Data Mining: Forensic Analysis

Monte F. Hancock
Published online on: 29 Dec 2015

How to cite: Monte F. Hancock. 29 Dec 2015, Data Mining: Forensic Analysis from: Encyclopedia of Information Systems and Technology CRC Press
Accessed on: 18 Aug 2023

PLEASE SCROLL DOWN FOR DOCUMENT

Full terms and conditions of use: https://www.routledgehandbooks.com/legal-notices/terms

This Document PDF may be used for research, teaching and private study purposes. Any substantial or systematic reproductions, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The publisher shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.
Data Mining: Forensic Analysis

Monte F. Hancock, Jr.
Chief Scientist, Celestech, Inc., Melbourne, Florida, U.S.A.

Abstract
The practical purpose of this entry is to characterize the information to be collected and organized during the Data Evaluation Step of a data mining spiral. Rather than developing these ideas in prose from which the reader must extract actionable chunks, the material is presented as a topically organized checklist of questions to be addressed. This entry also describes some techniques for conducting the preliminary analysis of domain data; that is, analysis which is performed before the Feature Extraction Step. The aspect that distinguishes this genre from others is that here we are searching data for instances of unknown patterns; patterns that are not well characterized, and for which representative examples are not available.

GOALS
After you have read this entry, you will understand supervised and unsupervised data mining methods as they relate to each other. You will understand how data clustering is an unsupervised data mining process. You will have seen examples of the formulation of objective functions, and a hierarchical architecture for unsupervised processing. You will be familiar with the fundamentals of multilayer perceptron neural networks (NN). You will have seen how text mining can be used to build sensitive and specific search applications.

Examples

- Motion Pattern Analysis
  - Vehicle traffic (Kirchhoff applies, because matter is conserved.)
  - Network traffic (Kirchhoff does not apply, because information is not conserved.)
- Data Clustering
- Document Analysis
  - Steganography
- Clique detection
  - Network analysis
  - Collaborative filtering
- Fraud and Abuse
  - Money laundering
  - Network intrusion detection
- Time Series Analysis
  - Forecasting
  - Market prediction
  - Customer acquisition/attrition prediction
- Finite State Modeling
  - Processes with random elements (hidden Markov model)
- Behavior modeling
- Process Optimization, Scheduling, and Planning
  - Constrained optimization
  - Resource management
- Change Detection/Anomaly Processing

INTRODUCTION

The approach taken in this entry is more descriptive than pedagogical. While supervised learning can be treated as a special case of the regression problem, unsupervised learning covers an application space much too broad to be covered by any single theory. However, we will establish a metaphor for unsupervised architectures that casts them as realizations of supervised learning, where training examples have been replaced by working hypotheses captured in an objective function.

With this metaphor in view, we will describe the fundamentals of unsupervised learning though carefully chosen case studies and examples that highlight the various design and development considerations for this genre.

GENRE OVERVIEW

This genre consists of problems whose solutions are instances of unsupervised learning. Techniques such as supervised learning rely on having labeled examples of target patterns. For unsupervised learning, the data input is a set of unlabeled feature vectors (no ground truth).

The goal of unsupervised learning is fundamentally the same as the goal of supervised learning: the
detection, characterization, and exploitation of actionable patterns in data.

Is it even possible to construct an application to search for unknown patterns? The answer to this depends upon what unknown means in the problem domain. Supervised learning knows target patterns in the sense that it has examples that can be used for modeling. These are known unknowns: we are certain they really exist, and we can more or less precisely characterize them.

But supervised learning is not designed to address the problem of unknown unknowns: phenomena of uncertain existence and nature, not heretofore observed, characterized, or necessarily even hypothesized.

Fortunately, saying that we don’t have specific examples or a precise characterization of a target pattern does not mean that we know nothing upon which to base a search. If that were the case, we wouldn’t recognize a target pattern if we found one.

It is possible in many search applications to form reasonable hypotheses about what some target patterns might look like, even though specific examples are not available. In this sense, supervised learning may be likened to a police officer who familiarizes himself or herself with photographs in a blotter. Unsupervised learning is like the partner, who relies on hypothetical descriptions created by a profiler. Discovery in the first case is done by matching suspects with a set of known examples, and in the second by matching suspects with a set of hypothesized properties.

**RECOMMENDED DATA MINING ARCHITECTURES FOR UNSUPERVISED LEARNING**

Unsupervised learning architectures have three functional components:

1. Unlabeled feature vectors (training set)
2. An objective function (measures the performance or quality of a model)
3. A learning algorithm that uses vigilance parameters and the output of the objective function to modify the model (incrementally updates the model)

A training set for unsupervised learning is just like a training set for supervised learning, except that the vectors do not have ground truth labels.

An objective function for a system accepts as input the state of the system, and assigns to it a measure of quality. It is a scoring mechanism. Suppose, for example, that the system is a chess game. A simple objective function might count the number of pieces a player has taken from his opponent. A more sophisticated objective function could incorporate measures derived from the configuration of pieces on the board, look-ahead to possible future states, etc.

Learning algorithms range from the mindlessly simple (e.g., Monte Carlo methods), to the tremendously complex (e.g., simulated annealing). The purpose of a learning algorithm is to make changes to the model being developed so that its performance (i.e., quality as determined by application of the objective function) attains a level that makes it a usable model.

