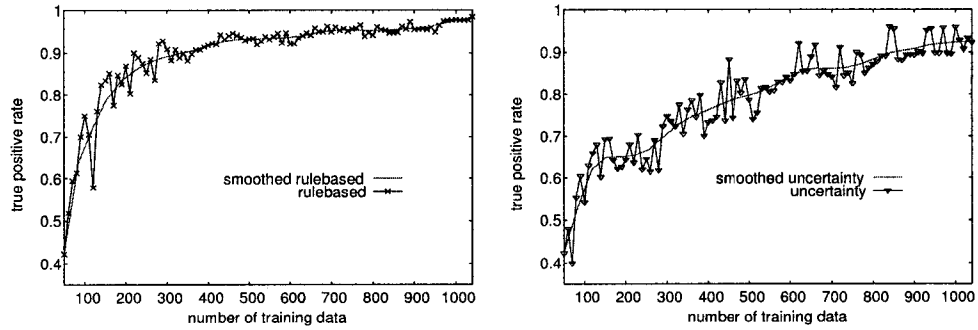
(a) Random

(b) Bagging



(c) Boosting

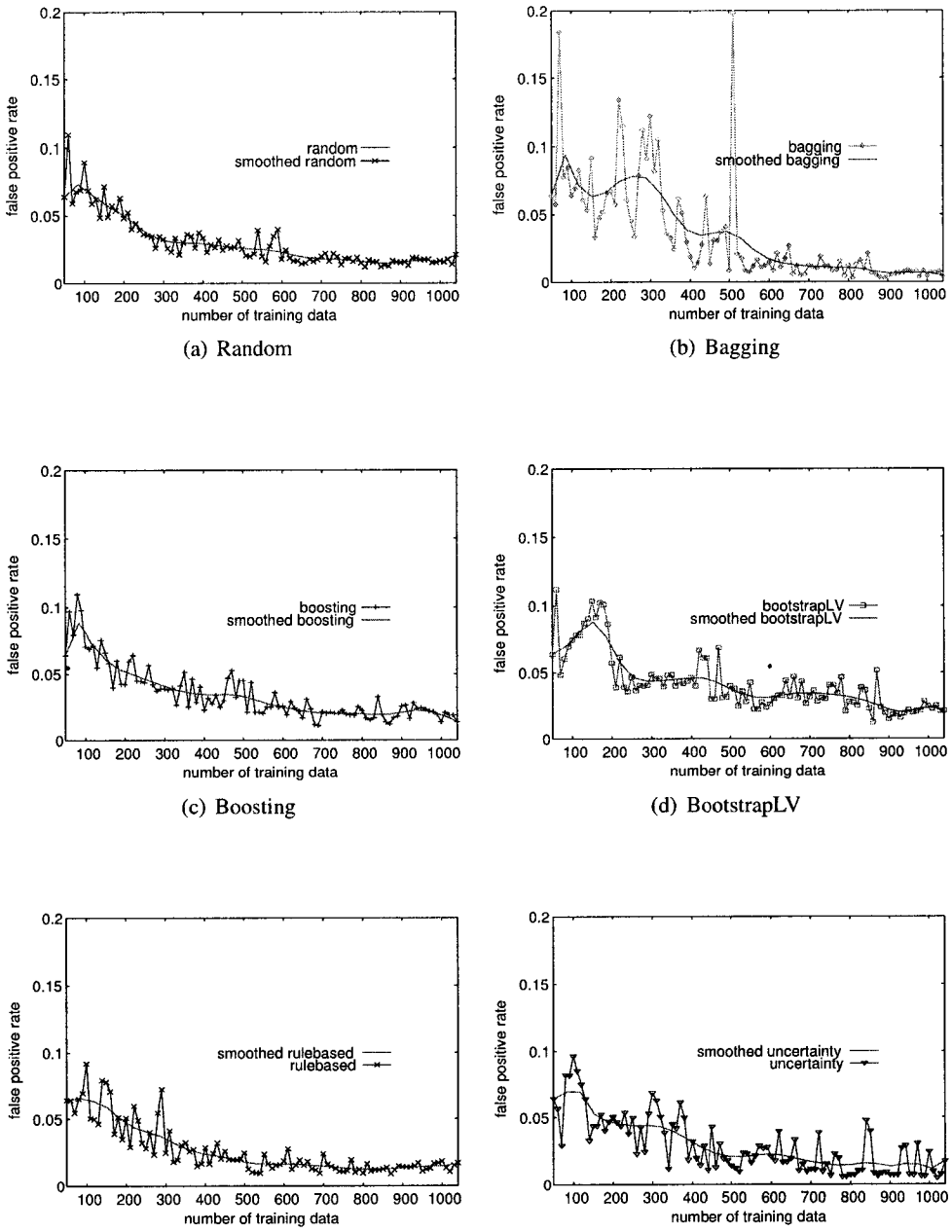
(d) BootstrapLV



(e) Rulebased

(f) Uncertainty

Figure 2.19: 4-D TP rate: smoothed *vs.* unsmoothed

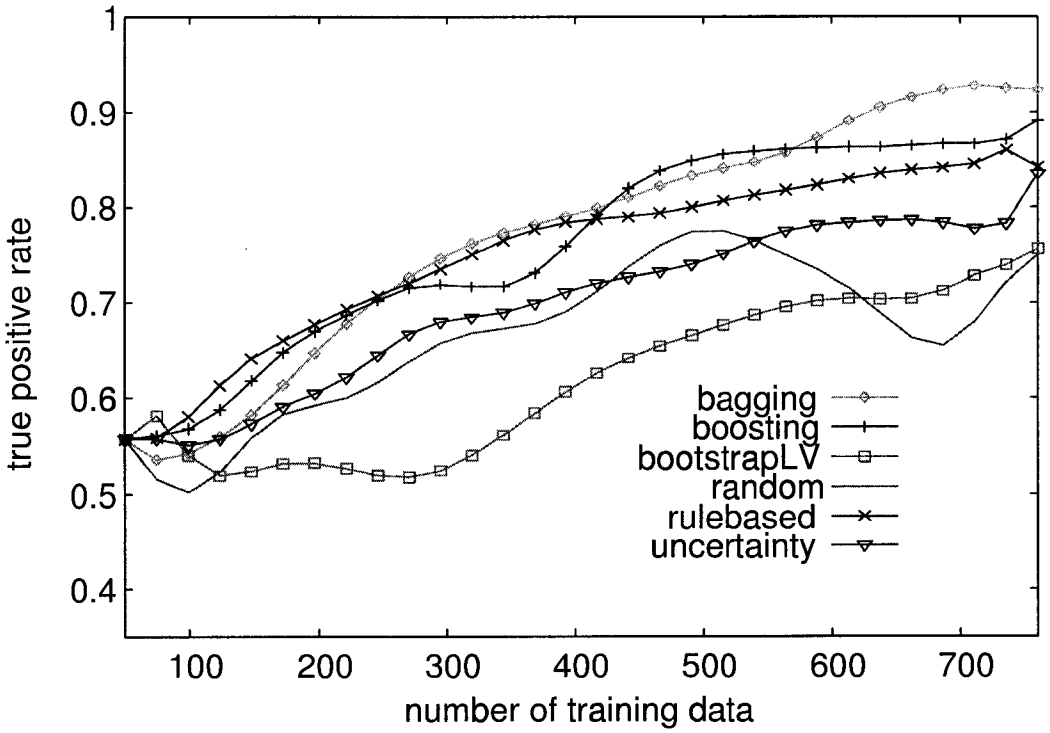


(a) Random (b) Bagging

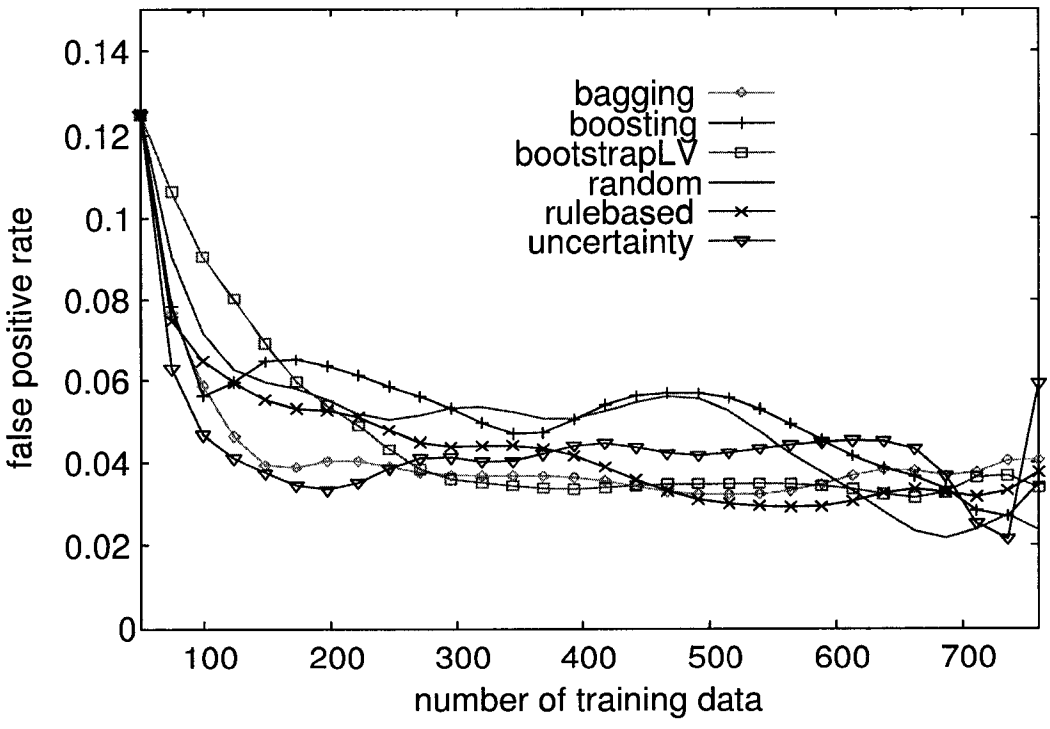
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 2.20: 4-D FP rate: smoothed *vs.* unsmoothed

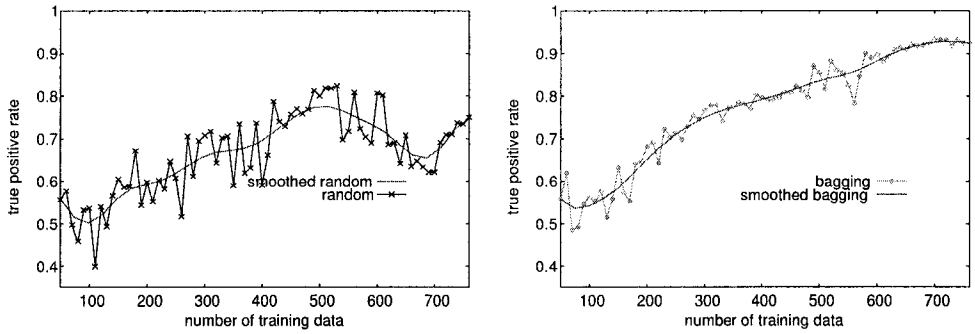


(a) TP rate



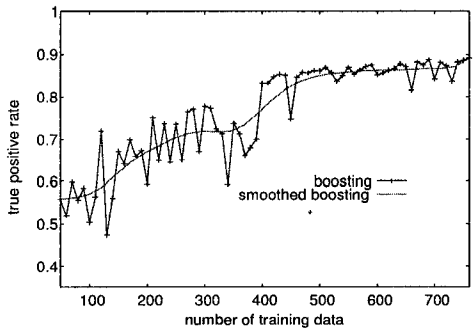
(b) FP rate

Figure 2.21: TP and FP rates comparison in a 6-D space

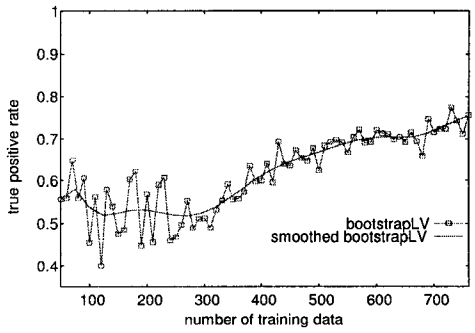


(a) Random

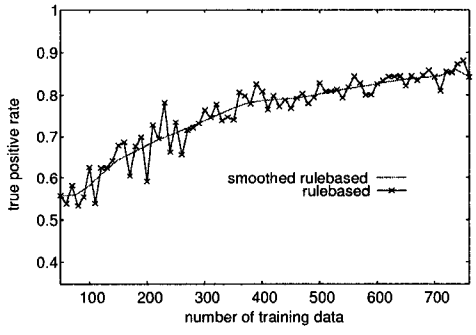
(b) Bagging



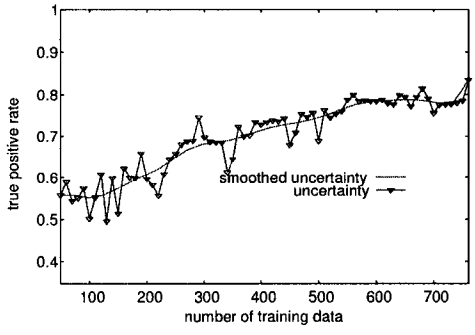
(c) Boosting



(d) BootstrapLV

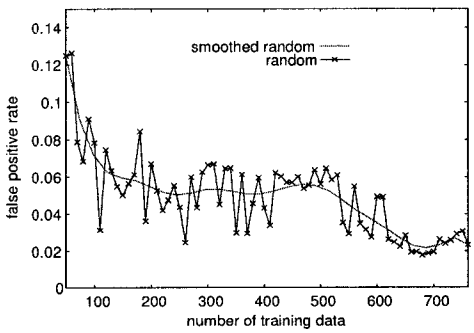


(e) Rulebased

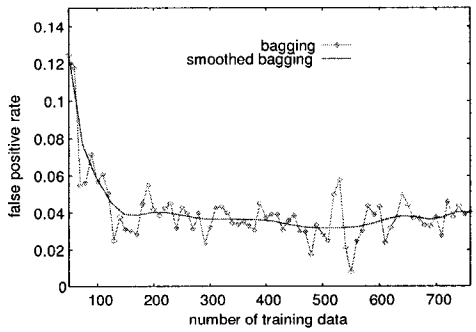


(f) Uncertainty

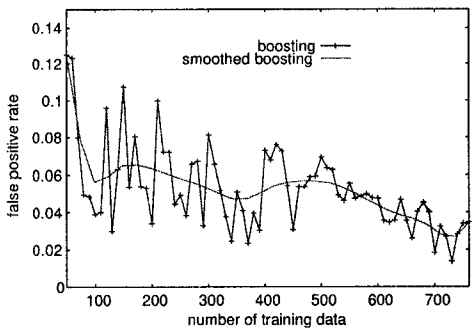
Figure 2.22: 6-D TP rate: smoothed *vs.* unsmoothed



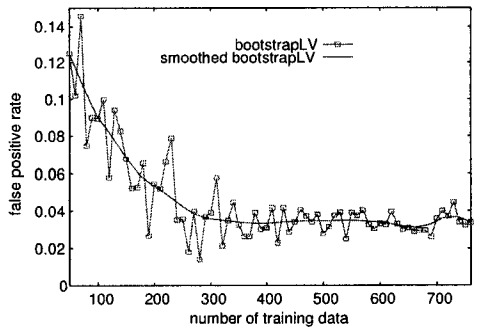
(a) Random



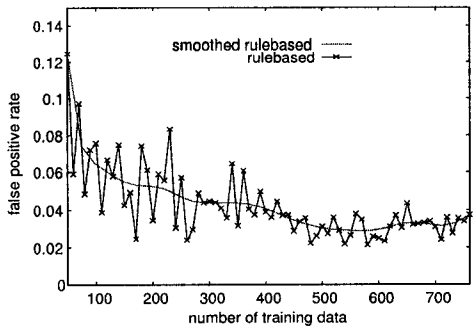
(b) Bagging



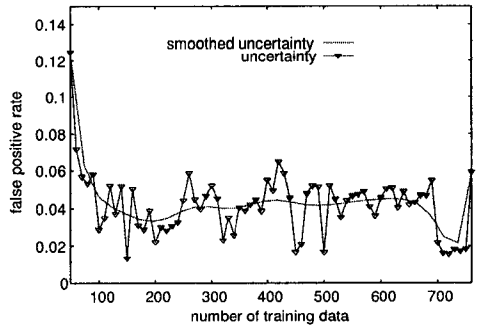
(c) Boosting



(d) BootstrapLV

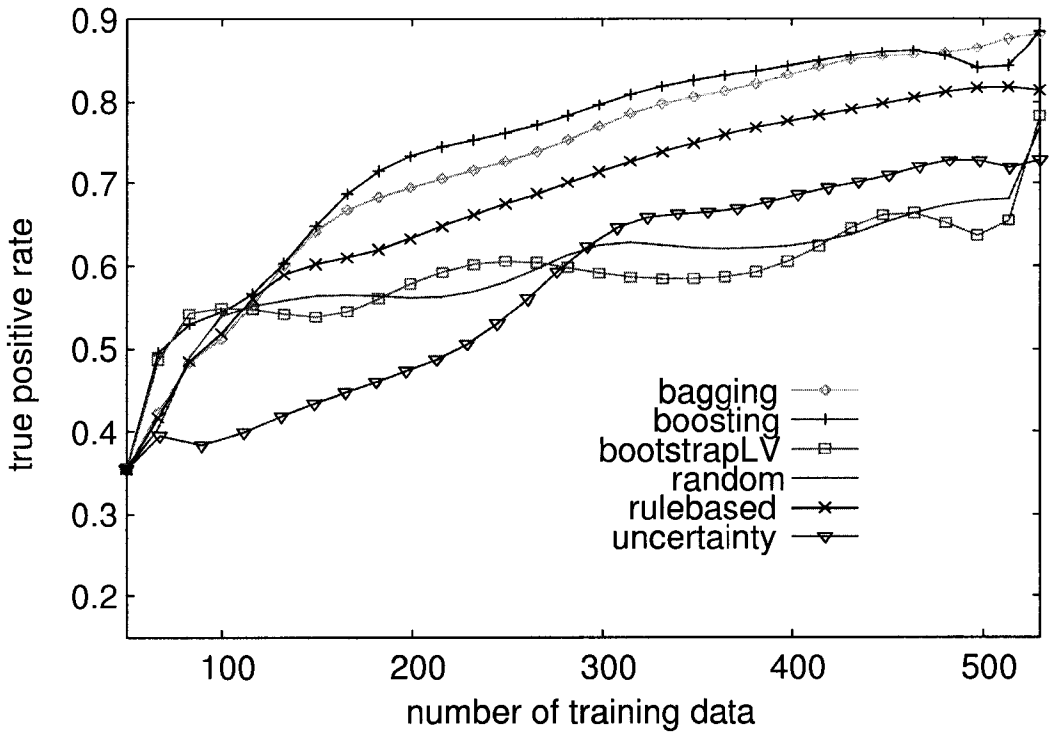


(e) Rulebased

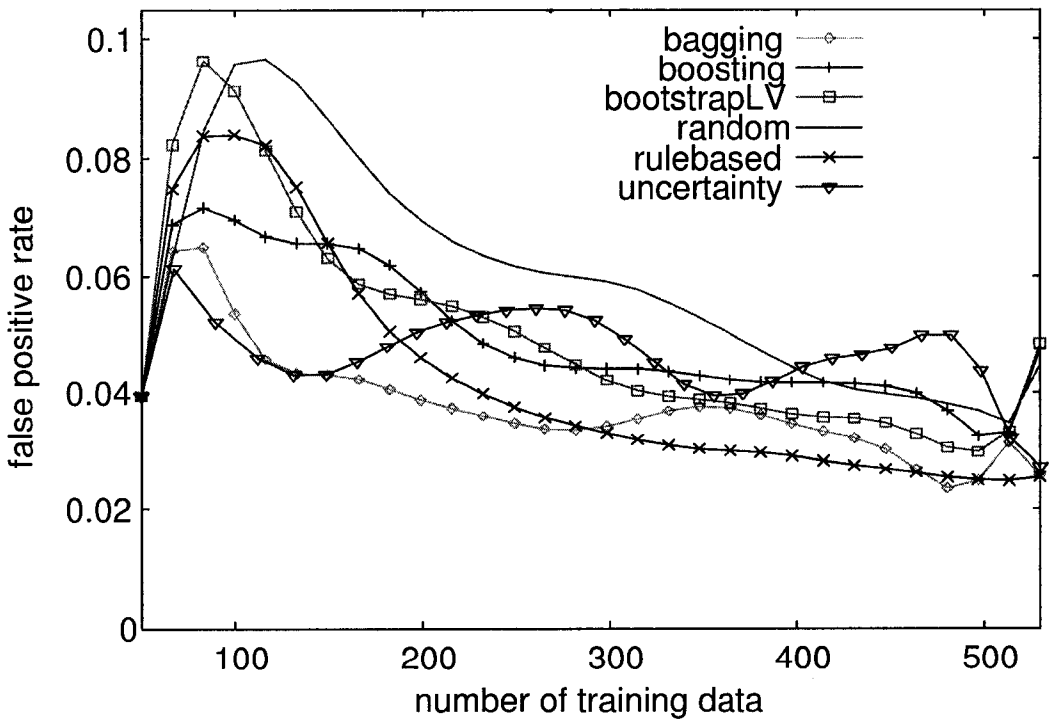


(f) Uncertainty

Figure 2.23: 6-D FP rate: smoothed *vs.* unsmoothed

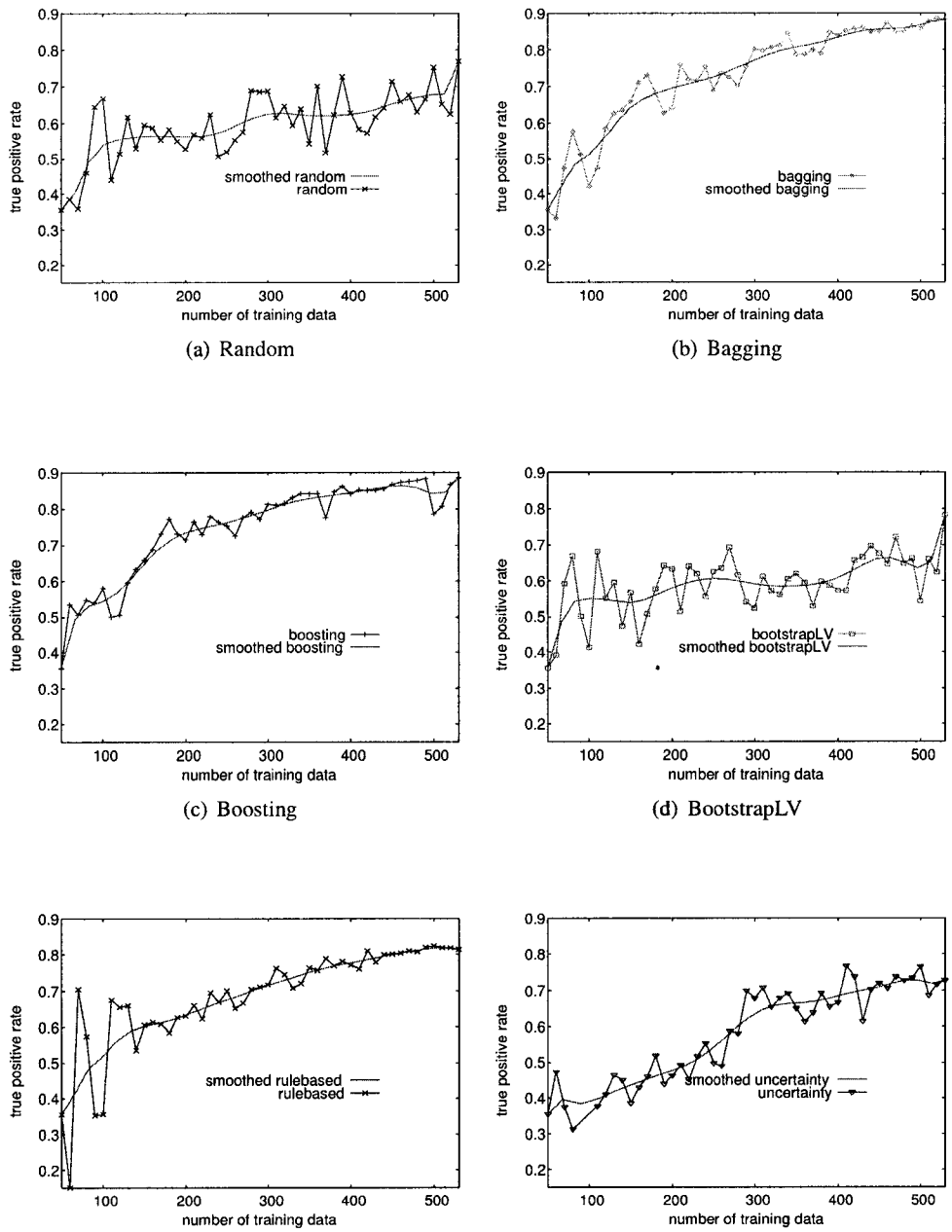


(a) TP rate



(b) FP rate

Figure 2.24: TP and FP rates comparison in a 8-D space

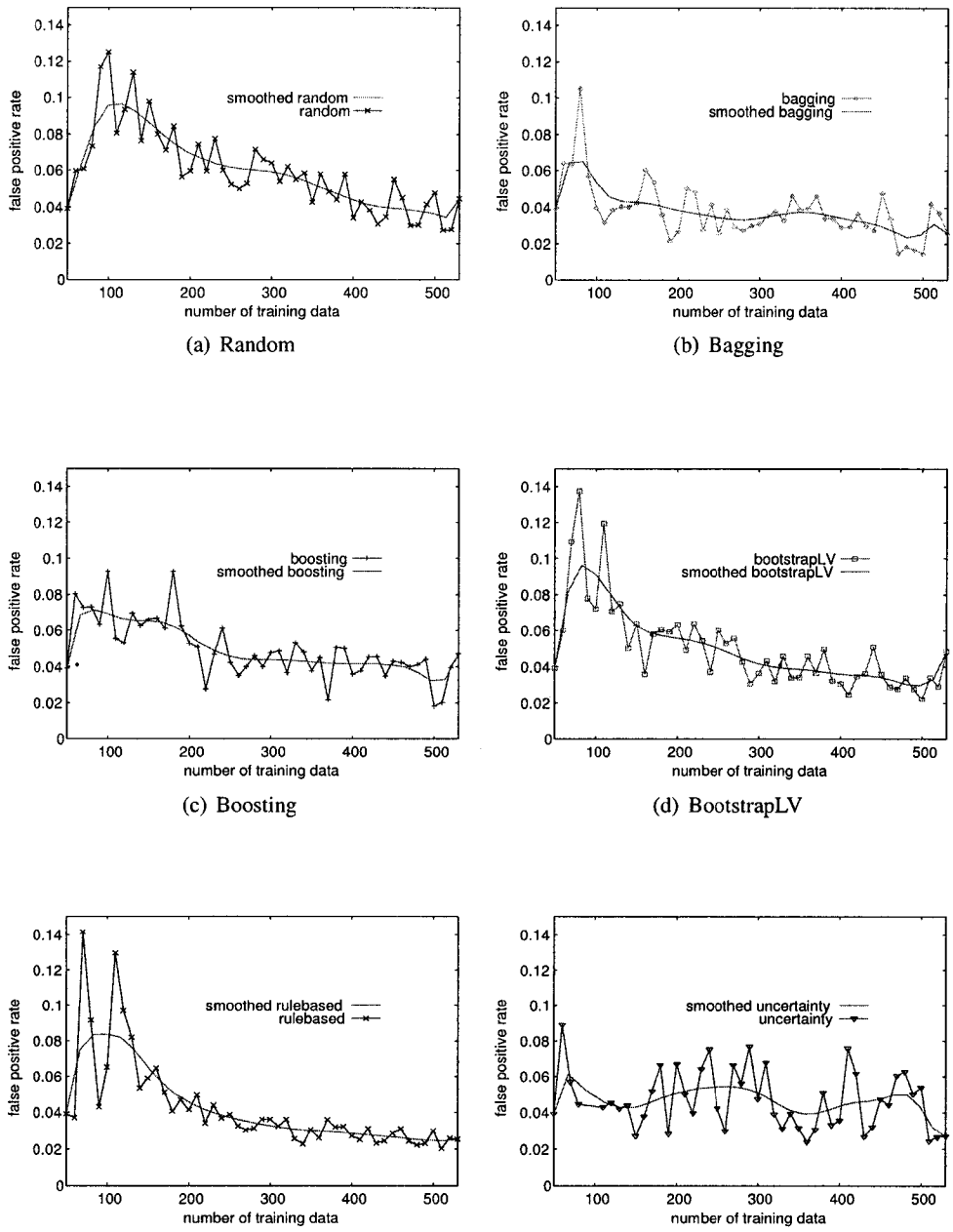


(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 2.25: 8-D TP rate: smoothed *vs.* unsmoothed

