Feature Extraction—A Pattern for Information Retrieval*

Dragoş-Anton Manolescu†
manolesc@cs.uiuc.edu

1 Context

Digital libraries handle *large amounts of information.* They offer access to collections of documents represented in electronic format. According to Bruce Schatz [Sch97]:

> A digital library enables users to interact effectively with information distributed across a network. These network information systems support search and display of items from organized collections.

An increasing number of users discovers online information retrieval and interactive searches. Once comfortable with the new tools, they demand new materials to be available in digital libraries. This requires obtaining digital representations of documents. Since the process is getting cheaper and faster, extending a digital library is not difficult.

Obviously, this increase in the amount of information has a strong impact on the supporting software. Consider for example the case of searching—text retrieval. This is a simple but basic operation for any digital library. Several different algorithms are available for traditional text retrieval [FO95]. However, they are not always applicable in the context of digital libraries. For example, full text scanning, regular expression searching and signature files have bad response times for large amounts of information. Inversion is scalable but has a large storage overhead (up to 300%), and index updates are expensive.

Large volumes of data are not the only challenging characteristic typical to digital libraries. Unlike conventional database systems, digital library users usually perform *similarity searches* (i.e., approximate) instead of exact searches. A typical query for a database user may be "what is the title of the book with the ISBN 0201633612." In contrast, in a digital library system, a user can ask "list in decreasing order of similarity all books that are on the same subject as the one with ISBN 0201633612." (Database systems can also answer such queries provided that an appropriate index structure has been created in advance.) This corresponds to a query by example. Conventional database systems can handle some approximate searches, but they were not designed for this purpose.

The emergence of multimedia content within electronic publications raises another issue. One can provide as a query a digital image and ask for all electronic documents that contain similar pictures. In this case, the challenge is "understanding" the contents of the image. Digital images (and any other multimedia data for that matter) correspond to *complex information.* Although computers are good at representing and manipulating digital representations of this type of information, decoding their contents is still a research

†Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Ave., Urbana, IL 61801.

issue. One workaround for this problem is to have a person annotate each image with a set of keywords. However, due to its inherent limitations, this is only a temporary Band-Aid.

2 Problem

Large amounts of information, similarity search and complex data are typical requirements for information retrieval applications. How does software handle these requirements in a suitable way?

3 Forces

- Systems like digital libraries handle large amounts of information;
- Similarity searching is useful in many domains;
- Multimedia databases handle complex information;
- Information retrieval systems require small space overhead but also low computational overhead for queries and insertions;
- Fast response time is important;
- Designing feature extraction functions and scalable multidimensional indexing methods is hard.

4 Solution

Compute an alternative, simpler representation of data. The representation contains only the information that is relevant for the problem at hand. This computation is actually a function. It maps from the problem space into a feature space. For this reason it is also called "feature extraction function."

A typical feature extraction function for text documents is automatic indexing. The function maps each document into a point in the k-dimensional keyword (or feature) space—k is the number of keywords. Automatic indexing consists of the following steps [FO95]. First, it removes common words like "and," "at," "the." Next, it reduces the remaining words to their stem. For instance, it reduces both "computer" and "computation" to "comput." Then a dictionary of synonyms helps to assign each word-stem to a concept class. (These three steps are also known as preprocessing. Preprocessing extracts concepts from contents.) Finally, the method builds the vector in keyword space. Each vector element gives the coordinate in one of the k dimensions and corresponds to a concept class. Two options for computing the coordinates in the feature space are the following:

- Binary document vectors use only two values to indicate the presence or absence of a term.
- Vectors based on weighting functions use values corresponding to term frequency, "specificity," etc.

Figure 1 illustrates how document indexing maps from document space to 3-dimensional feature space. A multi-dimensional index structure stores the feature space representation.

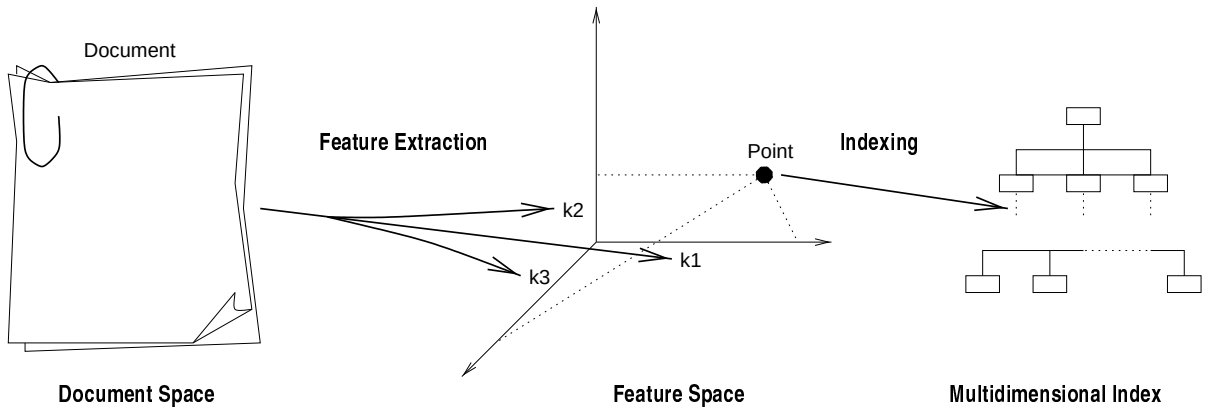

Figure 1: Mapping a document into a 3-dimensional feature space.

