Implementing Hierarchical Heavy Hitters in RapidMiner: Solutions and Open Questions

Marco Stolpe and Peter Fricke
firstname.surname@tu-dortmund.de
Technical University of Dortmund,
Artificial Intelligence Group
Baroper Strasse 301, Dortmund, Germany

Abstract

Huge masses of data and potentially infinite data streams pose big challenges to methods in data mining that analyse data off-line and in several passes. In the area of intrusion detection, algorithms that detect characteristical patterns in system call data could have to process several hundred megabytes of data per minute. We describe a plugin for the aggregation of data streams by determining frequent tuples of hierarchical elements, so called Hierarchical Heavy Hitters. We further discuss more general questions concerning the current state of RapidMiner in relation to stream mining.

1 Introduction

Current data mining research increasingly examines algorithms that are capable of detecting interesting patterns in or to learn prediction models from potentially infinite streams of data. The amount of data created per time interval is often too high to be saved efficiently on mass storage devices and to be analysed off-line in several passes by traditional data mining algorithms. Methods are needed that continuously update their internal models, adapting to underlying changes in a stream in real-time and incrementally. Even if no data stream is given, methods for mining streams can potentially be more efficient on huge data sets in comparison to algorithms that follow a more traditional paradigm. Also think of embedded systems and mobile devices which are highly resource constraint regarding processing power and memory, but could profit from data analysis. Examples are the reduction of energy consumption by the prediction of usage patterns or the automatic grouping of similar images on mobile phones.

In section 2, we motivate stream mining by a case study from the area of intrusion detection that deals with the automatic identification of applications. We shortly describe Hierarchical Heavy Hitters that were used by our research group to identify application behavior with high accuracy, but low consumption

of memory and processing power. Further, we describe implementation details and issues of an accompanying RapidMiner plugin. In section 3, we discuss our own approach critically and pose the more general question of how to integrate methods for stream mining into RapidMiner. The paper ends with a short summary and potential future research and development.

2 Aggregation of System Calls by Hierarchical Heavy Hitters

In the field of intrusion detection, malware is sometimes identified by comparing the system calls it executes with a profile representing the normal mode of execution. To achieve high prediction accuracy, it might suffice to compare relatively simple features, like the frequency of n-grams of system call sequences, where each call is simply represented by the name of its type [7]. If more complex features are needed, like information about parameter values, processing time naturally increases with a higher complexity of the representation. Being sensitive to the resource consumption of malware detectors, while nevertheless maintaining a sufficient prediction quality, is therefore vital.

In [4] and [5], approximation algorithms for finding Hierarchical Heavy Hitters (HHHs) [1] in data streams were used to aggregate system calls and their parameters. The resulting sets of hitters were interpreted as characteristical profiles and, by the introduction of new distance measures for sets, used to identify individual applications. By profiling system calls of eleven applications over intervals of about one minute, we were able to achieve a cross-validated (leave-one-out) classification error of 8.7% by using 7NN and our newly developed similarity measure DSM. In comparison, we arrived at only about 21.7% with the naive approach of comparing log files by the relative frequency of contained system calls. Resource consumption could be kept fairly low during all runs of the algorithms. Even processing the biggest log file and using all available hierarchical variables resulted in a memory consumption of just several hundred kilobytes, compared to 764.287 system calls in several megabytes of the original log file. The whole data set contained 1,8 Gbyte or about 23 million lines of system call data from eleven different Linux applications, monitored five times over ten minutes and five times over five minutes each. All data was gathered with the *strace* tool (version 4.5.17) under Ubuntu Linux (kernel 2.6.26, 32 bit).

For further information regarding the exact representation of system calls and conducted experiments, please consult [4]. In the following, we will describe implementation details and issues of the accompanying RapidMiner plugin, after shortly introducing the HHH problem.

2.1 Hierarchical Heavy Hitters

The *heavy hitter problem* consists of finding all frequent elements and their frequency values in a data set [1]. For a multiset S of size N and a threshold

$0 < \phi < 1$, an element e is a *heavy hitter* if its frequency $f(e)$ in S is not smaller than $\lfloor \phi N \rfloor$. The set of heavy hitters is then $HH = \{e | f(e) \geq \lfloor \phi N \rfloor\}$.