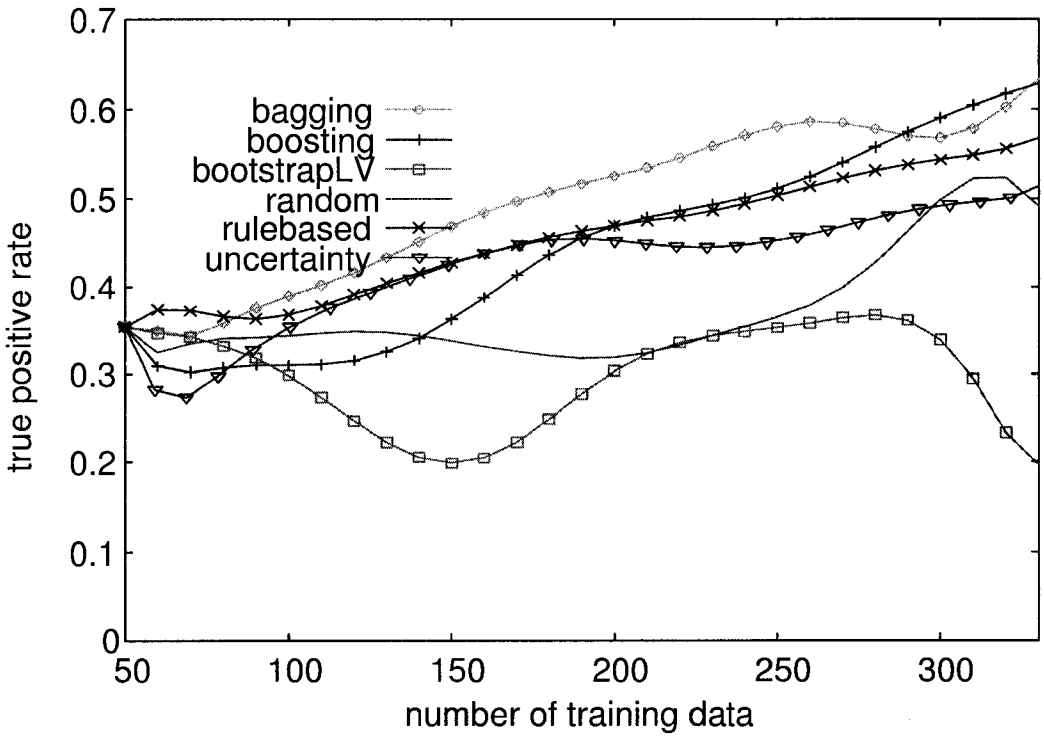


(a) Random (b) Bagging

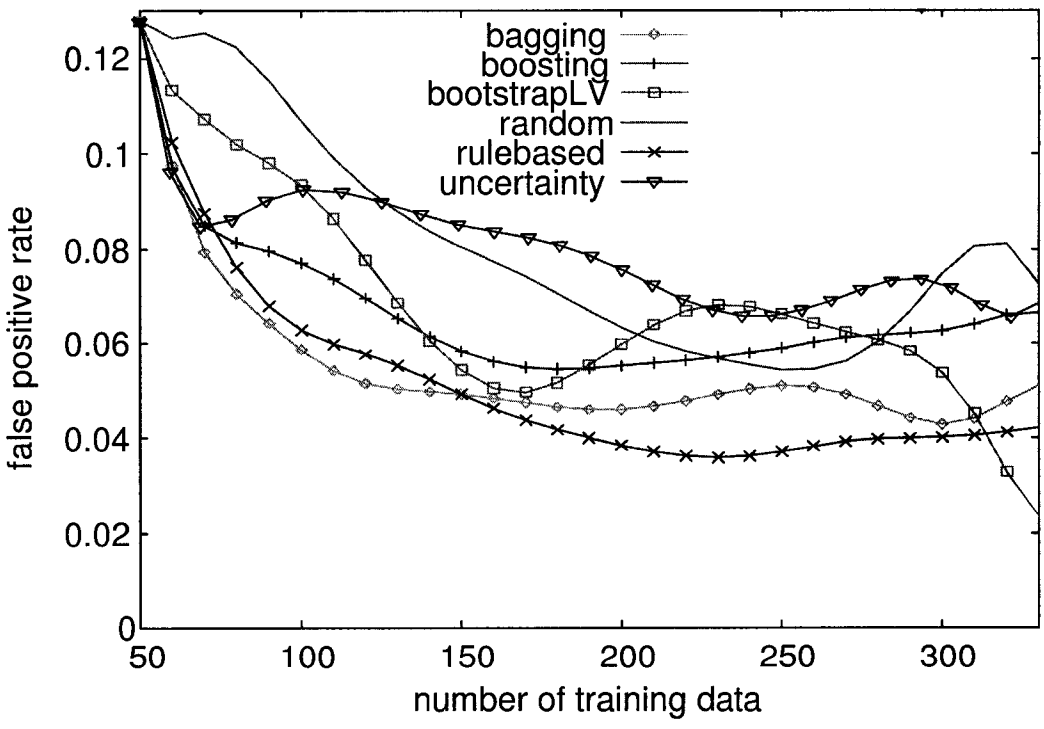
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 2.26: 8-D FP rate: smoothed *vs.* unsmoothed

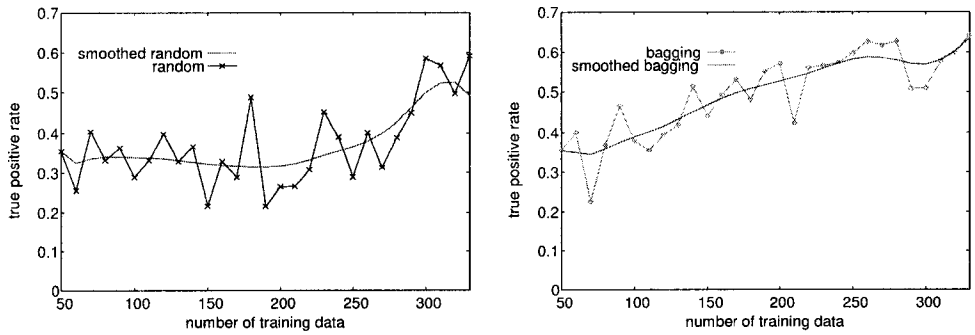


(a) TP rate



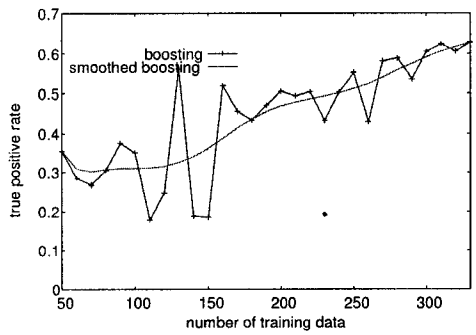
(b) FP rate

Figure 2.27: TP and FP rates comparison in a 10-D space

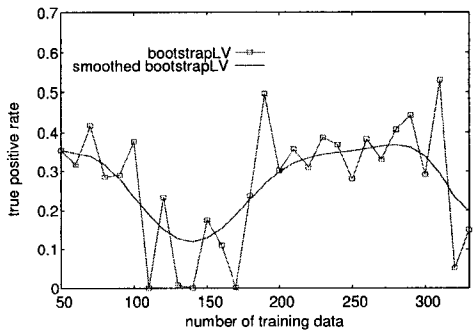


(a) Random

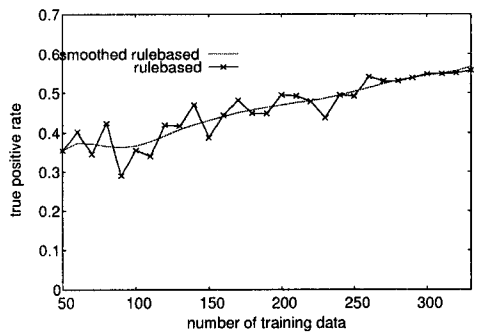
(b) Bagging



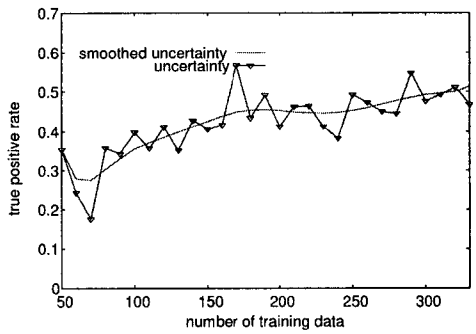
(c) Boosting



(d) BootstrapLV

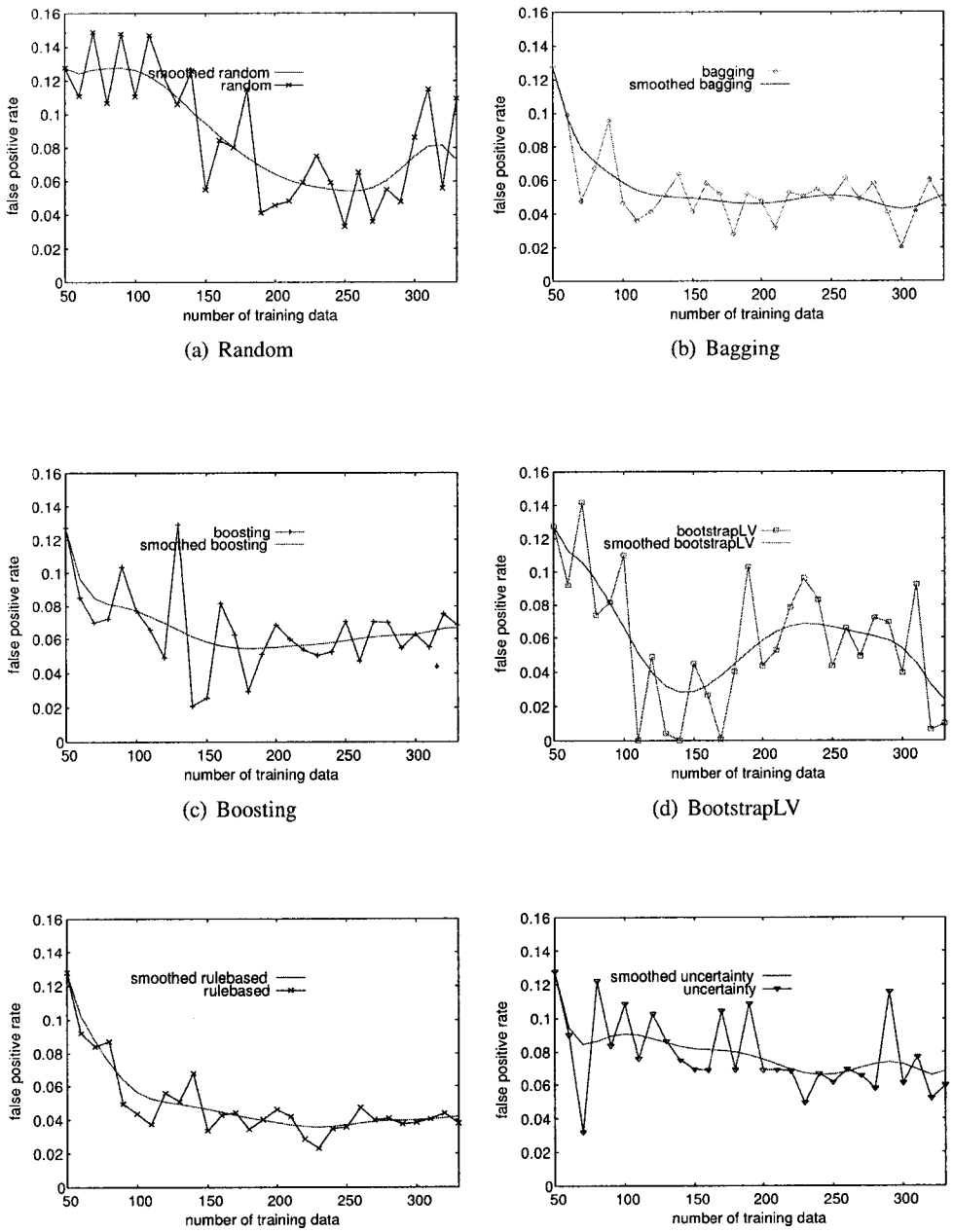


(e) Rulebased



(f) Uncertainty

Figure 2.28: 10-D TP rate: smoothed *vs.* unsmoothed



(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 2.29: 10-D FP rate: smoothed *vs.* unsmoothed

- FP rate (Figure 2.18(b))

  Bagging is not stable in the early stages, but achieves the lowest FP rate eventually (the performance difference is significant [$\alpha = 0.10$] at training set size 1030); rule-based sampling beats random after training set size is bigger than 400 (e.g., it significantly [$\alpha = 0.10$] beats random at training set size 500); random sampling, uncertainty sampling and boosting has no significant ($\alpha = 0.10$) FP rates difference at training set size 600 given our sample size. BootstrapLV is the worst in terms of FP rates (the performance difference is significant [$\alpha = 0.10$] at training set size 700).

- Overall

  Bagging and rule-based sampling are the overall winners.

• 6-D

- TP rate (Figure 2.21(a))

  Boosting, bagging and rule-based sampling are similar (no significant [$\alpha = 0.10$] difference at training set size 600 given our sample size); but each of them is better than random sampling (the performance difference is significant [$\alpha = 0.10$] at training set size 600); BootstrapLV is the overall worst (the performance difference is significant [$\alpha = 0.10$] at training set size 400).

- FP rate (Figure 2.21(b))

  There is no significant ($\alpha = 0.10$) difference between the different sampling methods, e.g, at training set size 700.

- Overall

  Selective sampling methods (except for Uncertainty and BootstrapLV) begin to be superior to random sampling (the performance difference is significant ($\alpha = 0.10$) at training set size 600).

• 8-D

- TP rate (Figure 2.24(a))

  Two groups are formed and the sampling methods within each group have similar TP rates. The first group consists of boosting, bagging and rule-based sampling. The second group has lower TP rates than the first group; it consists of uncertainty sampling, random and BootstrapLV.

- FP rate (Figure 2.24(b))

  Random sampling is the worst (the performance difference is significant [$\alpha = 0.10$] at training set size 200).

- Overall

  Boosting, bagging and rule-based sampling are better than random sampling (the performance difference is significant [alpha = 0.1] at training set size 300).

- 10-D

  - TP rate (Figure 2.27(a))

    Bagging, boosting, rule-based and uncertainty sampling all beat random sampling (the performance difference is significant [$\alpha = 0.10$] at training set size 200).

  - FP rate (Figure 2.27(b))

    Bagging and rule-based sampling seem to be better than other sampling methods, due to limited number of samples and iterations, above claim is not significant ($\alpha = 0.10$), e.g., at at training set size 200.

  - Overall

    All selective sampling methods except BootstrapLV beat random sampling in terms of TP rates (the performance difference is significant [$\alpha = 0.10$] at training set size 200).

Some experimental results (Figures 2.18-2.27) confirm the hypothesis that selective sampling methods are superior to random sampling when dimensionality increases.

2.3.4 Conclusion and Discussion

Experimental results (especially when dimensionality is high) proved that selective sampling algorithms are more efficient than random sampling. Given the same accuracy target, less training data is needed by selective sampling algorithms. The performance of the five implemented selective sampling methods is different with different dimensionality. Therefore, one conclusion is that the different sampling methods behave differently in different problems. Experiments must be done to get the most suitable sampling method for a given problem. In the artificial test environments, bagging and rule-based sampling are always among the best methods for all spaces.

A comment on Bootstrap-LV

Experiments indicate that the behavior of Bootstrap-LV is much different than the other selective sampling techniques. In Figure 2.10(a), Bootstrap-LV wastes many of its samples in areas already known to be negative. This is because the Bootstrap-LV algorithm uses example scores as probabilities and samples from this distribution instead of picking the top N examples. The problem is the following: if data points with high scores are greatly outweighed by data points with low scores, Bootstrap-LV will tend to select low scoring data points. For example, suppose there are 4 examples in a 100-example pool having very high scores (x each), and the other 96 examples all have the score of $x/10$ each. A low scoring example is 2.4 times more likely to be chosen than a high scoring example.

Chapter 3

Scenario Testing for (Sports) Games

In this chapter, the Semi-Automated Gameplay Analysis (SAGA-ML) system is presented. SAGA-ML will automatically play against real games to collect information used for gameplay analysis and this process actually is the functional test in the context of software testing. Therefore, the background of software testing is introduced in 3.1. The relationship between *automated testing* and *gameplay analysis* is discussed in 3.2. Then the details of SAGA-ML are described and illustrated with the example of the *corner kick* scenario in Electronic Arts's FIFA99 soccer game.

3.1 Background — Software Testing

Software testing is an important part of the software development cycle. Figure 3.1 shows the waterfall model[31] that separates the software development process into several phases, each of which only interacts with adjacent phases. First, requirements are analyzed and defined. Then, the system and software are designed based on the requirements. Testing involves in two phases: during the implementation phase, unit testing will be used to test individual modules; more testing will be done for integration and at the system level. Finally, the software will be delivered to customers and maintenance work will be done as needed. Software testing is playing a more and more important role, and accounts for 50% of the total cost of software development[3]. A study[27] by National Institute of Standards & Technology showed that "the national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion" or about 0.6 percent of the US gross domestic product.

In 1991, the International Organization for Standardization (ISO) adopted ISO 9126[13] as the standard for software quality. It is structured around six main attributes listed below:

Functionality (suitability, accurateness, interoperability, compliance, security)

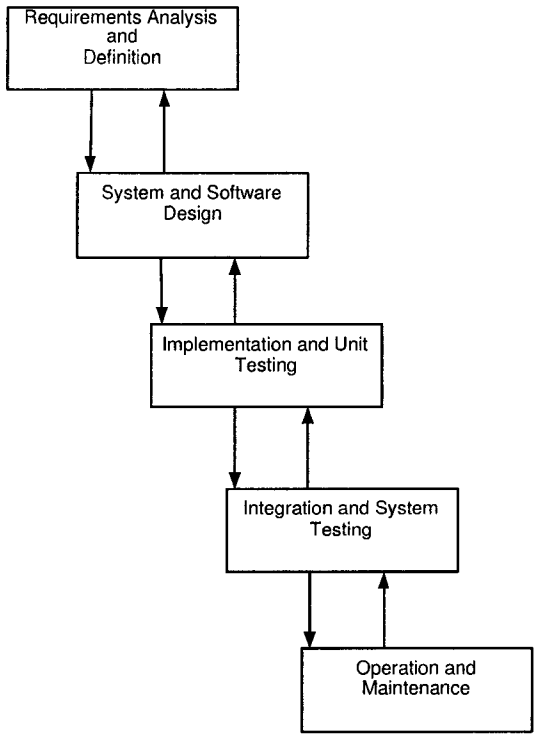


Figure 3.1: Waterfall Model of Software Development Process

Reliability (maturity, fault tolerance, recoverability)

Usability (understandability, learnability, operability)

Efficiency (time behavior, resource behavior)

Maintainability (analyzability, changeability, stability, testability)

Portability (adaptability, installability, conformance, replaceability)

Although above 6 categories are well defined software testing categories, there are no standard testing metrics and processes. Publications[26] by IEEE have presented numerous potential metrics that can be used to test each attribute.

Fault density

Defect density

Cumulative failure profile

Fault-days number

Functional or modular test coverage

Data or information flow complexity

Cause and effect graphing

Requirements traceability

Defect indices

Error distribution (s)

Software maturity index

Person-hours per major defect detected

Number of conflicting requirements

Number of entries and exits per module

Software science measures

Graph-theoretic complexity for architecture

Cyclomatic complexity

Minimal unit test case determination

Run reliability

Design structure

Mean time to discover the next K-faults

Software purity level

Estimated number of faults remaining (by seeding)

Requirements compliance

Test coverage

Reliability growth function

Residual fault count

Failure analysis elapsed time

Testing sufficiently

Mean time to failure

Failure rate

Software documentation and source listing

Rely-required software reliability

Software release readiness

Completeness

Test accuracy

System performance reliability

Independent process reliability

Combined hardware and software (system) availability

In practice, software testing usually uses the following categories.

Unit testing this type of testing will test a basic unit of the software (e.g. a method in object-oriented software).

Integration testing this type of testing is the module level testing. The integration of several objects or methods will be tested.

System testing this type of testing is the application level testing. The integration of modules will be tested.

Regression testing this type of testing will verify that new code does not introduce any additional bugs into the previously tested code.

Stress testing the purpose of stress testing is to make sure the software works properly beyond normal load conditions (e.g., testing the program in a system where all physical memory has been consumed).

Performance testing performance testing will test performance issues such as response time, efficiency etc.

Each testing category will consider at least one metric from IEEE's list. A general process for a software testing task can be divided into 4 phases[45]:

Modelling the software's environment

Selecting test scenarios

Running and evaluating test scenarios

Measuring testing progress

Firstly, the tester should model the software's environment to simulate realistic software usage. All interactions between users and software should be considered. Given the complexity of current software, the second testing phase is to select test cases (a test case is a choice of input testing data) to satisfy certain, criteria, such as execution path coverage and input domain coverage. Then, those test cases are run and evaluated according to the software's specifications. Finally, evaluation should be done to measure the whole testing progress to see if enough testing has been done.

3.1.1 Commercial Tools for Software Testing

The worldwide market for software testing tools was $931 million in 1999 and at that time was projected to grow to more than $2.6 billion by 2004[35]. Testing tools can be categorized into general purpose tools, application specific tools, and management tools. General purpose tools including tools for functional testing and performance testing, which can be applied to all software. On the other hand, many test tools are designed for specific software types such as databases, embedded systems, Java-based systems and web applications. Management tools include the tools for test management, bug tracking, requirement management, etc. There are a large number of test automation tools and venders in the market. Among them, Rational (Robot, Visual Test), Mercury Interactive (WinRunner), Segue (SilkTest) and Compuware (QA Run) are the more well-known test tool companies (products). Each product has its own strength and weakness. For example, Rational Robot is good at data creation facilities and Segue SilkTest has better text facilities. Reference [30] compares main functional test tools in the market.

3.1.2 Test-case Generation for Software Testing

Among the 4 phases of a software testing process, test-case selection and evaluation are the two phases that cost testers the most time. Although program-based (automatic) test data generation and verification has been believed to be effective at reducing software testing cost for many years, in real industry software development, those two tasks are still done manually and heavily rely on testers' expertise. In this section, some research on test-case automatic generation will be discussed.

Test cases are selected to simulate real software usage. Ideally, it will be best if all possible usage is tested, which is not practical due to the complexity of current software. The input domain of software usually is infinite. There are two criteria to evaluate a test case selection process: if selected test cases are enough to satisfy criteria (e.g., coverage) for execution paths and input domains, and the number of test cases. The best case is that minimal test cases are selected to achieve the testing goals.

There are two categories of automatic test-case generation techniques: structural (white-box) testing and functional (black-box) testing. If the structure of tested program itself is used for testing, the testing method is called white-box; otherwise, it is called black-box. Reference [41] indicates that modern software is too large to be tested by the white-box approach as a single entity. In practice, white-box testing approaches are usually used for subsystem levels. Black-box approaches are more commonly used for complex systems.

- **Structural (White-box) test-case generation techniques**

  Basic Concept Many studies in the literature use the concept of Control Flow Graph (CFG) to analyze structur-based testing methods. The following definitions are mostly taken from [23]:

  Control Flow Graph (CFG) CFG is the control flow graph of the program under test. Most structure-based testing methods rely on a CFG. A CFG is a directed graph $G = (N, E, s, e)$, where N is the set of nodes; E is the set of edges, s and e the entry/exit nodes of the program F. Each node $n \in N$ is a statement in F. Each edge $e = (n_i, n_j) \in E$ is a transfer of control from node n_i to n_j. Nodes corresponding to decision statements (e.g., *if...then*, *while* loop) are called *branching nodes*.

  Path a path P is a sequence $P =< n_1, n_2, ..., n_m >$, such that for all i, $1 \leq i < m, (n_i, n_{i+i}) \in E$. A path is *feasible* if there exists a program input for which the path is traversed, otherwise it is *infeasible*.

  Static structural test-case generation The methods in this category analyze the static program structure without executing the program. The general framework of such methods is this: given a goal node g, a path leading to g will be found by analyzing the static program structure solely. Inputs (test cases) are then generated from this path. If no inputs can be generated from this path, another path leading to node g will be chosen until inputs can be generated. *Symbolic execution*[5] is a representative approach in this category of test-case generation techniques. For any path p, all constraints involving the input variables used by p are generated. The task of selecting inputs is converted to solving those constraints. The main shortcoming of this method is that the symbols it uses do not include dynamic elements like dynamic data structures and arrays.

  Dynamic structural test-case generation The test-case generation techniques in this category execute the software being tested.

  Random approach The random approach is the simplest method: test cases are generated randomly from the input domain when the software is executed. However, this method is not efficient and effective. Many program paths can not be tested due to the very low probability of generating inputs that exercise these paths.

The goal-oriented approach This approach[16] starts the software with arbitrary inputs. When program runs into any code node, a procedure will be executed to decide if this node should be run or not. The criteria is the relationship between this node and the target node. If executing the current node does not lead towards the execution of target node, this node should be avoided.

