A System of Intelligent Agents for Organizing HTML Documents

Zach Cox
ComS 572 Final Project
Fall 2001

Abstract

Document organization is an important part of the broad field of Information Retrieval. Since documents can contain similar content, organizing similar documents on some display can allow a user to gain more desired information. Self-Organizing Maps present a useful method for performing this organization based on content. In this paper, a learning agent-based approach is taken to organize an arbitrary set of HTML documents and display it to the user. A system of four interchangeable agents is presented that promotes flexibility and extensibility.

Introduction

Information Retrieval (Chowdhury 1999) is a very broad area of research. Its goal is mainly to provide efficient and accurate information to a user. For instance, modern Internet search engines use Information Retrieval (IR) techniques to allow web surfers to find web pages containing certain desired information. The user enters a query, and the search engine provides a ranked list of web pages that should provide the information the user is looking for.

A basic object containing information is the document. A document is basically a collection of words that relate to some subject, and this subject can usually be inferred from the words in a document. There are many kinds of documents, in many different formats. In this paper, only HTML documents are considered.

Organizing documents in such a way that aids the user's search for information is a major part of IR. If a certain document were helpful to the user, perhaps similar documents would also be of help. Thus, a way to compare documents and group them together based on their contents is needed.

Recently, the Self-Organizing Map (SOM) has proven extremely useful for organizing documents (Kohonen 2000). SOM is an unsupervised learning method for approximating the distribution of a high dimensional set of data with a low dimensional grid of points (Kohonen 2001). It basically performs a non-parametric regression over the data set and then does non-linear projection from the high dimensional input space to a low dimensional feature space. This projection retains the original structure of the data so that data points near each other in input space are projected onto neighboring regions of the map in feature space.

By representing each document as a vector in a high dimensional space, the SOM can organize the documents on a two dimensional grid of points such that similar documents will be mapped to nearby points on the grid. This organization can direct the user to similar documents.

This project is mainly based on an idea from (Roussinov, Tolle et al. 1999). In their paper, they created a system to retrieve search results from the AltaVista search engine (http://www.altavista.com) and use SOM to organize the documents on a grid for display. Extending this idea to any arbitrary HTML documents can create a more general and flexible system.

A set of interacting agents is defined to handle the different parts of the document organization system, each having its own responsibilities. A learning agent, using SOM, learns about the documents in the collection and organizes them according to their content. Different specific types of these agents can be interchanged, providing different types of behavior. Also, new types of agents can be easily added to the system to experiment with new ideas. The Jsom package was created, in part, to handle the usage of SOM for the project.

Self-Organizing Maps

The SOM (Kohonen 2001) is an unsupervised neural network and can be used for non-parametric regression, non-linear projection, dimensionality reduction, feature extraction, clustering, and visualization depending on the interpretation and use of its results. Figure 1 below shows a typical architecture of the neural network.

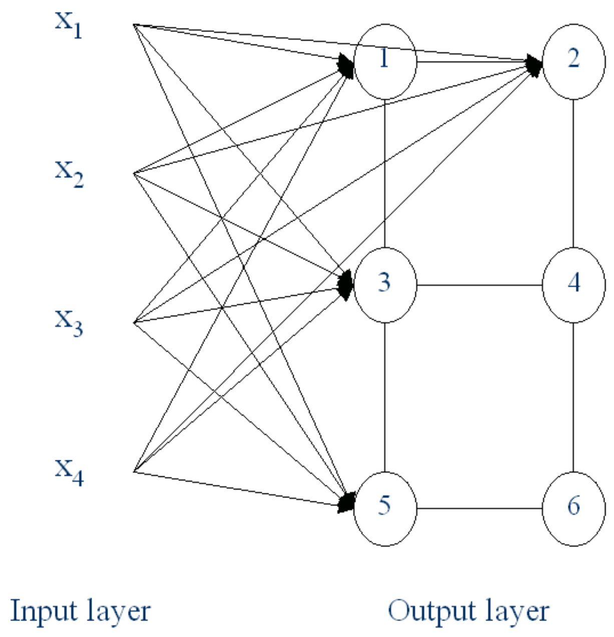


Figure 1. The SOM as an unsupervised neural network. The SOM consists of an input layer, where the input patterns are presented, and an output layer of connected neurons. These neurons exist as discrete points in some space, in this case 2-D, and each neuron has a weight vector of the same dimensionality as the input patterns.

Each neuron, or *node*, in the output layer has two vectors associated with it that exist in two different spaces: feature space and input space. Feature space is the space in which the nodes exist and input space is the space in which the data exists. The first vector describes the position of the node in feature space, and the second represents the weight vector of the neuron. Each neuron has a weight associated with each input variable; the lines in Figure 1 represent these weights, and together they form a vector of the same dimensionality as the input data. These weight vectors can also be represented as points in input space; in this case, they are referred to as *models* since they are the input space representation of the nodes. When referring to the map as a neural network, it is natural to refer to neurons and weight vectors; when referring to points in feature space and input space, nodes and models are used instead.

The goal of the SOM algorithm is to adjust the weight vectors of the neurons to match the training data. One can also think of this as positioning the models of the nodes in input space to match the distribution of the data. The algorithm attempts to do this so that nodes near each other in feature space will have models near each other in input space. Positioning the models to approximate the data distribution can be thought of as *non-parametric regression*, much like discretized principal curves (Hastie and Stuetzle 1989). The trained map can then perform *non-linear projection* of data points in input space, by finding the nearest model to the data point and projecting the point onto the model's node in feature space.

