FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D8.3
Continuous Query Prototype

WP 8 – Applications: Web Content Analytics

Nature: Report

Dissemination: Private

Version History

Version	Date	Author	Comments
0.1	September 15, 2015	Soudip Roy Chowdhury	Initial Version
0.2	October 28, 2015	Philippe	Structure and introduction
0.4	December 15, 2015	Soudip Roy Chowdhury	Update on Implementation details and updates on other sections
0.5	January 21, 2016	Philippe Rigaux	Full review
0.6	February 10, 2016	Philippe Rigaux	Operators and implementation choices
0.7	February 12, 2016	Thanh LAI NGUYEN	Python/Scikit implementation
0.8	February 22, 2016	Christos Mantas	Spark/Mllib implementation
0.9	February 24, 2016	Philippe Rigaux	Profiling figures

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Contents

List of Figures

Abstract

We describe a continuous classification process which constitutes one of the major use cases of the MIGNIFY platform. We expose the requirements, the conceptual on-line classification workflow and detail the operators. We then describe two implementations, the first one running in centralized mode and the second one in distributed mode. These implementations are integrated in ASAP as alternatives that can be selected at run-time based on the profiled cost of the various operators and the execution context.

1 Introduction

The present deliverable covers the requirements, issues and technical specifications related to the *continuous* execution of data processing workflows. We consider the practical context of MIGNIFY workflows (or *pipes*) which combine functional blocks (or *agents*) for Web data collection, curation, classification and other analytic tasks. The workflow model has already been validated for simple, linear workflows executed once. In the present deliverable, we extend the specifications to consider more sophisticated workflows. In particular:

- we introduce periodic iteration, priorities and dependency constraints to model complex analytic workflows;
- we develop a real-life use case of a *continuous* workflow that serves as a driving motivation for collaborative work with our partners; the workflow implements an on-line classifier for structured data extracted from the Web;
- we examined candidates for alternative implementations of the operators involved in the workflow, and came up with two solutions, one, centralized, based on scikit-learn, and the other one, distributed, on Spark-Mllib; implementation features and profiles are given.

This corresponds to strong practical needs for the MIGNIFY platform, and gives rise to some intricate issues when it comes to determine when and how a workflow execution has to be triggered. The MIGNIFY service we chose to focus on is called OUTWATCH, and relies on classification methods applied to merchandising products and sales in e-marketplaces. Product descriptions are

extracted from on-line marketplaces and classified in some category based on their description, brand, product category, price and other features. Business-wise, this allows sales and marketing teams to carry out price comparison of products more efficiently, and more generally to investigate on-line offers in order to adjust their strategy accordingly. Such a BI tool can for instance help a company to adapt its marketing campaign or to make informed strategical decision for its sales and marketing department.

Due to the fast-paced dynamics of marketplaces, new products and new categories are introduced very often, whereas existing ones may become obsolete. This determines the introduction of continuous query execution in OUTWATCH, with two tightly associated computations.

1. The classification model needs to be adjusted based on new categories and obsolete ones; since we cannot rebuild the model from past dataset, we must resort to on-line classification.

2. As soon as it is adjusted, the model is applied to incoming product descriptions to predict their category.

We need, in principle, to (re-)execute OUTWATCH's workflows as soon as some of the data source adds new content. Ideally, the re-execution strategy should avoid any delay so as to always present up-to-date information to the customers. In practice, the nature of our process requires the workflow execution to be split in *batches*, i.e., a set of conditions regarding the properties of newly collected data (size, but also freshness, distribution, etc.) triggers a new execution.

OUTWATCH agents are particularly complex. A same product showcased in different marketplaces often presents a specific description. Use of different languages, encoding and structure variations, presence of irrelevant texts are some of the factors that make the matching process non trivial. It is also observed that the use of irrelevant text in a product description or sometimes missing descriptions (which we use as a feature for product matching) makes the whole matching process even more complicated. In order to address these challenges, OUTWATCH relies on machine learning (ML) techniques encapsulated in agents and incorporated in the Web data processing workflows. They constitute the most complex class of agents, from both a functional and technical point of view. They are also the most time-consuming ones.

The structure of the deliverable is as follows. Section 2 presents the functional and technical requirements of OUTWATCH and discusses the main challenges to address. Section 3 specifies the service workflows and details the operators. Section 4 covers implementation aspects and evaluation of the operators. Finally, Section 5 describes ongoing work and Section 6 concludes the deliverable.

2 OUTWATCH usecase

We now detail the use case of product matching workflow for marketplace.

2.1 Catalog and product offers

The general goal of OUTWATCH is to build and maintain a *catalog* of product references, and to discover *product offers* related to this catalog on public marketplaces.

1. A *Catalog* is a tree of categories and sub-categories. An example of category is *Coffee machines*, and a sub-category is *Expresso machine*. Any product that belongs to a sub-category also belongs to its parent category. We aim at classifying at the sub-category level for a better accuracy, and simply refer to it as "category" in the following.

2. A *product offer*, abbreviated POFFER, is an on-line proposal to sale one or several items of a product, with specific conditions (price, delivery, etc.).

If, for instance, some eMarketplace proposes 100 items of the coffee machine xxP34, at a given price YY, this constitutes a product offer for product xxP34.

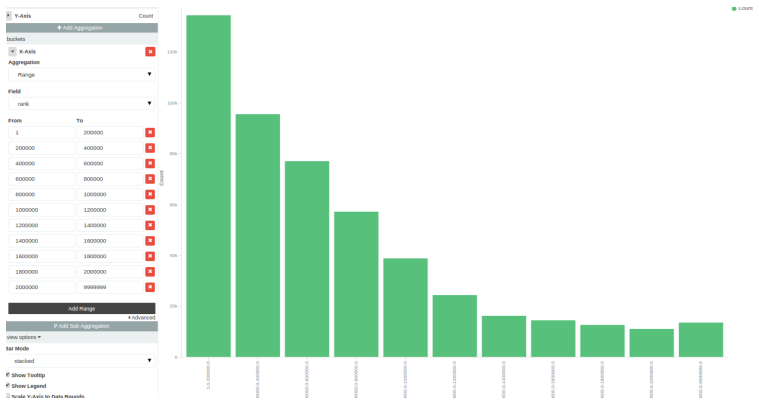

Figure 1: Ranking of product offers extracted from a large eCommerce site

Marketplaces constitute one of the most active and widespread economic activity on the Web, and the number of sites that can be considered as MarketPlaces is countless. Beyond the largest eCommerce companies, many small, community-oriented, web spots propose, in a dedicated forum, an exchange area where people can sell products and discuss their merits. The average evaluation of a product is an important information, as it helps to evaluate its popularity on a given site for comparison purposes. It can also be taken as an indirect indicator of the product sales. As an illustration of the reports supplied by MIGNIFY, Fig. 1 shows a graph showing an histogram of the product ranking from a large eCommerce site, built from the POFFER extraction.