The chaining approach Symbolic execution and the goal-oriented approach only use a program's control flow graph. In the chaining approach[12], data dependence is used as well to evaluate if a program node should be included into the path.

Assertion testing Assertion testing[15] inserts many assert statements into the code to test boolean conditions. If the boolean conditions are false, a failure will be detected. There are two tasks in these testing methods: the first one is to write the boolean conditions, which means the tester must fully understand the meaning of the program under test. This could be a limitation of this method. The second task of this testing method is to run the software and enumerate all paths containing assert statements.

- **Functional (Black-box) test-case generation techniques**

  Black-box testing methods are usually used to test the functionalities of a system without knowing the implementation details. In the context of test-case generation, the main idea is to use a function description (e.g., function specifications) to generate test cases.

  In [39], formal function specifications are used to generate test data. First, function specifications must be converted into a formal format: conditions on the inputs of the function are the pre-condition and the outputs of the function is the post-condition. A failure will be detected if inputs satisfy the pre-condition but the outputs violate the post-condition. Given an input of the function, an objective function is used to evaluate its *closeness* score to get a fault output. This score is then used to guide the generation of next input supposed to be *closer* to cause a failure.

  Many black-box test-case generation techniques are A.I. based, which will be discussed in more details in following section.

3.1.3 A.I. for Software testing

Artificial Intelligence techniques have been applied to all software testing phases. Many A.I. techniques, like Artificial Neural Networks (ANN) and Info Fuzzy Networks (IFN), have been successfully used for test-case generation and test-oracle creation. Some data mining techniques[10] have been used to discover knowledge from software metrics. Because statistics is one of the most important foundations of A.I., some statistics-based testing techniques are also included in this section.

Statistic Software Testing Techniques

Markov chain A Markov chain is used to model software usage in [46]. A Markov chain will be built based on software usage. Test cases then can be generated from this model and multiple probability distributions. A second Markov chain will be built from the generated test cases with the results of their having been run (failed or not). This second Markov chain can be used to analyze the failure distribution and evaluate the software. The quality of building the first Markov chain (how to simulate the realistic software usage before this software is released?) is the key of this technique.

Hill climbing *Hill climbing* has been used for test-case generation[24]. Starting from a seed test case, a hill climbing algorithm explores the neighborhood of the input domain of this seed. Depending on testing purpose, a objective function is used to evaluate test cases. This search process will continue until a locally optimal test case is found.

Simulated annealing *Simulated annealing*[39][40] is used to overcome the problems of hill climbing. By probabilistically accepting poorer solutions, *simulated annealing* can explore more of the input space than hill-climbing, which means higher quality test cases might be found.

Heuristic Search

Many test-case generation approaches discussed above use a heuristic method to search the target program node. The goal-oriented approach[16] and the chaining approach[12] will evaluate current program branch to decide if search should start from this branch. Reference [20] used heuristic search techniques to generate test program automatically based on test cases and closed algebraic specifications of the classes.

Evolutionary Algorithm

Some applications (e.g., [43], [18], [44]) applied evolutionary algorithms to software testing. Evolutionary algorithm based test-case generation methods start from a initial test case population, usually randomly generated. Depending on the testing purpose, a fitness function will be used to generate a real value for each test case. Test cases with high scores have higher probabilities to be selected to mate and mutate to generate the next generation — new test cases. In the end, high quality test cases are generated.

Artificial Neural Networks (ANN) and Info Fuzzy Networks (IFN)

Artificial Neural Networks (ANN) and Info Fuzzy Networks (IFN) are two important data mining approaches. The reason to discuss ANN and IFN together is that they are almost playing same roles in software testing. Both ANN and IFN are used on test-case generation and test-oracle generation. In [2], a neural network uses test case metrics as inputs and the error classification as outputs for training. Once such as a network is trained, test cases generated by random or from some generation tools will be fed into this network to predict the fault exposure. Only test cases classified as certain classes will be used as real test cases. In [19], an IFN is built as a simulated system of the system under test. Randomly generated test cases with corresponding system outputs are sent to the IFN. The IFN algorithm is run repeatedly to find a subset of input variables relevant to each output. Once the IFN is trained, a set of non-redundant test case covering the most common functional relationships existing in software will be automatically produced. In [1], the performance of ANN and IFN as test oracles is compared.

3.2 Gameplay Analysis and Automated Game Scenario Testing

As it is defined in Chapter 1, *gameplay* usually refers to the game behaviour as defined by the game's rules, logic, difficulty, goals, and constraints. Gameplay contributes much to the enjoyability of a game.

3.2.1 Gameplay Analysis

Figure 3.2 shows the current industry process for gameplay design, implementation, testing, and refinement. A box with a human icon means that that phase needs human involvement. Usually, the producers design the game's properties. Game developers implement those game properties according to the design specifications. Producers and testers will examine

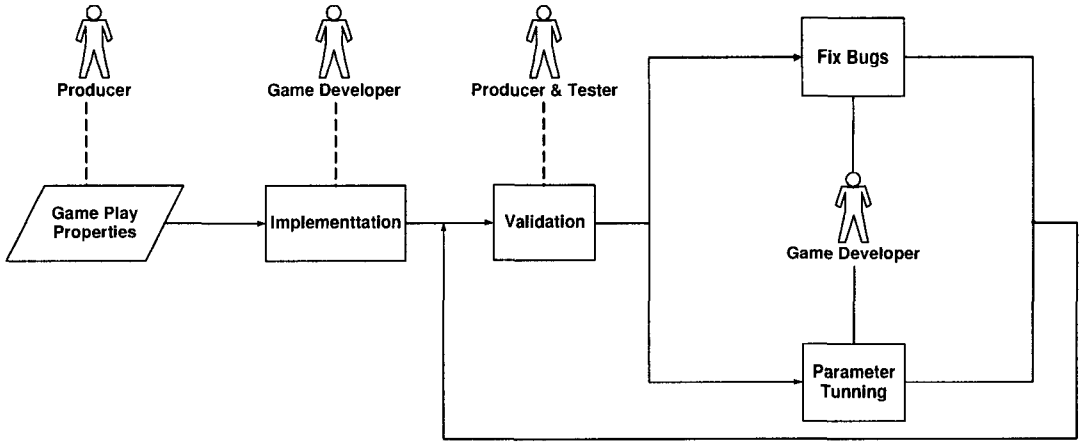


Figure 3.2: The gameplay development process in industry

the implementations and ask developers to fix related bugs and tune parameters. Game evaluation will be done during the *validation* phase. The common method of doing gameplay evaluation is a process of play-and-feel, which totally relies on a human's sensitivity and experience. The play-and-feel method has its benefits: it is simple, it just involves playing the game, and human feeling could be very sensitive if trained properly. That is why computers cannot replace artists. However, the play-and-feel method has some major drawbacks. First, it can be very slow. For instance, if a developer wants to test the game behaviour of the corner kick scenario in the FIFA99 game (defined below), he has to drive the game into a corner kick situation, which is a rare scenario during a game. Therefore, it might take at least 10 seconds to enter a corner kick. However, many repetitions of the same corner kick are needed for good testing, which is a slow process. Another drawback of play-and-feel is that this process can be human resource-consuming for producers, developers and testers.

3.2.2 Using Automated Testing for Gameplay Analysis

Given the fact of inefficient gameplay analysis in industry (the play-and-feel method), a new gameplay analysis method, which integrates automated testing into gameplay analysis, is proposed in this thesis. We will have a testing program that generates a wide variety of inputs to the game and measures a specific outcome, such as whether a goal is scored or not. This will generate a large file of gameplay examples, where each example records the sequence of actions taken by the testing program and the final outcome.

Challenges and Solutions in Using Automated Testing for Gameplay Analysis

Challenge 1: Normally, automated testing is used for testing tasks such as functionality testing, where the output of a test case will be known explicitly by referring to design specifications. It is also possible to apply the automated testing process when the design specifications do not provide explicit criteria for judging the result of testing: a test case will generate an output but it can't be evaluated automatically as being right or wrong, or good or bad. In this situation, there won't be any testing statistics directly coming from automated testing. Gameplay analysis is just such a case. Putting many game modules together, even game developers themselves are not sure of the exact behavior to expect. There is no formal definition of *correct* gameplay because gameplay is a kind of feeling. For example, in assessing the *difficulty* of a game, some people might say the *beginner level* is too easy, while someone else will say it is too hard. Therefore, a human is needed to make subjective judgements. But the huge gameplay log is incomprehensible to humans. If automated testing is to be used for gameplay analysis, the first challenge is – how to generate human-friendly testing output?

Solution to Challenge 1: In this thesis, *machine learning* is used to create a summary of the gameplay log that is comprehensible by humans. The learning task is defined as a *classification* task because we are trying to predict a class label (e.g., whether or not a goal was scored) given a game situation. The features in each training example are the game variables related to the outcome being investigated. After the game is played by using a vector of feature values as input, the desired class will be collected from the game. The feature value inputs and their corresponding class make up one training example. Training examples are fed into a machine learning algorithm (C4.5 here) to produce a model (a set of rules here) summarizing game behavior, which could be further visualized and presented to game designers.

Challenge 2: The set of possible inputs for the full game is far too large. How to efficiently guide automated testing is the second challenge.

Solutions to Challenge 2: Automated testing programs in this thesis are not designed for the whole game, but for an individual game scenario. Even restricted to a single scenario, sample spaces can be too large to exhaustively enumerate because they might be high dimensional, and dimensions can be continuous. Time and computation restrictions do not allow sampling a huge number of examples. Therefore, selective sampling (discussed in Chapter 2) is used to sample instance space intelligently to build a good model.

Game Analyzer: A Similar Gameplay Analysis Tool for Role-Playing Games

Game Analyzer[25] is a semi-automated tool to analyze gameplay for Role-Playing games. Game Analyzer transforms a user-defined game scenario into a state-transition model, and then samples and evaluates policies (like the action sequences in our system). Although Game Analyzer shares the same goal as this thesis, there are three main differences between the two systems. Firstly, Game Analyzer applies to Role-Playing Games, not sports games. Secondly, Game Analyzer does not use machine learning to summarize game behaviour. It just samples the space of possible "policies" and displays the particular outcomes of the policies it tried. Lastly, Game Analyzer does not use active learning.

3.3 Semi-automated Gameplay Analysis System (SAGA-ML)

Instead of traditional, totally manual gameplay analysis, the preceding section has outlined a method for gameplay analysis that involves automated testing, with human involvement only once machine learning has created a summary of the gameplay behaviour. The specific definition of gameplay used in the proposed method is the game behavior under the constraints defining a game scenario. Figure 3.3 is the architecture of the whole SAGA-ML system, which plays the role of the Validation box in Figure 3.2. Because of the human involvement, this system is not totally automated, therefore it is called the Semi-automated Gameplay Analysis System (SAGA-ML). Modules "Sample Generator" (selective sampling) and "C4.5" have been covered in the last chapter. The present chapter will give the details for the rest of the modules in Figure 3.3. To better explain SAGA-ML, a concrete application, the corner kick scenario in Electronic Arts's FIFA99, is used in this chapter.

3.3.1 The Corner Kick Scenario in Electronic Arts's FIFA99

A corner kick in a soccer game is a direct free kick from a corner of the field awarded to the attacking team when the ball has been driven out of bounds over the goal line by a defender (from the American Heritage Dictionary). FIFA99 corner kick scenario is well known because many human players take advantage of this scenario to score. There are some corner kicks in which the attackers almost always score. Therefore, this scenario actually is of great value to test for EA. This scenario is the simplest one in this thesis because it only has two features: the X and Y coordinates of the ball's landing position. This is because in FIFA99 this scenario has a fixed setup, which means given the same game configurations (teams, players etc.), the initial state (e.g. the position of each player)

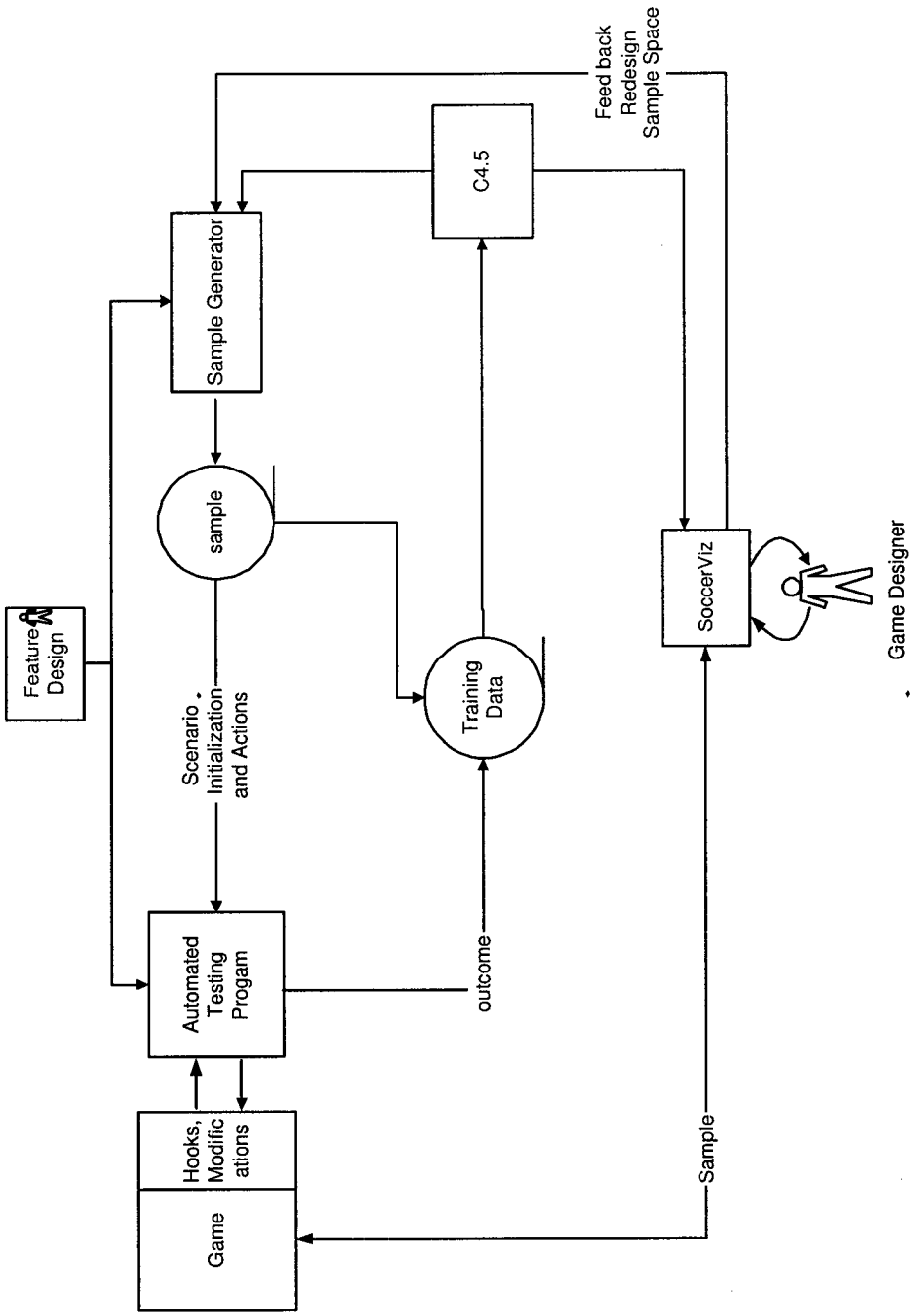
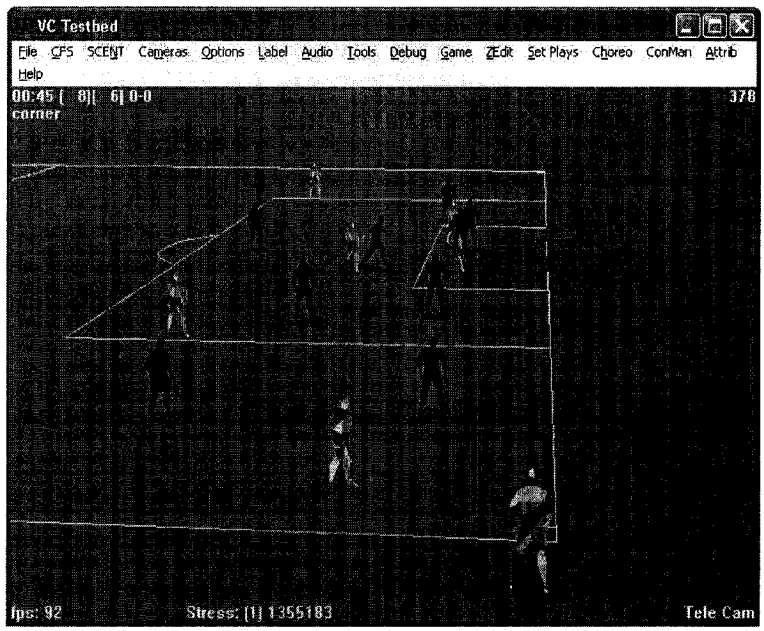


Figure 3.3: SAGA-ML architecture

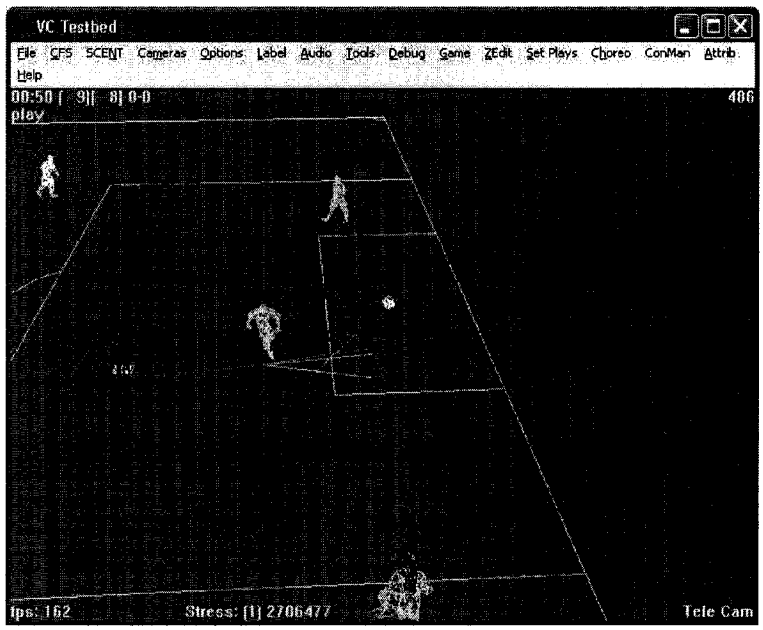
of this scenario is always the same, there are no player-controlled aspects of the initial state. In Figure 3.4(a), the soccer ball is placed at the near corner (it is obscured by the nearest player); the nearest player is the kick taker from the attacking team; every other player (on both the attacking and the defending team) in the figure is in his predetermined corner kick position. After the ball is kicked from the corner, the attacker who receives the ball will make a shot (with his head or feet) at his first touch. The end of this scenario is one of following 3 events: (1) the ball goes out of bounds or into the goal, (2) the defenders get control of the ball, or (3) a time limit is exceeded. In Figure 3.4(b), the ball is in the air and each player in the picture is responding to the kick. The outcome of this scenario is that a goal is scored (*score*) or not. As discussed in 2.3.1, each action sequence will be run 10 times and assigned to class *score* if the number of times it produces a goal exceeds a threshold. Therefore, *score* actually means "score with high probability".

3.3.2 Feature Design

Given a game scenario, a sample is a feature vector describing the scenario's initial conditions and the actions (if any) that will be taken by the user as the scenario plays out. The task of *feature design* is to define features representing the game scenario that are relevant to the testing goal being considered. In the corner kick scenario, the testing goal is to summarize the ways of scoring (with high probability) in terms of different ball landing positions. Therefore, the features are the ball's landing position (X and Y coordinates). The outcome of the scenario is added as the the sample's class label, which is a boolean value indicating if a goal is scored or not. The sample feature vectors are produced by the *Sample Generator*. Some features could exactly correspond to variables inside the game. For example, in the corner kick scenario, the inputs generated by the *Sample Generator* are the ball's landing position pair (X and Y co-ordinates). This pair of features directly corresponds to two game variables in FIFA99. However, in some more complicated cases, the variables inside the game might be different than the variables we want for machine learning. Two sets of features might be equivalent mathematically but very different in terms of how easily a concept can be expressed by machine learning algorithms. For instance, suppose that there are only two players (a shooter and an opponent goalie) standing on a soccer field. To measure their positions, two Cartesian co-ordinate pairs (Figure 3.5(a)) can be used. From the pure mathematics perspective, the Cartesian co-ordinates are equivalent to another set of measures: the distance and the angle between the two players, and the angles between the shooter and the two goal posts (Figure 3.5(b)). But from a machine learning algorithms'

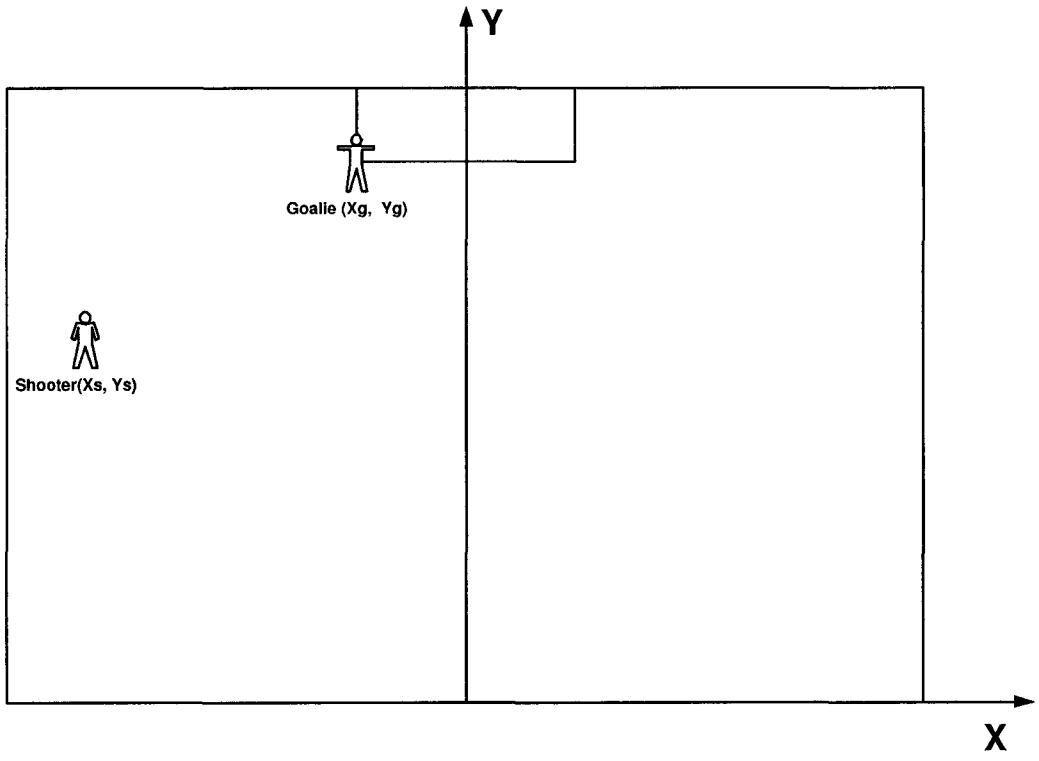


(a) prepare

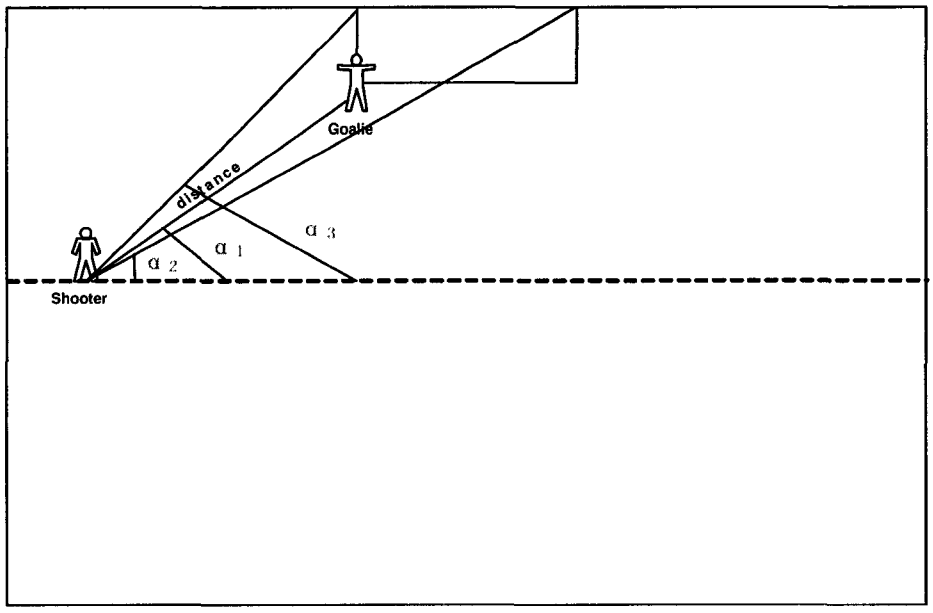


(b) shoot

Figure 3.4: Corner kick scenario



(a) Cartesian co-ordinates measure



(b) distance and 3 angles measure

Figure 3.5: Different but mathematically equivalent feature measures

point of view, the distance and angles might be preferred because they are more informative.

3.3.3 Sample Generator

The new training examples (without their labels) are generated by the *Sample Generator*, which implements the selective sampling algorithms covered by Chapter 2.

3.3.4 Game Scenario Automation

To get the output values needed by the machine learning algorithm, an automated testing program is run to execute the game scenario. The testing program will take the inputs generated by the *Sample Generator* to initialize itself, with data conversion, if necessary, to convert the features to values for actual game variables. Then a series of actions is executed to simulate playing the game scenario. Finally, the result of this test is collected from the game and converted into the form needed by the machine learning algorithm. In this thesis, a state machine is built for each game scenario to guide the testing process. The automated testing program will collect the required game information and transit between the states. For example, there are four states defined for the corner kick scenario:

CORNERKICK_PREPARING

CORNERKICK_BEGIN

CORNERKICK_FIRSTTOUCH

CORNERKICK_SECONDTOUCH

The corner kick scenario begins from the state CORNERKICK_PREPARING. In this state, a corner kick event is triggered; the ball is placed in a corner, every player in the field goes to his position, and the kick-taker stands in front of the ball. After the game state is stable, the automated testing program will go to the next state: CORNERKICK_BEGIN. In this state, the kick-taker kicks the ball so that it will land at the position specified by the *Sample Generator*. As soon as the ball is touched by a player after being kicked, the scenario goes to the third state: CORNERKICK_FIRSTTOUCH. If the first toucher is an attacker, he will make a shot at the goal. If the ball is out of bounds, is touched a second time, or a predetermined time limit is exceeded, the scenario goes to the last state: CORNERKICK_SECONDTOUCH. In this state, the result of this run (score or not) is collected and the corresponding output for the machine learning algorithm is written to a data file.

The above state transitions rely on some internal game variables, such as the game state, the ball's position and velocity, etc. The testing program tracks the values of these variables with hooks that are put into the game for this purpose. The testing process could be run thousands of times, but the game running environment has to be reset after each run.

