University of Nevada, Reno

SYCOPHANT
A Context Based Generalized User Modeling Framework for Desktop Applications

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in
Computer Science and Engineering

by

Anil Shankar

Dr.Sushil J. Louis/Dissertation Advisor

August, 2008

UNIVERSITY
OF NEVADA
RENO

THE GRADUATE SCHOOL

We recommend that the dissertation
prepared under our supervision by

ANIL SHANKAR

entitled

SYCOPHANT
A Context Based Generalized User Modeling Framework for Desktop Applications

be accepted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

Sushil J. Louis, Ph.D., Advisor

Sergiu M. Dascalu, Ph.D., Committee Member

Ramona Houmanfar, Ph.D., Committee Member

Monica N. Nicolescu, Ph.D., Committee Member

Anna K. Panorska, Ph.D., Graduate School Representative

Marsha H. Read, Ph.D., Associate Dean, Graduate School

August, 2008

~ Dedication ~

I lovingly dedicate this dissertation and the work that went into to my parents – **Rathna Shankar** and **Bhawrilal Shankar** – whose sacrifice and hard work has given me a better life.

Abstract

While current desktop applications provide input for an application's information processing by monitoring peripheral activity (keyboard, mouse, internal clock), these applications do not access external information from a user's environment to better adapt their behavior towards individual users. To address this lack of personalization, I design *Sycophant*, my context-aware user modeling framework. Sycophant leverages user-related information from a desktop's environment, *user-context*, to learn a user's preferences for different application actions. Results from four real-world user studies show that Sycophant can successfully predict a user-preferred action for a media player and a calendar. In the first pilot study, Sycophant establishes the feasibility of a context based approach to predict user-preferred calendar alarm types. The second study with Google Calendar further extends this alarm type personalization for multiple participants thereby validating my framework's generalizability across participants. Consistent with the second study's results, findings from the third study confirm that Sycophant successfully predicts a participant's preference for Winamp, a media player. Sycophant's Winamp personalization validates the generalizability of my framework across multiple applications. The fourth study examines and verifies Sycophant's user-preferred calendar alarm type prediction accuracy over long-term application use. In addition to highlighting Sycophant's generalizability across participants and applications, these user studies also show that removing user-context features significantly degrades the action prediction accuracy of various machine learning algorithms. This performance degradation emphasizes the need for context based approaches to personalize desktop applications and improve human-computer interaction.

Acknowldgements

I thank my advisor Dr. Sushil J. Louis for his invaluable guidance, friendship, and financial support over the last five years. Under his supervision I relearned how to read, talk, and write responsibly about research. From day one, Sushil gave me the best writing advice to tell a *story*, and to "Introduce something, explain it and summarize it again in the end". I am also grateful to him for forgiving many unproductive weeks when I had nothing to show for progress.

Over the course of this dissertation work, I was lucky to develop a close rapport with Dr. Sergiu Dascalu and Dr. Ramona Houmanfar as my committee members. I learned a lot and enjoyed having discussions with Sergiu about human-computer interaction issues and life in general. Sergiu's expertise, openness, encouragement, and friendship have made it a pleasure to have him on my committee. Ramona and her Behavioral Analysis group asked me tough questions and helped me improve the user studies design.

I greatly appreciate my other two committee members: Dr.Monica Nicolescu and Dr.Anna Panorska. Monica's first class in graduate robotics was one of the most fun classes which I have taken in graduate school. *Sycophant's* design was influenced by some of the principles discussed in her robotics classes. Anna's statistics class helped me analyze and present my results meaningfully. I am indebted to all my committee members for their patience and enthusiasm.

Patricia Spoon at the computer science department office and system administrator, Michael Dahl, made my interaction with paperwork and machines much easier on numerous occasions. I thank them for their support and help. This dissertation's progress hinged on the cooperation of my user studies' participants. I

thank them for their enthusiasm, time, and patience.

Past and present members of the Evolutionary Computing Systems Laboratory (ECSL) have made the years I spent finding my way through graduate school an enjoyable process. I learned useful programming lessons from Chris Miles who also coaxed me into playing many Real-Time Strategy games for *evaluation* purposes only. Ryan Leigh's cheerful demeanor, incessant quips, and appreciation of Guinness draught made it easier to share a corner in the lab or a pub. Shane Warren and Matt Parker as cubicle neighbors were enthusiastic for a random game of ping-pong during the long sluggish unproductive writing hours. Other ECSL members and people who hang around the lab – Mark Hammer, John Kenyon, Nathan Penrod, David Carr, Juan Quiroz, Britta Austin, Justin Schonfeld, and Amit Banerjee – have been good friends and positively affected my time spent at ECSL. I was also fortunate to discover rock climbing after taking Davide Sartoni's class during the Fall of 2006. Sebastian "extreme" Smith has been a great climbing partner and a good friend. The friendly, jovial, and welcoming climbing community at *RockSport* have been a second family to me.

Finally, I thank my family for their love and support. My cousins Ravi Kumar and Harinath encouraged me to seek knowledge outside books and have served as an inspiration to me from a young age. My friends: Abhinandan, Vishwas, Rajesh, Ashwin, Harsha, and Santhosh always supported my efforts and have been a constant source of encouragement; I consider these guys more as family rather than friends, and as per request "a shout-out to my grrrl j-dog, imaginary friends 4-ever". I especially thank Mia Ellis who endured my self-absorbed work habits and who is always close regardless of spatial distance. I don't have to be explicit about acknowledging my parents; my existence and this work would be impossible without them. I dedicate this dissertation to my mother Rathna and my father Bhawrilal.

This work was supported in part by contract No. 00014-0301-0104 from the Office of Naval Research and the National Science Foundation under Grant No. 0447416.

Contents

List of Tables

List of Figures

Chapter 1

Motivation

Today's desktop computer users regularly access a wide variety of interactive user interfaces (applications) such as media players, calendars, e-mail clients, and word-processors. These applications tend to rely on keyboard activity, mouse usage, or the activity of an internal clock to provide input or *context* for their information processing. Relying on sparse contextual information and not sensing whether a user is present (or absent) in her environment, these desktop user interfaces lack external user-related information to better adapt their actions towards individual users.

This chapter starts by describing two scenarios that illustrate how a user's interaction with an application can be improved if that application sensed a user's external environment and automatically adapted its actions according to that user's preferences. Next, I introduce the definition of user-context and state the central claim of this dissertation. After this, I give a brief overview of **Sycophant**, my generalized context-aware user modeling framework, that harnesses external information from a user's environment to enable desktop applications for better predicting a user's preferences. Finally, I summarize the main contributions of my research relevant to user-modeling and adaptive user interfaces and conclude the chapter by providing a detailed roadmap of this dissertation's organization.

1.1 User-Context and This Dissertation's Claim

In this section, I first examine an application's twin goals of effectiveness and utility to improve user-experience and emphasize the potential of improving user-experience by adapting an application's behavior to an individual user, Next, I define user-context, and then state the central claim of this dissertation.

According to Preece *et al.*, an application is *effective* if it is doing what it is supposed to do and provides *utility* to its users if it provides the right kind of functionality for allowing users to accomplish their tasks [59]. Consider these measures of utility and effectiveness for Jane and Jack with respect to their media player in the following two scenarios.

Scenario 1: If Jane prefers to turn her media player *volume down* when she is talking with someone in her office, her media player has no access to the presence (or absence) of speech in Jane's external environment for automatically decreasing the volume. In a different context, if Jane prefers to *pause* her music when she leaves her desk, her media player is again unable to detect Jane's absence to pause the music. Clearly, Jane's preferences for her media player changes depending upon whether she is talking with someone in her office or when she is not around her desk.

Scenario 2: Consider another user Jack who prefers to *pause* his media player while chatting with someone in his office; he prefers to *decrease* his media player's volume if he had to leave his desk. Like Jane, Jack's media player preferences depend upon whether he is absent at his desk and whether he is talking with someone.

These two example scenarios show that both Jane and Jack's media player preferences are context dependent. Note the following about user preferences from these two scenarios:

1. A user's preference for an application action (changing volume, pausing music) depends on the context of use (talking with someone or leaving the desktop area).

2. Application action preferences vary from user to user in the same context of use; Jane and Jack have different preferences for the same media player when talking with someone in their office or when they leave their desk.

If an application harnessed information from a user's external environment, it could learn to better predict their preferred actions. To be *effective*, Jane's (or Jack's) media player would play music according to her preferences by contextually pausing or decreasing the volume. This media player's adaptive behavior would also result in providing the right kind of functionality, that is, better *utility* to Jane (or Jack) for accomplishing her tasks since her media player successfully predicts her preferred actions.

Sycophant, my generalized user modeling framework, is based on these two goals of effectiveness and utility; Sycophant harnesseses contextual information from both the internal and external environment of a user. Extensive work has been done by researchers in ubiquitous computing to standardize on a clear definition of *context* [17, 18, 23, 47, 52]. Dey gives one of the widely accepted definitions of context and defines *context* as [24]:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.

I extend Dey's broad definition of context to make it applicable to desktop applications, and define *user-context* as:

> Any user-related contextual information in the vicinity of a desktop computer.

In my research, I consider any information that Sycophant senses surrounding a desktop's external environment as *external* user-context. This dissertation considers information related to the presence (or absence) of speech and movement (motion information) in a desktop's vicinity as external user-context. In contrast to external user-context, *internal* user-context is any information that Sycophant senses

from a desktop's internal environment. In this work, I consider keyboard activity and mouse usage as internal user-context.

Based on my definition of user-context, the central claim of this dissertation is:

> **External user-context enables applications to predict user preferred actions thereby leading to better personalization of applications to individual users.**

Structured around this hypothesis, Sycophant employs different sensors to gather internal and external environmental information from a user's desktop environment. A web-camera checks for movement in a user's environment (motion) and a microphone monitors the user's environment for the presence or absence of speech. In addition to sensing a user's external environment my system also monitors keyboard and mouse activity. Sycophant aggregates both the external and internal contextual data from these four sensors and processes the sensors' information to extract user-related contextual features. Sections 2.1 and 2.2 detail Sycophant's four layers, their functionality and describe the system's operation.

For learning user preferences, machine learning algorithms within Sycophant, map these context features to user-preferred actions and generate a preference model for that particular user. An application can then leverage this learned preference model to predict user-preferred actions. I briefly explain how a program learns user preferred actions in Chapter 4. This chapter describes my approach for using a learning classifier system and a decision tree learner to personalize applications towards individual users. In my research, an application, say Jane's media player, *personalizes* itself to Jane if it can contextually predict her preferred actions.

To remedy the lack of personalization in current desktop applications, I first demonstrate that my user-context based approach enables Sycophant to better predict a user-preferred application action. Second, I show that external user-context increases various machine learning algorithms' application action prediction accuracy.

Third, I highlight the feasibility of XCS, a genetics-based machine learning approach, to predict user-preferences for desktop application actions. Fourth, I

present results from a pilot study and three real-world user studies that highlight the generalizability of my novel approach to learn application action preferences across multiple participants.

Fifth, I context-enable a media player in addition to a calendar to validate Sycophant's flexible and modular design and thereby demonstrate that my user-modeling framework supports multiple desktop applications. This dissertation's final contribution is verifying that, in addition to predicting user-preferred application actions in short-term user studies, my user-context based application action prediction is successful over long-term use for multiple participants. I next provide a a brief outline of this dissertation and conclude this chapter.

1.2 Dissertation Outline

Chapter 2 describes Sycophant's four layer architecture in detail including examples of my User-Context Application Programming Interface (C-API) to access different services at these layers. Sycophant's novel API was published in the proceedings of the *2007 IEEE International Conference on Software Engineering Advances* [58].

Chapter 3 surveys related work in adaptive user interfaces research and applications of learning classifier systems. Chapter 4 gives an overview of machine learning and the *XCS* Classifier System, a genetics-based machine learning approach, that Sycophant employs to predict user preferred actions.

Chapter 5 gives the results from my first user study. In this pilot study, I investigated the feasibility of learning a user's calendar alarm preferences based on user-context. Results from this study were published in the proceedings of *2004 IEEE International Conference on Information Reuse and Integration*, *2005 IEEE Congress on Evolutionary Computation*, and the *2005 Indian International Conference on Artificial Intelligence* [44, 54, 55]

In Chapter 6, I give results from my second user study, a short-term study with *Google Calendar*. I evaluated whether user-context based calendar alarm preference learning generalized across multiple participants. This study's results were

published in the proceedings of *2007 International Conference on Intelligent User Interfaces* and *2007 Genetic and Evolutionary Computation Conference* [56, 57] .

Chapter 7 gives results from my third user study, another short-term study with *Winamp*, a media player. This user study demonstrated that Sycophant context-enabled Winamp and successfully learned participant preferences to validate the generalizability of my framework across multiple desktop applications. This work has been submitted to the *IEEE Transactions on Evolutionary Computation*.

Chapter 8 gives results from learning long-term Google Calendar alarm preferences. The findings of this study verified that my user-context based approach to predict application action preferences is successful over long-term calendar use for multiple users. I conclude with a summary of my user-context based approach to personalize applications and outline future research directions in the final chapter.

Chapter 2

The Sycophant User-Modeling Framework

This chapter describes the architecture and the Application Programming Interface (API) of my context-aware user modeling framework, Sycophant. Sycophant's architecture comprises four layers. From the bottom-up, they are: Sensors, , Learning Services, and Application. To access different services provided by these layers, I designed a User-context API (C-API) to allow easy insertion of new context features and to provide a readily available, reusable programming resource for developing new context-aware software applications. Sycophant along with C-API is available for use based on the Open Standards software requirements license at `http://www.cse.unr.edu/~syco` [3, 5]. The next three sections present Sycophant's high level architecture, functional capabilities, and detail C-API's design along with examples of its use.

2.1 Sycophant's Architecture

Figure 2.1 shows Sycophant's four layer architecture. User-context sensors in the sensors layer gather information from a user's environment and store this information in the context layer. User-context features extracted from the sensor data is stored in the context layer. For example, Sycophant checks whether or not a sensor

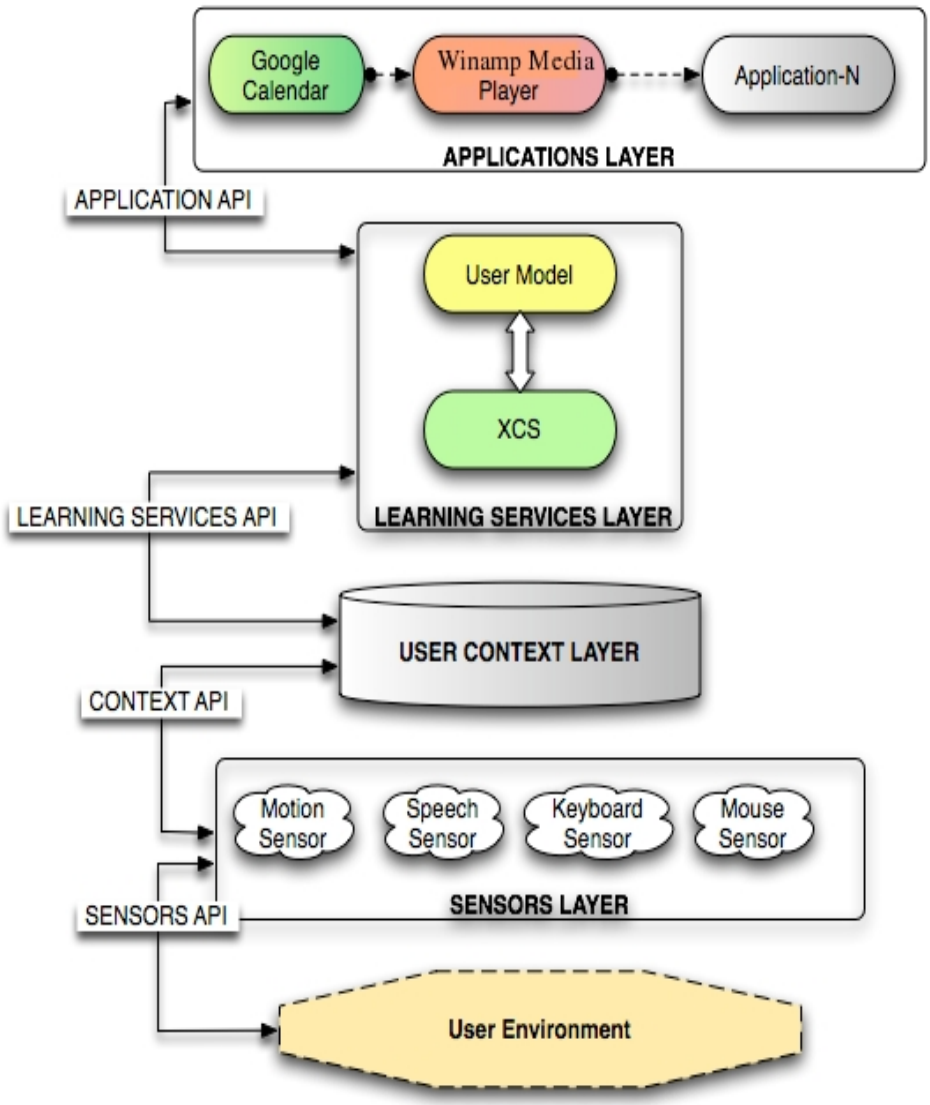

Figure 2.1: Sycophant's four layer architecture. The figure presents Sycophant's layers along with the APIs used to access different services in the layers.

was active in the last minute and labels this as the *Any1* feature. Section 2.1.2 gives additional details about the user-context features that Sycophant extracts from different sensors. The learning services API is used to format user-context features into a suitable format for machine learning algorithms in the learning services layer. Then, a selected machine learner at the learning services layer can create an application-specific user model that predicts a user's preferences. Multiple desktop applications in the application layer harness this user-model generated in the learning services layer to predict user-preferred actions. I next describe these layers in detail.

2.1.1 Sensors Layer

A clean, well-defined sensors API (Section 2.3) provides methods (procedures) to extract raw sensor information from a user's environment and store this information up at the context layer. Figure 2.2 shows the sensor and context layers. Sycophant currently uses four sensors: a motion sensor, a speech sensor, a keyboard sensor, and a mouse sensor. However, the sensors API was designed to be extensible for allowing easy insertion of new sensors. I give an example of creating a motion sensor, associating a log file with that sensor, and activating (or deactivating) it in Section 2.3. Sycophant's second layer, the context layer, aggregates the data assembled from different sensors for use by the learning services layer.

2.1.2 Context Layer

The context API gives access to methods for extracting user-context features from the information aggregated by the four sensors in the sensors layer. For example, I use the *checkAny1* service provided in this layer to examine a motion log file and examine if the motion sensor was active in the last minute. At this second layer, Sycophant stores a number of user-related contextual features for each of the four sensors in the sensors layer as shown in Figure 2.2.

I extract the following five user-related contextual features based on user studies conducted by Fogarty *et al.* and myself [29, 53, 56, 58]:

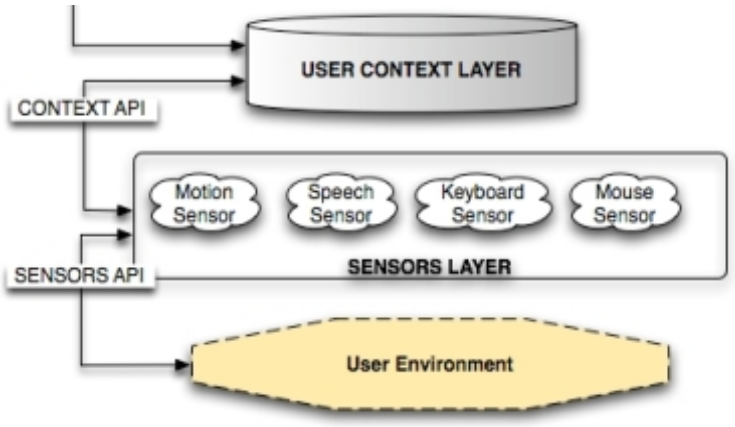

Figure 2.2: Sycophant's sensor and context layers

1. **Count**: checks the number of times a sensor was active in the last five minutes. The system polls a sensor every 15 seconds, and therefore *Count* can have a maximum value of 20 for a five minute history.

2. **All5**: checks if a sensor was active in all the 15 second intervals in the last five minutes.

3. **Any5**: checks if a sensor was active in the last five minutes when the sensor was polled every 15 seconds.

4. **All1**: this feature is similar to *All5* with the sensor history checked only for the last minute.

5. **Any1**: this is similar to *Any5*, except that the sensor activity history is checked only in the last minute.

Table 2.1 shows Sycophant's four user-context sensors' sample data values for each of the five user-context features. Note that Count has a value of 15 for the motion sensor thereby showing that this sensor was active 15 (out of 20) times in the last five minutes. The motion sensor's Any5 value is 1 indicating the sensor was active in the last five minutes when periodically polled. The same sensor's All5 value is 0 showing that it was not active during all of the 20 fifteen second intervals in the last five minutes. Similarly, Any1 is 1 for the motion sensor indicating

Table 2.1: Sycophant's user-context data example.

