AGGREGATOR MANUAL

Author: Andrej Gajduk

Contents

Data flow of a typical aggregator

Three different sections which are executed at different times and with different frequency are the core of every aggregator. First the initial run fetches most of the data out on the web and does some initial processing. Second, the incremental run which is executed every 24 hours e.g. gathers any newly published data and does some processing on it. The third section is related to presenting the data to the user in your web app. These three sections are made up of different tasks as detailed in the Table below, where you will also find the relevant section for each task.

No.	Main task	Relevant section
Initial batch processing		
1	Crawl data from web	DataReader
2	Define the relevant categories and build the Category Map	CategoryMap
3	Extract features on the gathered data	FeatureExtractor
4	Train a classifier for each set of categories	Classifiers
Incremental processing		
1	Crawl new data	DataReader
2	Extract features on the new data	FeatureExtractor
3	Classify any new features that have missing information	Classifiers
4	Do clustering on your data to identify duplicate entries	Clustering
Front-end		
	Get relevant search options/categories	CategoryMap
	Get data that fits a specified search criteria	Querying
	How to represent near-duplicate entries in your search results	Clustering

Getting started

First of all you need to decide the domain of interest. For our example we will go with scholarship applications listings. You should find some relevant that are easy to access, no login required, no AJAX just pure simple HTML. Next you should write a DataReader which often uses regex to extract the relevant information from the html source. DataReader are web-site specific, meaning that you will probably need to write a new Data Reader for every site you wish to add to your aggregator. Best to start with a few sites, then add new ones with time. Finally, send your data readers to do their job, i.e. fetch data which you should then store in a database using the DatabaseManager class.

In the database you now have data amassed from different websites, where different pieces of information are available. For example one website may tag their scholarship data with fields of study such as economics, law, medicine, technology etc. while another one may have them categorized by their purpose like study or research. This is defined in the CategoryMap where you can specify all the possible categories and sub-categories that users will be able to browse and search on, e.g. a news site will have sports and politics in tier 1 with sports splitting to football and tennis in level 2. Naturally not all data entries have the same labels on every website. This is where pattern recognition comes in. You will use tools data analysis from Weka an open source library, instead of manually tagging each data object in your database.

To use Weka you first need to perform feature extraction on the data. You can find more about feature extraction on the web. To extract useful features in the aggregator you will need a FeatureExtractor which will process the data you have stored in the database and produce an Instances object that you can use directly with Weka. The aggregator comes with an implementation of the term-document-frequency metrics, ready to be applied to any text.

Now that you have an Instances object you can use it to train a classifier, e.g. to tell you whether a news story is about sport or politics. Note that when training your classifier you can only use data that has the topic (sport/politics) information fetched from the website. Some websites may not have this information. After you have trained your classifier it can now be stored in database and used later to classify any data that does not have topic information.

Another important thing you have to consider is the grouping of near-duplicate data entries. This is done by clusterers. As classifiers, clusterers too need the Instances object provided by the FeeatureExtractor. Clustering can be used to bundle together all listings of one same scholarship that is published on different websites, for example.

DataReader

This is the part where you have to do all the work because every site is its own. Once you decided upon a site i.e. www.MySite.com you need to create a new class that extends DataReader and implements the abstract methods

`void init(int max_links)`: find all the relevant links and put them in *links*
`Data getDataForLink(String link)`: extract any relevant information on a link

For the `init` function you will need to find a URL that lists all links to the data featured on that website. Usually an empty search or a browse button will do the trick. If there is paging system (and there almost certainly is) you can either set the page size to a large number and get all the links at once or loop through all the pages and fetch them in groups. Note that all data readers are required to make use of the parameter *max_links*. This just means that as you're populating the links list you should limit yourself to this number.

```
public void init( int max_links) {
        html = 
        UrlFetcher.fetchGet(“http://www.mysite.com/scholarships/search/?start=0&length=10000”);
                //we get all the link in one GET
        Matcher matches = getLinksPattern().matcher(html); /*the getLinksPattern function is 
        specific to every site*/
        while ( matches.find() && links _counter < max_links ) {/*we respect the max_links 
        parameter*/
                String link = matches.group(1);/*we get a valid url i.e. 
        “http://www.mysite.com/scholarships/browse?id=124156” */
                links.add(link);//we add the link to the list
                ++ links _counter;
        }
}
```