To determine the type and magnitude of changes to make during learning, training algorithms must have notions of significance that make sense in the problem domain. This information is supplied by one or more vigilance parameters. A vigilance parameter is a threshold of some sort that is used to determine when things are sufficiently similar/different, good/bad, right/wrong, and so on, for a decision to be made. For example, if a police officer is looking for a tall suspect, does someone who is 5’10” qualify? If he is looking for a tall building, do six stories suffice?

Fig. 1 reiterates the general learning architecture for supervised learning on the left, and shows the general learning architecture for unsupervised learning on the right. It is the intentional parallelism seen here that allows us to understand unsupervised learning by noting its similarities and differences with supervised learning.

The objective function; and the inferred symbols are numbers, names, vectors, URLs, documents, complex structures, pieces of code—instances of whatever it is the unsupervised learning process is attempting to associate with the training vectors.

**EXAMPLES AND CASE STUDIES FOR UNSUPERVISED LEARNING**

We now apply the metaphor of Fig. 1 to implement an unsupervised learning application using what is usually regarded as a supervised learning paradigm (a multilayer perceptron).

For this example, a set of unlabeled feature vectors is provided. It is desired to aggregate these into clusters of vectors that are similar to each other to facilitate categorization for subsequent processing. We formulate the
following unsupervised learning process by selecting the three fundamental components of unsupervised learning:

1. **Unlabeled feature vectors (training set)**
   We already have a data set. The standard data preparation methods are applied as needed (e.g., segmentation divides the data into disjoint calibration, training, and validation sets; gaps are repaired; data are normalized, etc.)

2. **An objective function (measures the performance or quality of a model)**
   Because we don’t know anything about the problem space from which these data come, there aren’t any *a priori* formulae or heuristics to use as a basis for clustering. Therefore, we fall back on the metric properties of Euclidean space, and formulate our own definition for what good clustering might mean in terms of distances in feature space.

Here are some distance-based properties of good clustering of numeric data:

- **Property 1**—Clusters that are more tightly packed are better than clusters that are loose.
- **Property 2**—Vectors are less likely to be assigned to the wrong cluster when the clusters are far apart.
- **Property 3**—Each vector should be close to its own cluster, and far from the others.

These must now be turned into an objective function that can be efficiently evaluated. Rather than perform a large number of point-to-point distance computations, we compute the centroid (center of gravity = average) for each cluster by averaging its member vectors coordinate-wise. The centroid of cluster $J$ will be the vector:

$$
\mu_J = \frac{1}{L_J} \sum_{k=1}^{L_J} (v_{jk1}, v_{jk2}, ..., v_{jKN})
$$

Here $\mu_J$ is the centroid of cluster $J$, it is a vector having $N$ components. $L_J$ is the number of vectors in cluster $J$; $N$ is the number of features in a feature vector; and $v_{jk}$ is the $i$th feature of vector $k$ of cluster $J$.

Define a metric for each property:

- **Metric 1**—$A = $ Intra-cluster distance: the sum of the distances from each vector in a cluster to its centroid. Smaller is better.
- **Metric 2**—$E = $ Inter-cluster distance: the sum of the distances from each vector to the centroids of clusters of which it is not a member. Larger is better.
Metric 3—\( C = \text{Centroid distance}: \) the sum of the distances between all the centroids. Larger is better.

These can be rolled up into a single number whose value increases when any of these metrics gets better. When better means larger, the metric goes into the numerator. When better means smaller, the metric goes into the denominator:

\[
D = \frac{CE}{A + 0.1}
\]

The 0.1 is an arbitrarily chosen small number that is added in the denominator to avoid division by zero, which happens if all the vectors are at the same location (usually due to some data conditioning error).

This is our designer objective function. It is an abstract measure of clustering quality, so particular values don’t mean much. But larger values mean better quality, and that is all that matters here. The various component metrics can be seen later on (Fig. 2).

3. A learning algorithm that uses vigilance parameters and the output of the objective function to modify the model (incrementally updates the model). We decide to use my favorite learning method: LUCK.

Monte Carlo methods conduct repeated trials of random instantiations of the model being developed. This is an undirected search of the model’s parameter space for settings that optimize the objective function. The process is simple:

a. Initialize the model by assigning valid values to its parameters.
b. Ingest the best model parameters seen so far.
c. Jiggle the best model parameters by applying a random adjustment to them (e.g., if they are real numbers, add or subtract a tiny value from each one).
d. Apply the objective function to the jiggled model.
e. Run the entire training set using the jiggled model, and evaluate the objective function on the resulting clustering. If it is better than the best clustering, the jiggled model becomes the new best model. If not, it is discarded.

Steps 1–4 are repeated, producing occasional incremental improvements in model quality. Each repetition is one training epoch. If the objective function is well-chosen and efficiently computable, it can be repeated.

Fig. 2 Component metrics for clustering quality.

Fig. 3 Unsupervised clustering applied to Fisher iris data.
many times at a reasonable computational cost. The incremental improvements on an epoch are usually small, but a million small improvements can produce a high quality model.

This unsupervised clustering method was applied to the Fisher iris data, giving the result depicted graphically in Fig. 3. The vigilance parameter required by this application is the number of clusters to create. The value used for this run was 5.

The model form used was a three-layer neural network; it ingested the feature vectors and assigned them to clusters. During a Monte Carlo run, it was the weights of this neural network that were randomly jigged to vary the model. As these weights varied, the neural network would give higher and lower values for the objective function value, \( D \). Over many epochs, this allowed the neural network to find weight settings that maximized \( D \), resulting in a good clustering.