User-context Feature	Sample Data Value(s)
User-identifier	participant-2
Motion (Count, All5, Any5, All1, Any1)	15, 0, 1, 0, 1
Speech (Count, All5, Any5, All1, Any1)	3, 0, 0, 0, 1
Keyboard (Count, All5, Any5, All1, Any1)	0, 0, 0, 0,0
Mouse (Count, All5, Any5, All1, Any1)	0, 0, 0, 0, 0

that this sensor was active in the last minute, and All1 is 0 showing that the motion sensor was inactive during all of the four 15 second intervals during the same period. You can similarly interpret Table 2.1 for the speech sensor and note from Count that this sensor was active $3/20$ times in the last five minutes while being active in the last minute (Any1 $= 1$). User-context sensor values for keyboard and mouse are 0 indicating that these sensors were inactive while Sycophant gathered this row of sensor values (*exemplar*). This exemplar shows that there was some motion and speech activity but no keyboard or mouse activity. Machine learning algorithms can then use these user-context features to generate an application specific user-preference model at the learning services layer.

2.1.3 Learning Services Layer

Sycophant supports a learning classifier system, XCS, and a set of other machine learning algorithms provided by *Weka's* machine learning toolkit [65, 67]. Weka is an open source collection of machine learning algorithms for data mining tasks. I experimented with four Weka learners: a support vector machine, a decision-tree, a OneR classifier, and a Naive Bayes classifier. Chapter 5 briefly describes these techniques and explains my choice to use a decision-tree to predict user preferred application actions.

Figure 2.3 presents the services provided at this layer that are used in preprocessing user-related sensor features from the context layer into an appropriate data format for a machine learning algorithm. Using these services, Sycophant can

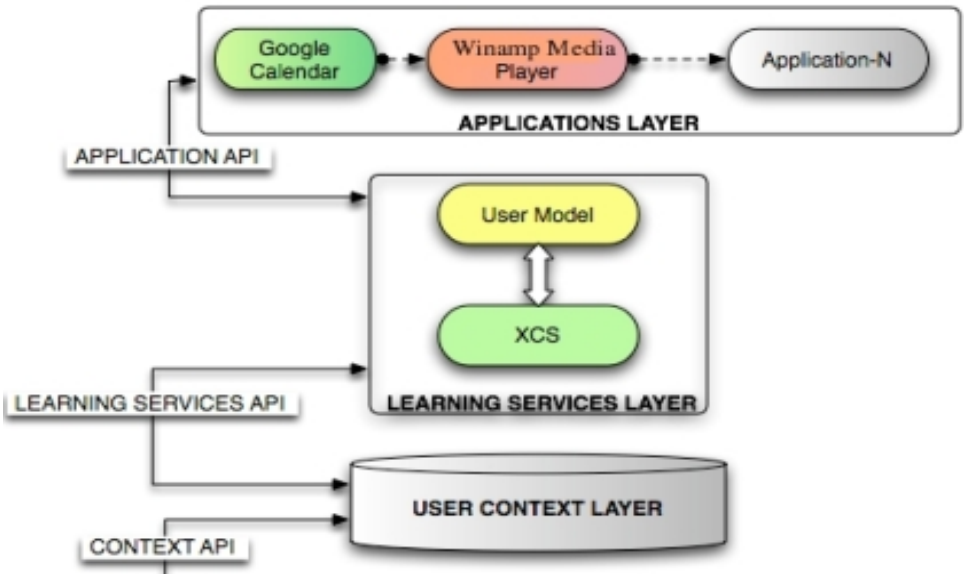

Figure 2.3: Sycophant's learning services and application layers

select any of the machine learners including XCS in this layer and generate an application-specific user model that predicts user-preferred actions. As an example, first I use the *buildUserModel* service (in Listing 2.4) to select XCS for learning a user's preference for Google Calendar alarm types. At the application layer above, Google Calendar harnesses this XCS generated user model to predict calendar alarm preferences for a user when new context data becomes available during calendar use at the context layer.

2.1.4 Application Services Layer

Figure 2.3 shows the services at this top most layer. The application layer provides services to plugin multiple desktop applications and context-enable their interface actions. For this work, I plugged in Google Calendar and Winamp. Google Calendar harnesses a learned user preference model generated in the learning services layer to learn alarm type preferences for that particular user. Similarly, Winamp accesses its own learned preference model for an individual user to predict one of its interface actions. The system design thus separates an application from the

user-context based preference model that predicts user-preferred actions. This separation makes Sycophant's framework flexible and modular. The next section explains Sycophant's functionality and system operation.

2.2 Sycophant's System Operation

Figure 2.4 presents a use-case diagram showing the major actors interacting with Sycophant. This use-case diagram explains system functionality. The actors and use-cases are elements of the Unified Modeling Language that captures system behavior as seen from outside the system [7, 50]. All use cases are triggered by actors, and at this level of representation the system can be seen as a black box, that is, the way use cases are implemented within the system is irrelevant to the actors [10].

The main actors include the user of the environment embedding a target application such as a calendar or a media player, user-context sensors for collecting data relevant to user behavior (motion, keyboard, mouse, and speech sensors), and the time which provides time-stamps for analyzing stored context data. Note that in Figure 2.4 all sensor actors inherit from an abstract actor, denoted *Sensor*. The *GenerateTimeStamp* use case indicates that the *Time* actor interacts with the system by associating timestamps with data items collected from sensors. The sensors notify the system of any changes. For example, when a web-camera detects motion in the vicinity of the computer (captured in the *NotifyChange* use case) it logs the current time stamp to a log file. When requested (by the *User*) the sensors also provide sensor data, indicated by their involvement in the *GetSensorData* use case. The *User* of the system initiates most use cases. These are as follows:

- *DefineServiceSet* allows the *User* to specify the types and the number of sensors available in the system.

- *CustomizeServiceSet* selects a subset of available sensors to be used in system operation. For example, I invoke this service to select the user-context sensors

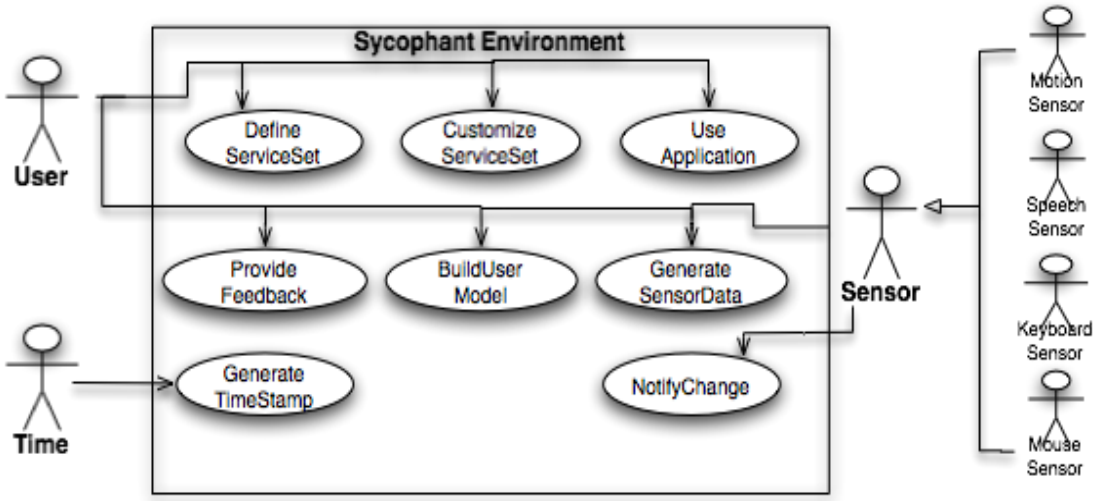

Figure 2.4: Sycophant's use-case diagram showing the major actors interacting with the system.

in a study designed to predict calendar alarm-type preferences for different participants.

- *GetSensorData* allows the *User* to specify the parameters of data collection, including a sensor's sampling interval value.
- *BuildUserModel* generates a user model based on context data collected.
- *UseApplication* is the actual use of the target application embedded within Sycophant. I use this service to context-enable a calendar and predict user-preferred alarm types.
- *ProvideFeedback* solicits feedback from the user during the use of the application on various aspects of use that help Sycophant learn user preferences. The next section presents the user interface used for requesting user feedback.

Figure 2.5 depicts Sycophant's high-level operation. Sycophant's four user-context sensors include a web-camera, a microphone, a keyboard, and a mouse. Starting at the top, context sensors monitor a user's desktop environment to col-

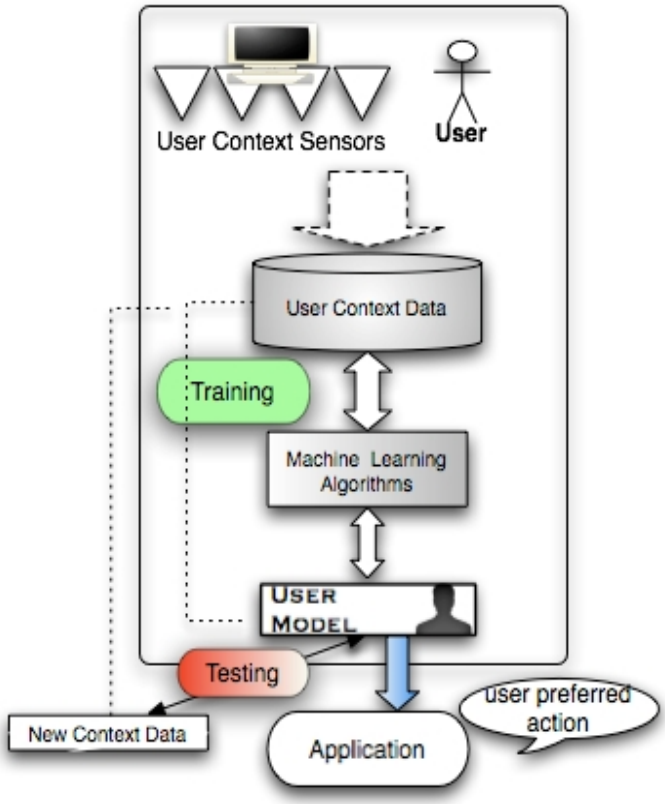

Figure 2.5: Sycophant's operation. This figure presents the high-level system operation

lect information related to movement, presence or absence of speech, keyboard usage, and mouse activity. A web-camera detects user-movement and logs this *motion* information at the sensors layer. Similarly, a microphone detects the presence or absence of speech and logs that information. Sycophant also checks for keyboard and mouse usage and logs those sensors' information as well.

The context sensors operate in a binary mode; that is, if the web-camera detects motion it logs a 1 into its log file and 0 otherwise. This work considers these sensors' user-related information as **user-context**. I explained earlier in Section 2.1.2 the procedure to extract Any1, All1, Any5, All5, and Count features for each sensor from the raw sensor data. Sycophant stores this preprocessed sensor information in an appropriate format as user-context data at the context layer.

Next, a machine learning algorithm at the learning services layer maps these user-context features to user preferred actions and generates an application specific preference model for that user. The learning algorithm is *trained* on user-context data for generating a user model. Chapter 4 explains how a program *learns* from data, the learning algorithms which my system uses for predicting user-preferred actions, and Sycophant's training (or testing) on this user preference learning task.

I envision two learning modes for Sycophant: the *training* mode and the *running* mode. In the training mode, during the initial use of an application embedded within Sycophant no training-data is available to the system. To provide training data for a machine learner, when Sycophant initiates an application action, it selects a random application action and also generates a user feedback request. Figure 2.6 shows the user feedback interface for a calendar alarm. A user picks one

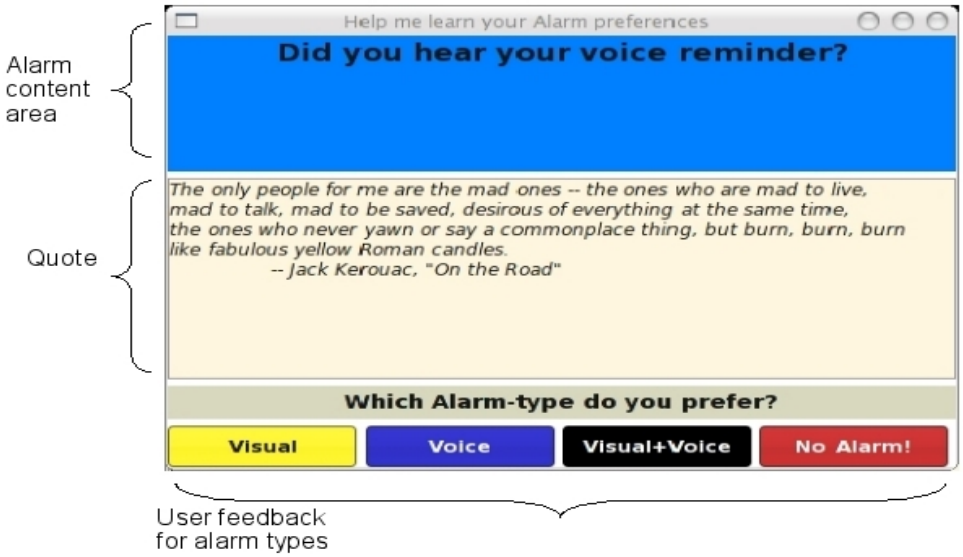

Figure 2.6: A calendar voice alarm. The figure shows the feedback interface highlighting the alarm content area, a quote displayed as incentive for a user's feedback, and the feedback buttons for different alarm types

of the interface actions in the feedback request window and Sycophant stores this user-feedback as their preference. To mitigate the effect of annoying a user with feedback requests the feedback window automatically disappears after 15 seconds.

A user's feedback along with the user-context data is stored as a training data exemplar in the context layer. Sycophant iteratively updates a user's learned model whenever the system gets new training exemplars.

In the running mode, I expect Sycophant to progressively generate fewer user feedback requests eventually relying on no user feedback but only on the generated user model to adapt its behavior. This adaptation could be contingent upon a two factors. The first factor could be the system achieving high predictive accuracy. The second, when a user decides that Sycophant predicts an application's to her satisfaction and thereby obviating any need for further training. I consider the running mode as a promising avenue for future work.

For research purposes, I also operate Sycophant in a *testing* mode to compare its performance to the actual use of the system. During testing, that is, when new user-context data is available to the system, Sycophant leverages the learned user model (generated during training) to predict a user-preferred action. The system adds this fresh test data along with the solicited user feedback as a new training exemplar to existing data at the user-context layer. Thus Sycophant iteratively accumulates training data whenever it predicts an interface action. The next section explains how to use different components of the User-Context API for context-enabling Google Calendar.

2.3 The User-Context API (C-API)

I present the C-API's sensor, context, learning, and application components using class diagrams. With a step-wise procedure, I next show how to use these C-API components for context-enabling Google Calendar. The set-up steps that I follow are general and you can use a similar procedure to context-enable other applications.

2.3.1 Sensors API

Figure 2.7 shows the class diagram of Sycophant's sensor API component. The

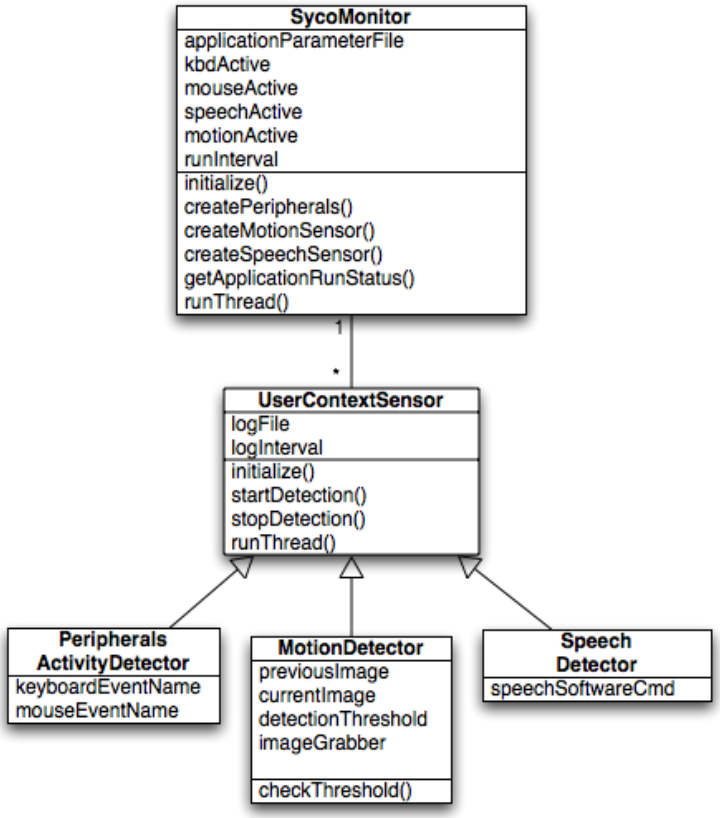

Figure 2.7: Sensors class diagram

SycoMonitor class contains and manages multiple instances of user-context sensors. SycoMonitor's attributes reflect the status of different user-context sensors. For example, *motionActive* checks if the motion detection sensor is active. SycoMonitor uses similar status flags for keyboard, mouse and speech sensors. The attribute *runInterval* specifies the frequency of polling context sensors for raw data. In SycoMonitor, the *createPeripherals* method initializes the keyboard and mouse; *createMotionSensor* and *createSpeechSensor* initialize motion and speech sensors, respectively. All these three methods create instances of the *UserContextSensor* class.

The *UserContextSensor* has these attributes: *logFile* to log a sensor's data and *logInterval* to specify the frequency of logging sensor data. A sensor uses the

startDetection and *stopDetection* methods to start and stop activity detection, respectively. The *runThread* method starts a thread that continuously tracks sensor activity. Three sensor specific classes are derived from *UserContextSensor*. The *PeripheralsActivityDetector* class manages keyboard and mouse sensors. Next, the *MotionDetector* class manages motion detection. This class has attributes to store the previous image (*previousImage*), the current image (*currentImage*), the minimum allowed threshold for the difference between the two images, and the program to use for grabbing images from a web-camera (*imageGrabber*). Lastly, the *SpeechDetector* class manages speech activity detection; its *speechSoftwareCmd* specifies the speech recognition software to use for detecting speech from a user's environment.

Listing 2.1 shows the steps to to use the Sensors API for setting up a sensor and activating it to log raw data (timestamp value) to a file. Note that the sensor setup described below is the same regardless of the type of target application or the type of sensor involved. Specifically, code excerpts provided below show how to set up a motion sensor. Sycophant uses a similar set up for the peripherals (keyboard and mouse) and the speech sensor. I first create a sensor by specifying its name and its associated log file (line 1), then I activate the sensor by calling the start method on it (line 2).

Listing 2.1: Sensors API: Setting up a motion sensor

```
1 motionSensor = Sensor('motion', motionLogFile)
2 motionSensor.start()
```

The raw sensor data collected by activating this motion sensor is next formatted by the context layer services for use by a machine learning algorithm at the learning services layer.