Typically, feature extraction maps from a *larger* problem space into a *smaller* feature space. Consider the previous example of document indexing. In document space, one document contains a large number of words. Searching a collection of documents requires many string matching operations. However, in keyword space, documents correspond to multi-dimensional vectors. With this pattern, searching for documents that contain a given set of keywords involves computing some linear expressions—see next paragraph. This is much faster than string matching. Therefore, feature extraction **(i)** enables scalable solutions for problems that deal with large amounts of information.

In the feature space, *similarity searching* corresponds to operations on (suitably normalized) vectors. A popular choice for similarity measure is the cosine function. For example, in the 3-dimensional space from Figure 2, $\vec{b}$ is more similar to $\vec{a}$ than $\vec{c}$, i.e., the point (b_x, b_y, b_z) is closer to (a_x, a_y, a_z) than (c_x, c_y, c_z). Consequently, $\cos\alpha > \cos\beta$, since $\alpha < \beta$.

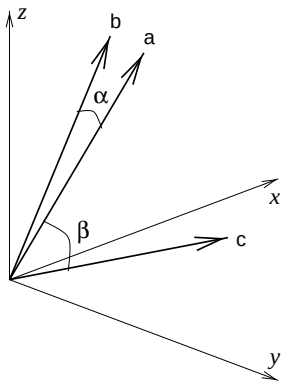

Figure 2: Cosine function and vector similarity.

In this case, the cosine value is easily computed from the inner product—see "Implementation Notes" (Section 5). Consequently, matching a query in the feature space can be represented by linear expressions [Kan94]. Moreover, the answers to a query can be ranked in order of similarity. Queries return only the answers that are above a given threshold. Therefore, feature extraction **(ii)** provides a natural and low-overhead solution for similarity search.

However, using feature extraction for similarity search has its own limitations. Any data items are similar as long as their representation in the feature space are similar. But this is not necessarily how humans perceive things. Two reasons for this are the following:

1. The problem space may be ambiguous. Text is a notorious example. Humans handle this problem by using the surrounding text to establish a context.
2. The feature extraction function is non-injective. Distinct points from problem space can map into the same point in feature space. More on this issue later.

Domain mappings are a widespread technique in mathematics. Usually they map from a *complex* domain into a *simpler* one—here, complexity refers to the operations within the domain. A well-known example is the operational method for the solution of differential equations [BS97]. The method consists in going from a differential equation, by means of an integral transformation, to a transformed equation. The transformed equation is easier to solve than its differential counterpart. Two possible integral transformations are the Laplace transform and the *z* transform.

In a software context, mapping from problem space into feature space also enables computers to manipulate complex information. Digital images are one example. Current image databases employ this pattern to obtain simplified representations for images. Unlike the typical domain mappings from mathematics, these simplified representations lose information. They consider only a *subset* of the image features. Common features for images are color histograms, textures, shapes or a combination of these. Therefore, feature extraction **(iii)** enables software systems to process different types of complex information without "understanding" the contents.

When it maps from a large problem space, feature extraction considers only a few "significant" features in the feature space, discarding the rest. This truncation yields a non-injective mapping. For example, two documents can map into the same point in keyword space. However, this does not mean they are identical. Since the function is not injective, there is no inverse mapping. Several points in problem space can map into a single point in the feature space. This property affects all applications that employ this pattern to provide answers to queries. The solution is to add a post-processing step that filters out the "false alarms." Since the typical number of false alarms is small, the post-processing step usually performs a sequential search to eliminate them.

Besides post-processing, this pattern requires some other additional processing:

1. Feature extraction works with the features of the working set of items (documents, images, etc.). Whenever a new item is added, the system computes its features (i.e., coordinates in the feature space). Therefore, each *insertion* needs this extra step.
2. Another operation that changes is *query processing.* The fundamental idea of feature extraction is to perform all computations in a smaller, simpler space. Processing then takes place in this space. Consequently, answering a query requires its representation in the feature space as well.

To summarize, feature extraction complicates **insertion** and **query** operations since it requires two additional blocks—Figure 3.