Ground truth is known for this problem, so it is possible to check the results of the unsupervised clustering (which did NOT use the ground truth). This was done by assigning all vectors in an inferred cluster, the actual ground truth of most of its members. This is the so-called the majority rule, and is often used in applications. This gives an accuracy score of 85.3% for this run.

Notice that 1624 epochs were run. During an epoch, all 150 patterns in the Fisher set are processed. Even so, the total execution time, including disc input/output (I/O), was 2.58 seconds. With an implementation this efficient, it is possible to run millions of training epochs per hour, providing fairly good coverage of the parameter space.

It is important to note that there is nothing special about using a neural network here. Any paradigm that has sufficient representational power could be used (e.g., a polynomial of sufficiently high-degree). Principled paradigm selection is a deep subject; however, in applications the choice is usually made empirically.

For comparison, Fig. 4 shows the output of a deterministic clustering routine, that is, it has no random element, and always gives the same clustering. It used the same vigilance parameter setting (five clusters). Its result does not align as well with the actual ground truth, and there is nothing that can be done about this. Since it is deterministic (rather than trained), its result cannot be improved by additional processing.

Case Study: Reducing Cost by Optimizing a System Configuration

This case study describes a project to determine the optimal placement of hundreds of vehicle tracking signposts around a city to facilitate tracking and timing of municipal buses. A signpost is a small, stationary radio installed near the roadway that periodically transmits its “ID number.” This transmission is picked up by passing buses and sent back to a tracking center, which knows the location and ID number of all the signposts. In this way, a precise position and time for buses is obtained inexpensively and can be monitored by the tracking station. Optimizing this system is an unsupervised learning problem because it is based not upon labeled examples, but upon domain heuristics that characterize a good model in terms of its properties. Sampling is used to create candidate models that are then evaluated using an
objective function. Many Monte Carlo epochs incrementally produce a good model.

Hundreds of signposts would be required for adequate bus location and schedule tracking, and to support emergency call-in. Each signpost required a site-survey, installation, and servicing, at a cost of thousands of dollars each. Any significant reduction in the number of signposts that did not compromise system requirements would result in substantial savings. A placement strategy had been prepared manually by a group of municipal transportation experts; this configuration was used as the starting point for the data mining modeling effort.

The work for this effort was carried out almost entirely by a custom-built unsupervised learning application. It used a Monte Carlo simulation very similar to that described above. During each epoch, it simulated a full 24-hour day of buses running the city’s 65,000 miles of bus routes. Each epoch used different locations for the signposts.

An ad hoc objective function was designed that totaled the vehicle speed and location error of the system moment-by-moment. These metrics were chosen because they were the basis for testing goals called for in the system requirement specification.

When the time allotted to simulation epochs was completed (nearly 200 hours of continuous machine time), the best placement scenario for the signposts was output. It reduced the number of signposts required in the model created by the human experts by approximately 75%, saving hundreds of thousands of dollars on labor and hardware on this fixed-price municipal program.

The simulation model used for this optimization project was so successful that a graphical user interface was added, and it was turned over to the municipality for use as a route planning tool.

Case Study: Stacking Multiple Pattern Processors for Broad Functionality

This case study describes a prototype constructed to monitor network traffic for suspicious behavior. Suspicious behavior might be the introduction of a known threat (e.g., a virus that has been seen before), or just changes in system operation that indicate that something unusual is going on.

The approach used was to implement a hierarchy of intelligent applications, each designed to perform a very specific task in the detect-assess-react cycle. Since these tasks are of different types, different kinds of machine reasoners were used for each component.

Operationally, the system activates only levels of the hierarchy needed for an adequate response to the perceived threat. The prototype had three subnetworks being monitored; the user could inject a particular attack process that was unknown to the system.

This Cognitive Intrusion Detection (CID) prototype is supposed to detect elevated activity levels on the network, and activate successively more intelligent applications up the hierarchy until the top level actually automatically retrains the bottom level detector to stop this attack process from gaining entry in the future (Fig. 5).

Multiparadigm Engine for CID

A layer-by-layer description of the CID components follows; refer to Fig. 5.

Layer 1: Data reduction layer

It is in this layer that raw packet data are conditioned for ingestion by the system. For example, it is here that packets are rolled up into sessions. The system processes these sessions.

Fig. 5 CID model.
The prototype is merely reading pre-sessionized data from disc. It would be far too complicated to incorporate a sessionizer into a demo, and would not add anything to the demonstration of the eight-layer architecture.

This system is using real session data. The basic data contain no attack sessions: they consist of normal sessions only. The demo does, however, allow the operator to inject real attack sessions into the data stream so that the response of the system can be observed.

The architecture for the prototype assumes a network having three separate subnetworks. Each subnetwork is processed independently, illustrating that the eight-layer architecture can be distributed in a natural way.

Layer 2: Intrusion detection layer

It is in this layer that CID engines process the sessions. The prototype has three real, trained nearest-neighbor classifiers (one for each subnetwork). The classifiers were trained separately on slightly different data sets. This illustrates the fact that the eight-layer architecture can have multiple, diverse imbedded CIDs.