2.3.2 Context API

Figure 2.8 presents the context and learning services class diagrams showing their relationship to a calendar embedded at the application layer. The *UserContextCreator* class has methods to select sensors (*setSensors*), choose the user-context features to extract (*setFeatures*) from these sensors, and extract context data from the

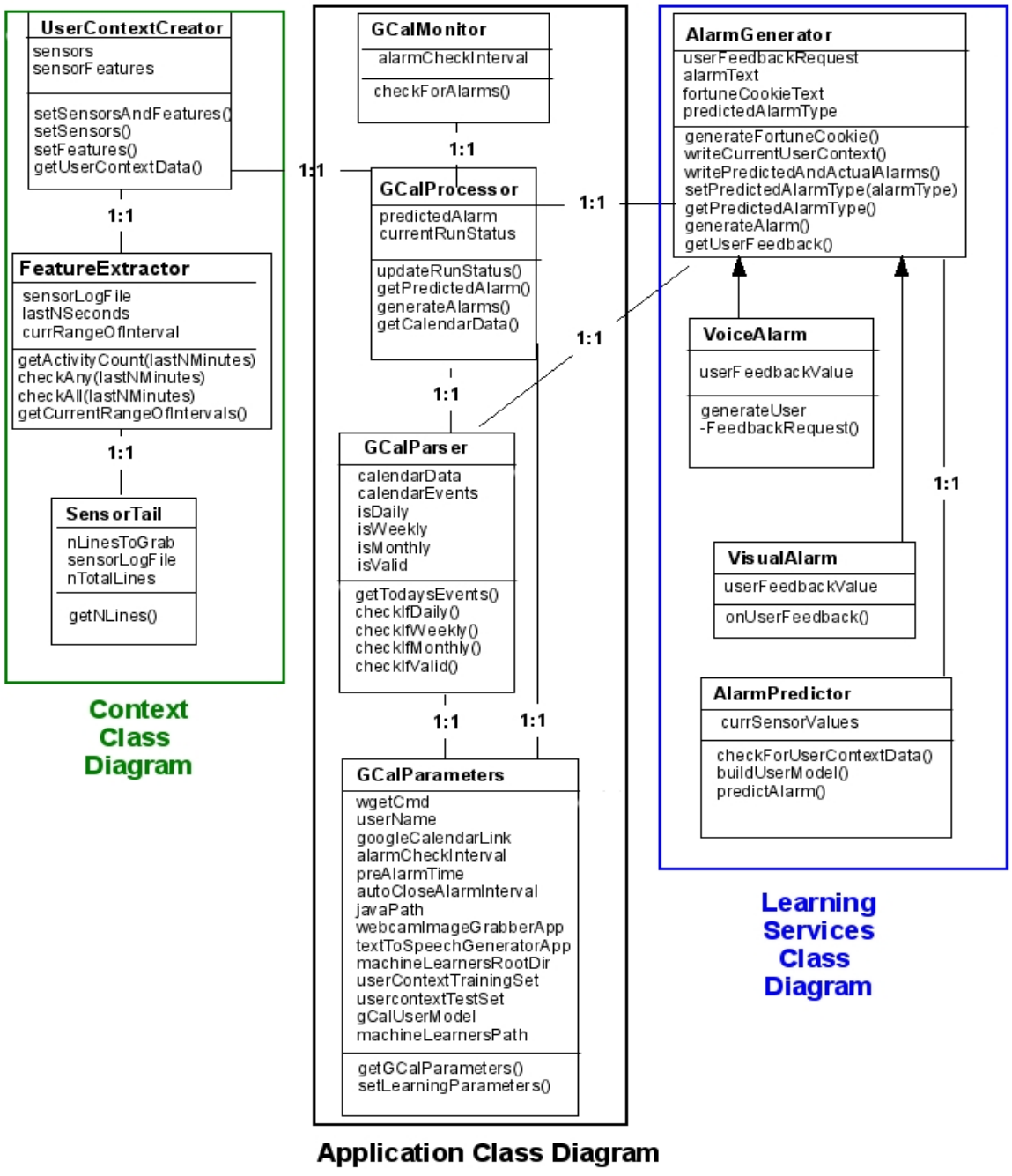

Figure 2.8: Class diagram of an application showing its relationship to the context and learning services classes

raw sensor data (*getUserContextData*). The *FeatureExtractorClass* extracts the context features used by *UserContextCreator* class. This class has methods to compute the sensor-active frequency (*getActivityCount*) and checks if a sensor was active during any or all of those active minutes using the *checkAny* and *checkAll* methods, respectively. The *FeatureExtractor* uses the *SensorTail* class which mimics the *tail* command on a Linux/Unix operating system and obtains the specified number of lines (*nLinesToGrab*) from a sensor's log file *sensorLogFile* with the *getNLines* method. The data in this log file is a time-stamp indicating the activity of a sensor.

Listing 2.2 outlines a procedure to extract user-context features from a sensor by setting up a feature extractor. To extract user-context features from raw sensor data, I first specify the length of a sensor's activity history in the past to examine (line 1), and next choose a feature extractor (line 2). In this case, I examine the motion sensor's activity in the last five minutes. I check if the sensor was active during any of the five minutes or in all the five minutes by specifying the *checkAny* and *checkAll* features (lines 3 and 4). I also check the number of times the motion sensor was active in the same five minute period by specifying the *getCountAll* feature for the motion feature extractor (line 5). Section 2.1.2 gave more details about the meaning of these features. I combine the extracted sensor context information and create user-context data.

Listing 2.2: Context API: Setting up user-context feature extractors

```
1 lastNMinutes = 5 # duration of history check for a sensor
2 motionFeatureExtractor = FeatureExtractor(motionLogFile)
3 checkAnyNMinutes = motionFeatureExtractor.checkAny(lastNMinutes)
4 checkAllNMinutes = motionFeatureExtractor.checkAlllastNMinutes)
5 getCountAllNMinutes = motionFeatureExtractor.getCount(lastNMinutes)
```

Listing 2.3 shows how to create user context data using different features extracted from the motion and speech sensors. This user-context data is formatted according to a machine learner's requirements for generating an application specific user preference model at the learning services layer.

Listing 2.3: Context API: Creating user-context data

```
1 motionFeatures = checkAny-1, checkAll-1, checkAny-5, checkAll-5,
      getCount-5
2 speechFeatures = checkAny-1, checkAll-1, checkAny-5, checkAll-5,
      getCount-5
3 motionUserContextData = UserContextExtractor(motionFeatures)
4 speechUserContextData = UserContextExtractor(speechFeatures)
```

2.3.3 Learning Services API

Figure 2.8 presented earlier showed the class diagram of the Learning Services API component of C-API. Listing 2.4 shows the learning services component of C-API in use. This listing is a step-wise procedure that leverages sensor data to predict a user-preferred alarm type, solicit user feedback, and store the user-context data along with user feedback. I first use the methods from the *AlarmPredictor* class. With the *checkForUserContextData* method, I check if user-context data is available from all the sensors in line 1. In lines 2 and 3, I gather the context information from different user-context sensors (*getUserContextData*) and use this data to generate a user model with *buildUserModel* method. Note that I select a machine learning algorithm for generating a user model with the *learner* parameter of *buildUserModel*. This user model in turn is used to predict an alarm with the *predictAlarm* method that predicts a user's preferred alarm type.

The *AlarmGenerator* class' *generateAlarm* method generates the predicted alarm and Sycophant solicits user feedback using the *FeedbackRequest* method. The predicted alarm, user-feedback and user-context data are stored back at the context layer using the *writeCurrentUserContext* method. At the top, in the application layer Google Calendar adapts its alarm types to an individual user with the services provided at this layer.

Listing 2.4: Learning Services API: Predicting user-preferred alarm types

```
1  dataAvailable = checkForUserContextData() # checks if context data is
       available
2  userContextData = getUserContextData(dataAvailable)
```

```
3 userModel = buildUserModel(learner, userContextData) # generates a user
      model with the specified learning algorithm
4 predictedAlarm = predictAlarm(userModel) # predicts an alarm type based
      on the user-model generated
5 alarmGenerated = generateAlarm() # generate an alarm using the
      predicted alarm type
6 userFeedback = FeedbackRequest() # get user feedback for the predicted
      alarm type
7 writeCurrentUserContext(userContextData, predictedAlarm, userFeedback)
      # log the current user-context data
```

2.3.4 Application API

Figure 2.8 presented earlier showed the class diagram of C-API's Application API component. The *GCalMonitor* class manages alarms for a user's calendar. The method *checkForAlarms* checks for any current or pending appointments every *alarmCheckInterval* seconds. The *GCalProcessor* gathers calendaring data from a user's calendar file (*getCalendarData* method), accesses a user-preferred alarm type (*getPredictedAlarm*) and generates an alarm for a current or pending appointment using the *generateAlarms* method. *GCalParser* parses calendar data from a user's calendar file. The methods *checkIfDaily, checkIfWeekly, checkIfMonthly* check for daily, weekly, and monthly repeating appointments, respectively.

The *AlarmGenerator* class generates different types of alarms and notifications. The *generateFortuneCookie* method generates a fortune cookie (an interesting quote) along with the alarm generated for an appointment using the attribute *fortuneCookieText*. User-context information from the four sensors (sensor features related to motion, speech, keyboard and mouse) are logged using the *writeCurrentUserContext* method. The method *generateAlarm* generates an alarm for a user with the attribute *alarmText*. A user feedback for her preferred alarm type is stored in *userFeedbackRequest* using the method *getUserFeedback*. This class also has methods related to alarm prediction: *getPredictedAlarmType* uses the *predictedAlarmType* to record an alarm type predicted for a user by executing a machine learning algorithm trained on that user's context data, and the method *writePredictedAndActualAlarms* logs the

predicted alarm and the user-preferred alarm type obtained from user feedback to a file .

The classes *VoiceAlarm* and *Visual Alarm* inherit from the *AlarmGenerator* class and generate voice and visual alarms, respectively. The *AlarmPredictor* method gets the current sensor values using *currSensorValues* and checks for existing user context data using *checkForUserContextData* information. Based on the availability of this context information, I can use this class's methods to generate a user model that reflects her preferences for alarm types using the *buildUserModel* method. The method *predictAlarm* accesses a machine-learning algorithm, feeds it with the current sensor values and obtains a predicted alarm type using the generated user model. Most of target application-specific parameters (in this case, Google Calendar related parameters) are set in an *AppParameters* class (*GCalParameters* class).

Some of the application-specific parameters include: *wgetCmd* obtains a user's calendar (*userName*) using the attribute *googleCalendarLink* which specifies the user's calendar web link. Alarms are checked every *alarmCheckInterval* seconds and generated *preAlarmTime* seconds before the actual time of the appointment. Visual alarms and feedback request pop-ups auto-close after *autoCloseAlarmInterval*. The *webcamImageGrabberApp* attribute specifies the application to use for grabbing images from the user's web-camera and the *textToSpeechGeneratorApp* attribute specifies the text to speech generator used for generating voice alarms. The *gCalUserModel* specifies the location of a user model that reflects the user's alarm type preferences.

A user model is generated using the machine learning algorithms in the *machineLearnersRootDir*. The *machineLearnersPath* specifies the machine learning algorithm to use for generating a user model. This algorithm gets trained on *userContextTrainingSet*, which is the training data and an alarm is predicted for the current set of context values stored in the *userContextTestSet*. Methods *setGCalParameters* and *setLearningParameters* set the parameters related to the calendar and the learning algorithms for a user, respectively.

Listing 2.5 shows how to embed one of the target applications, Google Calendar, at the application layer and associate a calendar file (line 1). Figure 2.8

provides additional details about the target application wrapped with my C-API.

Listing 2.5: Application-Specific Use

```
1 calendarMonitor = GCalMonitor(calendarFile)
2 calendarMonitor.generateAlarms()
```

I similarly wrap Winamp, another desktop application using the same procedure listed above.

2.4 Chapter Summary

This chapter described Sycophant's four-layer architecture and presented the sensors and services available at each layer along with the C-API used to access these services. Using the architectural details, I next specified the system functionality with use-cases and described the high level operation of the system for collecting user-context data, generating a user-model, and predicting a user-preferred action. Sycophant's system operation explained how the different layers work to context-enable a desktop calendaring application. Finally I gave C-API's step-wise setup for sensors, user-context feature extraction from these sensors, and learning user preferences to predict alarm type preferences for an individual user. C-API highlighted Sycophant's reusable, modular design for personalizing user interfaces.

My user-context based framework is one of the approaches to address the lack of personalization in current desktop applications. In the next chapter, I first review other projects that adapt interface behavior to system users and then briefly summarize the applications of Learning Classifier Systems (LCS) to real world problems.

Chapter 3

Related Work

This chapter synergizes the recent advances in context-aware adaptive user interfaces and genetics-based machine learning research to provide a background for my generalized user modeling framework. Section 3.2 summarizes the applications of learning classifier systems mainly in predictive data mining areas. I next give a broad overview of related work in adaptive user interfaces focusing on projects in attention and interruption management.

3.1 Adaptive User Interfaces

Current adaptive (intelligent) user interfaces research has focussed on managing a user's attention, building statistical models to predict the interruptibility of a user, and tracking a user's interactions with all applications to enable desktop applications become aware of tasks. The work reviewed here provided the following important guidelines during Sycophant's design phase. First, a user's attention must be carefully managed among competing applications to avoid having a negative effect on that user's task performance. Second, interruptions are better scheduled during task executions (task suspension and task resumption). Third, task related user activity data may be useful to make desktops more task-aware.

3.1.1 Attention Management Systems

Bailey *et al.* addressed the problem of user-attention management by interactive systems [8, 13, 14]. In an attention management system, the system tries to select the most opportune moment in a user's task sequence to interrupt that user. In attention management research, they identify a user's susceptibility to interruption overload caused by systems that fail to reason about the cost of interrupting a user.

In two user studies, they measured the effects of a news alert service interrupting a user engaged in different tasks such as editing, searching, and media tasks. During these studies they gauged interruption effects in terms of task performance, social attribution, and emotional state of a user. They developed task models that relied on event perception techniques (based on a user's predicted cognitive load) to predict the best points for system interruptions. These task models showed that the best interruption points produced less annoyance and frustration for users engaged in their *primary* tasks, which are daily tasks performed by users as their primary responsibility in their computing environments.

Based on their task models, Bailey *et al.* developed a framework for specifying and monitoring user tasks [13]. Their framework provided a language for specifying tasks, a database with a handler that managed events from applications, and a task monitor that observed a user's progress through the tasks they specified with their framework language. One of their findings was that their monitoring system could accurately follow a user's progress and that their attention manager learned task models over time to be more effective in decreasing disruptive interruption effects for a user. Bailey's attention management studies quantitatively showed the negative and disruptive effects of interruptions on on task completion time for users. From their studies, they also suggested that interruptions can be less disruptive if a system generated interruptions during task boundary points (users switching between tasks) due to a user's increased availability of mental resources at these boundary points. Extending these attention management studies, Iqbal worked on managing a user's attention across multiple desktop applications and devices.

3.1.2 Interruption Management Systems

Iqbal and Horvitz primarily focused on interruption management systems, that is, systems that manage notification cues generated by applications such as email clients, windows, and instant messengers [39]. They analyzed the multitasking behavior of 27 users over a two week period while the users suspended and resumed tasks (programming, document editing, presentation creation) during their normal work day. They logged a users' interactions with various applications and evaluated the effect of computer-based alerts (notification cues) on users' task execution behaviors. From their study, they first identified that users spent an average of 10 minutes on switches caused by alerts. Their second finding was that, after the alert, users spent an additional 10 to 15 minutes on other applications before returning to their disrupted task. To minimize this context loss associated with task switching, Iqbal and Horvitz recommended the use of visual cues to serve as reminders for a user for returning to their suspended applications.

Iqbal and Bailey also evaluated the feasibility of statistical models to detect and differentiate between breakpoints across different tasks [38]. Their models detected each breakpoint type across different tasks for a user with an accuracy ranging from 69 to 87 percent. The primary focus in this work was to enable their interruption management system to better realize breakpoint policies for interactive tasks. Similar to this interruption management research, Herlocker and Horvitz have worked on TaskTracker and Lumière attention management systems, respectively.

3.1.3 TaskTracer and Lumière

Herlocker *et al.* developed TaskTracer for helping multitasking users locate, discover, and reuse processes (applications) for completing their tasks [25, 60]. Their system collected user activity data related to resources used and accessed by a user. TaskTracer associated each user-defined activity with a set of files, folders, email messages, contacts, and web pages that the user accessed when performing that activity. Based on this logged data, their system detected a user's task

switches to predict her current task. They used two learning schemes to predict a user's current activity. With the first scheme, they predicted a user's current TaskTracer registered task with a precision of 80 percent; this prediction was based on a window's title, document and pathname information. Their second component leveraged email activity information (message's sender, recipient and subject information) to predict a user's current TraskTracer task with 90 percent precision.

Horvitz *et al.* similarly worked on managing a user's attention using probability and utility measures in their Lumière project [36]. Lumière considered a user's background (gaze-tracking, active applications), queries and actions to infer that user's needs. Horvitz and Apacible also used a dynamic Bayesian network for modeling a user's attentional focus and predicted the cost of interrupting that user [37, 33]. In contrast to these projects in adaptive user interfaces where the focus was on just task management for improving user productivity, the work that comes closest to my own is Fogarty's Subtle.

3.1.4 Subtle: Predicting Interruptibility

Fogarty's work focussed on building statistical models that predicted that state of interruptibility (highly non-interruptible and interruptible) of office workers. In his first study, Fogarty gathered audio and video information from the office environment of three participants [29]. He measured an office worker's level of interruptibility by periodically collecting self-reports from that participant.

After collecting self-reports of the interruptibility levels, Fogarty simulated sensors using a *Wizard of Oz* approach to evaluate which sensor(s) could best predict the level of interruptibility of a user. In a Wizard of Oz study, a human simulates the required intelligence behind the interface exposed to a study participant. The advantage of this approach is that an application designer can evaluate an interface without the implementation of face recognition or other artificial intelligence system aspects. Fogarty's paper on this topic and his dissertation give his Wizard of Oz methodology in detail [27, 29].

An example sensor he found from this Wizard of Oz approach was a phone-sensor. This sensor detected if a phone was off the hook in a participant's office and proved to be one of the indicators of that participant's interruptibility level (highly non-interruptible). These simulated sensors also helped him to gauge the potential of different sensors (without actually constructing them) for generating a predictive model of interruptibility levels.

In his next study, Fogarty compared the interruptibility levels assessed for the recordings in his earlier Wizard of Oz study. Fogarty used both human observers and statistical models (built using his simulated sensors) to evaluate the interruptibility levels of these recordings. The interruptibility levels predicted by his statistical models performed as well or better than human observers thereby showing the feasibility of statistical sensor-based models to estimate the interruptibility levels of office workers.

Informed by these studies, Fogarty developed the *Sutble* tool-kit mainly designed for a notebook (laptop). He deployed Sutble in a diverse office environment that included managers, researchers, and programming interns [28]. Subtle collected data from a system's opening, closing, audio analyses, mouse-clicks and WiFi sensing activities. Fogarty showed that his sensor-based statistical models in Subtle could predict the interruptibility of a participant (highly non-interruptible versus interruptible) better than human observers. Next, I integrate all these advances in adaptive user interfaces and differentiate my user-context based framework from the related projects discussed earlier in this chapter.

3.1.5 Incorporating these advances in Sycophant

Informed by attention and interruption management studies, to minimize the disruptive effect of an inappropriate interruption, I relied on user-feedback and user-context and designed Sycophant to generate an application action at the most opportune moment. In contrast to research that has mainly tried to address task

management by focussing only on a computer's internal environment (tasks, processes, applications), my user-modeling framework gathers *both* internal and external user-related information from a desktop's environment to better enable applications learn user preferences. While Fogarty's work focussed on deciding *when* to interrupt a user, in addition to predicting when, my approach also predicts *what* interruption type an application should generate for that user. Sycophant uses XCS, a learning classifier system (LCS) as a data mining technique to generate a user preferred interface action. The next chapter describe a traditional LCS and XCS in detail.

3.2 LCS Applications

In Learning Classifier Systems (LCS) research, Kovacs lists real-world LCS applications to solve a wide variety of problems in optimization, medial domains, data mining, control and modeling [41]. The XCS classifier system, in particular, has been successfully applied in diverse domains for data mining.

Wilson used XCS to mine the Wisconsin Breast Cancer (WBC) data (a widely-used benchmark) and highlighted XCS' promise from both performance and pattern-discovery viewpoints [63]. XCS mean test set performance of 95.56 percent on a stratified ten fold cross validation of WBC data was comparable to other widely used machine learnering algorithms.

Bagnall and Cawley evaluated XCS' potential for a supervised learning task by measuring its test-set predictive accuracy on the *Forest Cover Type* dataset, another bench-marking data set available from the University of Irvine's Machine Learning Repository [11, 12, 16]. They evolved classifiers to predict the forest cover type chosen from a set of seven classes. An exemplar in this data set consisted of ten continuous variables and two nominal categorical variables, four types of wilderness area designation, and forty soil types along with the forest type designation. Unlike smaller bench-marking data sets, this data set consisted of 581012 exemplars

in total. Their results showed that XCS, C5 (a decision-tree), *Clementine's* neural-network, and an implementation of support-vector machines (*libSVM*) achieved mean test-set prediction accuracies of 71.15, 83.44, 74.83, and 70.66 percent, respectively [4, 20, 49]. Based on XCS' competitive predictive accuracy, Bagnall and Cawley further demonstrated XCS' feasibility for predictive data mining tasks.

Llòra and Garrell have shown that genetic-algorithm based data mining techniques are competitive and robust schemes when compared with widely used machine learning techniques such as production systems, support-vector machines , a naive bayes classifier, and a decision tree [43]. Their genetic algorithm (GA) based classifier system's competitive results on public domain datasets again highlighted the promise of a GA-based approach for predictive data mining.

Saxon and Barry have evaluated XCS' performance on Thrun's Monks data set [51]. These three data sets included binary classification tasks of differing problem complexity. Monks-1 was designed to evaluate a new classification technique's ability to generate simple concepts. Monks-2 represented a much more complex relationship between the features in the data-set when compared to Monks-1's features. Monks-3's design was similar to that of Monks-1 with added noise and a few misclassified examples. XCS was successful on all these three bench-marking data sets thereby showing that the classifier system could not only learn a simple concept but was robust enough to learn both complex and noisy concepts.

Encouraged by XCS' potential as a data mining tool, Sycophant harnesses this classifier system to adapt an application's behavior to an individual user. To context-enable and personalize a desktop application within Sycophant, XCS predicts an interface action type to use for a particular user.

3.3 Chapter Summary

The previous chapter already outlined Sycophant's novel architecture and modular APIs based on adaptive user interfaces research discussed in this chapter. In this chapter I first consolidated the recent advances in context-aware systems and LCS research. Then, I differentiated Sycophant 's sensors that monitor both the

internal (keyboard, mouse) and external (motion, speech) environment of a user's desktop computer from other projects in attention management that have focused only on the internal environment of a desktop computer (tasks, applications). In contrast to these approaches, Sycophant leverages external user-context to enable Google Calendar and Winamp to learn an individual's preferences for different application actions. Before presenting results that establish Sycophant's successful personalization to user studies' participants, I give a general overview of different machine learning techniques that Sycophant employs to predict user-preferred actions in the next chapter.