3.3.5 Game Hooks and Modifications to the Original Game

To run a game scenario automation, the game must be modified. There are two types of interactions between the testing program and the original game. The first type are Game Hooks, which are the entry points from the original game to the automated testing program. The second type are Modifications to the original game specifically for testing a given game scenario. The following are the hooks and modifications for the corner kick scenario.

void GAME_updateFrame()

This is a standard FIFA99 function that is called once every game frame. Our code is added here to update the game scenario testing state machine. The scenario testing state machine then inspects specific variables to determine if a state transition has been made. If the final state is reached, the final output will be printed to a file for this run and the state machine reset to begin the next run. This function is the main entrance of the state machines for each game scenario.

void REFEREE_process_ballout()

Originally, this function call will be followed by a sequence of game video (e.g., the goalie picks up the ball or players cheer for a goal). To prevent those game videos in the corner kick scenario, this function is changed to do nothing when the ball is out of bounds.

void FIELD_get_corner_position(COORD *pos, COORD *newpos)

Originally, this function will calculate which corner should be used to do the corner kick based on the position where the ball went out of bounds. To get identical initial states, this function is changed to return a fixed corner position when the corner kick scenario is being tested.

void PLAYERTASK_free_kick(PLAYER_DEF *this)

Originally, the game has its own strategy to choose the ball's landing position. This function is changed to use the landing position generated by the Sample Generator.

void TACTIC_doBestKick()

Originally, this function will select a *best* action (e.g., dribbling, passing, or shooting) for the attacker. It is changed to force the ball's receiver (on the attacking side) to make a shot in the corner kick scenario.

3.3.6 SoccerViz: Visualization Tool

A single IF...THEN... rule is easy to read, but a large set of rules is not that easy for humans to read. Moreover, if the rule set is ordered and the rules can overlap (as is the case with the rules produced by C4.5), it will be very hard to understand. Therefore, for some game scenarios, Mark Trommelen at the University of Alberta implemented a visualization tool called SoccerViz to display the rule sets in a way more natural to the designer. Three scenarios have been visualized: a 2-d artificial test scenario, the corner kick scenario, and the breakaway scenario.

Figure 3.6 is a screen shot of SoccerViz for the corner kick scenario. There are 4 main areas in the screen shot: the visualization area (central middle), the rule selection area (right), the rule details area (bottom), and the control buttons area (top).

Visualization Area

The visualization area is the graphic part of SoccerViz. Visualized components (soccer field, rules, data points etc.) for a game scenario will be drawn here. In Figure 3.6, the background of the visualized corner kick scenario is a soccer field and it is zoomed in to one side of the penalty area. There are many dots, each of which represents an example, mostly inside the penalty area. The color of a dot indicates the class of the corresponding example: black is negative (not a score) and white is positive (score). Because each action sequence will be executed multiple times (10 in this study) to get a probabilistic outcome, there are overlapping data points which can have different labels (colors in the visualization tool). If they are drawn directly in the tool, they will be on top of one another and so people won't be able to see what proportion of them are black and what proportion are white. To clearly visualize these data points, a grid color is introduced, as follows. The soccer field is divided into small rectangles, and the color (shade of grey) of a rectangle is determined by the ratio of the number of black dots inside this grid *v.s.* the number of white dots. If there are too many data points inside a single grid, a cross (X) is used to represent all data points inside the rectangle. The color of the cross (X) indicates the overall grey color of the data points inside this grid.

Rule Selection area

The rule set for the current selective sampling iteration is listed in this area. Rules are grouped by their rule class and sorted by their accuracy. The default rule is placed last. The user can select or unselect a rule. If a rule is selected, a visualized rule area is drawn in Visualization area. In Figure 3.6, rule1 is selected, and the rectangle representing rule1 is drawn roughly in the middle of penalty area.

Rule Details Area

The rule details area gives the detailed description of the selected rules. In Figure 3.6, the details of rule1 are shown.

Command Buttons Area

The top area of Figure 3.6 is the command buttons area. These buttons give users controls such as: the user can choose to see individual rules, or a rule set; the user can choose to see the data points covered by the selected rules, or all the data points; the user can feed a data point back to the game to watch it being played out.

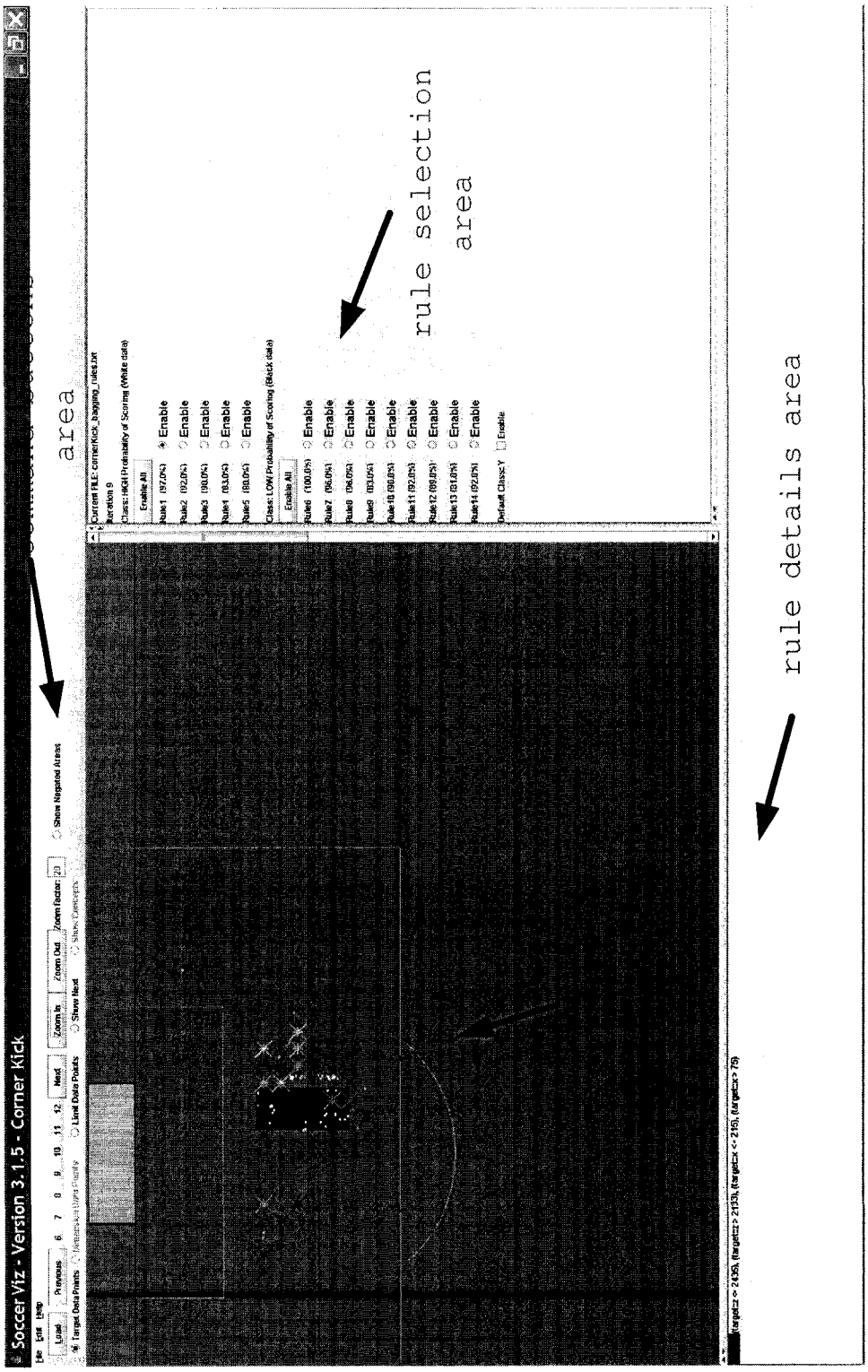


Figure 3.6: SoccerViz

Chapter 4

Game scenario testing in FIFA99

Electronic Arts's FIFA Soccer is a complex sports game. There are 22 players on the field. Many rules are implemented in the gameplay system of FIFA to coordinate all the players to simulate real soccer games. The complexity of FIFA's gameplay system causes difficulty in developing and testing. Many gameplay sweet spots are utilized by human players to decrease the difficulty level of the game, which will impact the playability of the FIFA game eventually. In Chapter 3, the Semi-Automated Gameplay Analysis (SAGA-ML) system was introduced. In this chapter, SAGA-ML will be tested on FIFA99. It will summarize FIFA99 behaviors as human readable rules, which can be presented to game designers to check if those behaviors are as intended. Unexpected game behaviors can be found this way.

Unlike the evaluation methods used in the artificial testing environment, FIFA99 game scenarios will also be evaluated using "blurred" target concepts; this will be explained in Section 4.1. Then, the experimental results for three FIFA99 scenarios – corner kick, breakaway, and dribble-dribble-shoot – are presented.

4.1 Evaluation methods

In Chapter 2, an artificial experiment environment was set up. The target concepts are exactly known in advance, so experimental results can be evaluated by directly comparing the geometry of the learned concepts with the target concepts. However, there are no target concepts known in advance for a real application like FIFA99. In this chapter, all experiments are evaluated by two methods: the first one is the standard machine learning evaluation method, which estimates TP rates and FP rates by using a large test set. For an application like FIFA99, we believe another evaluation method is more informative: get the *approximate* target concepts first; blur the approximate target concept; and then evaluate learned concepts by comparing their geometry with the blurred target concepts. The blurring-based

evaluation method will generate *larger* regions in its concept definition by eliminating small regions and merging neighbouring regions. The motivation for preferring large regions is driven from the real demands of gameplay evaluation. Only large regions are important to game developers and human players. For example, in the cornerkick scenario of FIFA99, suppose there is a rule saying that if the attacker gets the ball and makes a shot right on the penalty kick spot, he will score. It is true that that rule can be considered as a sweet spot of FIFA99. However, the chance of meeting the requirement of that rule is very unlikely in the real play of FIFA99. Such small rules will be removed by the blurring-based evaluation method. Therefore, this method might be more meaningful for real gameplay analysis. Bezier smoothing is done to all the TP and FP plots in this chapter, as was done in Chapter 2.

4.1.1 The Generation of Target Concepts

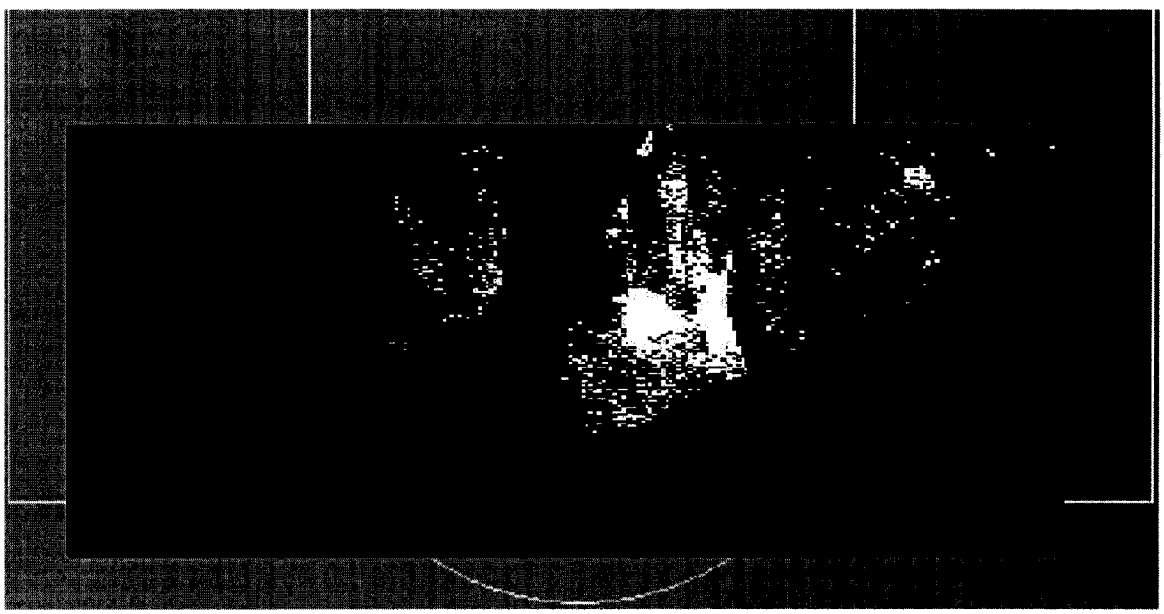
To generate target concepts as accurate as possible, a large set of training examples are generated by uniform sampling (for the corner kick scenario and breakaway scenario) or random sampling (for the dribble-dribble-shoot scenario because the sample space is too big to do uniform sampling). Each example is sent to the real game 10 times and gets labelled according to the classification threshold (as described in Section 2.3.1). Figure 4.1 is the visualized data point distribution for FIFA99 corner kick scenario. In Figure 4.1(a), the classification threshold = 4 and in Figure 4.1(b), the classification threshold = 7. White datapoints are examples labelled as *score with high probability* and black ones are examples labelled as *does not score with high probability*.

The algorithm for creating a target concept from data points is given in Figure 4.2. This figure also defines the thresholds used in the algorithm. Instance space is divided into grid cells (Step 1). The size of each cell is determined by a user-controlled threshold $T_{cellsize}$, which essentially depends on the sample density of the game scenario. Then all data points are put into those cells and the label of each cell is the majority class of the cell (Steps 2 and 3). The connected cell chains are the approximation of the target concept regions.

Target Concept Blurring

Blurring is used to refine the target concepts generated in Step 3 of Figure 4.2. Simply setting $T_{minimalsize}$ for a minimal target concept size, some potential useful target concepts might be filtered out. For example, if a block of connected cells has a *hole*, which might be due to noise, the whole block might be eliminated because the number of connected cells

(a) classification threshold = 4



(b) classification threshold = 7

Figure 4.1: Corner kick real classification distribution

1 Divide the sample space into grid cells (the granularity of the cell is a threshold relying on the application, $T_{cellsize}$)

2 Put each example into its corresponding cell.

3 Label each cell with the majority class of the examples in the cell.

4 Scan all labelled cells and group them according to their connectivity

5 Sort the connected cell groups (from step 4) by the number of cells each contains.

6 For each group g whose size is less than a threshold ($T_{BlurCandidate}$), check its neighbors. If it is mostly (another threshold $T_{BlurSurround}$) surrounded by groups whose size is bigger than $T_{BlurCandidate}$ with a label different than g, then the label of g is converted into the opposite class (we assume there are just two classes).

7 For those groups which have the same label, if they are close enough (threshold T_{close}), connect them into one group.

8 Remove those groups whose size is less than a threshold ($T_{minimalsize}$).

9 Save the cells and group information into a file as target concepts.

Figure 4.2: The Algorithm For Generating Blurred Target Concepts

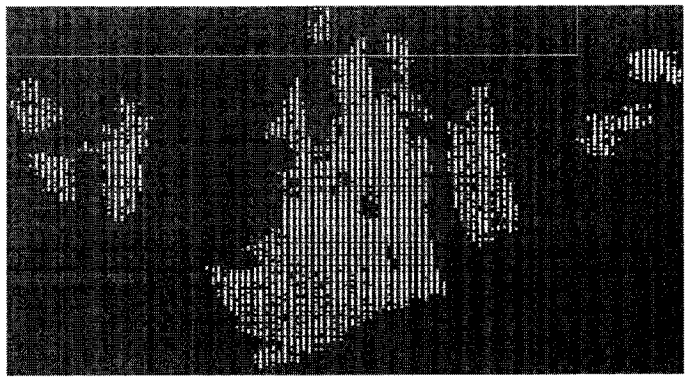
is just below the threshold. Another example is if there are two close but not connected blocks of cells, each of which is just below the threshold. Both of them will be filtered out, but actually they might be joined together to become a big region. A technique called *blurring* is implemented in this thesis. The idea of blurring is to fill those small *holes* and connect *nearby blocks*. The connected cell groups whose number of cells is less than $T_{BlurCandidate}$ are the candidates for the *blurring* process (the value of $T_{BlurCandidate}$ is 30 in all the experiments in this chapter). For each blurring candidate, its neighbor groups are checked (Step 6 in Figure 4.2). If this candidate is surrounded mostly (determined by threshold $T_{BlurSurround}$, which is 0.75 in our experiments) by cell groups whose size is bigger than $T_{BlurCandidate}$ and which have a label different than the candidate, then the label of all cells in the candidate will be reversed. The next step of blurring looks at small groups with the same label; if they are close enough (determined by threshold T_{close}, which is 2 cells in our studies), those groups will be merged into one group. Once this is done, all concepts will be filtered one more time using another threshold, $T_{minimalsize}$, the minimal size of a region, which is measured as the number of cells inside the region ($T_{minimalsize}$ = 30 in our implementation). Regions with too few cells in them will be removed by this step.

Figure 4.3(a) and (b) compares the same target concept without and with blurring.

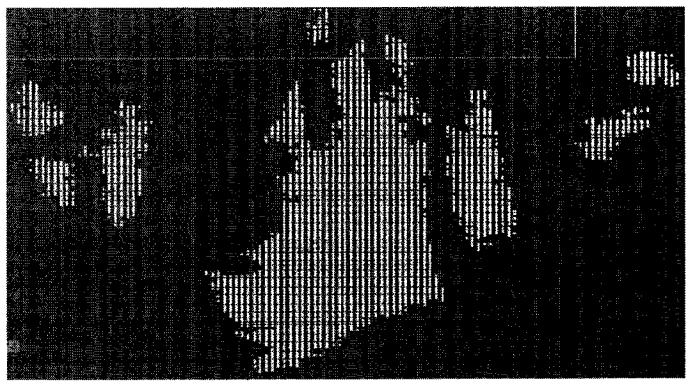
4.2 Corner Kick Scenario Experiments

Experimental results for the corner kick scenario are presented in this section. The initial training data set for all experiments contains 100 uniformly sampled examples. On each iteration 30 more examples are selected and added to the training set. The classification threshold = 4 (also used in the rest of the game scenarios).

Figure 4.4 and Figure 4.5 show the data points after 100 iterations of the selective sampling methods. Figure 4.4(a) is the same as Figure 4.1(a) – it is redrawn here for clear comparison. *Uncertainty sampling* (Figure 4.4(b)) concentrates its sampling on the central area; *bagging* (Figure 4.4(c)) has a tight sampling behavior surrounding positive areas; *boosting* (Figure 4.5(a)) also explored all positive areas but it explored slightly more areas than necessary; *rule-based sampling* (Figure 4.5(b)) put sampling points mostly around positive areas. Note: BootstrapLV is excluded from Figure 4.4 and Figure 4.5 because its behavior is much like random sampling.



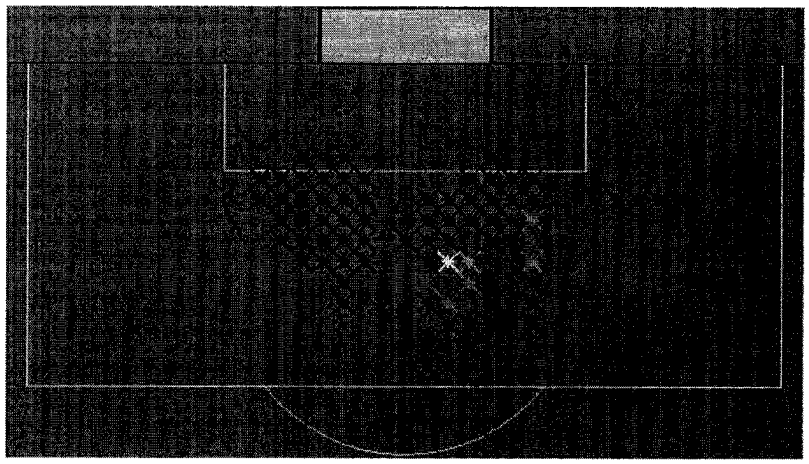
(a) classification threshold = 4, not blurred, $T_{minimalsize} = 30$



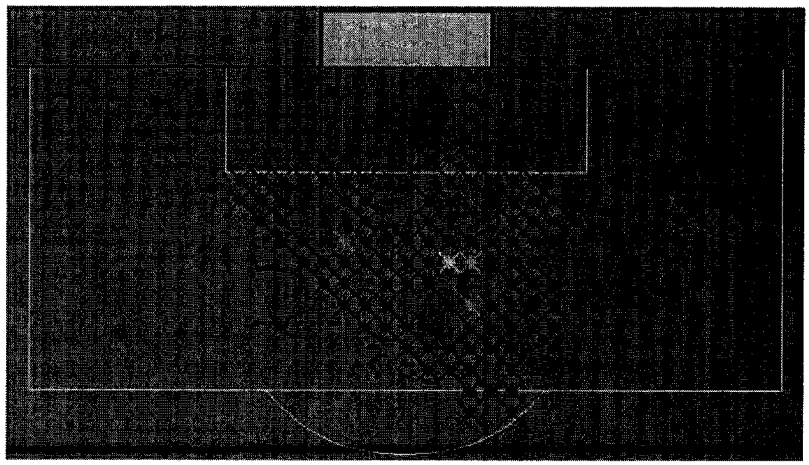
(b) classification threshold = 4, blurred, $T_{minimalsize} = 30$

Figure 4.3: Corner kick: generated target concepts

(a) real data distribution (classification threshold = 4)