Note that the SOM algorithm is not restricted to working with strictly numerical data. Patterns can be anything, such as text strings or documents. The algorithm just needs to create model patterns for the nodes, and make these models approximate the distribution of the data in an ordered fashion. The algorithm is iterative, and fairly simple:

```
For each iteration $k$
    Pick a data point $\mathbf{x}(k)$
    Find node $c$ with closest model $\mathbf{m}_i(k)$ to $\mathbf{x}(k)$
        $c = \arg\min_i \left[ \left\| \mathbf{x}(k) - \mathbf{m}_i(k) \right\| \right]$
    Update each model $\mathbf{m}_i(k)$
        $\mathbf{m}_i(k+1) = \mathbf{m}_i(k) + n(c,i,k) * [\mathbf{x}(k) - \mathbf{m}_i(k)]$
```

So during iteration *k*, the algorithm finds the node *c* with the most similar model to data point *x* in the input space. Then all of the models are updated by moving them towards the data point *x*. The neighborhood function *n(c,i,k)* returns a scaling value that determines how far each model moves toward *x*. The model of node *c* moves the farthest towards *x* while models of nodes farther away from *c* in feature space move less towards *x* in input space (models of nodes "very far away" from *c* may not move at all). Also, *n(c,i,k)* goes to zero as *k* increases so that the models "learn" quickly in the beginning and then are more finely adjusted in later iterations.

Thus, *n(c,i,k)* decreases with both increasing distance between nodes *c* and *i* in feature space, and with increasing *k*. It is this property of the neighborhood function that ensures neighboring models in input space will belong to neighboring nodes in feature space, and the non-linear projection from input space to feature space will retain the original structure of the data.

Many other ways to position the models in input space exist; thus, the previous algorithm is not unique. Many variants on this original SOM algorithm can be found in the literature (Kohonen 2001), and will not be presented here.

Jsom

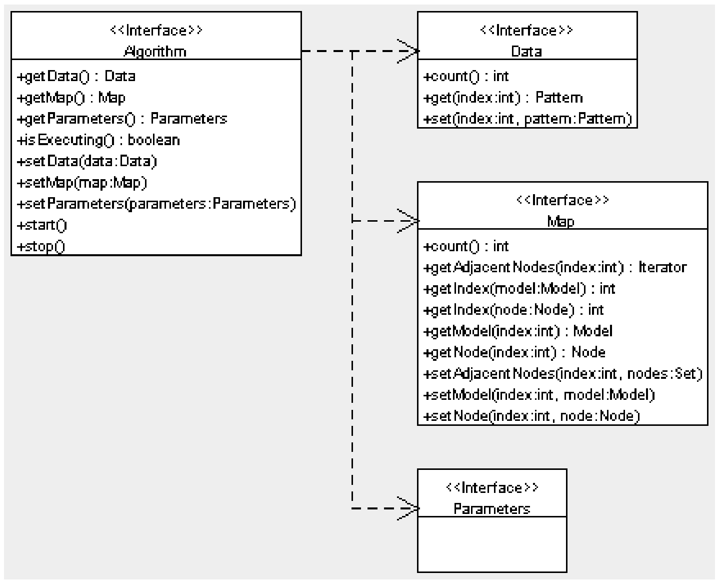
Jsom is a Java-based package for working with Self-Organizing Maps. Utilizing object-oriented and smart software design principles (Gamma, Helm et al. 1995) (Bloch 2001), it is intended to be incredibly easy to use in a variety of applications, from pure number crunching to interactive demos, while also being very flexible and supportive of customization. A collection of Java interfaces specifies the different parts involved in the SOM algorithm, and is backed by fully functioning default implementation classes, ready for use by the user. Custom implementations are easily integrated into the existing class structure since the API refers to interfaces, not concrete classes.

There are four main interfaces in the Jsom package: Algorithm, Data, Map, and Parameters. These represent exactly what they are named for: the SOM algorithm, the training data, the map (nodes and models), and any parameters used by the algorithm, respectively. Figure 2 below shows a UML diagram for these interfaces.

Many concrete implementations of these classes, performing basic functionality, are also provided in the Jsom package. For instance, the following code is a complete program to read the training data, initial models, node positions, and node adjacency from files, train the map, and print out the resulting models:

```
import java.io.*;
import jsom.*;
import jsom.data.*;
import jsom.algorithm.*;
import jsom.algorithm.parameters.*;

public class jsomTest {

        public jsomTest() {
                Data data = new FileDoubleData(new File("data.txt"));
                DoubleData nodes = new FileDoubleData(new File("nodes.txt"));
                DoubleData models = new FileDoubleData(new File("models.txt"));
                IntegerData adjMatrix = new FileIntegerData(new File("adjMatrix.txt"));
                Map map = new BasicMap(nodes, models, adjMatrix);

                int iterations = 10000;
                double initialValue = 0.9d, finalValue = 0.0001d;
                Metric metric = EuclideanDoubleMetric.getInstance();
                ModelUpdater updater = FlowDoubleModelUpdater.getInstance();
                LearningRate rate = ExponentialLearningRate.getInstance(initialValue,
                        finalValue, iterations);
                Neighborhood neighborhood = GaussianDoubleNeighborhood.getInstance(rate);
                PatternSelector selector = BasicPatternSelector.getInstance();
                Parameters params = new FlowParameters(iterations, metric, updater,
                        neighborhood, selector);

                Algorithm algorithm = new FlowAlgorithm(data, map, params);
                algorithm.start();

                System.out.println("models = ");
                for (int i=0; i<map.count(); i++) {
                        System.out.println("" + i + " = " + map.getModel(i).getPattern());
                }
        }

        public static void main(String args[]) {
                new jsomTest();
        }

}
```

Specifying these components as interfaces instead of classes creates a very flexible API. For instance in the above code example, a `FileDoubleData` instance is created, which implements the `Data` interface, and is given to the `Algorithm`. Any other concrete `Data` implementation could have been used to provide training data to the algorithm. For example, `BasicDoubleData`, which just wraps a `double[][]`, could have also been used. Or, a user could define a custom `Data` implementation, and give that to the algorithm as training data. This custom implementation could access any data source, such as a database or some existing data object.



<<Interface>>
Algorithm
+getData() : Data
+getMap() : Map
+getParameters() : Parameters
+isExecuting() : boolean
+setData(data:Data)
+setMap(map:Map)
+setParameters(parameters:Parameters)
+start()
+stop()
<<Interface>>
Data
+count() : int
+get(index:int) : Pattern
+set(index:int, pattern:Pattern)
<<Interface>>
Map
+count() : int
+getAdjacentNodes(index:int) : Iterator
+getIndex(model:Model) : int
+getIndex(node:Node) : int
+getModel(index:int) : Model
+getNode(index:int) : Node
+setAdjacentNodes(index:int, nodes:Set)
+setModel(index:int, model:Model)
+setNode(index:int, node:Node)
<<Interface>>
Parameters

Figure 2. UML diagram for the four main interfaces in the Jsom package.