Information retrieval (IR) is one of the domains that employs feature extraction extensively. IR has expanded into fields such as office automation, genome databases, fingerprint identification, medical image management, data mining and multimedia [Kan94]. In many of these applications the objective is to minimize response times for different sorts of queries. Performance depends on how fast the system performs searches

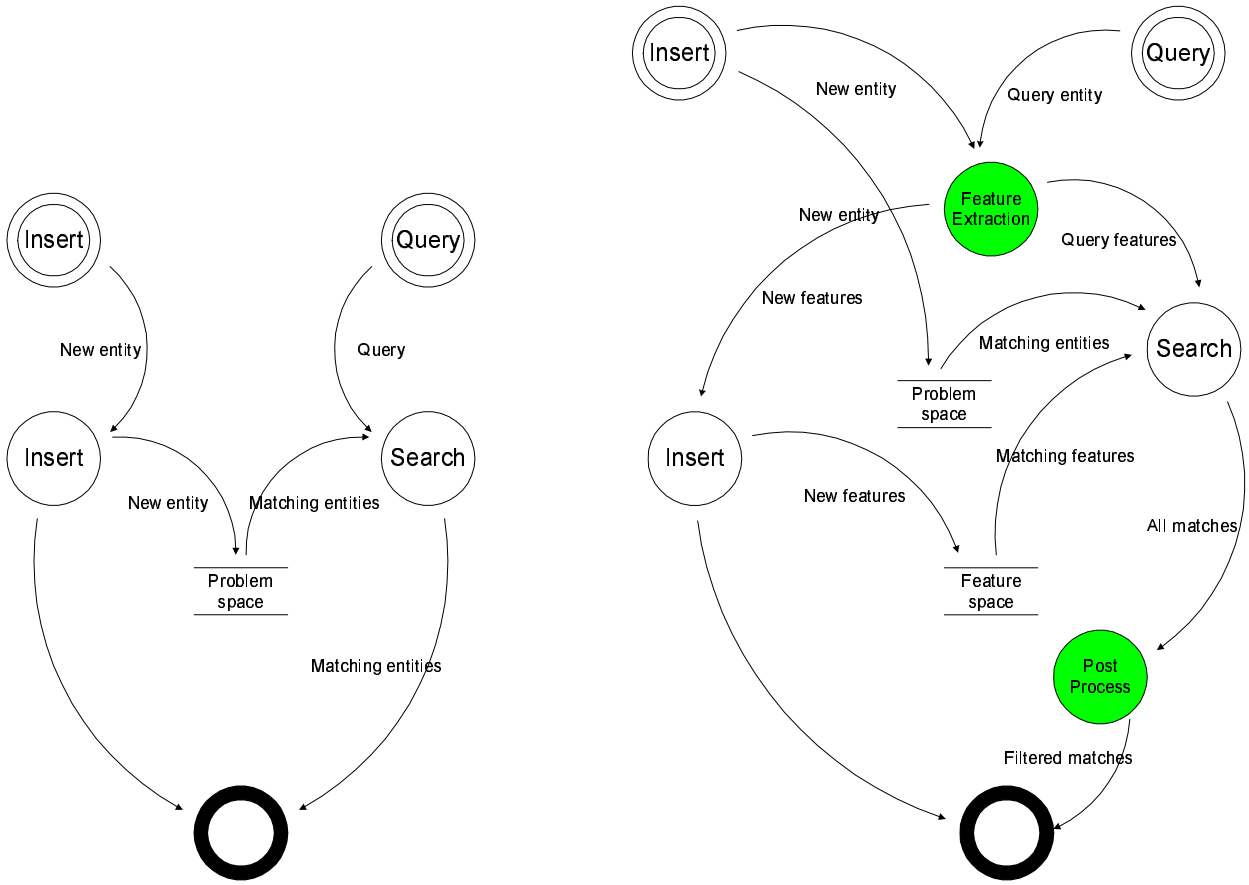

Figure 3: Insertion and query processing without and with feature extraction.

in the multidimensional feature space. Therefore, the choice of a multidimensional indexing method is critical. However, this is not easy. Good unidimensional indexing methods scale exponentially for high dimensionalities, eventually reducing to sequential scanning [AFS93]. Consequently, they apply only when a small number of dimensions is sufficient to differentiate between data items. One solution is to use R-tree variants, which are usable for a larger number of dimensions.

The hard part about this pattern is obtaining a suitable feature extraction function. Obviously, this is domain and problem dependent. One of the important requirements for the domains that employ feature extraction is correctness. A query should return all the qualifying information, without any "misses." The "false alarms" due to the non-injective mapping are not a problem: post-processing (Figure 3) removes them. However, a formal proof is required to demonstrate correctness. Alternatively, a domain-specific algorithm may automatically construct a correct feature-extraction function for a given problem. For example, [FL95] describes such an algorithm for indexing, data-mining and visualization of traditional and multimedia datasets.

