File system wide file classification with agents

Ben Martin

Information Technology and Electrical Engineering
The University of Queensland
St. Lucia QLD 4072, Australia

monkeyiq@users.sourceforge.net

Abstract

Many semi structured information systems such as file systems and email clients allow data to be tagged as belonging in many categories. Some such systems support notions similar to emblems, where files can be semantically tagged as fitting into a broad category by associating a file with an emblem. This paper presents a file system that makes use of Supervised machine learning for the creation of agents to offer fuzzy assertions and retractions of semantic tags on a per file basis. Such assertions are then subject to a belief resolution system to obtain an overall picture for a file's emblem attachments.

Keywords Information Retrieval, Document Management, Supervised Machine Learning, Semantic file system.

1 Introduction

This section first outlines some of the relevant concepts in file system design with emphasis on a particular implementation: libferris[1]. This should give the reader sufficient information to follow the presentation of emblems in libferris and how agents can interact with emblems to assert beliefs. It is important to show some of the capabilities of libferris so that the reader is aware of the sources of information that agents can access and how the user can efficiently recall data based on agent predictions.

The file system has become the de facto standard for the storage and management of semi structured data on computers. File systems have evolved from presenting a list of named objects (files) which contain a contiguous range of bytes to the modern file system comprised of a tree structure augmented with soft and hard links, sparse files, extended attributes and transparent support for many on-disk storage formats. Extended Attributes (EA) [2] allow the creation, update and removal of key-value data that is stored on a per file basis. File systems such as NTFS, ext3, reiserfs4 and XFS have support for EA.

The libferris project was created in order to provide both an enhanced API for file system access and to present more information through that interface. Libferris builds on the grounds of Semantic file systems [4], and thus provides both virtual directories populated with query results and support for extracting and presenting metadata from within files. The core abstractions in libferris can be seen as the ability to present many trees of information overlaid into one namespace, the presentation of key-value attributes (EA) that files posses, a generic stream interface for file and attribute content and the creation of arbitrary new files. Common stream interfaces and creation of objects are outside the scope of this paper[2].

The EA interface in libferris allows the presentation of both derived and persisted attributes. Each attribute has an associated schema many of which are derived from XSD basic types[3].

Overlaid trees are presented because one can drill into composite files such as XML, ISAM databases or tar files and view them as a file system. The overlaying of trees is synonymous with mounting a new file system over a mount point on a UNIX machine to join two trees into one globally navigable tree. Being able to overlay trees in this fashion allows libferris to provide a single file system model on top of a number of heterogeneous data sources[4].

Both fulltext and EA can be added to inverted file indexes [5] by libferris. This allows the set of documents that contain a term, satisfy a ranked query or have a given EA to be found quickly. Lookup on EA can be done on exact string match, regular expression string match, greater or less

[1] http://witme.sourceforge.net/libferris.web/

Proceedings of the 8th Australasian Document Computing Symposium,
Canberra, Australia, December 15, 2003.

[2] Interested readers should see [7] and http://witme.sourceforge.net/libferris.web/ferriscreate.paper2001/index.html respectively

[3] http://www.w3.org/TR/xmlschema-2/

[4] Some of the data sources that libferris currently handles include: http, ftp, db4, dvd, edb, eet, ssh, tar, gdbm, sysV shared memory, LDAP, mbox, sockets, mysql, tdb and XML.

than match. Comparisons are handled differently for attribute value matching based on attribute type and can be integer, double floating, binary or case insensitive string.

2 Emblems for classification

Although attaching and interacting with typed arbitrary key-value data on files is very convenient in libferris it leaves applications open to interpret the data how they choose. For this reason specific EA have been defined for semantic categorization of files on a system wide basis. These EA allow one to associate files with many emblems to show what categories those files are in. The set of all emblems ξ is maintained in a partial order $(\xi, \leqslant)$. The relation for $\mu, \varphi \in \xi$ of $\mu \leqslant \varphi$ means that μ logically `is-a` φ. Consider the example of marking an image file: one may attach the emblem "sam" to the image file. One of the parents of the emblem "sam" may be "my friends" to indicate that sam is one of my friends. It follows that the image file is also of one of my friends. This is due to the partial ordering on the emblems imposing transitivity on emblem assignment[5]. It follows that only the most specific emblems applicable to a file need be associated explicitly.

The collection of emblems that are associated with a file is stored in a single EA per user. This serialization of emblems is called a medallion and it is not recommended that applications read medallions directly. Each user is expected to have their own partially ordered emblem set and so medallions are attached to files on a per user basis. For application access the medallion is split into many EA by libferris shown in Table 1.

attribute	value	type
medallion.600	`<xml...>`	binary
e:has-cat	false	boolean
e:has-fuzzy-cat	0.0	double
e:has-animal	false	boolean
e:has-interesting	true	boolean
e:has-burnt	true	boolean
e:has-burnt-cd5	true	boolean
e:list	has-interesting, has-burnt has-burnt-cd5	string list
e:list-ui	has-interesting has-burnt	string list
e:upset	has-interesting has-burnt has-burnt-cd5	string list

Table 1: A medallion is broken down into its individual emblems at runtime. The "e:" prefix is really "emblem:" but is shortened for presentation.