The interfaces in Figure 2 also reference several other interfaces: Pattern, Node, and Model. These interfaces represent patterns, nodes, and models of the SOM, respectively, and are shown in Figure 3. Again, Jsom can use any class implementing the `Pattern` interface. A default implementation, `DoublePattern` which just wraps a `double[]`, is provided with Jsom. However, a class representing a text string could also implement `Pattern` and be used by Jsom. In this case, a custom `Algorithm` would also need to be created, to handle the new type of `Pattern`.

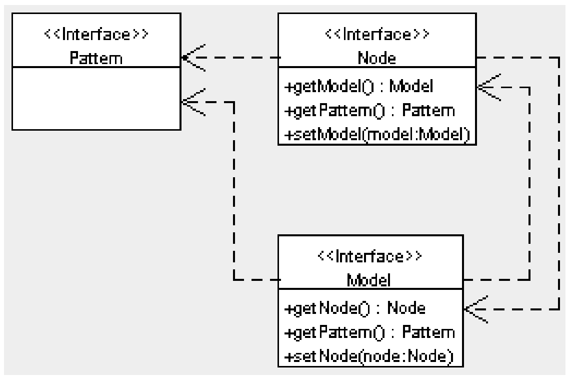


Figure 3. UML diagram for basic data structure interfaces.

Jsom also provides support for event notification (via the `AlgorithmListener` interface and `AlgorithmEvent` class), multi-threaded use (via the `RunnableAlgorithm` wrapper class), and more fine-controlled algorithms (via the `ControllableAlgorithm` interface). A complete software design description is beyond the scope of this paper, although the above descriptions give a flavor of the overall API design. For more information, please see the Jsom website at http://www.public.iastate.edu/~zcox/jsom/.

Vector Space Representation of Documents

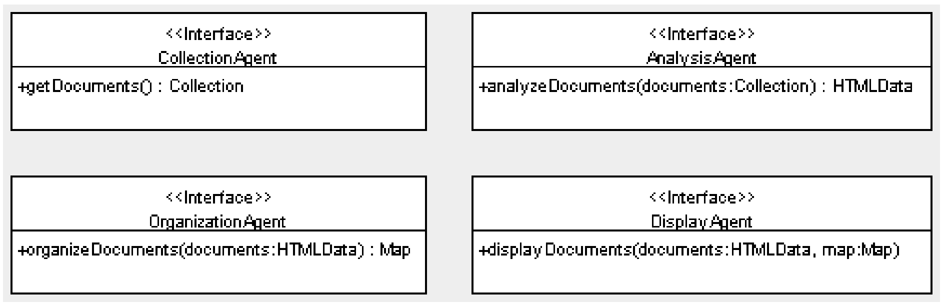
A document is a collection of words. A vector, whose components represent the frequency of words in the document, can represent a document numerically. In order to compare documents by their vectors, all of the vectors must be of the same length, and their elements must represent frequencies of the same words. Therefore each element represents a unique word in the entire collection of documents, which each individual document may or may not contain. Typically, there are many zeros in these vectors since documents tend to contain different words.

In any sizable collection of documents, this set of possible words is very large. This makes the dimensionality of the document vectors very large as well. Computationally, working with vectors with dimensionality in the tens to hundreds of thousands is not efficient. There are, however, methods of greatly reducing the dimensionality of the document vectors.

(Kaski 1998) recently published a method for projecting vectors to a lower dimensionality such that similarity calculations between two vectors is the same in both spaces. This results in greatly reduced dimensionality of the document vectors, while still retaining information used to compare the documents. This method is implemented this project.

System of Agents

There are several different parts to the document organization project, and an appropriate agent can carry out each part. These different agents are specified as Java interfaces to promote flexibility; different implementations can then be used safely to provide different overall functionality. The UML diagram in Figure 4 shows these interfaces.



<<Interface>>
CollectionAgent
+getDocuments() : Collection
<<Interface>>
AnalysisAgent
+analyzeDocuments(documents:Collection) : HTMLData
<<Interface>>
OrganizationAgent
+organizeDocuments(documents:HTMLData) : Map
<<Interface>>
DisplayAgent
+displayDocuments(documents:HTMLData, map:Map)

Figure 4. UML diagram of the four agent types.

The CollectionAgent is responsible for obtaining the HTML documents, and providing them to the rest of the system. The documents can be from any source, as long as they are formatted in HTML. Figure 5 below shows two possible implementations of the CollectionAgent interface. AltaVistaCollectionAgent queries the AltaVista Internet search engine with a user-provided String, and provides the resulting HTML documents in a Collection returned by the getDocuments() method. DirectoryCollectionAgent, on the other hand, returns all of the HTML files in a specified directory. The documents are referred to by URL objects so that local copies are not needed, as this could consume a great deal of memory. Since both classes implement the CollectionAgent interface, they can be used interchangeably to provide different functionality.

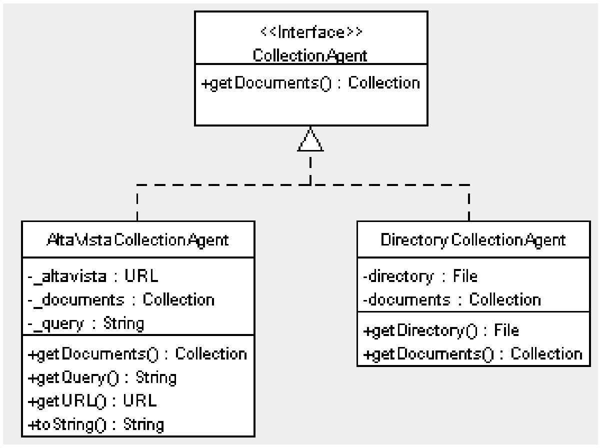


Figure 5. Two different implementations of the CollectionAgent interface.