The Discrete Fourier Transform (DFT) is an example of a feature extraction function. This function is suitable for pink noise "signals," whose energy spectrum follows $O(f^{-1})$. A wide range of data (e.g., stock prices, musical scores, etc.) fits this description. Consequently, DFT is usable in many different domains. This transform has been successfully used for similarity search [AFS93]. Its properties guarantee the completeness of feature extraction—i.e., correctness. Since DFT is orthonormal (i.e., distance-preserving), the distance between two data items in problem space is the same as the distance between their corresponding

points in the feature space. Therefore, DFT is applicable with any similarity measure that can be expressed as the Euclidean distance between feature vectors in some feature space.

After finding a feature extraction function, the second part of this pattern is deciding which features to consider further. As stated before, not all features are used. For example, systems that use DFT keep only a few low-frequency coefficients. This "lossy" part of the pattern ensures that feature space is smaller than problem space. Deciding on the number of features involves a tradeoff between accuracy and speed. At one extreme, the system is "lossless" and keeps all features. This ensures no false alarms. However, searching a large (feature) space is what the pattern is trying to avoid. At the other extreme, only one feature is used. In this case, the degenerate search in the feature space is fast—it simply returns everything. Post-processing takes a long time though, since it filters all data items. Therefore, the number of features determines the balance between the searching time in the feature space and the post-processing time.

The third part of this pattern is choosing a suitable multidimensional indexing method. The choice depends on the number of features—dimensions of the feature space. Many methods are available for indexing low dimensionality domains—for example, hashtables or B-tree variants. However, as the number of dimensions grows, they degenerate into sequential scanning. R-tree variants (e.g., R*-trees [BKSS90] and SS-trees [WJ96]) offer good performance for a larger number of dimensions.

To summarize, the feature extraction pattern has the following **benefits** (✔) and **liabilities** (✘):

- ✔ It can manage large amounts of data. Compared to sequential scanning, applications using this pattern obtain an increasingly better performance as the volume of data increases [AFS93].
- ✔ Similarity searching corresponds to vector operations in feature space. These have low computational overhead and rank the results.
- ✔ Software can manipulate complex information without having to decode its semantics. This is key for implementing multimedia databases.
- ✔ Users can easily refine queries. Once results are available, they mark only the ones that are relevant. The system adjusts the original query and performs a new search. Provided that the user's feedback is consistent, such queries converge in a few iterations. This mode of operation is also known as "relevance feedback" [SB88]. In the feature space, relevance feedback and consists of adding the selected vectors to the query vector.

- ✘ Efficient search in the feature space requires multidimensional indexing methods. Not all good indexing methods scale well with the number of dimensions. Obtaining an efficient and scalable multidimensional index structure is not easy.
- ✘ Inserting new items and answering queries require additional processing. The architect has to determine the right balance between the number of features and the post-processing time.

5 Implementation Notes

A possible implementation solution is to group all the feature extraction code into a Manager [Som97] object. Three properties of the manager pattern make it an excellent candidate for encapsulating feature extraction. First, the manager has access to all subject instances. Consequently, it is an ideal place to implement post-processing. Second, within a given domain, its functionality is independent from the subject

classes. Thus, developers can change the implementation of the subject class without affecting the manager. Finally, other applications that need feature extraction can reuse the manager's code.

Therefore, the manager object encapsulates feature extraction and indexing. This provides a flexible solution. For example, you can begin without a fancy multidimensional index and concentrate on getting the feature extraction function right. Once you are satisfied with this part, you can experiment with different indexing algorithms, persistence mechanisms, etc. All these changes are transparent for the rest of your application.

The feature extraction manager acts as a Factory [GHJV95] for subject objects. It is responsible for the lifecycle of its subjects. Clients first use the manager to bring new subjects into the system. After "population," other clients employ the manager's services to obtain subject objects in response to queries. When a client does not need the answers for a query any longer, it asks the manager to destroy the corresponding subjects.

As explained in "Solution" (see Section 4), queries that rely on feature extraction return a variable number of answers, ranked in the order of similarity. Therefore, the manager needs an ordered collection to store its subjects. Since it has access to all subjects in this collection, another useful service is iteration in increasing or decreasing order of similarity. The manager can provide this functionality to its clients by implementing the Iterator [GHJV95] pattern.

Figure 4 shows how a client adds a new subject to the information retrieval system:

- The `Client` sends the `insert` message to the `Manager` with the `Subject` object as the argument.
- The `Manager` object responds by sending the `irMapSubject` message to itself. This returns a vector that represents the `subject` object in the feature space.
- Finally, the `Manager` sends the `dbInsert` message. This updates the index (if any) and adds the `Subject` object and its feature vector to the appropriate data stores.

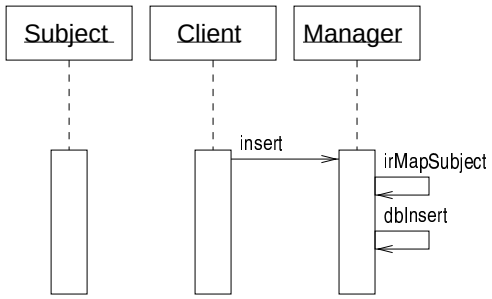

Figure 4: Insert sequence diagram.