At Internet Memory, we created a "Wep map" of classified sites that references hundreds of thousands of such potential spots. Our crawler scans the pages and identifies those that contain lists of products. We then, thanks to a semi-supervised approach, analyse the product page structure and produce a wrapper apt at supporting structured data extraction to obtain a product offer record.

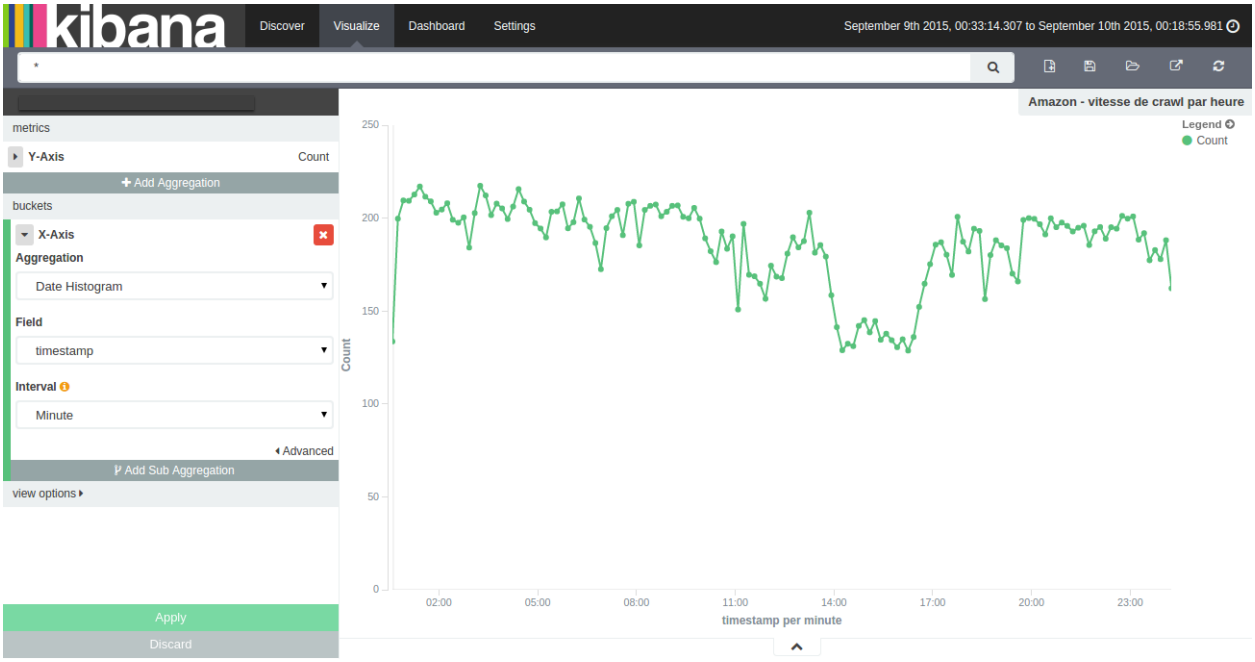

Figure 2: POFFER collecting rate from a large eCommerce site

This mixes all kinds of product-related information, brand, type, price, textual description, user comments. Needless to say, this information varies from one site to the other, in completeness, accuracy, language, structure, etc.

The number of POFFER crawled per day depends on the size of the site on the one hand, and on the crawl policy which is due to respect navigation and politness rules. Fig. 2 shows a graph showing the crawl efficiency for a very large site, where the politeness is the limiting factor (i.e., we cannot get the whole site catalog in one day). In this specific case, we collect about 300K POFFER per day. It takes about a week to obtain what we estimate to be the whole catalog (the site claims to propose 5M products, but we found many duplicates or almost-duplicates (a same book in several formats) which are merged by our data pre-processing workflow.

Once a site has been crawled, we revisit it periodically. The so-called refreshment policy depends on several factors and is decided by our crawl engineers. To state it briefly, what we consider as the ”golden” sites are continuously crawled, whereas less top-ranked market places are visited with a varying periodicity, ranging from a week to a few months.

Overall, we currently collect about 4 millions of POFFER daily, and we expect to increase our crawling rate constantly in a near future.

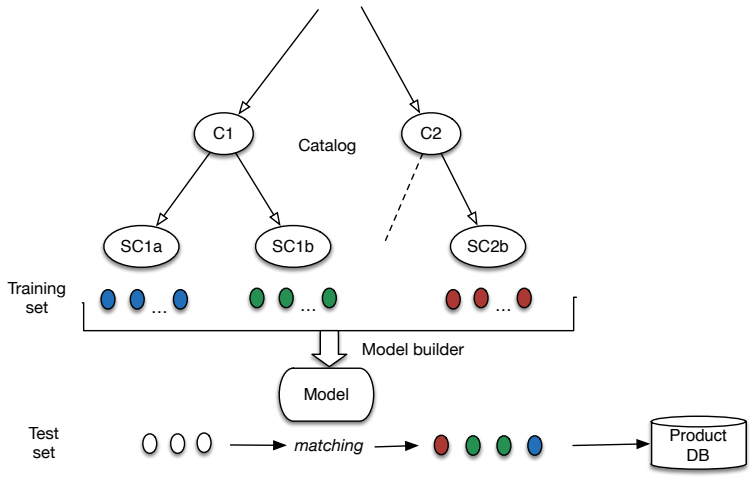

Figure 3: The classification process

2.2 Product classification

Beyond the data extracted from the page, it would be an important added value for us to be able to detect the category of the product, in order to supply our customers with a well-organized and accurate report of the web activity related to their specific business area. We therefore decided to undertake a study of a classification workflow apt at predicting a category from a POFFER description.