The AnalysisAgent is responsible for creating the vectors for each document, based on their word contents. Two implementations were created in this project, as shown in Figure 6. BasicAnalysisAgent simply creates a weighted histogram, or word frequency vector, for each document. ProjectionAnalysisAgent performs the projection described in (Kaski 1998) to reduce the dimensionality of the vectors. It should also be noted that these two agents only use the text of the HTML documents, not the formatting tags. The javax.swing.text.html package provides excellent support for HTML parsing.

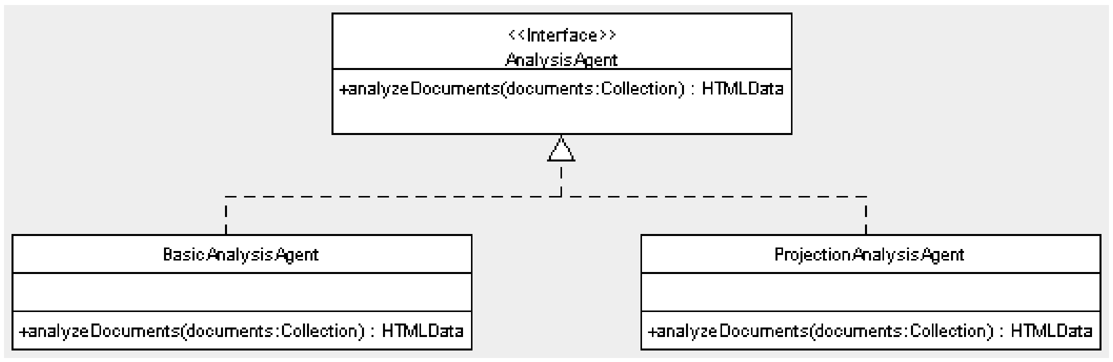

<<Interface>>
AnalysisAgent
+analyzeDocuments(documents:Collection) : HTMLData
BasicAnalysisAgent
+analyzeDocuments(documents:Collection) : HTMLData
ProjectionAnalysisAgent
+analyzeDocuments(documents:Collection) : HTMLData

Figure 6. Two different implementations of the AnalysisAgent interface.

Figure 6 also refers to the HTMLData interface. HTMLData extends the Jsom Data interface, to support HTMLPattern objects. HTMLPattern stores a double[] as its document vector, as well as some other information. These classes are shown below in Figure 7. Because they extend basic Jsom types, they can be integrated into Jsom and used in the SOM algorithm.

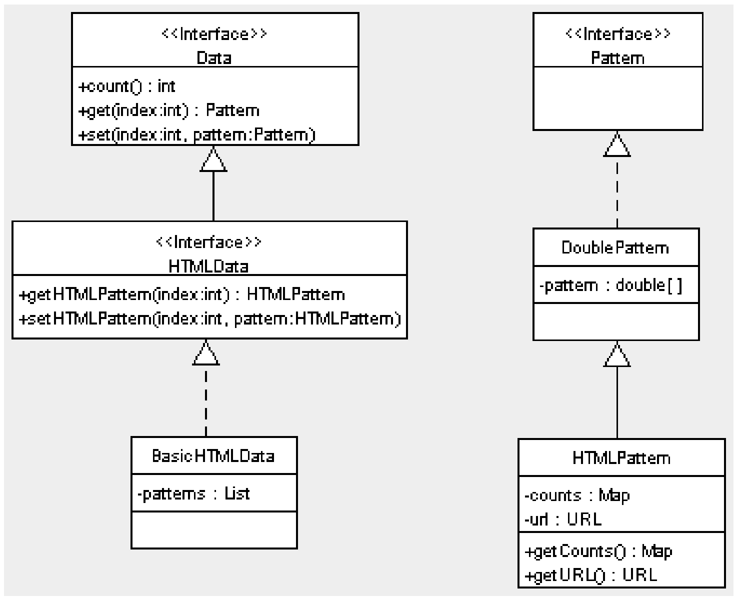
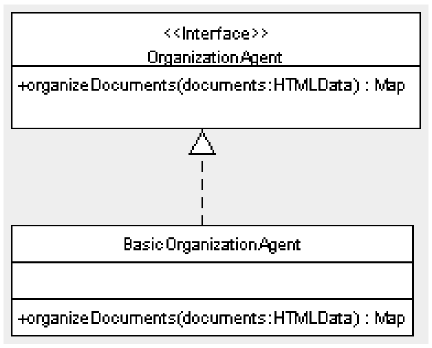


Figure 7. Classes for storing the HTML documents as patterns.

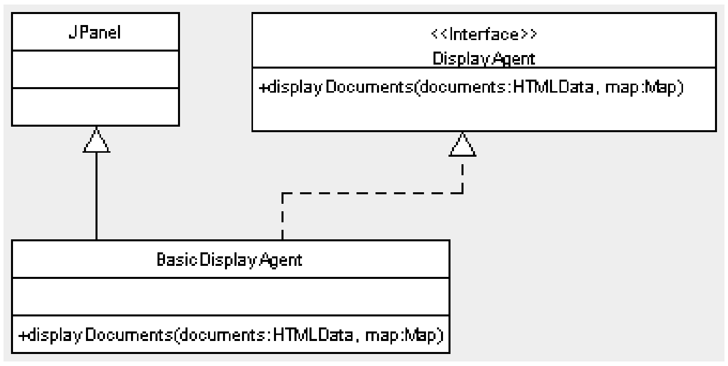
Once the documents are converted to vector form, an OrganizationAgent uses a SOM to organize the documents onto a two-dimensional grid. BasicOrganizationAgent, shown in Figure 8, uses the Jsom package. It returns a Jsom Map object, which represents the nodes and models of the trained map. This agent effectively learns about the documents in the collection, and organizes them according to their content. Documents with similar content are assigned to nearby nodes in the map.



<<Interface>>
OrganizationAgent
+organizeDocuments(documents:HTMLData) : Map
BasicOrganizationAgent
+organizeDocuments(documents:HTMLData) : Map

Figure 8. Implementation of the OrganizationAgent interface.

