AN ABSTRACT OF THE DISSERTATION OF

Xinlong Bao for the degree of Doctor of Philosophy in Computer Science presented on August 24, 2009.

Title:

Applying Machine Learning for Prediction, Recommendation, and Integration

Abstract approved: ______________________________

Thomas G. Dietterich

This dissertation explores the idea of applying machine learning technologies to help computer users find information and better organize electronic resources, by presenting the research work conducted in the following three applications: FolderPredictor, Stacking Recommendation Engines, and Integrating Learning and Reasoning.

FolderPredictor is an intelligent desktop software tool that helps the user quickly locate files on the computer. It predicts the file folder that the user will access next by applying machine learning algorithms to the user's file access history. The predicted folders are presented in existing Windows GUIs, so that the user's cost for learning new interactions is minimized. Multiple prediction algorithms are introduced and their performance is examined in two user studies.

Recommender systems are one of the most popular means of assisting internet users in finding useful online information. The second part of this dissertation presents a novel way of building hybrid recommender systems by applying the idea of Stacking from ensemble learning. Properties of the input users/items, called runtime metrics, are employed as additional meta features to improve performance. The resulting system, called STREAM, outperforms each component engine and a static linear hybrid system in a movie recommendation problem.

Many desktop assistant systems help users better organize their electronic resources by incorporating machine learning components (e.g., classifiers) to make intelligent predictions. The last part of this dissertation addresses the problem of how to improve the performance of these learning components, by integrating learning and reasoning through Markov logic. Through an inference engine called the PCE, multiple classifiers are integrated via a process called relational co-training that improves the performance of each classifier based on information propagated from other classifiers.

Applying Machine Learning for Prediction, Recommendation, and Integration

by

Xinlong Bao

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 24, 2009
Commencement June 2010

ACKNOWLEDGEMENTS

I owe a profound debt of thanks and appreciation to Tom Dietterich for persevering with me as my advisor throughout the time it took me to complete my PhD program. Dr. Dietterich is incredibly knowledgeable, energetic, inspirational and supportive. His commitment to the pursuit of excellence in research work guided me through the past five years, and I am sure it will benefit me a lot in my future career.

I would like to thank my committee members, Larry Bergman, Alan Fern, Jon Herlocker and Prasad Tadepalli. They generously contributed their time and expertise to better shape my research work. I also thank Anne Trehu for taking her time off to serve as Graduate Council Representative in my committee. I am immensely grateful for having been given the opportunity of spending these years with my fellow graduate students, faculty and staff here at Oregon State University.

I have had great teammates working together on the TaskTracer project, including but not limited to Jed Irvine, Simone Stumpf, Michael Slater, Kevin Johnsrude, Jianqiang Shen, Lida Li, Twinkle Lettkeman, Erin Fitzhenry, Wei Wu, Jill Cao, Ben Porter and Phuoc Do. It has been a great pleasure working with them, and I really enjoyed it. I also wish to thank my wonderful research collaborators at SRI International: Chris Brigham, Bruno Dutertre, Sam Owre, Natarajan Shankar and Tomas Uribe. Without their help, I wouldn't be able to build my own research on top of the giant CALO system.

I want to especially thank many people from IBM T. J. Watson Research Center, where I spent the summer of 2008 as an intern. My mentors Larry Bergman and Rich Thompson, my manager Ravi Konuru, and colleagues Vittorio Castelli, Yan Liu and Claudia Perlich all helped me shape the recommender systems research reported in this dissertation. I had a lot of fun working with them.

Last in deed but first in thought, I want to express my deepest gratitude to my family. Without the tremendous support and encouragement from them, I couldn't have made this far. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

DEDICATION

To my family.

Chapter 1 – Introduction

The explosion of the electronic information has introduced a big challenge for computer scientists to invent smart and efficient ways of finding information and improving the user experience. Machine learning, on the other hand, has been an active research area for decades. Recently, there is a growing interest in applying machine learning technologies to help computer users find information and better organize electronic resources.

In this chapter, we briefly introduce the three projects that are discussed in the following chapters, and give the outline of this dissertation.

1.1 FolderPredictor

The first part of the dissertation discusses the **FolderPredictor**, an intelligent desktop software tool that helps the user quickly locate files stored on the computer.

FolderPredictor [Bao *et al.*, 2006] is implemented as a part of the **TaskTracer** project at Oregon State University. TaskTracer [Dragunov *et al.*, 2005; Stumpf *et al.*, 2005] is an intelligent desktop software system designed to help multi-tasking knowledge workers better organize the computer resources around their tasks (or projects). FolderPredictor makes use of the user's tasks defined in TaskTracer, collects the user's desktop activities, and applies machine learning algorithms to predict the next file folder the user wants to access. The predicted folders are incorporated into the existing Microsoft Windows user interface (File Open/Save Dialog Boxes and Windows Explorer Toolbar) so that they can be easily accessed to reduce the cost of reaching the target folder.

Two user studies have been conducted to evaluate the precision and usability of FolderPredictor. A variety of prediction algorithms are introduced and evaluated on the real users' data collected in the user studies. The experimental results show that, on average, FolderPredictor reduces the cost of locating a file by more than 50%, and the best folder predition algorithm so far is a mixture of the most recently used (MRU) folder and the cost-sensitive predictions.

1.2 STREAM

The second part of the dissertation discusses the **STREAM** (**ST**acking **R**ecommendation **E**ngines with **A**dditional **M**eta-features) project.

Recommendation engines, including Collaborative Filtering, Content-based approaches, etc., have been invented and widely used in the past decade as important tools for finding online information and improving the user experience. Each of these algorithms has pros and cons; as a result, real-world recommendation systems are often hybrids [Burke, 2002] that combine multiple recommendation engines to generate better predictions.

Previous research on hybridization has focused on building a static hybridization scheme (for example, a linear combination of engine predictions) that does not change at runtime for different input users/items. However, this approach can not adjust the ways that the component engines are combined for different types of input users/items. For example, the collaborative filtering engine should be trusted more in the hybrid when the input user has rated a lot of items before, because it's well-known that the collaborative filtering engine works well for users with high number of ratings. This motivates us to propose a dynamic hybridization scheme that can adjust the ways that the component engines are combined depending on the inputs with different properties.

In the dissertation, we take a novel approach to the problem of hybridizing recommendation engines, by applying the idea of Stacking from ensemble learning. Properties of the input users/items, called runtime metrics, are employed as additional meta features to improve performance. To demonstrate this idea, we build a STREAM system for a movie recommendation application, which shows significant improvements over each single component engine and a static hybrid system.

1.3 Integrating Learning and Reasoning

The third part of the dissertation discusses the **Integrating Learning and Reasoning** project, which is a part of the CALO project [CALO, 2009] that helps computer users better organize their electronic resources.

This work addresses the question of how statistical learning algorithms can be integrated into a larger AI system both from a practical engineering perspective and from the perspective of correct representation, learning, and reasoning. The goal is to create an integrated intelligent system that can combine observed facts, hand-written rules, learned rules, and learned classifiers to perform joint learning and reasoning. To achieve this, we employ a Markov Logic [Domingos and Richardson, 2006] inference engine, named the Probabilistic Consistent Engine (PCE), that can integrate multiple learning components so that the components can benefit from each other's predictions.

In this dissertation, we investigate two designs of the learning and reasoning layer, covering the architectures, interfaces, and algorithms employed, followed by experimental evaluations of the performance of the two designs on a synthetic data set. We also conduct experiments on the real user's activity data collected by the TaskTracer system. The results show that by integrating multiple learning components through Markov Logic, the performance of the system can be improved.

1.4 Outline

This dissertation is structured as follows. Chapter 2 discusses the FolderPredictor. Chapter 3 discusses the STREAM project. Chapter 4 discusses the Integrating Learning and Reasoning project. In each chapter, the research background, system design and experimental evaluation are discussed. We then conclude the dissertation in Chapter 5 with proposal for future research directions.

Chapter 2 – FolderPredictor: Reducing the Cost of Reaching the Right Folder

Helping computer users rapidly locate files in their folder hierarchies has become an important research topic for intelligent user interface design. This chapter reports on FolderPredictor, a software system that can reduce the cost of locating files in hierarchical folders. FolderPredictor applies a cost-sensitive prediction algorithm to the user's previous file access information to predict the next folder that will be accessed. Experimental results show that, on average, FolderPredictor reduces the cost of locating a file by 50%. Several variations of the cost-sensitive prediction algorithm are discussed. An experimental study shows that the best algorithm among them is a mixture of the most recently used (MRU) folder and the cost-sensitive predictions. FolderPredictor does not require users to adapt to a new interface, but rather meshes with the existing interface for opening files on the Windows platform.

2.1 Research Background

Computer users organize their electronic files into folder hierarchies in their file systems. But unfortunately, with the ever-increasing numbers of files, folder hierarchies on today's computers have become large and complex [Boardman and Sasse, 2004]. With large numbers of files and potentially complex folder hierarchies, locating the desired file is becoming an increasingly time-consuming operation. Some previous investigations have shown that computer users spend substantial time and effort in just finding files [Barreau and Nardi, 1995; Jul and Furnas, 1995; Ko *et al.*, 2005]. Thus, designing intelligent user interfaces that can help users quickly locate desired files has emerged as an important research topic.

Previous research on helping users find files has focused on building Personal Information Management (PIM) systems in which documents are organized by their properties [Dourish *et al.*, 2000; Dumais *et al.*, 2003]. These properties include both system properties, such as name, path and content of the document, and user-defined properties that reflect the user's view of the document. In these systems, users can search for files by their properties using search engines. Although these search engines can be effective in helping the user locate files, previous user studies have indicated that instead of using keyword search, most users still like to navigate in the folder hierarchy with small, local steps using their contextual knowledge as a guide, even when they know exactly what they were looking for in advance [Barreau and Nardi, 1995; Jones *et al.*, 2005; Jul and Furnas, 1995; Teevan *et al.*, 2004].

In this chapter, we try to address the file-locating problem from another perspective, using a system that we call the *FolderPredictor*. The main idea of FolderPredictor is that if we have observations of a user's previous file access behavior, we can recommend one or more folders directly to the user at the moment he/she needs to locate a file. These recommended folders are predictions — the result of running simple machine learning algorithms on the user's previously observed interactions with files.

Ideally we want to identify when the user has just started to look for a file and provide a shortcut to likely choices for the folder containing that file. In today's graphical user interfaces, there are several user actions that strongly indicate the user is or will be initiating a search for a file. These include the display of a file open/save dialog box or the opening of a file/folder browsing application such as Windows Explorer. Our approach intervenes both cases.

First, FolderPredictor presents predicted folders by changing the default folder of the open/save file dialog that is displayed to computer users from within an application. It also provides shortcuts to secondary recommendations, in case the top recommendation is not correct. Second, FolderPredictor presents predicted folders as buttons inside a Windows Explorer toolbar. Users can easily jump to the predicted folders by clicking these buttons. An important advantage of FolderPredictor is that it reduces user cost without requiring users to adapt to a new interface. Users have nothing new to learn.

This chapter is organized as follows. In the next section, we introduce the TaskTracer system that FolderPredictor is built upon. Section 2.3 presents the user interfaces of FolderPredictor and the instrumentation for presenting predictions in the open/save file dialog box and the Windows Explorer. Section 2.4 describes how folder predictions are made, together with the cost-sensitive prediction algorithm and several variations of it. Section 2.5 reports the experimental results from two user studies: the first one establishes that FolderPredictor reduces the user's cost for locating files; the second one compares the performance of different variations of the basic prediction algorithm. Section 2.6 discusses some related work in email classification. At the end of the chapter, we have some additional discussions.

2.2 The TaskTracer System

FolderPredictor is built upon our TaskTracer system [Dragunov *et al.*, 2005; Stumpf *et al.*, 2005] — a software system designed to help multi-tasking computer users with interruption recovery and knowledge reuse. In this section, we briefly introduce the multi-tasking hypothesis and the data collection architecture of the TaskTracer system.

2.2.1 Multi-tasking Hypothesis

FolderPredictor monitors user activity to make predictions to optimize the user experience. Previous research in intelligent "agents" has explored this kind of approach [Horvitz *et al.*, 1998; Maes, 1994; Rhodes, 2003]. However, one of the challenges faced by these intelligent agents is that users are highly multi-tasking, and they are likely to need access to different files depending on the exact task they are performing. For example, a professor may have a meeting with a student for one hour and then switch to writing a grant proposal. The notes file for the student meeting and the files for the grant proposal are normally stored in different folders. After the professor switches to working on the grant proposal, a naive agent might assume that the student notes are still the most likely choice for prediction, because they were the most recently accessed. This prediction would fail, because the agent is not taking into consideration the context of the user's current task.

Previous work on intelligent agents or assistants has attempted to produce more context-aware predictions by analyzing the content of the documents that the user is currently accessing to generate an estimated keyword profile of the user's "task" [Budzik and Hammond, 2000]. This profile is then employed as a query to locate web pages with similar keyword profiles. This approach is limited because a) it cannot recommend resources that do not have text associated with them, b) there are substantial ambiguities in text, resulting in significant uncertainty in mapping from text profiles to user tasks, and c) the active window may not contain sufficient text. In the last case, the agent would be forced to scan recently-accessed documents to generate text profiles — documents that may have been accessed as part of a previous task.

In our TaskTracer system, three core hypotheses are made in order to characterize and utilize the user's multi-tasking behavior:

1. All information workers break their work into discrete units to which they can give names — we call these *tasks*.

2. At any moment in time, the user is working on only one task.

3. Knowledge of the user's current task will substantially reduce the uncertainty in predicting what a user is trying to do.

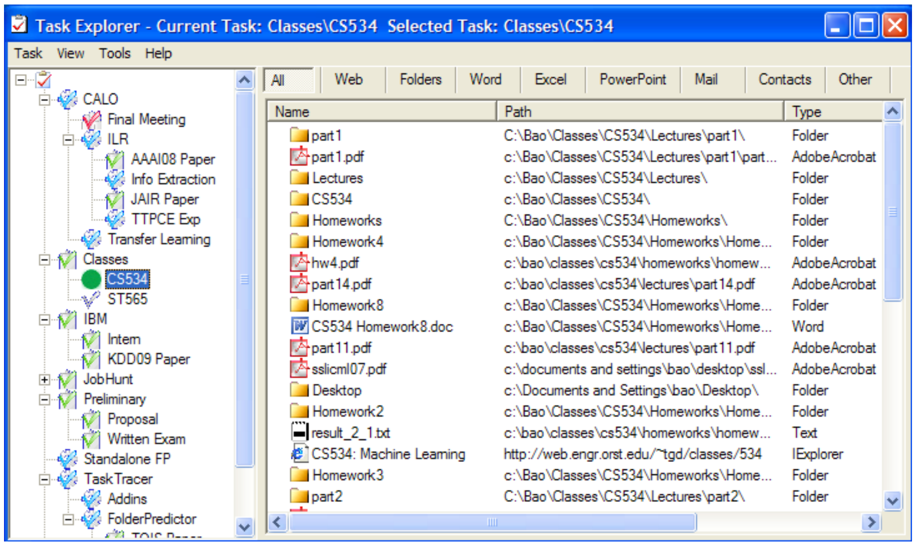
Users define new tasks in the TaskTracer system via an application called the

TaskExplorer. Users can then indicate their current task through three mechanisms, which are shown in Figure. 2.1. Users can switch to the TaskExplorer (Figure. 2.1(a)) and select the task from their defined hierarchy of tasks, or they can set the task by clicking on a taskbar widget that appears as a drop-down box at the bottom-right corner of the screen. This taskbar widget is called the TaskSelector (Figure. 2.1(b)), where the user can switch tasks by manually selecting another task from the drop-down menu or by typing in the textbox (with auto-completion). The third mechanism is through a tool called HotTask (Figure. 2.1(c)). It shows a pop-up menu of tasks that are most-recently worked on, to provide quick access to them. The user can switch tasks by pressing Ctrl + ∼ and stop at the desired task, just like switching windows by pressing Alt + Tab in Windows OS.

We also have a system called TaskPredictor, which can automatically detect task switches by observing the activity of the user [Shen *et al.*, 2009b]. If a probable task switch is detected, the user can be notified, actively or peripherally. Alternatively the task can be automatically changed if task prediction confidence is high enough.

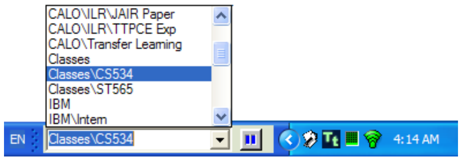
2.2.2 Data Collection Architecture

The TaskTracer system employs an extensive data-collection framework to obtain detailed observations of user activities. It instruments a wide range of applications under the Windows XP operating system, including Microsoft Office (Word, Excel, PowerPoint and Outlook), Internet Explorer, Adobe Acrobat, GSView and

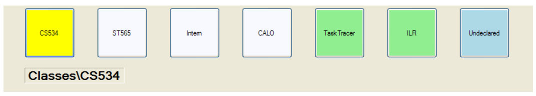


Name	Path	Type
part1	C:\Bao\Classes\CS534\Lectures\part1\	Folder
part1.pdf	c:\Bao\Classes\CS534\Lectures\part1\part...	AdobeAcrobat
Lectures	c:\Bao\Classes\CS534\Lectures\	Folder
CS534	c:\Bao\Classes\CS534\	Folder
Homeworks	C:\Bao\Classes\CS534\Homeworks\	Folder
Homework4	c:\Bao\Classes\CS534\Homeworks\Home...	Folder
hw4.pdf	c:\bao\classes\cs534\homeworks\homew...	AdobeAcrobat
part14.pdf	c:\bao\classes\cs534\lectures\part14.pdf	AdobeAcrobat
Homework8	c:\Bao\Classes\CS534\Homeworks\Home...	Folder
CS534 Homework8.doc	c:\Bao\Classes\CS534\Homeworks\Home...	Word
part11.pdf	c:\bao\classes\cs534\lectures\part11.pdf	AdobeAcrobat
sslicml07.pdf	c:\documents and settings\bao\desktop\ssl...	AdobeAcrobat
Desktop	c:\Documents and Settings\bao\Desktop\	Folder
Homework2	c:\Bao\Classes\CS534\Homeworks\Home...	Folder
result_2_1.txt	c:\bao\classes\cs534\homeworks\homew...	Text
CS534: Machine Learning	http://web.engr.orst.edu/~tgd/classes/534	IExplorer
Homework3	c:\Bao\Classes\CS534\Homeworks\Home...	Folder
part2	C:\Bao\Classes\CS534\Lectures\part2\	Folder

(a) TaskExplorer



(b) TaskSelector



(c) HotTask

Figure 2.1: User interfaces for defining tasks and declaring current task in the TaskTracer system.

Notepad. It also instruments some operating system components including the system clipboard and window focus switches.

