Bugs or Anomalies? Sequence Mining based Debugging in Wireless Sensor Networks

Kefa Lu, Qing Cao, Michael Thomason
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, Tennessee 37919
Email: {klu3, cao, thomason}@utk.edu

***Abstract*—WSN applications are prone to bugs and failures due to their typical characteristics, such as being extensively distributed, heavily concurrent, and resource restricted. In this paper, we propose and develop a flexible and iterative WSN debugging system based on sequence mining techniques. At first, we develop a data structure called the vectorized Probabilistic Suffix Tree (vPST), an elastic model to extract and store sequential information from program runtime traces in compact suffix tree based vectors. Then, we build a novel WSN debugging system by integrating vPST with Support Vector Machines (SVM), a robust and generic classifier for both linear and nonlinear data classification tasks. Finally, we demonstrate that the vPST-SVM debugging system is efficient, flexible, and generic by three different test cases, two on the LiteOS operating system and one on the TinyOS operating system.**

I. Introduction

In the past decade, Wireless Sensor Networks (WSNs) have been widely developed and deployed for various purposes, such as environmental monitoring and data collection [1], [2], [3]. However, WSN applications are still suffering from numerous types of bugs and frequent failures [3], [4], due to their typical characteristics, such as distributed architecture, concurrent execution model, and strict resource limitations. It is difficult to perform efficient debugging on WSN applications, because many of them are context sensitive and event driven. It is usually infeasible to fully control their operating context and triggering events. In addition, many WSN bugs are transient and irreproducible [5]. Therefore, it becomes a big challenge for current WSN researchers and developers to design and develop robust WSN debugging systems.

In this paper, we design, implement, and evaluate a flexible and generic debugging system based on sequential data analysis and outlier detection techniques. Our approach is based on two theoretical models, the vectorized Probabilistic Suffix Tree (vPST) model and the Support Vector Machine (SVM) model. The original PST model is a flexible probabilistic model that can efficiently extract and store sequential information from sequences in compact suffix tree data structures [6], while SVM is a robust and generic classification technique that can solve both linear and nonlinear classification problems [7]. By extending PST to vPST, we are able to not only retain the sequential information but also the most significant substructures within sequences in compact and simple vectors. SVM can be easily applied on these vectors to detect outliers in the sequences. By combining the vPST model and the SVM classifier together with an efficient tracing subsystem, we find that the resulting technique proves to be immensely effective to locate real bugs.

Our contributions in this paper are two-fold. First, we extend the PST model to the vPST model, which provides researchers a new methodology for extracting and analyzing sequential information. Specifically, the vPST model breaks sequences into pieces and stores them in meaningful data structures. Second, we propose the vPST-SVM system, which is especially helpful for detecting transient bugs. The whole debugging process is iterative, meaning that it allows the user to adjust debugging settings dynamically to achieve the best results. The whole system is evaluated by comparing prediction results for various test cases on different operating systems, where we incrementally changed the vPST depths during iterative debugging cycles.

The following of this paper is organized as follows. In section 2, we briefly discuss related work, including some proposed WSN debugging systems. In section 3, we describe details on our vPST model and our iterative vPST-SVM anomaly detecting approach. In section 4, we describe our system design and implementation. In section 5, we present three interesting test cases for system evaluation. Section 6 concludes this paper with some discussions.

II. Related Work

In the past decade, many different WSN debugging systems have been proposed [8], [9], [10]. However, some of them were not easily portable due to strong dependency on specific operating systems [10], [8]. Some others were restricted to source code analysis or simulation trace analysis [8], [9]. There is still a significant lack of efficient debugging systems that can fully take advantage of runtime traces from real deployments. In fact, there are many tricky bugs caused by race conditions or inappropriately controlled concurrencies that can only be triggered in real deployment under some specific circumstances.