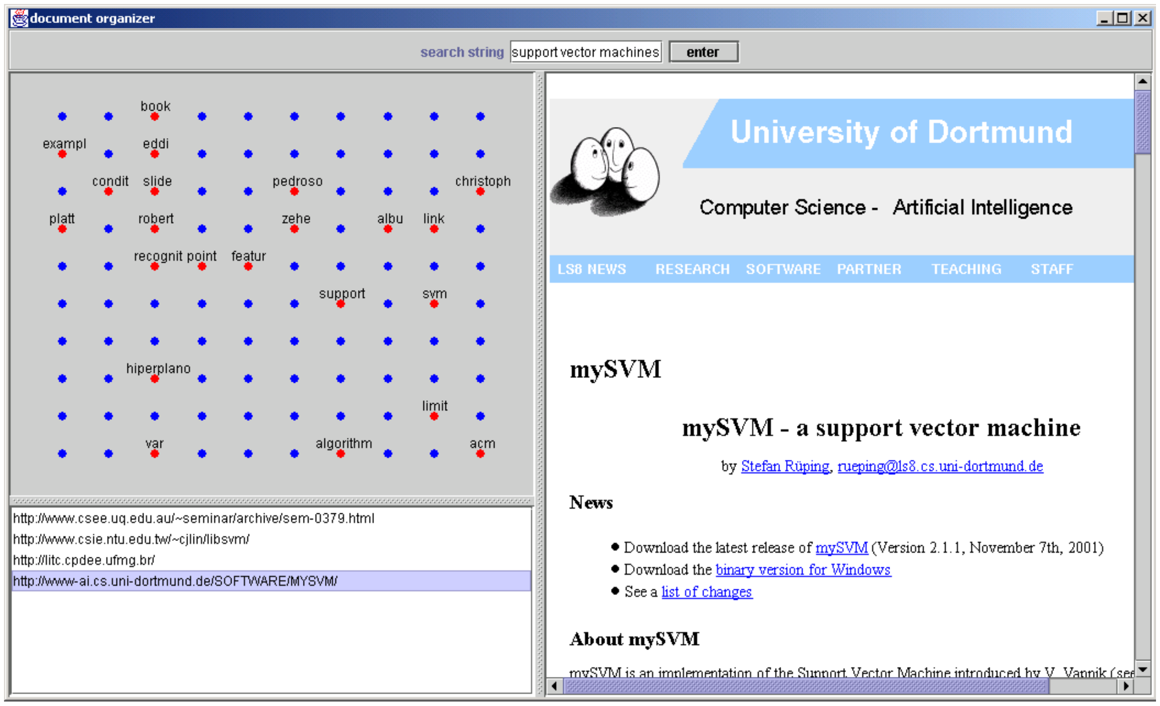
The DisplayAgent handles the display of the trained map and organized documents. The implementation in this project, BasicDisplayAgent, extends the JPanel class so that it can be easily added to any Swing user interface. This is shown in Figure 9 below. It consists of three display areas: the map view, the document list, and the document view. The map shows the organization of documents, with keywords displayed for nodes with documents. The keyword selection algorithm is described in (Lagus and Kaski 1999).



JPanel
<<Interface>>
DisplayAgent
+displayDocuments(documents:HTMLData, map:Map)
BasicDisplayAgent
+displayDocuments(documents:HTMLData, map:Map)

Figure 9. Implementation of the DisplayAgent interface.

So far, only the four agent types and a few implementations have been discussed. These are highly reusable components; each one does a specific job and they can be combined in different ways to get different functionality. Figure 10 below shows a screenshot of an example application that uses the agents to visualize results of a search from the AltaVista search engine on a 2-D grid of points.



http://www.csee.uq.edu.au/~seminar/archive/sem-0379.html
http://www.csie.ntu.edu.tw/~cjlin/libsvm/
http://litc.cpdee.ufmg.br/
http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/

mySVM

mySVM - a support vector machine

by Stefan Rüping, rueping@ls8.cs.uni-dortmund.de

News

- Download the latest release of mySVM (Version 2.1.1, November 7th, 2001)
- Download the binary version for Windows
- See a list of changes

About mySVM

mySVM is an implementation of the Support Vector Machine introduced by V. Vapnik (see

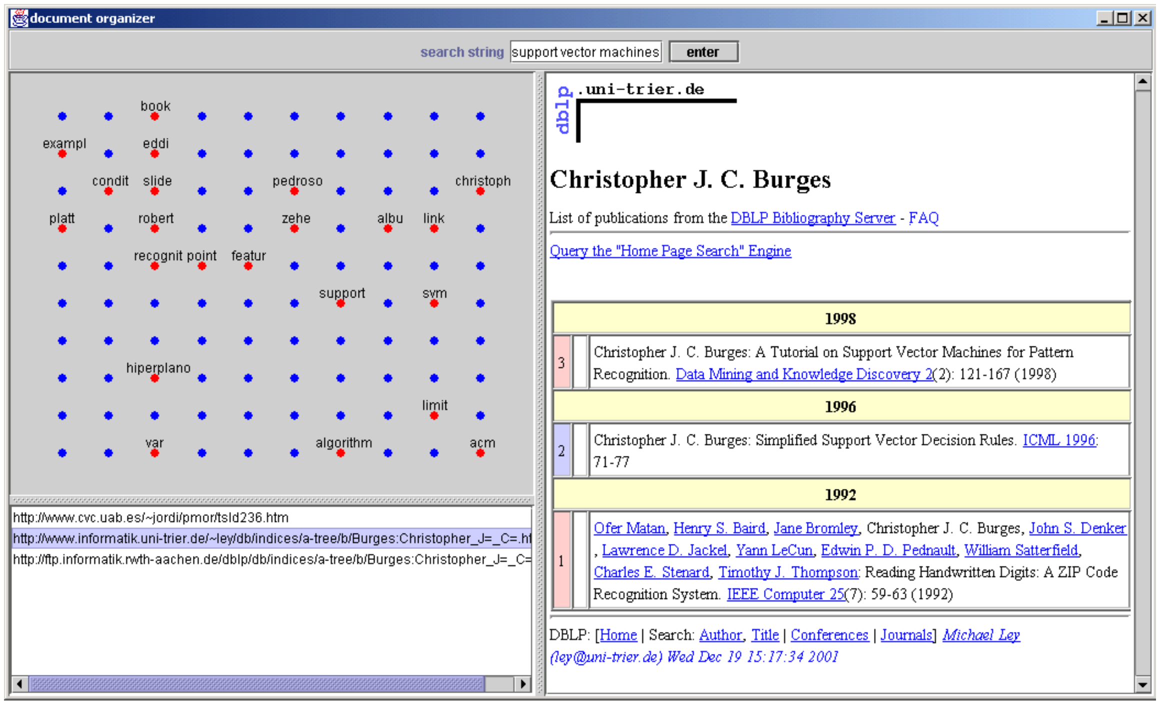
Figure 10. Screenshot of the document organizer application. The top left is the map view, the bottom left is the document list, and the right is the document view.