The following problem can be stated if the elements in S originate from a hierarchical domain D [1]:

Definition 2.1 (HHH Problem) *Given a (multi)set S of size N with elements e from a hierarchical domain D of height h, a threshold $\phi \in (0, 1)$ and an error parameter $\epsilon \in (0, \phi)$, the* Hierarchical Heavy Hitter Problem *is that of identifying prefixes $P \in D$, and estimates f_p of their associated frequencies, on the first N consecutive elements S_N of S to satisfy the following conditions:*

- *accuracy: $f_p^* - \varepsilon N \leq f_p \leq f_p^*$, where f_p^* is the true frequency of p in S_N.*
- *coverage: all prefixes $q \notin P$ satisfy $\phi N > \sum f(e) : (e \preceq q) \wedge (\nexists p \in P : e \preceq p)$.*

Here, $e \preceq p$ means that element e is *generalizable* to p (or $e = p$) in the lattice induced by forming all possible combinations of hierarchical values and their generalizations. For the extended multi-dimensional heavy hitter problem introduced in [2], elements can be multi-dimensional d-tuples of hierarchical values that originate from d different hierarchical domains with depth $h_i, i = 1, \ldots, d$. In our case, features of system calls were encoded as tuples of values stemming from a handcrafted taxonomy of system call types, the file system hierarchy and a hierarchy induced by the possible combinations of call sequences having a fixed length. For instance, the HHH (`FILESYS/open/*`, `/etc/hosts`, `fstat64/*`) would mean that the event "file /etc/hosts is accessed by an open call with an arbitrary read write mode, preceded by a fstat64 call" occurred frequently in the stream.

There exist two variants of algorithms for the calculation of multi-dimensional HHHs: Full Ancestry and Partial Ancestry. Both have been implemented in a plugin for RapidMiner[1]. For further information concerning the particular details of the algorithms, please consult [3]. In the following, we will instead focus on an overiew and description of the accompanying packages, classes and operators.

2.2 The `hitters.*` package

The algorithms for the approximation of HHHs from data streams, their helper classes and test code can be found in the `hitters.*` packages, in particular `hitters.multi`. As the packages have almost no dependencies to RapidMiner, they also could be used on their own.

The core of package `hitters.multi` is class `AbstractComplexHHH`, which as an abstract superclass contains all common functionality of the Full Ancestry (class `FullAncHHH`) and Partial Ancestry (class `PartAncHHH`) variants. After

[1]The code, all data sets and installation instructions can be downloaded at `http://www-ai.cs.uni-dortmund.de/PUBDOWNLOAD/HHHPlugin/`. The plugin was written for RapidMiner 4.6, but work is in progress to port it to version 5.

initialization with parameter ε and some information about the used hierarchical variables, single stream elements can be (incrementally) inserted into the data structure by calling the `insert` method. All stream elements themselves are instances of class `Element`, which represents tuples of hierarchical values as native Java arrays internally and provides helper methods for the construction of such tuples. In case the taxonomy a hierarchical value originates from has unknown depth, like in the case of file system paths, hierarchical values are simple strings. Path elements are separated from each other by slashes ("/"), similar to Unix file system paths. For taxonomies of known depths and with static values, hierarchical values may be represented by single integer values. Each component of the path must then be (statically) mapped to particular bits of the integer. The exact mapping can be defined by instances of the `Parameter` class, which is expected by several methods that deal with hierarchical value tuples. By using integer values, the run-time of all related operations could be considerably decreased, since path elements are extracted by constructing and applying appropiate bit masks and bit operations are fast. In addition, the bit representation saves lots of memory, since the `Element` class is also used for frequent prefixes in the internal data structures of the algorithms.

At any time, it is possible to get the current set of hitters, whose elements are again represented by the `Element` class. The set is determined internally by approximating the elements' frequencies and comparing them to threshold parameter ϕ. The method returns a hash map which contains all elements (heavy hitters) as keys and further information about an element, like its estimated frequency, as a value of type `MultiHitterInfo`.

The domain specific functionality for the analyses of system call data generated by *strace* is contained in class `MultiDatabase` and its dependent classes. It allows to read in a whole log file by calling method `readSystemCalls`. Internally, all lines are parsed by regular expressions und transformed into tuples of hierarchical values (`Element` instances). By iterating over this set, single elements can then be incrementally inserted by calling the `insert` method.

2.3 Integration into RapidMiner

For the domain specific aggregation of *strace* generated system call data and resulting hitter sets, the whole functionality described above is wrapped by a single RapidMiner operator `HHHExtractionPlain` and a special `ResultObject`, `HHHResult`. As input, the operator receives an `ExampleSet`. Each example represents one single log file from which the hitter sets are to be extracted as a characteristical profile. Regarding later classification tasks, this aggregation is a preprocessing step. The log data isn't saved in the examples themselves, but instead referenced by the file system path to the log file. This solution was chosen since log file sizes can vary and be big. It was not straightforward to see how several megabytes of system call data could be represented efficiently by a single feature vector. The remaining attributes represent weights for the individual types of system calls and allow for the automatic selection of relevant calls. For further information, see [4]. The result of `HHHExtractionPlain` is an

`ExampleSet` that contains the calculated set of hitters for each log file. Again, it was unclear how to encode sets of varying sizes in a single feature vector. As a solution, we used a special attribute of type `ObjectAttribute` (see [6]), which can map arbitrary Java objects to `double` values as they are needed by RapidMiner's internal data structures like `DataRow`.