Chapter 4

Predicting User Preferred Actions

This chapter describes the two main techniques which Sycophant leverages to predict application-specific user preferred actions. Sycophant primarily relies on *XCS*, a learning classifier system, and a *decision-tree* learner for predicting user preferences. Sections 4.2 and 4.3 explain Genetic Algorithms (GAs) and Learning Classifier Systems (LCS) respectively. The subsequent section covers XCS, a modified form of LCS, its main components and operation. Finally, I explain a decision-tree learning algorithm and give an example tree that predicts user-preferred calendar alarm types. The next section explains how a program *learns* user preferences.

4.1 Machine Learning

Earlier in section 2.1.2, I described the process of storing user-context data (sensor data along with user-feedback) at the context layer. A machine learning algorithm at the learning services layer mapped context features in this user-context data to application-specific user preferred actions. Before describing this learning process, I explain training and testing exemplars.

4.1.1 Definition

Training exemplars are rows of user-context data (sensor values and user-feedback) stored at the context layer. Sycophant generalizes from these training exemplars for learning to predict the correct type of action to use for interrupting a user when it is presented with a new (unseen) set of contextual features called a *test exemplar*. This method of inducing a concept from training exemplars has been researched extensively in the field of Machine Learning [21, 45, 9, 48]. In this area, the learning algorithms rely on example data and past experience to program computers for optimizing a desired performance criterion. One of the goals in machine learning is to build systems that do not need explicit hard coded instructions to deal with every special circumstance.

Tom Mitchell gives one widely accepted definition of machine learning [46]. According to him:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

When I apply this definition to Sycophant, its experience (E) is the set of sensor-collected contextual features along with the user-preferred application action (to use for that particular exemplar) stored in the training data. Sycophant's class of tasks (T) include generating an appropriate interface action-type. This dissertation evaluates Sycophant's performance (P) by measuring the application action prediction accuracy on a test exemplar.

Predicting a user-preferred application action based on labeled training exemplars is a form of *supervised* learning. More specifically, this process of forecasting a value (application action type) based on labelled data is *predictive data mining*.

4.1.2 Supervised Learning

Learning to classify the application action types into an appropriate class chosen from a set of finite classes is a form of supervised learning. In supervised learning

the actual outcome (correct application action) for each of the training exemplars is provided by a teacher (a user in Sycophant's case). That is, each training exemplar is explicitly labeled with the correct class to which it belongs.

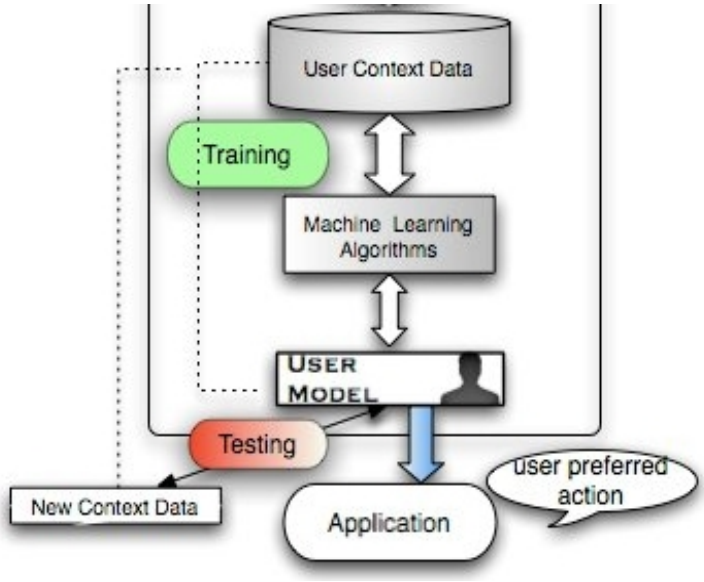

Figure 4.1: Supervised learning in Sycophant.

Figure 4.1 presents an overview of the supervised learning process in Sycophant. In Sycophant, the system first solicits user-feedback whenever it generates an interface action for a context-enabled application. Next, Sycophant labels the training exemplar with this feedback (user-preferred action) and stores this information back in that user's context data. Then, a learning algorithm uses this user-context data to generate a model that maps sensor features to application actions.

This work uses a test set (unseen exemplars) to check the predictive accuracy of the model generated by a learning algorithm. The task of predicting a user's preference for application actions based on her feedback is a supervised learning task. Sycophant operates a learning classifier system in the supervised learning mode to predict user preferences. A Genetic Algorithm (GA) is a key component of such a learning classifier system.

4.2 Genetic Algorithms: A Short Description

In the United States, John Holland along with his colleagues and students at the University of Michigan developed the concept of Genetic Algorithms (GAs) [34]. I do not attempt to fully describe all GA aspects, but instead refer readers to a standard evolutionary computing text such as Goldberg's book [31].

A GA's search mechanism is guided by the principles of natural selection and natural genetics. Figure 4.2 shows a genetic algorithm. Starting at the top, the figure shows a GA initializing its search from a randomly generated set of strings which constitute the initial population. Generally, these strings are binary strings made up of either 0's or 1's. Sometimes a metacharacter ($\#$) in a bit string represents a 0 or 1. A GA population is a collection of potential solutions to the problem tackled by the GA. The randomly initialized population forms the parent population. In a GA, a structured, yet randomized exchange is combined with survival of the fittest among string structures to operate robust search. Selection, crossover, and mutation are the operators used in guiding search within a GA.

Next, during the selection process, two bit strings are chosen to reproduce and these two strings create new bit strings for the offspring population. There are many selection mechanisms, one of the widely used ones is the *roulette-wheel* process where the selection probability depends on the *fitness* of the bit string [15]. To evaluate a string, a fitness function is defined which helps the GA to discriminate between two candidate strings. This fitness function depends on the problem which the GA is trying to solve.

During crossover and mutation, bits and pieces of the fittest strings in the parent population are used in creating a new set of strings in the offspring population. A very low probability operator that just flips a specific bit is used for mutation. Crossover is a high probability operator and is used to exchange parts of reproducing strings between points on the strings participating in reproduction. Crossover and mutation operators generate new solutions for evaluation. Strings (solutions) that perform poorly are filtered out probabilistically. Eventually, the population converges to highly fit solutions. There are many options to manage parent and

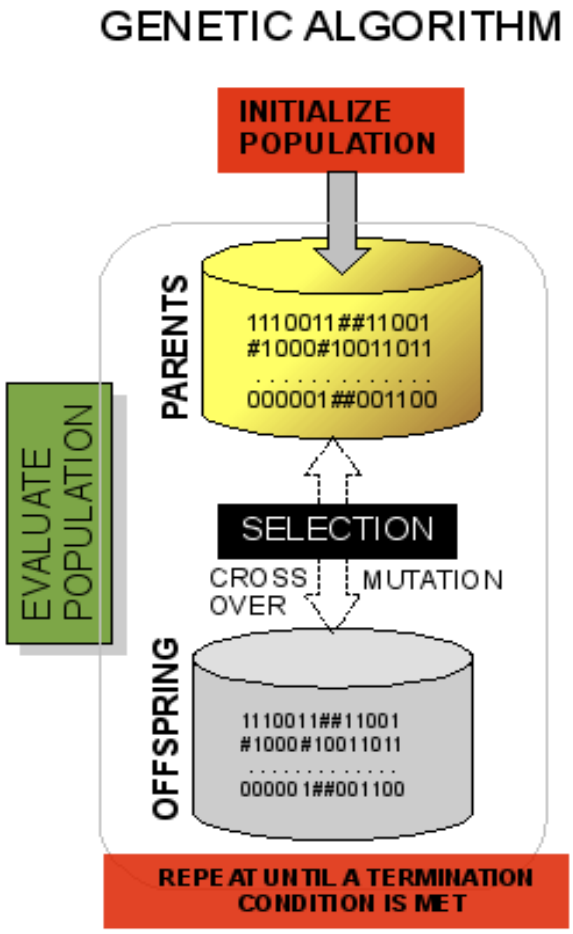

Figure 4.2: A Genetic Algorithm

offspring populations; in a canonical GA, the offspring population replaces the parent population.

The process of evaluation, selection, and string recombination is an iterative process. The genetic search process continues until some termination condition is met. A GA forms a crucial component of a Learning Classifier System (LCS), a genetics-based complex adaptive scheme.

4.3 Learning Classifier Systems: Brief Overview

Learning Classifier Systems (LCS) were again John Holland's innovative work in creating a domain-independent rule-based machine learning complex adaptive scheme [34]. Holland's work in this area laid a comprehensive foundation for Genetics-Based Machine Learning (GBML) techniques [31]. A Learning Classifier System (LCS/CS) is a complex adaptive scheme that learns syntactically simple string rules. These string rules are called *classifiers*. Here, I describe an LCS to contrast its operation with the XCS classifier system that is described in detail in the next section. Goldberg gives more information about LCS' architecture and operation [32]. Figure 4.3 shows an LCS' three main components - the rule and message system, an apportionment of credit system, and a genetic algorithm. Here is a brief description of these three components.

1. **Rule and message system:** is a production system which consists of rules of the form:
   `if <condition> then <action>`
   A rule directs the CS to execute the `<action>` part when the `<condition>` component is satisfied.
   A classifier is of the form :
   `<classifier> :: <condition> : <message>`
   Figure 4.3 shows LCS' *detectors* sensing an environmental information input (message) and this message gets posted to a finite-length message list. Classifiers that match this environmental input may get activated. An activated classifier posts its message to the *message list*. This message in turn might invoke other classifiers and eventually may cause an action to be taken by the system's *effectors*.

2. **Apportionment of credit system:** LCS uses the apportionment of credit mechanism to learn the relative value of different classifiers (rules). A competition is held amongst classifiers where the right answer to relevant messages goes to the highest bidders, with subsequent payment of bids serving as a source of income to previously successful message senders. This is the *bucket brigade*

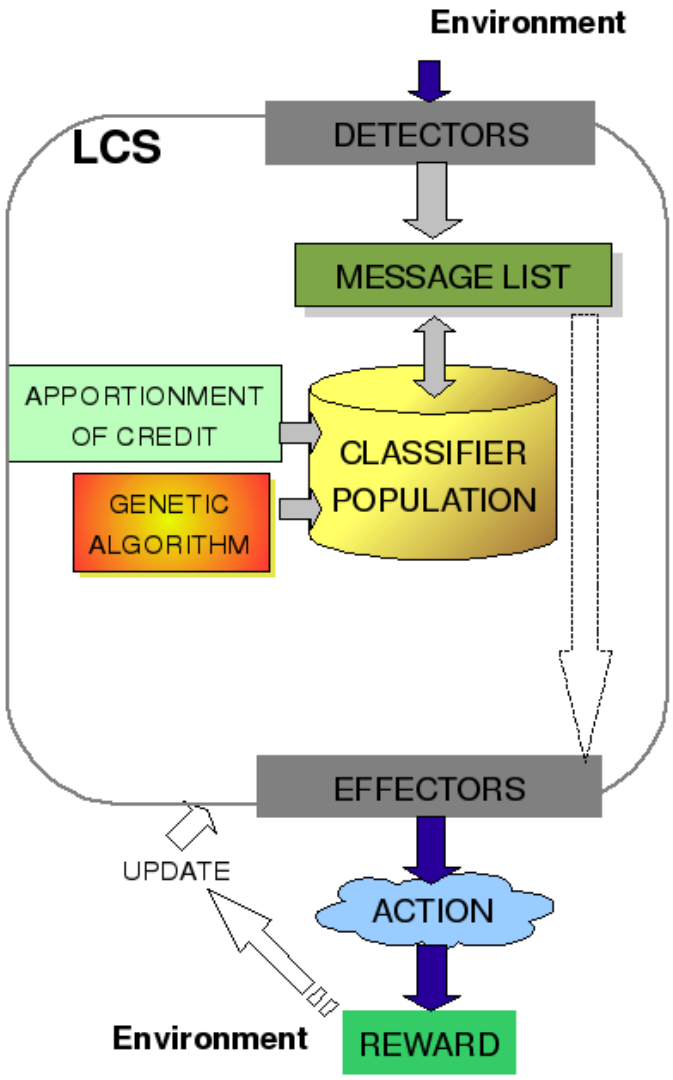

Figure 4.3: A Learning Classifier System's (LCS) architecture

algorithm where a chain of middlemen is formed from the manufacturer (the detectors) to the consumer (environmental action and payoff). The competitive nature of this economy ensures that good (profitable) rules survive and bad rules are eliminated.

3. **Genetic Algorithm:** a tripartite process of reproduction, crossover, and mutation is used for creating new, possibly better rules into the system. A GA is run periodically after a predetermined threshold number of steps or based on a metric related to the system's performance.

The rules (classifiers) in an LCS are of fixed length and this fixed rule size allows easy execution of genetic operators on the classifier population. In an LCS, multiple classifiers (rules) can match the same environment message, and thereby activate multiple rules in parallel. This parallel rule activation is an advantage compared to to a traditional model of an expert system where only one rule can be activated. Sycophant uses an LCS variant, XCS, as one of its learning approaches to automatically predict user-preferred application actions.

4.4 XCS

To address some of LCS' drawbacks, Wilson simplified Holland's LCS by removing the *bucket brigade* and the internal message list and he derived XCS from his ZCS and Animat programs [40, 61, 64, 65, 66]. Wilson's paper and Goldberg's book provide more details on the bucket-brigade algorithm and the internal message list [31, 65]. The next three sections describe XCS' architecture, the operation of the system for generating new classifiers, and a procedure called condensation used to reduce the size of a classifier set.

The key distinction between Wilson's XCS and a traditional LCS is that an XCS' classifier fitness depends on the *prediction of its expected payoff* while an LCS' classifier fitness depends on the *actual prediction itself*. Next, I describe XCS' architecture, the operation of the system when subjected to an accuracy criterion to evolve a minimal, maximally general, and accurate model for a learning task, and the parameter settings used for learning to predict user preferences within Sycophant [40].

4.4.1 Architecture

Figure 4.4 depicts XCS' architecture. This figure shows XCS interaction with an environment via *detectors* for sensing the environmental input and *effectors* for executing an action. The environment provides a *reward*, a scalar reinforcement, that is action dependent. XCS' core consists of a classifier population where each classi-

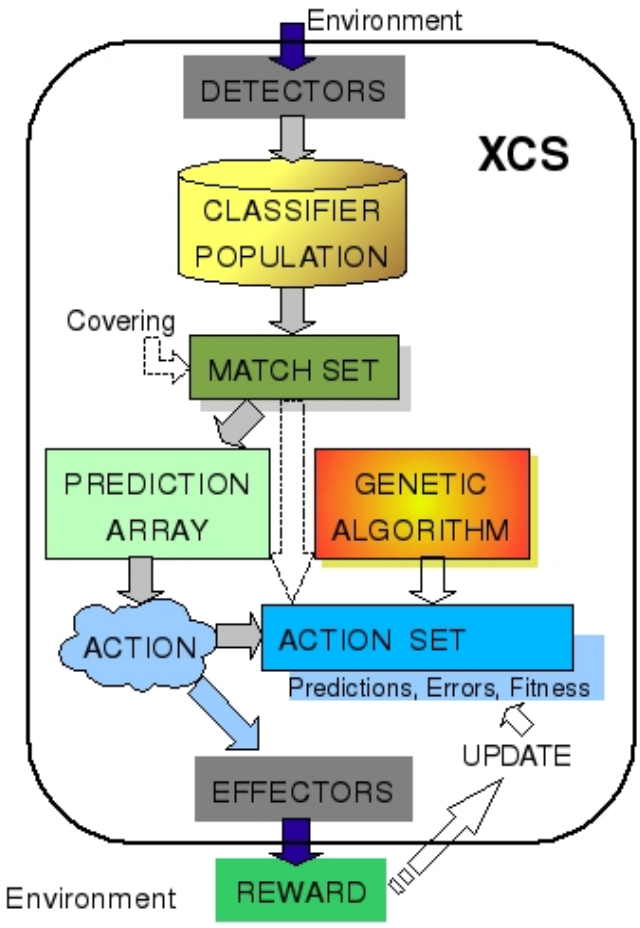

Figure 4.4: The XCS classifier system

fier represents a simple if-then rule; a classifier's left side consists of a single condition and its right side codes an environmental action. An XCS' classifier is similar to an LCS' classifier described in the previous section. Figure 4.5 shows a classifier population. In addition to the classifier population, XCS components include

the *match-set* formed for a sensed input message, the *prediction array* created from the match-set, an *action-set* corresponding to one of the actions selected from the prediction array, and a GA that searches through the space action-set classifiers. Before describing these XCS components, I briefly explain the following important classifier attributes to aid in a better understanding of XCS' overall operation. Wilson's papers give finer details about these classifier components [64, 65, 66]. Each XCS classifier consists of the following important attributes:

- **Condition**: is defined over the ternary alphabet $\{0, 1, \#\}^L$, where L is a classifier's individual bit-string length. The meta character $\#$ matches either a 0 or 1. Whenever the system senses an input from the external environment, XCS' classifiers try to match the input's condition part. For example, when XCS learns to predict a user's Google Calendar's alarm preference, the condition component corresponds to an exemplar from the user-context data. Table 2.1 shown earlier in Chapter 2 presented sample user-context data values. The corresponding exemplar for these sample values is:

  participant-1,1010(16),1000(4),0000(0),0000(0)

  A system generated classifier's condition component for the above exemplar would be as shown below. Section 4.4.4 explains this binarization in detail.

  0001,101#(10000)1000(00100)000#(00000)00#0(00000)

- **Action**: is an operation which a classifier can execute; generally, an action is chosen from a set of finite classes. For example, in case of Google Calendar, a classifier's action could be a voice-alarm, a visual-alarm, both voice and visual alarms, or none. Section 5.3 in the next chapter describes these alarms in detail.

- **Prediction Estimate**: is the value of an estimated pay-off if a classifier's condition part matches the environment's input and the system executes this classifier's action. It is this key attribute which distinguishes XCS from a traditional LCS.

- **Prediction error**: is an estimate of error made by a classifier's predictions. The system computes prediction error after it receives an environmental reward.

- **Fitness**: represents a classifier's fitness value. In XCS, a classifier's fitness is calculated by reestimating its attributes if that classifier is present in the action-set. The system receives an environmental reward as a result of a classifier executing a particular action. This reward influences fitness reestimation. The next section describe action-set formation. A classifier's reward also factors in the system's predictions and error updates. Subsequently, the system estimates accuracy and converts this estimate into a relative accuracy value. This relative accuracy value along with the learning rate, β, is used to update a classifier's fitness. Wilson's papers give mathematical details about these computations [65, 66].

I next explain the other main XCS components including the *match-set*, the *action-set* and the GA in the context of system operation.

4.4.2 XCS System Operation

Figure 4.5 shows the system's operation during training and testing phases. I first give a brief overview of these phases and then explain training/testing in detail.

In the training phase, XCS evolves a classifier set for a *training fold* (a subset of all data collected for a user) and stores these classifiers after an experimentally determined stopping criterion of repeatedly sampling 20000 random problems (exemplars) or when the training performance reaches 1.0. Thus, from the training data XCS learns a classification model of the target concept, that is, the interface action type to predict.

During the testing phase, these training-set evolved classifiers get tested on the *testing fold* (a mutually exclusive subset of the training fold) and record the system performance. The testing fold thus helps to evaluate XCS' predictive accuracy on unseen cases.

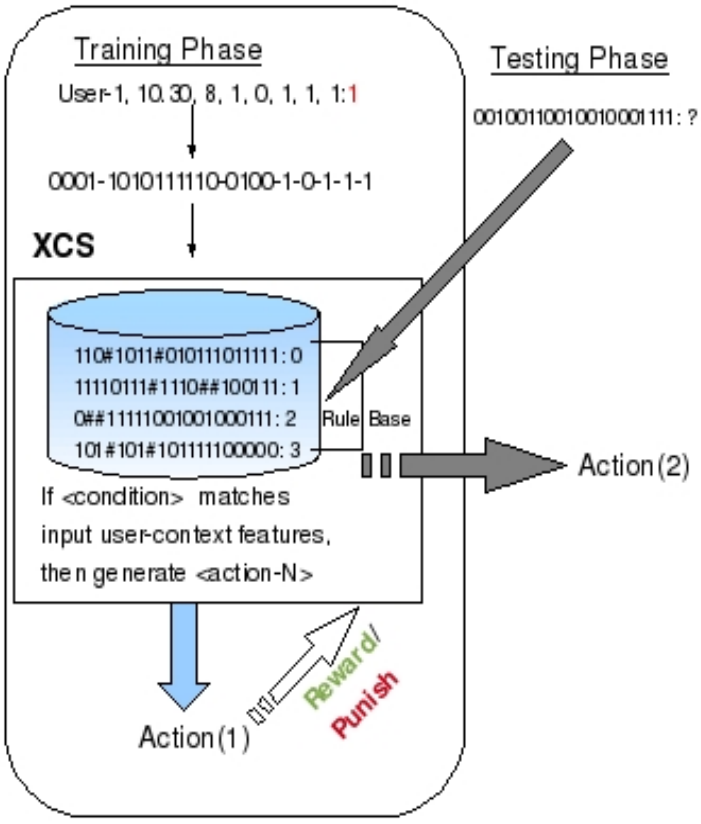

Figure 4.5: XCS' system operation. The figure shows a trained XCS that predicts a voice alarm (action type-2) for an input test exemplar.