Listener components are plug-ins into applications. They capture user interaction events and send them to the *Publisher* as *Events*, which are XML strings with pre-defined syntax and semantics. One event of interest is the TaskBegin event, which is sent to the Publisher when the user switches tasks. The Publisher stores these events in its database and also distributes them to *Subscriber* applications that need to react to events online. FolderPredictor is one of the subscriber

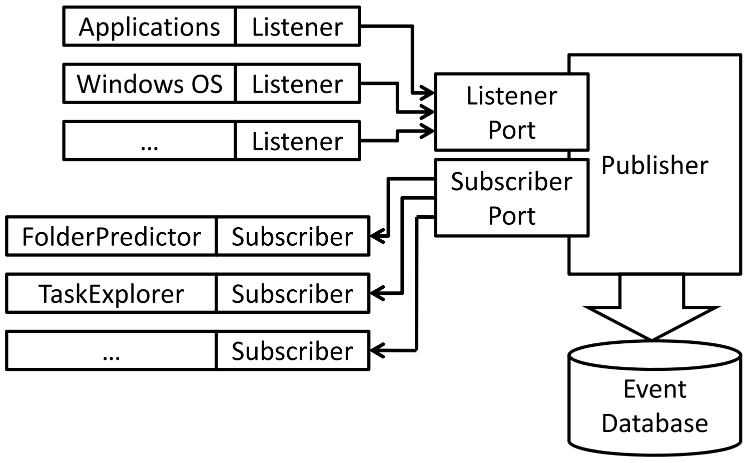


Figure 2.2: Publisher-Subscriber architecture in the TaskTracer system.

applications of TaskTracer. This *Publisher-Subscriber Architecture* is shown in Figure 2.2.

2.3 FolderPredictor User Interfaces

A basic design principle of FolderPredictor is simplicity. This basic principle greatly influences the UI design — FolderPredictor actually has *no independent UI*.

2.3.1 Display Predicted Folders in Existing User Interfaces

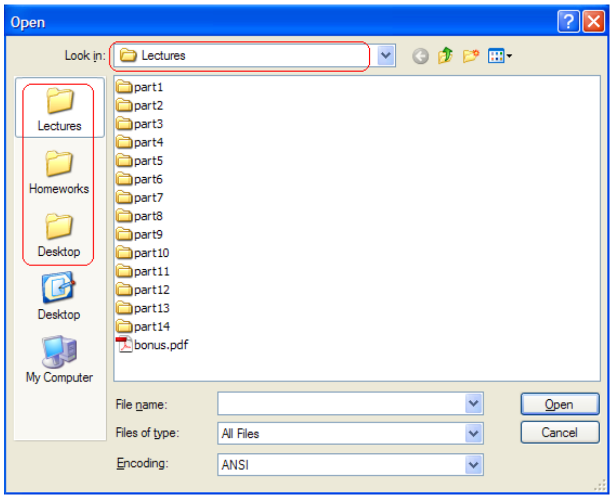
FolderPredictor provides three shortcuts to the likely choices for folders when the user has just started to look for a file (in particular, the display of a file open/save dialog box or the opening of a Windows Explorer window). FolderPredictor achieves this by modifying the existing user interfaces.

First, FolderPredictor enhances the well-known Windows Open/Save File Dialog boxes by adding its predicted folders. Figure 2.3(a) shows an example Open File Dialog box enhanced by FolderPredictor. Three predicted folders are shown as the top three icons in the "places bar" on the left. The user can jump to any of them by clicking on the corresponding icon. The top predicted folder is also shown as the default folder of the dialog box so that the display will show this folder initially. Mousing over a folder icon will display the full path of that folder.

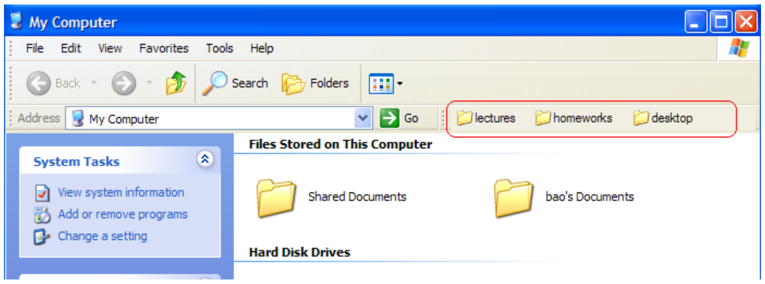
There are five slots in the places bar. By default, Microsoft Windows places five system folders (including "My Computer" and "Desktop") there. Informal questioning of Windows users revealed that several of these shortcuts were not commonly used. Thus, we felt it was safe to replace some of them by predicted folders. By default, FolderPredictor uses three slots for predicted folders and leaves two system folders. This behavior can be configured by the users if they want to see more system folders in the places bar.

Second, FolderPredictor adds an explorer toolbar to the Window Explorer window as shown in Figure 2.3(b). Three predicted folders are accessible to the user while browsing inside the Windows Explorer. Clicking on a folder icon will navigate the current Windows Explorer window directly to that folder. Mousing over a folder icon will display the full path of that folder.

These two interfaces hook into the native Windows environment and carry no overhead for the user to learn additional interactions. An important advantage of FolderPredictor is that it reduces user cost while introducing only minor changes



(a) Folder predictions in the Open/Save File Dialog box.



(b) Folder predictions in the Windows Explorer window.

Figure 2.3: FolderPredictor user interfaces.

to the existing user interfaces.

2.3.2 Implementing the Instrumentation

Microsoft Office applications employ a file dialog from a custom library, while most other Windows applications use the file dialog from the common Win32 dialog library provided by the operating system. Thus, FolderPredictor needs three separate sets of instrumentation: the common Win32 file dialog, the Microsoft Office file dialog and the Windows Explorer toolbar.

2.3.2.1 Instrumentation for the common Win32 file dialog

In order to modify the places bar in the common Win32 dialog, FolderPredictor creates five entries named “Place0” to “Place4” under the registry key “HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\ComDlg32\Placesbar\”. These five entries correspond to the five slots, and their values can be set to the paths of the predicted folders or to the system-independent numeric IDs (CSIDL) of the system folders.

Modifying the default folder is much more difficult. There is no documented support for this feature from Microsoft. FolderPredictor modifies the default folder by injecting an add-in (a .DLL file written in C with embedded assembly language) to all applications when they are started. This add-in intercepts Win32 API calls invoked by the injected application. Common Win32 applications show the Open/Save file dialog by invoking API calls named “GetOpenFileName” or “GetSaveFileName” with the default folder passed as a parameter. Thus, the add-in can modify the default folder by intercepting the API call, changing the parameter, and then passing it on. A detailed introduction of API interception technology can be found in [Pietrek, 1995].

By intercepting the above two API calls, we can also get the original default folder of the file dialog and the folder returned by the file dialog. This information was used in the evaluation of FolderPredictor, which is presented in Chapter 2.5.

2.3.2.2 Instrumentation for the Microsoft Office file dialog

As with the common Win32 file dialog, the places bar in the Microsoft Office file dialog can be modified by manipulating registry keys. The pertinent registry key is "HKEY_CURRENT_USER\Software\Microsoft\Office\ *VersionNumber*\Commo n\Open Find\Places\", inside which *VersionNumber* should be replaced by the version number of Microsoft Office software installed on the computer — for example, "11.0" for Office 2003 and "12.0" for Office 2007.

Microsoft Office uses a file dialog from a custom library, and the API calls in this library are not exposed. Therefore, API interception technology can not be used to modify the default folder of Microsoft Office file dialog. FolderPredictor hooks into the Microsoft Office file dialog by loading add-ins created by Visual Studio Tools for Office (VSTO) into Microsoft Office applications. Code in these add-ins, written in C#.NET, is invoked when the file dialog is called. When invoked, the code changes the default folder of the file dialog to the predicted folder and then shows the dialog box.

2.3.2.3 Instrumentation for the Windows Explorer toolbar

FolderPredictor's toolbar in the Windows Explorer is a customized COM add-in registered as an explorer toolbar by adding its GUID under the registry key

“HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Toolbar\”. This add-in implements the IObjectWithSite interface to initialize and dispose itself. It subscribes to the TaskTracer events that contain FolderPredictor's predictions and updates the three folder buttons accordingly.

2.4 Folder Prediction Algorithms

The main goal of FolderPredictor is to reduce the cost of locating files. In this chapter, we assume that the user navigates in the folder hierarchy using a mouse, and the number of “clicks” necessary to reach the destination folder is an appropriate measure of the cost to the user. One “click” can lead the user from the currently selected folder to its parent folder or to one of its sub-folders. Our folder prediction algorithms seek to reduce the number of clicks needed to reach the user's destination folder from the predicted folders.

In this section, we first introduce our approach for collecting possible target folders and assigning weights to them. Then we discuss the idea of including ancestor folders as candidate folders for predictions. After that, we introduce our cost-senstive prediction algorithm, followed by several variations of it.

2.4.1 Collecting Possible Target Folders with Weights

Our approach assumes that computer users, for the most part, separate files for different tasks into different folders. We further assume that, for the most part, the working set of folders for a task is relatively small. Given such assumptions, the paths of the files that the user has accessed when working on a task provide useful information for making folder predictions for future accesses for that task. For example, imagine that, during one day, a student opens and saves files under the folders "C:\Classes\CS534\Homeworks\" and "C:\Classes\CS534\Presentations\" when he is working on a task named *CS534*. The next day, when he returns to working on the *CS534* task, it should be useful to recommend these two folders or even the parent folder "C:\Classes\CS534\".

FolderPredictor generates its predictions by applying a simple machine learning method to a stream of observed file open/save events. Each of these events includes the path of the folder containing the file that was opened or saved. For each task, FolderPredictor maintains statistics for each folder — how many times the user opened files from it or saved files to it.

A statistical approach to making folder predictions is important for two reasons: 1) more-frequently-accessed folders should be more probable predictions and 2) users sometimes access the wrong files, or forget to specify that they have changed task. In the second case, the misleading events will add to the statistics. If observed accesses to a particular folder are really just noise, then we are unlikely to observe repeated future accesses to that folder over time. Thus these folders should have relatively low access frequencies. We can use this information to filter out these folders from recommendations.

Another factor that should be considered is recency. Our hypothesis is that a user is more likely to need to access recently-accessed folders. For example, a programmer working on a big project may need to access many different folders of source code, working on them one by one, but accessing each folder multiple times before moving on to the next folder. Thus, the folders with older access times should be quickly excluded from the predictions when the programmer begins to work on source code under new folders. To achieve this, we have incorporated a recency weighting mechanism into FolderPredictor. Instead of keeping a simple count of how many times a folder is accessed, we assign a recency weight w_i to each folder f_i. All weights are initially zero. When a file in folder f_i is opened or saved while working on a task, the weights of all the folders that have been accessed on that task are multiplied by a real number α between 0 and 1, and w_i is then increased by 1. α is called the discount factor. Multiplying by the discount factor exponentially decays the weights of folders that have not been accessed recently. When $\alpha = 0$, only the most recently-accessed folder will have a nonzero weight, and it will always be predicted. When $\alpha = 1$, no recency information is considered, and weights are not decayed. Experiments show that a discount factor in the range [0.7, 0.9] performs the best. Another benefit of recency weighting is that folders erroneously identified as relevant due to noisy data are excluded from consideration quickly, because their weights decrease exponentially.

In the implementation of FolderPredictor, we apply an incremental update method to maintain the folder weights. The first time FolderPredictor is run on a computer, historical TaskTracer data, if available, are used to build the initial list of folders and associated weights for each task. This information is stored in FolderPredictor's private database. Then FolderPredictor incrementally updates this database as new open or save events arrive, until the FolderPredictor is shut down. The next time FolderPredictor is started, it updates its database using only the events that have been added since the last time FolderPredictor was shut down. This incremental update method helps FolderPredictor keep its data up-to-date without any perceivable delay in prediction time or startup.

2.4.2 Predicting Ancestor Folders Is Better Sometimes

After we have collected a list of folders with weights assigned to them, the folder prediction problem seems trivial: we could just recommend the folder having the highest weight. However, while that might maximize the chance that we pick the best possible folder, it may not minimize the average cost in clicks. Recall that our goal here is to minimize the number of clicks to reach the target folder. To do this, we may want to sometimes recommend an ancestor folder. We will motivate the need for this decision by an example. Suppose a student has a folder hierarchy as shown in the tree structure in Figure 2.4. His files are stored in the bottom level folders, such as “Part3” and “Hw2”. When he worked on the *CS534* task, he accessed almost all of the bottom level folders with approximately similar counts. In this circumstance, predicting a bottom level folder will have a high probability of being wrong, because all the bottom level folders are equally probable. This will cost the student several more “clicks” to reach his destination folder — first to go up to an ancestor folder and then go back down to the correct child. On the other hand, predicting a higher level folder, such as “Presentations” or “Homeworks” or

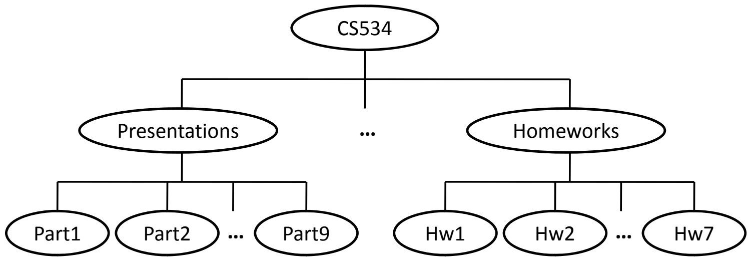


Figure 2.4: An example folder hierarchy. Each node of this tree denotes a folder. Child nodes are sub-folders of their parent node.

even "CS534", may not be a perfect hit, but it may reduce the cost in half — the student only needs to go downward to reach his destination folder.

Figure 2.5 shows the folder tree of a real user who has been using FolderPredictor for approximately a month. The folders shown here are only the ones that were accessed by this user during this period and their ancestors. The darkness of a node in this tree is proportional to how frequently the corresponding folder was accessed. There are quite a few cases where predicting the parent folder is better than predicting a child folder. One example is folder #5. It has 13 child folders, most of which were accessed. Another example is folder #40. It wasn't directly accessed by the user, but it has 6 child folders that were accessed by the user with approximately equal frequency.

Furthermore, we believe that incorrectly predicting a leaf node will be on average more frustrating for users than picking an ancestor node that is an incomplete path to the desired folder. For example, in Figure 2.4, if the user is going to access the leaf folder "Part1" and FolderPredictor predicts the parent folder "Presentations" and initializes the file open dialog box with it, the user will be able to see

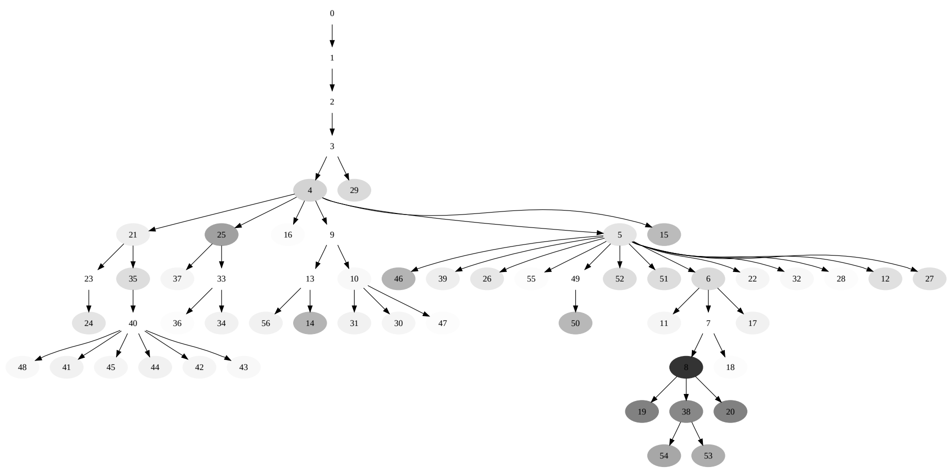


Figure 2.5: Folder tree from a real user. Each node of this tree denotes a folder. Darker nodes denote the folders that have been accessed more often. The number on each node is the identification number of the folder. The root node marked with “0” denotes “My Computer”.

the folder “Part1” when the dialog box is displayed and thus easily jump to it. On the other hand, if the dialog box is initialized with a leaf node such as “Part2”, it will be cognitively harder for the user to recognize that “Part2” is a sibling of the target folder “Part1”, because there is no visual cue of this.

2.4.3 Cost-Sensitive Prediction Algorithm

Based on the ideas presented above, we developed the following Cost-Sensitive Prediction (CSP) algorithm shown in Figure 2.6.

The goal of the CSP algorithm is to find three folders from the candidate set that jointly minimize the expected number of clicks required to reach the user’s target folder. The candidate set `H` here consists of the folders that the user has

```
Input: A finite set $\mathsf{F} = \{f_1, f_2, \ldots, f_m\}$, where $f_i$ is a folder with a
       positive weight $w_i$, $1 \leq i \leq m$.
Output: Three recommended folders $a_1$, $a_2$ and $a_3$(descending in
        predicted preference).
1  for $i \leftarrow 1$ to $m$ do        /* normalize weights to get probability
   distribution */
2      $p_i = w_i / \sum_{j=1}^{m} w_j$;
3  end
4  $\mathsf{H} = \emptyset$;
5  for $i \leftarrow 1$ to $m$ do       /* construct the candidate set by adding
   ancestor folders */
6      $H_i \Leftarrow$ all ancestors of $f_i$, including $f_i$ itself;
7      $\mathsf{H}$ = Union ($\mathsf{H}$, $H_i$);
8  end
9  forall folder $h$ in $\mathsf{H}$ do
10     forall folder $f$ in $\mathsf{F}$ do
11         $dis[h, f]$ = TreeDistance ($h$, $f$);
                       /* tree distance from h to f, in clicks */
12     end
13 end
14 $\{a_1, a_2, a_3\} =$
   $\arg\min_{\{a_1,a_2,a_3\}} \sum_{i=1}^{m} p_i \times \min\{dis[a_1, f_i], dis[a_2, f_i] + 1.0, dis[a_3, f_i] + 1.0\}$;
```

Figure 2.6: Cost-Sensitive Prediction (CSP) Algorithm.

accessed before (set `F`) and their ancestor folders. The probability distribution of the user's target folder is estimated by normalizing the weights of the folders in `F`, which are collected as described in Chapter 2.4.1.

In line #14 of the CSP algorithm, the cost of getting to a folder f_i from a prediction $\{a_1, a_2, a_3\}$ is computed as $\min\{dis[a_1, f_i], dis[a_2, f_i] + 1.0, dis[a_3, f_i] + 1.0\}$, because of the following facts:

1. a_1 will be set as the default folder of the open/save file dialog. This means

that the user will be in folder a_1 with no extra "click" required. Therefore, $dis[a_1, f_i]$ is the cost if the user navigates to f_i from a_1.

2. a_2 and a_3 will be shown as shortcuts to the corresponding folders in the "places bar" on the left side of the open/save file dialog box (see Figure 2.3). The user must execute one "click" on the places bar if he wants to navigate to f_i from a_2 or a_3. Therefore, an extra cost 1.0 is added. (In fact, this doesn't affect which three folders are chosen, but it will order them to make sure a_1 is the best folder to predict.)

3. We assume the user knows which folder to go to and how to get there by the smallest number of clicks. Therefore, the cost of a prediction $\{a_1, a_2, a_3\}$ is the minimum of $dis[a_1, f_i]$, $dis[a_2, f_i] + 1.0$, and $dis[a_3, f_i] + 1.0$.