Indeed, there have been some efforts on developing trace based WSN debugging systems by data mining techniques. Dustminer [11] is based on frequent pattern mining. There are three significant drawbacks of it. First of all, it is based on and limited to frequent patterns mining. So Dustminer will fail to detect the bugs that only generate infrequent patterns. Secondly, it requires a lot of human effort to figure out clear

978-1-4673-2433-5/12/$31.00 ©2012 IEEE

definitions of good patterns and bad patterns for data training. As a matter of fact, before the bug is found, it is usually not practical to define clearly good patterns and bad patterns. Third, frequent patterns mining is usually very expensive if the sequence is long. Sentomist [10] is another trace based WSN debugging system. It suffers from several major drawbacks as well. First, it is strongly restricted to TinyOS operating system, as its event handling interval model is specifically based on the event-driven execution mechanism of TinyOS. Second, it only collects application execution traces from simulations, but not real deployment. Third, it uses only instruction counters, so all sequential information is practically ignored. For example, two different sequences (1, 1, 2) and (2, 1, 1) can be considered identical when they are converted into instruction counters.

III. vPST-SVM Anomaly Detecting System

A. Vectorized Probabilistic Suffix Tree

For the traditional PST model, analyzing PSTs to derive meaningful insights is always challenging because PSTs constructed from different sequences usually have different structures. In the past decade, different approaches were proposed to analyze the similarities between different PSTs [12]. Given two sequences S1 and S2, previous proposed approaches, such as the *Odds* and the *Normalized* measure [12], calculate their similarity based on statistical analysis of similarities of their subsequences. However, similarities of the entire sequences as whole structures are not fully taken into account. In particular, high similarities between some subsequences of two sequences does not necessarily convert to their entire structural similarity. Therefore, we have to extend the traditional probabilistic suffix trees to address our target systems.

Our proposed Vectorized Probabilistic Suffix Tree (vPST) solves the aforementioned issue as follows. Observe that there is a probability vector on each PST node, which is composed of conditional probabilities of every symbol occurring after the given symbol subsequence of that node. For those subsequences with no occurrence, we can still construct corresponding nodes by associating them with vectors formed by zeroes. In this way, we can construct full PSTs to some predefined depth. Next, we vectorize all these PSTs by combining separate vectors on all nodes. Take a full PST in Figure 1 for example. It is constructed from a sequence (1 3 2 2 3 2 1) with all nodes up to length 2. A vector can be constructed from this PST by sequentially combining probability vectors on all nodes in the order (root, 1, 2, 3, 1_1, 2_1, 3_1, ..., 3_3) as shown in Figure 2.

Because of the way that they are constructed, vPSTs have many advantages over PSTs when they are analyzed by SVM. First, the most important sequential information and substructures within the sequences are retained as well by vPST vectors as by the original PST, but in a simpler data structure. Second, analyzing vectors is much easier than analyzing suffix trees for SVMs. In fact, SVMs can be applied directly on vPST vectors for outlier detection. Third, when analyzing sequences using vPSTs, we also take into account the similarity between entire structures instead of partial similarities between subsequences

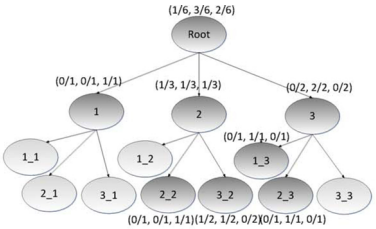

Fig. 1: a PST Example

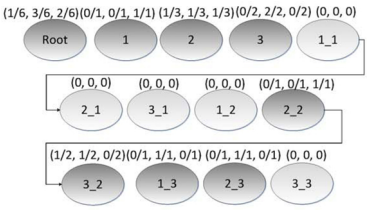

Fig. 2: a Vectorized PST Example

[12]. Therefore, this approach is more suitable for detecting bugs that are usually outliers with subtle structural changes of program control flows in runtime traces.

B. vPST-SVM Iterative Debugging System

As a natural next step for vPST, we integrate it with a robust classification approach, SVM, which is powerful on analyzing high-dimensional data. As a result, we develop vPST-SVM sequence analyzing approach and demonstrate its efficiency by case studies. Detailed steps of the iterative vPST-SVM debugging process are listed below.