While training the system, a user-context data exemplar serves as the environmental input message string to the system. XCS' classifiers whose condition component match this input message string are grouped to form the match-set (M). For each of the actions present in M the system computes a fitness-weighted average for each of M's classifiers. The system next selects the *best-action* which is the action associated with the highest fitness in the prediction array. XCS' effectors send this best-action to the external environment. Based on this action the system receives an appropriate reward from the external external environment. This cycle of environmental input sensing, action selection and factoring in the reward is continued

until the system meets the training-phase termination condition of randomly sampling 20000 problems or achieving a training-set prediction accuracy of 100 percent (experimentally determined). The system stores the classifiers (rules) at the end of the training phase. XCS' is thus operated in a *single-step supervised learning* mode.

During system testing a new user-context exemplar whose action component is unknown is available to the system. XCS uses the training-phase evolved classifiers to match the condition component of the test-exemplar. Similar to the training phase the system forms a match-set for this test-exemplar and predicts an action to be executed.

In more detail, during the training phase the system forms the match-set for a sensed input message string from the environment. If none of the classifiers in the population match the condition component of the input message string, the system creates new classifiers with each of the possible environmental actions to form the match-set. This procedure is called *covering*. My set-up starts with an empty classifier population and initiate covering only at system initialization.

After the system creates M, it calculates a fitness-weighted average of each of M's classifiers for each of the actions present in M. There are many action selection mechanism, for the system's optimal performance, this work uses the best-action selection mechanism for choosing one of M's actions (a). The system sends this action, a, to the external environment for the sensed input message string. XCS gets a 1000 (experimentally determined) reward value if it takes the correct action and 0 otherwise. At the same time, the match-set classifiers which proposed a are grouped to form the action-set (A). The action-set classifier attributes (including fitness) get updated based on the actual reward received from the environment. Either to improve system or to generate new classifiers or both, a GA is run on the action-set based on θ_{GA}, a threshold parameter.

When the system chooses an action to execute and an appropriate reward is received from the environment, classifiers in the action-set whose action was chosen get their fitness values updated. This procedure ensures that accurately predicting classifiers tend to reproduce more due to their increased fitness for choosing the correct action, while inaccurately predicting low-fitness classifiers tend to get

weeded out of the population. This cycle of environmental input sensing, action selection, factoring in the reward and running a GA on the action-set classifiers is continued until a termination condition is met. Based on experimental runs, I use a termination condition of randomly sampling 20000 problems (context data exemplars from the training set) or a training set performance prediction accuracy of 100 percent. XCS also uses *condensation* to extract a minimal classifier subset to represent the final solution for predicting user-preferred interfaces actions.

4.4.3 Condensation

Condensation consists of running XCS with crossover and mutation rates set to zero. This process suspends genetic search since no new classifiers are generated. However, XCS' selection and deletion processes continue to operate whenever the GA gets triggered. As a result, there is a tendency for less fit, less general classifiers to be weeded out of the population. Reducing the number of classifiers enables the system to extract a minimal classifier subset capable of representing the final solution [40, 65]. The system enables condensation after a stopping criterion is met. Based on trials to tune system performance, the system halts condensation after sampling an additional 20000 problems from the training set. Next, I give the XCS parameters values used in this dissertation.

4.4.4 XCS' Encoding and Parameter Settings

Table 4.1 shows important XCS parameters. The other parameter values are the default values provided by Butz [19]. The system starts with a maximum population size of 100000. A GA in a classifier system provides a niching facility that allows cooperative rule-sets to coexist in the classifier population. At the same time, a GA allows competing rule-sets to converge on optimum rule attributes within a niche. Hence, having an adequate population size allows the GA to experiment with recombination.

The remaining system parameter values are based on attempts to optimally tune the system performance (test-set prediction accuracy). For an XCS classifier,

Table 4.1: XCS parameter settings. This table shows the values of different parameters used within Sycophant.

I Parameter	II Notation	III Value
Population Size	N	100000
Learning Rate	β	0.2
GA crossover-rate	χ	0.8
GA mutation-rate	μ	0.04
GA meta character probability	#	0.33
GA Threshold parameter	θ_{GA}	25

β is the learning rate for updating predictions, prediction error, fitness and the action set size estimate. I set β to 0.2. To tune system performance, the threshold for applying the GA in an action set, θ_{GA}, is set to 25, the GA's probability of crossover(χ) to 0.8, GA mutation probability(μ) to 0.04, and GA meta character (#) probability to 0.33.

Table 4.2 shows the encoding steps used to convert a user-context data exemplar into an equivalent XCS classifier. Column II shows sample data and Column III shows the datum's corresponding values.

Table 4.2: A procedure to encode a user-context exemplar into an XCS classifier.

I Row	II Data	III Corresponding Value
1	user-context	participant-1, 1010(22), 0000(0), 1010(26), 1010(2):3
2	binarized user-context	0001, 1010(10110), 0000(00000), 1010(11010), 1010(00010):3
3	XCS classifier	0001101010110000000000101011010101000010:3

Row 1 in the table shows a user-context exemplar. The first attribute denotes a study participant, and the next five are motion attributes - Any1, All1, Any5, All5, and Count5 (shown in parentheses); Section 2.1.2 described each of these user-context features. Similarly, the next fifteen attributes in groups of five relate to speech, keyboard, and mouse activity. The value after ":" denotes the class to which this exemplar belongs. Show here is an exemplar for Google Calendar alarms, where class 3 corresponds to both voice and visual alarms. The binary

equivalent of the user-context exemplar is shown in row 2. Row-3 shows a binary string that represents one of XCS' classifiers in the population. Note that the binary values before the ":" denote the *condition* part and the 3 denotes this particular classifier's *action* part. In addition to using XCS for predicting a user-preferred application action, Sycophant also relies on a decision-tree, a widely machine learning technique.

4.5 A Decision Tree Learner

I first explain a decision-tree and then illustrate with an example how to interpret personalization rules by using a decision-tree learned for participants' calendar alarm types.

4.5.1 Brief Description

A decision tree is a classifier which contains a structure that either indicates a *leaf* or a *decision node* [49]. A *leaf* indicates the class to which an exemplar belongs. In a decision-tree an attribute's value is tested at a *decision node*. A decision node has one branch and subtree for each possible outcome of the test carried out on the attribute at that single node. To classify a case (exemplar), you first start at the root of a decision tree and traverse the tree until you encounter a leaf (class of the case to be predicted). Next, you test a case's outcome at nodes which are not leaves and move down the subtree corresponding to the outcome of this test. When you repeat this testing process for a case at non-leaf nodes your path eventually ends at a leaf node.

Sycophant uses *J48*, a decision tree learner available with the *Weka* machine learning toolkit to predict user preferred application actions [67]. *Weka*'s J48 decision tree algorithm is based on Quinlan's widely used decision tree, C4.5 [49]. I chose a decision-tree to view the task of predicting an application action preference as *descriptive data-mining* problem. In descriptive data mining, we are interested in

describing patterns that exist in collected data in contrast with *predictive data mining* where we forecast values based on existing data patterns. Note that you can consider a decision-tree learner as both a descriptive and a predictive data mining technique. In this work, a decision-tree not only predicts a user-preferred application action but also generates the rules which predict that action. To illustrate a descriptive data mining task, I next examine a decision-tree generated in a user study conducted to learn a participant's calendar alarm type preferences.

4.5.2 An Example Decision-Tree

Figure 4.6 shows a part of a decision tree generated for participants in the second user study with Google Calendar. I describe this study in detail in Chapter 6. The tree shown is constructed after being trained on the user-context data and the tree nodes are labeled according to the features already explained in Section 2.1.2.

To interpret *Example Rule* 1, start at the root node of the tree and traverse the tree along the dotted line towards a leaf node. The rule here checks to see if there was any speech in the last five minutes (speechCount > 0), next examines if keyboard usage was low (Count5 = $4/20$), and monitors for low motion activity (motionAny1 = 0). The rule also checks for low mouse usage (mouseCount5 < 8) and examines the user-id to predict an alarm type. This is an instance of personalization since the alarm type prediction depends upon the identity of a user. From the decision-tree snapshot, you can infer that if there was speech, low keyboard and mouse usage, user-1 preferred a voice alarm if the mouse was not used and no alarm if mouse was used. At the same tree level (user), unlike user-1, user-2 preferred a visual alarm if mouse was used and preferred not to be interrupted (no alarm) with low mouse usage. In the same scenario, user-3 preferred to be not interrupted. The user-context features present at different nodes in the decision-tree emphasize the importance of external user-context (speech derived user-context features) to predict an appropriate alarm for these three users. I have found similar evidence of personalization to individual users where user-context has played a significant role in the other user studies.

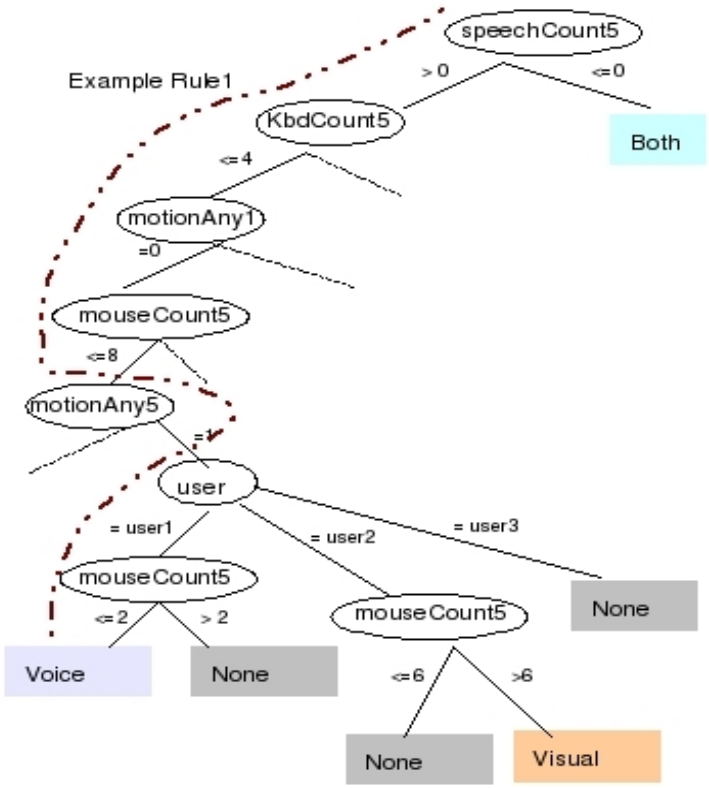

Figure 4.6: A decision-tree example. This figure shows a rule that predicts a Winamp action for participants 1, 2, and 3

4.6 Chapter Summary

In this chapter, I outlined two machine learning techniques – XCS, a learning classifier system, and J48, a decision-tree learner – which Sycophant primarily relies on to predict application-specific user-preferred actions. Based on these two approaches, the next four chapters give results from a series of real-world user studies that I conducted to evaluate Sycophant's feasibility and robustness for learning user preferences. The *Office of Human Research Protection* at our university validated and approved my studies involving human subjects. The same office provided the necessary certification to conduct these studies.

Chapter 5

Study 1: Pilot Study

This chapter first gives the rationale for conducting four user studies and next describes a methodology used to evaluate a learning algorithm's performance for predicting user preferences. Informed by the outlined study rationale and the results evaluation methodology I finally provide the results from a pilot study which investigated whether Sycophant could successfully predict a user's interface action preferences.

5.1 User Studies Rationale

I designed the user studies to answer the following questions:

1. **Feasibility**: Can Sycophant accurately predict a user-preferred application action?

2. **Relevance of User-Context**: Does user-context help a machine learner to better predict a participant's application action preference?

3. **Robustness**: Which machine learner is best-suited for learning user-preferences for different types of application actions?

4. **Generalizability**: I explored generalizability related to users, applications, and duration of use.

(a) *Across users*: Does my user-context based approach to learn participant preferences generalize across multiple participants?

(b) *Across applications*: Can Sycophant context-enable multiple desktop applications?

(c) *Duration of Use*: Can Sycophant successfully predict user-preferences both in short-term and long-term use of applications for an individual user?

For the remainder of this dissertation, I refer back to these questions and use the evaluation methodology outlined in the next section whenever I discuss user study results.

5.2 Results Methodology

Sycophant primarily uses, XCS as a predictive data miner and a decision tree learner as a descriptive data miner, to predict user-preferred application action types. Chapter 4 gave broad overview of these two machine learning schemes.

I use a two sample t-test and make decisions on a five percent significance level. With this test, I assess whether XCS' mean test-set performance is statistically significantly better than that of a decision-tree machine learning algorithm (J48) [22]. More specifically, the null hypothesis is that XCS' and the decision-tree learner's performances are the same. The alternate hypothesis is that XCS' performance is better than that of J48 on the application action prediction tasks. With this t-test, I assume that the models generated by XCS and J48 are independent. This independence assumption may not be completely satisfied since the learning algorithms generate their models based on data collected from the same participant. Even though the nature of the dependence structure is unknown, it is our belief that the statistical dependence between the samples is not really strong. Additionally, since the sample sizes are large, the sample dependence does not hinder the analysis or influence the results very much, and the noted performance differences between XCS and J48 are still valid.

To record XCS' performance, I note the percentage of correctly solved problems in the last 50 sampled problems. Sycophant assumes that a problem is solved if a classifier correctly predicts the interface action type to take for the classifier's condition component. For example, if the environment input message string is:
00011010101100000000001010110101010000010:3
and an XCS classifier:
`0001101010#0##00000#0010#011#0010#0#0###`
predicts 3, then I assume that the environmental input message string (problem) is correctly solved by the system.

This work uses a *cross-validation* scheme to evaluate a machine learning algorithm's test-set prediction accuracy. In this procedure, I divide the data-set into N mutually exclusive folds (subsets). There are many ways to create these folds, in this work preserves the class distributions for each fold during cross-validation. Next, I train a machine learning algorithm on $N - 1$ folds and evaluate its testing performance the other unsampled fold. The same process is repeated iteratively for all the N folds by considering each of these folds as a testing fold and record an average of the testing (or training) performance for all these N folds. This cross validation procedure is a useful and widely used metric to assess a learning algorithm's performance of unseen (test) exemplars.

I repeat the same cross validation procedure for the decision-tree learner, J48. Based on experiment runs to optimally tune XCS, I train XCS by making it evolve a classifier set on a training fold and store these classifiers after a stopping criterion of repeatedly sampling 20000 random problems (exemplars) or when the training performance reaches 1.0. Thus, from the training data XCS learns a classification model of the *target concept*, that is, the interface action type to predict. I test these training-set evolved classifiers on the testing fold and record the system performance. The testing fold thus helps to evaluate XCS' predictive accuracy on unseen cases. With this evaluation methodology, I next investigated the feasibility of predicting a calendar's alarm type preferences.

5.3 Predicting Application Action Preferences

The first study was a pilot study to evaluate the feasibility of learning a user's interface action preferences based on user-context. For this study, I developed a functional calendaring application that had options for setting (or deleting) appointments. This calendaring application generated four types of alarms:

1. *None*: No alarm was generated to interrupt a user.
2. *Visual Alarm*: A popup window displayed the appointment text.
3. *Voice Alarm*: A text-to-speech synthesizer voiced out the alarm text.
4. *Both*: Combined visual and voice alarms.

I investigated whether Sycophant could successfully leverage user-context information to predict one of these four calendar alarm types. Three participants set appointments for their daily activities over a period of six to eight weeks. During this period the system collected 323, 347, and 354 exemplars from these three users, respectively. I labelled the task of predicting whether or not to interrupt a user with an alarm as the *Two-Class Problem*, and the task of predicting an alarm from one of the four alarm types as the *Four-Class Problem*. The 2Class problem prediction accuracy helped to evaluate Sycophant's performance on deciding *when* to interrupt a user for comparing my results with Fogarty's interruption studies [29]. The 4Class problem prediction accuracy helped to assess Sycophant's performance on deciding *how* to interrupt a user.

I first evaluated the performance of the following seven machine learning algorithms in the *Weka* machine learning tool-kit on the 2Class and 4Class problems: Zero-R, One-R, J48 (decision-tree), Bagging, Logit-Boost and NaiveBayes [35, 49, 30, 26]. One-R generates a one level decision tree which tests only one particular attribute and forms a set of rules based only on that attribute. Chapter 4 explained a decision tree; Weka's J48 builds a C4.5 decision tree. Bagging creates n artificial data sets from the original data set and applies a decision tree inducer on each of them. The n generated classifiers then vote for the class to be predicted. LogitBoost

uses a learning algorithm for numeric prediction and a combined model is formed which is then used for classification. NaiveBayes selects the most likely classification based on a set of attribute values using prior probabilities and conditional densities of the individual features.

On the preliminary data collected over two weeks from the first participant, these six machine learners (except Zero-R) had alarm-prediction accuracies close to 83 and 64 percent on the 2Class and 4Class problem, respectively. [44]. I chose J48 (decision-tree learner) for its descriptive data mining characteristic that helped to inspect how the alarm predictions were being made (by interpreting the decision-tree rules). I next ranked context-features for this participant using the *Information-Gain Ratio* (IGR), a measure for evaluating the relevance of an attribute for a decision-tree, in Weka. IGR chose these features amongst the top 10 selected:

```
 Keyboard-Count5,Keyboard-Any5,Mouse-Count5,Mouse-Any5,
Keyboard-Any1,Mouse-Any1,Keyboard-Immed,Mouse-Immed,
Motion-Count5,Motion-Any5
```

The *Sensor-Immed* feature checked if a sensor was active in the last 15 seconds before an alarm was generated; Section 2.1.2 explains the other user-context features in detail. At this stage, it was still unclear whether user-context features (motion and speech) were important for the decision-tree learner to predict this participant's alarm type preferences. However, the presence of motion-related features encouraged me to collect more data from my pilot study participants. This initial evaluation also showed that Sycophant could predict whether or not to generate an alarm with an accuracy comparable to that of Fogarty's interruptibility studies. Sycophant still had a low predictive accuracy on the 4Class problem. To address this issue I investigated whether XCS could better predict this participant's alarm-type preferences on the 4Class problem.

I extended an implementation of XCS in Java (XCSJava1.0) and modified it to work with Sycophant's user-context data [19]. With XCS, Sycophant's training and test set prediction accuracies were both 94 percent for this user [55]. To identify the most robust learner, I next compared XCS and the decision-tree's performance on

the 2Class and 4Class problems for all three users [54].

Table 5.1 gives the results of this evaluation. In this table, Column I lists the

Table 5.1: Study 1: This table shows the performance of XCS and J48 on the 2Class and 4Class alarm problems, respectively.

	2Class			4Class		
I	II	III	IV	V	VI	VI
participant	ZeroR	J48	XCS	ZeroR	J48	XCS
1	0.57	0.74	0.75	0.57	0.70	<u>1.00</u>
2	0.60	0.79	<u>0.98</u>	0.60	0.73	<u>1.00</u>
3	1.00	1.00	1.00	0.99	1.00	0.92
all	0.61	0.87	0.88	0.39	0.82	<u>0.88</u>

participant, here *all* denotes the combined data from all three participants; I combined the participant data to evaluate Sycophant's generalizability across different participants. ZeroR's (base-rate) test-set predictive accuracy on the 2Class problem (performance) is shown in Column II. Similarly, the tables shows the decision-tree's and XCS' performance in Columns III and IV, respectively. Columns V, VI, and VII show the performance of ZeroR, J48 and XCS on the 2Class problem. The underlined values indicate the cases where XCS significantly outperformed the decision-tree.

On the 2Class problem, XCS and J48 performed better than Zero-R's base rate performance. XCS' performance of 75 and 100 percent matched that of J48 whose prediction accuracies were 74 and 100 percent for two participants. XCS with a predictive accuracy of 98 percent also outperformed J48 which had a predictive accuracy of 78 percent for the another participant.

On the 4Class problem of learning to predict one of the four alarm types (none, visual, voice, and both), again, XCS and J48 performed better than Zero-R's base rate performance. XCS significantly outperformed J48 for two participants with predictive performance of 100 percent (for both participants) when compared to J48's 70 and 72 percent for these two participants. Also XCS' performance (92 percent) was worse than J48 (99 percent) for participant-3. When I examined this

participant's data, I found that visual and voice alarms were preferred by this participant for $349/352$ instances. I attribute XCS' lower predictive accuracy to this participant's static alarm type preference; this also explained the majority voting Zero-R's high predictive accuracy. Again, XCS outperformed the decision-tree learner on the 4Class problem on the combined data from all three users. This study measured a learning algorithm's performance on this combined data set to evaluate the generalizability of predicting alarm preferences across all three participants.

The pilot study results demonstrated that Sycophant could accurately predict a participant-preferred alarm type, thus answering the *Feasibility* question posed at the beginning of this chapter in section 5.1. The 2Class problem's predictive accuracy (deciding whether or not to interrupt a participant) was comparable to that of Fogarty's interruption based studies [29]. Also, the results on the 4Class problem showed that Sycophant successfully learned *when* to interrupt a user.