Figure 5 provides an example for a query operation:

- A `Client` sends the `search` message to the `Manager` and passes a query as the argument.
- The `Manager` responds by sending the `irMapQuery` message to itself. This maps the query document into the feature space and returns its corresponding vector.

- Next the `Manager` sends the `dbSearch` message. This message involves a similarity function to perform the search in the feature space. For example, the cosine function can be used to determine the similarity between two vectors. This is easily computed from the inner product of the two vectors—see also Figure 2:

$$\cos\alpha = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}|\,|\vec{b}|},$$

  where $|\vec{a}|$ and $|\vec{b}|$ are the vector norms. In Cartesian coordinates this corresponds to a linear expression, since

$$\vec{a}\cdot\vec{b} = a_x b_x + a_y b_y + a_z b_z$$

- Once the answers are available, the `irFilter` message implements post-processing and discards all false alarms. For the sequence diagram illustrated in Figure 5, `irFilter` leaves 3 matches.
- The `Manager` creates the corresponding `Subject` objects.
- The `Client` iterates through these subjects by sending the `next` message to the `Manager`. This message returns one `Subject` object for each invocation.
- The `Client` sends the `dispose` message to the `Manager` whenever it does not need the subjects any longer.
- Finally, the `Manager` retires each `Subject` by sending it the `dispose` message.

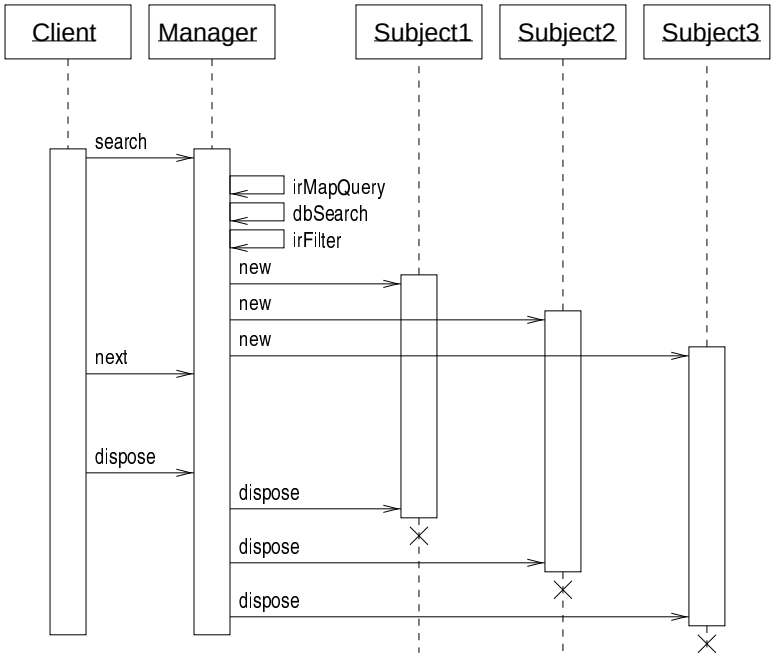

Figure 5: Query sequence diagram.

In Figures 4 and 5, all messages that implement feature extraction are prefixed with `ir`. Similarly, messages that involve database operations are prefixed with `db`. All these operations are localized within the `Manager` object. You can start with a traditional database implementation (only `dbInsert` and `dbSearch`) and add feature extraction later, e.g., when the size of the database starts to cause scalability problems. This addition will require the following changes:

1. Add the `irMapQuery`, `irFilter` (Figure 5) and `irMapSubject` (Figure 4) messages.
2. Modify `dbSearch` (Figure 5) to use the similarity function. For example, in SQL this translates into replacing the `WHERE` clause with a call to a stored procedure that computes the cosine value.

"Related Patterns" (see Section 7) lists other well-known patterns that may be useful when implementing feature extraction.

A Scenario

It is time to show how these pieces fit together with a simple scenario. Since digital libraries or multimedia databases may seem exotic, this scenario considers a more familiar domain.

Professional recruiters assist companies to find matches for their openings. A company looking for new employees provides the recruiter a "wish-list." The list contains some qualifications the company is looking for, but it is not exhaustive. Then the recruiter uses the list to query her database. This query returns all the people within the database that satisfy the search criteria. Since this type of application requires a similarity search, it is a good candidate for feature extraction.

To use feature extraction, the recruiter begins by studying several wish-lists to get an idea about what companies are looking for. She has to decide what features to use. For this example, let's assume that they are knowledge of 3 programming languages (C, Smalltalk and Java) and a modeling language (UML). Therefore, the feature space has four dimensions. Every resumé will correspond to a point in this space.

> In Figures 4 and 5 this corresponds to deciding that:
>
> ➜ `irMapSubject` and `irMapQuery` map into a 4-dimensional space; and
>
> ➜ `dbInsert` and `dbSearch` use a 4-dimensional indexing algorithm.