[5]The assignment of an emblem μ to a file will also assign all the parent emblems of μ to that file

The assignment of an emblem to a file is called a belief and logically collects the: file, emblem, time of assertion or retraction, sureness of this belief and who holds this belief. To represent the holder of the belief a portion of the emblem set is used to define "personalities". There can be one belief held for each personality for a file and emblem combination.

As seen in Table 1 the emblem attachment, when attachment was done and the fuzzy belief for an emblem are all exposed as EA for each file[6]. By exposing this data as simple typed information it can be added to the EA inverted file index and files can be quickly found based on emblem association.

Because many beliefs about emblem attachment can exist for any given file a belief resolution system was introduced to show an overall picture about a file-emblem association. This is expressed in the form of a `double` ranging from `-100` for full retraction to `100` for absolute assertion. The default belief resolution gives the "user" personality veto status and if the user doesn't have a belief it presents the mean of all beliefs for the file-emblem association. An example of belief resolution is shown in Figure 1.

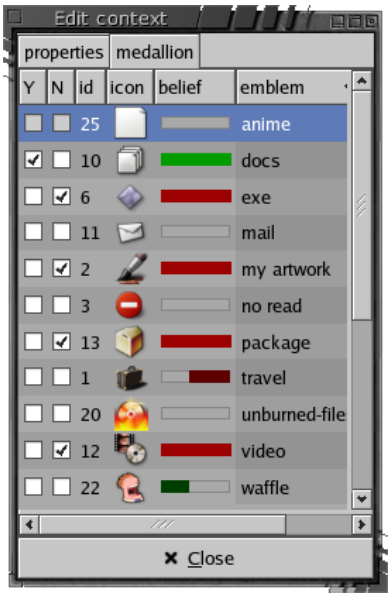

Figure 1: Viewing the emblems associated with file: docs is fully asserted, exe is fully retracted, agents have offered partial retraction on travel and partial assertion on waffle. Assertion is shown in green extending from left to right, retraction is shown in red extending right to left.

[6]All EA in the "emblem:" namespace are derived EA that are generated from the medallion EA for each file. The "emblem:" EA are not stored separately for each file.

3 Supervised Machine Learning

At an abstract level supervised Machine Learning (ML) [8, 6, 3, 1] involves two steps: training and classification. Of the many kinds of ML we are most interested in binary classification algorithms which can tell if a file is in a single category or not.

During training two lists are presented to the ML algorithm: a list of desired documents and a list of non desired documents. The ML algorithm then builds a model to use in its classification mode when it will tell if a presented document fits into the desired, undesired or unknown category.

The accuracy of the classifications that ML offers depends on the quality and size of its training data. Some ML algorithms can perform relatively better when offered limited training data. This is very important when attempting to use ML for file system classification where the user is not inclined to manually construct a large initial training collection. To adapt to the less complete initial model the system needs to be able to add and remove test cases and retrain agents quickly to support a more evolutionary model building process. The user can then enhance the model from a small initial one as they see agents as being a valuable part of the system.

4 Agents

To connect ML with libferris one can use emblem associations to train on and then in turn the ML can be used to obtain a fuzzy assertion or retraction for if an emblem is applicable for a new file.

A clear separation of the algorithm used for predictions (AgentImplementation) and the agent objects themselves (Agent) is maintained in libferris. This allows agents to be persistent objects that have a libferris view of the world, have a convenient interface and manage common state information while the ML code can maintain a more byte centric, non-libferris view.

There are many design choices when integrating ML into the file system. Firstly there is what ML algorithms to use. Usually the algorithm has different requirements for how much state is required during training, the size of the model produced during training, the ease with which the model can be updated, speed of execution in training and classification and relative quality of results.

Two algorithms have been chosen for initial testing: Bayesian [1] and Support Vector Machine (SVM) [6]. Bayesian filtering is very efficient in terms of speed and its model allows individual documents to be added or removed without having to retrain on all test cases.

SVM operates on the FeatureSets of its input files. A file's feature set is a map of each term to its relative importance μ to the document. The value of μ is usually calculated as the frequency of that term (TF) in the document multiplied by the Inverse Document Frequency (IDF) for that term. The IDF is the reciprocal of the number of times a term appears in all documents. Many SVM implementations require a collection of positive and negative FeatureSets to train on and can not "add" a new case to an existing model.

4.1 SVM agent implementation

We will now take a closer look at the SVM Agent Implementation (SAI). It should be noted that the SAI is mainly glue code between svm_light and libferris.

Generated FeatureSets are cached for two reasons: true belief capturing and efficiency. True belief capturing reflects that when the user highlights a document as a training example they are making assertions about that document at that point in time. For example, when the user asserts that they like a web page they are really talking about the page as it stands at assertion time. Calculating FeatureSets is a costly act and must be avoided in order to support timely addition of training examples when the number of existing FeatureSets is high.