5.4 Chapter Summary

In this chapter I first outlined the rationale for my study designs and formulated the main questions that this dissertation answers. Then, I explained the results evaluation methodology. After this, I described a pilot study whose results substantiated the feasibility (first question posed) of predicting user preferences for interface actions. Informed by the pilot study and encouraged by XCS' superior predictive accuracy, I next investigated whether Sycophant could learn Google Calendar alarm type preferences for multiple participants.

Chapter 6

Study 2: Google Calendar

While the pilot study results in the previous chapter confirmed that Sycophant could successfully predict when to generate an alarm for a participant, two issues still needed clarification. First was the issue of user-context's relevance to learn user preferences. The second issue was checking whether Sycophant could choose how to interrupt multiple participants (alarm type to use) with high accuracy. Encouraged by XCS' 88 percent predictive accuracy for the 4Class problem on the combined data from all three participants, I next examined whether Sycophant could predict user-preferred interface action types for multiple users. Specifically, I investigated whether XCS was one of the machine learners better suited for predicting user preferences and also verified whether user-context enhanced XCS' interface action prediction accuracy.

This chapter starts by outlining my short-term Google Calendar study design to learn alarm type preferences for multiple participants. I then give results for the 2Class and 4Class problems that show user-context helped Sycophant to better personalize its alarm generation to individual study participants.

6.1 Study Design

Figure 6.1 shows the user-context sensor positions for the study enabled desktop computer. [1] This study simulated an average work day in our research lab where

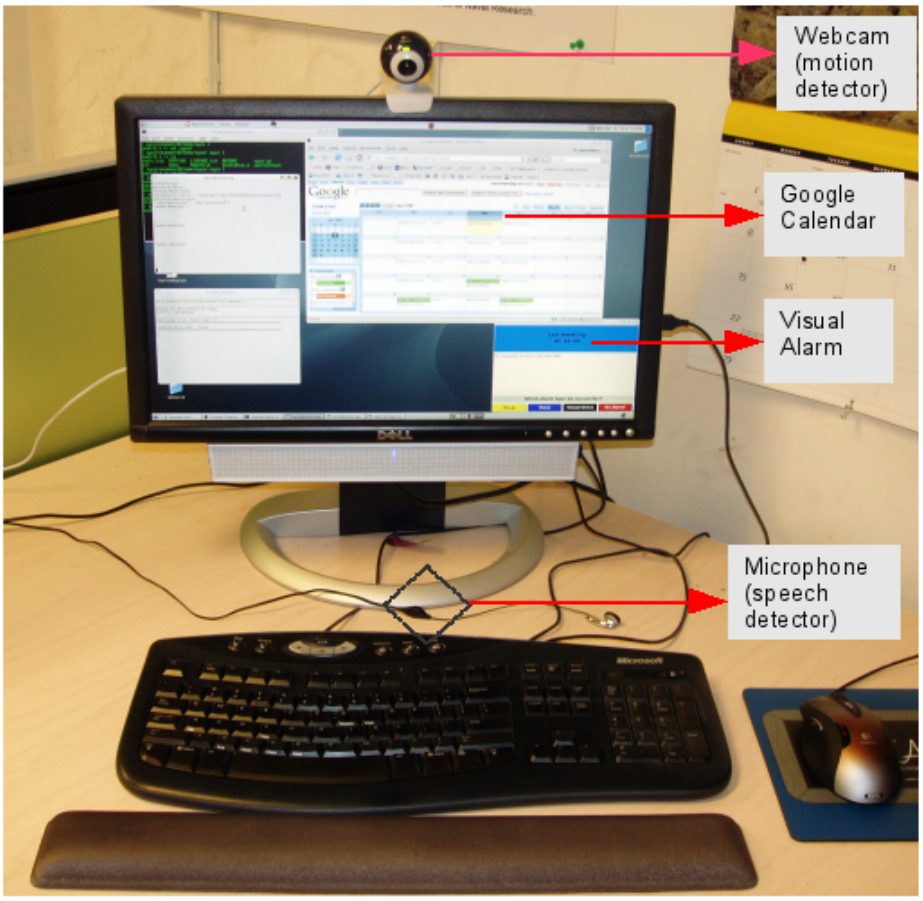

Figure 6.1: User study computer setup. The figure highlights the motion sensor (web-camera), the speech sensor (microphone), and a visual-alarm generated by Google Calendar

students read research papers. During the study, the system collected short-term calendar usage data from 10 participants in four separate 45 minute sessions. Note that this study duration was short-term when compared with the first pilot study that collected calendar usage data from three participants over a span of four to six weeks. Most students in our lab listen to music while they read research articles and these students are sometimes interrupted with conversations from neighbors. The study's objective was to periodically gather user-context during each session

[1]Section 2.2 described the approach to gather motion, speech, keyboard and mouse information.

along with a user's alarm type preference. Since the actual alarm type preference is explicitly labelled by a participant for each of the training exemplars, I could frame this alarm-preference prediction as supervised learning task for Sycophant. On this supervised learning task, this study investigated whether user-context helped Sycophant to accurately predict a participant's preferred alarm-types.

In a session, a participant read an article on the study desktop for the first 30 minutes, and answered article related questions in the last 15 minutes. Sycophant generated alarms for a participant during the first 30 minutes of each session. The alarm content was either study related or related to a session article. A participant provided feedback by selecting one of her preferred alarm-types whenever an alarm was generated. The study computer collected an average of 60 user-context exemplars from each participant during the four study sessions.

Figure 2.6 shown earlier in Section 2.2 presented the feedback user interface that a participant accessed to select one of the four alarm types. This feedback interface also displayed a *fortune-cookie* (similar to `fortune` on Unix/Linux) where a quote was displayed as an incentive for the participant to click on one of the alarm types. The alarm feedback request disappeared after 15 seconds to minimize user annoyance while requesting their feedback.

Table 6.1 presents the experimental design for the four study sessions. In this table, Column I shows the session numbers and Column II shows the article reading lengths. Column III shows the randomized alarm order. This randomized design controlled study variation, and I preserved the same randomized alarm order for all the study participants. The alarms $0, 1, 2$ and 3 correspond to no-alarm, visual alarm, voice alarm and both visual and voice, respectively. Column IV shows the four study conditions, that is the treatments applied to all study participants. This study applied Talk, Music, No-Music, and No-Talk as treatments.

For *Talk*, I used scripted talk to initiate conversation with a study participant. I played music which was selected through participant survey using a five point Likert scale with levels strongly dislike, dislike, neutral, prefer, and strongly prefer [42]. I labelled this as the *Music* treatment. *No-Talk* and *No-Music* corresponded to suspending conversation or music, respectively.

Table 6.1: Study 2: Google Calendar study's experimental design.

I Session	II Task	III Alarm Order	IV Conditions
1	short article	0, 2, 3, 1 3,1,2, 0 1, 0, 2, 3 2, 1, 0, 3	Talk, No-music Music, No-talk No-music, No-talk Music, Talk
2	long article	0,2,3,1 1, 2, 0, 3 1, 3, 2, 0 3, 0, 2, 1	Talk, No-music Music, No-talk No-music, No-talk Music, Talk
3	short article	1, 3, 0, 2 2, 3, 1, 0 2, 0, 3, 1 3, 2, 0, 1	Talk, No-music Music, No-talk No-music, No-talk Music, Talk
4	long article	0, 3, 2, 1 1, 0, 2, 3 1, 0, 2, 3 3, 0, 1, 2	Talk, No-music Music, No-talk No-music, No-talk Music, Talk

A self-report was a participant preferred alarm type, and the construct measured was the calendar generated alarm-type. This measurement ensured content validity by making the self-report and the measured construct the same. Participants were also secluded during the study duration to control any confounding study variables.

A participant read a short article in the first session, a longer article in the second session, an article of similar length in the third session, and finally a short article in the last session. A subsequent participant read a long, short, short, and long articles in the study. This variation thus counter balanced the article lengths for every participant pair to reduce the chances of article length order affecting alarm-type preferences. The findings from this study confirmed that user-context

significantly increased XCS alarm type prediction accuracy for multiple participants.

6.2 Predicting Google Calendar Alarm Preferences

Table 6.2: Study 2: Sycophant's test set performance on Google Calendar's 2Class alarm problem.

Learning → Algorithm Participant ↓	II Zero-R	III J48^{+uc}	IV J48^{-uc}	V XCS^{+uc}	VI XCS^{-uc}	VII XCS better than J48? (V > III)
1	0.5806	0.8871	0.8548	0.9530	0.9015	✓
2	0.5000	0.7407	0.7778	1.0000	0.9608	✓
3	0.7121	1.0000	1.0000	1.0000	1.0000	–
4	0.6200	0.9400	0.9400	1.0000	0.9213	✓
5	0.5455	0.8485	0.8485	0.8788	0.8030	✓
6	0.7143	0.8889	0.8889	0.9008	0.8349	✓
7	0.7460	0.9841	0.9048	1.0000	0.9833	✓
8	0.5625	0.7083	0.6875	0.8915	0.8301	✓
9	0.8163	0.9184	0.8980	1.0000	0.9792	✓
10	0.5246	0.6885	0.7377	0.9153	0.7025	✓
			4		9	9

Tables 6.2 and 6.3 present the test set predictive accuracy of Zero-R, J48, and XCS on the 2Class and 4Class Google Calendar alarm problems, respectively. Both tables list the participants in Column I and Column II shows Zero-R's test-set prediction accuracy. Column III shows J48's prediction accuracy with user-context features and Column IV shows J48's prediction accuracy when I removed user-context features from the same data. Similarly, Columns V and VI show XCS' prediction accuracy with user-context and without user-context, respectively. Column VII is the statistical inference that checks whether XCS outperformed J48 on the alarm prediction tasks. In the last row, the tables show the number of participants for whom removing user-context degraded J48's performance, the number of participants for whom removing user-context resulted in XCS' performance

Table 6.3: Study 2: Sycophant's test set performance on Google Calendar's 4Class alarm problem.

Learning → Algorithm Participant ↓	II Zero-R	III J48^{+uc}	IV J48^{-uc}	V XCS^{+uc}	VI XCS^{-uc}	VII XCS better than J48? (V > III)
1	0.5806	0.6129	0.5806	0.9697	0.9182	✓
2	0.5000	0.5185	0.5926	1.0000	0.9227	✓
3	0.7121	0.7347	0.7755	1.0000	1.0000	✓
4	0.6200	0.7400	0.6000	1.0000	0.9236	✓
5	0.5455	0.5152	0.5303	0.7879	0.4545	✓
6	0.7143	0.6825	0.7143	0.7047	0.6548	✓
7	0.7460	0.9683	0.8889	1.0000	0.9167	✓
8	0.5625	0.5625	0.5625	0.9556	0.9163	✓
9	0.8163	0.6818	0.7121	0.9702	0.9563	✓
10	0.5246	0.5574	0.4754	0.8833	0.5072	✓
			4		9	10

degradation, and the number of participants for whom XCS outperformed J48 on the alarm prediction tasks.

Table 6.2 shows that both J48 and XCS performed better than Zero-R's base rate performance on the 2Class problem. For 4 participants - 1, 7, 8, 9 (Column IV underlined values), J48's prediction accuracy degraded when I removed external user-context features. This performance degradation verified that external user-context (motion and speech) helped J48 to better learn these participant's preferences.

The underlined values in Column VI show that XCS' performance significantly degraded without external user-context for 9 out of 10 participants. XCS' performance remained the same for the remaining participant whose data indicated that he had no variation in alarm type preferences. For this participant, I concluded that the presence (or absence) of user-context had no effect on the alarm type prediction accuracy. The tally of checkmarks in Column VII shows that XCS significantly outperformed J48 for 9 participants who had varying alarm type preferences.

Table 6.3 shows a similar behavior of the three machine learning algorithms

on the 4Class problem of predicting one of the four calendar alarm types (none, visual, voice, and both visual and voice). Again, when I removed external user-context features, J48's alarm prediction accuracy degraded for four participants (1, 4, 7, 10) and XCS' performance degraded for 9 out of 10 participants.

To evaluate external user-context's influence for participants for whom removing this information degraded J48's performance, I next ranked J48's context-features using the Information-Gain Ratio (IGR) metric. IGR is a measure for evaluating the relevance of an attribute for a decision-tree, in Weka. I found that amongst the top 15 ranked features, participants 1, 4, 7, and 10 had more user-context features (related to motion or speech) when compared to the other participants. That is, IGR showed that external user-context features were important for these participants and hence the lack of this information negatively affected J48's prediction accuracy. The tally of checkmarks in Column VII also shows that XCS significantly outperformed J48 for all the 10 study participants. The study's results provided answers to some of the user-study rationale questions posed in Section 5.1.

6.3 Discussion

Consistent with the results from the first study, this short-term study's findings strongly supported my hypothesis of relying on user-context to better predict a participant's alarm type preference. The performance degradation of both J48 and XCS highlighted the importance of harnessing external user-related contextual information from a participant's environment to learn their preferences for different alarm types. This result answered the second *Relevance of User-Context* question that was posed in Section 5.1.

XCS' superior predictive performance across all participants when compared to the decision-tree learner established that it is viable technique for predicting application action preferences. This result answered the third *Robustness* study rationale question that was posed to select an appropriate machine learning scheme for personalizing applications.

The high predictive accuracies of XCS and the decision-tree learner across this

study's participants on both the 2Class and 4Class problems showed that my user-context based approach for learning user-preferences generalized across multiple participants. This result answered the *Generalizability across participants* question. [2]

Figure 6.2 presents the decision-tree generated for the ten Google Calendar study participants. I give this tree not for rule-interpretation but to illustrate that the tree selected the *user* (participant) attribute as the root-node (most important node), thereby showing that rules generated were in context of a *specific* user. Also, this figure illustrates the challenge faced when you attempt to interpret a complicated decision-tree for filtering out meaningful rules.

To simplify this decision-tree's rule interpretation, Figure 6.3 presents only a part of the decision-tree generated on the combined user-context data from this study's ten participants. Section 4.5.2 described in detail how to interpret rules from a decision-tree. R1 through R5 hightlight five example rules for this decision-tree. You can similarly parse the tree for other rules. This figure denotes No-Alarm, Visual-Alarm, Voice-Alarm, and Both alarm types by the values 0, 1, 2, and 3, respectively. Section 5.3 already explained these alarm types in detail. These user-dependent rules are a clear indication of Sycophant personalizing alarm-types to specific users.

Next, I found that for our study participants, each participant had a *different* set of rules predicting the *same* interface action preference. For example, R4 and R5 show two rules that predict alarm type-0 (not interrupting a user with an alarm). To interpret R4, start at the root node of the tree and traverse the tree along the dotted line towards a leaf node. When you traverse the tree and follow R4's path, note that this rule checks if user5's keyboard was active in the last minute (*KbdAny1 = 1*), whereas R5 checks for absence of motion (*motionAny1 = 0*) in user7's environment.

Continuing with personalization examples, for user4, R3's path shows that a visual alarm is generated if there was a lot of motion activity ($motionCount5 > 17$) along with the mouse being used in the last minute (*mouseAny1 = 1*) by this user. R2 shows that, with the mouse usage, user4 also preferred both visual and voice

[2] Section 5.1 lists all the study rationale questions.

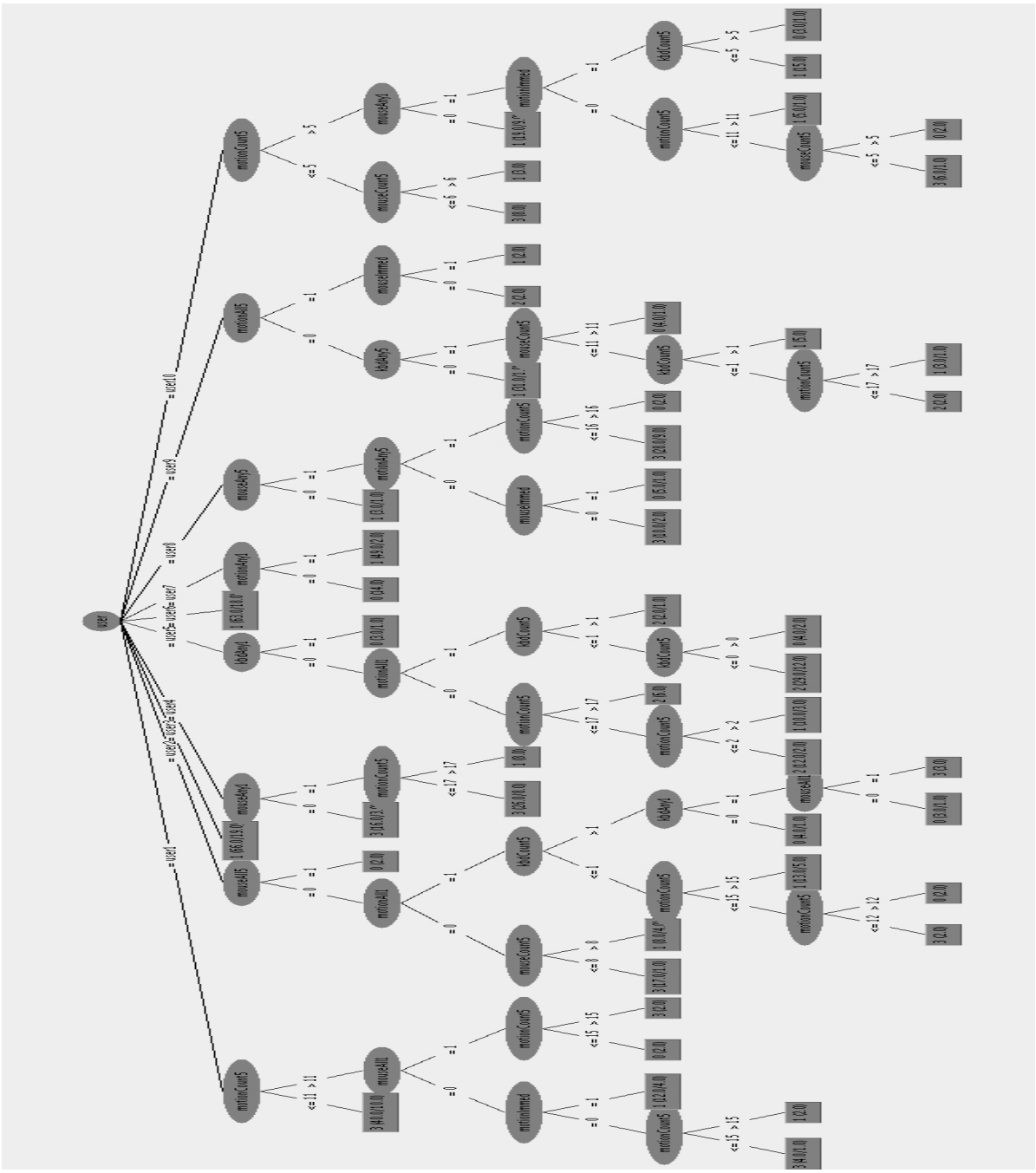

Figure 6.2: A decision-tree generated for Google Calendar study participants showing the personalization of alarms to an individual participant.

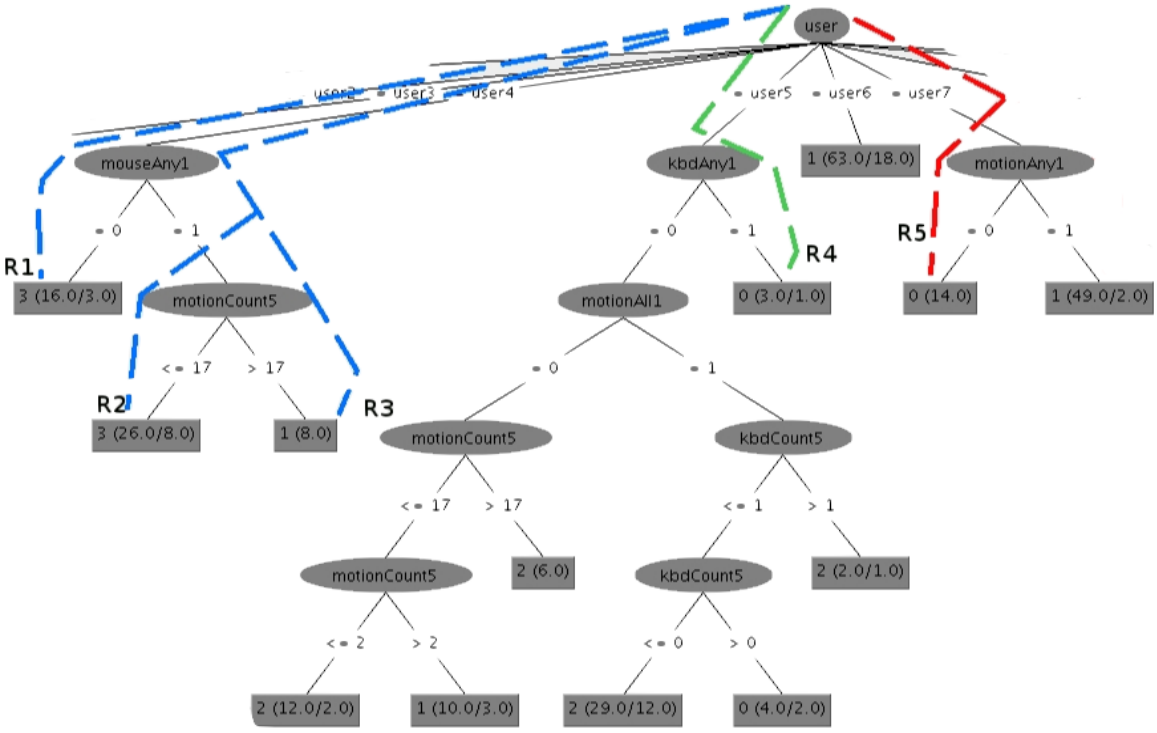

Figure 6.3: Example rules that show alarm type personalization.

alarms if the motion activity (*motionCount5* < 17) was below a certain threshold. For R1, note that user4 also preferred both visual and voice alarms if there was no mouse activity in the last minute (mouseAny1 $= 0$). These results show that every user is different and may have *different* preferences for the *same* interface actions and hence the need to personalize applications to individual users.