The *product matching* operation, denoted PMATCH, consists in associating a POFFER with a product category in the catalog. Given the description of an offer extracted from some e-marketplace, we try to find the category it belongs to. Given these concepts, OUTWATCH aims are twofold:

1. building and maintaining a product catalog covering the widest posible range of products that can be sold in public marketplaces;

2. discovering, via continuous crawling and matching, the POFFERs that can be related to the categories of the catalog.

We aim at building a large database where each POFFER is associated to one of the leaves of the catalog. From this database we will build reports and aggregations for brands and eCommerce companies. We will, for instance, report the list of POFFERs for a given category or sub-category, along with statistical indicators built from the prices, location, sales period, etc.

The assignment of a POFFER to a catalog is essentially a classification process, which infers, from the POFFER description and the catalog "model", the category of the POFFER. Starting from a training set which consists of a catalog and a set of labeled products, we build a model which is then used for assigning a label (a category) to each unlabeled POFFER supplied by our crawler/wrapper.

This process gives rise to two important issues. First, the catalog evolves constantly, and we need to adapt the model according to this evolution (a task called *on-line classification* in Machine Learning). Second, the PMATCH operation is difficult, because every vendor has his/her own way in describing the product. Non-uniformity in description, usage of noisy description and different languages add another level of complexity in the classification steps.

Handling continuous changes. New products are constantly proposed, and the catalog therefore constantly needs to be updated in order to accurately reflect the state of the market. The matching process is highly sensitive to these changes, as we need to maintain an up-to-date catalog of references, along with product details. The model by which a new POFFER can be matched with the catalog is therefore a continuously changing one.

Matching from partial and incomplete descriptions. Each marketplace typically maintains its own catalog and offers. For instance, an e-marketplace like FNAC (`http://www.fnac.fr`) has several million products and offers from thousands of vendors. The domain-specific web-scrappers (tailor-made for e-commerce sites) that can extract relevant POFFER description from the product pages hosted on e-commerce sites provide noisy and incomplete descriptions.

The text is often not grammatically well formed, i.e., different sellers often use different names for the same attribute. Usage of different languages and usage of emoticons and other unicode character in the product description add another level of complexity in the product matching and disambiguation.

2.3 Online classification

Figure 4 shows the (abstract) classification workflow of OUTWATCH. It relies on several operators that will be described in details in the next sections. We focus for the time being on the general process.

The input consists of POFFER descriptions supplied by the crawler / scrapper (not elaborated here). Since the crawler operates continuously, we constantly get new POFFERs from this source. In order to simulate a continuous execution, we split the input in *batches*. The actual reason for batch processing is explained in the next section

In each batch, we distinguish two kinds of POFFER.

1. *Training* POFFER*s*. They come from a set of seed websites (e.g., FNAC, CDiscount, Amazon etc.) which are considered reliable enough to provide a ground truth on the categories. POFFER issued from these sites are labeled with a category taken from the seed website itself.

2. *Testing* POFFER*s*. Product information extracted from other pages (other than the seed sites) are considered as the *testing dataset*, which requires labeling.

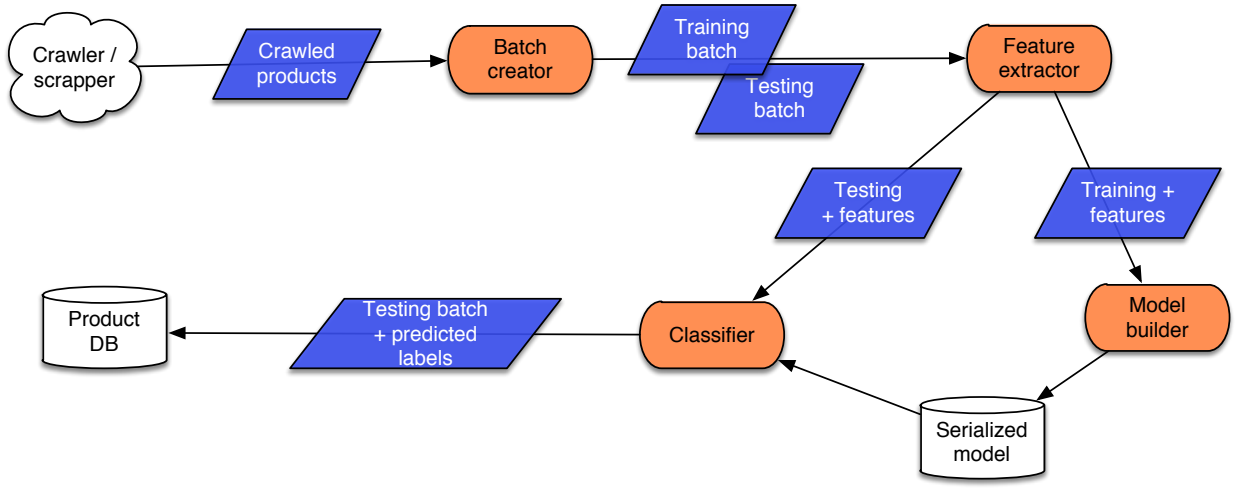

Figure 4: The OUTWATCH product matching workflow

Model building and updating. A model is initially built, and regularly updated, for each new batch, from the training set. We resort to a *supervised machine learning* technique. The reason for choosing the supervised learning was backed by the superiority of supervised learning over unsupervised for generating high precision for classification result. However, it suffers from performance bottleneck (for training and classifying) due to the high throughput of our dataset, which is a serious limitation for OUTWATCH in production.

To mitigate these challenges, we have made few design decisions in OUTWATCH, such as processing data in batches, serializing classification models or incrementally enriching models by using *on-line classification algorithms*. It consists of updating an existing model by considering a delta training dataset, as opposed to regular machine learning where the classification model is built from an entire training set.