The document organizer application does very little work on its own. It is a JFrame with a BasicDisplayAgent in the center of its content pane, and simply creates the agents and passes their results between them. For example, the following code shows what happens when the user clicks the “enter” button.

```
System.out.println("collecting documents...");
CollectionAgent collector = new AltaVistaCollectionAgent(
_searchString.getText(), 200);
Collection documents = collector.getDocuments();
System.out.println("got " + documents.size() + " documents");

System.out.println("analyzing documents...");
AnalysisAgent analyzer = new ProjectionAnalysisAgent(new
StemmedStopList(), 5, 315);
HTMLData data = analyzer.analyzeDocuments(documents);

System.out.println("organizing documents...");
OrganizationAgent organizer = new BasicOrganizationAgent();
((BasicOrganizationAgent) organizer).setMapParameters(10, 10,
BasicOrganizationAgent.RECTANGULAR);
Map map = organizer.organizeDocuments(data);

System.out.println("displaying documents...");
_display.displayDocuments(data, map);
```

Results

Documents that contain words that are not present in all the documents in the collection should be mapped onto neighboring nodes. The SOM does seem to do this, as shown in Figure 11 and Figure 12 below. The two web pages shown are lists of publications for Christopher Burges. No other HTML documents in the collection contain his name. Thus, the SOM correctly mapped these two documents to the same node. The third document in that node, http://www.cvc.uab.es/~jordi/pmor/tsld236.htm, was a broken link and should be removed by a preprocessing step; there simply wasn't enough time to code this.



http://www.cvc.uab.es/~jordi/pmor/tsld236.htm
http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/b/Burges:Christopher_J=_C=.ht
http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/b/Burges:Christopher_J=_C=

Christopher J. C. Burges

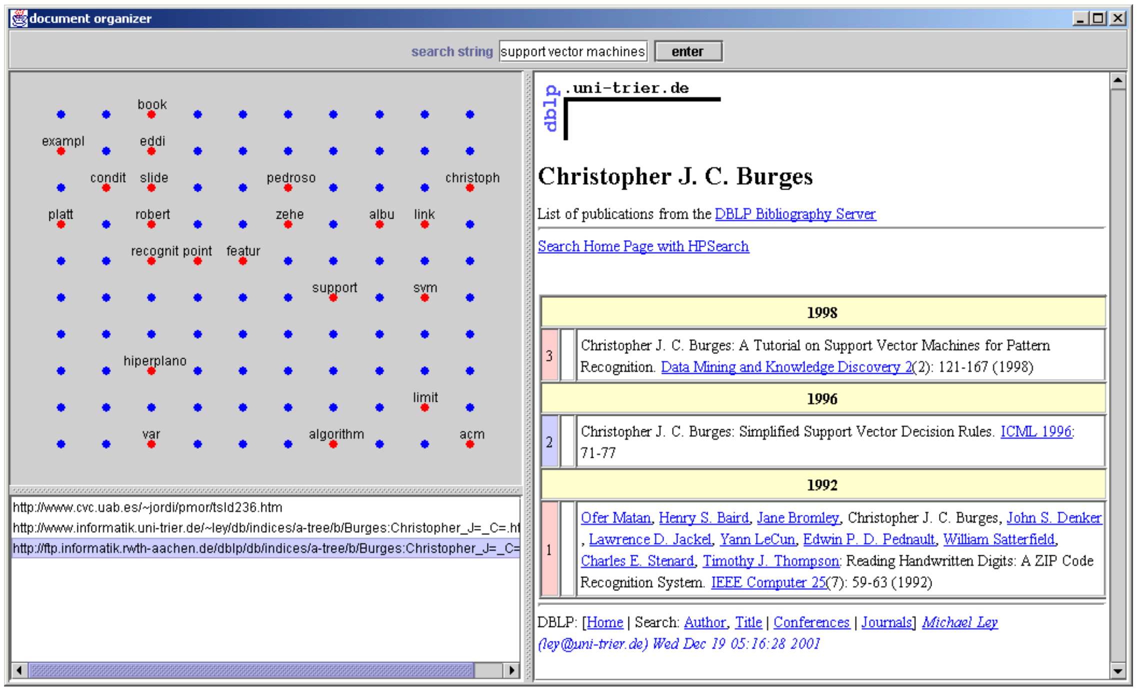
List of publications from the DBLP Bibliography Server - FAQ

Query the "Home Page Search" Engine

		1998
3		Christopher J. C. Burges: A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery 2(2): 121-167 (1998)
		1996
2		Christopher J. C. Burges: Simplified Support Vector Decision Rules. ICML 1996: 71-77
		1992
1		Ofer Matan, Henry S. Baird, Jane Bromley, Christopher J. C. Burges, John S. Denker, Lawrence D. Jackel, Yann LeCun, Edwin P. D. Pednault, William Satterfield, Charles E. Stenard, Timothy J. Thompson: Reading Handwritten Digits: A ZIP Code Recognition System. IEEE Computer 25(7): 59-63 (1992)

DBLP: [Home | Search: Author, Title | Conferences | Journals] *Michael Ley (ley@uni-trier.de) Wed Dec 19 15:17:34 2001*