FolderPredictor runs this algorithm whenever the user switches task and whenever the folder weights are updated. The three predicted folders are then displayed in the file dialog boxes and the Windows Explorer.

2.4.4 Variations of the CSP Algorithm

During our initial deployment of FolderPredictor, we observed that, although FolderPredictor largely reduces the cost of reaching the right folder, the approach of predicting ancestor folders that lead to multiple possible folders tends to make the prediction less "perfect": the top predicted folder sometimes is close to but not exactly the same as the user's target folder. This observation inspired us to consider several variations of the CSP algorithm that may increase the chance of making a perfect prediction.

The first variation is to make FolderPredictor more "aggressive": choosing the most probable folder (the folder with the highest weight among the folders that the user has accessed before) as the top prediction (a_1). The other two predicted folders are choosen by running the CSP algorithm with the top prediction fixed. That is, line #14 of the CSP algorithm in Figure 2.6 is changed to be $\{a_2, a_3\} = \arg\min_{\{a_2,a_3\}} \sum_{i=1}^{m} p_i \times \min\{dis[a_1, f_i], dis[a_2, f_i] + 1.0, dis[a_3, f_i] + 1.0\}$. By doing this, the top predicted folder will tend to be a "leaf" folder that contains the files that the user is most-likely to access. Therefore, it may increase the chance that FolderPredictor makes a perfect prediction, but once the top predicted folder is wrong, it may cost more clicks for the user to reach the target folder.

The second variation is to make use of the most recently-used (MRU) folder. Microsoft Windows usually initialize the file open/save dialog boxes with the MRU folder. This approach works fairly well and in our experiments presented in the next section, it perfectly hits the target folder about 53% of the time. Therefore, it's worth combining it into our predictions — let the MRU folder be the top prediction (a_1) and choose the other two predicted folders by running the CSP algorithm with the top prediction fixed.

There are two different types of MRU folder: the system-wide MRU folder and the application-specific MRU folder. The application-specific MRU folder is the last folder accessed by the user using the same application. For example, the application-specific MRU folder for Microsoft Word is the folder containing the file that was most recently opened or saved by Microsoft Word. We store one application-specific MRU folder for each application. On the other hand, the system-wide MRU folder is the last folder accessed by the user, no matter which application was used. It is hard to say which type of MRU folder is better, so we implemented both and compared them in our experiments.

The third variation is slightly different from the second one. A potential problem with always predicting the MRU folder as the top prediction is that when there is a task switch, the MRU folder may be a bad choice. Therefore, we want to apply the original CSP algorithm to predict for the first file access within a task episode (period between two task switches), and apply the second variation described above to predict for the remainder of the task episode. A "task episode" for task T is the segment of time in which the user is working on task T. It starts with a "TaskBegin" message, when the user indicates he/she is starting to work on task T, and it ends with another "TaskBegin" message when the user switches to a different task.

Table 2.1 summarizes the folder prediction algorithms we have described above.

2.5 Experimental Evaluations

To measure the effectiveness of FolderPredictor and compare the performance of different folder prediction algorithms, we have conducted two user studies. The goal of the first user study was to measure how many clicks FolderPredictor with the CSP algorithm saves, by comparing FolderPredictor's predictions with *Win-*

Table 2.1: Summary of the folder prediction algorithms.

Name	**Description**
CSP	Cost-Sensitive Prediction algorithm described in Chapter 2.4.3.
ACSP	Use the most-probable folder as top prediction and run CSP to get the other two folders.
SMCSP	Use the system-wide MRU folder as top prediction and run CSP to get the other two folders.
SMCSP2	Use CSP for the first file access within a task episode, and use SMCSP for the remainder of the task episode.
AMCSP	Use the application-specific MRU folder as top prediction and run CSP to get the other two folders.
AMCSP2	Use CSP for the first file access of each application within a task episode, and use AMCSP for the remainder of the task episode.

dows Default (without folder predictions). The goal of the second user study was to compare the performance of the various folder prediction algorithms listed in Table 2.1.

2.5.1 Comparing the CSP Algorithm with Windows Default

The first user study was conducted at Oregon State University. The participants were two professors and two graduate students. The TaskTracer system (with FolderPredictor) was deployed on the computers that the participants performed their daily tasks on. Every participant ran the TaskTracer system for a fairly long time (from 4 months to 1 year). The algorithm employed in this version of FolderPredictor was the CSP algorithm. The discount factor α was set to 0.85.

At the end of this user study, we collected four data sets as shown in Table 2.2. Each data set is a list of predictions that FolderPredictor made for a user, ordered by time. Each prediction is marked with the name of the task that was active at the moment this prediction was made. The size of a data set is the number of predictions it contains.

#	**User Type**	**Data Collection Time**	**Set Size**
1	Professor	12 months	1748
2	Professor	4 months	506
3	Graduate Student	7 months	577
4	Graduate Student	6 months	397

Table 2.2: Information about the data sets collected in the first user study.

2.5.1.1 Average Cost

Figure 2.7 compares the average cost (in "clicks") of the FolderPredictor and the Windows Default on all four data sets. The cost of the Windows Default is the distance between the original default folder of the file dialog and the destination folder. In other words, the cost of Windows Default is the user cost when FolderPredictor is not running. The figure also shows 95% confidence intervals for the costs.

Statistical significance testing shows that FolderPredictor surely reduces the user cost of locating files (P-value $= 1.51 \times 10^{-29}$ using an ANOVA F-test). On average, the cost is reduced by 49.9% when using FolderPredictor with the CSP algorithm.

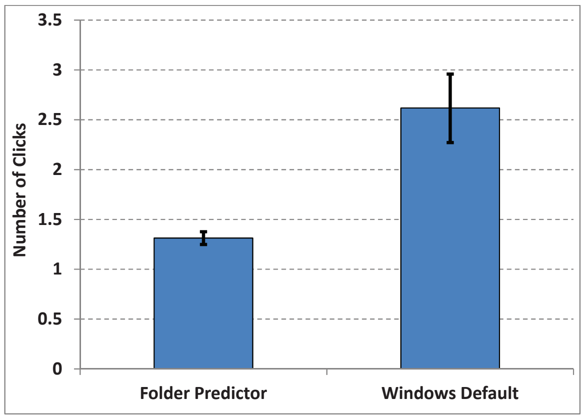


Figure 2.7: Average cost of FolderPredictor and Windows Default.

2.5.1.2 Distribution of Costs

Figure 2.8 shows a histogram of the number of clicks required to reach the target folder under the Windows Default and the FolderPredictor. We can see from the figure that:

1. About 90% of the FolderPredictor's costs are less than or equal to three clicks. Only a small fraction of the FolderPredictor's costs are very high.

2. Although about half of the Windows Default's costs are zero, about 40% of the Windows Default's costs are above three.

This means that FolderPredictor not only reduces the overall average cost of locating files, but also decreases the probability of encountering very high costs in locating files.

It is interesting to see that Windows Default actually gets the default folder perfectly correct more than FolderPredictor. This most likely happens because

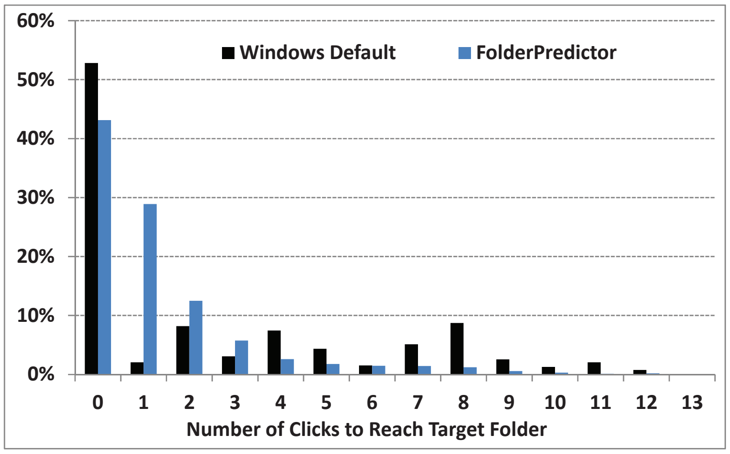


Figure 2.8: Histogram of the number of clicks required to reach the target folder.

Windows typically picks a leaf folder (i.e., the most recently-used folder) as the default folder. FolderPredictor sometimes plays it safe and picks an ancestor folder that is more likely to be close to multiple possible folders and less likely to be perfect. Thus we see a large number of cases relative to the Windows Default where FolderPredictor is one, two, or three clicks away. In fact, this discovery inspired us to come up with algorithm variations described in Chapter 2.4.4.

2.5.1.3 Learning Curve

Machine learning systems usually perform better as more training data is acquired. In FolderPredictor's case, the training data are the user's opens and saves per task. For each open/save, FolderPredictor makes a prediction and uses the user's actual destination folder to update itself. Therefore, the cost of the folders recommended by FolderPredictor should decrease as more predictions are made for a task. To

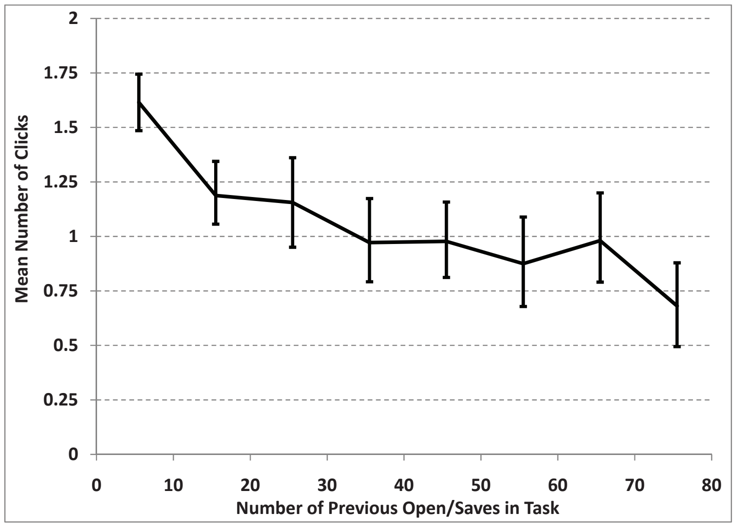


Figure 2.9: Learning curve of FolderPredictor.

evaluate this, we computed the learning curve for FolderPredictor shown in Figure 2.9.

In the figure, the X-axis is the number of predictions within one task aggregated into ranges of width 10, and the Y-axis is the average cost of one prediction within this range. For example, the first point of the curve shows the average cost of all predictions between (and including) the 1st and 10th predictions of all tasks. The figure also shows 95% confidence intervals for the average costs.

The curve shows that the cost decreases as more predictions made. The average cost decreases from 1.6 (first 10 predictions) to 0.7 (71st to 80th predictions).

2.5.2 Comparing Different Folder Prediction Algorithms

The second user study was conducted at Intel Corporation. The participants were knowledge workers who perform their daily work on the computers. During a 4-week period, a software system called Smart Desktop (Beta 3), which is a commercialized version of TaskTracer, was running on their computers. Smart Desktop incorporates a version of the FolderPredictor but, more importantly, it collects all of the information needed to evaluate alternative folder prediction algorithms. At the end of the study, we ran an analysis program on the data to evaluate the cost of each folder prediction algorithm by simulating the algorithm at each File Open/SaveAs event in time order.

Seven participants completed the study. One of them is excluded from the analysis because the number of file opens/saves is too low (< 20) in his data. Table 2.3 shows the information about the data sets collected from the remaining 6 participants. The set size is the number of File Open/SaveAs events (i.e., the number of evaluation points) in the data set.

#	**Data Collection Time**	**Set Size**
s1	4 weeks	138
s2	4 weeks	99
s3	4 weeks	395
s4	4 weeks	146
s5	4 weeks	102
s6	4 weeks	352

Table 2.3: Information about the data sets collected in the second user study.

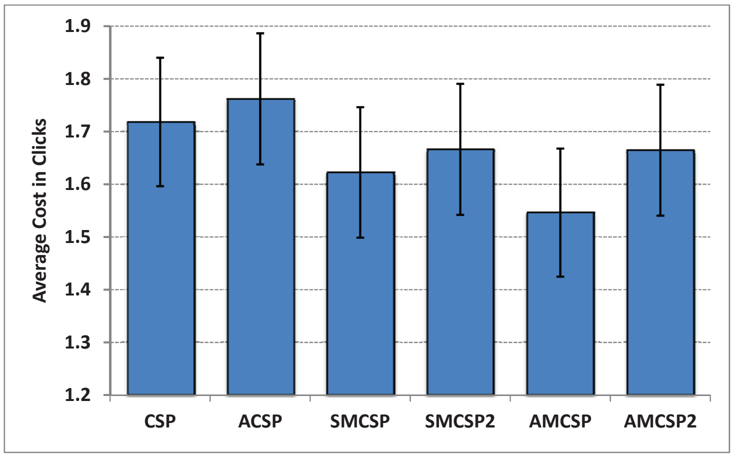


Figure 2.10: Average cost of different folder prediction algorithms over all data sets.

2.5.2.1 Average Cost of Different Algorithms

Figure 2.10 compares the average cost (in "clicks") of different folder prediction algorithms over all data sets. The figure also shows 95% confidence intervals for the costs.

There are several interesting observations from this figure:

1. The ACSP algorithm performs worse than the original CSP algorithm. This means predicting the most probable folder as top prediction actually increases the overall cost of the prediction.

2. Algorithms using an MRU folder as the top prediction (SMCSP, SMCSP2, AMCSP, AMCSP2) have better performance than the original CSP algorithm.

3. The SMCSP2/AMCSP2 algorithms are worse than the corresponding SM-

CSP/AMCSP algorithms. This is a little surprising, because we thought that the MRU folder would not be good for the first file access after a task switch. One possible explanation is that different tasks may share the same folders, especially for two tasks that are worked on sequentially in time. Another factor is that the users may not declare task switches accurately due to human errors.

4. The AMCSP algorithm has the best performance among all the algorithms.

The differences among the average costs of these algorithms are generally not statistically significant. However, the difference between the average cost of the AMCSP algorithm (the best one) and that of the original CSP algorithm is statistically significant with $p\text{-}value = 0.05$ assessed with a paired t-test. On average, the AMCSP algorithm further reduces the cost of FolderPredictor by 10%, compared with the original CSP algorithm.

2.5.2.2 Comparison of Algorithms across Different Users

Different users behave differently in folder organization and file access. We wanted to investigate how the folder prediction algorithms compare with each other for different participants. Figure 2.11 illustrates the average cost of the folder prediction algorithms across different participants. In this figure, the cost of the CSP algorithm is used as baseline, and the values shown are the change in the number of clicks compared to the CSP algorithm. The horizontal axis shows the ID's of our study participants.

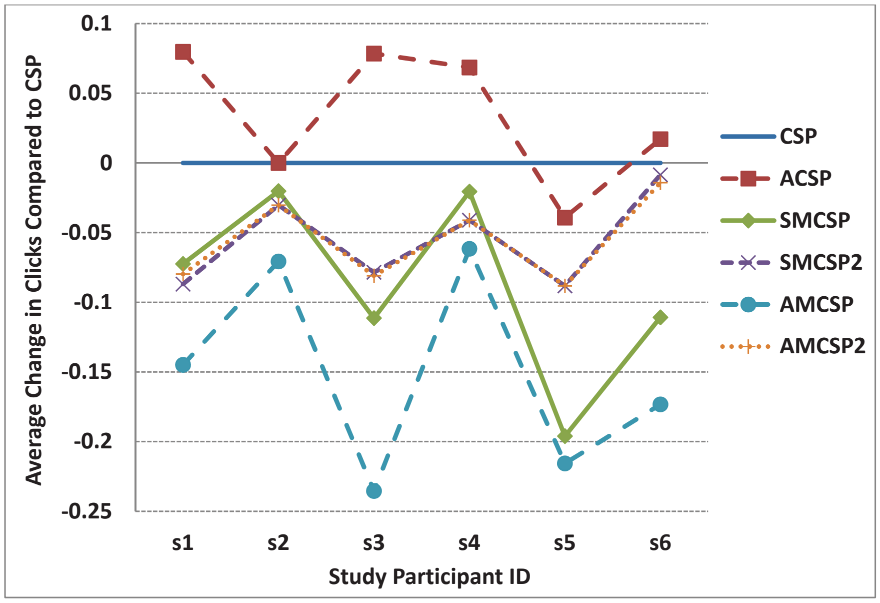


Figure 2.11: Average cost of algorithms across different participants.

The figure clearly shows that the AMCSP algorithm is consistently the best algorithm across all participants. And other observations we discover from Figure 2.10 are also valid for each study participant.

2.5.2.3 Distribution of Costs

To investigate why the AMCSP algorithm outperforms the CSP algorithm, we compared the distribution of costs of these two algorithms, as shown in Figure 2.12.

As illustrated in the figure, both algorithms show the desired property of reducing the probability of encountering very high costs in locating files. The main advantage of the AMCSP algorithm is that it makes more perfect predictions (zero clicks) than the CSP algorithm, which validates our hypothesis that using the MRU

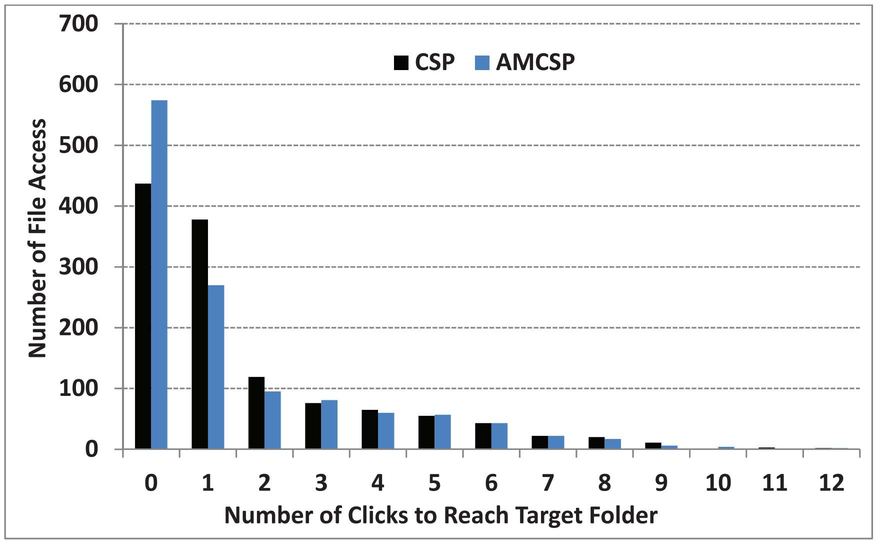


Figure 2.12: Histogram of the number of clicks required to reach the target folder.

folder as the top prediction can increase the chance of making perfect predictions. If we sum up the number of file accesses with cost of 0, 1 and 2 clicks, this value is approximately equal for the two algorithms. Therefore, the reason why the AMCSP algorithm outperforms the CSP algorithm is that the AMCSP algorithm replaces some of the CSP's 1- and 2-click predictions into perfect predictions.