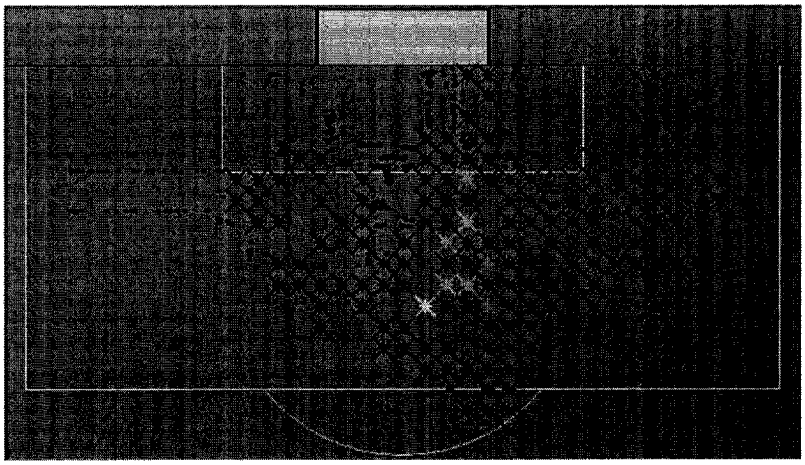


(b) uncertainty

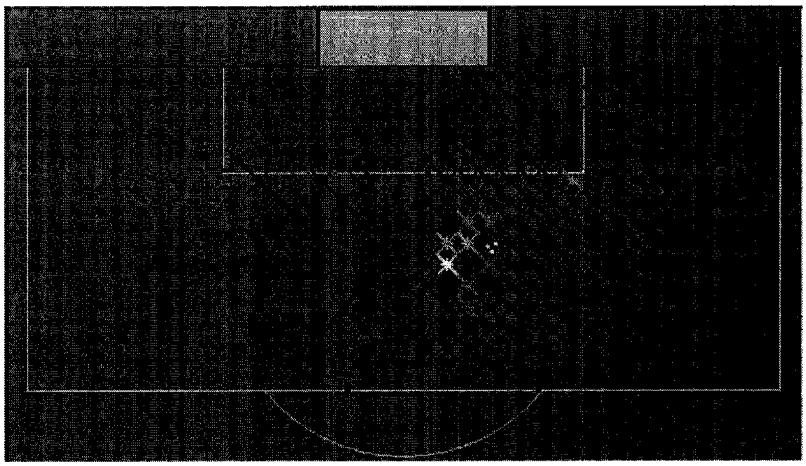


(c) bagging

Figure 4.4: Corner kick scenario, data points distribution comparison:1

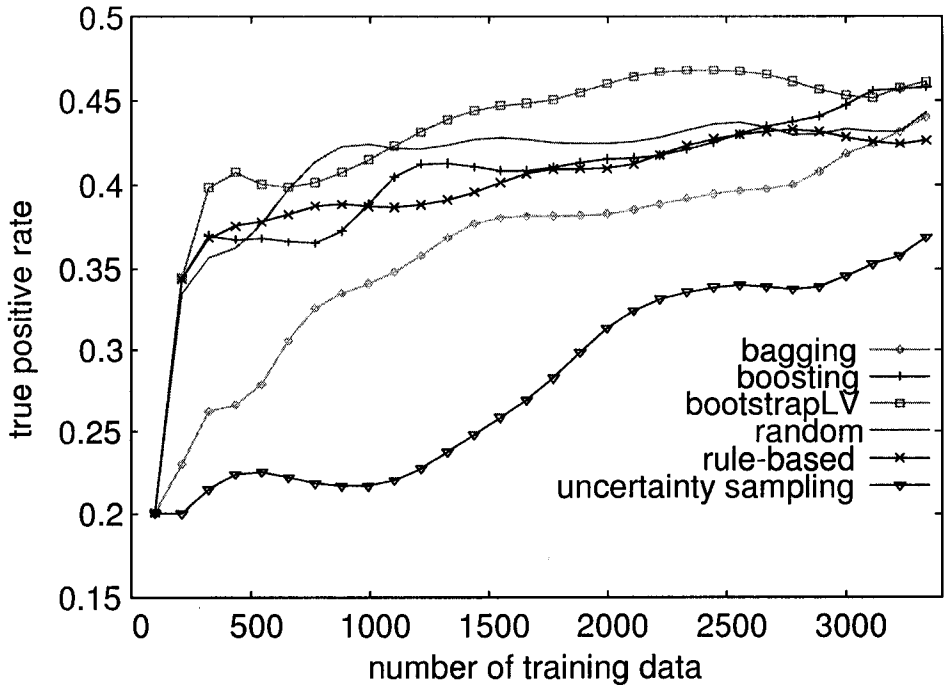


(a) boosting

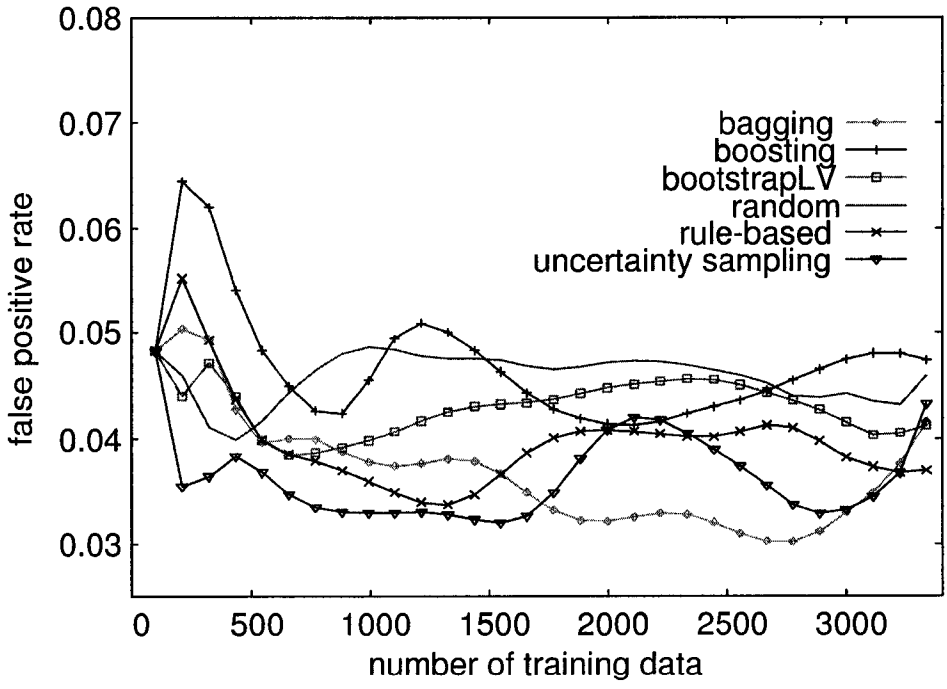


(b) rule-based

Figure 4.5: Corner kick scenario, data points distribution comparison:2



(a) TP rate



(b) FP rate

Figure 4.6: Corner kick: standard evaluation

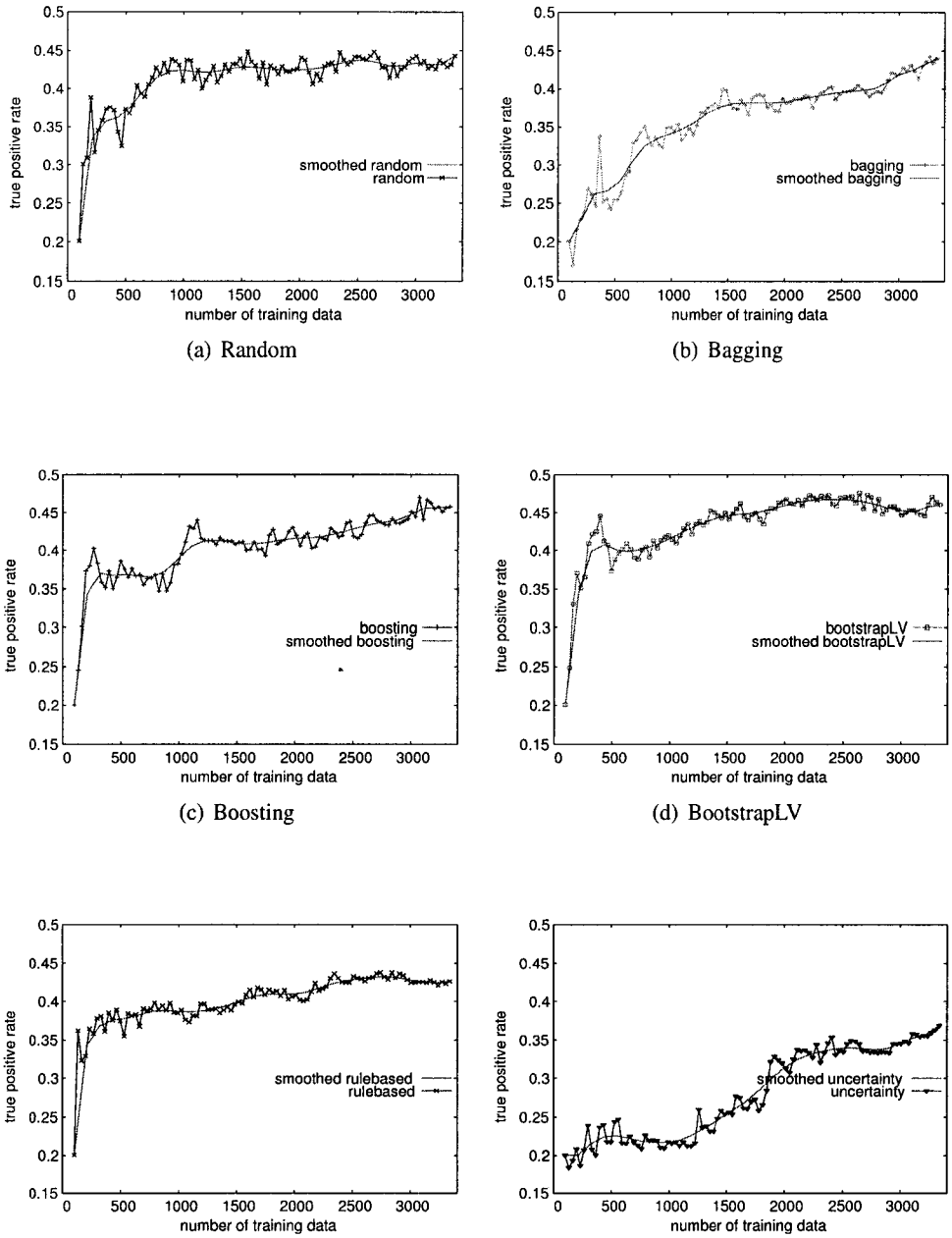
Standard evaluation

Figure 4.6 shows the results using the standard evaluation method. Before about 500 examples, the FP rates are not stable. After 500 examples, random sampling have consistently higher FP rates than bagging, uncertainty and the rule-based sampling methods (the performance difference is significant [$\alpha = 0.1$] at training set size 1500). As for TP rates, each TP rate increases very quickly before about 500 examples, and increases slowly afterwards. Uncertainty sampling has the worst TP rate than the others (the performance difference is significant [$\alpha = 0.1$] at training set size 2000). Boosting, rule-based sampling and random sampling do not show significant ($\alpha = 0.1$) TP rates difference at training set size 2000 given our sample size. BootstrapLV has the best TP rate after 1000 examples (the performance difference is significant [$\alpha = 0.1$] at training set size 2000).

Evaluation on blurred target concepts

The cell size threshold $T_{cellsize}$ used in this scenario is 0.005, which means each edge of the cell is 0.005 of the whole range of a dimension. Figure 4.9 shows the evaluation on blurred target concepts (the threshold of the minimal size of connected cells = 30). Compared with the standard evaluation (Figure 4.6), the FP rates in this figure are decreased by around 40% and the TP rates are increased by around 40%; in addition, the curves are more stable. Thus, the evaluation using the blurred target concepts in the corner kick scenario is much better than the standard evaluation method. More importantly, evaluation with blurred target concepts focuses on the regions big enough to be of interest to game developers, which means the results of this evaluation method will make more sense; the same conclusion will apply other FIFA99 game scenarios in this thesis.

Evaluation on blurred target concepts gives the following results for the corner kick scenario. Random sampling has the worst FP rates than the other methods (the performance difference is significant [$\alpha = 0.1$] at training set size 1500). The rule-based method has lowest FP rates between 1000 examples to 2000 examples (the performance difference is significant [$\alpha = 0.1$] at training set size 1500). Uncertainty sampling still has the worst TP rate (the performance difference is significant [$\alpha = 0.1$] at training set size 1500). After 1000 examples, BootstrapLV and random sampling have better TP rates than other sampling methods (the performance difference is significant [$\alpha = 0.1$] at training set size 1500).

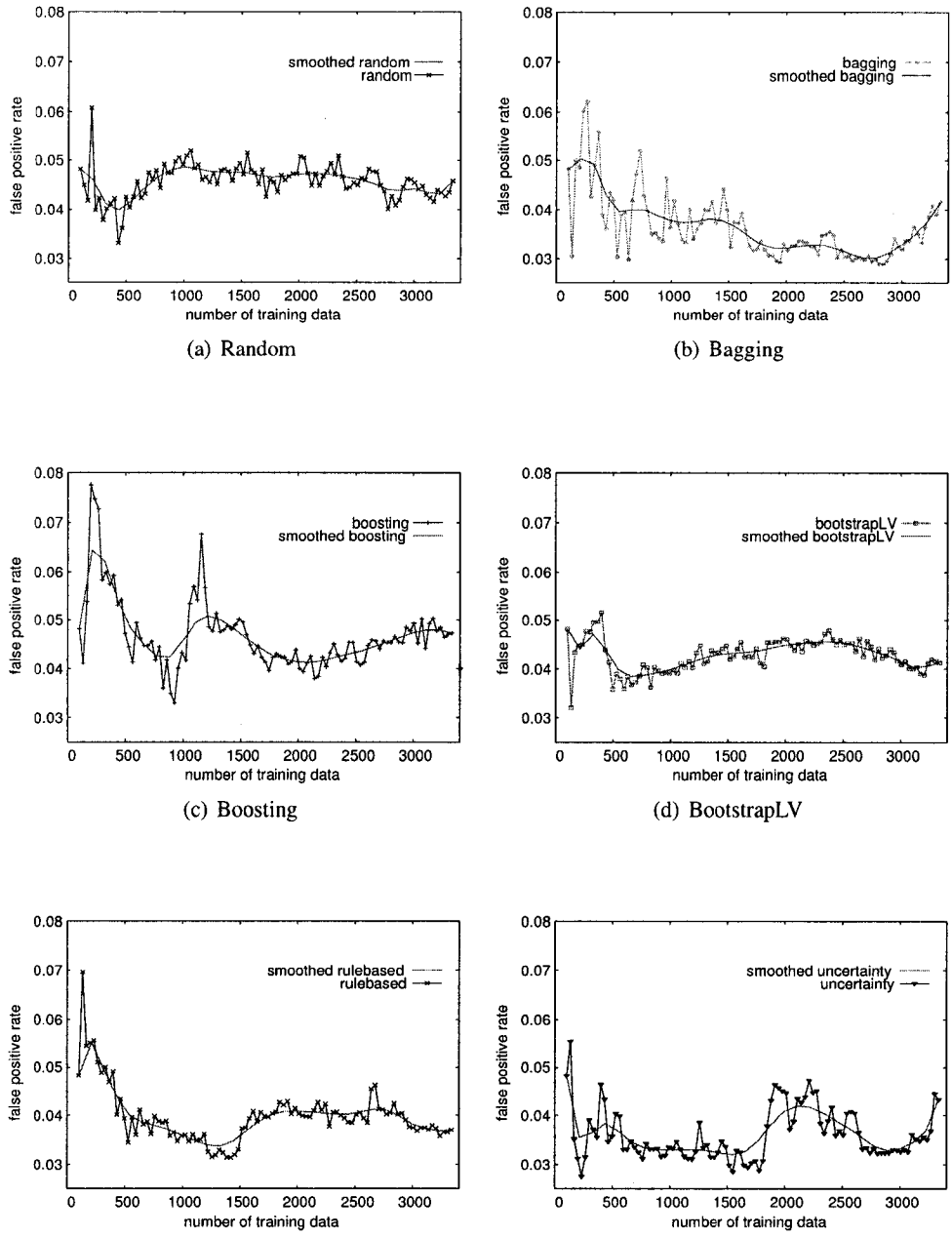


(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.7: Corner kick, standard evaluation, TP rate: smoothed *vs.* unsmoothed

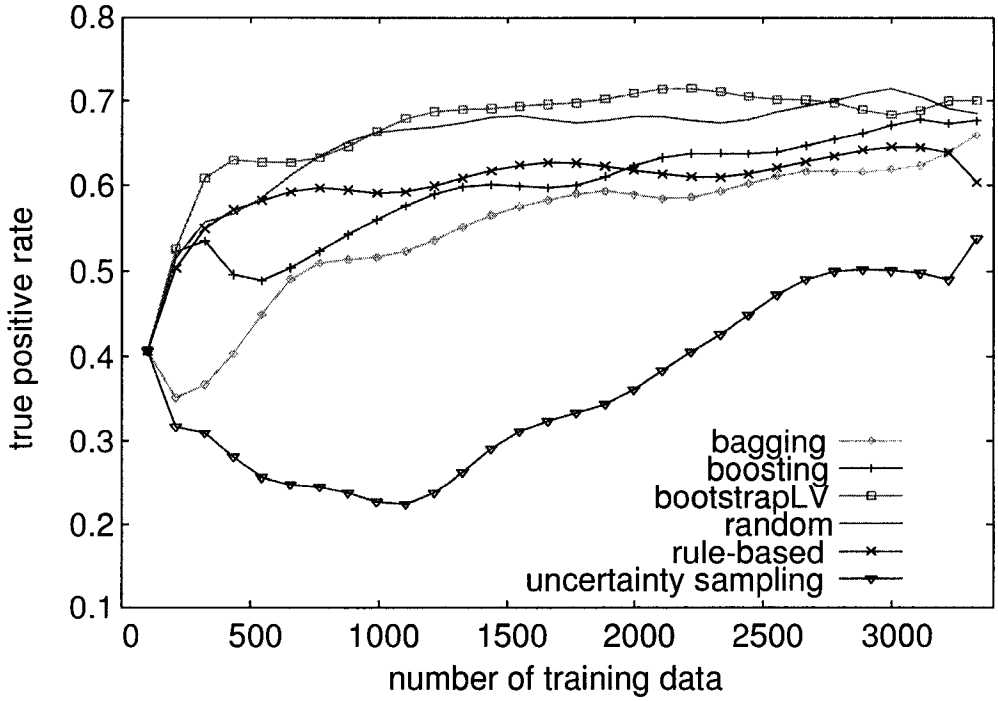


(a) Random (b) Bagging

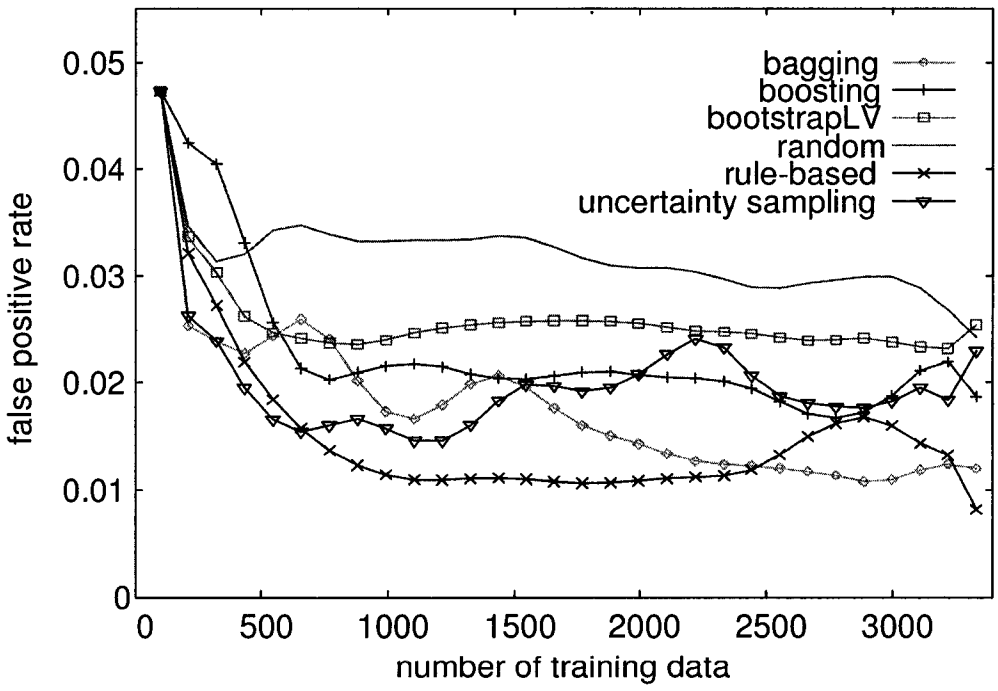
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.8: Corner kick, standard evaluation, FP rate: smoothed *vs.* unsmoothed

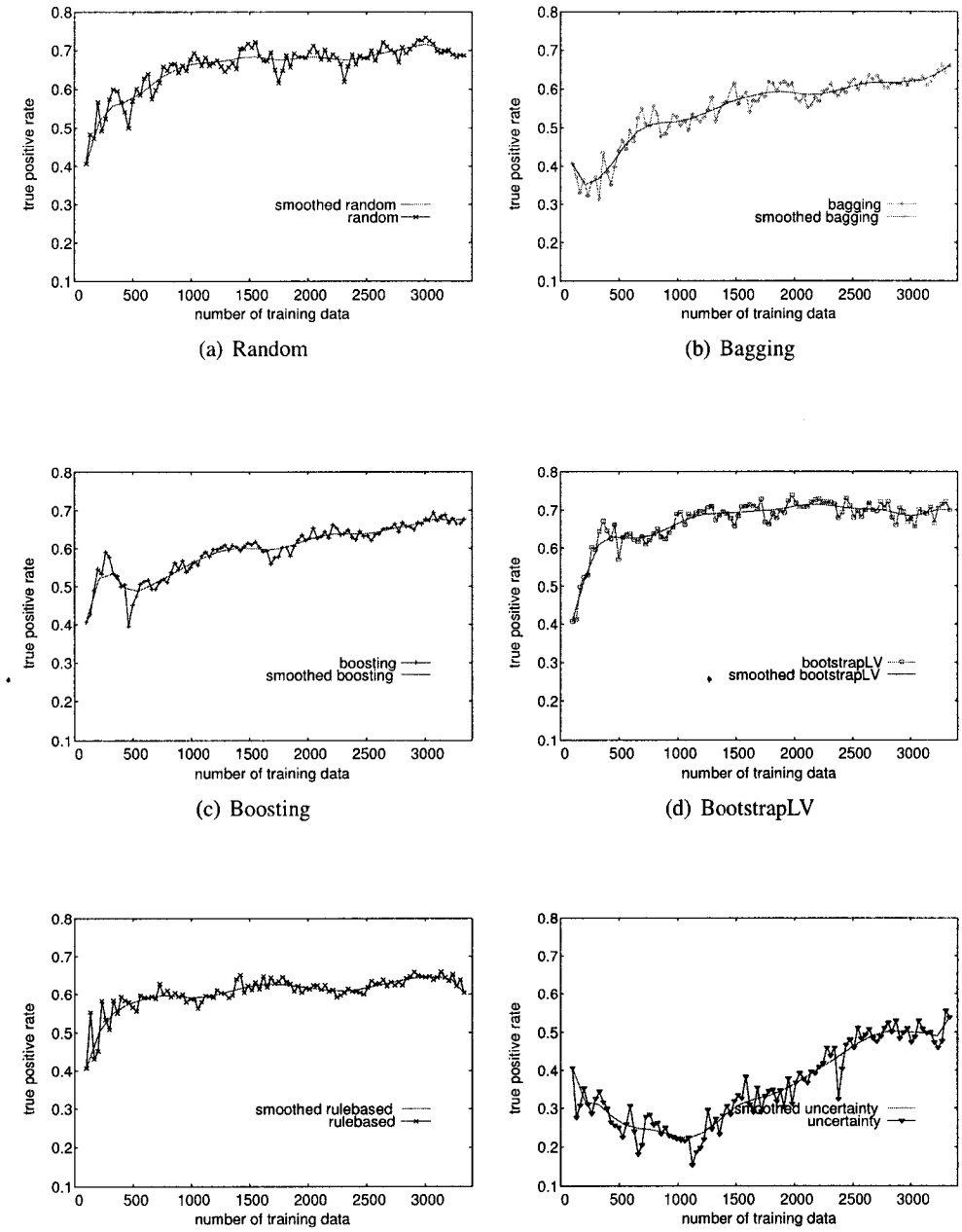


(a) TP rate



(b) FP rate

Figure 4.9: Corner kick: evaluation on blurred target concepts



(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.10: Corner kick, evaluation on blurred target concepts, TP rate: smoothed *vs.* unsmoothed

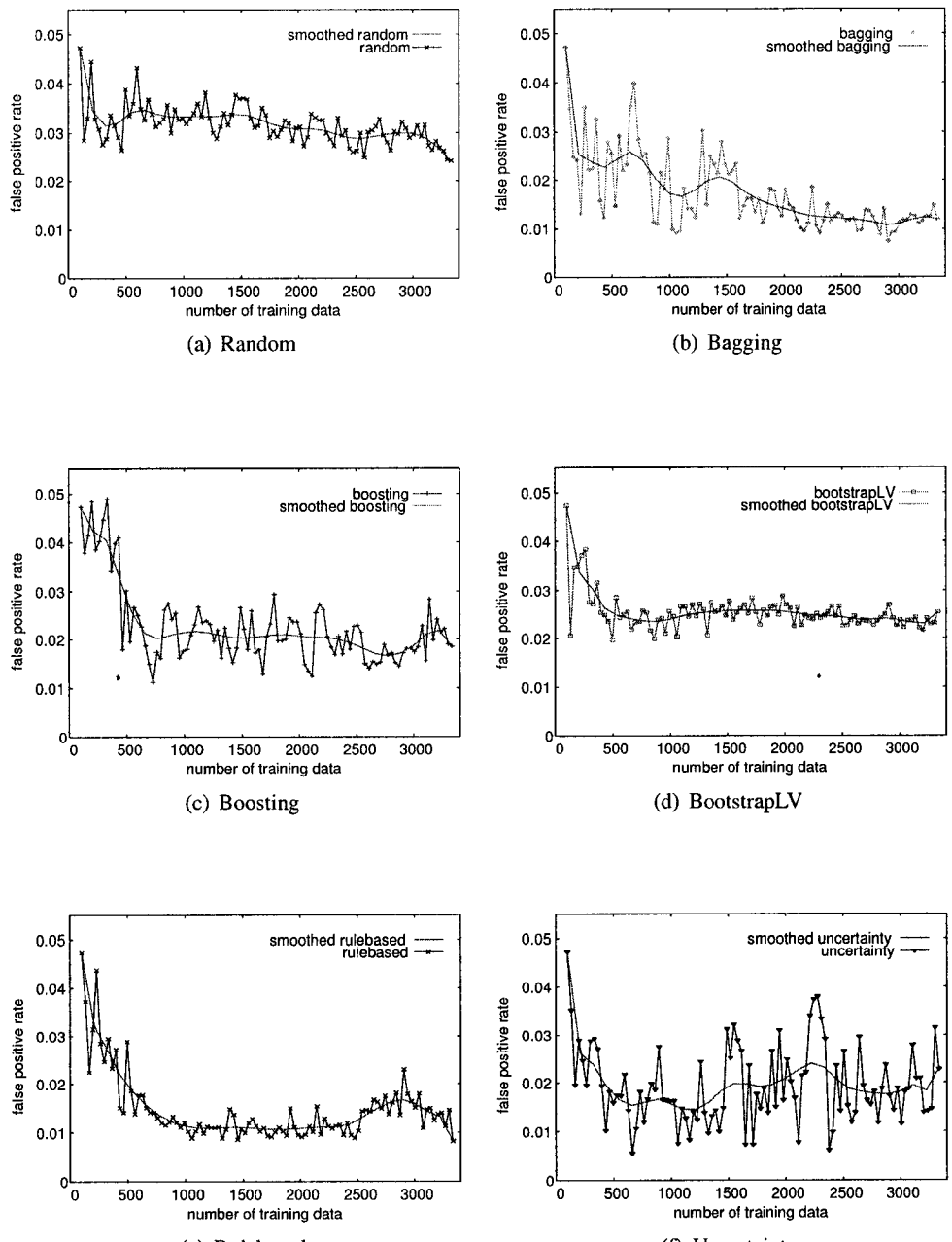


Figure 4.11: Corner kick, evaluation on blurred target concepts, FP rate: smoothed *vs.* unsmoothed

Figure 4.12: Corner kick summarized rules

Scenario Analysis

Figure 4.12 (based on the result after 100 iterations of bagging) shows the positive (score with high probability) rules learned for the corner kick scenario (the kicker is placed in the top right corner of the field). The areas covered by rules summarize the real data point distribution (Figure 4.4 (a)) very well. There is a very large black region in the middle of Figure 4.12.

By loading the data points of that area back into FIFA99 cornerkick scenario, a defender in that black area is observed to react poorly to the incoming ball, running away from the ball instead of intercepting it. This might be a *sweet spot* of this game scenario.

4.3 Breakaway scenario

4.3.1 Scenario description

The *breakaway* scenario in FIFA99 is defined as: one attacker is on a breakaway, there is only the opponent goalie between him and the goal, and the attacker shoots immediately from his location when the scenario is initialized. The classes of this scenario are the same as for the corner kick scenario: *score with high probability, or not.* The feature set used for this scenario is: the distance between the shooter and the goalie; the angle between the shooter and the goalie (Figure 4.13:α_1); the angle between the shooter and the right goalpost (Figure 4.13:α_2); and, the angle between the shooter and the left goalpost (Figure 4.13:α_3). Figure 4.14 illustrates this scenario.

4.3.2 State Machine and Game Hooks for the Breakaway Scenario

The breakaway state machine has 3 states:

PREPARE

SHOOT

END

Before the state machine starts, the feature vector from the sample generator is read and the feature values are converted to internal game variables.The automatic testing program then enters the first state: PREPARE. In this state, the shooter and the goalie will run to their initial locations, and the other players will be sent off the field. After everybody is in their designated positions, the SHOOT state is triggered: the shooter makes a best kick (the

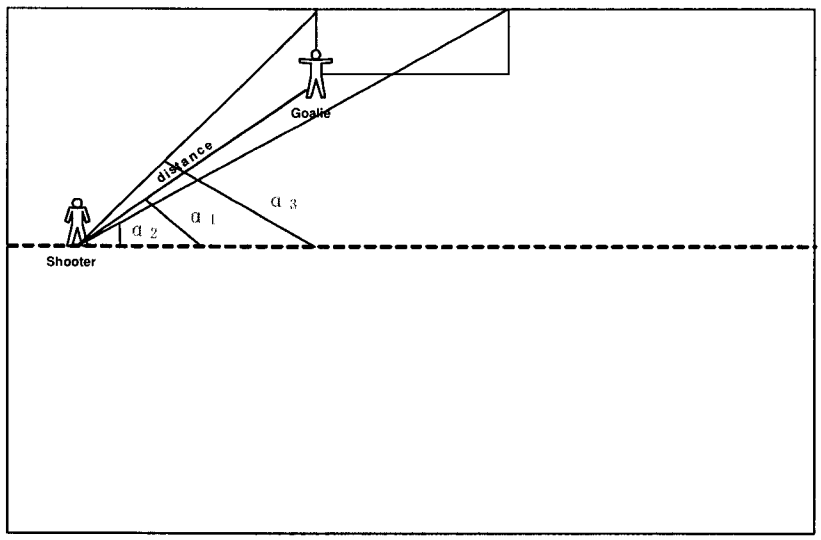


Figure 4.13: Features used in the breakaway scenario

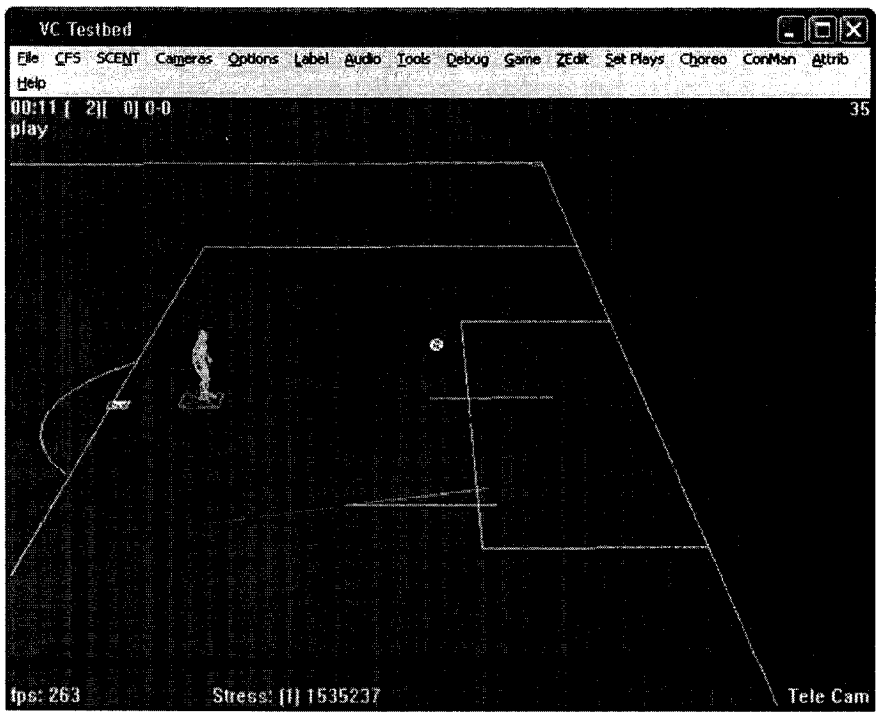


Figure 4.14: Breakaway scenario screen shot

game will decide what kick is best based on the current situation, such as the shooter and goalie's positions). Several events will trigger the END state: the ball is out of bounds; the goalie has the ball; the ball is moving in the opposite direction or is still; time out.