1) Collect runtime traces from a possibly faulty program.
2) Construct PSTs using given sequences from these traces.
3) Vectorize PSTs to generate vPST vectors.
4) Apply SVM on vPST vectors to obtain a short list of top anomalous sequences (usually 10 to 20 sequences) based on calculated anomaly scores.
5) Check the short list of anomalous sequences to decide if they are real bugs.
6) Fix the confirmed anomalies, and go back to step 1.
7) Stop until no more bugs are detected.

In the debugging process, the depth of vPSTs is a significant parameter that can be tuned adaptively for efficient bugs detection. It should be increased for tricky bugs that are hard to be detected by low-depth vPSTs. Meanwhile, it should be decreased for simple bugs to save resources and improve efficiency. In principle, the larger the vPST depth is, the more sequential information is extracted and retained from sequences, and bugs are more likely to be detected by analyzing the collected sequences. However, a larger vPST depth also indicates more expensive computing in later analysis. In fact, the size of a vPST will increase exponentially with its size, which limits the maximum depth of vPSTs. Fortunately, we observe that sequences have the so-called *short memory* property, which guarantees that vPSTs with reasonable low depths can still detect a majority of potential bugs. In our

case studies, we observed reasonable debugging results even we only explored vPSTs with a depth no larger than 5.

The computational complexity of the iterative vPST-SVM debugging process is mainly determined by the construction of PSTs and the application of SVMs on the vPST vectors. The former is approximately $O(Ln \log n)$ based on results from [13], where L is total number of sequences in the analysis, and n is average length of all sequences. The latter is approximately $O(Ln_{sv} + {n_{sv}}^3)$ based on [14], where n_{sv} is total number of support vectors. So the total computational complexity of the iterative vPST-SVM debugging approach is approximately $O(N(Ln \log n + Ln_{sv} + {n_{sv}}^3))$, where N is number of iterations.

IV. System Design and Implementation

A. System Design

The basic assumption in our overall design stems from our experience of actual WSN application development. For most of the time when a developer encounters problems with his/her application development, he/she usually has some sense that some parts of the source code are most probably the cause of the problem, even he/she is not sure exactly what it is. These problematic blocks of codes are called *hot spots*. Based on this assumption, we designed the system so that it can focus on hot spots in the source code, so that we can significantly reduce collected data and analyzing overhead.

The vPST-SVM debugger is composed of four components as shown in Figure 3, a runtime trace collector, a preprocessor, a vPST analyzer, and a SVM classifier. In the front end, the trace collector generates and collects program runtime traces based on trace points in hot spots of applications. In the back end, the preprocessor, vPST analyzer and SVM classifier work coordinately to detect the most significant outliers in collected traces. The preprocessor filters out noises and uninteresting data, compresses the raw data, and reduces data dimensionality through a preliminary sequence analysis procedure. The vPST analyzer then goes through all the collected runtime traces and extracts sequential features from them. Next, the SVM classifier performs classification based on those vPST vectors, and provides the results to the developers. After manually reviewing the very top anomalies in the list, the developer may fix these bugs, and iteratively go through the steps again to detect new bugs.

In the implementation of the trace collector, we built lightweight tracing subsystems on both LiteOS and TinyOS. On LiteOS, all the trace points were sent to the desktop computer through the serial port via a USB cable and printed out using pre-implemented printing functions. On TinyOS, the existing printf library was used to collect data in a similar way. We implemented the vPST sequence analyzer also in C++ for efficiency considerations. The implemented PST construction algorithm followed the published algorithm by Ron et al. [6]. The step of PST vectorization followed our discussions of vPST earlier, where it adaptively extracts sequential information from given sequences and stores all the information into compact vPST vectors up to a given PST depth of L. Finally,

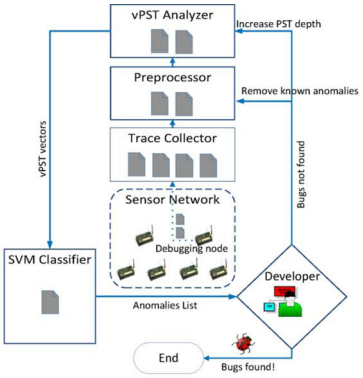

Fig. 3: System architecture

the SVM sequence classifier is another key component of the vPST-SVM debugging system. In our debugging system, we used a SVM library Libsvm [15]. We used both of the Java version and C++ version of it in our experiments. There is no significant difference in their performance.