Figure 11. First document containing Christopher Burges.



http://www.cvc.uab.es/~jordi/pmor/tsld236.htm
http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/b/Burges:Christopher_J=_C=.ht
http://ftp.informatik.rwth-aachen.de/dblp/db/indices/a-tree/b/Burges:Christopher_J=_C=

dblp.uni-trier.de

Christopher J. C. Burges

List of publications from the DBLP Bibliography Server

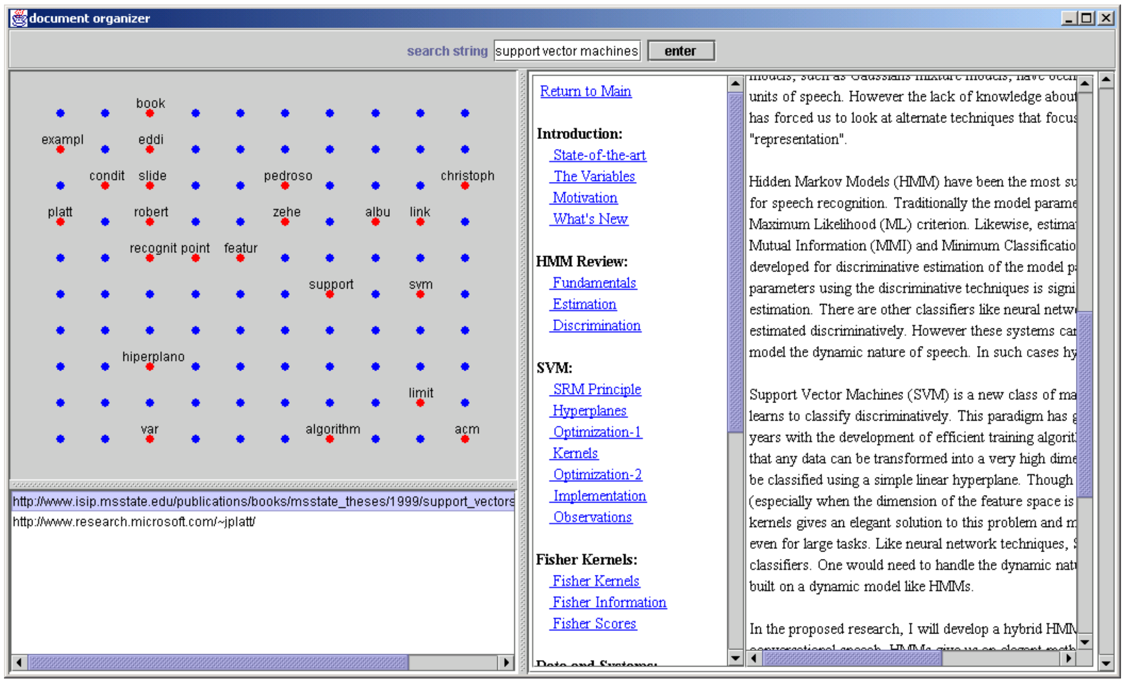
Search Home Page with HPSearch

	1998
3	Christopher J. C. Burges: A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery 2(2): 121-167 (1998)
	1996
2	Christopher J. C. Burges: Simplified Support Vector Decision Rules. ICML 1996: 71-77
	1992
1	Ofer Matan, Henry S. Baird, Jane Bromley, Christopher J. C. Burges, John S. Denker, Lawrence D. Jackel, Yann LeCun, Edwin P. D. Pednault, William Satterfield, Charles E. Stenard, Timothy J. Thompson: Reading Handwritten Digits: A ZIP Code Recognition System. IEEE Computer 25(7): 59-63 (1992)

DBLP: [Home | Search: Author, Title | Conferences | Journals] *Michael Ley (ley@uni-trier.de) Wed Dec 19 05:16:28 2001*

Figure 12. Second document containing Christopher Burges.

Another good example is the documents shown in Figure 13 and Figure 14 below. They are the only documents in the collection discussing things like probabilistic models and maximum likelihood. A strange phenomenon is that the node they're projected onto is labeled "platt" although only one document contains this word (it's the last name of a researcher for Microsoft, who owns the page shown in Figure 14.



http://www.isip.msstate.edu/publications/books/msstate_theses/1999/support_vectors
http://www.research.microsoft.com/~jplatt/

Return to Main

Introduction:
State-of-the-art
The Variables
Motivation
What's New

HMM Review:
Fundamentals
Estimation
Discrimination

SVM:
SRM Principle
Hyperplanes
Optimization-1
Kernels
Optimization-2
Implementation
Observations

Fisher Kernels:
Fisher Kernels
Fisher Information
Fisher Scores

Data and Systems:

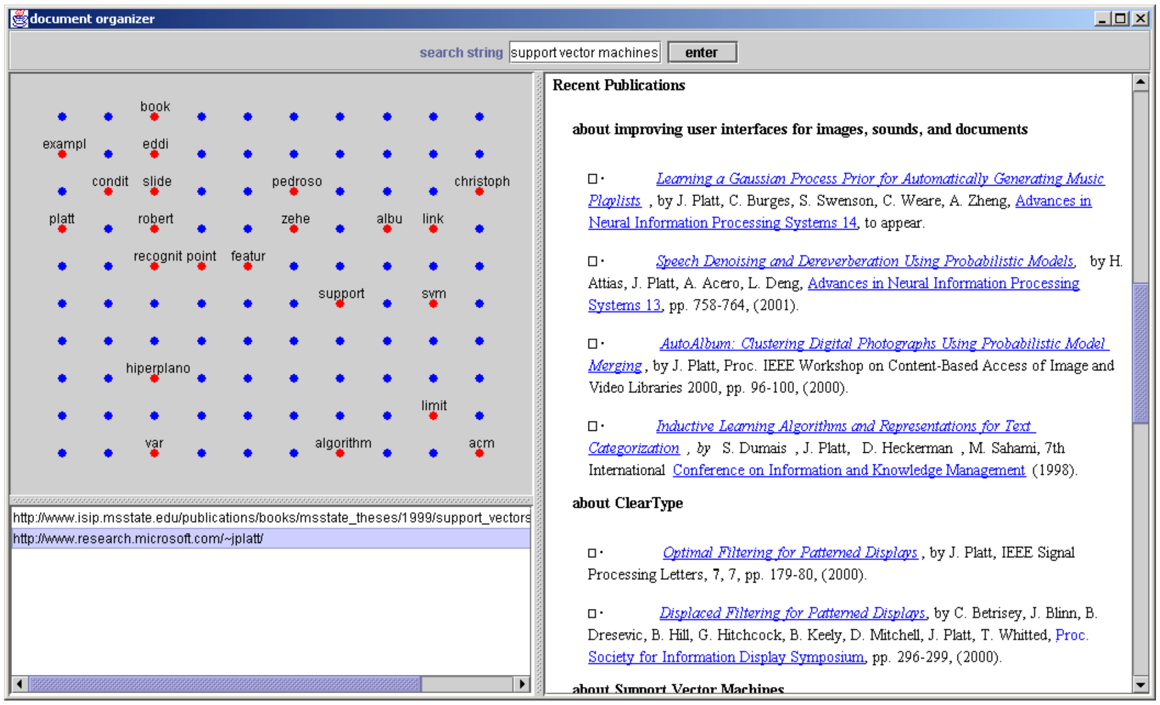
models, such as Gaussians mixture models, have been
units of speech. However the lack of knowledge about
has forced us to look at alternate techniques that focus
"representation".