The running of this state machine relies on some game hooks and modifications in the FIFA99 game, as follows.

- **void GAME_updateFrame()**

  This function has been described in 3.3.5. For the breakaway scenario, the breakaway state machine manager will be triggered here every game frame.

- **void REFEREE_process_ballout(void)**

  As described in 3.3.5, this function is modified here as well to prevent unnecessary game videos.

4.3.3 Experimental set up and results

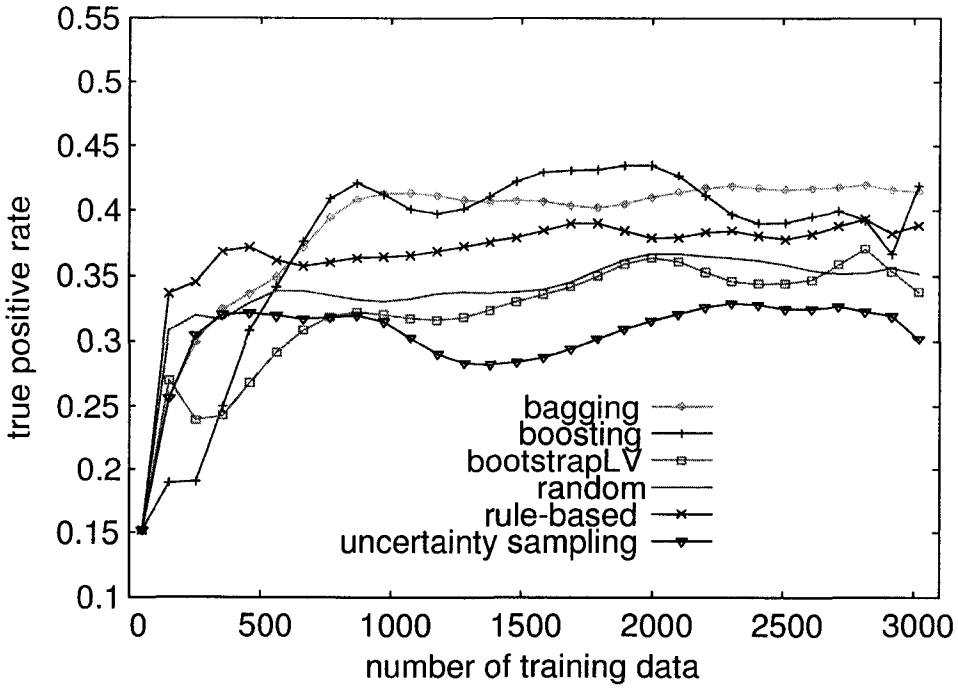
There are 100 randomly sampled initial training examples. 30 more new examples will be added into the training set at each iteration.

Standard evaluation

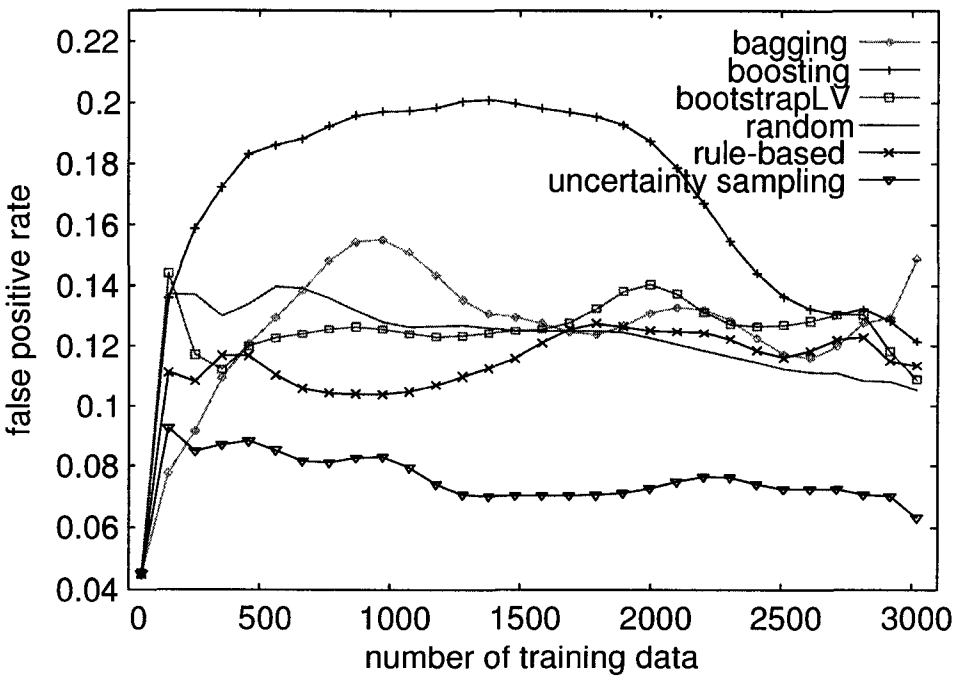
Figure 4.15 shows the results of the standard evaluation method. Before around 500 examples, TP and FP rates are not stable. In terms of FP rates, boosting is worse than the others (the performance difference is significant [$\alpha = 0.1$] at training set size 1500); uncertainty sampling has the best FP rates (the performance difference is significant [$\alpha = 0.1$] at training set size 1500); the others do not show significant ($\alpha = 0.1$) FP rate difference at training set size 2000 given our sample size. In terms of TP rates, before around 500 examples, the rule-based sampling method increases the fastest of all sampling methods (the performance difference is significant [$\alpha = 0.1$] at training set size 400). After 500 examples, bagging, boosting and rule-based sampling methods show better TP rates than random sampling (the performance difference is significant [$\alpha = 0.1$] at training set size 1500).Overall, the rule-based method is best, with a low FP rate and a high TP rate (especially before 500 examples).

Evaluation on blurred target concepts

The cell size threshold $T_{cellsize}$ used in this scenario is 0.01, which means each edge of the cell is 0.01 of the whole range of a dimension. Figure 4.18 shows the results using blurred

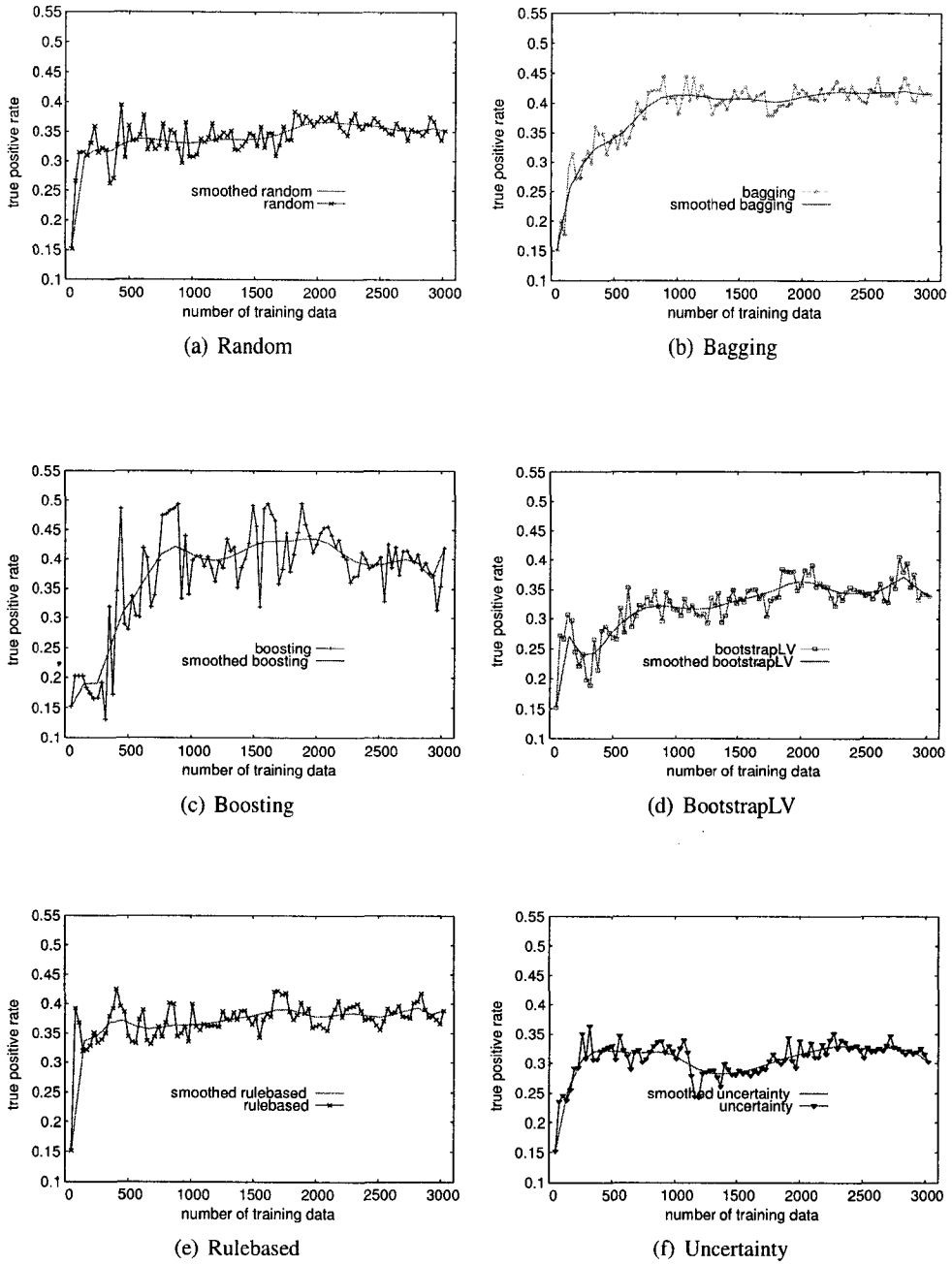


(a) TP rates



(b) FP rates

Figure 4.15: Breakaway: standard evaluation

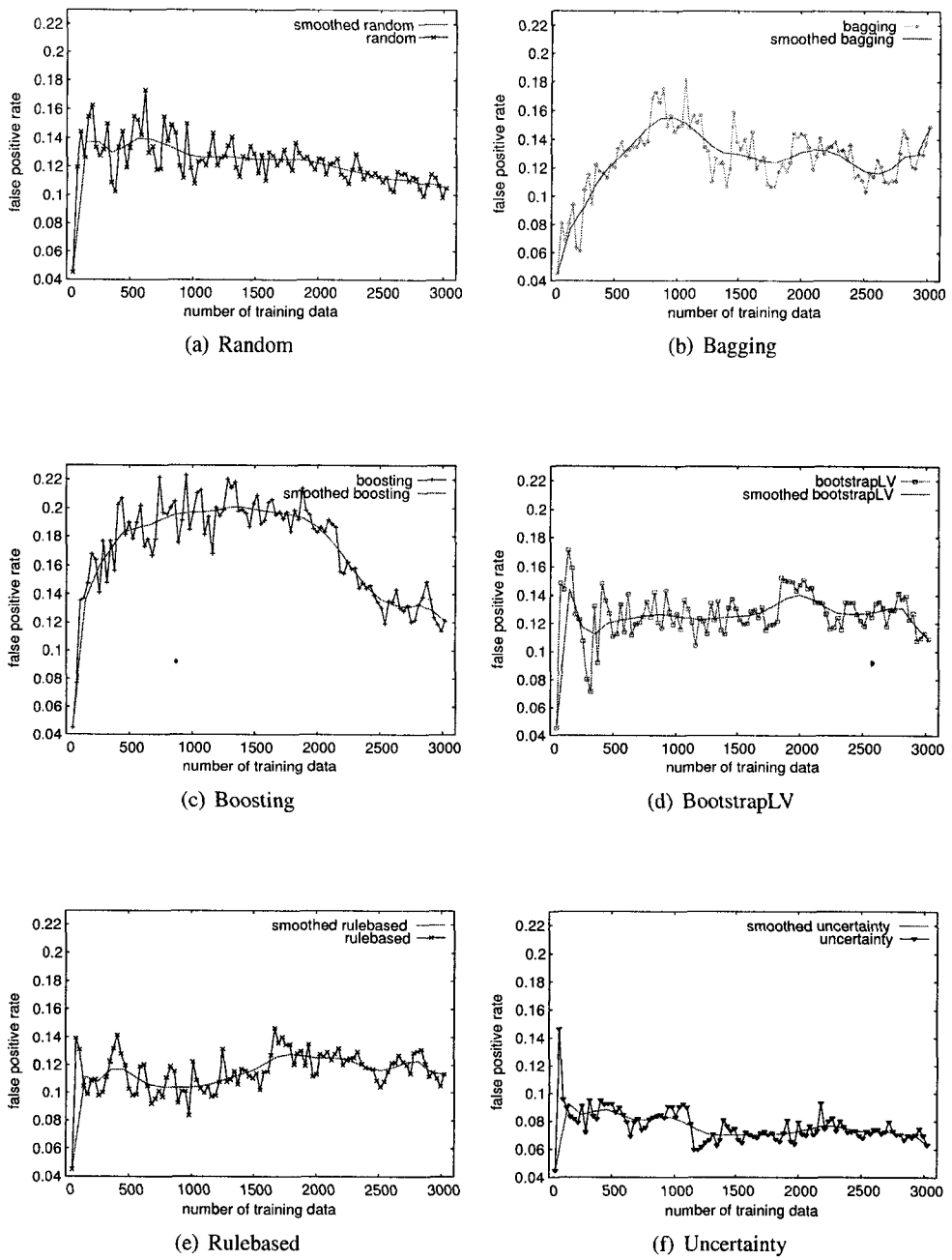


(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.16: Breakaway, standard evaluation, TP rate: smoothed *vs.* unsmoothed

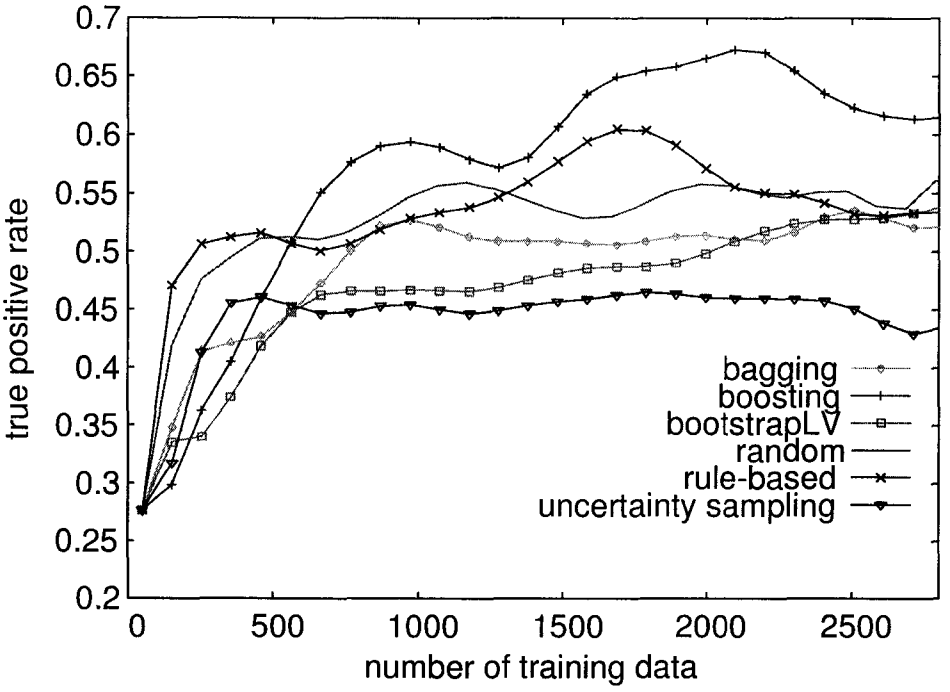


(a) Random (b) Bagging

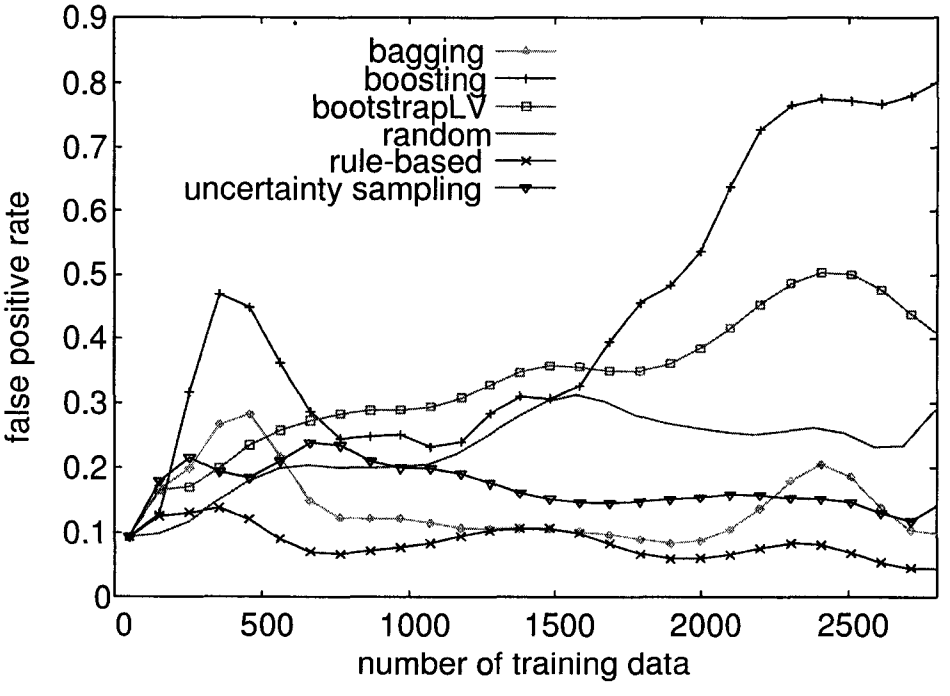
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.17: Breakaway, standard evaluation, FP rate: smoothed *vs.* unsmoothed



(a) TP rates



(b) FP rates

Figure 4.18: Breakaway: evaluation on blurred target concepts

target concepts, where the classification threshold = 4 , number of cells >= 30. In terms of FP rates, the rule-based method has the lowest FP rate all the time (the performance difference is significant [$\alpha = 0.1$] at training set size 2500); boosting is not stable. In terms of TP rates, before the rapidly increasing phase at around 500 examples, the rule-based method has better TP rates than the others (the performance difference is significant [alpha = 0.1] at training set size 250) except for random sampling).

Evaluation summary

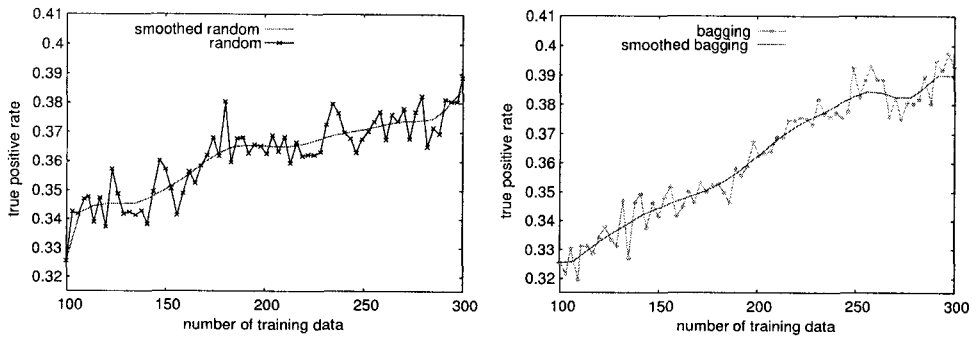
Evaluation results show that some selective sampling methods are better than random sampling (e.g., rule-based), but some are not (e.g., uncertainty). Before the stable phase (around 500 examples), the rule-based method always has the best overall performance. The game environment includes many dynamic and random factors, which might create noise for the active learning. Therefore, the result of breakaway scenario is not as good as its counterpart (4-d) in the artificial tests (Figure 2.18).

4.3.4 Scenario Analysis

There are some interesting results found by SAGA-ML for the breakaway scenario. Figure 4.21 shows one of the rules learned. The white triangle in the top left corner is the shooter area defined by the rule. By left-clicking with SoccerViz at a point in this area, a specific shooter position is selected and the corresponding goalie area is then drawn on the fly. Usually, scoring from sharp angles such as the one shown in Figure 4.21 is not easy. But if the goalie's position is constrained as shown, scoring from this position is not unreasonable. Game designers will examine a visualized rule such as this to decide if they are correct.

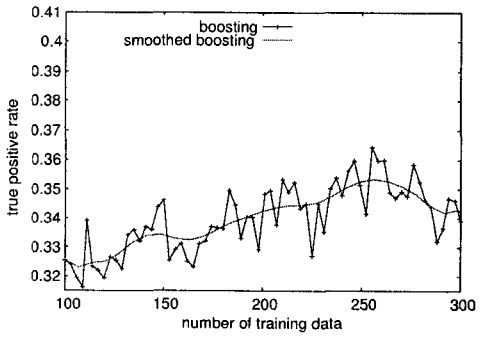
4.3.5 Higher Dimensional Experiments

There is an alternative, higher dimensional setup for the breakaway scenario. Six more dimensions are added (making 10 in total): the velocities of the ball (2-D, assuming the ball is on the ground), the shooter (2-D) and the goalie (2-D). Figure 4.22 shows a visualized rule by SoccerViz for this 10-D breakaway scenario. Without considering the ball's velocity constraints, the rule in Figure 4.22 can be explained as: the probability of scoring will be very high if the shooter makes a chip shot when the shooter and goalie are in the positions indicated. In the same rule, the ball's y-velocity (vertical direction) actually is constrained to be below the about 4/5 of its maximal limit. This constraint is very reasonable for a real soccer game: if the ball is moving too quickly away from the shooter, it will be hard to

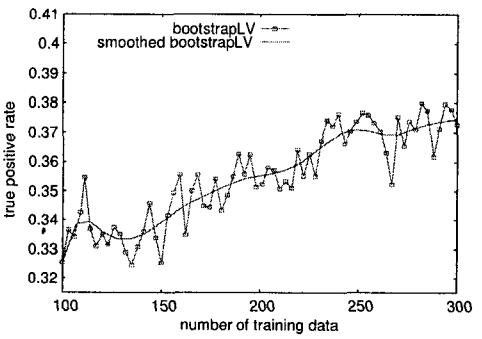


(a) Random

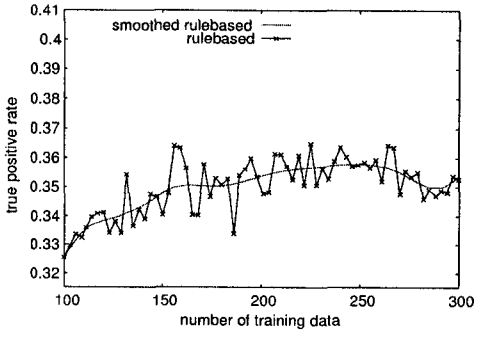
(b) Bagging



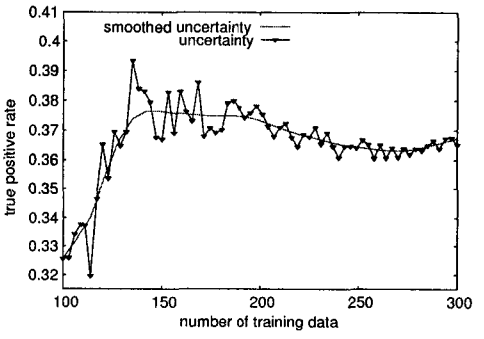
(c) Boosting



(d) BootstrapLV

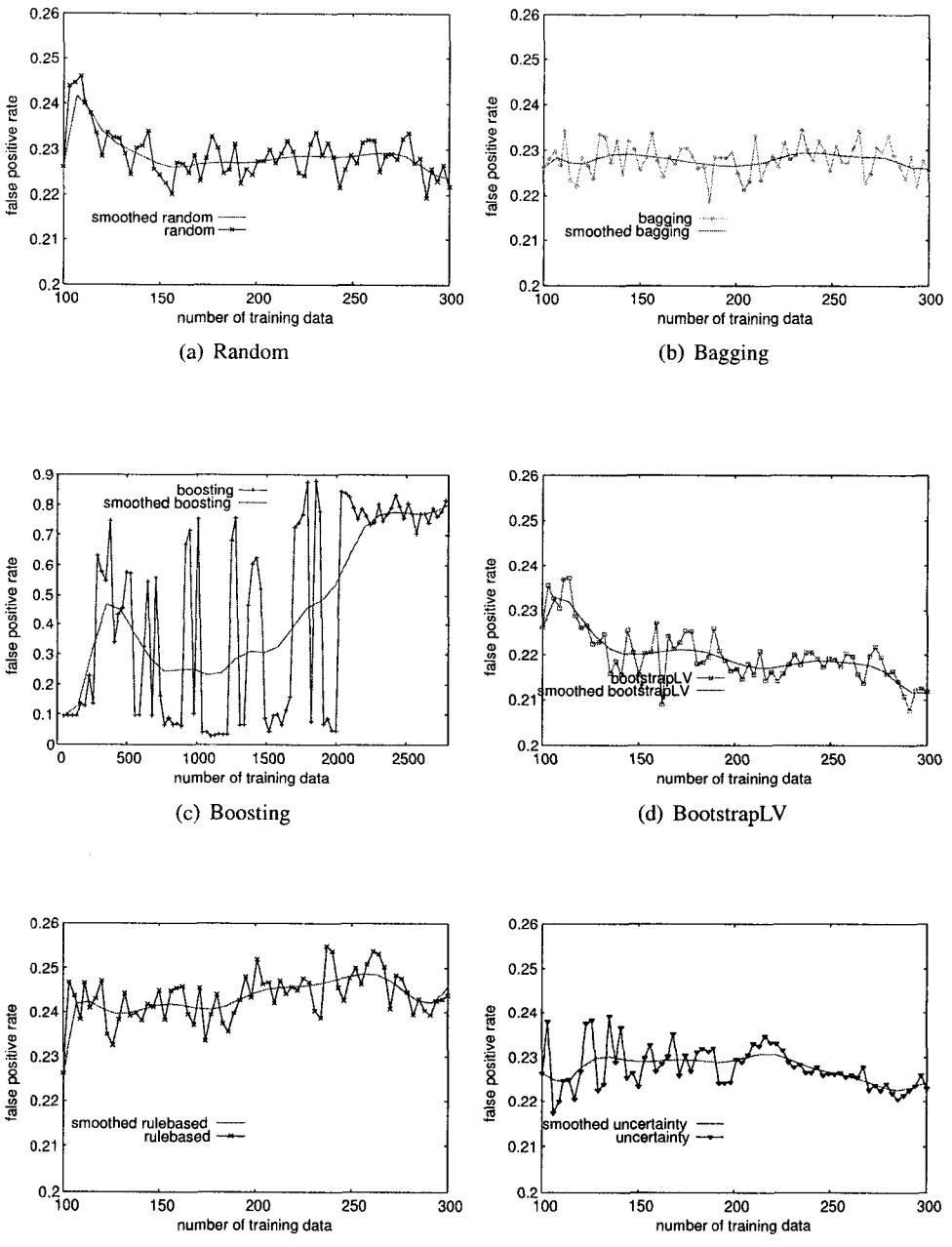


(e) Rulebased



(f) Uncertainty

Figure 4.19: Breakaway, evaluation on blurred target concepts, TP rate: smoothed *vs.* unsmoothed