Layer 3: Correlation layer

It is in this layer that the reporting results from all the subnet CIDs are brought together, along with system performance information. For this prototype, the correlator is ingesting synthesized subnetwork packet loading information, and statistics on e-mail traffic for each subnetwork. In practice, any measurable operational phenomena could be processed in layer 3.

The prototype correlator uses a real, fully parametric rule base to implement a fuzzy correlation. This correlation merges disparate pieces of evidence to develop estimates of the likelihood that the system is in various operational modes (e.g., nominal vs. intruded). The rule sets create Conclusion Justification Reports, which are written to the display for review by the operator.

Layer 4: Fusion layer

It is in this layer that all the information generated by layers 1–3 is brought together in an intelligent join, which is interpreted by a real, operational Bayesian Belief Network (BBN).

This layer is also responsible for checking support data; to illustrate this, the prototype simulates access of a database of known attack phenomenology. Synthesized results are passed back to the system, and are used in subsequent processing.

The prototype’s BBN consists of 11 nodes in four levels. These nodes correspond to components of the system state variable, and logically partition this variable into modes that are mutually exclusive within each level. Default prior conditional probabilities are specified at initialization time; these are updated dynamically as the system transitions through various states to yield optimized assessments of system state.

Layer 5: Indications and warnings

It is in this layer that alerts and status are developed based upon notifications from the layers below.

This layer is only invoked when the layer 4 (fusion) determines there are indications the system might have been compromised (based upon system behavior, whether the CIDs detected an intrusion or not).

The prototype’s decision engine for layer 5 is just a case statement that selects from among several reasonable alert and warning messages, based upon information for the layers below. In practice, it would probably be a knowledge-based expert system (KBES) or a decision tree.

Layer 6: Reactive response

It is in this layer that the alert and status information from layer 5, along with information from earlier layers is reviewed, and hypotheses are developed about the system state. These hypotheses are expressed in the form of an option list, from which subsequent layers may select courses of action.

The prototype’s decision engine for layer 6 is just a case statement that selects from among several reasonable options for action, based upon information for the layers below. In practice, it would probably be a KBES or a decision tree.

Layer 7: Awareness

It is in this layer that the system reasons about the network state, and selects actions from those presented by layer 6, or decides that no action is required. These actions are bound into a system level action plan which is passed to layer 8 for implementation.

The prototype’s decision engine for layer 7 is just a case statement that selects from among several reasonable options for action, based upon information for the layers below. In practice, it would probably be a KBES or a decision tree.

Layer 8: Proactive response

It is in this layer that the system-level action plan developed in layer 7 is carried out. Any action the system is capable of performing can be commanded from this layer. For the prototype, there are three actions that are
supported: do nothing, close/watch a port, or retrain a CID.

When layer 4 determines that there has been a network attack by reviewing system behavior and checking a database of known attacks, but the attack session was not detected by the subnet CID, it will retrain the CID. It retrieves the sessionized data from the attack sessions (e.g., by using the database information), constructs a training set, and retrains the failed CID dynamically as the system runs. This prototype actually implements the retraining capability, illustrating the fact that the 8-layer architecture supports self-awareness and online self-correction.

TUTORIAL ON NNS

NN as the basis for a computing paradigm (both supervised and unsupervised) have been around since the work of McCulloch and Pitts (1943), and perhaps longer. The neural approach has had a colorful history, and has been successfully applied to some very hard problems. One of the first generally acknowledged commercial applications was the use of one-layer NNs for time-domain noise cancellation on long-distance phone lines. This is an unsupervised application, because the machine is not trained on examples of noise types; rather, it dynamically adapts its response to previously unseen line noise as it occurs.

The Neural Analogy

The human brain has tremendous processing capability. If this capability has a physiological basis, it might be possible to build artificial analogs of the human brain that exhibit some of its processing characteristics. Of course, it is entirely possible that this function follows form approach will not work; we can try it anyway.

The human brain appears to consist of many billions of small processing elements (living neurons) that are organized into cortices, lobes, and hemispheres, and highly-interconnected to form a massively parallel device (Fig. 6).

Artificial NN are constructed to mimic this brain architecture. Artificial NN are highly-interconnected networks of simple artificial processing elements called neurons (Fig. 7) that have been organized (trained) in such a way that their stimulus/response patterns solve domain problems.

So, while the conventional approach to hard problems has led to the development of sophisticated compute-bound single-processor systems with few inputs and outputs, the neural approach has led to the development of primitive I/O bound multiprocessor systems with many inputs and many outputs.

Artificial Neurons: Their Form and Function

Some (but not all) artificial neurons are direct analogs of biological neurons. The Threshold Logic Unit (TLU) is one such commonly used analog and will serve as the basis for the following discussion. The TLU is in many ways archetypal: it appeared early in the history of the field, and most other artificial neuron architectures have formal and functional characteristics, very much like those of the TLU. A direct comparison of the TLU with a biological neuron shows clearly the structural and functional similarity of the two.

The TLU (Fig. 7) emulates the biological neuron in both form and function. Input stimulation (in the form of numeric data rather than ions) arrives at the TLU, where each datum is multiplied by a weighting factor to simulate the various dendritic sensitivities (e.g., their electrical resistances, or gains). Just as an electrical charge obeys the superposition principle in accumulating on the cyton, the TLU sums the weighted inputs, and applies a response function.