To implement `getDataForLink` function you should identify what information is relevant for you and what would like to extract. A good practice is to extract everything that seems remotely interesting even if currently you have no need for it. For this we recommend the Pattern class provided from Java. Always try to extract small elements from the page and organize them in a logical way. Don't just have one huge information element. To clarify what this means let us suppose the site provides the following information, again taken from the scholarship example:

Summary:
Duration: 1 month
Purpose: Study
Level: Bachelor
Benefits: travel expenses covered, student dorm accommodations
Description:
The University of “some university” offers these fine scholarships to all….
lengthy text … looking forward to next semester.

Instead of just having the contents split in summary and description you should extract the duration, purpose, level and benefits as separate elements. If you want you can nest them with summary so you keep the initial structure. In this step you can also specify what information will be used by the classifiers, although this can also be done later. The code for the data reader would look something like:

```
public Data getDataForLink(String link) {
        String duration = null;
        Matcher duration_matcher = duration_pattern.matcher(html);
        if ( duration_matcher.find() ) {
                duration = duration_matcher.group(1);
        }
        //repeat for every pattern
        //…
        //create a new TextData object
        TextData text_data = new TextData(title, source,new Date(),application_deadline);
        //create the content, layer by layer
        Content summary = new Content(“summary”);
        summary.addContent(“duration”, duration);
        //…
        //add content to the data object
        text_data.addContent(summary);
        //set the main content (the largest piece of text)
        text_data.setMainContent(description); /*it’s very important to set something as the main
content, because by default some of the feature extraction methods use only the main content*/
        //set which fields can be classified upon
        text_data.addCategoryInformaion(“level”,level);
        return text_data;
}
```

The DataReader is organized as a Java iterator meaning it should support the Boolean hasNext() and Data next() functions. These functions are already implemented to work with the links list so you don’t need to override them.

Now that you have written your DataReaders using one is straighforawrd.

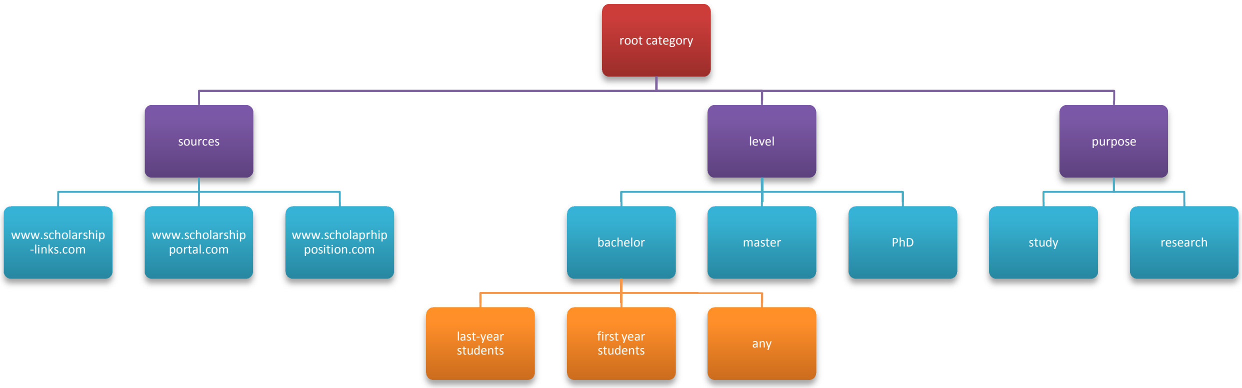
```
DatabaseManager databasemgt = DatabaseManager.getInstance();
DatabaseManager DataReader reader = new MySiteTextDataReader();
reader.init(1000);
while(reader.hasNext() ) {
        Data data = reader.next();
        databasemgt.addDataToProblem(data,”scholarships”);
}
```

If you wanted to check if the data already exists in the database you would use the method:

`databasemgt.checkDataForProblem(data,”scholarships”,compare_sources)` - the last argument should be true if you are only interested in matching with data from the same source (website).