Continuous classification. New products continuously arrive through the crawler stream. In principle, we are facing a never-ending classification process which takes each new POFFER as it is delivered by the crawler/scrapper, and submits it to the workflow. However, in practice repeating the model building and classification processes on per product description arrival is not a good solution since it adds additional performance overhead due to the I/O operations during the classification steps. On the other hand, processing the data in batches helps to fit in-memory the partial model trained on the training batches. Considering these aspects, in OUTWATCH we leverage batch learning and testing rather than single/online learning/testing cycles.

Since the product catalog is not static, re-training time for a model is another important aspect that we also considered while choosing the classification model in OUTWATCH. To address these requirements, we have implemented a model builder using linear stochastic gradient descent (SGD) algorithm with logarithmic loss function. The benefit of using SGD is to leverage the fact that the

gradient of the loss is estimated each sample at a time and the model is updated along the way (using partial model fitting). SGD also allows batch (out-of-core) learning.

Once the models are generated with the training set, the *classifier* acts on the testing set for generating labels associated with them. Classifier takes as input (i) the model, which is generated from the features and labels of training set, and (ii) the feature set from the testing data. It *predicts* the label for the testing data point for which the label was either missing or ambiguous in the original dataset.

2.4 Batches

In practice, the conceptually continuous processing scenario has to be decomposed in discrete, repeated workflow executions (called *runs* in the following), which take as input batches (groups) of products.

To overcome the performance bottleneck, in OUTWATCH we use out-of-core or external memory machine learning algorithms to process data that is too large to fit into a computer's main memory at once. However for such algorithm to work as expected, we also had to design mechanism to efficiently fetch and access data stored in the hard drive. One of the strategy that we have implemented in OUTWATCH is processing data in batches instead of one at-a-time or all at-a-time.

The *batch Creator* operator shuffles the data input, distributes it using stratified sampling method [2] into predefined batches, and stores them in temporary memory. Each batch constitues an input unit for the rest of the workflow. The benefits of having test and training sets in batches are twofold; firstly for large training dataset it helps to fit the partial model trained from sample batches in-memory, during testing it performs better since test data with similar features (similarity pre-computed for optimization) can be classified within a single pass. Considering these aspects, in OUTWATCH we leverage on batch learning and testing rather than single/online learning/testing cycles.

Scheduling and managing the batches decomposition is currently not supported by the ASAP execution model, and we therefore use an external ad-hoc scheduler. In the following we therefore focus on a single batch classification execution,, and discuss the creation and execution of sequences of batches in the concluding remarks.

3 The workflow

We now focus on the processing of a single batch, and examine in detail the operators. The workflow involves two phases: *feature extraction* and *classification*. The first phase produces, from the product description supplied by our wrappers, a set of features that describe a product offer and support the measure of similarity with other offers. The second phase combines the creation/update of the classification model, and the application of the model to the flow of product offers.

Several candidates for the features set have been considered. The tf/idf is a simple choice,

easily implemented and readily available in ASAP. It leads, however, to a rather poor classification precision (about 60%). We therefore evaluated a more sophisticated approach based on the *Word2Vec* model, as originally proposed by Google. Word2Vec takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words, present in the input corpus. The resulting *word vector model* captures word-association features, i.e., co-occurence patterns for words along with the context in the input text.

Using W2V requires a pre-processing step to build the Word2Vec model. This model is then used for feature extraction during the second phase.

3.1 Pre-processing

Figure 5 shows the W2V preprocessing step. It takes as input Web data suplied by the crawler, extracts a structured description of the products thanks to the web scrappers, processes textual information (tokenization, stop-word removal, etc.) and finally produces a W2V model, i.e., essentially, a matrix that associates to each token a vector of the token's co-occurrences indicators.

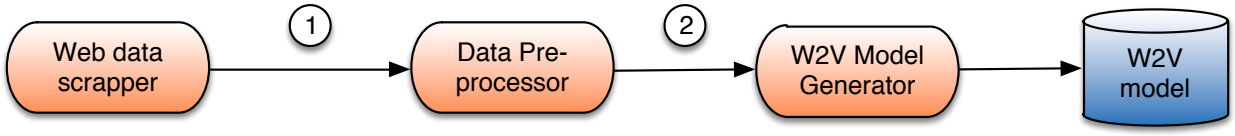

Figure 5: Preprocessing workflow: building the W2Vec model

3.1.1 Data preprocessor

This operator is responsible for preprocessing the raw data so that the following modules can effectively use the data for an accurate model. We use the *textual descriptions* of a product (e.g., product name, product description, price, brand etc.) as deterministic features for classification. Two of the natural language processing steps that we have exploited in our data preprocessing step are: (1) *Stemming* and (2) *Stop-word removal* from the raw input text. In the current use case of OUTWATCH, we implement our algorithms on the dataset (i.e., product catalog) extracted from French marketplaces such as FNAC.com. CDiscount.fr etc. To handle efficiently the product catalog data (such as product name, product description, etc.) for the classification purpose, we have used language processing libraries that are fine-tuned to handle French language, such as `French stemmer` implementation from the `nltk.stem.snowball` library, and stop words for French.

Our W2V model generator takes as input the features extracted from the trained dataset. However, for a sufficiently large product database these textual features (the term-document frequency

matrix) take a lot of buffer and application memory space, which becomes a performance bottleneck. To address this challenge, we have used other efficient data-structure for the feature storage. Since this term-document feature matrix is highly sparse for large volume of data, we transform this matrix to a hash table. In our current solution we rely on such feature hashing trick, similar to what has established by Weinberger et al. in [6] for scalable machine learning.

3.1.2 W2V model generator

This W2V model generator is responsible for generating the learned model that maps each discrete word id (0 through the number of words in the vocabulary) into a low-dimensional continuous vector-space from their distributional properties observed in the input text corpus. Based on several experiments, we chose to fix the parameters as follows. The `dimensionality` of the word vectors is set to 200 and the window size parameter for skip-gram to 10. In case we want to run this model building process in parallel, we set the `thread'` parameter as 10. To make sure the quality of the learned model (and to mitigate the risk of overfit or underfit for the model), we have set the iteration parameter `iter` in our algorithm as 5.

Once the model is built, it is used for subsequent Word2Vec feature extraction phases for the future input data.