The user-context features present at different nodes in the decision-tree emphasize the importance of external user-context (speech derived features in this case) to predict an appropriate alarm for these three users. I found similar evidence of personalization to individual users where user-context has played a significant role.

6.4 Chapter Summary

Consistent with the first study's results that established the feasibility of predicting user-preferred alarm types for three pilot study users, results of this second study validated the necessity of user-context for XCS to better predict alarm types for multiple users. Results also showed Sycophant's generalizability across multiple participants. The decision tree that generated alarms specific to individual participants verified Sycophant's personalization for alarm types. In my next study, I examined whether Sycophant could context-enable Winamp and thereby evaluate the generalizability of my framework for supporting multiple desktop applications.

Chapter 7

Study 3: Winamp

The two user studies results in the previous chapters established the relevance of user-context to predict interface-action preferences and the generalizability of this preference learning across multiple participants. Informed by the study rationale outlined in Section 5.1, I next evaluated Sycophant's generalizability across multiple desktop applications.

In my third user study I investigated whether Sycophant could context-enable Winamp (a popular media player) in addition to Google Calendar and thereby verify the generalizability of my user-context learning approach to another desktop application [6]. This chapter starts with an overview of the third study's design which was similar to the second study's design. I next give results which show that Sycophant successfully predicted a participant's Winamp action preferences. Before concluding this chapter, I examine a few rules that demonstrate Sycophant's personalization to individual study participants and its generalization across another desktop application. Finally, I motivate the last user study to test whether my user-context based approach for learning interface action preferences is successful over long-term application use.

7.1 Study Design

Similar to the second study, 10 graduate students in the computer science and engineering department participated in this short-term user study. In this study I simulated graduate students reading an article, listening to music on Winamp while having short conversations or leaving their desk for various reasons. The study's objective was to periodically gather user-context during each session along with a user's Winamp action preference. Since a Winamp action was explicitly specified by a participant for each of the training exemplars, I could frame this action prediction as supervised learning task for Sycophant. On this task, I investigated whether user-context helped Sycophant to accurately predict a participant's preferred Winamp action types.

Table 7.1 shows the experimental design for four study sessions. The independent variable in this study was a Winamp action. This table shows the session number in Column I. Column II shows a session article length, and the study conditions (treatments) applied to all study participants is show in Column IV. The *Talk* treatment (scripted talk) was the same as in the previous short-term study with Google Calendar. The article lengths were counter-balanced for every participant pair and the same treatments applied to all the 10 participants.

The ten study participants completed a survey questionnaire that listed songs and instructed them to indicate their song preference based on a five point *Likert* scale (strongly dislike, dislike, neutral, prefer, and strongly prefer) [42]. After the 10 participants anonymously completed this survey, I chose songs that all the participants preferred at the neutral level or higher. Participants listened to these songs during their four study sessions. This study maintained the article reading and question answering session times the same as the previous study.

Each participant attended four separate sessions and each session lasted 45 minutes. While a participant read an article, she was interrupted with talk (using a script) and/or made to leave the study area to perform a place-location task on a map. Participants left their desk to find a place on a map of Yosemite National Park. I labelled this as the *MapReading* treatment (Column III) in the study design.

Table 7.1: Study 3: Winamp study's experimental design.

I Session	II Task	III Conditions
1	short article	No Talk, No Map Reading No MapReading, Talk Map Reading, Talk MapReading, No Talk
2	long article	No Talk, No Map Reading No MapReading, Talk Map Reading, Talk MapReading, No Talk
3	long article	No Talk, No Map Reading No MapReading, Talk Map Reading, Talk MapReading, No Talk
4	short article	No Talk, No MapReading No MapReading, Talk Map Reading, Talk MapReading, No Talk

During these interruptions, my hypothesis was that different participants tend to have different preferences for Winamp.

To capture these preferences, Sycophant periodically generated requests (every 70 seconds) and solicited participant feedback for one of Winamp's four actions (pause a song, play a song, increase volume by 10 percent, and decrease volume by 10 percent). Sycophant's feedback request interface for recording a participant's feedback was similar to the one shown earlier in Figure 2.6. A participant selected one of Winamp's four interface actions listed in the parentheses above, instead of selecting one of the four alarm types. Similar to the previous study, the system again collected an average of 60 user-context exemplars (user feedback) for every participant during the four study sessions.

During the last 15 minutes of a 45 minute study session, a participant answered questions related to the article without being interrupted. The system recorded a participant's Winamp action preference, motion activity, speech activity, keyboard activity, and mouse activity as contextual information from their desktop environment. From this gathered contextual data, I expected to find a mapping from user-context features to a participant's Winamp action preference.

7.2 Predicting User Preferences for Winamp

Table 7.2 presents the test-set prediction accuracy of three machine learners for one of the four Winamp actions: play, pause, increase volume by 10 percent, and decrease volume by 10 percent. This table lists the participants in Column I. Col-

Table 7.2: Study 3: Test set performance of different machine learners for predicting one of Winamp's four actions (play, pause, increase volume, decrease volume).

Learning → Algorithm Participant ↓	II Zero-R	III J48^{+uc}	IV J48^{-uc}	V XCS^{+uc}	VI XCS^{-uc}	VII XCS better than J48? (V > III)
1	0.4940	0.4940	0.4337	0.5185	0.3951	✓
2	0.5000	0.4630	0.5556	0.7113	0.3442	✓
3	0.6136	0.5682	0.6136	0.7897	0.3333	✓
4	0.5306	0.5102	0.4082	0.7520	0.7122	✓
5	0.8061	0.7653	0.8061	1.0000	0.6559	✓
6	0.7111	0.6889	0.7111	0.8790	0.5801	✓
7	0.7733	0.7733	0.7733	0.9321	0.8812	✓
8	0.7315	0.6944	0.7315	0.7660	0.4926	✓
9	0.6727	0.6727	0.6545	0.8519	0.7593	✓
10	0.5800	0.6200	0.5800	0.8741	0.5667	✓
			4		10	10

umn II shows ZeroR's performance. Columns III and IV show J48's performance with and without user-context features, respectively. Similarly Columns V and VI show XCS' performance with and without user-context. Column VII tests (two sample t-test with 95 percent confidence interval) whether XCS outperformed J48

for predicting a participant's Winamp action preferences.

When I compared the prediction accuracies of J48 (Column III) with ZeroR (Column II), I observed that the decision-tree's (J48) performance was worse than the base-rate for 9 out of 10 participants. However, when I compared XCS' performance (Column V) with that of ZeroR (Column II) , unlike J48, I noted that XCS outperformed Zero-R for all 10 study participants. Next, when I compared XCS' performance (Column V) with that of J48 (Column III), I observed that XCS significantly outperformed J48 for all our 10 study participants. XCS' superior predictive accuracy demonstrated that it is a more robust learner when compared to either Zero-R or a decision-tree learner for predicting participant preferred Winamp actions.

Analogous to the previous study with Google Calendar, when I removed motion and speech derived features, the decision-tree's performance again degraded for four participants (1, 4, 9, 10) thereby emphasizing the importance of external user-context (motion and speech) for learning these participants' preferences. Also, when I removed external user-context XCS' Winamp action predictive accuracy significantly degraded for all study participants. This result again emphasized user-context's importance for an application to better predict user preferences. Thus Sycophant successfully context-enabled Winamp and predicted individual participant's Winamp action preferences. The decision-tree generated for all study participants indicated Winamp action personalization to an individual participant.

7.3 Discussion

Figure 7.1 presents the decision tree generated for the combined data of all the participants in this study. Again, I present this tree not for rule-interpretation but to illustrate that Sycophant personalized its Winamp action prediction to individual study participants. Similar to the study with Google Calendar, the user (participant) attribute at the root node of the tree showed that for most of the participants, if they had preferences, each participant had a different set of rules predicting the

same interface action preference. In addition to presenting this tree to show personalization to individual study participants, I present this tree to highlight the complexity of interpreting meaningful rules from a complex decision-tree.

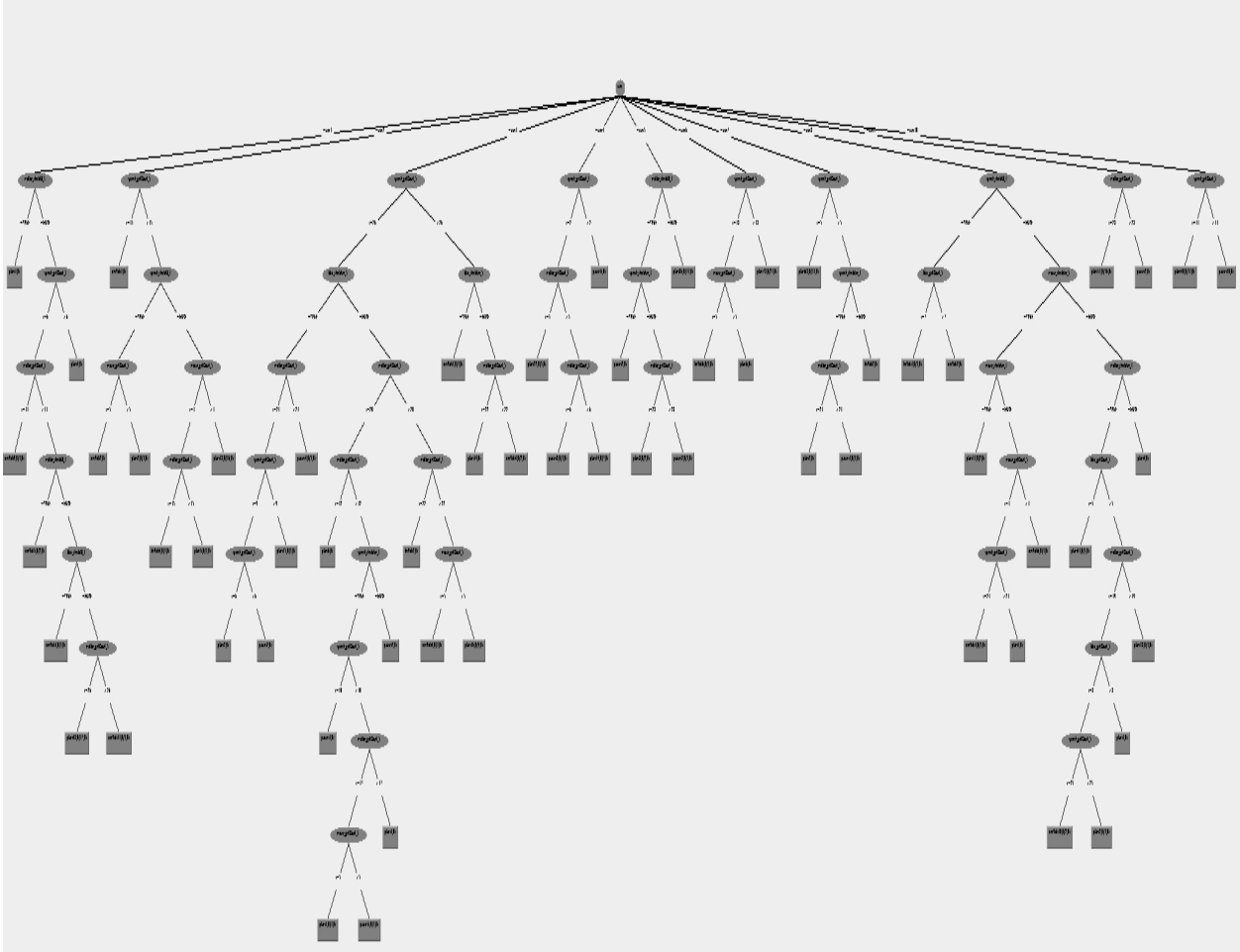

Figure 7.1: A decision-tree generated for Winamp study participants showing the personalization of Winamp's actions (play, pause, increase volume, decrease volume) to an individual participant.

For rule-interpretation, Figure 7.2 presents a part of the decision-tree generated on the combined data from all participants. This tree shows Winamp action personalization to three study participants. For example, I found that the rules predicting user4's preference for *pausing* Winamp's were different from the rules predicting user5's preference for the same Winamp action. $R1$ shows that

the decision-tree paused Winamp if the speech activity was above a threshold for user4 (speechGetCount5 > 2), whereas motion (motionCheckAll) and speech (speechCheckAny1) activity in the last minute were checked to pause Winamp for user5.

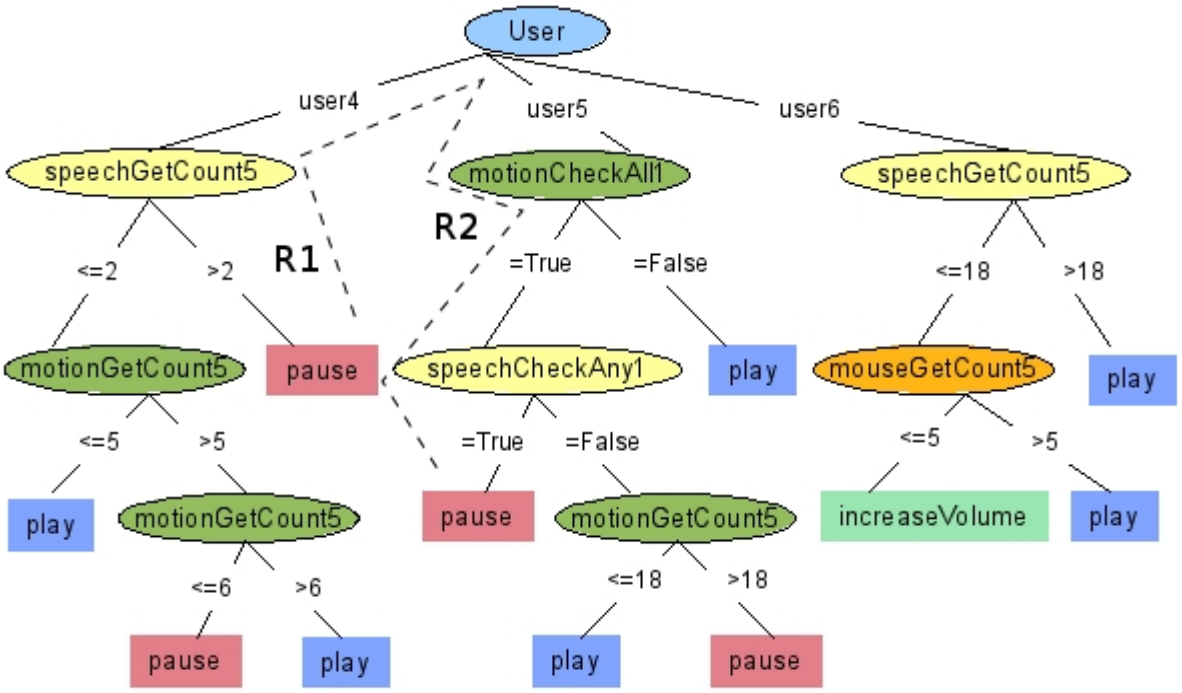

Figure 7.2: A part of the complete decision-tree generated for Winamp study. This figure shows a rule that predicts a Winamp action for participants 4, 5, and 6

That is, again the rules generated were in context of a *specific* user, thereby demonstrating that user-context helped Sycophant to successfully personalize Winamp to a particular user. You can note similar rules based on user-context features for other Winamp actions for these three users. These results confirmed that Sycophant had successfully generalized its interface action prediction to another desktop application, Winamp. Results of this study answered the *Generalizability across applications* question (4-b) of Section 5.1.

7.4 Chapter Summary

In this third user study I investigated whether Sycophant could context-enable Winamp for multiple study participants. Results here confirmed that user-context helped Sycophant to better predict Winamp action preferences for all the ten study participants and again highlighted XCS' feasibility on this action prediction task. Encouraged by the success of our short-term studies, I next assessed Sycophant's user-context based personalization approach over long-term use for Google Calendar. The next chapter presents results from this long-term study that further substantiated my hypothesis of harnessing user-related contextual information to personalize desktop applications.

Chapter 8

Study 4: Long-term Study

While the user studies' results in the last three chapters established that my user-context based approach personalized application actions over short-term application use, I had yet to verify Sycophant's generalizability over long-term use for multiple participants. This chapter presents long-term preference learning results from the last user study with Google Calendar.

8.1 Study Procedure and Results

I deployed Sycophant among three participants who set their appointments in context-enabled Google Calendar for 2-4 weeks. During calendar use, I investigated whether Sycophant's generalized its application action prediction for these users. This was a field study where participants used Google Calendar in their work environment. I instructed the participants to set 3-7 appointments per day for their appointments. All study participants set appointments for their daily or weekly repeated activities. Each participant provided an average of 75 user-context exemplars during the data collection phase. Specifically, participants 1, 2, and 3 provided 40, 85, and 100 exemplars, respectively.

Tables 8.1 and 8.2 presents the test set prediction accuracy of ZeroR, J48, and XCS on Google Calendar's 2Class and 4Class problems, respectively. In both tables, Column I lists the participants and Column II gives the prediction accuracy

Table 8.1: Study 4: This table shows the long-term predictive accuracies of XCS and J48 on Google Calendar's 2Class alarm problem.

Learning → Algorithm Participant ↓	II ZeroR	III J48^{+uc}	IV J48^{-uc}	V XCS^{+uc}	VI XCS^{-uc}	VII XCS better than J48? (V > III)
1	0.8531	0.9247	0.7849	0.6422	0.6422	-
2	*0.7837*	0.8378	*0.7837*	0.8611	0.8611	✓
3	*0.9047*	0.8333	*0.9047*	0.8343	0.8343	✓

of Zero-R, the base-rate. Columns III and IV show the decision-tree learner's (J48) performance with and without user-context features, respectively. Similarly Columns V and VI show XCS' performance with and without user-context.

On the 2Class problem, note from Table 8.1 that the decision-tree (Column III) outperformed ZeroR (Column II), the base-rate, for two participants while XCS (Column IV) outperformed the base-rate for one participant. Removing user-context degraded J48's performance for two participants (underlines values in Column IV) while XCS' predictive accuracy remained the same for all the three particpants with no user-context. Column VII shows that XCS outperformed the decision-tree learner for two participants. For participant-3, removing user-context resulted in J48's performance degrading to that of ZeroR despite the increase in prediction accuracy. The tree for this participant collapsed and J48 predicted a default visual alarm without user-context for this partipant, thereby losing its context-based personalization.

On the 4Class problem, in Table 8.2 comparing Columns III and V with Column II shows that both XCS and the decision-tree learner outperformed ZeroR for all three participants. Columns VII shows that XCS significantly outperformed the decision-tree learner for all three participants. Additionally, Columns IV and VI show that removing user-context degraded the predictive accuracy of both XCS and J48 for all three participants.

Table 8.2: Study 4: This table shows the long-term predictive accuracies of XCS and J48 on Google Calendar's 4Class alarm problem.

Learning → Algorithm Participant ↓	II ZeroR	III J48^{+uc}	IV J48^{-uc}	V XCS^{+uc}	VI XCS^{-uc}	VII XCS better than J48? (V > III)
1	0.3097	0.6106	0.4778	0.7232	0.3838	✓
2	0.4237	0.5762	0.4237	0.6422	0.6366	✓
3	0.3333	0.5556	0.3333	0.5925	0.3703	✓

8.2 Discussion

The long-term study results were still consistent with the results from earlier studies. Sycophant's behavior on the 4Class problem was similar to the previous two short-term studies with Google Calendar and Winamp. Even though Sycophant's behavior on the 2Class problem also matched the previous studies' results, collecting additional calendar usage data from the three participants would enable a deeper analysis of the performance of the three machine learners. More data would lead to a better understanding of XCS' lower predictive accuracy on the 2Class problem for participant-1 since this participant provided fewer data when compared to the other two participants.

Figure 8.1 presents the decision-tree learned for all three study participants. I do not present this tree for detailed rule-interpretation but to highlight that this tree was similar to the trees generated for all participants in the previous two short-term studies. This decision-tree verifies that Sycophant personalized alarm types to an individual participant.