2.6 Related Work

On the surface, the FolderPredictor approach is similar to some email classification systems that predict possible email folders based on the text of incoming email messages [Rennie, 2000; Segal and Kephart, 1999; 2000]. In particular, both MailCat and FolderPredictor present their top three predicted folders to the user [Segal and Kephart, 1999]. However our approach is different in the following aspects:

1. Our predictions are made for file folders, not email folders. Predicting file folders is probably much more difficult than predicting email folders. One reason is that in most situations, there are many more file folders than email folders. Furthermore, there are many different types of files under file folders, not only email messages.

2. Our predictions are based on user activities, not text in files. Text-based approaches may be applicable for email foldering, but they are more challenging to apply to folder prediction. Challenges include

    (a) tasks with similar keyword profiles but different folders (e.g., the class I taught last year versus the class I am teaching this year);

    (b) files from which it is hard to extract text;

    (c) ambiguity in language;

    (d) computational time to extract and analyze text, which is essential for a real time application like FolderPredictor.

There are also some software tools that help users to quickly locate their files in the open/save file dialog boxes, e.g. Default Folder X [Default Folder X, 2009] for Mac and Direct Folders [Direct Folders, 2009] for Windows. These tools make the open/save file dialog boxes more configurable and comprehensive to the user. The user can put more shortcuts in the dialog, as well as define a default folder for each application. However, these tools cannot adjust the shortcuts and default folders automatically based on the context of the user's activities. On the other hand, our approach makes intelligent folder predictions based on the user activities within each task. Therefore, FolderPredictor can be a good complement to these tools.

2.7 Discussion

The results reported in this chapter are likely a substantial underestimate of the value of the FolderPredictor. One of the key assumptions of this chapter is that the user always knows which folder they want to get to and where it is located — in such cases, we have demonstrated that FolderPredictor will get them there faster. The reality is that people have limited memory, and highly multitasking users often cannot maintain in their memory the locations of files for all of their tasks, particularly tasks that they have not worked on recently. Thus users may need many more clicks to "search" for the right folder. By default, Windows only remembers what was worked on most recently, regardless of task. FolderPredictor on the other hand remembers multiple folders used on each task, regardless of how long ago the task was last active. Thus FolderPredictor's recommendations can help remind the user where files related to a task have been stored.

There are other psychological benefits of FolderPredictor that are harder to evaluate. For example, being placed consistently in the wrong folder can generate frustration, even if the click distance is not far. At this stage, all that we have is qualitative evidence of this. During the deployments of FolderPredictor, multiple participants reported becoming "addicted" to FolderPredictor — they were distressed when they had to use computers that did not have TaskTracer installed. They also reported that they did a better job of notifying TaskTracer of task switches in order to ensure that the FolderPredictor recommendations were appropriate for the current task.

Chapter 3 – Stacking Recommendation Engines with Additional Meta-features

In this chapter, we apply stacking, an ensemble learning method, to the problem of building hybrid recommendation systems. We also introduce the novel idea of using runtime metrics which represent properties of the input users/items as additional meta-features, allowing us to combine component recommendation engines at runtime based on user/item characteristics. In our system, component engines are level-1 predictors, and a level-2 predictor is learned to generate the final prediction of the hybrid system. The input features of the level-2 predictor are predictions from component engines and the runtime metrics. Experimental results show that our system outperforms each single component engine as well as a static hybrid system. Our method has the additional advantage of removing restrictions on component engines that can be employed; any engine applicable to the target recommendation task can be easily plugged into the system.

3.1 Research Background

The growth of online information has stimulated the use of recommendation engines as an important way of finding information and improving user experience. For example, Netflix.com recommends movies based on the user's scoring of previously watched movies (both the user and others); Amazon.com recommends goods to the online shopper based on the previously viewed/purchased items and the purchases of other customers.

In the past two decades, a number of recommendation engines have been developed for a wide range of applications. Content-based recommendation engines are typically used when a user's interests can be correlated with the description (content) of items that the user has rated. An example is the newsgroup filtering system NewsWeeder [Lang, 1995]. Collaborative filtering engines are another popular type which utilize users' preferences on items to define similarity among users and/or items. An example is the GroupLens system [Konstan *et al.*, 1997]. Other recommendation technologies include knowledge-based approaches, utility-based filtering, etc [Burke, 2002].

Previous research has shown that each of these engines has pros and cons [Adomavicius and Tuzhilin, 2005; Burke, 2002; Ramezani *et al.*, 2008]. For example, collaborative filtering engines depend on overlap in ratings (whether implicit or explicit) across users, and perform poorly when the ratings matrix is sparse. This causes difficulty in applications such as news filtering, where new items are entering the system frequently. Content-based engines are less affected by the sparsity problem, because a user's interests can be based on very few ratings, and new items can be recommended based on content similarity with existing items. However, content-based engines require additional descriptive item data, for example, descriptions for home-made video clips, which may be hard to obtain. And experiments have shown that, in general, collaborative filtering engines are more accurate than content-based engines [Alspector *et al.*, 1997].

Real-world recommendation systems are typically hybrid systems that combine multiple recommendation engines to improve predictions (see Burke [Burke, 2002] for a summary of different ways that recommendation engines can be combined). Previous research on hybridization has mostly focused on static hybridization schemes which do not change at runtime for different input users/items. For example, one widely used hybridization scheme is a weighted linear combination of the predictions from component engines [Bell *et al.*, 2007a; Claypool *et al.*, 1999], where the weights can be uniform or non-uniform. Pazzani [Pazzani, 1999] also proposed a voting schema to combine recommendations.

However, this approach can not adjust the ways that the component engines are combined for different types of input users/items. For example, the collaborative filtering engine should be trusted more in the hybrid when the input user has rated a lot of items before, but it should not be trusted at all when the input item is a new item that no one has rated before. Therefore, in this chapter, we focus on building hybrid recommendation systems that exhibit the following two properties:

1. The system should adjust how component engines are combined depending on properties of the inputs. For example, collaborative filtering engines are less accurate when the input user has few ratings on record; the system should reduce the weights of these engines for this type of user.

2. The system should allow not only linear combinations but also non-linear combinations of predictions from component engines. For example, a piece-

wise linear function may be preferable to a simple linear function if a component engine is known to be accurate when its predictions are within a certain range but inaccurate outside this range.

To achieve these two goals, we apply stacking, an ensemble learning method, to solve the problem of building hybrid recommendation systems. The main idea is to treat component engines as level-1 predictors, and to learn a level-2 predictor for generating the final prediction of the hybrid system. We also introduce the novel idea of using runtime metrics as additional meta-features, allowing us to use characteristics of the input user/item when determining how to combine the component recommendation engines at runtime. These runtime metrics are properties of the input user/item that are related to the precisions of the component engines. For example, the number of items that the input user has previously rated may indicate how well a collaborative filtering engine will perform. By employing different learning algorithms for learning the level-2 predictor, we can build systems with either linear or non-linear combinations of predictions from component engines. We name our method and the resulting system **STREAM** (**ST**acking **R**ecommendation **E**ngines with **A**dditional **M**eta-features).

The chapter is organized as follows. In the next section, we discuss related work. Section 3.3 describes our STREAM approach in detail. Section 3.4 demonstrates how to build a STREAM system for a movie recommendation application and discusses how to apply the concepts to other domains. Section 3.5 presents experimental results. Section 3.6 concludes the chapter with discussion.

3.2 Related Work

The BellKor system that won the first annual progress prize of the Netflix competition [Netflix, 2009; Bell *et al.*, 2007b] is a statically weighted linear combination of 107 collaborative filtering engines. The weights are learned by a linear regression on the 107 engine outputs [Bell *et al.*, 2007a]. This method is actually a special case of STREAM wherein no runtime metrics are employed and the level-2 predictor is learned by linear regression.

Some hybrid recommendation systems choose the "best" component engine for a particular input user/item. For example, the Daily Learner system [Billsus and Pazzani, 2000] selects the recommender engine with the highest confidence level. However, this method is not generally applicable for two reasons. First, not all engines generate output confidence scores for their predictions. Second, confidence scores from different engines are not comparable. Scores from different recommendation engines typically have different meanings and may be difficult to normalize.

There are also hybrid recommendation systems that use a linear combination of component engines with non-static weights. For example, the P-Tango system [Claypool *et al.*, 1999] combines a content-based engine and a collaborative filtering engine using a non-static user-specific weighting scheme: it initially assigns equal weight to each engine, and gradually adjusts the weights to minimize prior error as users make ratings. This scheme combines engines in different ways for different input users. However, the prior error of an engine may not be a sufficient indicator of the quality of its current prediction. For instance, the prior error of a collaborative filtering engine is probably lower than that of a content-based engine for a user who has rated 100 items. But if the two engines are asked to predict this user's rating on a new item, the content-based engine will probably make a better prediction because the collaborative filtering engine is unable to predict ratings for new items. Another disadvantage of this method is the rise in computational cost of minimizing the prior error as ratings accumulate.

There have been several research efforts to apply machine learning / artificial intelligence methods to the problem of combining different recommendation technologies (mostly content-based and collaborative filtering). These typically focus on building unified models that combine features designed for different recommendation technologies. For example, Basu, Hirsh and Cohen applied the inductive rule learner Ripper to the task of recommending movies using both user ratings and content features [Basu *et al.*, 1998]. Basilico and Hofmann designed an SVM-like model with a kernel function that is based on joint features of user ratings as well as attributes of items or users [Basilico and Hofmann, 2004]. Our goal in this chapter, however, is to build hybrid recommendation systems that combine the outputs of individual recommendation engines into one final recommendation. We treat the component engines as black boxes, making no assumption on what underlying algorithms they implement. In the latter sections of this chapter, we will show that any engine applicable to the target recommendation task can be easily plugged into our STREAM system. Anytime a new engine is added or an old engine is removed, all we need to do is re-learn the level-2 predictor. This allows system designers to flexibly customize the hybrid recommendation system with their choice of component engines.

3.3 Our Approach

In this section, we first introduce the stacking method in ensemble learning. We then describe how we apply it to solve the engine hybridization problem with runtime metrics as additional meta-features. Finally we demonstrate our STREAM framework.

3.3.1 Stacking: An Ensemble Learning Method

Stacking (also called *Stacked Generalization*) is a state-of-the-art ensemble learning method that has been widely employed in the machine learning community. The main question it addresses is: given an ensemble of classifiers learned on the same set of data, can we map the outputs of these classifiers to their true classes?

The stacking method was first introduced by Wolpert in [Wolpert, 1992]. The main idea is to first learn multiple level-1 (base) classifiers from the set of original training examples using different learning algorithms, then learn a level-2 (meta) classifier using the predictions of the level-1 classifiers as input features. The final prediction of the ensemble is the prediction of the level-2 classifier. Training examples for the level-2 classifier are generated by performing cross-validation [Hastie *et al.*, 2001] on the set of original training examples. The idea of stacking classifiers was extended to stacking regressors by Breiman [Breiman, 1996], where both level-1 predictors and the level-2 predictor are regression models that predict continuous values instead of discrete class labels.

The level-2 predictor can be learned using a variety of learning algorithms. We call these learning algorithms meta-learning algorithms in order to distinguish them from the learning algorithms used to learn the level-1 predictors. Dzeroski and Zenko [Dzeroski and Zenko, 2004] empirically compared stacking with several meta-learning algorithms, reaching the conclusion that the model tree learning algorithm outperforms others. They also reported that stacking with model trees outperforms a simple voting scheme as well as a "select best" scheme that selects the best of the level-1 classifiers by cross-validation.

3.3.2 Stacking Recommendation Engines

We are addressing the problem of combining predictions from multiple recommendation engines to generate a single prediction. To apply the stacking method to the engine hybridization problem, we first define each component recommendation engine as a level-1 predictor. We treat each engine as a black box that returns a prediction given the input. Then we learn a level-2 predictor, using a meta-learning algorithm, with predictions of the component engines as meta-features. The level-2 predictor can be either a linear function or a non-linear function based on the meta-learning algorithm employed. This satisfies one of our two goals: support for non-linear combinations of predictions from component engines, as well as linear combinations.

However, this method fails to achieve the other goal: we want a system that can adjust how the component engines are combined depending on the input values. For example, suppose there are two users A and B, with user A rating only 5 items, while user B rates 100 items. It is likely that the collaborative filtering engine works better for user B than for user A, while the content-based engine may work equally well for both of them. Thus, the weight on the collaborative filtering engine should be higher when the system is predicting for user B than for user A.

To achieve our goal of a system that adapts to the input, we define new meta-features that indicate the expected quality of the predictions from the component engines. These new meta-features are properties of the input users/items that can be computed at runtime, in parallel with the predictions from the component engines. We call these new meta-features runtime metrics. For example, the runtime metric, "the number of items the input user has previously rated", might be applicable to the problem in the previous paragraph. In general, the runtime metrics are both application domain specific and component engine specific. Therefore, we cannot define a set of runtime metrics that work for all applications. Instead, in the next section we will describe a set of runtime metrics defined for a movie recommendation application and discuss general characteristics of these metrics for other applications.

3.3.3 STREAM Framework

Figure 3.1 illustrates our STREAM framework. To be concrete, we assume that the recommendation task is to predict $R(u, i)$, the rating of the input user u on the input item i. We call the input (u, i) a user-item pair. The system's background data consists of historical ratings known to the system and possibly additional information such as item content. The framework does not place restrictions on the algorithms used inside the component engines. The only requirement for an engine is that given an input user-item pair and a set of background data, it must return a predicted rating. *MetricEvaluator* is a component for computing the runtime metrics. The engines' predictions $\langle P_1, P_2, ..., P_n\rangle$ and the values of the runtime metrics $\langle M_1, M_2, ..., M_m\rangle$ are passed to the level-2 predictor $f(\cdot)$, which is a function of the engines' predictions and the runtime metrics, to generate a final prediction $R(u, i)$.

Figure 3.2 shows the underlying meta-learning problem in STREAM. The input vector to the level-2 predictor is in the top dotted ellipse and the output value of the level-2 predictor is in the bottom dotted ellipse. This gives us a standard machine learning problem. To learn the model, we first generate a set of training examples in the format $(\langle M_1, M_2, ..., M_m, P_1, P_2, ..., P_n\rangle, P_T)$ where M_i is the value of the i-th runtime metric evaluated for a user-item pair, P_j is the prediction of the j-th component engine for this user-item pair, and P_T is the user's true rating for this item. We then apply an appropriate meta-learning algorithm to learn a model from these training examples. If the ratings are ordered numbers, this meta-

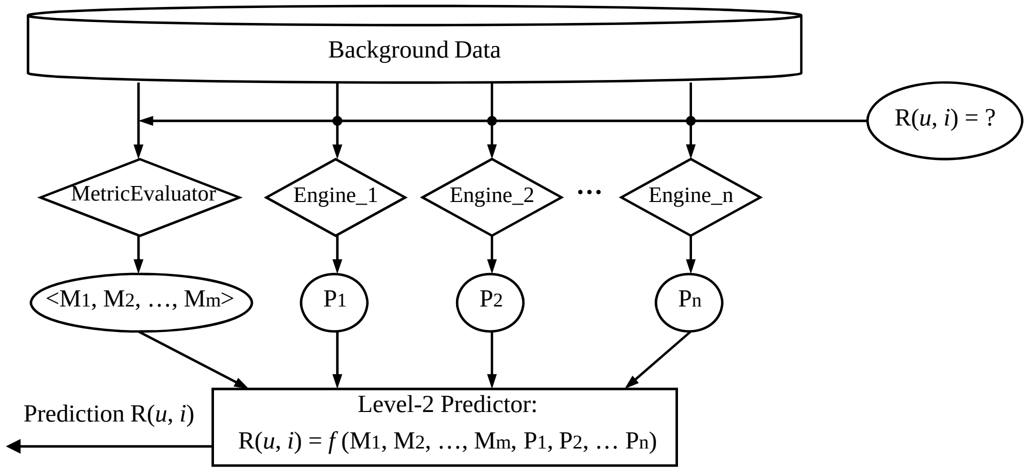


Figure 3.1: STREAM Framework.

learning problem is a regression problem. If the ratings are unordered categories (e.g., "Buy" / "No Buy"), this meta-learning problem is a classification problem.

To generate the training examples, we perform a cross-validation on the background data. The general idea is to simulate real testing by splitting the original background data into two parts: *cv_background* data which is used as background data for the component engines and the MetricEvaluator, and *cv_testing* data on which the learned model is tested. For each user-item pair in the testing data, an input vector $\langle M_1, M_2, ..., M_m, P_1, P_2, ..., P_n \rangle$ can be generated by running the MetricEvaluator on the input user-item pair and by requesting predictions of the component engines. The true rating for this user-item pair (P_T) is known, giving us a complete training example.

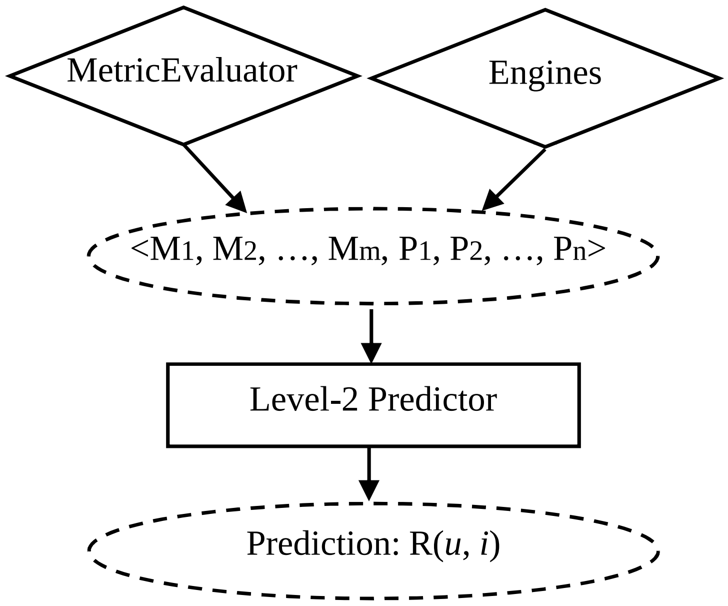


Figure 3.2: The meta-learning problem in STREAM.

3.4 Predicting Movie Ratings: An Application

Predicting users' movie ratings is one of the most popular benchmark tasks for recommender systems. Ratings are typically represented by numbers between 1 and 5, where 5 means "absolutely love it" and 1 means "certainly not the movie for me". There are several publicly available data sets for this problem. To demonstrate our STREAM method, we built a movie recommendation system and evaluated it on the widely used MovieLens data set [MovieLens, 1997]. This data set consists of 100,000 ratings from 943 users on 1682 movies — each user rates 4.3% of the movies on average. Each record in this data set is a triplet $\langle user, item, rating \rangle$.

The MovieLens data set contains only the title, year, and genre for each movie.

This is insufficient for useful recommendations from a content-based engine. Therefore, we augmented this data set with movie information extracted from the IMDb movie content collection [IMDb, 2009]. After augmentation, the movie contents included the title, year, genre, keywords, plot, actor, actress, director, and country. Note that not all movies contain complete information; some fields are missing for some movies.