V. Case Studies

We evaluated the proposed debugging system by three test cases, two on LiteOS and one on TinyOS, as described as follows.

A. Test Case I: a variable overflow bug on LiteOS

In this test case, a LiteOS application that transfers packets between different nodes was developed based on a simple reliable data transfer protocol. Several sender nodes continuously send radio packets to a single receiver node. After receiving a packet, a receiver always sends back an ACK packet to the sender. For each sender node, after sending out a packet, it always waits a while for an ACK. If an ACK is not received on time, it just resends the packet. Only after an ACK is successfully received, a new packet will be sent out. Before the bug was found and fixed, we did not have any prior knowledge of any hidden bugs in the application. So the test case studied here was actually a real debugging process of an actual development of a WSN application. Even for such a simple application, there were still a few interesting bugs coming out during the development process. The following debugging process of a variable overflow bug shows how the vPST-SVM debugging system could help developers to find and fix WSN bugs efficiently.

```
 1: /* sender node */
 2: while(1)
 3: {...
 4:  if(AckReceived && !MsgSent){
 5:    ...
 6:    lib_radio_send_msg(...);
 7:  }
 8:  else if(!AckReceived && MsgSent){
 9:    lib_radio_receive_timed(...);
10:    // Bug #1
11:    PacketID=256*Buffer[1]+Buffer[0];
12:  }
13: }
```

Listing 1: Bug 1 source code

PST depth	False Alarms	False Positive Rate
0	8	10.26%
1	2	2.56%

TABLE I: Bug 1 trace analyzing results

Initially, the application with the bug seems to be running normally without obvious failures. A total of 500 sequences were collected during runtime. The vPST-SVM analyzer identifies two distinct classes in the trace. One class with repeated patterns, which is "1 2 6 1 3 5 6", starts from the 79th subsequence in the trace. This pattern repeats after the application has been running for a while, therefore, it worths further investigation. After further checking this pattern, we find that when this sequence occurs, the sender just repeatedly re-sends the same packet forever. On the receiver, the same packet is always dropped. Obviously such behavior corresponds to a bug. By checking the source code that generates this corresponding pattern, we finally confirm and locate the bug within the pattern "1 3 5 6", which is caused by a variable overflow. The simplified source code is show in Listing 1. In the code, both `PacketID` and `Buffer[]` are defined as 16-bit arrays. But the number "256" is by default eight bits. Therefore, in calculating the expression `256*Buffer[1]` with a 16-bit variable and an 8-bit number mixed, when `Buffer[1]` becomes 3 or larger, an overflow will occur. As a result, `PacketID` will never exceed 512. Therefore, the receiver will always consider the packet it has received to be very old, and keeps dropping it.

```
 1: /* Thread 1 */
 2: /* receiver node */
 3: while(1)
 4: { lib_sleep_thread(1000);
 5:   if(AckSent && !MsgReceived){
 6:     lib_radio_received_timed(...);
 7:     ...
 8:     fromnodeid = incomingMsg[2];
 9:     if(thisPacketID >= lastPacketID){
10:       lastPacketID = thisPacketID;
11:       AckSent = false;
12:       MsgReceived = true;
13:     }
14:   }
15: }
16: ...
17: /* Thread 2 */
18: /* receiver node */
19: while(1)
20: { lib_sleep_thread(1000);
21:   if(MsgReceived && !AckSent){
22:     ...
23:     // Bug #2.
24:     AckSent = true;
25:     MsgReceived = false;
26:     ...
27:     lib_radio_send_msg(...);
28:   }
29: }
```

Listing 2: Bug 2 source code

For this simple bug, a single iteration of vPST-SVM can differentiate buggy patterns very well. The classification results are shown in Table I. The false positive rate decreases from 10.26% to 2.56% when the vPST depth is increased from 0 to 1. There are 8 false alarms in the total of 500 sequences classified by the vPST-SVM debugging system when vPST depth is set to 0, in contrast to only 2 false alarms when vPST goes just one level deeper. The results clearly demonstrate that significant improvements can be achieved by using deeper vPST in the analysis.