If a threshold check determines that the resulting transformed weighted sum is sufficiently high, the TLU fires, forwarding its transformed weighted sum for processing by other neurons. Notice that the simulation of dendrite action in the TLU is performed by the computation of the dot product of the input data with the input weights. It is, therefore, a correlation measurement. For the TLU of Fig. 7, note that if the response function is \( R(t) \), then: \( N(x, y) = R(ax + by) \).

In order to perform useful work, both biological and artificial neurons must be arranged in networks so that they can process and forward information. Infinitely many arrangements (topologies) are possible, but layered architectures occur frequently in nature and

Fig. 6 A biological neuron.

Fig. 7 Functional block diagram of an artificial neuron.
artificial neural network. In a layered architecture, neurons are organized into successive layers, with layer $N$ receiving its input from layer $N-1$, and forwarding its output to layer $N+1$. In artificial NN, layers are usually either one or two dimensional arrays of artificial neurons. The layer which receives raw input data from the world is called the input layer. The layer which dumps its output back out to the world is called the output layer. All other layers have no direct contact with the world, and are called hidden layers (Fig. 8).

### Using NN to Learn Complex Patterns

One of the fundamental application problems in computing today is the development of systems that can carry out the rapid, reliable, automatic recognition and classification of complex patterns.

Artificial NN are naturally suited to solving pattern classification problems through machine learning. The TLU of Fig. 9 has two input weights, $a$ and $b$, and its response function is the identity function, $R(d) = d$, where $d = ax + by$ is the dot product of the input ordered pair $(x, y)$ with the vector of input weights $(a, b)$. A firing threshold of minus infinity is assumed, so the neuron will always produce an output. This single artificial neuron, viewed as a one-layer neural network, is a linear classifier which assigns positive values to points in one half-plane and negative values to points in the other. With properly chosen values of $a$ and $b$, it would discriminate between the two clusters shown in Fig. 10. The values of $a$ and $b$ are determined during training, and will depend upon the application.

The 1-neuron network just described can ingest any pair of numeric features and render a linear classification decision. It could, for example, be used to classify computers as good or bad, where $x$ is cost and $y$ performance. For any particular candidate computer, measurements of the two features $x$ and $y$ would be made, the feature vector $(x, y)$ formed, and the neuron allowed to render its decision. This decision will appear at the neuron’s output as either a positive or negative number. To consider more than two features, additional input weights must be used, and the dot product becomes longer, say $d = ax + by + cz + ew$ for four features.

A more general problem is the assignment of a phenomenon to one of several classes based upon its feature vector. For example, it might be desirable to classify a seismic event into one of several possible categories, based upon a suite of numeric features obtained through direct measurement. For such a problem, it is natural to build a neural network with the same number of output neurons as there are classification categories. After training (to obtain the correct input weights), the neural network can be shown newly measured feature vectors, and the output neuron producing the highest response will correspond to the machine’s classification decision. Further, the relative magnitudes of the output responses can sometimes be used to develop a network confidence factor, and catch-22 output neurons can be used to flag I-Don’t-Know answers, or classify to subcategory.

Of course, few interesting problems yield to a linear classifier. It has been shown that artificial NN with
multiple layers can solve arbitrarily complex, but well-posed, classification problems (Fig. 11).

For problems too hard to attack directly, learning machines provide an implementation strategy that gives results that are as good as the developer’s ability to collect examples.

Text mining takes a different approach to unsupervised learning. Text mining technology has two broad components: syntactic methods, and semantic methods. Syntactic methods are those that base their processing only on the representation of information in text: grammar, vocabulary, and term statistics. The simplest of these are so-called *bag of words* methods, which rely entirely on relative word frequencies, ignoring word order entirely.

Syntactic methods are based upon the assumption that particular words are chosen to say specific things about particular topics, and that by tabulating these correlations and co-occurrences, it is possible to indirectly infer the text’s latent information.

Semantic methods are those that base their processing on information content: aspects of *meaning* such as context, word order, document externals, denotative and connotative definitions of terms, and structure of discourse. Semantic methods are much more sophisticated and computationally expensive. They might use combinations of grammatical processing to parse a block of text, lexicons to assign meaning to words, and ontologies to recognize and assign meaning to phrases, figures of speech, idioms, and perhaps even similes, metaphors, and other analogies.

Semantic methods generally require the use of a *domain ontology*: a structure that enumerates and characterizes all domain entities, and gives the facts, rules, and processes governing their operations and interactions. An ontology is a comprehensive, high-fidelity domain model. They are difficult and expensive to construct.

Syntactic methods, being statistical in nature, can be used without any understanding of the actual meaning of a document. In fact, when using bag of words methods, it is often not even necessary to understand the language involved as long as terms can be reliably identified and counted. However, this can be a problem for inflected languages.

When semantic methods are used, it is often in conjunction with, and subsequent to, the application of syntactic processing. This presents an opportunity to understand text using both structure and meaning by building a hybrid syntactic/semantic application. An obvious architecture for such an application would be to cascade the functionality by first applying inexpensive syntactic methods, and invoking the more expensive (Fig. 12).

**MAKING SYNTACTIC METHODS SMARTER:**

**THE SEARCH ENGINE PROBLEM**

Suppose we have a collection of documents, \( D = \{d_1, d_2, \ldots, d_M\} \). Each document is composed of character strings, which we call terms. Most terms will be words, but some will be numbers or more complex symbols, such as 12/25/2011.