3.4.1 Recommendation Engines

Three widely-used but significantly different recommendation engines were chosen for the system: a user-based collaborative filtering engine, an item-based collaborative filtering engine, and a content-based engine.

Our user-based collaborative filtering engine is built according to the basic algorithm described in [Herlocker *et al.*, 1999]. The similarity between two users is defined by the Pearson Correlation. To predict the rating of the user u on the item i, this engine selects the most similar 300 users as u's neighborhood, and outputs the average of the neighbors' ratings on the item i weighted by the corresponding similarities.

Our item-based collaborative filtering engine is built according to the basic algorithm described in [Sarwar *et al.*, 2001]. The similarity between any two items is defined as the Pearson Correlation between the rating vectors of these two items, after normalization to the interval between 0 and 1. To predict the rating of the user u on the item i, this engine computes the similarities between item i and all items u has rated, and outputs the average rating of all items u has rated weighted by the similarities between them and item i.

Our content-based engine is the same as the item-based collaborative filtering engine except that the item similarity is defined as the TF-IDF similarity [Salton and McGill, 1986] calculated from the movie contents. Apache Lucene [Lucene, 2009] is employed to compute the TF-IDF scores.

There are cases where one or more engines are unable to make predictions. For example, none of the three engines can predict for new users who do not have ratings recorded in the background data. Similarly, the two collaborative filtering engines cannot predict users' ratings on items that no one has yet rated. Our engines will predict the overall median rating in its background data if their underlying algorithms are unable to make predictions.

3.4.2 Runtime Metrics

The runtime metrics were designed based on characteristics of the component recommendation engines. We sought measures that we expected to correlate well with the performance of each engine, and that would distinguish between them. We considered the following general characteristics of the engines:

1. The user-based collaborative filtering engine works well for users who have rated many items before but not for users who have rated few items. It also works poorly for the users who tend to rate items that no one else rates.

2. The item-based collaborative filtering engine works well for items that have

been rated by many users but not for items that few users have rated.

3. The content-based engine performs consistently no matter how many items the input user has rated and how many users have rated the input item, but it works poorly when the content of the input item is incomplete or non-descriptive.

Based on these properties, we design the runtime metrics. Table 3.1 shows the runtime metrics we have defined for the movie recommendation application and the three engines described above. We assume $\langle u, i \rangle$ is the input user-item pair. All eight runtime metrics are normalized into the range between 0 and 1 by dividing by the corresponding maximum possible value (e.g., total number of items for RM_1).

Table 3.1: Runtime metrics defined for the movie recommendation application.

ID	**Runtime Metric Definition**
RM_1	Number of items that u has rated
RM_2	Number of users that have rated i
RM_3	Number of users that have rated the items u has rated
RM_4	Number of users that have rated more than 5 items u has rated
RM_5	Number of neighbors of u that have rated i
RM_6	Number of items that have rating similarity more than 0.2 with i
RM_7	Number of items that have content similarity more than 0.2 with i
RM_8	Size of the content of i

Note that these eight runtime metrics are ones that we consider related to the performance of the three component engines. It is by no means a complete set, and others might define different runtime metrics, even for the same engines. On the other hand, we will show in the next section that it is unnecessary to use all eight runtime metrics. Using just a subset of these metrics, we can achieve almost the same performance as using them all.

It is important to note that runtime metrics are specific to both an application domain, and to the specific engines to be hybridized. Application specificity in designing our metrics can be seen in the use of user ratings and item contents, integral to the movie recommendation application. For other applications, other runtime metrics would be defined. For example, in an online shopping application, one could define a binary runtime metric "whether the user inputs query words" because one might expect that the content-based engine will work better when query words are presented.

Engine specificity can also be seen in our runtime metrics. For example, we expect better performance of the user-based collaborative filtering engine as values of RM_5 rise, because this engine's predictions improve when more neighbors have rated the same item. Similarly, the content-based engine should perform better when RM_8 is higher, because the content similarity computed for this item has higher accuracy when the content is more descriptive. Some of the runtime metrics are predictive of the performance of multiple engines. For example, we expect all three engines to perform better when RM_1 is higher, but the two collaborative filtering engines to be affected by this runtime metric more than the content-based engine. It is important to select metrics that do a good job of differentiating the engines, i.e., that show a different response across the range of values for each engine.

3.4.3 Meta-Learning Algorithms

There are several properties of the target application to be considered when choosing meta-learning algorithms:

- Is the final prediction numerical or unordered categorical? If numerical, regression algorithms are required, e.g., linear regression, regression trees, SVM regression, etc. If the final prediction is unordered categorical, classification algorithms are required, e.g., naive Bayes, decision trees, SVM, etc.

- Are the input features (predictions from components engines) numerical or categorical or mixed? Some learning algorithms, such as nearest neighbors, are good at dealing with numerical features, while others, such as naive Bayes, are good at dealing with categorical features. Many algorithms, such as linear regression and SVM, cannot work with mixed data without additional data conversions.

In the movie rating application, both input features and final prediction are numerical (real numbers between 1 and 5). Therefore, we tested the following three learning algorithms:

1. **Linear regression**: learns a linear function of the input features. Note that there could be a non-zero intercept term in the learned function. In cases where all component engines tend to overrate (or underrate) in their predictions, the intercept term may help reduce the error of the final prediction.

2. **Model tree** [Wang and Witten, 1997]: learns a piece-wise linear function

of input features. As mentioned in Chapter 3.3.1, model tree algorithms have been shown to be good candidates for the meta-learning algorithm. We anticipate that this algorithm can capture some of the non-linearity of this meta-learning problem.

3. **Bagged model trees**: bagged version of model trees. Bagging is an ensemble learning method to improve the accuracy of machine learning models by reducing the variances [Hastie *et al.*, 2001]. Tree models generally have small biases but large variances. Therefore, bagging them is usually a good practice.

We use the implementations of these three algorithms in Weka, a widely-used open source machine learning library [Witten and Frank, 2005]. The size of the bagged ensemble is set to 10 for the bagged model trees algorithm. The default values in Weka are retained for other learning algorithm parameters.

3.5 Experimental Results

In this section, we evaluate the performance of our STREAM system on the MovieLens data set and compare with the performance of each component engine as well as a static equal-weight hybrid system. We also compare the effectiveness of the three learning algorithms in our STREAM system, and evaluate the utilities of different sets of runtime metrics.

3.5.1 Setup

We randomly split the entire MovieLens data set into a training set with X% ratings and a testing set with (100-X)% ratings. The split is performed by putting all ratings in one pool and randomly picking ratings regardless of the users. The training set serves as background data for all three component engines and the MetricEvaluator. In addition, the background data for the content-based engine includes content from all movies; this is reasonable, since movie content is available whether or not any given film has been rated. For each triplet $\langle user, item, rating\rangle$ in the test set, we compare the predicted rating with the true rating using mean absolute error (MAE), a widely-used metric for recommendation system evaluation. Smaller MAE means better performance.

We vary the value of X from 10 to 90 in order to evaluate the performance of the system under different sparsity conditions. The background data is sparser when the value of X is smaller. For each value of X, we repeat the random split 10 times and report the average performance of the system.

In each experiment, the level-2 predictor is learned by individually running the three meta-learning algorithms on the training examples generated by performing a 10-fold cross-validation on the training set (X% of total data). The cross-validation is performed as described in Chapter 3.3.3. The number of training examples generated is the same as the size of the background set. For the MovieLens data set, this number is 10,000 for X=10 and 90,000 for X=90. Since the model to be learned only has 11 input features (three engine predictions plus eight runtime metrics), it is not necessary to use all the training examples. Therefore we extract a random sample of 5,000 training examples for learning the level-2 predictor.

3.5.2 Comparison

Figure 3.3 compares the performance of the different systems. The three dotted curves correspond to the three single component engines: the user-based collaborative filtering engine, the item-based collaborative filtering engine, and the content-based engine. As anticipated, the two collaborative filtering engines perform badly when X is small due to the sparsity problem and their performance improves quickly as X increases, while the content-based engine's performance is less sensitive to the value of X, yielding a much flatter curve.

The "Equal Weight Linear Hybrid" curve in the figure corresponds to a static linear hybrid of the three engines with equal weights $\langle 1/3, 1/3, 1/3 \rangle$. Its overall performance is significantly better than the single engines. One possible explanation is that averaging the predictions from the three engines reduces the variance of the predictions.

The "STREAM - linear regression", "STREAM - model tree" and "STREAM - bagged model trees" curves show the performance of our STREAM system with three different meta-learning algorithms. All three systems are consistently better than the equal weight hybrid. The bagged model trees algorithm is slightly better than the model tree algorithm, and they are both better than the linear regression algorithm.

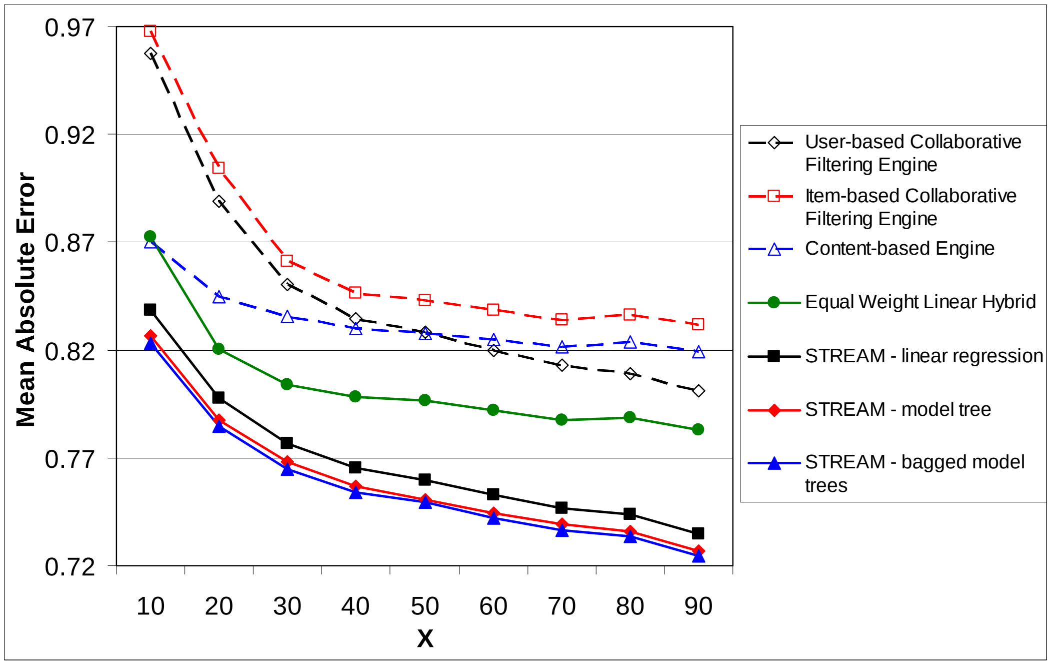


Figure 3.3: Comparison of performance on the MovieLens data set.

3.5.3 Different Sets of Runtime Metrics

Note that some of the runtime metrics are engine-specific and computationally expensive. For example, RM_5 involves a compute-intensive neighborhood search operation that is specific to the user-based collaborative filtering engine. We want to eliminate such expensive runtime metrics and find a small set that are easily computed but still provide good results. This corresponds to the feature selection problem in machine learning because the runtime metrics are employed as input features for the meta-learning problem. Therefore, we conduct experiments to compare the STREAM system with different sets of runtime metrics. We use the bagged model trees algorithm as the meta-learner, since it gave the best results in

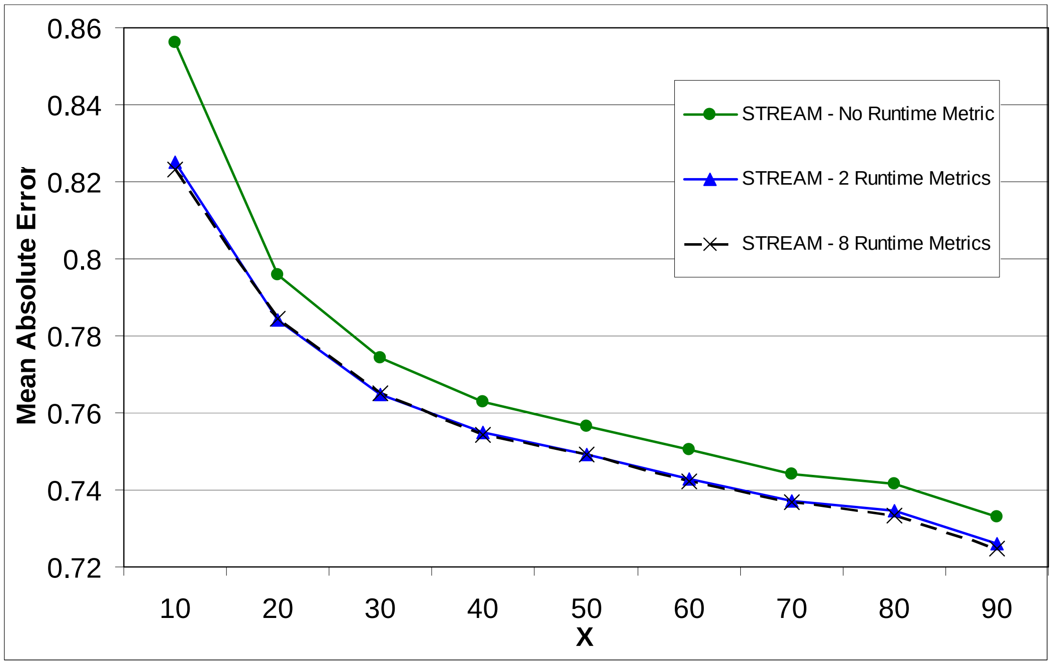


Figure 3.4: Performance of the STREAM system with different sets of runtime metrics.

the previous experiment.

Experimental results are shown in Figure 4. The "STREAM - No Runtime Metric" curve shows the performance of the STREAM system without any runtime metrics, using only the predictions of the three engines as input meta-features to the level-2 predictor. The curve shows consistently poorer performance than the systems with runtime metrics, especially when the value of X is small. We believe this results from having many users who rated few items when X is small; the runtime metrics let the system weight the content-based engine more heavily when predicting for these users.

The "STREAM - 8 Runtime Metrics" curve shows the performance of the

STREAM with all eight runtime metrics, while the "STREAM - 2 Runtime Metrics" curve shows the performance with only two runtime metrics: RM_1 and RM_2. We selected these two runtime metrics because they reflect local sparsity for the input user and item, and are easy to compute. The curve shows that using only these two metrics, the system can achieve approximately the same performance as the system with all eight runtime metrics; adding additional metrics does not necessarily improve the performance of the system.

3.6 Discussion

Since the STREAM framework is about hybridizing recommendation engines, we do not consider the computational cost of the component engines. The only concern is the additional computational cost of the STREAM system over the cost of the component engines. We identify two different costs: runtime cost and offline cost. At runtime, the STREAM system incurs additional computation for the runtime metrics and for evaluation of the prediction model on the current inputs. If chosen carefully, the runtime metrics can be computed quickly. For example, the runtime metrics RM_1 and RM_2 in Table 1 can be stored in a look-up table, with a table update whenever there is a new rating. The cost of evaluating the prediction model (a linear or non-linear function) depends on the learning algorithms used. Model-based algorithms, such as the three employed in our experiments, compute predictions very quickly. The offline cost of learning the prediction model is high, however. Most of the time is spent generating the training examples from the background data. In our experiments, the training example generation is performed by 10-fold cross-validation, which suggests we need to re-learn the prediction model only when 10 percent or more of the data has changed[1]. In summary, the STREAM system has a low runtime overhead, while offline model learning is costly, but can be performed infrequently.

Depending on the meta-learning algorithm employed, it's possible for the STREAM system to make predictions without running all component engines. For example, if the prediction model is a linear function, there is no need to run the engines whose coefficients are close to 0. For more complex models, we may create a decision process that decide at runtime for each input user/item which component engine(s) to run, taking into account both the prediction model and the values of the runtime metrics.

[1]Further experiments show that prediction models learned by leave-one-out and 10-fold cross-validation have approximately the same performance. Therefore, offline model learning when 10 percent or more of the data has changed is as good as online model learning for every single piece of new data.

Chapter 4 – Integrating Multiple Learning Components Through Markov Logic

This chapter addresses the question of how statistical learning algorithms can be integrated into a larger AI system both from a practical engineering perspective and from the perspective of correct representation, learning, and reasoning. Our goal is to create an integrated intelligent system that can combine observed facts, hand-written rules, learned rules, and learned classifiers to perform joint learning and reasoning. Our solution, which has been implemented in the CALO system [CALO, 2009], integrates multiple learning components with a Markov Logic inference engine, so that the components can benefit from each other's predictions. We introduce two designs of the learning and reasoning layer in CALO: the MPE Architecture and the Marginal Probability Architecture. The architectures, interfaces, and algorithms employed in our two designs are described, followed by experimental evaluations. We show that by integrating multiple learning components through Markov Logic, the performance of the system can be improved via a process called relational co-training.

4.1 Research Background

Statistical machine learning methods have been developed and applied primarily in stand-alone contexts. In most cases, statistical learning is employed at "system build time" to construct a function from a set of training examples. This function is then incorporated into the system as a static component. For example, in the NCR check reading system [LeCun *et al.*, 1998], the neural network system for reading check amounts is a sophisticated but static component of the system.

Our aims are more ambitious. We want to address the question of how statistical learning algorithms can be integrated into a larger AI system through a reasoning layer. We seek AI systems that exhibit end-to-end learning across all (or at least most) components of the system after deployment. We also anticipate that these systems will be built out of many separately-constructed components which may or may not have learning capabilities. Hence, we cannot insist on a uniform learning and reasoning algorithm for all of the components of the system. Instead, we seek to provide a learning and reasoning layer that can interconnect these components in such a way that if the components implement specified interfaces, the system will exhibit the desired end-to-end learning behavior.

In this chapter, we present and evaluate two designs for the learning and reasoning layer, both based on Markov Logic [Domingos and Richardson, 2006] as the reasoning engine. The rest of this chapter is organized as follows. First, we describe the CALO system, which provides the framework and constraints within which this work was carried out. Second, we describe the architecture, interfaces, and algorithms employed in our two designs. This is followed by two sets of experimental evaluations.

4.2 Learning in CALO

The work described here arose as part of the CALO project. CALO (**C**ognitive **A**gent that **L**earns and **O**rganizes) is an integrated AI system to support knowledge workers at the computer desktop. One of the primary functions of CALO is to help the user keep files, folders, email messages, email contacts, and appointments organized. CALO models the user's computer work life as consisting of a set of projects, where each project involves a set of people, meetings, files, email messages, and so on.

Figure 4.1 shows a portion of CALO's relational model of this information with objects as nodes and relations as arcs. All of the objects and many of the relations are observed with certainty (e.g., which files are stored in which folders), but the associations between projects and all of the other objects are inferred by applying a mix of hand-written rules and learned classifiers. CALO also provides interfaces for the user to associate objects with projects, so that when an incorrect association is made, the user can correct it and establish new associations.