Sequence	Anomaly Score	Bug/Anomaly
3033	-4.5467	Anomaly
3070	-4.5467	Anomaly
642	-1.073	Bug
3372	-1.073	Bug
1112	-1.0217	Bug
3144	-1.0217	Bug
380	-1.0002	Bug
...	...	...

TABLE II: Bug 2 trace analyzing results

B. Test Case II: a race condition bug in a LiteOS application

In this test case, a WSN bug with a root cause of race condition is analyzed by the vPST-SVM debugging system. Here, we implement a similar functionality as the application in test case I, but in a multi-threaded mode. There are two application threads on each sensor node, one for sending packets, the other for receiving packets. These two threads share four global state variables as shown in the source code in Listing 2, `MsgSent`, `MsgReceived`, `AckSent`, and `AckReceived`. The application is deployed on 5 different micaZ nodes, including 1 receiver and 4 senders. The 4 senders keep sending messages to the receiver, which in turn sends back ACKs to corresponding senders after it receives a message.

In the actual execution, two threads are running in an interleaved manner. A failing scenario of this bug is as follows. First, thread B changes the variables `AckSent` and `MsgReceived` at line 24 and 25, but before it sends out `AckMsg`, the thread yields CPU to thread A. When thread A receives a new message from a different sender node and updates the variable `fromnodeid` on line 8. When the CPU is returned to thread B, an ACK packet will be sent to a wrong destination because `fromnodeid` has been modified. Since such an occasion is rare, those buggy subsequences will be sparsely distributed in a majority of normal subsequences.

To determine whether the vPST-SVM system is effective to debug this problem, we insert 15 trace points into the source code. A total of 3400 subsequences are collected and analyzed. A single vPST-SVM iteration is conducted with several vPST depth adjustments. The anomaly scores are calculated, and the results are ordered with a vPST depth of 5, as shown in Table II. The lower the anomaly score is, the more likely the corresponding subsequence contains bugs. Out of the total 3400 subsequences, there are 5 confirmed bugs within the top 7 anomalies. The first 2 subsequences, 3033 and 3070, are in fact not caused by bugs but are only anomalous sequences that contain the shortest sequence of a thread A cycle and a thread B cycle.

C. Test Case III: a race condition bug in a TinyOS application

In the third test case, we develop an application that is based on the TinyOS 2.x operating system using the TestFtsp application. The outline of the code is shown in Listing 3, where data are periodically collected using the two arrays `tracedata[]` and `tmpdata[]`. A task is posted whenever the `tracedata[]` becomes full.

vPST depth	0	1	2	3	4	5
Iteration	Bugs Detected					
First	0	0	0	2	2	1
Second	0	1	2	2	3	5
Third	0	4	3	3	5	5

TABLE III: Bug 3 trace analyzing results: Number of bugs identified in 10 top anomalies at different iterations with different vPST depths

```
 1:   task void collecttrace(){
 2:     sendtrace(tracedata);
 3:     printf("Tracepoint reached ID 23\n");
 4:     ...
 5:   }
 6:   ...
 7:   somefunction(){
 8:     tmpdata[] = ...;
 9:     if(tmpdata is ready){
10:       tracedata[]=tmpdata[];
11:       post collecttrace();
12:     }
13:   }
```