Consider the search engine problem: locating a document or set of documents based upon a list of search terms (in this context, these are often referred to as keywords). Assuming that the search will be using an exact match (usually referred to as a hard match) on the search terms to identify documents of interest, what list of keywords is best?

A bit of thought suggests that this problem is analogous to the clinical screening problem: when screening...
for a particular disease or disease group in the general population, what set of tests is best? In the clinical context, the measures of accuracy used for such a disease search are sensitivity and specificity.

- **Sensitivity**—The proportion of persons with the condition who test positive.
- **Specificity**—The proportion of persons without the condition who test negative.

In the search engine problem, a search having a high sensitivity will be able to identify instances of the desired pattern. It will have a high detection rate. A search having a high specificity will be able to identify (and therefore, reject) instances that do not have the pattern. It will have a low false alarm rate. These are exactly the characteristics desired: a good search result is one that has all of what you want and none of what you do not want.

There are many ways our hard match search engine could be implemented. Here are three simple bag of words strategies:

- **Implementation 1**—A document is returned if, and only if, it contains every term in the keyword list.
- **Implementation 2**—A document is returned if it contains at least \( N \) of the terms in the keyword list, where \( N \) is a user-adjustable parameter.
- **Implementation 3**—A document is returned if it contains any of the terms in the keyword list.

The stringent match criterion of Implementation 1 will probably give few false alarms, but will also miss many good documents. It is specific, but not sensitive. The loose match criterion of Implementation 3 will probably return most good documents—along with many false alarms. It is sensitive, but not specific.

Implementation 2 is a compromise that falls between these extremes. However, it requires that the user provide a search-specific tuning parameter that they probably cannot estimate accurately. It is a step in the right direction, but a step that the user cannot take.

Additional analysis is required. Thinking again about sensitivity and specificity as they relate to keyword selection, two facts are seen:

- A keyword is **good** for a given document if that document uses it multiple times, because a term that a document uses frequently is likely to be related to what the document is actually about. This will make a search more sensitive to those documents most likely to be relevant.
- A keyword is **good** if it is in documents that are about the topic of interest, but not in documents about unrelated topics. This will enable the search to detect and ignore irrelevant documents, making a search more specific.

These two considerations suggest a way to calibrate search terms by assigning each one a numeric weight that is determined by its effect on sensitivity and specificity. Moreover, these weights can be computed \emph{a priori} by the search engine, and need not be provided by the user.

The perfect keyword would occur many times in the documents I want and nowhere else. That probably won’t be the case for most keywords, but we can still create a metric that measures how close a particular keyword comes to this standard of perfection for a particular document.

**A Submetric for Sensitivity**

Recall that our entire collection of documents is \( D = \{d_1, d_2, \ldots, d_M\} \). We form the set of all terms that occur anywhere in any document; call it \( W = \{w_1, w_2, \ldots, w_L\} \).

Define:

\[ T_i(w_j) = \text{number of times term } w_j \text{ occurs in document } i \]

From this, it is possible to compute the total number of times a term occurs, counting multiplicities, in the entire collection \( D \) by adding the occurrences in each document:

\[ w_i = \sum_{k=1}^{M} T_k(w_i) \]

To measure a keyword’s sensitivity to a specific document, we compute the proportion of all occurrences of a term that are in that one document. This is called the **term frequency** for that term and document. The term frequency (TF) for term \( j \) in document \( i \) is:

\[ TF_{ij} = \frac{T_i(w_j)}{\sum_{k=1}^{M} T_k(w_j)} \]

This is the number of times \( w_j \) appears in document \( i \) divided by the number of times \( w_j \) appears in all documents, and \( M \) is the number of documents in the entire collection.

The term frequency is a number between 0 and 1 inclusive. It is zero for documents in which the term does not occur. If there are lots of documents, \( TF \) will be a very small number for most terms and it will be a larger number for rare terms. If a term occurs in only
one document, its $TF$ will be 1 for that document, and 0 for all other documents.

### A Submetric for Specificity

To incorporate a metric component that will keep precision high, we must measure how common a term is across the entire set of documents. Terms that occur in lots of documents will not be very discriminating for any particular document. This is the reason that certain parts of speech are poor standalone search terms:

We will return to this concept when we discuss stopwords below.

The specificity of a search term for a corpus $D$ of documents measures whether the occurrence of that term is concentrated in a small percentage of the documents, or found in many of the documents. A natural way to measure this is to compute the proportion of all documents that contain the term.

Define:

$$Aw_j (d_i) = 1 \text{ if term } w_j \text{ occurs in document } d_i, \ 0 \text{ if it does not}$$

The total number of documents among the $M$ documents in $D = \{d_1, d_2, \ldots, d_M\}$ that contain term $w_j$ is then given by:

$$DocCount(w_j) = \sum_{k=1}^{M} A(w_j, d_i)$$

Then the proportion of all documents that contain the term is given by the document frequency (DF) for $w_j$:

$$DF(w_j) = \frac{DocCount(w_j)}{M}$$

The $DF$ is a number between 0 and 1 inclusive. It is 1 if the term occurs in every document and it will be smaller number for terms that occur in few documents. If a term occurs in no documents, its $DF$ will be 0 for all documents.

### Combining the Submetrics to Obtain a Single Score

A good search term $w_j$ for a document $d_i$ will have a large $TF$ and a small $DF$. These can be combined into a single metric by taking a ratio where the large = good submetric $TF$ is in the numerator, and the small = good submetric $DF$ is in the denominator.