In this case, extracting features is easy. Each resumé is scanned and when any of these four languages is found, its corresponding coordinate is set to 1. For example, if Joe lists in his resumé Ada, C, Fortran, and Smalltalk, the corresponding vector is $\vec{v}_{\text{Joe}} = (1, 1, 0, 0)$. The information about Ada and Fortran is lost.

> Therefore, `irMapSubject` from Figure 4 performs a full text search for these four keywords. Once it computes the feature vector, `dbInsert` stores it in the database along with its corresponding resumé.

After all resumés are mapped into the feature space, the recruiter is ready to use this system. Let's assume that a company has an opening for someone who knows Smalltalk (3+ years experience), Java and UML. In the feature space, this query corresponds to the vector $\vec{v}_{\text{query}} = (0, 1, 1, 1)$. Note that the experience requirement does not have a corresponding feature and therefore is lost.

The `search` message from Figure 5 has the supplied wish list as a parameter. Next, `irMapQuery` maps this query into the feature space and obtains $\vec{v}_{\text{query}}$.

A search computes the similarity between the query and all the other vectors in the database. Only the ones that are above a given threshold are returned. Assume that they are (in this order) Adam with $(1,1,1,1)$, Bob with $(0,1,1,1)$, Clark with $(0,1,1,1)$ and Donna with $(0,1,1,0)$. Adam, Bob and Clark are exact matches for the query vector and would have been found by a traditional database system as well. However, while Donna's vector does not have the UML component, the similarity function is above the threshold and therefore she is a match. In a traditional database system, this sort of matching requires a complex boolean expression. Feature extraction provides a more elegant solution. Here, similarity matching is essential, since the "wish-list" is not carved in stone. For example, if Donna is proficient with OMT and her other credentials are better than the others, she will probably get an offer.

In Figure 5, this corresponds to the `dbSearch` message. `dbSearch` computes the similarity between the query vector and all the other vectors stored in the database.

Post-processing compares the original query with the resumés of each potential candidate and returns only the ones that meet the experience requirement as well.

The `irFilter` message from Figure 5 implements post-processing. In this case, it does a full text search in the 4 resumés returned by `dbSearch` and discards Bob's resume since it does not satisfy the experience requirement. Once `irFilter` completes, the manager creates a `Subject` object for each match. Clients send the `next` message to iterate through these objects.

Therefore, instead of comparing *all* the resumés with the original query, feature extraction maps this problem into a four-dimensional space where the solution is simpler. This returns all qualifying answers plus a few false alarms. Post-processing performs a full text scan *only* on these answers and discards the false alarms.

Sample Code

The following code fragments illustrate how an image retrieval system employs feature extraction. This code has been used in a Smalltalk implementation of the MARS[1] system [ORC+97].

The `addImage` method adds new images in the system. It corresponds to the situation illustrated in Figure 4. First the `addImage:` method computes all features of the image in the `newImageFeatures` variable. Next, it updates the index `imageRepresentationSet` with the image identifier, its file name and the computed features.

```
ImageDatabase>>addImage: anImgFilename
| newImageId newImageFeatures newImageRepresentation |

newImageFeatures := ImageFeatures extractFeaturesFromImageFile: anImgFilename
                                  withTextureNormalizer: textureNormalizer
                                  withDfTable: dfTable.
newImageRepresentation := ImageRepresentation representImageWithId: newImageId
                                              withFilename: anImgFilename
                                              withFeatures: newImageFeatures.
imageRepresentationSet addLast: newImageRepresentation.
```

[1] A working demo of the MARS system is available on the Web at `http://jadzia.ifp.uiuc.edu:8000/`.

In this application, feature extraction consists of color histogram, color layout and texture information. The code fragment that does feature extraction follows. Each type of feature is extracted in one of the variables `colorHistogram`, `colorLayout` and `texture`.

```
ImageFeatures>>extractFeaturesFromImageFile: aString
               withTextureNormalizer: aTextureNormalizer withDfTable: dfTable
| image |

image := ImageReadWriter createImageFromFileNamed: aString.
colorHistogram := ColorHistogram extractFromImage: image histogram: 8 by: 4.
colorLayout := ColorLayout extractFromImage: image grid: 5 by: 5 histogram: 8 by:4.
texture := image extractTexture: aTextureNormalizer.
```

Extraction of the actual features is delegated to the `Image` class. The following code shows the implementation for the color histograms. This type of processing is domain-dependent.

```
Image>>colorHistogram: aNumber1 by: aNumber2
"returns aNumber1 by aNumber2 color histogram flattened as an array,
saturation being more significant and hue being less significant"
| length area colorhist quantizedhue quantizedsat histindex  |
length := aNumber1 * aNumber2.
area := width * height.
colorhist := Array new: length withAll: 0.
self pixelsDo: [:x :y |
    quantizedhue := self quantizedHueAtPoint: x @ y levels: aNumber1.
    quantizedsat := self quantizedSaturationAtPoint: x @ y levels: aNumber2.
    histindex := (quantizedsat - 1) * aNumber1 + quantizedhue.
    colorhist increment: histindex.].
^(colorhist collect: [:each | (each / area) asFloat])
```