Listing 3: Bug 3 outline code

However, a tricky race condition bug may occur in this seemingly simple code. To detect it, we apply the vPST-SVM method, and collect the runtime data traces. The process is tricky because a single iteration of vPST-SVM can hardly detect and confirm the bug. Iterative analysis is then conducted to identify instances of triggered bugs. We start with a collected runtime trace of 2000 total subsequences. During the first iteration, 6 experiments based on vPST-SVM are conducted with the vPST depth changing from 0 to 5. Very few bugs are identified even when the vPST depth is set to 5. Then we go to second iteration. At first we delete top 10 outliers that were identified as non-buggy subsequences according to the results from the first iteration with the vPST depth as 1. Then another 6 vPST-SVM experiments are conducted with the vPST depth varying from 0 to 5. Finally, a third iteration is performed based on a total of 1980 sequences from the second iteration with another 10 non-buggy outliers deleted. As shown in the Table 3, the deeper vPST is reached, the more likely bugs will be identified in the top 10 anomalies. It clearly shows that a deeper vPST can improve debugging performance of the vPST-SVM debugging system. Also it demonstrates the system can improve quickly by learning from the developer's input, even when only a little feedback is provided to the system.

VI. Conclusions

In this paper, we proposed, implemented and evaluated a novel vPST-SVM WSN debugging system with case studies. By applying the debugging system iteratively on collected program runtime traces, hidden bugs in WSN applications can be revealed and identified efficiently. The case studies demonstrated that the vPST-SVM debugging system is a robust, generic and flexible debugging system that can efficiently learn and improve by taking feedback from the user. In general our extension of PST to vPST shed insightful light on future research on sequential data, and our proposed vPST-SVM debugging system is demonstrated to be a flexible and iterative debugger for WSN applications that can detect transient bugs very efficiently.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. The work reported in this paper was supported in part by the National Science Foundation grant CNS-0953238 and CNS-1117384.

References

[1] A. Kulakov and D. Davcev, "Tracking of unusual events in wireless sensor networks based on artificial neural-networks algorithms," in *International Conference on Information Technology: Coding and Computing, ITCC*, vol. 2. IEEE, Apr. 2005, pp. 534– 539.

[2] J. Kwon, C. Chen, and P. Varaiya, "Statistical methods for detecting spatial configuration errors in traffic surveillance sensors," *Transportation Research Record*, vol. 1870, no. 1, pp. 124–132, Jan. 2004.

[3] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture," in *20th International Parallel and Distributed Processing Symposium, IPDPS*. IEEE, Apr. 2006.

[4] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI. Berkeley, CA, USA: USENIX, 2006, pp. 381–396.

[5] N. Ramanathan, E. Kohler, and D. Estrin, "Towards a debugging system for sensor networks," *International Journal of Network Management*, vol. 15, no. 4, pp. 223–234, Jul. 2005.

[6] D. Ron, Y. Singer, and N. TIshby, "The power of amnesia: Learning probabilistic automata with variable memory length," *Machine Learning*, vol. 25, pp. 117–149, 1996.

[7] A. Aizerman, E. M. Braverman, and L. I. Rozoner, "Theoretical foundations of the potential function method in pattern recognition learning," *Automation and Remote Control*, vol. 25, pp. 821–837, 1964.

[8] P. Li and J. Regehr, "T-check: bug finding for sensor networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ser. IPSN. New York, NY, USA: ACM, 2010, pp. 174–185.

[9] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks - IPSN '10*, Stockholm, Sweden, 2010, p. 186.

[10] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "Sentomist: Unveiling transient sensor network bugs via symptom mining," in *IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Jun. 2010, pp. 784–794.

[11] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys. New York, NY, USA: ACM, 2008, pp. 99–112.

[12] P. Sun, S. Chawla, B. Arunasalam, J. Ghosh, D. Lambert, D. Skillicorn, and J. Srivastava, "Mining for outliers in sequential databases." in *SDM*. SIAM, 2006.

[13] G. Mazeroff, J. Gregor, M. Thomason, and R. Ford, "Probabilistic suffix models for API sequence analysis of windows XP applications," *Pattern Recogn.*, vol. 41, no. 1, pp. 90–101, Jan. 2008.

[14] S. S. Keerthi, O. Chapelle, and D. DeCoste, "Building support vector machines with reduced classifier complexity," *JOURNAL OF MACHINE LEARNING RESEARCH*, vol. 7, pp. 1493–1515, 2006.

[15] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.