3.2 The classification workflow

Figure 6 depicts the main, on-line, classification workflow. It represents the processing of one batch. The workflow relies on several operators, each detailed in the following.

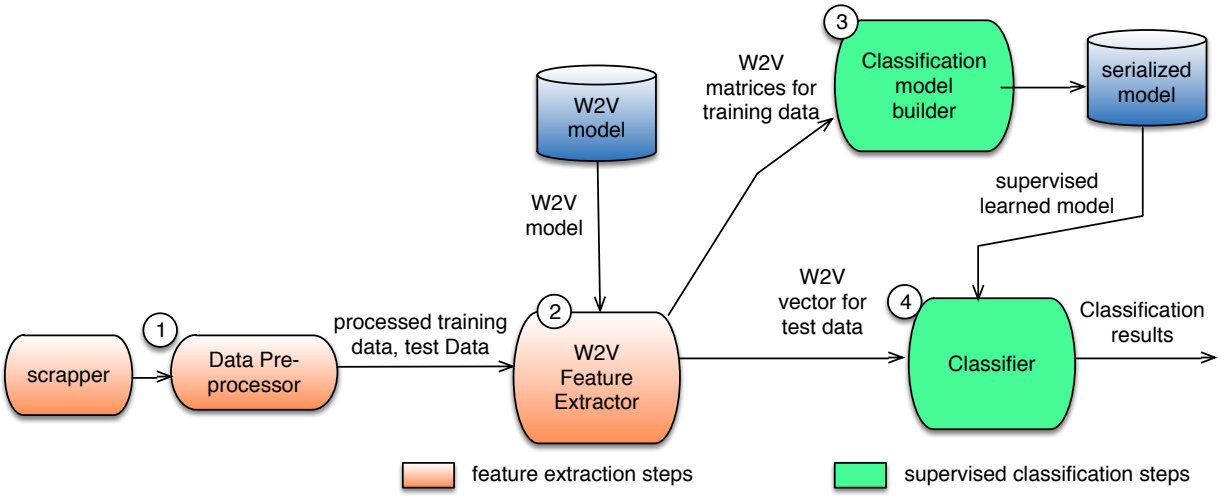

Figure 6: OUTWATCH workflow with Word2Vec as a feature extractor / processor

3.2.1 W2V Feature Extractor

This operator is responsible for deriving word-vector representations for the input data (i.e., both for the training and testing dataset). It analyses the input text, tokenizes it to find the terms, and derives the feature vectors for each terms based on the W2V model. Once the feature vectors for all the words in a text are derived, their values are aggregated and averaged to derive the feature vector for the line.

These aggregated feature vectors are then used for building the classifier model.

3.2.2 Classification Model Builder

It takes as input the vectors against each line in the training set as produced by the *W2V Feature Extractor* module, along with the label for the products (e.g., category) as present in the training set, and learns a model by using multi-class machine learning algorithms (such as Stochastic gradient descent logistic regression algorithms). Once the model is built, it is serialized and stored in a persistent storage.

Due to the continuous nature of the input, and the constant changes affecting the products catalog, products descriptions and products offers, the model needs to be constantly updated. We therefore restrict our choice to on-line classifiers apt at adapting an existing model to fit a new input batch.

3.2.3 Classifier

It takes as input the vectors against each line in the test set as produced by the *W2V Feature Extractor* module, along with the serialized model, which was produced by the *Classification Model Builder* and produces the labels for each of the line in the test dataset for which the labels are not known.

4 Implementation: centralized and distributed workflows

We implemented two versions of the operators involved in the workflow. Essentially, a first set of operators is based on the Python Scikit-learning library (`http://scikit-learn.org`). It cannot run in distributed mode but is adapted to small datasets, or as a backup solution if a distributed computing environment is not available. The second version relies on Spark MlLib,

4.1 Centralized implementation (Scikit)

The centralized implementation consists of a data-preprocessor, a word2vec model generator, a word2vec feature extractor, a classification model builder and finally a classifier.

4.1.1 Operators

Data pre-processor. The data pre-processor contains essentially a list of French stop words to be removed in the text description. For stemming and other text pre-processing tasks, we also use the *SnowballStemmer* of the NLTK package for Python.

W2V Model generator. The word2vec model generator is called on the output of the data pre-processor. We use the Python encapsulation gensim[1] of the C W2V code[2], wrapped as an operator, and configured as follows:

- `size`: 200 . The default vector's size is 100. We double the size to keep more context information.
- `window`: 10 . The default size for skip-gram. It is the number of surrounding words for a given word.
- `negative`: 0 the negative sampling is deactivated
- `hs`: 1 the hierarchical softmax is used for the training algorithm
- `sample`: 1e-5 . the sub-sampling of frequent words. Since we have a small number of words with a very high occurrence, we set this parameter to the lowest value recommended by word2vec
- `iter`: 5 default number of iteration
- `min-count`: 10 all words that appear less than 10 times are removed

The model file is then saved in binary format.

Feature extraction. Gensim lets us reuse the model in the feature extractor component. In this vectorization step, a product's text description is split into individual words. Each word is then converted into a numeric vector through word2vec model. Since text have varying length, we take the average of all word vectors as the input to a classification algorithm. It is an usual practice when we convert word2vec output into input for classification models.

Classification Model Builder. In the classification model builder, the SGDClassifier model from scikit-learn permits us to train the logistic regression classifier in an incremental manner. The "loss" parameter in SGDClassifier is set as "log", the number of iteration *n_iter* is 100, and the regularization parameter *alpha* is 0.0001. The model updates with new batches of data using the function *partial_fit*. When the training step finishes, the classification model is serialized to reuse in the prediction component.