Rules are employed to capture domain knowledge. Some of the rules used in CALO are shown in Table 4.1. For example, the Attachment Rule says that if a document is attached to an email message, both the document and the message tend to be associated with the same project. End-users can write their own rules

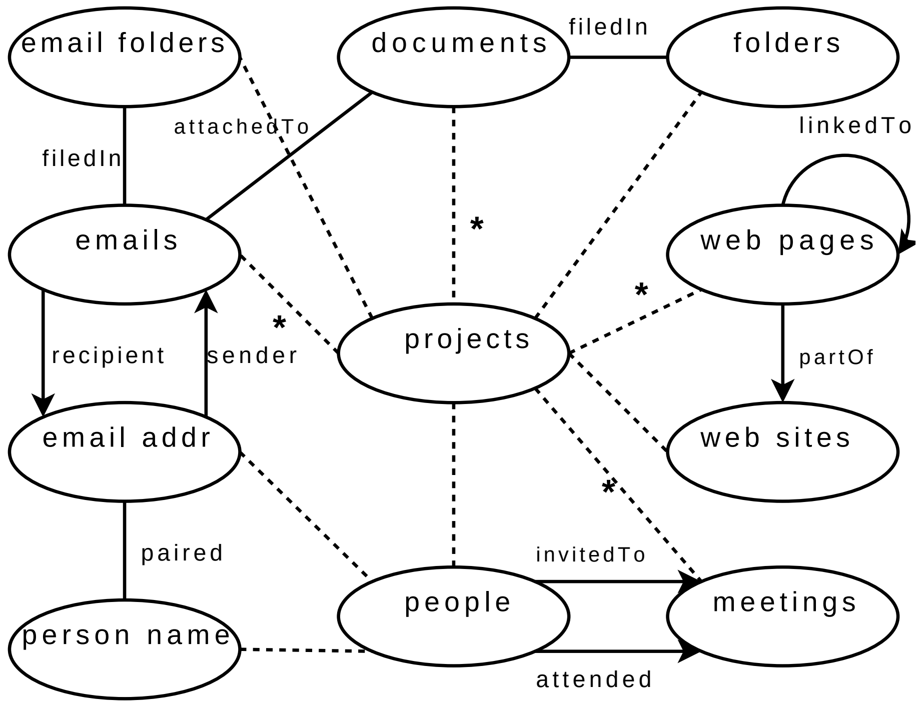


Figure 4.1: A portion of CALO's relational model. Dashed lines indicate uncertain relations; asterisks indicate relations learned by classifiers.

as well.

A key observation of our work is that when rules and classifiers are intermixed, the rules can establish connections between learning tasks. This creates the opportunity to perform *relational co-training.* Co-training is a method for semi-supervised learning in which there is a single, standard supervised learning problem, but each object has two "views" or representations [Blum and Mitchell, 1998]. Two classifiers are learned from labeled training examples, one from each view, but with the added constraint that the two classifiers should agree on the available unlabeled data. There are many algorithms for co-training, e.g., [Blum

Name	Rule
Attachment Rule	$\forall\, E, P, D\ attached(D, E)\ \supset\ [projectOf(E, P)$ $\equiv\ projectOf(D, P)][2.20]$
Sender Rule	$\forall\, E, P, H\ sender(E, H)\ \supset\ [projectOf(E, P)$ $\equiv\ projectOf(H, P)][2.20]$
Recipient Rule	$\forall\, E, P, H\ recipient(E, H)\ \supset\ [projectOf(E, P)$ $\equiv\ projectOf(H, P)][2.20]$
User Input Rule	$\forall\, E, P\ userFeedback(E, P)\ \supset\ projectOf(E, P)[2.94]$
Single Project Rule	$\forall\, E\ \exists!\, P\ projectOf(E, P)[1.73]$

Table 4.1: Some rules used in CALO. Markov Logic weights are given in square brackets.

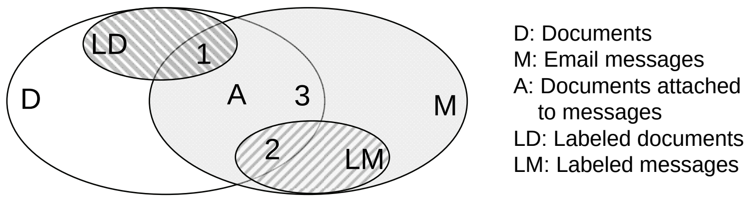


Figure 4.2: Venn diagram showing document-email relations.

and Mitchell, 1998; Nigam and Ghani, 2000].

In the case of CALO, instead of multiple views of objects, we have multiple objects with relations among them. Figure 4.2 is a Venn diagram showing documents, email messages, and the subset of documents attached to email messages. The combination of the document classifier f_D, the email classifier f_M, and the Attachment Rule creates an opportunity for relational co-training. The two classifiers can be trained on their respective labeled data but with the added constraint that for the documents and email messages in region A, the predictions should be consistent. Because this co-training happens through the relations between objects or the chains of relations/rules, we call this learning behavior *relational co-training.*

Through relational co-training, multiple learning components can be integrated into the CALO system. We now describe two designs for an architecture that can support this behavior in a general, unified way.

4.3 Integrating Learning and Reasoning

The starting point for our design is to divide the knowledge of the system into indefeasible knowledge and defeasible (probabilistic) knowledge. Indefeasible knowledge includes events that are observed with certainty. The defeasible knowledge includes all things that are uncertain. For example, in Figure 4.2, document objects, email objects and the $attached(D, E)$ relations are indefeasible knowledge. The Attachment Rule and predictions from the email and document classifiers are defeasible knowledge.

The probabilistic knowledge base is represented in Markov Logic. A Markov Logic knowledge base consists of a set of weighted first-order clauses and a set of constant symbols. (Conceptually, the formulas that appear in the indefeasible knowledge base are treated as having infinite weights in Markov Logic.) The knowledge base defines a probability distribution over the set of possible worlds that can be constructed by grounding the clauses using the constant symbols and then assigning truth values to all of the ground formulas. A possible world is a truth assignment that does not entail a contradiction. Given such a truth assignment α, let $SAT(\alpha)$ be the set of ground formulas that are satisfied. For each formula F, let $w(F)$ be the weight assigned to that formula in the Markov Logic. The score of the truth assignment is defined as the sum of the weights of the satisfied formulas:

$$\text{score}(\alpha) = \sum_{F \in SAT(\alpha)} w(F).$$

The probability of truth assignment is then defined as

$$P(\alpha) = \frac{\exp \text{score}(\alpha)}{\sum_{\omega} \exp \text{score}(\omega)}, \tag{4.1}$$

where ω indexes all possible (contradiction-free) truth assignments. The normalized exponentiated sum on the right-hand side defines a Gibbs distribution that assigns non-zero probability to every possible world. The score of a possible world α is proportional to the log odds of that world according to α.

If we consider the weight $w(F)$ on a particular formula F, we can interpret it as the amount by which the log odds of a possible world will change if F is satisfied compared to a world in which F is not satisfied (but all other formulas do not change truth value).

In Table 4.1, the numbers in the square brackets after the formulas are the weights assigned to them. These weights can be adjusted through a weight learning process, but that is beyond the scope of this chapter. In this chapter, we assume that the weights of the rules are all fixed. Weights for the classifiers' predictions are assigned by taking the predicted probabilities and converting them to log odds according to:

$$w = \log \frac{P(Class|Object)}{1 - P(Class|Object)}, \tag{4.2}$$

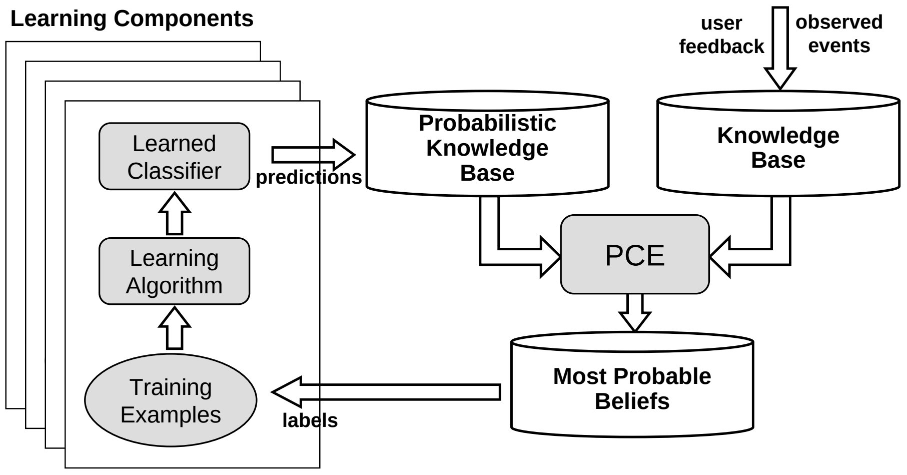


Figure 4.3: The MPE Architecture.

4.3.1 The MPE Architecture

Figure 4.3 shows our first architecture, called the MPE Architecture. The box labeled "PCE" denotes the `Probabilistic Consistency Engine`. This is the inference engine for the Markov Logic system. In this first architecture, its task is to compute the possible world with the highest score. This is known in probabilistic reasoning as the Most Probable Explanation (MPE). The grounded Markov Logic knowledge base defines a Weighted Maximum Satisfiability (Weighted MaxSAT) problem, for which many reasonably efficient solvers are available. Our implementation employs a mix of the MaxWalkSat [Kautz *et al.*, 1997] algorithm for fast, approximate solution and the Yices linear logic solver [Dutertre and de Moura, 2006] for exact solution. The implementation also employs the Lazy-SAT algorithm [Singla and Domingos, 2006] which seeks to compute only those groundings of formulas that are needed for computing the weighted MaxSAT solution.

Under this architecture, the learning components draw their training examples from the MPE and post their predictions into the Probabilistic Knowledge Base. We need to take care, however, that a learning component does not treat its own predictions (which may be very weak) as labels for subsequent training. That is, a prediction such as $projectOf(report1.doc, CALO)$ might be asserted with a probability of 0.51 (log odds of 0.04). But this might then cause the MPE to contain $projectOf(report1.doc, CALO)$. If the document classifier treated this as an ordinary training example, it would rapidly lead to "dogmatic" behavior where the classifier would become absolutely certain of all of its predictions.

To address this problem, the PCE computes a separate MPE for each learning component. When computing the MPE for learning component i, the PCE ignores all assertions that were made by component i. Hence, we call this the "Not Me MPE" for component i.

In order to be integrated into the architecture, each learning component must be able to accept (unweighted) training examples of the form "$TargetPredicate(Obje ct, Class)$" and produce probabilistic predictions of the form "$TargetPredicate(Ob ject, Class)$ with probability p". The architecture ensures that the following three invariants are always true:

- For each learning component, the set of training examples for that component consists exactly of all instances of "$TargetPredicate(Object, Class)$" that appear in the Not Me MPE for that component.

- For each learning component, the learned classifier is always the classifier that results from applying the learning algorithm to the current set of training examples.

- For each learning component, each $Object$, and each $Class$, the Probabilistic KB has an assertion of the form "$TargetPredicate(Object, Class)$" with weight w computed according to Equation 4.2 where $P(Class|Object)$ is the probability assigned to $Class$ by the current learned classifier.

These invariants are maintained by the following infinite loop:

1. Accept new constants and new assertions.
2. Compute the global MPE and the Not Me MPEs for each learning component.
3. Update the training examples for each learning component (if necessary).
4. Recompute the learned classifier for each learning component (if necessary).
5. Update the probabilistic predictions of each learning component (if necessary).
6. Repeat

In practice, these steps are performed concurrently and asynchronously. We employ anytime algorithms for the MPE calculations.

The MPE Architecture supports relational co-training over multiple learning components. Consider the Attachment example shown in Figure 4.2. The Not

Me MPE for the document classifier includes: (1) labeled examples from LD; (2) documents in region 2 with class labels propagated from email labels via the Attachment Rule; and (3) documents in region 3 with class labels propagated from the predicted email labels by the email classifier. The Not Me MPE for the email classifier is similar.

Then both classifiers will be retrained based on their own Not Me MPEs. Each classifier will be influenced by the predictions from the other classifier, as transmitted by the Attachment Rule. This may cause the classifiers to change some of their predictions. This will result in changes in the Not Me MPEs and, therefore, additional changes in the classifiers, which will cause the process to repeat.

Experiments have revealed that this process can loop. The predicted class labels of some of the "unsupervised" documents and email messages can flip back and forth forever. To prevent this, we relax slightly the enforcement of the third invariant. After a classifier has been retrained, if its predicted probability for an example does not change by more than ϵ (e.g., 0.05), then the change is ignored. Obviously, setting ϵ large enough will cause the iterations to terminate.

4.3.2 The Marginal Probability Architecture

The MPE Architecture was initially adopted because the MPE computation is relatively efficient. However, there are several problems with this design. First, there are often "ties" in the computation of the MPE, because there can be many literals that can be either true or false without changing the score of the possible worlds. We attempt to identify and ignore such literals, but this is difficult in general. Consequently, some of the training examples can receive random labels, which can hurt performance. Second, each new classifier that is added to CALO requires an additional Not Me MPE computation. Since our long-term goal is to have 10-20 classifiers, this will slow down the reasoning by a factor of 10-20. Finally, from a learning perspective, this approach to joint training is known to produce biased results. The MPE approach is directly analogous to the use of the Viterbi approximation in training hidden Markov models and to the k-Means clustering algorithm for training Gaussian mixture models. In all of these cases, it is well-established that the resulting model fit is biased and does not maximize the likelihood of the training data.

These considerations led us to develop the Marginal Probability Architecture, which is shown in Figure 4.4. The basic structure is the same. The key change is that instead of computing the MPE, the PCE computes the marginal probability of each ground instance of the target predicates for the various classifiers. This is the probability $P(TargetPredicate(Object, Class))$ given the contents of both the Knowledge Base and the Probabilistic Knowledge Base.

In addition, instead of standard training examples, each learning algorithm that is integrated into the architecture must be able to accept *weighted training examples* of the form "$TargetPredicate(Object, Class)$ with probability p" and produce probabilistic predictions of the same form. The architecture ensures that the following three invariants are always true:

- For each learning component, the set of training examples for that component

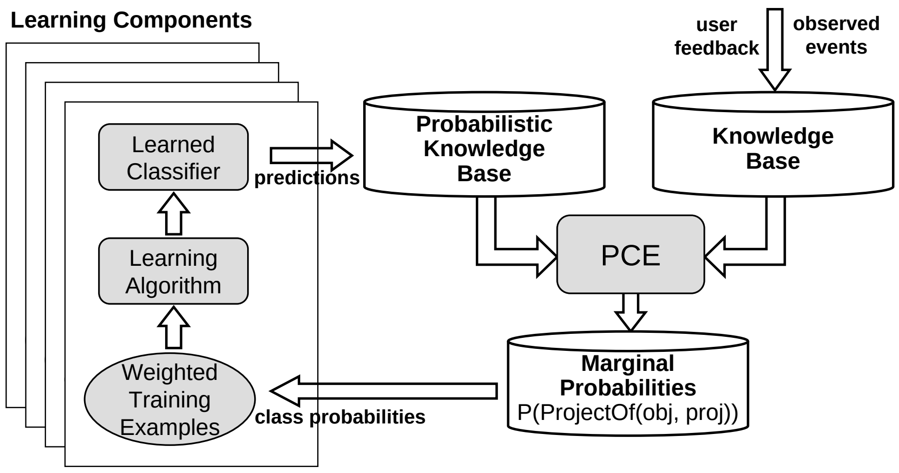


Figure 4.4: The Marginal Probability Architecture.

consists exactly of the set of all groundings of the form "$TargetPredicate(Object, Class)$" weighted by the marginal probability of those ground instances as computed by the PCE.

- For each learning component, the learned classifier is always the classifier that results from applying the learning algorithm to the current set of weighted training examples.

- For each learning component, each $Object$, and each $Class$, the Probabilistic KB has an assertion of the form $TargetPredicate(Object, Class)$ with weight w computed according to Equation 4.2 where $P(Class|Object)$ is the probability assigned to $Class$ by the current learned classifier.

The PCE computes the marginal probabilities by applying the Lazy MC-SAT algorithm [Poon and Domingos, 2006]. MC-SAT is a Markov Chain Monte Carlo algorithm based on slice sampling. It generates a sequence of possible worlds that (after convergence) is drawn from the distribution defined by Equation 4.1. Lazy MC-SAT extends the MC-SAT algorithm to instantiate ground clauses only as needed. In our implementation, the Lazy MC-SAT sampling process is constantly running in the background. Because new evidence and new or updated predictions are continually being added to the indefeasible knowledge base and the probabilistic knowledge base, the MC-SAT process probably never converges. Consequently, we base our probability estimates on the 200 most recent samples from this process.

Note that in this design, the predictions of each learning algorithm influence its own weighted training examples. If the learning algorithm maximizes the weighted log likelihood of the training examples, then this architecture can be viewed as an implementation of the EM algorithm [McLachlan and Krishnan, 1997] applied to semi-supervised learning, and it will converge to a local maximum in the data likelihood.

The Marginal Probability Architecture also supports the relational co-training over multiple learning components. In the Attachment example, the document classifier will be trained with (1) labeled examples from LD; (2) its own probabilistic predictions on unlabeled documents that are not attached to any email messages; (3) documents in region 2 with class probabilities propagated from email labels via the Attachment Rule; and (4) documents in region 3 with class probabilities combining the predictions from both the document classifier and the email classifier. In this example, the marginal probabilities of any document in region 3 computed by the PCE are equivalent to the probabilities obtained by multiplying the predicted probabilities from the two classifiers for each class and renormalizing so that they sum to 1. The email classifier is trained similarly. Then the trained classifiers will predict the projects of all unlabeled objects and assert the weighted predictions into the PCE. Again, new marginal probabilities will be computed and fed to the learning components. This process will iterate until the changes in the marginal probabilities are smaller than ϵ.

In practice, in the CALO system, new observations are arriving continually and both the MPE and the marginal probability computations are only approximate, so the whole learning and reasoning process never terminates.

4.4 Experimental Evaluations

In order to compare the effectiveness of the two architectures and demonstrate the performance boost through relational co-training, we have conducted two sets of experiments. In the first set of experiments, we study the behavior of the classifiers under the two architectures, both separately and combined together, and compare their performance. In the second set of experiments, we demonstrate the performance boost through relational co-training with different sets of rules on a dataset collected using TaskTracer.

4.4.1 MPE Architecture vs. Marginal Probability Architecture

We conducted our first set of experiments on a dataset with real objects but synthetic attachment relations. This dataset was collected from 14 CALO users (all knowledge workers). These users manually labeled their own files into 4 projects. Each of them labeled about 100 files. The results presented here are averaged across all 14 users. Unfortunately, these files do not include email messages, nor are any of the files attached to email messages. Instead, we randomly split the files for each user into sets D and M of equal size to simulate documents and emails. Then we generated a set A of attachments by randomly pairing files from D and M under two constraints: (1) each pair contains one file from D and one file from M; (2) two files in a pair must be labeled with the same project. By doing this, we enforce the Attachment Rule to be perfectly true in our experiments. In the experiments, we vary the size of A to contain 20%, 30%, 40%, 50% and 60% of the files.