The SAI reuses a lexicon implementation[7] from the fulltext indexing code to map unique terms to unique integers. These integers are subsequently used in FeatureSets to identify the terms. The IDF values are stored in a FeatureSet which is maintained by the SAI and updated by the SAI Trainer (SAIT).

To classify a file it needs to be tokenized and a FeatureSet μ generated for it. Then μ is multiplied by the IDF and normalized by dividing it by the euclidean length of the entire feature vector for that document. For example, given the initial FeatureSet $\mu = (x_1, ..., x_N)$ and the IDF FeatureSet $(\beth_1, ..., \beth_n)$ the result $\phi = (y_1, ..., y_N)$ is calculated as $y_i = x_i * \beth_i / \sqrt{(\sum_{t=1}^{N}(x_t^2))}$.

It is acceptable for the training of the agent to be a more costly operation than its use for predictions. The SAIT maintains a FeatureSet τ which is the total number of times each term appears in the Training Data (TD). When a new Training Case (TC) is presented it is tokenized and a FeatureSet of the documents term frequencies is created while at the same time τ is updated. The maintenance of τ is due to the many possible formulae for calculating the IDF [5]. If one is calculating the IDF as shown above then τ is not required and should be the element wise reciprocal of the IDF.

The two major efficiency requirements for the SAIT are the caching of FeatureSets and the ability to generate a current IDF FeatureSets quickly. The

[7] Currently supported lexicon storage: 3-in-4 prefix coded custom file format, Berkeley db, XML or as a file system

storage and update of τ achieves the latter goal while allowing flexibility in how the IDF is generated. The SAIT applies the same IDF multiplication and normalization as the SAI before training on those FeatureSets.

4.2 SVM agent testing

Tests were conducted in order to test both the implementation's correctness and the utility of the agent based classification under a relatively low training data size. A subset of the Reuters-21578 test collection[8] was used with only 179 positive and 191 negative cases. The Reuters-21578 files are a collection of news-feed stories and have been assigned zero or more categories based on their content. For this test the positive cases are documents that are about corporate acquisitions and negative cases are documents that aren't.

When trained on this data the SVM agent correctly predicted 5 of the 6 classification examples with the incorrect result's prediction value being closer to a zero than full assertion or retraction. These examples were chosen at random from the news-feed documents.

The SAIT storing all FeatureSets does consume more storage than is strictly required. For the above training example the SVM SAI consumed $1.7M$ of disk for all of its state, of which the lexicon was $508K$ (uncompressed in XML format) and the cached FeatureSets $399K$. The svm_light model was $474K$. Given that hard disk costs are well under two dollars per gigabyte it should not be considered unreasonable for agents to consume tens of megabytes for state information.

5 Future directions

The testing in Section 4.2 should be extended to include a larger test on the Reuters-21578 data and more general purpose prediction tasks such as the automatic assignment of emblems to academic papers based on past agent training.

Web browsers such as epiphany[9] currently allow the user to put a page into many categories instead of one as traditional browsers like IE allow. One of the main goals of agents was to allow a new style of bookmarking in web browsers such as ego[10] where the agents can offer the "top emblems" that are relevant to a web page and all the user has to do is accept this list. The interface would need to support overriding the predictions given by agents. Such overrides should be fed back into the training data for agents that offered an incorrect prediction. Also, whenever the user goes through the burden of assigning emblems to a website the browser interface should allow the page to form a training example for agents.

The interface for agents in libferris is generic enough to support image recognition agents. Applications for such agents could include assignment of an emblem if a particular person appears in a image or if an image.

6 Acknowledgements

The ML code used by libferris is bogofilter [1] for Bayesian and svm_light [6] for SVM. A big thanks to Thorsten Joachims (author of svm_light) for putting up with my pesky questions during development of my SVM agent. Kudos to Peter Eklund, Roger Duke and Robert Murphy for proof reading.

References

[1] bogofilter, http://bogofilter.sourceforge.net/. Visited Sep 2003.

[2] http://acl.bestbits.at/ ea and acl for linux website. Visited Sep 2003.

[3] Svmfu, http://five-percent-nation.mit.edu/svmfu/. Visited Sep 2003.

[4] Mark A. Sheldon David K. Gifford, Pierre Jouvelot and James W. Jr O'Toole. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principales*, ACM SIGOPS, pages 16–25, 1991.

[5] Alistar Moffat Ian H. Witten and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1999.

[6] T. Joachims. Making large-scale support vector machine learning practical. In A. Smola B. Schlkopf, C. Burges (editor), *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[7] Angelike Langer and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced programmer's Guide and Reference*. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 2000.

[8] Mark Rosen. E-mail classification in the haystack framework, 2003. MIT, Masters thesis, describes the design and implementation of a text classification framework for the Haystack project.

[8] http://www.daviddlewis.com/resources/testcollections/reuters21578/

[9] http://epiphany.mozdev.org/

[10] http://witme.sourceforge.net/libferris.web/