The parameters of operator `HHHExtractionPlain` allow for the choice between the Full Ancestry and Partial Ancestry variants of the hitter algorithms and the specification of ε and ϕ. Moreover, it can be specified which hierarchical variables to use and, if needed, one can limit paths from the hierarchy to a particular depth. In addition to outputting sets of HHHs in an `ExampleSet`, optionally a `ResultObject` can be output which allows for viewing the calculated HHHs in a table. For faster processing in iterative scenarios like a cross-validation, which makes repeated use of the same hitter sets, the sets can be cached on hard disk and don't need to be calculated repeatedly.

The resulting hitter sets can serve as input to already existing RapidMiner operators, like kNN (see [4, 5]). Special similarity measures have been implemented for HHHs and can be used in all operators that make use of similarity measures. For example, one might also cluster sets of HHHs.

3 Discussion

The implemented algorithms are true data stream algorithms, as elements from the stream can be inserted at any time into the data structures, potentially ad infinitum. Nevertheless, the `HHHExtractionPlain` operator doesn't read from a stream. Instead, all log files referenced by the entries of a traditional `ExampleSet` are read in a whole, returning a vector of stream elements. So although *strace* could in principle monitor system calls in a stream-like fashion, the whole processing doesn't appear stream-like at all. In a real-world deployment scenario, like a malware detector, this type of constant monitoring would be necessary.

We believe that the current shortcomings of our plugin can at least partly be explained by the lack of support for real stream mining in RapidMiner. The paradigm shift from traditional off-line to on-line processing of data needs to be supported by the right tools and frameworks. As RapidMiner is such a successful tool for the traditional off-line analysis of databases, we pose the challenging question if and how RapidMiner can be made into a tool that also allows for the design and execution of true stream mining processes.

Our proposal would be to introduce a general operator `StreamReader` that is capable of reading from streams which could be internally represented by pipes or sockets, for instance. The operator could gather data, like text lines, concurrently in the background and buffer them in a queue, either by starting threads or using asynchronous I/O. In a main loop, single elements from the queue could then be transformed into `IOObject` instances and pushed consecutively to the first nested child operator. A `StreamReader` would thus closely resemble the already existing "Loop Examples" operator, but instead read the examples from one or several streams. The implications of such a processing model aren't

entirely clear yet though and therefore need to be discussed further.

4 Conclusion

We have motivated stream algorithms by shortly presenting the positive results our working group was able to achieve by calculating HHHs from logs of system calls. We then described shortly the HHH problem and gave an overview and implementation details of the accompanying RapidMiner plugin. Although the plugin is ready for use to analyse data from *strace* logs, we have also discussed some shortcomings of our plugin regarding its stream mining character. We have also identified the current lack of support for stream mining in RapidMiner and proposed a new type of operator that might be considered as a first step in a new direction. We see the necessity to discuss and research this subject further. Future work on the plugin will include its transition to RapidMiner 5 and the introduction of an operator that allows for a more domain independent extraction of HHHs from data streams.

References

[1] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB '2003: Proc. of the 29th int. conf. on Very large data bases*, pages 464–475. VLDB Endowment, 2003.

[2] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data. In *Proc. of the 2004 ACM SIGMOD int. conf. on Management of data*, pages 155–166, New York, NY, USA, 2004. ACM.

[3] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in streaming data. *ACM Trans. Knowl. Discov. Data*, 1(4):1–48, 2008.

[4] P. Fricke. Datenaggregation von Betriebssystemdaten durch Hierarchical Heavy Hitters. Master's thesis, TU Dortmund, Computer Science, LS 8, 2010.

[5] P. Fricke, F. Jungermann, K. Morik, N. Piatkowski, and M. Stolpe. Towards adjusting mobile devices to user's behaviour. In *Proc. of the Int. Workshop on Mining Ubiquitous and Social Environments (MUSE 2010)*, 2010. To appear.

[6] F. Jungermann. Information Extraction with RapidMiner. In *Proc. of the GSCL Symposium Sprachtechnologie und eHumanities*, 2009.

[7] S. M. Varghese and K. P. Jacob. Anomaly detection using system call sequence sets. *Journal of Software (JSW)*, 2(6):249–278, 2007.