[1] `https://radimrehurek.com/gensim/models/word2vec.html`.
[2] `https://code.google.com/archive/p/word2vec/`

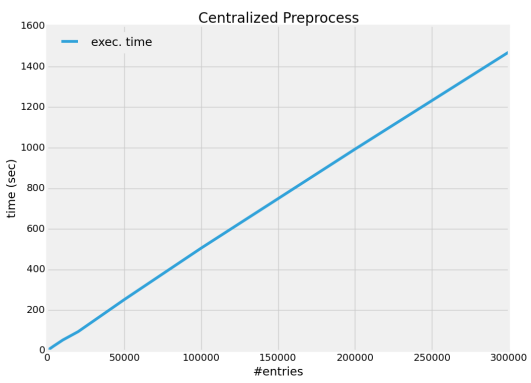

Figure 7: Pre-process

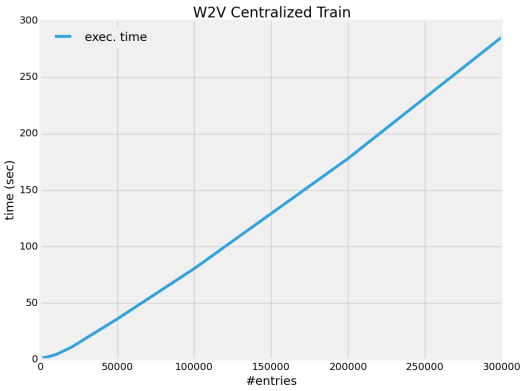

Figure 8: W2V Model Builder

4.1.2 Profiling

We ran the ASAP profiling modules on these operators. Basically, they all exhibit an almost perfect linear behavior with respect to the input size.

Fig. 7 shows the cost of the data pre-processing operator. It takes about 1/2 hour to process 300K POFFER in centralized mode. We recall that 300K POFFER is what we can get daily from a single large eCommerce site. A machine would therefore be able to process about 12M POFFER per day, which is sufficient in the moment, but might become a bottleneck in a near future.

Fig. 8 shows the cost of W2V model builder. It takes only 5mns hour to build the W2V model for a training set of 300K POFFER. Given that the W2V model is built off-line and can be used for feature extraction during a period of days or even weeks, this constitutes a marginal part that does not impact the classification use case.

The cost of the feature extraction operator and classifier (Figs. 9 to 11) is similar to that of the data pre-processing step. So, essentially, the same conclusions holds: a centralized implementation is enough for the time being to face our daily input of POFFERs, but is likely to become a major limitation as we will scale our crawling process. The classification model building performance lies in the same. Once the model is built, the prediction (classifier) presents a much better throughput, since about 10 mns suffice to classify our batch of 300K POFFERs.

To summarize this part of the study, a centralized implementation can handle the classification workflow as long as the daily number of collected POFFERs remains below 10M. It is therefore important to anticipate another solution, scaling to larger data flows.

4.2 Distributed implementation (Spark)

Spark's MLlib has its own distributed implementation for the Word2Vec algorithm. Both training a W2V model and vectorizing words using that model are supported. For large collections of doc-

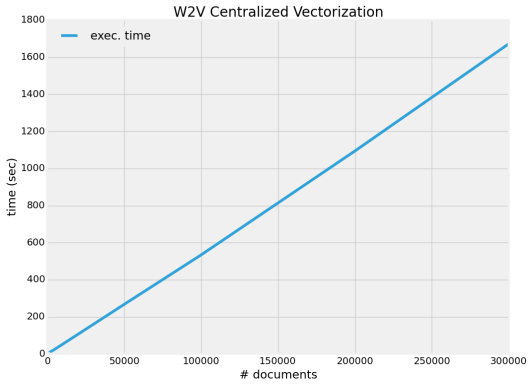

Figure 9: Feature extraction

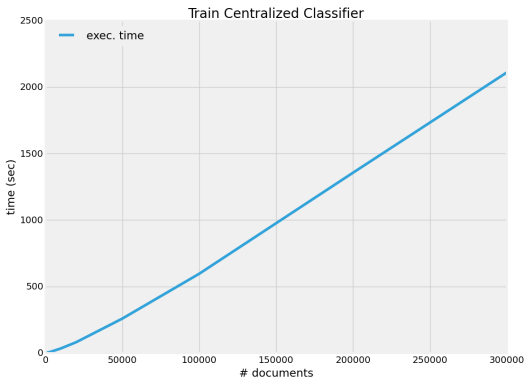

Figure 10: Model building

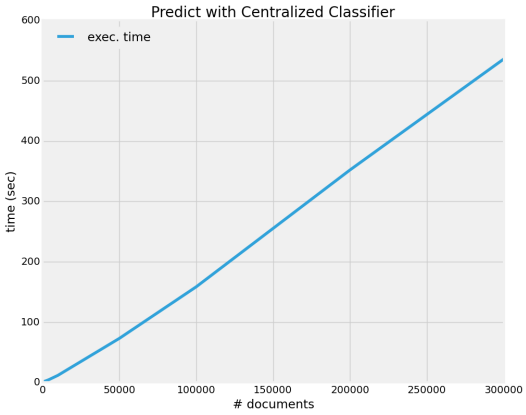

Figure 11: Classification

uments such a distributed implementation of those steps can be beneficial. We have implemented both those steps in Spark's MLlib, using its native Scala API.

4.2.1 Operators

W2V Model Generator. For a large input corpus we are training a W2V model using the top dictionary terms. The output model is stored in HDFS in order to be re-used later, in the vectorization step. Note here, that it is not theoretically possible to update a W2V model, as one needs the whole text corpus to produce it.

Feature Extraction. Given a previously trained model, in this step we use that model in order to vectorize a set of input documents. For aggregating the multiple word-vectors into a single vector

representing the whole document, we take the mean of all the word vectors' coordinates.

Classification Model Builder. MLlib has a rich feature-set with regards to classification. MLlib uses a general Logistic Regression Model that can be trained with a number of algorithms. The one we have opted for is an implementation of L-BFGS (Limited-memory BFGS). This uses an approximation of the Broyden Fletcher Goldfarb Shanno (BFGS) optimization algorithm with a capped memory footprint. BFGS is an iterative optimization algorithm that falls into the of Quasi-Newton family of methods. The choice of this particular training method was chosen as a best approach for multi-label classification. One of its advantages is that it allows updating a previously formed (trained) model with new training data. We have implemented the training and classification using the Python API of Spark's MLlib.