To avoid carrying around a quotient of fractions that is four lines high, it is customary to write this quotient as a product of $TF$ with the reciprocal of $DF$ referred to as the Inverse Document Frequency ($IDF$). A term’s $IDF$ value is usually quite a bit larger than its $TF$ values, so it is customary to take the logarithm of $DF$ to control its magnitude so that it doesn’t overwhelm $TF$:

$$\log(IDF) = \log\left(\frac{M}{\text{DocCount}(w_j)}\right)$$

Since there is no need to compute $DF$ for terms that do not occur in any document in $D$, $\text{DocCount}(w_j)$ will always be at least 1.

Combining the sensitivity enhancing $TF$ submetric with the specificity enhancing submetric $\log(IDF)$, we obtain:

$$\text{TF.IDF} = \left(\frac{\text{TF}(w_j, d_i)}{\text{IDF}(w_j)}\right)$$

$$= TF(w_j, d_i) \log\left(\frac{M}{\text{DocCount}(w_j)}\right)$$

The $TF$. $IDF$ score for a term is a numeric measure of how specific and sensitive that term will be as a keyword for documents on topics associated with the term. The embedded period in the name is a reminder that this is the product of $TF$ and $IDF$. (The log is usually taken using to the base 2, but the base doesn’t really matter.)

Some important observation can now be made:

1. TF. IDF can be pre-computed for every term in a document by just counting terms and performing a couple of quick calculations. It boils down to computing some term and document histograms and taking log-normalized ratios. This can be done efficiently and quickly even for large, dynamic collections of documents. When documents are added or removed from the corpus, updating the term score is accomplished by adjusting the frequencies for the affected terms.

2. Because this computation is performed by the search engine at the document repository, the user need not provide weights or thresholds for a search. They simply have to choose salient keywords, which will then be TF. IDF, weighted by the search engine.
3. The search engine doesn’t necessarily need to look at all the words in a document to determine whether to return it on a search. By sorting the TF. IDF terms scores within a document, the search engine can match on only the best keywords within each document—those with the highest TF. IDF values. These will be the most sensitive and discriminating terms in that document, so in theory, this will give both high sensitivity and high precision.

Putting It All Together: Building a Simple Search Engine

Given what has been done to this point, describing a step-by-step process for building a simple search engine is relatively easy.

It is assumed that a corpus of documents is to be searched using keywords. The goal is to conduct a document retrieval application that will return documents based upon a list of search terms supplied by the user.

Note: Single-word search terms only; no multiword phrases for our simple engine.

Step 1

Prepare three word lists: a spell-check list, a stop-word list, and a synonym list, and use them as follows:

The spell-check list is just a large word list. A good one to start with is the Orchy Word List, which is available free online. It contains over 100,000 English words.

To perform a naive spell-check for a term, locate the term in the spell-check list. If it is present, assume it is correctly spelled. If the term is not present, either discard the search term, or replace it with a near-match from the spell-check list.

A stop-word is a word that serves only a structural purpose (e.g., a, the), so it will never be helpful in a search. Lists of stop-words can be found on the Internet (Fig. 13).

Before using the search terms passed by the user, remove any that are in the stop-word list. These contribute nothing to our simple search.

The synonym list will be used to allow the search engine to provide a limited search by concept capability. Instead of requiring the user to select exactly the right search term in exactly the right form (e.g., plural/singular, present/past tense), append the synonyms for each search term to the search term list passed by the user.

Synonym lists are available free on the internet (e.g., a public-domain version of Roget’s Thesaurus).

Note: To really make this search fast, apply spell-check and stop-wording to all the documents in the corpus.

Step 2

For each document, compute the TF. IDF scores for the words it contains (Fig. 14).

Step 3

Place the $N$ keywords having the highest TF. IDF scores for each document into a keyword table for the documents. It is the terms in this table, rather than the document itself, that will be matched. In Fig. 15, $N = 10$.

If you want to try to assign some kind of meaning to terms, then a lexicon must be created (Fig. 16).

The Objective Function for This Search Engine and How to Use It

Document retrieval is done by evaluating an objective function based upon the TF. IDX scores for the search terms, as follows:

1. Spell-check the search terms passed by the user.
2. Remove search terms that are in the stop-word list
3. Append search term synonyms to the search term list if a pseudo-semantic search-by-concept capability is desired.
4. Step through each of the rows in the document keyword table (Fig. 15). For each document, retrieve the terms from the search term list that are also in the document’s keyword table entry. Place the matching terms into what we will call a match list for that document.
5. For the terms that are in the match list for a document, sum the TF. IDF scores from the TF. IDF table (Fig. 14). This gives the match score for this document, and completes the computation of the objective function for this document.
Data Analytics – Data Mining: Forensic Analysis