A subset of D was randomly chosen to be Labeled Documents (LD), and similarly a subset of M was randomly chosen to be Labeled Emails (LM). All other files in D and M are treated as unlabeled. We varied the sizes of LD and LM to contain 4, 8, 12, and 16 files.

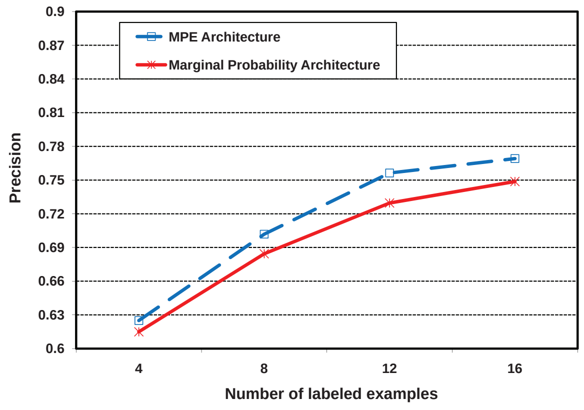
Two widely-used learning algorithms, Naive Bayes and Logistic Regression, were examined in these experiments. Because our aim is to create a general architecture in which any learning algorithm can be used, we did not tune the two algorithms. They are used as black boxes that support the standard learning interfaces: telling examples, training classifier, and predicting examples. The files and emails are represented using Bag-Of-Words features. The accuracy of the final classifiers over all unlabeled documents/emails was measured at convergence. In each single experiment, the two classifiers are trained using the same learning algorithm.

4.4.1.1 Results - Single Classifier

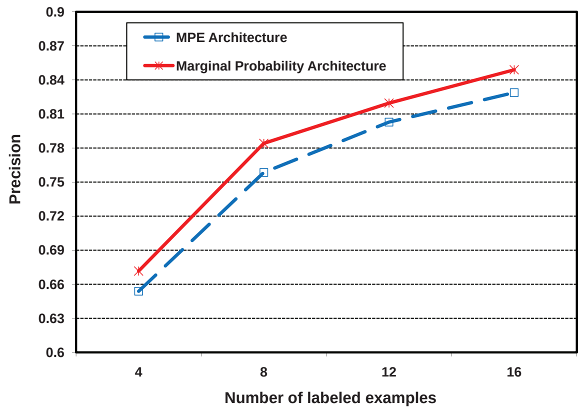
Before we can evaluate the effects of relational co-training among multiple classifiers, we want to make sure that if there is only one classifier in the system, the application of the reasoning layer will not result in any weird behavior.

In fact, it is quite interesting to examine the behavior of a single classifier under our two architectures. Assume we only have one classifier, the document classifier. Under the MPE Architecture, the Not Me MPE contains only the labeled documents in LD, and it will not change. That means the classifier works exactly as if there is no reasoning layer. Therefore, the performance of the single classifier under the MPE Architecture can be treated as the baseline measure. Under the Marginal Probability Architecture, the classifier will be trained with the labeled documents in LD and its own probabilistic predictions on the unlabeled documents in D. The training-and-predicting iterations will continue until convergence.

Figure 4.5 shows the learning curves of the single classifier under the two architectures. We can see that both learning algorithms show expected learning behaviors under either architecture. Under the Marginal Probability Architecture, Logistic Regression performs better than baseline, while Naive Bayes performs worse. However, the differences are small enough that we can claim it is safe to integrate single classifiers into the reasoning layer.



(a) Naive Bayes



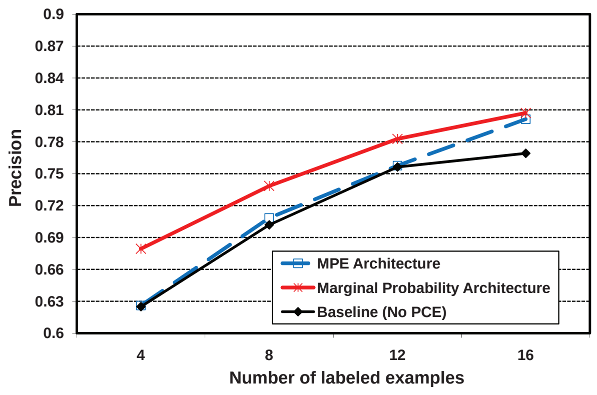
(b) Logistic Regression

Figure 4.5: Learning Curves - Single classifier.

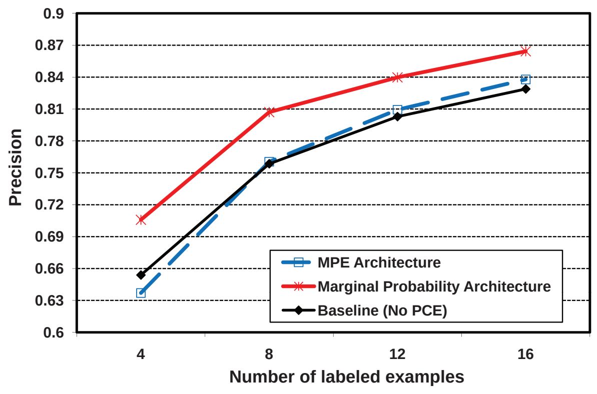
4.4.1.2 Results - Two Classifiers

Now we integrate two classifiers, the document classifier and the email classifier, to test their performance under the simulated Attachment scenario.

Figure 4.6 shows the learning curves of the classifiers under the two architectures when 40% of the documents and emails are attached to each other. Note that the classifiers' performance under the MPE Architecture is very close to the



(a) Naive Bayes

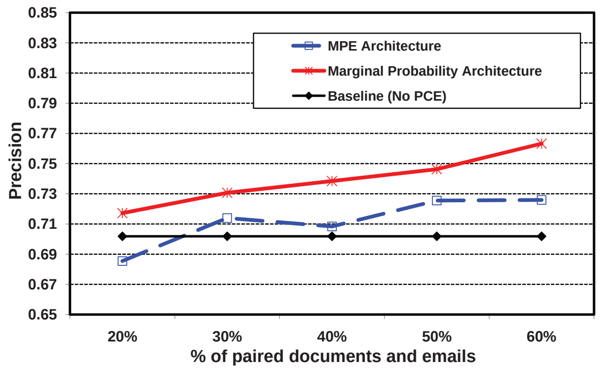


(b) Logistic Regression

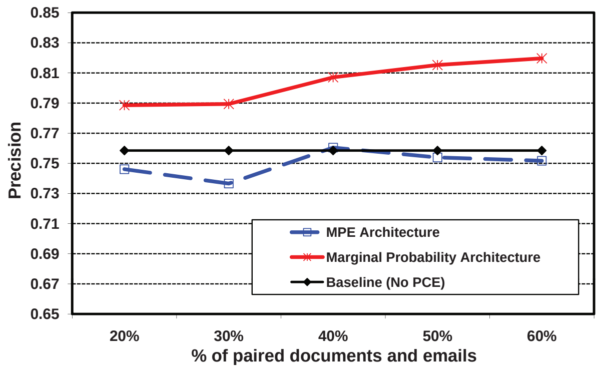
Figure 4.6: Learning Curves - Two classifiers.

Baseline performance. Under the Marginal Probability Architecture, however, we observe an improvement over the Baseline performance, and Logistic Regression benefits from relational co-training more than Naive Bayes.

Figure 4.7 shows how the performance of the "integrated" classifiers changes with the size of the attachment set A. The number of labeled examples is fixed at 8 in this experiment. As expected, the performance improvement becomes larger when the number of attached pairs increases, because more attached pairs



(a) Naive Bayes



(b) Logistic Regression

Figure 4.7: Co-training performance with different size of the attachment set A.

lead to more co-training opportunities. Under the MPE Architecture, this trend is not so obvious, and the performance is close to the baseline even when there are more attachment pairs. Under the Marginal Probability Architecture, on the other hand, for both Naive Bayes and Logistic Regression, performance goes up with more co-training opportunities, and it is significantly better than the baseline performance.

4.4.2 Experiments on TaskTracer Dataset

We conducted our second set of experiments on a dataset collected using the TaskTracer system. The goal of these experiments is to demonstrate how the performance of the classifiers can be improved through relational co-training with variety of sets of hand-written rules. Unlike previous experiments in which the Attachment Rule is perfectly correct on the data, the rules used in these experiments are not 100% correct on the data and thus have finite Markov Logic weights.

4.4.2.1 Data Collection

As mentioned in Chapter 2, TaskTracer records user's activities on the computer, including accessing files, receiving emails, attaching files to emails, storing files into folders, and so on. It also associates user's files/emails with tasks. The dataset was collected from a user who has run TaskTracer for several years. We collected all files and incoming emails associated with four different tasks, as well as the *attached* relations between them. We also collected the folders that these files are filed into with $fileIn$ relations, and the contacts that are senders of the emails with *sender* relations. Table 4.2 shows how many objects and relations of each type are collected.

The rules employed in the experiments are shown in Table 4.3. These rules can be rendered in English as "files and emails attached to each other should associate with same task", "files and folders that they are filed in should associate with same task", "emails and contacts that send them should associate with same task".

Object/Relation	**Number**
File Object	425
Email Object	827
attached Relation	154
$fileIn$ Relation	422
sender Relation	603

Table 4.2: TaskTracer dataset statistics.

Name	**Rule**
Attachment Rule #1	$fileHasTask(f,t) \wedge attached(f,e)$ $\Rightarrow emailHasTask(e,t)$ [1.69]
Attachment Rule #2	$emailHasTask(e,t) \wedge attached(f,e)$ $\Rightarrow fileHasTask(f,t)$ [1.61]
Foldering Rule #1	$fileHasTask(f,t) \wedge fileIn(f,fo)$ $\Rightarrow folderHasTask(fo,t)$[2.32]
Foldering Rule #2	$folderHasTask(fo,t) \wedge fileIn(f,fo)$ $\Rightarrow fileHasTask(f,t)$[0.78]
Sender Rule #1	$emailHasTask(e,t) \wedge sender(e,c)$ $\Rightarrow contactHasTask(c,t)$[3.22]
Sender Rule #2	$contactHasTask(c,t) \wedge sender(e,c)$ $\Rightarrow emailHasTask(e,t)$[$-0.08$]

Table 4.3: Rules used in the experiments on the TaskTracer dataset. Weights are learned from the data.

Each rule in Table 4.3 has its own Markov Logic weight, as shown in the squared brackets. These weights are learned from the dataset by running a simple weight learning procedure: we first count the number of positive (P) and negative (N) examples for a rule, then compute the weight as the log odds: $log(P/N)$.

Based on the experimental results in Chapter 4.4.1, we chose the Marginal Probability Architecture as the architecture and Logistic Regression as the learning algorithm for training classifiers in these experiments.

4.4.2.2 Experiment Design

The experiments simulate the following real usage scenario: the user initially labels some files and emails to train a document classifier f_D and an email classifier f_E with or without the PCE; then these two classifiers make predictions on newly observed files and incoming emails. We want to show that, by integrating the classifiers using the PCE and adding relations, the performance of the classifiers can be improved. Since the relations are obtained for no cost as the system observes the user's actions (e.g., TaskTracer and CALO), the performance boost is gained for "free".

We identify three scenarios in training the two classifiers:

1. **Baseline**: This is the traditional supervised learning scenario. f_D is trained on the files labeled by the user, and f_E is trained on the emails labeled by the user.

2. **PCE-Attached**: We know that some of the user-labeled files are attached to

emails, and some of the user-labeled emails have files as attachments. When f_D and f_E are integrated via the PCE and these *attached* relations are also asserted, the labels of the labeled files can be propagated to the unlabeled emails to which they are attached, and the labels of the labeled emails can be propagated to the unlabeled files that are attached to them [1]. Then the two classifiers are updated in the EM-style as we described in Chapter 4.3.2.

3. **PCE-All**: In addition to the *attached* relations, we assert the $fileIn$ and *sender* relations for the labeled files/emails into the PCE. Thus, the labels on files will be propagated to the folders that contain those files and labels on email messages will be propagated to the contacts that sent those messages. Then the labels will propagate to the other files inside folders and other emails sent by the contacts, as well as additional emails/files through the *attached* relations. This will bring more unlabeled files/emails that are connected to the originally labeled files/emails (through either a single rule or a chain of rules) into the relational co-training.

In order to simulate these three scenarios, we randomly split the dataset into three parts: *TRAIN1*, *TRAIN2* and *TEST*, with 25%, 25% and 50% of the total data. The idea is that the files in *TRAIN1* and emails in *TRAIN2* are treated as labeled objects, and the files/emails in *TEST* are only used to test the trained classifiers. The settings of the three scenarios are summarized in Table 4.4. Not all files in *TRAIN1* and emails in *TRAIN2* are provided as labeled examples. Instead,

[1]Since the Attachment Rules are probabilistic, these propagated labels are "soft" labels (probability distributions over all four tasks).

we randomly pick 4, 8, 16 and 32 of them to label in order to generate learning curves.

The traditional way of testing classifiers is to have them make predictions for the files and emails in $TEST$ one by one. However, in the CALO system, the prediction for a new object does not come directly coming from the classifier. Instead, the classifier makes a prediction and asserts it into the PCE. The PCE then assigns labels to the new object based on its marginal probability computation. In the case where an email with an attached file is received, f_E will predict for the email and f_D will predict for the attached file, and both predictions will be asserted into the PCE which will then make the final predictions for the email and the file together. In our experiments, we implement both testing protocols (named *SimplePredict* and *PCEPredict*) and compare them.

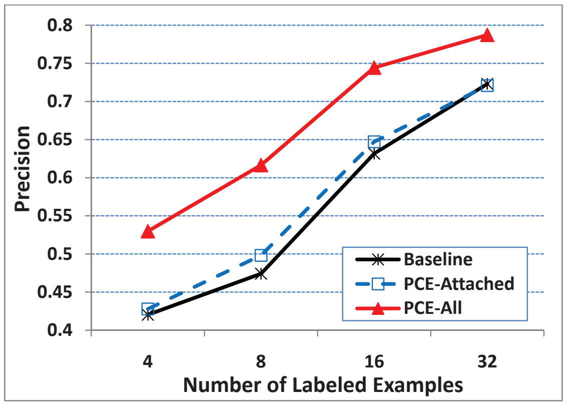
4.4.2.3 Results - Comparison of Three Scenarios

Figure 4.8 shows the learning curves of the two classifiers in the three scenarios. The classifiers are tested using the *PCEPredict* protocol. Both classifiers work better for the **PCE-Attached** scenario than for **Baseline**, but the improvements are relatively small. The reason is that the *attached* relation is quite sparse.[2] Even though the relational co-training does help, the chance of co-training is small under the **PCE-Attached** setting, since only a few unlabeled objects are asserted into the PCE.

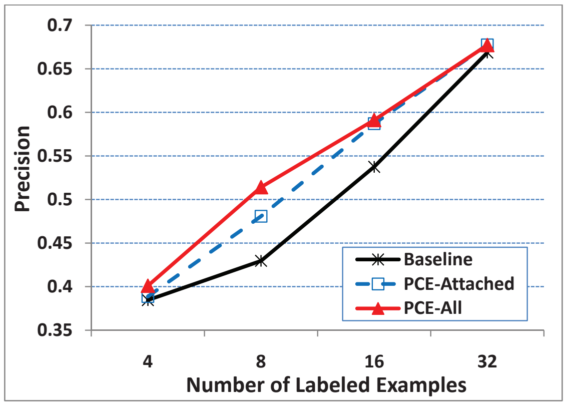
[2]In the dataset, 135 of 827 emails (16.3%) have attachments, and 96 of 425 files (22.6%) are attached to emails.

Scenario	Objects Asserted	Relations Asserted
Baseline	Labeled files in *TRAIN1*. Labeled emails in *TRAIN2*. No unlabeled objects.	None
PCE-Attached	Labeled files in *TRAIN1*. Labeled emails in *TRAIN2*. Unlabeled emails in *TRAIN1* that have labeled files as attachments. Unlabeled files in *TRAIN2* that are attached to the labeled emails.	*attached* relations
PCE-All	Labeled files in *TRAIN1*. Labeled emails in *TRAIN2*. Unlabeled emails/folders/files in *TRAIN1* that connect to the labeled files. Unlabeled files/contacts/emails in *TRAIN2* that connect to the labeled emails.	*attached* relations $fileIn$ relations *sender* relations

Table 4.4: Experiment setting of the three scenarios.



(a) Document Classifier f_D



(b) Email Classifier f_E

Figure 4.8: Precision of the classifiers under three scenarios.

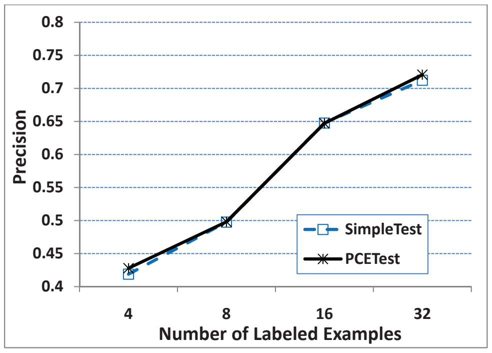
Both classifiers show greater improvement in the **PCE-All** scenario. This is due to the large number of unlabeled objects linked to the labeled objects through the chain of rules including the Foldering and Sender rules. The improvement is much larger for the document classifier than the email classifier. We believe the reason is that the Sender rules are much weaker than the Foldering rules.

This result shows that, by adding relations that connect unlabeled objects to the labeled objects into the PCE, performance of the classifiers can be improved.

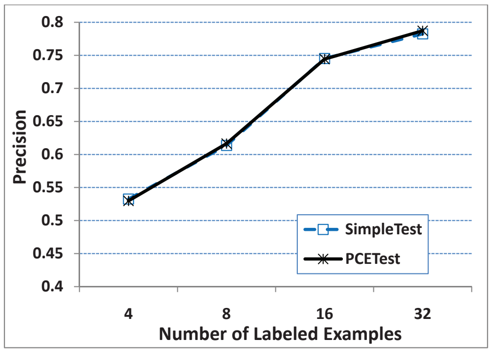
In the CALO system, these relations (attaching file to email, saving file to folder, receiving emails, etc.) are captured automatically without extra user interactions. Therefore, classifiers integrated via the PCE obtain this performance boost for "free".

4.4.2.4 Results - PCEPredict vs. SimplePredict

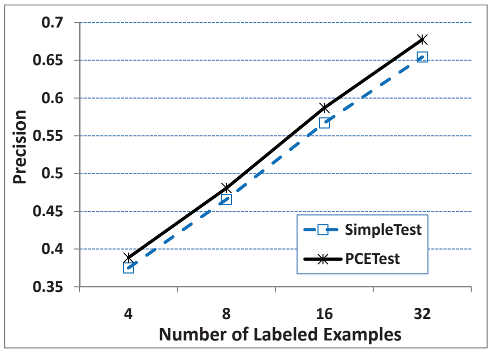
Figure 4.9 illustrates the comparison between the two testing protocols — *SimplePredict* and *PCEPredict*. For the document classifier, they are almost identical. For the email classifier, *PCEPredict* is consistently better than *SimplePredict*. Thus, it is a good idea to let the PCE, instead of the classifiers, make the predictions on new objects.