Train a Logistic Regression Model: Using a set of labeled input data we train a Logistic Regression model that is stored in HDFS. The input format is the same as the ouput of the W2V vectorization step. By default this step calculates the training error of the produced model (the percentage of the training data that the model would predict wrongly). It is also possible (by using an "evaluate" flag) that this step performs a cross-evaluation of the trained model with 20% of the input data as a validation set.

Update a Logistic Regression Model: As a variation of the previous step, the user can provide an "update" flag. In this case a previously stored LR Model is loaded from HDFS and updated with new training Data.

Log. Regression Classifier: Using a previously calculated LR model, this step classifies an input set of document vectors.

4.2.2 Profiling

The Spark operators have been profiled as well, and the results are summarized on Figs. 12 to 15, in a distributed cluster with 10 nodes. They show that the expected advantages of a distribution evaluation are confirmed for this particular workflow, an exception being the W2V model builder for which the C Google code, that runs from a binary compiled executable, is very well optimized and can handle large inputs very well.

First, partitioning the input pays off in terms of mere efficiency since operators like feature extraction and classification can operate in parallel. So, even for reasonably small datasets (less than 300K POFFER), we obtain much better performances. Second, and perhaps more importantly, each operation appears to be almost linearly scalable, exhibiting a modest super-linear behavior in ne the number of computing nodes. We can therefore safely envisage to scale up the number of POFFER, as long as the necessary computing resources are available.

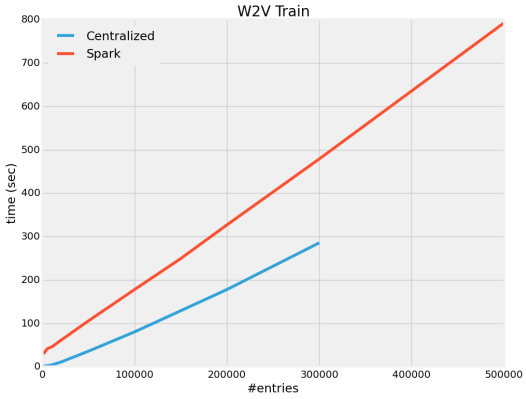

Figure 12: W2V Model Building

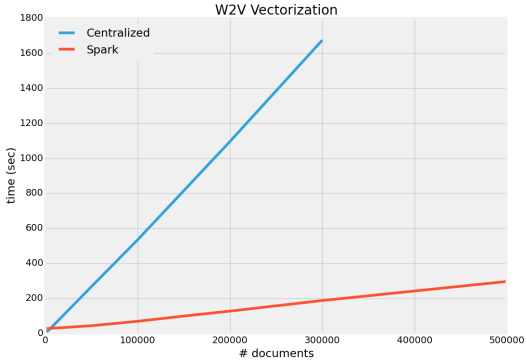

Figure 13: Feature extraction

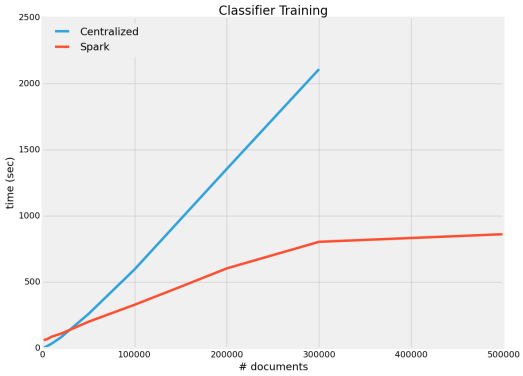

Figure 14: Classification model building

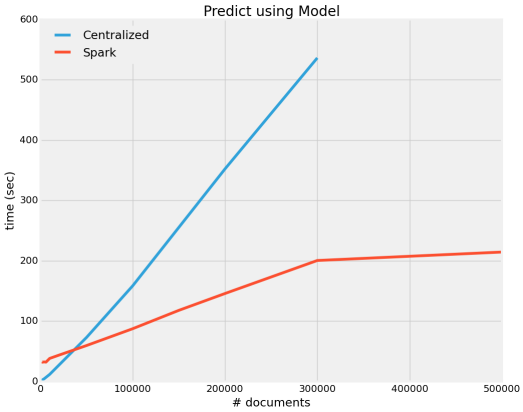

Figure 15: Classifier

The choice of a particular implementation, centralized or distributed, depends on the size of the input which, in our case, is highly variables. We might, for instance, run the workflow occasionally for a small marketplace dedicated to a particular class of product, in which case the centralized solution appears to be efficient enough, and avoids to overload a cluster with tiny tasks. Generally, if we consider a setting where the cluster could be smaller and the host running the centralized code could be a, much faster, physical machine, with faster dedicated Disk interfaces, the performance of the centralized implementation could be much better and the comparative advantage of the distributed implementation smaller. As a rule of thumb, from these experiments, we can expect that the centralized code would be faster for datasets up to 5-10K lines, which fits with flows of POFFER issued from small, highly specific marketplaces. The ASAP optimizer and scheduler is expected to be able to make an informed decision based on the contextual parameters.

5 Ongoing work

We are currently working on several extensions of the platform functionalities, and on the integration of the use case with the ASAP dashboard.

5.1 Dashboard integration

Extracting and visualizing context information from the textual descriptions contained in the collected POFFERs will result in a business intelligence tool that transforms noisy and unstructured Web content streams into valuable repositories of actionable knowledge[3]. In Year 3 of the ASAP project, we will ingest textual POFFER descriptions by means of the Document API developed in WP5, making the real-time stream of selected products online offerings available within the ASAP dashboard to be complemented by time series data on average daily prices for a given product, uploaded via the Statistical Data API. The integration into the ASAP dashboard of WP5 will extend price comparisons by (i) visualizing aggregated keywords computed from noisy textual descriptions contained in the POFFERs collected from e-commerce sites such as FNAC and Amazon.com, and (ii) identifying specific features that impact the perception of the product in online media coverage, creating additional value for sales and marketing decision makers as the main target group of business intelligence tools. We will focus on selected high-impact consumer goods such as smartphones, digital cameras, or popular car models.

Figure 16: Keywords associated with Googles Nexus 5x (left) and Apples iPhone 6 (right)

Fig. 16 demonstrates the use of knowledge extraction techniques to compare product perceptions via aggregated keywords based on international news media coverage on Googles Nexus 5x and Apples iPhone 6, respectively.