(a) Random (b) Bagging

(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.20: Breakaway, evaluation on blurred target concepts, FP rate: smoothed *vs.* unsmoothed

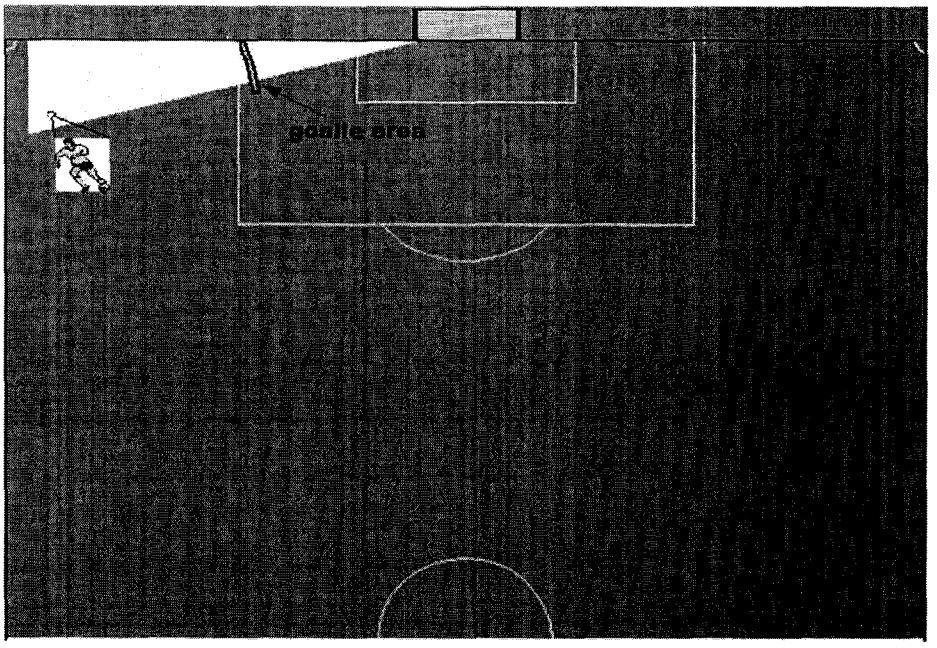


Figure 4.21: A visualized rule for the breakaway scenario

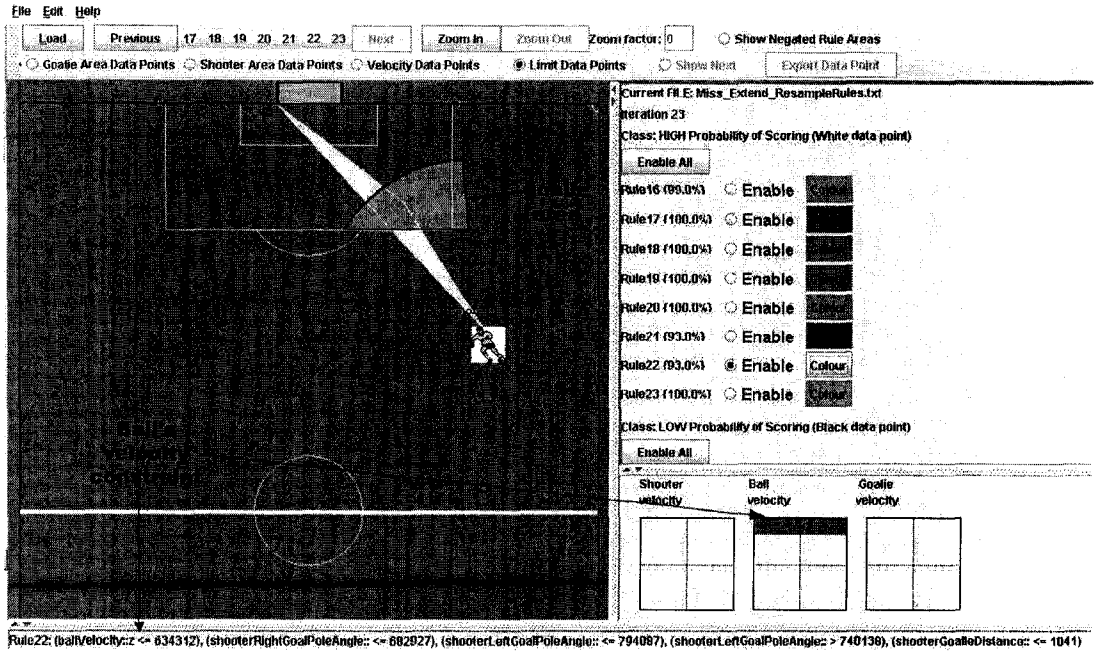


Figure 4.22: SoccerViz screen shot for 10-D breakaway scenario

make a chip shot.

4.4 Dribble-dribble-shoot scenario

The dribble-dribble-shoot scenario (Figure 4.23) is defined as: there is just one attacker with the ball, and no defenders except the goalie; the attacker dribbles the ball from a starting position and moves forward at a certain angle for a certain distance; then he changes his running direction and runs with the ball along the new direction until he gets a good opportunity to score, or until he runs out of bounds without getting a good opportunity to score. There are 7 dimensions in this scenario: this first 4 are for the initial positions of the attacker and the goalie (the distance between those two players and 3 angles as in the breakaway scenario); the fifth dimension is the direction of the attacker's first dribble action; the sixth dimension is the dribble distance the attacker will make along the first dribble direction; the last dimension is the direction (relative to the first dribble direction) of the attacker's second dribble action. The testing program will not control the goalie's movement. The main goal of this scenario is to test the applicability of *retrograde analysis* in this setting.

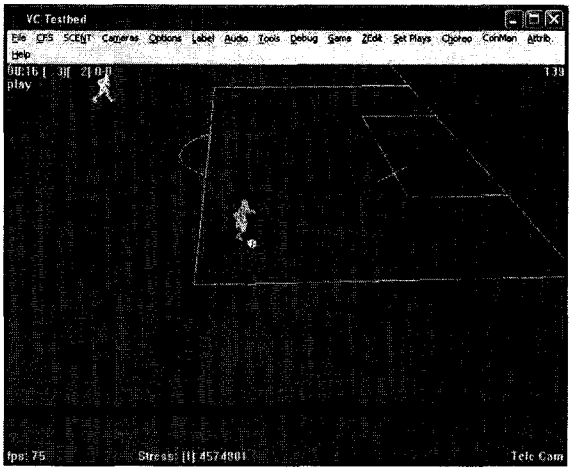
4.4.1 Retrograde analysis

Retrograde analysis means working back from simpler situations to more complex ones that depend on the simpler ones. For example, Jonathan Schaeffer used retrograde analysis to build a endgame database for his *checkers* program[33]. One of the benefits of retrograde analysis is that it can keep the number of dimensions fairly low. The retrograde analysis technique is used in this experiment in the following way: start by learning rules for simple game scenarios; the summarized rules for simple game scenarios are stored as retrograde analysis databases; the behavior of higher level game scenarios based on one or more low-level game scenarios will be summarized without repeating the same work that has already been done in the simpler game scenarios; instead, the results of the low-level game scenarios will be used directly.

Dribble-dribble-shoot is built on the top of the breakaway scenario. In the dribble-dribble-shoot scenario, the attacker will not actually try a shot; instead, after he has changed the dribble direction, the program will check his position and the goalie's position continuously and send that information to the decision tree learned for the breakaway scenario. If that decision tree says there is a high probability of scoring at any point, the label for this example is positive (meaning that there is a high probability of scoring for this dribble-

(a) the first dribble direction

(b) change the dribble direction



(c) the second dribble direction

Figure 4.23: Dribble-dribble-shoot scenario

dribble-shoot example); otherwise – if the shooter dribbles all the way to the field boundary without getting a good chance of scoring according to the breakaway decision tree – this example will be labelled as negative. Without retrograde analysis, the 4 (or 10) dimensions of the breakaway scenario would be added to dribble-dribble-shoot scenario. Therefore, at least 4 dimensions are saved by using retrograde analysis technique.

4.4.2 State Machine and Game Hooks for the Dribble-dribble-shoot Scenario

The dribble-dribble-shoot state machine has 4 states:

PREPARE

FIRST_DRIBBLE

SECOND_DRIBBLE

END

Before the state machine starts, the feature vector from the sample generator is read and the feature values are converted to internal game variables. Then the automatic testing program enters the first state: PREPARE. In this state, the attacker and the goalie will run to their initial positions, and the other players will be sent off the field. After everybody is in their designated positions, the FIRST_DRIBBLE state is triggered. The attacker dribbles the ball in the first direction. When the designated first dribble distance is reached, the state SECOND_DRIBBLE starts. The attacker will change the dribble direction and run with the ball along the new direction. The END state can be triggered by 4 events: the ball is out of bounds; the goalie has the ball; the breakaway retrograde analysis decision tree reports that the probability of scoring is high in the current situation; time out. The running of above state machine relies on some game hooks and modifications in the FIFA99 game, as follows.

- **void GAME_updateFrame()**

  This function has been described in 3.3.5. For this game scenario, the dribble-dribble-shoot state machine manager will be triggered here every game frame.

- **int TACTIC_doBestAction(PLAYER_DEF *this)**

  The original function will decide the current best kick action from the options of dribbling, passing and shooting etc. In the dribble-dribble-shoot scenario, we want the attacker to dribble twice along two designed directions. Therefore, this function is modified to serve the testing program by enabling dribbling only.

4.4.3 Experimental setup

There are 100 random sampled initial training examples, and 30 more examples will be added to the training set at each iteration. The model of the breakaway scenario is saved in files and will be loaded on the fly. Because the classification of each example is quite stable in this scenario, we do not run each example 10 times.

4.4.4 Experimental results

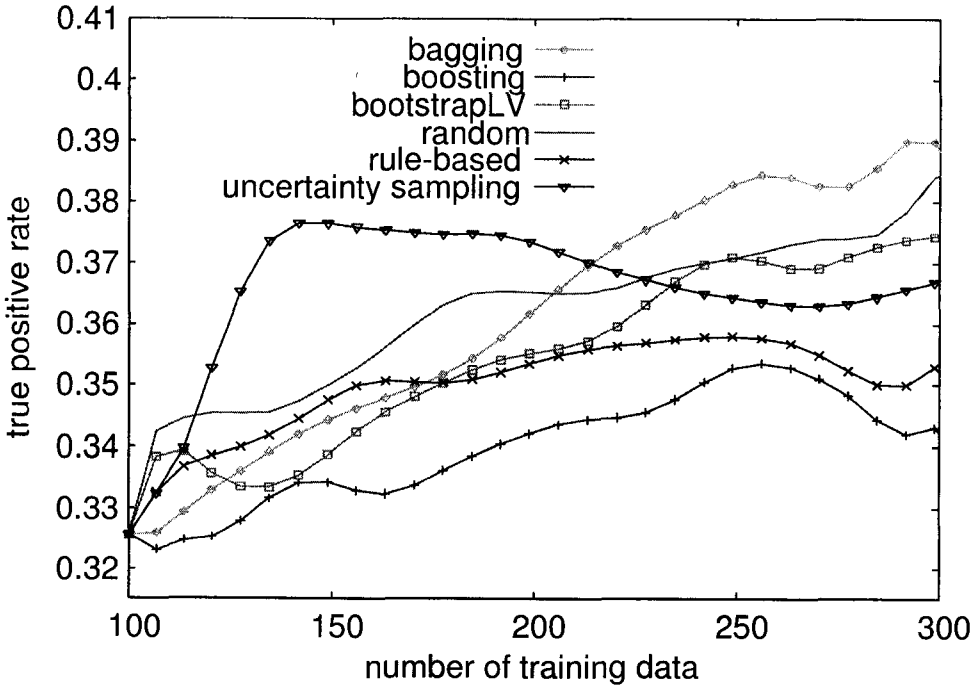
Figure 4.24 shows the standard evaluation results. In terms of FP rates, the rule-based method is worse than the others (the performance difference is significant [$\alpha = 0.1$] at training set size 250). In terms of TP rates, uncertainty is better than the others before 200 examples (the performance difference is significant [$\alpha = 0.1$] at training set size 150). Figure 4.27 shows the evaluation results on blurred target concepts. The cell size threshold $T_{cellsize}$ used in this scenario is 0.05, which means each edge of the cell is 0.05 of the whole range of a dimension. The minimum size for a group of cells to be kept is 30 cells. Surprisingly, the performance of all the sampling methods (including random sampling) is much worse than expected (compare to the results in the 10-D artificial tests): the TP rates are low; the improvements (for all sampling methods) are small even after 100 iterations; and the selective sampling methods do not behave better than random sampling as they did in the 8-D and 10-D artificial tests. It could be because this scenario is a complicated one and the number of samples is still too small to learn an accurate classifier. It also could be because of the retrograde analysis technique. The breakaway decision tree used in this scenario is not perfect (it cannot be perfect) for the real breakaway game scenario. The errors coming from this decision tree will be passed into the dribble-dribble-shoot analysis and these errors might be magnified. The investigation of these problems are left for future work.

Scenario Analysis

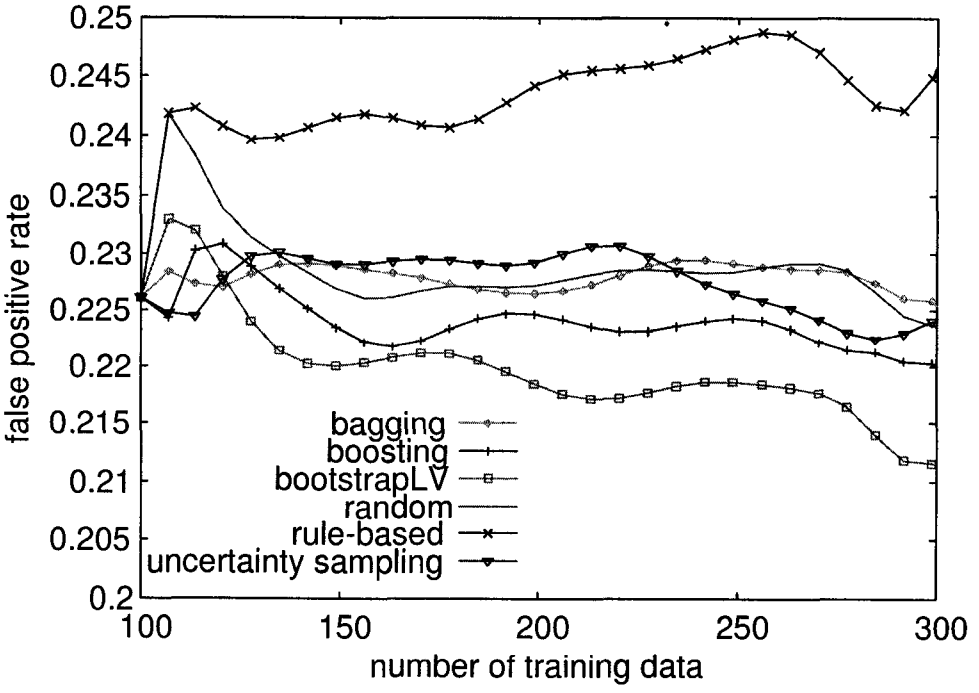
The dribble-dribble-shoot scenario is not visualized in SoccerViz. But manual analysis revealed some interesting rules. Figure 4.30 is a manually drawn picture of an interesting rule in this scenario. The exact rule definition is as follows:

Class: Positive, 7 conditions, 96.7% accuracy supported by 30 data points

$$\text{attackerRightGoalPoleAngle}(\alpha_1) > 65.13^\circ$$
$$\text{attackerFirstDribbleAngle}(\alpha_2) > 182.2^\circ$$

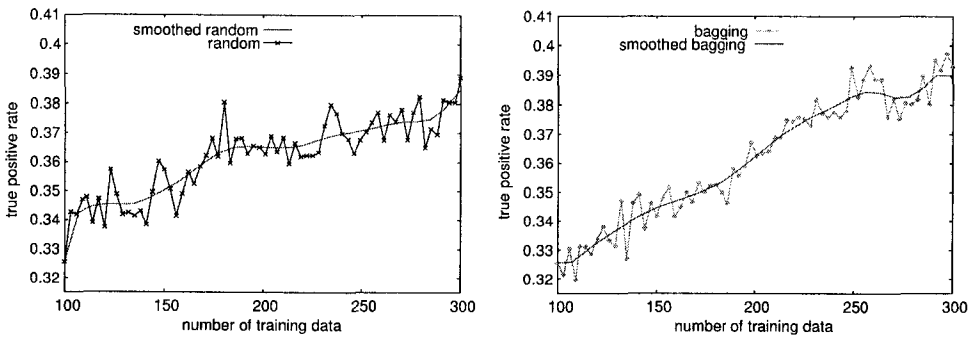


(a) TP rates



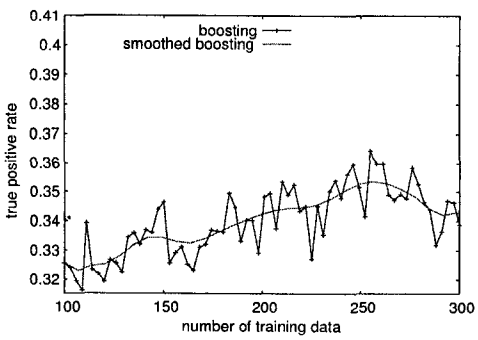
(b) FP rates

Figure 4.24: Dribble-dribble-shoot: standard evaluation

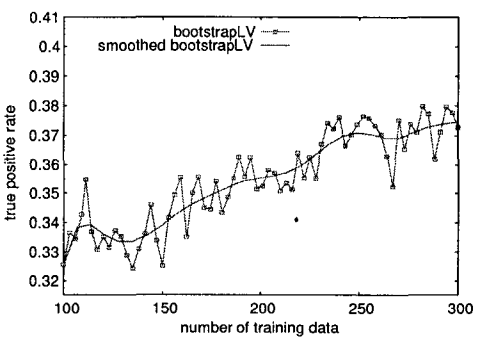


(a) Random

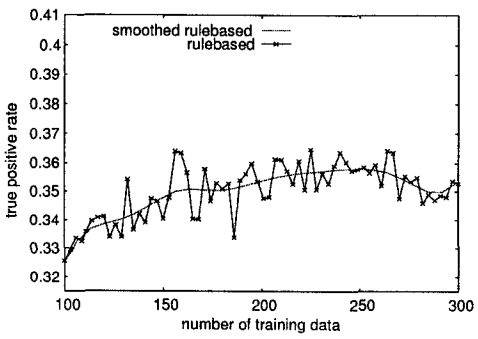
(b) Bagging



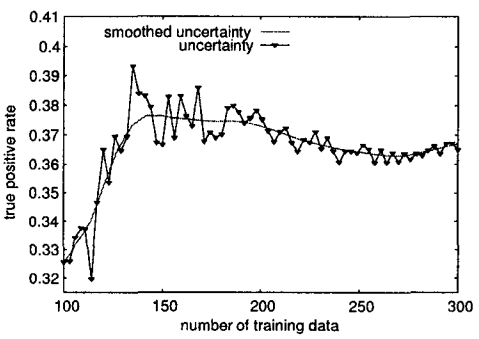
(c) Boosting



(d) BootstrapLV

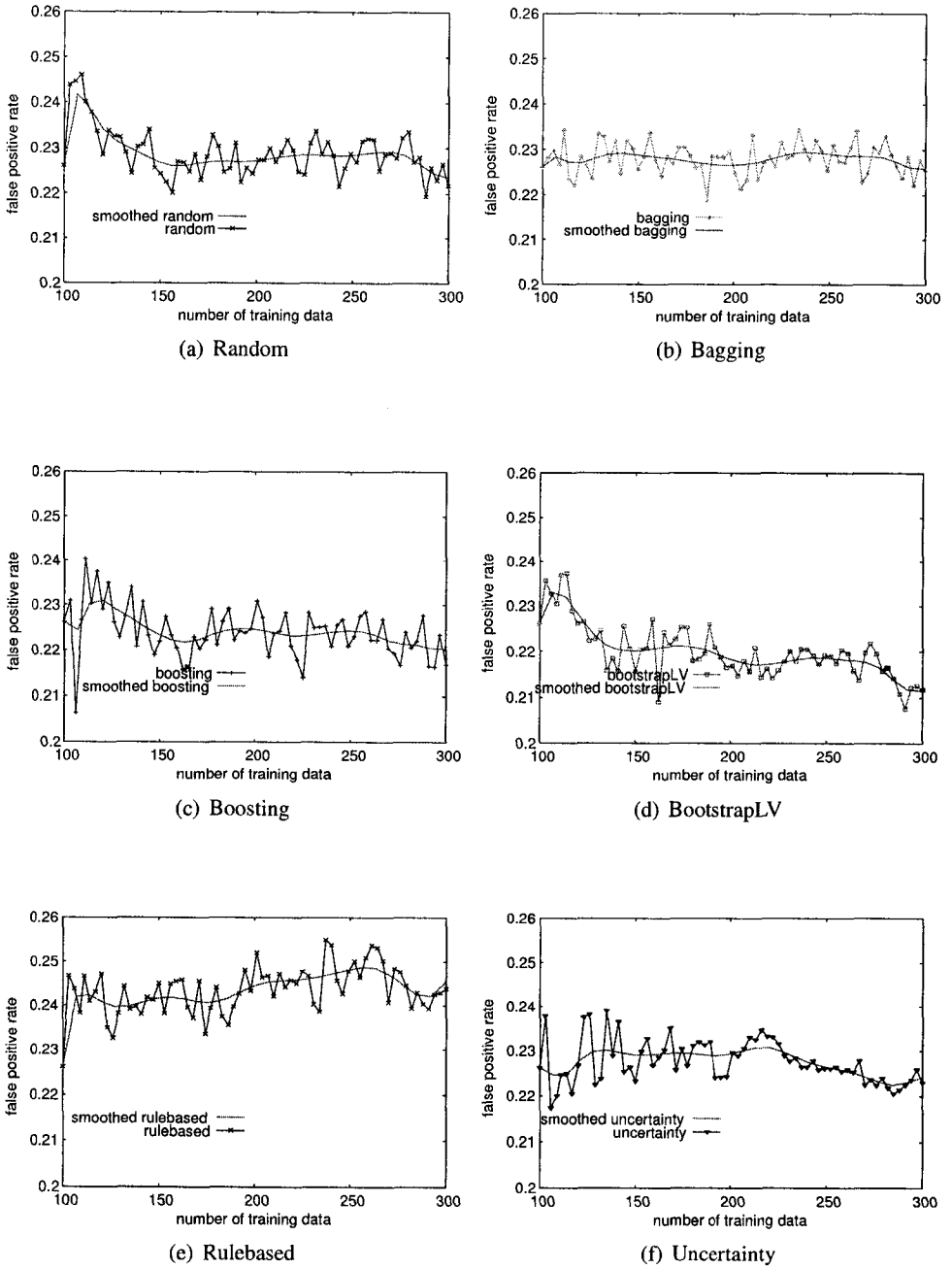


(e) Rulebased



(f) Uncertainty

Figure 4.25: Dribble-dribble-shoot, standard evaluation, TP rate: smoothed *vs.* unsmoothed

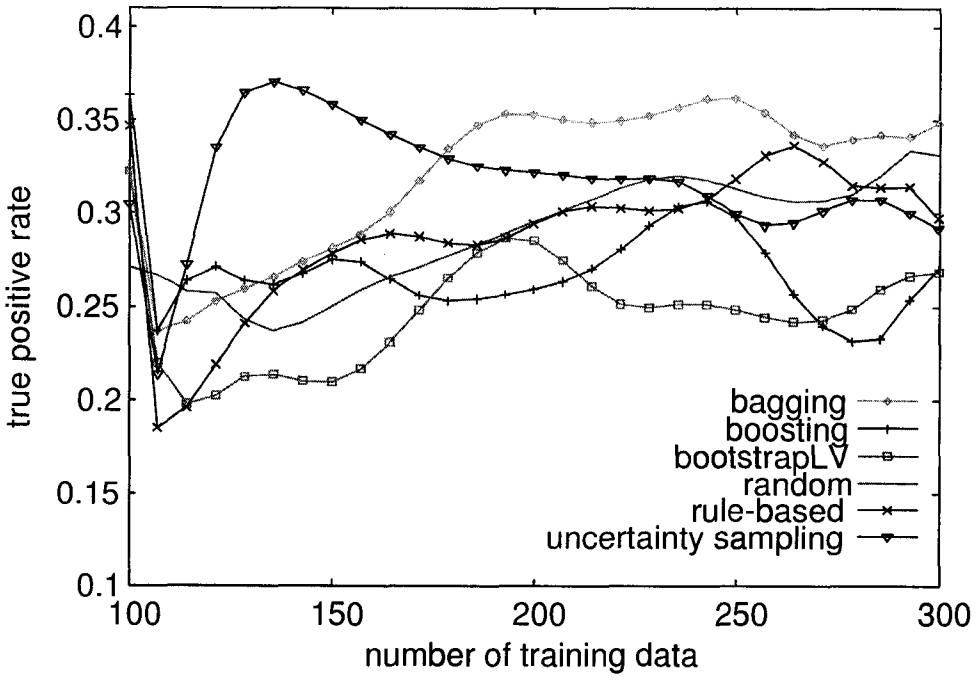


(a) Random (b) Bagging

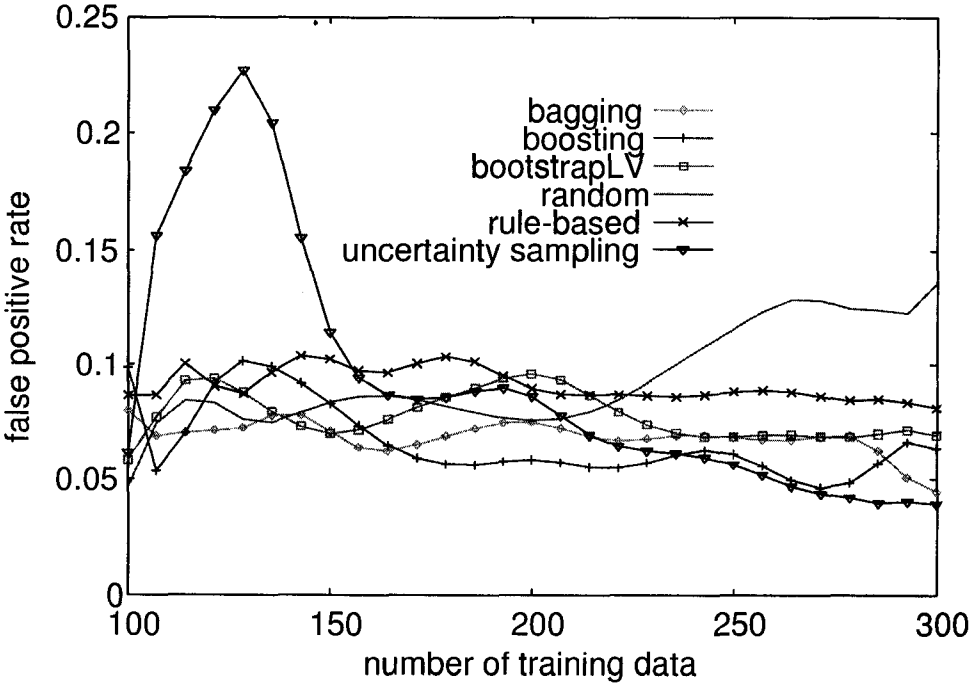
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