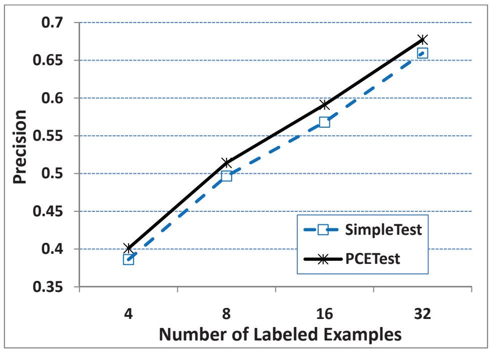
(a) Document Classifier, PCE-Attached Scenario



(b) Document Classifier, PCE-All Scenario



(c) Email Classifier, PCE-Attached Scenario



(d) Email Classifier, PCE-All Scenario

Figure 4.9: Comparison of two testing protocols.

Chapter 5 – Summary and Future Work

5.1 Summary

There are two research contributions of this dissertation. First, we demonstrate, in real-world software systems, how machine learning can be applied to help the user find information on the computer and in the internet. We also explore the idea of creating intelligent user interfaces by enhancing existing user interfaces with machine learning predictions. Second, we propose novel ways of combining multiple separately-engineered learning components to improve overall system performance. Along this direction, our research work covers both combining learning components that perform the same task and combining learning components that perform different tasks. These contributions are presented in the following three applications.

To help computer users quickly access their files, we applied machine learning to records of user activity to recommend file system folders. We described our first application, called FolderPredictor, which reduces the user's cost for locating files in hierarchical file systems. FolderPredictor predicts the file folder that the user will access next, by applying a cost-sensitive online prediction algorithm to the user's previous file access activities. Experimental results show that, on average, FolderPredictor reduces the cost of locating a file by 50%. Perhaps even more importantly, FolderPredictor practically eliminates cases in which a large number of clicks are required to reach the right folder. We also investigated several variations of the cost-sensitive prediction algorithm and presented experimental results showing that the best folder prediction combines an application-specific MRU folder with two folders computed by our cost-sensitive prediction algorithm. An advantage of FolderPredictor is that it does not require users to adapt to a new interface. Its predictions are presented directly in the open/save file dialogs. Users are able to easily adapt to exploit these predictions.

Recommender systems are one of the most popular means of assisting internet users in finding useful online information. Our second application is called STREAM, which is a novel framework for building hybrid recommender systems by stacking single recommendation engines with additional meta-features. In this framework, the component engines are treated as level-1 predictors, with a level-2 predictor generating the final prediction by combining component engine predictions with runtime metrics that represent properties of the input users/items. The resultant STREAM system is a dynamic hybrid recommendation system in which the component engines are combined in different ways for different input users/items at runtime. Experimental results show that the STREAM system outperforms each single component engine in addition to a static equal weight hybrid system. This framework has the additional advantage of placing no restrictions on component engines that can be employed; any engine applicable to the target recommendation task can be easily plugged into the system.

Our third application is called Integrating Learning and Reasoning, which is a part of the CALO project that helps computer users better organize their electronic resources. In the CALO system, multiple machine learning components (e.g., classifiers) are employed to make intelligent predictions. Our goal is to combine these learning components, observed facts, and rules to perform joint learning and reasoning. To achieve this, we employed a Markov Logic inference engine, named the Probabilistic Consistency Engine (PCE). Two designs of the interfaces between the PCE and the learning components, the MPE Architecture and the Marginal Probability Architecture, were described and evaluated. Experimental results showed that both designs can work, but that the Marginal Probability Architecture works better on improving the performance of the learning components when there are co-training opportunities. Further experiments with TaskTracer data show that relational co-training via the PCE greatly improves the performance of the classifiers without requiring the user to label more data.

5.2 Future Work

Beyond the research work discussed in this dissertation, there are several interesting problems that require further research.

5.2.1 Unsupervised FolderPredictor

The FolderPredictor discussed in Chapter 2 is "supervised" — the user is asked to declare what task they are working on at each point in time. However, we find from user feedback that declaring the current task is the biggest overhead of the TaskTracer system. Therefore, it would be very interesting to create an "unsupervised" version of FolderPredictor that makes folder predictions without knowing what task the user is working on. The ultimate goal is to create standalone FolderPredictor software that runs silently in the background and provides high-quality folder predictions to the user without requiring any additional user interaction.

There are several ways to create an unsupervised FolderPredictor. One simple approach is to assume that there is only one task and just apply the prediction algorithms proposed in this dissertation. We have implemented this approach and the initial results look very promising — it shows performance close to the "supervised" version. Another approach is to use provenence information to make folder predictions. Provenance relations are the relations that connect multiple resources, for example, attach files to emails, save files into folders, copy texts from one file to another, and so on. TaskTracer captures these relations and draws provenance graphs from them [Shen *et al.*, 2009a]. We could use these graphs to estimate the distribution of target folder by traversing the provenance graphs that contain currently active resources.

5.2.2 Learn to Select the Best Folder Prediction Algorithm

In Chapter 2.4, multiple folder prediction algorithms are proposed. Experiments show that, on average, the best algorithm is the AMCSP algorithm. However, instead of implementing only the AMCSP algorithm in FolderPredictor, a smarter approach might be to run all the algorithms to make different predictions and select the prediction that has the minimum expected cost. The reasons are:

1. The average performance of other prediction algorithms, comparing with that of Windows Default, is relatively close to the AMCSP algorithm.
2. User behaviors are quite different from each other. The AMCSP algorithm may not be good for everyone.
3. The AMCSP algorithm is not necessarily the best algorithm at each single prediction point.

The challenge is how to select the algorithm. We can formulate this as an online learning problem and train a classifier to predict which algorithm to use at each prediction point. The input features can include historical precision of each algorithm, historical precision of each algorithm for the current task, predicted folders from each algorithm, user's current desktop environment (e.g., windows opened/shown, applications in use, current action is open vs. save), and so on. A possible start point might be [Herbster and Warmuth, 1998].

5.2.3 Discovery and Selection of Runtime Metrics

The ultimate goal of the STREAM project is to enable construction of application-specific hybrid recommendation systems from sets of individual engines by a computer engineer who is not an expert in recommender technology. However, the runtime metrics used in our experiments are manually defined by domain experts who have some knowledge of how properties of the input users/items are related to the quality of the engines' predictions. Automatic or semi-automatic discovery of runtime metrics given the target application and the individual engines would be an interesting subject of future research.

On the other hand, given a set of runtime metrics, we want to further investigate how to identify the best subset. As shown in the experiments, incorporating more runtime metrics does not necessarily increase the performance of the system. There is also a tradeoff between system performance and computational cost. Since the runtime metrics are employed as additional input features to the machine learning problem, it would be natural to apply feature selection techniques to select runtime metrics.

5.2.4 Combining Other Kinds of Learning Components with the PCE

In Chapter 4, the learning components integrated with the PCE are supervised classifiers (document classifier and email classifier), and the interface protocols are defined specifically for this type of learning component. However, there are many other forms of reasoning and problem solving that remain beyond the scope of our current designs. Some that we hope to incorporate in the near future include entity resolution, ranking, and information extraction. The long term goal is to support all kinds of reasoning and problem solving found in AI systems.

5.2.5 Weight Learning and Rule Learning within the PCE

Another direction we want to pursue is to have the PCE automatically adjust the weights of the rules in the Probabilistic Knowledge Base. Humans are good at writing rules but not so good at assigning weights to the rules they write. A solution is to let the PCE assign weights to the hand-written rules by running a weight learning algorithm on the data stored in the PCE. In Chapter 4.4.2, we employed specifically-written weight learning routines that simply count the number of positive and negative examples for each rule. When the rules are not known in advance, we could apply other general purpose weight learning algorithms like the one from Alchemy [Lowd and Domingos, 2007].

Another important direction is to investigate methods for allowing CALO to learn its own rules. There are two ways to do this: (1) employ a Markov Logic rule learning algorithm like the one in Alchemy [Kok and Domingos, 2005] that learns rules and weights together, or (2) run a first-order rule learner like FOIL [Quinlan and Mostow, 1990] to learn the unweighted rules and then separately learn weights for them. Preliminary experiments on the TaskTracer data show that the second method (FOIL + Weight Learning) works better. We can successfully learn rules like the Attachment and Foldering Rules in Table 4.3. Some other interesting rules are also learned, for example, “If a folder contains two files and both files belong to task T, then the folder belongs to task T”.

We anticipate that running weight learning or rule learning procedures every time a new piece of data comes into the PCE will be extremely time-consuming.

Therefore, a good practice will be to run them at night, or when a certain percentage of new data arrived since the last run.

5.2.6 Controlling the Semi-Supervised Learning within the PCE Marginal Probability Architecture

In the PCE experiments on the TaskTracer data presented in Chapter 4.4.2, not all unlabeled objects were included in the EM-style semi-supervised learning (SSL). Instead, we only included those (relatively few) unlabeled objects that were related to the labeled objects by one of the key relations (*attached*, *filedIn*, *sender*). The reason is that when we included a large number of unrelated objects in our preliminary experiments, SSL did not work well and in some cases performed worse than the baseline.

Therefore, the aspect of the experiment design, in which the EM-style SSL was limited only to objects that are meaningfully connected to labeled objects through relations, was important. This is a way of fusing the graph-based approach to SSL [Zhu, 2005] with the EM approach to SSL. A topic for future research is to understand why and to understand how to automatically control SSL within the PCE marginal probability architecture.

Bibliography

[Adomavicius and Tuzhilin, 2005] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transaction on Knowledge and Data Engineering*, 17(6), 2005.

[Alspector *et al.*, 1997] J. Alspector, A. Kolcz, and N. Karunanithi. Feature-based and clique-based user models for movie selection: A comparative study. *User Modeling and User-Adapted Interaction*, 7(4):279–304, 1997.

[Bao *et al.*, 2006] X. Bao, J. Herlocker, and T. G. Dietterich. Fewer clicks and less frustration: Reducing the cost of reaching the right folder. In *Procceedings of 10th International Conference on Intelligent User Interfaces*, pages 178–185, 2006.

[Barreau and Nardi, 1995] D. K. Barreau and B. Nardi. Finding and reminding: File organization from the desktop. *ACM SIGCHI Bulletin*, 27(3):39–43, 1995.

[Basilico and Hofmann, 2004] J. Basilico and T. Hofmann. Unifying collaborative and content-based filtering. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, 2004.

[Basu *et al.*, 1998] C. Basu, H. Hirsh, and W. Cohen. Recommendation as classification: Using social and content-based information in recommendation. *Recommender Systems. Papers from 1998 Workshop, Technical Report WS-98-08*, 1998.

[Bell *et al.*, 2007a] R. Bell, Y. Koren, and C. Volinsky. The BellKor solution to the netflix prize. *KorBell Team's Report to Netflix*, 2007.

[Bell *et al.*, 2007b] R. Bell, Y. Koren, and C. Volinsky. Chasing $1,000,000: How we won the netflix progress prize. *Statistical Computing and Statistical Graphics Newsletter*, 18(2):4–12, 2007.

[Billsus and Pazzani, 2000] D. Billsus and M. J. Pazzani. User modeling for adaptive news access. *User Modeling and User-Adapted Interaction*, 10(2-3):147 – 180, 2000.

[Blum and Mitchell, 1998] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *COLT: Proceedings of the Workshop on Computational Learning Theory*, pages 92–100, 1998.

[Boardman and Sasse, 2004] R. Boardman and M. A. Sasse. Stuff goes into the computer and doesn't come out: A cross-tool study of personal information management. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems*, pages 583–590, 2004.

[Breiman, 1996] L. Breiman. Stacked regressions. *Machine Learning*, 24:49–64, 1996.

[Budzik and Hammond, 2000] J. Budzik and K. J. Hammond. User interactions with everyday applications as context for just-in-time information access. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, pages 44–51. ACM, 2000.

[Burke, 2002] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, November 2002.

[CALO, 2009] CALO. Calo project. http://caloproject.sri.com/, 2009.

[Claypool *et al.*, 1999] M. Claypool, A. Gokhale, T. Miranda, P. Murnikov, D. Netes, and M. Sartin. Combining content-based and collaborative filters in an online newspaper. In *Proceedings of the ACM SIGIR '99 Workshop Recommender Systems: Algorithms and Evaluation*, August 1999.

[Default Folder X, 2009] Default Folder X. St. Clair Software. http://www.stclairsoft.com/defaultfolderx/, 2009.

[Direct Folders, 2009] Direct Folders. Code sector inc. http://www.codesector.com/directfolders.asp, 2009.

[Domingos and Richardson, 2006] P. Domingos and M. Richardson. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[Dourish *et al.*, 2000] P. Dourish, W. K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. B. Terry, and J. Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 18(2):140–170, January 2000.

[Dragunov *et al.*, 2005] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: A desktop environment to support multi-tasking knowledge workers. In *Procceedings of 9th International Conference on Intelligent User Interfaces*, pages 75–82, 2005.

[Dumais *et al.*, 2003] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *Proceedings of the 26th Annual International ACM SIGIR conference on Research and Development in Informaion Retrieval*, pages 72–79, 2003.

[Dutertre and de Moura, 2006] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(t). In *International Conference on Computer Aided Verification (CAV-06), Vol. 4144 of Lecture Notes in Computer Science*, pages 81–94, 2006.

[Dzeroski and Zenko, 2004] S. Dzeroski and B. Zenko. Is combining classifiers with stacking better than selecting the best one? *Machine Learning*, 54(3):255–273, 2004.

[Hastie *et al.*, 2001] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[Herbster and Warmuth, 1998] Mark Herbster and Manfred Warmuth. Tracking the best expert. *Journal of Machine Learning*, 32(2):151–178, 1998.

[Herlocker *et al.*, 1999] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 1999 Conference on Research and Development in Information Retrieval*, pages 230–237, 1999.

[Horvitz *et al.*, 1998] 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 *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265, 1998.

[IMDb, 2009] IMDb. Internet movie database. Downloadable at http://www.imdb.com/interfaces, 2009.

[Jones *et al.*, 2005] W. Jones, A. J. Phuwanartnurak, R. Gill, and H. Bruce. Don't take my folders away! Organizing personal information to get things done. In *CHI-05 extended abstracts on Human factors in Computing Systems*, pages 1505–1508, 2005.

[Jul and Furnas, 1995] S. Jul and G. W. Furnas. Navigation in electronic worlds: Workshop report. *ACM SIGCHI Bulletin*, 29(2):44–49, 1995.

[Kautz *et al.*, 1997] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. *The satisfiability problem: Theory and applications*, pages 573–586, 1997.

[Ko *et al.*, 2005] A. Ko, H. H. Aung, and B. Myers. Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the International Conference on Software Engineering*, pages 126–135, 2005.

[Kok and Domingos, 2005] S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-2005)*, pages 441–448, 2005.

[Konstan *et al.*, 1997] J. Konstan, B. Miller, D. Maltz, J. Herlocker, L. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[Lang, 1995] K. Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the 12th International Machine Learning Conference (ML-95)*, pages 331–339, 1995.

[LeCun *et al.*, 1998] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[Lowd and Domingos, 2007] D. Lowd and P. Domingos. Efficient weight learning for Markov logic networks. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 86–93, 2007.

[Lucene, 2009] Lucene. Apache lucene. http://lucene.apache.org/, 2009.

[Maes, 1994] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, 1994.

[McLachlan and Krishnan, 1997] G. McLachlan and T. Krishnan. *The EM algorithm and extensions.* Wiley, New York, 1997.

[MovieLens, 1997] MovieLens. Movielens data sets. downloadable at http://www.grouplens.org/node/73, 1997.

[Netflix, 2009] Netflix. Netflix prize, http://www.netflixprize.com/, 2009.

[Nigam and Ghani, 2000] K. Nigam and R. Ghani. Analyzing the effectiveness and applicability of co-training. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, pages 86–93, 2000.

[Pazzani, 1999] M. J. Pazzani. A framework for collaborative, content-based, and demographic filtering. *Artificial Intelligence Review*, 13(5-6):393–408, 1999.

[Pietrek, 1995] M. Pietrek. *Windows 95 System Programming Secrets.* IDG Books Worldwide, Inc., Foster City, CA 94404, USA, 1995.

[Poon and Domingos, 2006] H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-2006)*, pages 458–463, 2006.

[Quinlan and Mostow, 1990] J. R. Quinlan and J. Mostow. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Ramezani *et al.*, 2008] M. Ramezani, L. Bergman, R. Thompson, R. Burke, and B. Mobasher. Selecting and applying recommendation technology. In *IUI-08 Workshop on Recommendation and Collaboration (ReColl2008)*, 2008.

[Rennie, 2000] J. Rennie. ifile: An application of machine learning to e-mail filtering. In *KDD 2000 Workshop on Text Mining*, Boston, MA, 2000.

[Rhodes, 2003] B. Rhodes. Using physical context for just-in-time information retrieval. *IEEE Transactions on Computers*, 52(8):1011–1014, 2003.

[Salton and McGill, 1986] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.

[Sarwar *et al.*, 2001] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International World Wide Web Conference*, pages 285–295, 2001.

[Segal and Kephart, 1999] R. B. Segal and J. O. Kephart. Mailcat: An intelligent assistant for organizing e-mail. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 276–282, ACM Press, 1999.

[Segal and Kephart, 2000] R. B. Segal and J. O. Kephart. Incremental learning in SwiftFile. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 863–870. Morgan Kaufmann, 2000.

[Shen *et al.*, 2009a] J. Shen, E. Fitzhenry, and T. Dietterich. Discovering frequent work procedures from resource connections. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, pages 277–286, 2009.

[Shen *et al.*, 2009b] J. Shen, J. Irvine, X. Bao, M. Goodman, S. Kolibaba, A. Tran, F. Carl, B. Kirschner, S. Stumpf, and T. G. Dietterich. Detecting and correcting user activity switches: algorithms and interfaces. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, pages 117–126. ACM, 2009.

[Singla and Domingos, 2006] P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-2006)*, pages 488–493, 2006.

[Stumpf *et al.*, 2005] S. Stumpf, X. Bao, A. Dragunov, T. G. Dietterich, J. Herlocker, K. Johnsrude, L. Li, and J. Shen. Predicting user tasks: I know what you're doing! In *AAAI-05 Workshop on Human Comprehensible Machine Learning*, 2005.

[Teevan *et al.*, 2004] J. Teevan, C. Alvarado, M. S. Ackerman, and D. R. Karger. The perfect search engine is not enough: A study of orienteering behavior in directed search. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems*, pages 583–590, 2004.

[Wang and Witten, 1997] Y. Wang and I. H. Witten. Inducing model trees for continuous classes. In *Proceedings of the 9th European Conference on Machine Learning*, pages 128–137, 1997.

[Witten and Frank, 2005] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques, 2nd Edition*. Morgan Kaufmann, San Francisco, CA, USA, 2005.

[Wolpert, 1992] D. H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[Zhu, 2005] Xiaojin Zhu. Semi-supervised learning with graphs. *Doctoral dissertation, Carnegie Mellon University*, 2005.