For a business intelligence tool based on price comparisons, the temporal context is of particular importance. Fig. 17 focuses on this context dimension by showing time series data on three products from the 'electric / hybrid car' category BMW i3, Tesla Model X and Toyota Prius (left). It also aggregates the longitudinal data to show relative share of coverage in international news media (middle), and uses a radar chart to convey brand personality based on the classification of Aaker[4].

[3]Scharl, A., Weichselbraun, A., Gbel, M., Rafelsberger, W. and Kamolov, R. (2016). "Scalable Knowledge Extraction and Visualization for Web Intelligence", 49th Hawaii International Conference on System Sciences (HICSS-2016). Kauai, USA: IEEE Press. 3749-3757

[4]Aaker, J. (1997). "Dimensions of Brand Personality", Journal of Marketing Research, 34(3): 347-356. (1997)

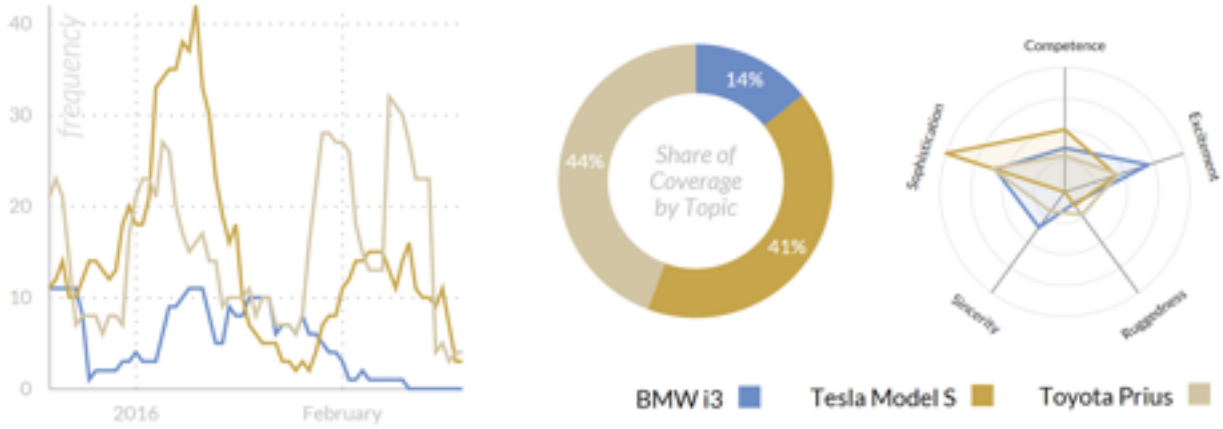

Figure 17: Time series, share of coverage and brand personality (BMW i3, Tesla Model X, Toyota Prius)

5.2 Batches

The decomposition of the input in batches transforms the conceptual continuous execution in a practical sequence of ”static” executions, one for each batch. The ASAP workflow model requires an explicit triggering of an execution. In the MIGNIFY context, although we are able to run a delta-computation that only considers the products newly collected with respect to the previous execution, there is no ”scheduler” that could automatically decide on the appropriate moment to run a workflow. Relying on a human decision jeopardizes the ability of the system to provide timely results.

We can therefore envisage an evolution of the workflow mechanism that can automatically detect conditions (based on *predefined rules*) to trigger new (re-)execution of a workfow on a specific input. The scheduler should be invoked by *pushing* the (re-)execution instruction, e.g., for new every new data/batch arrival, It should trigger the *execution monitoring tool* to (re-)execute a workflow. Conversely, the scheduler would also *pull* some system information to trigger a (re-)execution. Triggering rules for (re-)execution can be of types:

1. Time-window based batch processing.
2. Frequency-based batch processing.
3. Other business rules based batch processing.

In summary, we can identify the need of a model and language for defining rules for continuous executions of workflows. This also gives rise to the need of run-time monitoring the rules for triggering a new execution.

6 Conclusion

We studied in this task a much more sophisticated workflow that the simple linear sequential one implemented during the first year of the project. Motivated by some important requirements from the MIGNIFY OUTWATCH service, the workflow takes the form of a tree. After a common rooted branch, two separated computations take place:

1. The model is updated thanks to the new incoming labeled data (i.e., data taken from a seed site).

2. The updated model is used by the classifier.

This provides a simple but representative example of some features which were absent from use cases we had to deal with so far. First, the workflow exhibits a *priority* among edges (branches in general): the model has to be updated from the current batch *before* application of the classifier to the very same batch. Second, there is a *dependency* from one branch to the other: the classification should not start until the updated model has been produced and distributed. In general, this shows the need to introduce scheduling information in the workflow model to capture the semantics of the whole operation coordination.

We successfully identified several possible implementation choices for the workflow, implemented the operators and evaluated their individual performance thanks to the profiling facility of ASAP. This confirms that both solutions may be relevant, depending on the context (size of the input, available resources). At the time of writing, we are working on a full integration of these operators with the rest of the ASAP components (user interface, scheduler / monitor). We will evaluate the benefit of a high-level specification coupled with an automatic selection of the execution setting during the last year of the project, with the goal to come up with an industrial solution apt at saving human and computing resources in the management of our services.

References

[1] V. Kantere and M. Filatov. Deliverable 5.2 - Workflow management tool. Technical Report 5.2, ASAP, August 2015.

[2] David D Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In *Proceedings of the eleventh international conference on machine learning*, pages 148–156, 1994.

[3] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics*, pages 55–60, 2014. http://nlp.stanford.edu/software/corenlp.shtml.

[4] P. Rigaux and T. Sugibuchi. Deliverable 8.2 - Use Case Requirements. Technical Report 8.2, ASAP, February 2015.

[5] A. Scharl, A. Weichselbraun, A Hubmann-Haidvogel, and W Rafelsberger. Deliverable 6.1 - ASAP InfoViz Services Early Design . Technical Report 6.1, ASAP, February 2015.

[6] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D8.3
Continuous Query Prototype

WP 8 – Applications: Web Content Analytics

Nature: Report

Dissemination: Public