Finding answers for a query involves mapping the query in feature space, finding all the matches and filtering out the false alarms—see Figure 5. The following code fragment performs the search in the feature space using a distance function specific to image processing. `currBatch` is an ordered collection that holds the matches.

```
ImageDatabase>>searchForFeatures: anImageFeatures withWeights: aWeightArray
| index currBatch |
anImageFeatures start.
searchResult initialize.
currBatch := OrderedCollection new.
imageRepresentationSet do: [:each |
     searchResult addSearchObject:
                  (each distance: anImageFeatures withWeights: aWeightArray)].
index := 1.
[index <= ((searchResult size) min: 8)]
     whileTrue:
            [currBatch add: (searchResult at: index).
             index := index + 1.].
^currBatch
```

6 Examples

Feature extraction is not new. One of the pioneers of this pattern was Gerald Salton. He employed it in the SMART system [Sal69] at Cornell, a long time before the term "digital library" was coined.

Since all of the information currently produced is available in electronic format, many application domains use feature extraction. These include telecommunications, multimedia, medicine, business, etc. However, despite its widespread use, few studies document feature extraction per se. Several documented examples from different domains follow.

1. The authors of [KJF97] use feature extraction to perform ad-hoc queries on large datasets of time sequences. The data consists of customer calling patterns from AT&T and is in the order of hundreds of gigabytes. Calling patterns are stored in a matrix where each element has a numeric value. The rows correspond to customers (in the order of hundreds of thousands) and the columns correspond to days (in the order of hundreds).

   In this case, the problem is the compression of a matrix which consists of time sequences, while maintaining "random access." Generic compression algorithms (e.g., Lempel-Ziv-Huffman, etc.) achieve good compression ratios. However, queries do not work on compressed data and require decompression. This is not viable for the amounts of data corresponding to calling patterns.

   Feature extraction avoids the need for decompression. The function for feature extraction is singular value decomposition (SVD). This truncates the original matrix by keeping only the principal components of each row and achieves a 40:1 compression ratio. Therefore, SVD maps the large customer calling pattern matrix into a smaller matrix in the feature space. The compressed format is lossy but supports queries on specific cells of the data matrix, as well as aggregate queries. For example, a query on a specific cell is "What was the amount of sales to ACME, Inc. on August 16th, 1997?" The method yields an average of less than 5% error in any data value.

2. Large amounts of data are also typical in the financial domain. Feature extraction provides a fast way for searching stock prices [FRM94] and is useful for any other time-series databases (e.g., weather, geological, environmental, astrophysics or DNA data).

   The problem here is finding a fast method for locating subsequences in time-series databases. The system needs to answer queries like "Find companies that have similar sales patterns with ACME, Inc." Sequential scanning is not viable for several reasons. First, it does not scale for large amounts of data. Second, it has a large space overhead since each search requires the availability of the entire time sequence.

   Feature extraction provides a fast and dynamic solution. In this case, the feature extraction function is a *n*-point Discrete Fourier Transform (DFT). This maps each time-series into a trace in a multi-dimensional feature space. Since the method considers only a few low-frequency coefficients, queries return a superset of the actual results. However, post-processing eliminates all "false alarms." The space overhead is small, and the response times are orders of magnitude faster than a sequential scan.

3. Besides handling large amounts of data, feature extraction is also applicable for software systems that manipulate complex information. Digital images are a typical example. Computers are good at manipulating the basic image components like luminance and chrominance. However, decoding the semantics of the information contained within an image (its contents) is still a research issue.

   Petrakis and Faloutsos use this pattern for similarity searching in image databases [PF97]. The problem is to support queries by image content for a database of medical images. A typical query is "Find

all X-rays that are similar to Bob's X-ray." This problem has the following requirements. First, it needs to be accurate. The results of a query must return all qualifying images. Second, query formulation must be flexible and convenient. The user should be able to specify queries by example, through a GUI. Finally, response times and scalability are important. Performance must remain consistently better than sequential scanning as the size of the database grows.

The system represents image content by attributed relational graphs holding features of objects and relationships between them. This representation relies on the (realistic) assumption that a fixed number of objects are common in many images—e.g., liver, lungs, heart, etc. All these common objects are "labeled." The method considers five features for each labeled object in the image. Five features are sufficient for medical purposes. However, the method can handle any other additional features that the domain expert may want to consider. This approach outperforms sequential scanning and scales well with the size of the database.

4. Trademark images are important elements of a company's industrial property. They identify the producer of a product or service. To gain legal protection, trademark symbols must be formally registered. The patent office has to ensure that all new trademarks are sufficiently distinctive to avoid confusion with existing marks.