**Fig. 14** Computation of TF.IDF scores.

**Fig. 15** Document keyword table.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>bilingual</td>
<td>195</td>
<td>0.1411378</td>
<td>4.560678</td>
</tr>
<tr>
<td>3</td>
<td>fram</td>
<td>92</td>
<td>0.1187613</td>
<td>2.809603</td>
</tr>
<tr>
<td>4</td>
<td>nomex</td>
<td>1697</td>
<td>0.1419402</td>
<td>7.60569</td>
</tr>
<tr>
<td>5</td>
<td>coalescing</td>
<td>1076</td>
<td>0.32602</td>
<td>5.995965</td>
</tr>
<tr>
<td>6</td>
<td>resurface</td>
<td>893</td>
<td>0.36302</td>
<td>6.605676</td>
</tr>
<tr>
<td>7</td>
<td>jeleplag</td>
<td>761</td>
<td>0.1310189</td>
<td>7.60569</td>
</tr>
<tr>
<td>8</td>
<td>apartheid</td>
<td>307</td>
<td>0.04087</td>
<td>4.560678</td>
</tr>
<tr>
<td>9</td>
<td>mcdouglas</td>
<td>620</td>
<td>0.34502</td>
<td>7.60569</td>
</tr>
<tr>
<td>10</td>
<td>saurophaga</td>
<td>1862</td>
<td>0.1610622</td>
<td>7.60569</td>
</tr>
<tr>
<td>11</td>
<td>lough</td>
<td>1852</td>
<td>0.87502</td>
<td>4.6206985</td>
</tr>
<tr>
<td>12</td>
<td>jai</td>
<td>712</td>
<td>0.1654027</td>
<td>4.469898</td>
</tr>
<tr>
<td>13</td>
<td>versed</td>
<td>1712</td>
<td>0.1184784</td>
<td>5.995965</td>
</tr>
<tr>
<td>14</td>
<td>repeaters</td>
<td>1281</td>
<td>0.1214953</td>
<td>6.219098</td>
</tr>
<tr>
<td>15</td>
<td>deficient</td>
<td>1125</td>
<td>9.31E-02</td>
<td>5.040443</td>
</tr>
<tr>
<td>16</td>
<td>schmidt</td>
<td>254</td>
<td>5.17E-02</td>
<td>6.605676</td>
</tr>
<tr>
<td>17</td>
<td>americans</td>
<td>104</td>
<td>2.70E-02</td>
<td>7.605392</td>
</tr>
<tr>
<td>18</td>
<td>transcribed</td>
<td>670</td>
<td>0.1553239</td>
<td>5.12046</td>
</tr>
<tr>
<td>19</td>
<td>segregating</td>
<td>836</td>
<td>2.90E-02</td>
<td>6.912245</td>
</tr>
<tr>
<td>20</td>
<td>abashidze</td>
<td>782</td>
<td>2.72E-02</td>
<td>7.60569</td>
</tr>
<tr>
<td>21</td>
<td>sacrificed</td>
<td>1192</td>
<td>0.1490607</td>
<td>5.995965</td>
</tr>
<tr>
<td>22</td>
<td>pollinizers</td>
<td>104</td>
<td>0.1186804</td>
<td>7.605392</td>
</tr>
<tr>
<td>23</td>
<td>padishah</td>
<td>1758</td>
<td>7.04E-02</td>
<td>7.60569</td>
</tr>
<tr>
<td>24</td>
<td>easter</td>
<td>1681</td>
<td>8.67E-02</td>
<td>7.60569</td>
</tr>
<tr>
<td>25</td>
<td>intensity</td>
<td>935</td>
<td>0.1361563</td>
<td>3.916513</td>
</tr>
<tr>
<td>26</td>
<td>concubine</td>
<td>400</td>
<td>0.1337654</td>
<td>7.605392</td>
</tr>
<tr>
<td>27</td>
<td>phoebebis</td>
<td>54</td>
<td>0.1150292</td>
<td>7.60569</td>
</tr>
<tr>
<td>28</td>
<td>trulemy</td>
<td>1506</td>
<td>0.1612902</td>
<td>6.912245</td>
</tr>
<tr>
<td>29</td>
<td>valenciana</td>
<td>80</td>
<td>0.1425678</td>
<td>7.605392</td>
</tr>
<tr>
<td>30</td>
<td>struggle</td>
<td>532</td>
<td>0.1372978</td>
<td>3.116757</td>
</tr>
<tr>
<td>31</td>
<td>kalighet</td>
<td>1030</td>
<td>0.139699</td>
<td>7.60569</td>
</tr>
<tr>
<td>32</td>
<td>sentiment</td>
<td>236</td>
<td>6.01E-02</td>
<td>4.309566</td>
</tr>
<tr>
<td>33</td>
<td>parsecs</td>
<td>1766</td>
<td>4.08E-02</td>
<td>5.655842</td>
</tr>
<tr>
<td>34</td>
<td>connection</td>
<td>95</td>
<td>0.1794118</td>
<td>2.99419</td>
</tr>
</tbody>
</table>

**Fig. 15** Document keyword table.
6. Return to the user the \( N \) documents having the highest match scores. Sort them in order of decreasing score so the best choice is at the top of the list.

That's it. You have used text mining to construct an objective function that implements a customized search engine for your set of documents, \( D \).

**Note:** The technique described here can be applied to data mining in any problem domain where fast search upon subpatterns is desired. This includes image search, graph search, and many other domains. This is obvious once you realize that TF.IDF is really just a statistical feature extraction method.

---

**SUMMARY**

Having read this entry, you now understand supervised and unsupervised mining methods as they relate to each other. You understand data clustering as an unsupervised mining process. You have seen examples of the formulation of objective functions, and a hierarchical architecture for unsupervised processing. You are familiar with the fundamentals of multilayer perceptron NN. You have seen how text mining can be used to build sensitive and specific search applications.