Figure 8.2 presents a part of the tree generated on the combined data from all three participants. In this tree, the no-alarm, visual, voice, and both alarm types correspond to classes 0, 1, 2, and 3, respectively. Rule R1 checks if the speech activity was above a certain threshold (speechGetCount5 > 9) and predicts a voice alarm for participant-1 (p1) if the motion activity's threshold was greater than 10

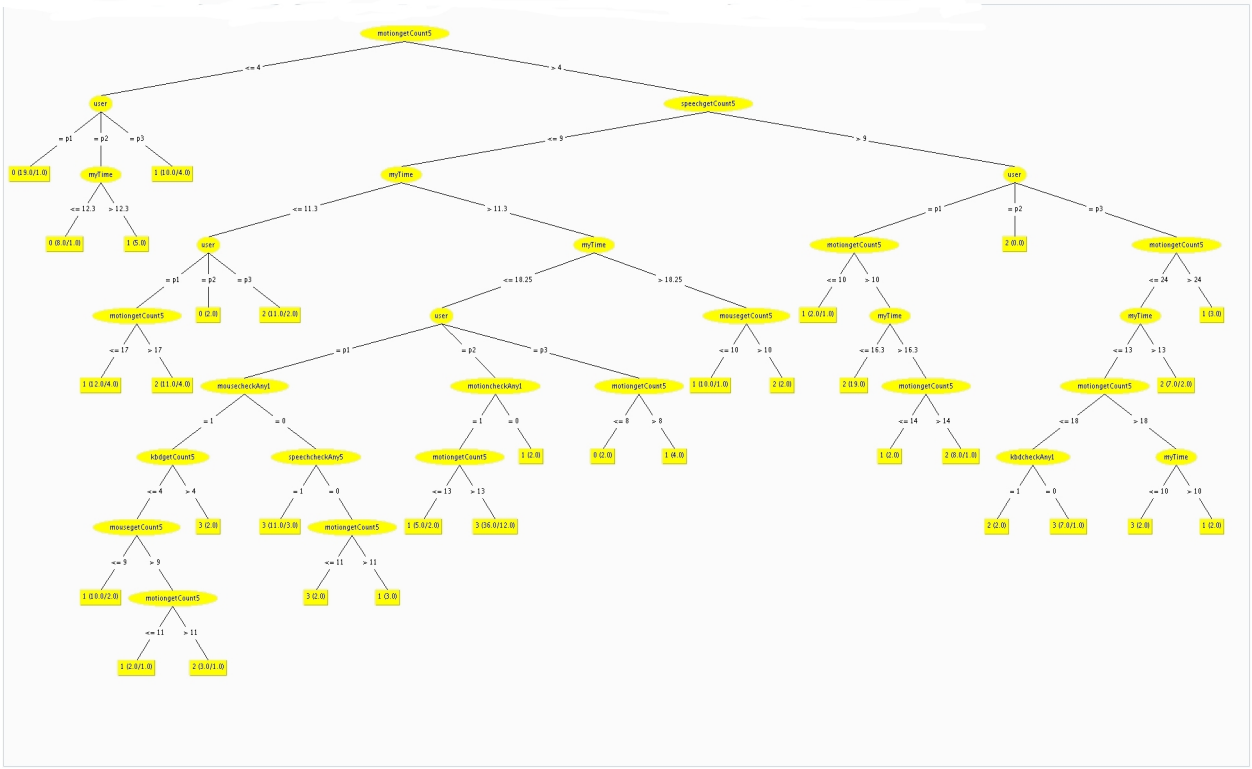

Figure 8.1: A decision-tree generated for the long-term study participants showing Google Calendar alarm types personalization to an individual participant.

(motiongetCount5 > 10). R2 shows that participant-2 preferred a voice alarm if the speech threshold was greater than 9 five minutes before alarm generation time. R3 shows that unlike participant-1, this user preferred a visual alarm if the motion threshold in the last five minutes was greater than 24. Thus, Sycophant generates alarm types based on both a user and external user-context and personalized its alarm generation to an individual participant.

This final study again provided evidence supporting my user-context based approach to personalize an application and demonstrated XCS' feasibility for this action prediction task. Further, the degradation of both the machine learners predictive accuracy after deleting user-context emphasized the necessity of user-related contextual information to predict user-preferred actions. The results of this study

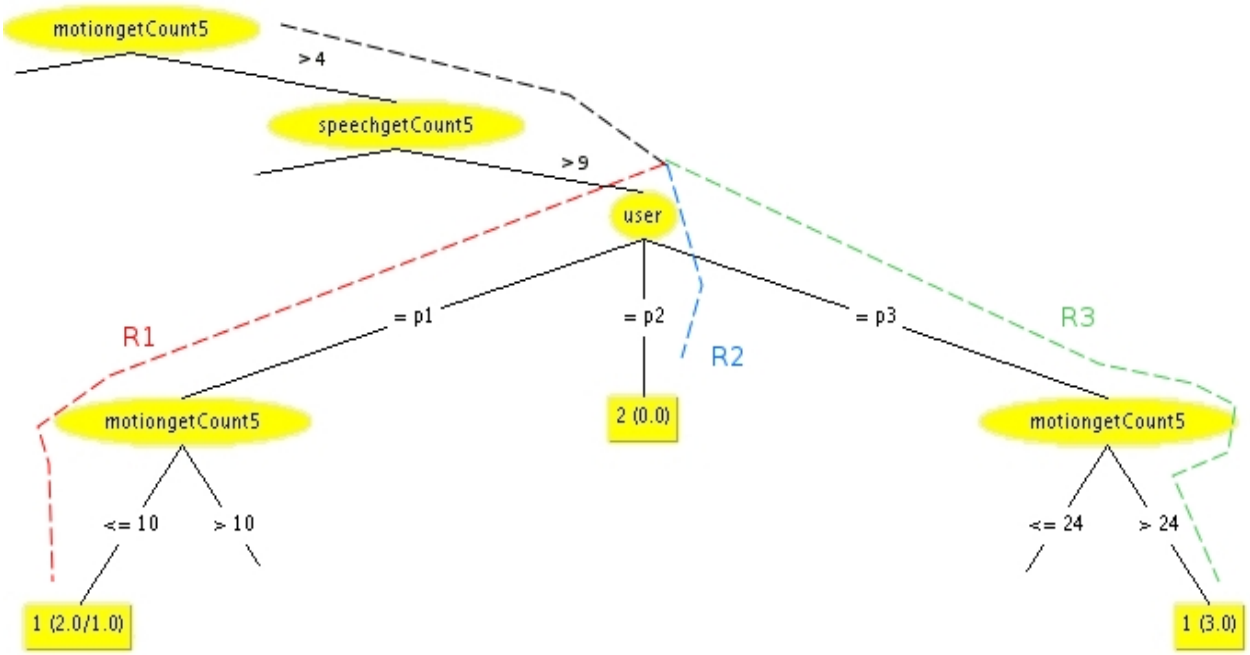

Figure 8.2: A part of the complete decision-tree generated for long-term study participants showing the personalization of Google Calendar alarm types to participants 1, 2, and 3

answered the final *Generalization over duration of use* question that I posed in Section 5.1.

8.3 Chapter Summary

The four user studies results successfully answered all the questions posed in this dissertation. Sycophant context-enabled desktop applications and personalized their actions towards individual study participants. XCS enhanced Sycophant's prediction accuracy on this action prediction tasks over both short-term and long-term application use. Not only did my user studies provide evidence to support this dissertation's main hypothesis of harnessing external user-context to adapt application behavior towards individual users, these user studies also suggested promising avenues for future research.

Chapter 9

Conclusions and Future Work

Modern desktop applications weakly personalize their actions towards individual users due to their reliance on just keyboard activity, mouse movement, and clock information. My four user studies showed that harnessing information from a user's environment (user-context) is one solution to address this need for application personalization. Sycophant, my generalized user modeling framework, successfully adapted the behavior of a calendar and a media player and tailored their interface actions to an individual user's preferences.

While my initial pilot study established the feasibility of predicting a user-preferred application action type, three issues demanded further investigation. The first issue was to verify that user-context could improve action type prediction accuracy. Confirming this prediction generalization across multiple participants was the second issue, and the third was to verify Sycophant's generalizability across different desktop applications. Subsequently, three studies addressed these questions and the system generated user models demonstrated that Sycophant generated application actions *specific* to a particular user. These user studies' results also confirmed that application action personalization generalized across multiple users and desktop applications over both short-term and long-term application use.

Four user studies substantiated this dissertation's central claim of using external user-context for enabling applications to better predict user preferred actions

thereby leading to enhanced application personalization. Before outlining some promising avenues for future work, I next summarize this work's main contributions.

9.1 Main Contributions

This dissertation proposed harnessing user-related contextual information from a desktop's external environment to personalize applications to individual users. I then designed my user modeling framework, Sycophant, as a test-bed to evaluate this hypothesis.

First, this thesis identified the potential of leveraging user-related information from a desktop's environment (user-context) to address the lack of personalization in current desktop applications. Next, I designed Sycophant, my modular, flexible, user modeling framework, to harness user-context and adapt application behavior to an individual user. Then, this work gave a step-wise procedure that illustrated the use of my User-Context API (C-API) to context-enable a calendar.

Using C-API, I conducted a pilot study with three participants. Sycophant's alarm type prediction accuracy of 88 percent verified the feasibility of predicting one of the four calendar alarm types. Extending previous interruption based studies, these results also showed that Sycophant could not only predict *when* but also how to interrupt each participant.

Building upon these initial results, my second short term study with Google Calendar further confirmed that this alarm type preference learning generalized across ten participants. Sycophant successfully predicted these participants' preferences for alarm types with an average predictive accuracy of 90 percent.

Consistent with the second study's results, the results of the third user study with Winamp (a media player) established that Sycophant could predict a participant-preferred Winamp action type. Sycophant predicted one of the four Winamp's actions (play, pause, increase volume, decrease volume) with an average accuracy of 80 percent. Thus, Sycophant not only personalized interface actions for multiple participants in the four user studies but also accomplished this personalization for

Google Calendar and Winamp. These evaluation results provided supporting evidence for the generalizability of my user modeling framework across participants and applications over both short-term and long-term application use.

While the previous studies evaluated Sycophant's action-prediction over short-term application use, the fourth study's results verified Sycophant's alarm type personalization over long-term use by three participants. Sycophant's average alarm type predicting accuracy was 65 percent.

In all these user studies, removing external user-context features from participant data degraded the performance of a learning classifier system (XCS) and a decision-tree learner (J48). This performance degradation confirmed that external user-context increased the interface action prediction accuracy of these machine learners. This finding emphasized the need for user-context based approaches to personalize desktop applications.

This dissertation used XCS, as a predictive data mining technique, for learning user preferred application action types and showed that XCS consistently outperformed a widely used decision-tree learner. This work thus highlighted the promise of a learning classifier systems based approach for user interface personalization and further established XCS' applicability for solving real world problems.

The relative ease of using C-API to support research and development in context-aware desktop applications highlights some promising directions for future work.

9.2 Future Work and Conclusion

The work presented in this dissertation suggests a variety of future work. In terms of research, Sycophant can be deployed to conduct additional long-term user studies. User-context data gathered from multiple desktop applications across different user populations can provide more insight about predicting user preferred actions. Sycophant's generalizability can be further evaluated across applications and user groups.

I developed C-API iteratively based on user feedback during Sycophant's deployment during different phases of my research. Like any software, this API can

be refined further and more rigorously tested. Even though C-API's design was not specific to any operating system, its use was mostly tested on Win32 platforms. Modifying C-API and testing it on Linux/MAC platforms to reach a wider user population would provide avenues for context-enabling desktop applications on these two platforms.

C-API's learning services layer supports methods to easily select any machine learning algorithm for predicting user preferred application actions. Sycophant's action prediction task can be cast as problem of predicting a user's behavior in a dynamic environment (where user preferences are contextual). Predicting user-preferred application action types through trial-and-error can then be considered as reinforcement learning problem [62]. Reinforcement learning provides a wide variety of learning solutions that can be evaluated for desktop application personalization.

C-API can be incorporated as a thin layer within an operating system. C-API's services could then facilitate the construction of new context sensors and features. These new CPI services would support application programmers develop and deploy novel context aware desktop applications.

Conducting user studies that evaluate a desktop's personalization based on task-related data and user-context is another avenue for extending my research. Merging task-aware user interfaces research with my user-context based approach would collect richer information related to a user's primary task and her external environment. Application personalization can then be evaluated based on these two data sources.

The latest advances in customizable hardware (community electronics) would also enable the creation of new context sensors. For example, *Bug Labs* provides hardware with electronic modules that can be used to build context sensors [1]. *BUGview* is an LCD screen with a touch sensitive interface that functions both as a display and an input device [2]. Instead of using a pop-up interface, Sycophant can solicit user feedback through BUGview thereby ameliorating some of the annoyance caused by pop-up feedback. Wider adoption of such community electronics devices can lead to novel user-context sensors for richer personalization.

Learning user-activity patterns through applications' use could also enrich existing computer security methods. For example, if a system discerns that there is motion in Jack's environment after 9 A.M. during weekdays, Jack's computer can generate a security notification if it detects motion in Jack's environment at 6 A.M. during the weekend. My investigation into context-based user preference learning approaches also has the potential to address the needs of non-traditional users. Such user preference learning can improve accessibility for visual/hearing impaired users, senior citizens and young adults. For example, graphical user interfaces could automatically learn the position and size of interactive elements and improve user experience for these under-represented user communities.

The context based user modeling framework outlined in this dissertation is one approach for designing interfaces that adapt their actions depending on the context of use. Sycophant is a significant advance towards systems that tune their behavior based on a user and her environment. Sycophant's success in personalizing desktop applications provides the impetus to harness external contextual confirmation to improve human-computer interaction and thereby ensure broader end-user acceptance for personalized desktop applications.

Bibliography

[1] Bug labs (accessed may. 10, 2008). http://www.buglabs.net.

[2] Bugview (accessed may. 10, 2008). http://www.buglabs.net/modules/bugview.

[3] Open source initiative (accessed may. 10, 2008). http://www.opensource.org.

[4] Spss data mining tool: Clemenentine (accessed may. 10, 2008). http://www.spss.com/clementine/index.htm.

[5] Sycophant (accessed may. 10, 2008). http://www.cse.unr.edu/~syco.

[6] Winamp media player (accessed may. 10, 2008). http://www.winamp.com.

[7] Object Management Group, Unified Modeling Langauge, April 10, 2007. http://www.uml.org.

[8] Piotr D. Adamczyk and Brian P. Bailey. If not now, when? the effects of interruption at different moments within task execution. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 271–278, New York, NY, USA, 2004. ACM.

[9] Ethem Alpaydin. *Introduction to Machine Learning*. Cambridge University Press, New York, NY, USA, 2005.

[10] Jim Arlow and Ila Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

[12] A.J. Bagnall and G.C. Cawley. Learning classifier systems for data mining: a comparison of xcs with other classifiers for the forest cover data set. *Proceedings of the International Joint Conference on Neural Networks*, 3:1802–1807 vol.3, 20-24 July 2003.

[13] Brian P. Bailey, Piotr D. Adamczyk, Tony Y. Chang, and Neil A. Chilson. A framework for specifying and monitoring user tasks. *Computers in Human Behavior*, 22:709–732, July 2006.

[14] Brian P. Bailey and Joseph A. Konstan. On the need for attention-aware systems: Measuring effects of interruption on task performance, error rate, and affective state. *Computers in Human Behavior*, 22:685–708, July 2006.

[15] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[16] J.A. Blackard and D.J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. In *Computers and Electronics in Agriculture*, pages 131–151, 1999.

[17] P. J. Brown. The stick-e document: a framework for creating context-aware applications. In *Proceedings of EP'96, Palo Alto*, pages 259–272. also published in it EP–odd, January 1996.

[18] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware applications: from the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.

[19] Martin V. Butz. XCSJava 1.0: An Implementation of the XCS classifier system in Java. Illinois Genetic Algorithms Laboratory (IlliGAL) . Technical Report 2000027, 2000.

[20] Chih C. Chang and Chih J. Lin. *LIBSVM: a library for support vector machines*, 2001.

[21] Peter Clark and Tim Niblett. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[22] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury, 1999.

[23] Anind K. Dey, Gregory D. Abowd, Mike Pinkerton, and Andrew Wood. Cyberdesk: A framework for providing self-integrating ubiquitous software services. In *ACM Symposium on User Interface Software and Technology*, pages 75–76, 1997.

[24] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware appplications. *Human Computer Interaction*, 16, 2001.

[25] Anton N. Dragunov, Thomas G. Dietterich, Kevin Johnsrude, Matthew McLaughlin, Lida Li, and Jonathan L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82, New York, NY, USA, 2005. ACM.

[26] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, November 2000.

[27] James Fogarty. Constructing and evaluating sensor-based statistical models of human interruptibility, February 2006.

[28] James Fogarty and Scott E. Hudson. Toolkit support for developing and deploying sensor-based statistical models of human situations. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 135–144, New York, NY, USA, 2007. ACM.

[29] James Fogarty, Scott E. Hudson, and Jennifer Lai. Examining the robustness of sensor-based statistical models of human interruptibility. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 207–214, New York, NY, USA, 2004. ACM.

[30] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: A statistical view of boosting. *Technical report,Department of Statistics, Stanford University*, 1998.

[31] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[32] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning, Chapter 6, Introduction to Genetics-Based Machine Learning, pp 217-259*. Addison-Wesley, Reading, MA, 1989.

[33] D. Heckerman. A tutorial on learning with bayesian networks, 1995.

[34] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[35] Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90, 1993.

[36] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265, Madison, WI, July 1998.

[37] Eric Horvitz and Johnson Apacible. Learning and reasoning about interruption. In *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*, pages 20–27, New York, NY, USA, 2003. ACM.

[38] Shamsi T. Iqbal and Brian P. Bailey. Understanding and developing models for detecting and differentiating breakpoints during interactive tasks. In *CHI*

'07: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 697–706, New York, NY, USA, 2007. ACM.

[39] Shamsi T. Iqbal and Eric Horvitz. Disruption and recovery of computing tasks: field study, analysis, and directions. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 677–686, New York, NY, USA, 2007. ACM.

[40] Tim Kovacs. *XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions.*, pages 59–68. Springer-Verlag, August 1997.

[41] Tim Kovacs. *Bibliography of Real-World Classifier Systems Applications*, volume 150, pages 300–305. Springer, April 2004.

[42] Rensis Likert. *A technique for the measurement of attitudes*. New York, New York, 1932.

[43] Xavier Llora and Josep M. Garrell. Knowledge-independent data mining with fine-grained parallel evolutionary algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 461–468, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[44] Sushil J. Louis and Anil Shankar. Context learning can improve user interaction. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, las Vegas Hilton, Las Vegas, NV USA*, pages 115–120, 2004.

[45] Tom M. Mitchell. *Machine Learning*, chapter 2, pages 32 – 34. McGraw-Hill, March 1997.

[46] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, March 1997.

[47] Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *International Semantic Web Conference*, pages 92–99, 1998.

[48] Michael J. Pazzani and Dennis F. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[49] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1992.

[50] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, 2nd Edition*. Pearson Higher Education, 2004.

[51] Shaun Saxon and Alwyn Barry. XCS and the monk's problems. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, page 809, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[52] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[53] Anil Shankar. Simple user-context for better application personalization. Master's thesis, University of Nevada, Reno, 2006.

[54] Anil Shankar and Sushil J. Louis. Better personalization using learning classifier systems. In *Proceedings of the 2005 Indian International Conference on Artificial Intelligence, December 20-22 2005, Poona, India*, 2005.

[55] Anil Shankar and Sushil J. Louis. Learning classifier systems for user context learning. In *2005 IEEE Congress on Evolutionary Computation,September 2-5 2005, Edinburgh, UK*, 2005.

[56] Anil Shankar, Sushil J. Louis, Sergiu Dascalu, Linda J. Hayes, and Ramona Houmanfar. User-context for adaptive user interfaces. In *IUI '07: Proceedings*

of the 12th international conference on Intelligent user interfaces, pages 321–324, New York, NY, USA, 2007. ACM.

[57] Anil Shankar, Sushil J. Louis, Sergiu Dascalu, Ramonah Houmanfar, and Linda J. Hayes. Xcs for adaptive user-interfaces. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1876–1876, New York, NY, USA, 2007. ACM.

[58] Anil Shankar, Juan Quiroz, Sergiu M. Dascalu, Sushil J. Louis, and Monica N. Nicolescu. Sycophant: An api for research in context-aware user interfaces. In *ICSEA '07: Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007)*, pages 83/1–6, Washington, DC, USA, 2007. IEEE Computer Society.

[59] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human Computer Interaction*. Wiley, March 2007.

[60] Jianqiang Shen, Lida Li, Thomas G. Dietterich, and Jonathan L. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 86–92, New York, NY, USA, 2006. ACM.

[61] Stewart W. Wilson. *Classifier Systems and the Animat Problem*. Number 3 in Machine Learning. CCSDS, Hingham, MA, USA, 1987.

[62] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[63] Stewart W. Wilson. Mining oblique data with XCS. *Lecture Notes in Computer Science*, 1996.

[64] Stewart W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

[65] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[66] Stewart W. Wilson. Generalization in the XCS classifier system. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E.Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[67] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, USA, 2000.