   Currently, manual assignment of classification codes is the main method of organizing trademark image collections. The method typically employs the Vienna classification system, developed by the World Intellectual Property Organization. The top level of this hierarchy has 28 distinct categories.

   Trademark image retrieval has several unique characteristics. First, trademark examiners search for images by primitive features, e.g., shape. Second, trademark registries hold large collections of images in electronic format. And finally, in the trademark field, successful retrieval criteria are well-defined. These characteristics make trademark image retrieval an ideal candidate for content-base image retrieval techniques.

   The Artisan project (automatic retrieval of trademark images by shape analysis) [EBG98] is intended to replace the Trademark Image System (Trims), currently in use at the UK Patent Office. This system needs to answer only one type of query: given a candidate trademark, is it sufficiently similar to any existing mark to cause confusion? After studying how trademark examiners work, the researchers concluded that shape is the most important characteristic. Other attributes can be neglected. For example, color information is discarded, since the images are deliberately registered in back-and-white. Consequently, Artisan works only with shape features. These features are organized in two vectors. The *boundary shape vector* consists of 4 features: aspect ratio, circularity, transparency and relative area. Likewise, the *family characteristics vector* consists of 5 features: right-angleness, sharpness, complexity, directedness and straightness. However, the authors are still experimenting with alternatives.

7 Related Patterns

- The manager can return Proxy [GHJV95] objects for subjects. This may be useful in circumstances such as when subjects have large memory footprints or are available on remote databases.

- The pattern is independent of the feature extraction function. Domain experts select any function that is suitable for some problem, ensuring that it produces correct results. Strategy [GHJV95] objects can represent various feature extraction functions. This is useful for domains that consider multiple

feature sets. One example is image databases, for which popular feature choices are patterns, colors and textures.

- Digital library systems are likely to use feature extraction for compound documents which contain text and multimedia information. In this case, components will probably use the Extension Object [Gam97] pattern to provide interfaces for information-retrieval operations.

Acknowledgments

Ralph Johnson originally encouraged me to document feature extraction as a pattern, and directed me toward the pioneering work of Gerald Salton. The members of the software architecture group (John Brant, Ian Chai, Brian Foote, Ralph Johnson, Don Roberts and Joe Yoder) have provided substantial feedback on an earlier draft. My shepherd, Kyle Brown, made additional suggestions. The Smalltalk implementation of MARS was written by Michael Ortega-Binderberger, who also provided feedback and suggestions for improvement. I am grateful to all of them.

References

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Proc. FODO conference*, 1993.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[BS97] I. N .Bronshtein and K. A. Semendyayev. *Handbook of Mathematics*. Springer-Verlag, third edition, 1997.

[EBG98] John P. Eakins, Jago M. Boardman, and Margaret E. Graham. Similarity retrieval of trademark images. *IEEE Multimedia*, 5(2):53–63, April–June 1998.

[FL95] Christos Faloutsos and King-Ip (David) Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. ACM SIGMOD*, pages 163–174, May 1995. Also available as technical report (CS-TR-3383, UMIACS-TR-94-132; Institute for Systems Research: TR 94-80).

[FO95] Christos Faloutsos and Douglas W. Oard. A survey of information retrieval and filtering methods. Technical Report 3514, Department of Computer Science, University of Maryland, College Park, MD 20742, August 1995.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. SIGMOD Conference*, pages 419–429, 1994.

[Gam97] Erich Gamma. *Extension Object*, chapter 6. In Martin et al. [MRB97], October 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Kan94] Paul B. Kantor. Information retrieval techniques. In Martha E. Williams, editor, *Annual Review of Information Science and Technology*, volume 29. American Society for Information Sciences, 1994.

[KJF97] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *Proc. SIGMOD Conference*, 1997.

[MRB97] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3*. Software Patterns Series. Addison-Wesley, October 1997.

[ORC+97] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Sharad Mehrotra, and Thomas S. Huang. Supporting similarity queries in MARS. In *Proc. 5th ACM international multimedia conference*, November 1997.

[PF97] Euripides G. M. Petrakis and Christos Faloutsos. Similarity searching in medical image databases. *IEEE Trans. on Knowledge and Data Engineering*, 9(3):435–447, May/June 1997.

[Sal69] Gerard Salton. Interactive information retrieval. Technical Report TR69-40, Cornell University, Computer Science Department, August 1969.

[SB88] Gerard Salton and Chris Buckley. Improving retrieval performance by relevance feedback. Technical Report TR88-898, Cornell University, Computer Science Department, February 1988.

[Sch97] Bruce R. Schatz. Information retrieval in digital libraries: Bringing search to the net. *Science*, 275:327–334, January 1997.

[Som97] Peter Sommerland. *Manager*, chapter 2. In Martin et al. [MRB97], October 1997.

[WJ96] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523. IEEE Computer Society, February 1996.