Figure 4.26: Dribble-dribble-shoot, standard evaluation, FP rate: smoothed *vs.* unsmoothed

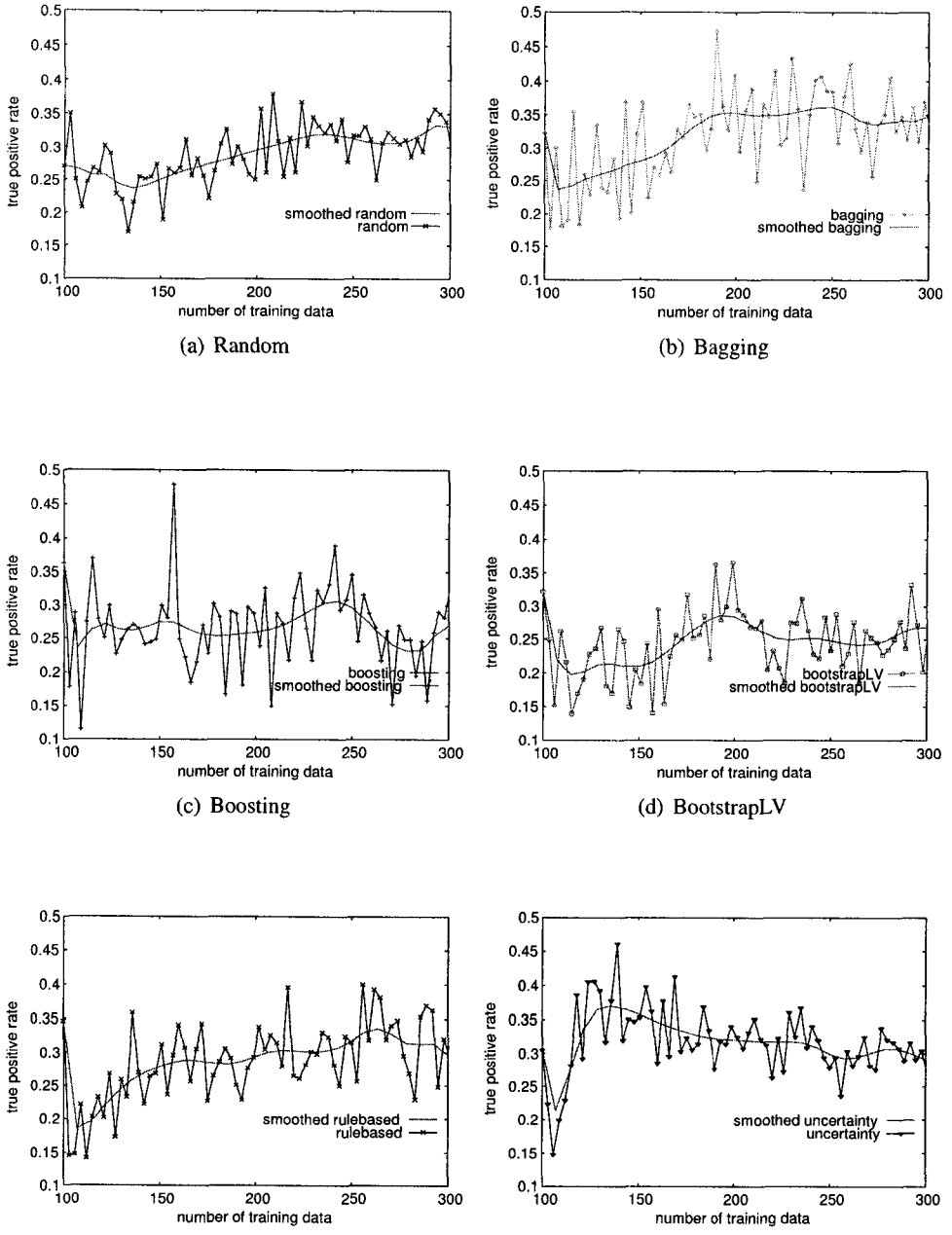


(a) TP rates



(b) FP rates

Figure 4.27: Dribble-dribble-shoot: evaluation on blurred target concepts

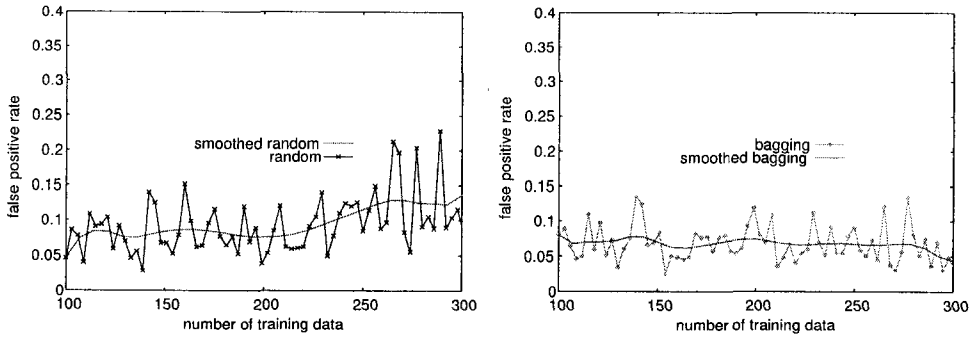


(a) Random (b) Bagging

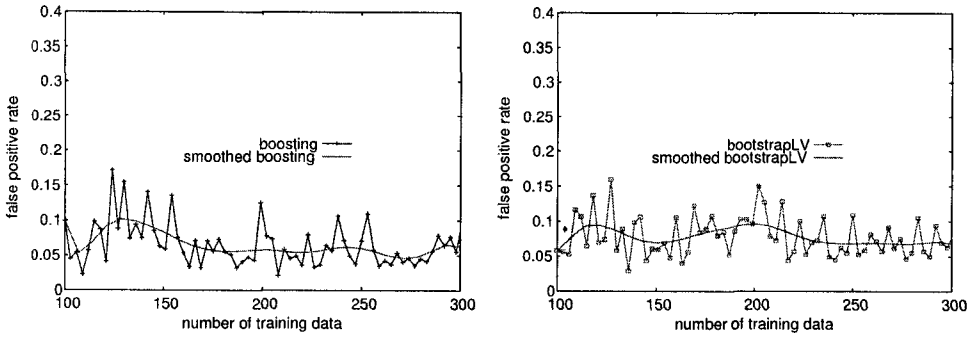
(c) Boosting (d) BootstrapLV

(e) Rulebased (f) Uncertainty

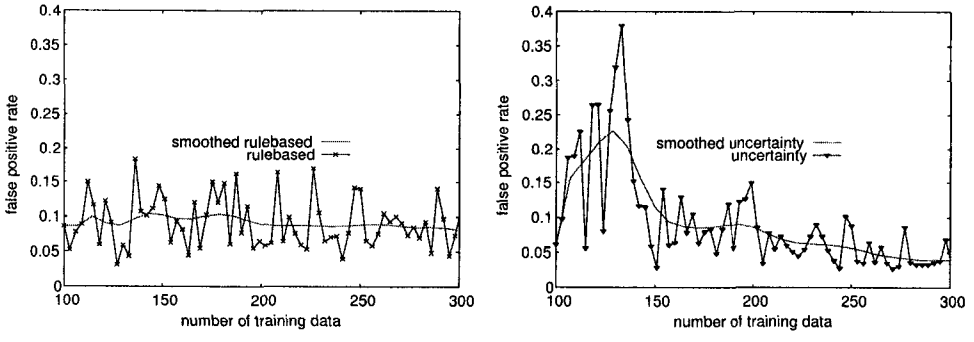
Figure 4.28: Dribble-dribble-shoot, evaluation on blurred target concepts, TP rate: smoothed *vs.* unsmoothed



(a) Random (b) Bagging



(c) Boosting (d) BootstrapLV



(e) Rulebased (f) Uncertainty

Figure 4.29: Dribble-dribble-shoot, evaluation on blurred target concepts, FP rate: smoothed *vs.* unsmoothed

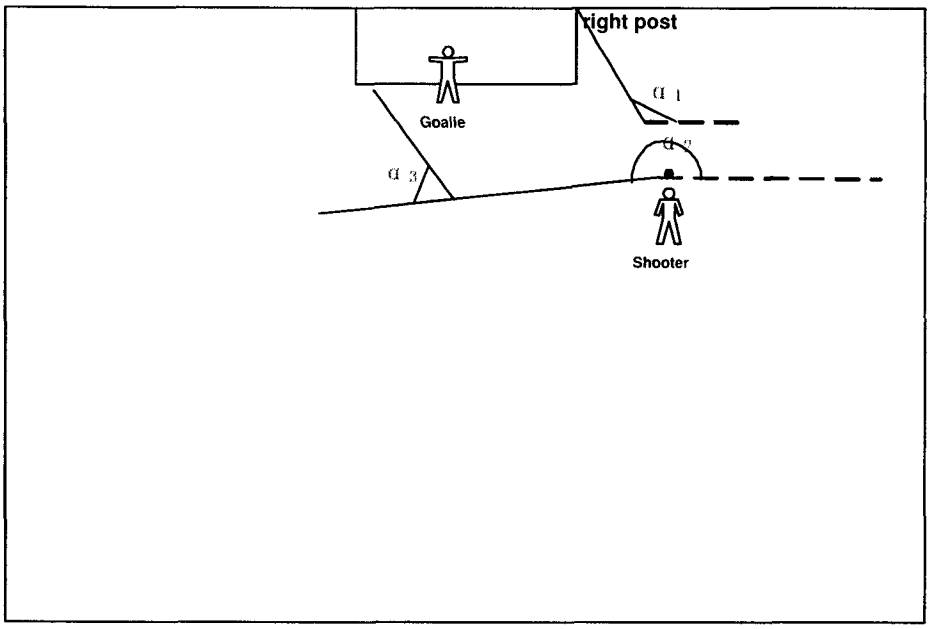


Figure 4.30: Interesting rule learned for the dribble-dribble-shoot scenario

$$\text{attackerFirstDribbleAngle}(\alpha_2) <= 210.57°$$

$$\text{attackerFirstDribbleDistance}(distance) > 226$$

$$\text{attackerFirstDribbleDistance}(distance) <= 748$$

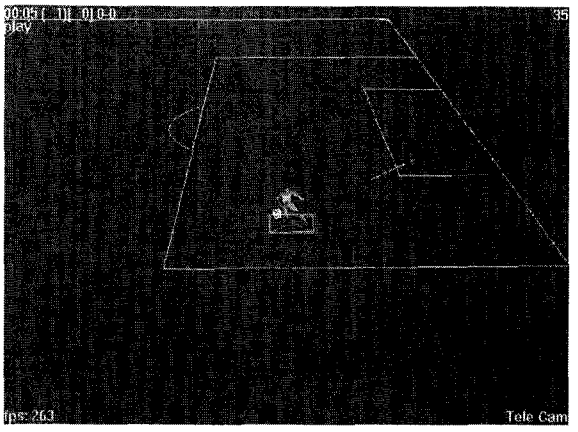
$$\text{attackerSecondDribbleAngle}(\alpha_3) > 42.95°$$

$$\text{attackerSecondDribbleAngle}(\alpha_3) > 114.46°$$

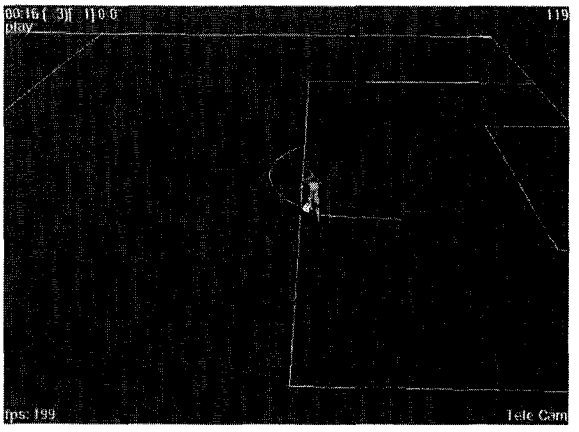
where α_1 is the angle between the attacker and the right goalpost for the shooter's starting position; α_2 is the first dribble direction; $distance$ is the first dribble distance; α_3 is the second dribble direction (relative to α_2).

An English paraphrase of this rule is: if the the attacker starts from the right of the right goalpost, runs roughly horizontally across the goal for a certain range of distances and then cuts sharply toward the goal, he will have an opportunity to score with high probability.

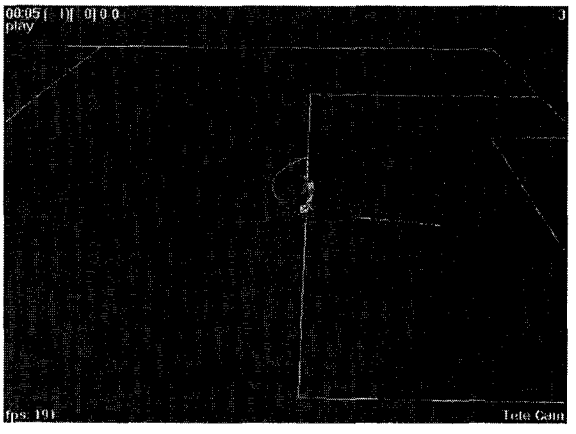
A game designer might want to know what happened to cause this pattern of attack to have a high scoring rate. When the 30 training examples supporting this rule are put back into FIFA99, some interesting *goalie* behaviors are found. When the attacker runs with the ball nearly horizontally along the field, the goalie sometimes starts out from the goal to challenge the attacker; but when the goalie gets far from the goal, he will stop the chase and run back to the goal; at the moment when the goalie is running back, if the attacker changes the dribble direction (towards the goal approximately), then the attacker will get



(a)



(b)



(c)

Figure 4.31: Goalie behaviour in the dribble-dribble-shoot scenario

the scoring chance. Figure 4.31 shows some FIFA99 game screen shots for one of the training examples supporting this rule. In Figure 4.31(a) and Figure 4.31(b), the goalie is coming out to challenge the attacker. In Figure 4.31(c), the goalie is running back to the goal and the attacker changes the dribble direction towards the goal, leading to a good scoring opportunity.

4.5 Conclusion

SAGA-ML and SoccerViz have been used to analyze three different FIFA99 game scenarios (corner kick, breakaway and dribble-dribble-shoot). The rules generated from SAGA-ML summarize the game's behavior in terms of relationships between various distances and angles and the probability of scoring. Some sweet spots have been identified. In addition to the standard evaluation method (measure FP and TP using a large test set), an evaluation method based on blurred target concepts has been introduced and used for FIFA99 scenarios. The use of blurred target concepts has two advantages over the standard evaluation method: (1) in most game scenarios (e.g. corner kick and breakaway), it produces better TP rates; and (2) the TP and FP rates generated using blurred target concepts are more meaningful because it only evaluates large regions, which are of the greatest interest to game developers and human players. Random sampling performs well in the corner kick scenario, just as it did in 2-D artificial testing. Bagging, boosting and rule-based selective sampling methods are better than random sampling overall, just as they were in 4-D artificial testing. In the beginning stages of all 3 game scenarios, the rule-based sampling method always has the best TP rates while having FP rates similar to the other sampling methods. Thus, given a limited number of examples, the rule-based sampling method is the best one among the 6 sampling methods tested.

Chapter 5

Conclusion and Future Work

5.1 Summary

A game scenario testing framework, Semi-Automated Gameplay Analysis (SAGA-ML) is presented in this thesis. SAGA-ML summarizes game behaviors as human readable rules, which can be presented to game designers to check if those behaviors are as intended. SAGA-ML has been tested on Electronic Arts' FIFA99 soccer game and shown to be a practical game behavior testing solution.

Three Fifa99 game scenarios — *cornerkick*, *breakaway* and *dribble-dribble-shoot* — were tested by the SAGA-ML system. Some interesting rules were found by SAGA-ML. Among those findings, some of them are sweet spots of the game (e.g., the easy scoring area in the corner kick scenario).

Four existing selective sampling algorithms (*Uncertainty Sampling, Bagging, Boosting* and *BootStrapLV*) were implemented, and a new rule-based selective sampling method was introduced. Those 5 selective sampling methods and random sampling were compared. Experimental results (especially when dimensionality is high) proved that selective sampling algorithms are more efficient than random sampling. Given a limited number of examples, the rule-based sampling method is the best one among these 6 sampling methods.

A new evaluation method based on blurring the target concept was introduced and used for FIFA99 game scenario. Blurring generates larger regions in its concept definitions by eliminating small regions and merging neighbouring regions. Compared to the standard evaluation method, the use of blurred target concepts produces better TP rates in most game scenarios (e.g. corner kick and breakaway), and this evaluation method is more meaningful because it only evaluates large regions, which are of the greatest interest to game developers and human players.

5.2 Limitations

SAGA-ML was proposed as a general gameplay analysis framework. However, it is not a perfect solution to the entire problem of gameplay analysis for all possible games. After all, the cost of the whole process of learning and sampling is not cheap. Extra design and programming will be added to game development if SAGA-ML is used. The second limitation is about *thresholds*. As we have seen in this thesis, there are many thresholds used throughout SAGA-ML system. Currently, those thresholds are tuned manually for each game scenario. When a new game scenario is implemented, some thresholds must be adjusted based on experiments.

5.3 Future Work

5.3.1 Initial training data size

An issue that was not addressed in this thesis is how to get the best initial training data size for active learning. The following initial idea has been briefly explored. A pre-defined threshold is used to indicate the minimum size of target concepts, and the number of initial training samples is set proportional to this threshold. Ideally, in this way the learner will find the rough locations of the target regions whose sizes are bigger than the given threshold with the initial training data. Then, the active learning process will refine the initial concepts iteratively. One problem with this method is that it can be hard to determine a reasonable threshold for real applications. Therefore, this method is not used in this thesis. More work should be done on this topic in the future.

5.3.2 How many new points are added in each iteration

Anther issue that was not addressed in this thesis is how many new data points should be picked in each iteration. Selective sampling is a sort of *sequential sampling*, which can be traced back to 1940s when Wald [42] introduced the concepts *Sequential Testing and Sampling* to the statistics community. In the database community, a branch of sequential sampling was investigated by some researchers, e.g., [28]. They believed there exists a learning curve (*exponential curves* are a common assumption) during the sequential learning process. Suppose there is a set of sequential example sets, denoted as $S_1, S_2....S_n$, where S_i is the new example set for iteration i. The focus of this branch of research is to determine the proper size for each S_i, until stopping conditions are met. For example, a power law curve can be assumed for the learning curve, which is denoted as $y = a + b * x^c$.

Many methods are used to get the parameters a, b and c. The first solution to this task is using *nonlinear regression* techniques. Once the learning curve is determined, the value of each S_i can be drawn from a power law formula. In this thesis, the fixed size of new examples is generated in each iteration. The future work is to apply more complex methods (e.g., the learning curve method) to determine the new example size for each iteration.

5.4 Final Word

Gameplay testing in the current development cycle of commercial games is a kind of play-and-feel method, which relies on developers' experience. This thesis has presented a Semi-Automated Gameplay Analysis (SAGA-ML) system to help game developers to summarize and analyze gameplay behaviors. SAGA-ML is a system which incorporates *machine learning* techniques and the *active learning* framework into gameplay testing. The gameplay behaviors of real game scenarios are summarized, iteratively refined, and visualized by SAGA-ML. As a new system, SAGA-ML needs to be improved and tuned by applying it to additional commercial games.

Bibliography

[1] Deepam Agarwal. A comparative study of artificial neural networks and info fuzzy networks on their use in software testing. In *Master's thesis*. Department of Computing Science and Engineering, University of South Florida, 2004.

[2] C. Anderson, A. von Mayrhauser, and Rick Mraz. On the use of neural networks to guide software testing activities. *Procs. International Test Conference.*, 1995.

[3] B. Beizer. Software testing techniques, 2nd edition. Van Nostrand Reinhold, 1990.

[4] Paul Bourke. Bezier curves, reviewed by Paul Bourke. http://astronomy.swin.edu.au/ pbourke/curves/bezier, 1989.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt. Select — a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[6] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[7] John Buchanan, Finnegan Southey, Gang Xiao, Robert C. Holte, and Mark Trommelen. Machine learning for semi-automated gameplay analysis. *Game Developers Conference (GDC)*, 2005.

[8] Ben Chan, Jorg Denzinger, Darryl Gates, Kevin Loose, and John Buchanan. Evolutionary behavior testing of commercial computer games. *Proceedings of the 2004 Congress on Evolutionary Computation (CEC)*, pages 125–132, 2004.

[9] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 705–712. The MIT Press, 1995.

[10] Scott Dick, Aleksandra Meeks, Mark Last, Horst Bunke, and Abraham Kandel. Data mining in software metrics databases. *Fuzzy Sets and Systems*, 145(1):81–110, 2004.

[11] Chris Drummond and Robert C. Holte. Exploiting the cost of (in)sensitivity of decision tree splitting criteria. In *Proc. 17th International Conf. on Machine Learning*, pages 239–246. Morgan Kaufmann, San Francisco, CA, 2000.

[12] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[13] International Organization for Standardisation. (iso)(1991). iso/iec: 9126 information technology-software product evaluation-quality characteristics and guidelines for their use. 1991. [verified 10 Oct 2004].

[14] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[15] Bogdan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *ICSE*, pages 71–80, 1996.

[16] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[17] Miroslav Kubat, Robert C. Holte, and Stan Matwin. Machine learning for the detection of oil spills in satellite radar images. *Machine Learning*, 30:195–215, 1998.

[18] Frank Lammermann, André Baresel, and Joachim Wegener. Evaluating evolutionary testability with software-measurements. In *GECCO (2)*, pages 1350–1362, 2004.

[19] Mark Last, Menahem Friedman, and Abraham Kandel. The data mining approach to automated software testing. In *KDD*, pages 388–396, 2003.

[20] Wee Kheng Leow, Siau-Cheng Khoo, Tiong Hoe Loh, and Vivy Suhendra. Heuristic search with reachability tests for automated generation of test programs. In *ASE*, pages 282–285, 2004.

[21] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In W. Bruce Croft and Cornelis J. van Rijsbergen, editors, *Proceedings of SIGIR-94, 17th ACM International Conference on Research and Development in Information Retrieval*, pages 3–12, Dublin, IE, 1994. Springer Verlag, Heidelberg, DE.

[22] Andrew K. McCallum and Kamal Nigam. Employing EM in pool-based active learning for text classification. In Jude W. Shavlik, editor, *Proceedings of ICML-98, 15th International Conference on Machine Learning*, pages 350–358, Madison, US, 1998. Morgan Kaufmann Publishers, San Francisco, US.

[23] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[24] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[25] Jonathan Newton. Semi-automated gameplay analysis for role-playing games. In *Master's thesis*. Department of Computing Science, University of Alberta, 2005.

[26] Institute of Electrical and Electronics Engineers. Dictionary of measures to produce reliable software. In *IEEE*, New York, 1988. IEEE Standard 982.1-1988.

[27] National Institute of Standards & Technology. The economic impacts of inadequate infrastructure for software testing. *Planning Report 02-3*, May 2002.

[28] Foster J. Provost, David Jensen, and Tim Oates. Efficient progressive sampling. In *Knowledge Discovery and Data Mining*, pages 23–32, 1999.

[29] J. Ross Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, 1993.

[30] Ray Robinson. Automation test tools. http://www.vcaa.com/tools, 2001.

[31] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[32] Maytal Saar-Tsechansky and Foster Provost. Active learning for class probability estimation and ranking. *IJCAI*, pages 911–920, 2001.

[33] Jonathan Schaeffer, Joseph C. Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[34] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 287–294, 1992.

[35] Billie Shea. Software testing gets new respect. http://www.informationweek.com/793/testing.htm, 25.10.2004.

[36] Finnegan Southey and Robert C. Holte. Semi-automated gameplay analysis. *AAAI-04 Challenges in Games Workshop*, 2004.

[37] Finnegan Southey, Gang Xiao, Robert C. Holte, Mark Trommelen, and John Buchanan. Semi-automated gameplay analysis by machine learning. *Proceedings of the 2005 Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE-05)*, pages 123–128, 2005.

[38] F. Souvannavong, B. Merialdo, and B. Huet. Partition sampling: an active learning selection strategy for large database annotation. *IEE Proceedings - Vision, Image, and Signal Processing*, 152:347–355, 2004.

[39] Nigel Tracey, John A. Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *ISSTA*, pages 73–81, 1998.

[40] Nigel Tracey, John A. Clark, Keith Mander, and John A. McDermid. An automated framework for structural test-data generation. In *ASE*, pages 285–288, 1998.

[41] Jeffrey M. Voas and Gary McGraw. Software fault injection: Inoculating programs against errors. Wiley, 1998.

[42] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, 1947.

[43] Joachim Wegener, André Baresel, and Harmen Sthamer. Suitability of evolutionary algorithms for evolutionary testing. In *COMPSAC*, pages 287–289, 2002.

[44] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *GECCO (2)*, pages 1400–1412, 2004.

[45] J. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1), January/February 2000.

[46] J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. In *IEEE Transactions on Software Engineering*, pages 812–824, 1994.

[47] Gang Xiao, Finnegan Southey, Robert C. Holte, and Dana Wilkinson. Software testing by active learning for commercial games. *AAAI*, pages 898–903, 2005.

[48] Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. *The Twentieth International Conference on Machine Learning (ICML)*, pages 912–919, 2003.