Hidden Markov Models (HMM) have been the most su
for speech recognition. Traditionally the model parame
Maximum Likelihood (ML) criterion. Likewise, estima
Mutual Information (MMI) and Minimum Classificatio
developed for discriminative estimation of the model p
parameters using the discriminative techniques is signi
estimation. There are other classifiers like neural netw
estimated discriminatively. However these systems ca
model the dynamic nature of speech. In such cases hy

Support Vector Machines (SVM) is a new class of ma
learns to classify discriminatively. This paradigm has g
years with the development of efficient training algorit
that any data can be transformed into a very high dime
be classified using a simple linear hyperplane. Though
(especially when the dimension of the feature space is
kernels gives an elegant solution to this problem and m
even for large tasks. Like neural network techniques, S
classifiers. One would need to handle the dynamic nat
built on a dynamic model like HMMs.

In the proposed research, I will develop a hybrid HMM
conventional speech. HMMs give us an elegant math

Figure 13. First document containing probabilistic models and maximum likelihood.



http://www.isip.msstate.edu/publications/books/msstate_theses/1999/support_vectors
http://www.research.microsoft.com/~jplatt/

Recent Publications

about improving user interfaces for images, sounds, and documents

- *Learning a Gaussian Process Prior for Automatically Generating Music Playlists* , by J. Platt, C. Burges, S. Swenson, C. Weare, A. Zheng, Advances in Neural Information Processing Systems 14, to appear.
- *Speech Denoising and Dereverberation Using Probabilistic Models*, by H. Attias, J. Platt, A. Acero, L. Deng, Advances in Neural Information Processing Systems 13, pp. 758-764, (2001).
- *AutoAlbum: Clustering Digital Photographs Using Probabilistic Model Merging* , by J. Platt, Proc. IEEE Workshop on Content-Based Access of Image and Video Libraries 2000, pp. 96-100, (2000).
- *Inductive Learning Algorithms and Representations for Text Categorization* , *by* S. Dumais , J. Platt, D. Heckerman , M. Sahami, 7th International Conference on Information and Knowledge Management (1998).

about ClearType

- *Optimal Filtering for Patterned Displays* , by J. Platt, IEEE Signal Processing Letters, 7, 7, pp. 179-80, (2000).
- *Displaced Filtering for Patterned Displays*, by C. Betrisey, J. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. Mitchell, J. Platt, T. Whitted, Proc. Society for Information Display Symposium, pp. 296-299, (2000).

about Support Vector Machines

Figure 14. Second document containing probabilistic models and maximum likelihood.

The SOM did seem to project documents with distinguishing words onto the same node in the map. However, it wasn't always clear if documents in neighboring nodes were similar, or if they just happened to be projected onto neighboring nodes. Another useful view in the application would be a list of keywords for a node, like the list of url's, instead of just drawing a single keyword on the map.

Conclusions

This paper described a system of four types of agents, and how they can be used together to organize any set of HTML documents using Self-Organizing Maps. In effect, the OrganizationAgent learns about the documents in the collection and based on the contents, assigns documents to discrete points in a two dimensional space. Documents with similar content are assigned to the same points.

This is a very rough implementation of several recently developed concepts in Information Retrieval. For example, the WEBSOM project (http://websom.hut.fi/websom/) has used SOM to organize massive document collections, on the order of millions of documents, and provides a web-based user interface for exploring the map. The system presented here is much smaller, but also performs the mapping in real time on any decent computer and allows the user to organize their own set of documents.

References

Bloch, J. (2001). Effective Java: Programming Language Guide. Boston, Addison-Wesley.

Chowdhury, G. C. (1999). Introduction to Modern Information Retrieval. London, Library Association Publishing.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Boston, Addison-Wesley.

Hastie, T. and W. Stuetzle (1989). "Principal Curves." Journal of the American Statistical Association **84**(406): 502-516.

Kaski, S. (1998). Dimensionality reduction by random mapping: Fast similiarity computation for clustering. Proceedings of IJCNN'98, International Joint Conference on Neural Networks.

Kohonen, T. (2000). Self-Organizing Maps of Massive Document Collections. International Joint Conference on Neural Networks (IJCNN 2000), Como, Italy.

Kohonen, T. (2001). Self-Organizing Maps. Berlin, Springer.

Lagus, K. and S. Kaski (1999). Keyword selection method for characterizing text document maps. Proceedings of the Ninth International Conference on Artificial Neural Networks (ICANN'99).

Roussinov, D., K. M. Tolle, et al. (1999). Visualizing Internet Search Results with Adaptive Self-Organizing Maps. Proceedings of the ACM SIGIR '99 International Conference on Research and Development in Information Retrieval, Berkeley, CA.