CategoryMap

The category map is a very important part of our framework. It defines the logical way data is separated. Let’s see an example of a category map for our scholarships project.



You can create this CategoryMap with the CategoryMapBuilder

```
        CategoryMap cm = new CategoryMap();
        CategoryMapBuilder builder = new CategoryMapBuilder();
        builder.addClassToCategoryMap(cm,data,”level”);
        builder.addClassToCategoryMap(cm,data,”bachelor”,”level”);
        builder.addClassToCategoryMap(cm,data,”purpose”);
        builder.addSourcesToCategoryMap(cm,data);
```

You would normally do this after your initial batch of data crawling. Saving and loading from database is done by:

```
        databasemgt. saveCategoryMapForProblem(cm,”scholarships”);
        databasemgt. getCategoryMapForProblem(”scholarships”);
```

The CategoryMap is useful when querying on the database, as detailed next.

Feature Extractor

Feature extractors are paired with a data type e.g. the TermDocumentFrequency is paired with TextData. This means that you can use this feature extractor for any text data and only text data. This is good because you can reuse them (unlike DataReaders). Since FeatureExtractors are reusable lets see how you would use one, and leave the tutorial on writing one for later.

We assume that you have created you problem e.g. scholarships and populated its data with listings from different websites using your Data Reader. Before you start with automatic classification you need to extract features on your data which are later used by the classifier.

```
public FeatureExtractorData performFeatureExtraction ( FeatureExtractor feature_extractor , String
feature_extractor_name , List<Data> data ) {
        FeatureExtractorData feature_extractor_data = new FeatureExtractorData(feature_extractor_name);
        try {
                feature_extractor_data.setFeatures(feature_extractor.getFeatures(data));
        } catch (UnsupportedClassTypeException e) {
                e.printStackTrace();
        }
        feature_extractor_data.setDataset(feature_extractor.getDataset());
        return feature_extractor_data;
}
```

FeatureExtractorData is a data model that wraps all the features we get from a feature extractor on a given list of data. In particular it contains the following information:

`weka.core.Instances dataset` - an instances object containing the descriptions of all the attributes which your extractors measures or calculates i.e. TDF uses terms as attributes and counts their frequency so in the dataset we will have all possible terms we can encounter,

`List<Feature> features` – a list of the features, one for each data

A Feature is simply a feature vector (`weka.core.Instance`) and a `data_id(UUID)` to link it to the data from which it was extracted. If you want all the data you can just call the method: `databasemgt.getAllDataForProblem(String problem_name)` – which returns a list of all the data. To get a list of data which has some properties you can run a query on the database. For example to get all data where the level field is present in category information we would use:

```
DBCollection coll = dmbgt.getCollection(“data”);
DBObject query = new BasicDBObject(“category_information.level”,new
BasicDBObject(“$exists”,”true”));
DBCursor cursor = coll.find(query);
List<Data> data = new ArrayList<Data>(cursor.count());
while ( cursor.hasNext() ) {
        DBObject data_object = cursor.next();
        Data d = databasemgt.getParser().readValue(data_object.toString(), Data.class);
}
```

You can use the same method if you are trying to extract features for a huge list of data, so huge it doesn’t fit in memory. You iterate through them with the cursor and extract features on the fly:

```
//sample data
while ( cursor.hasNext() ) {
        DBbject data_object = cursor.next();
        Data d = databasemgt.getParser().readValue(data_object.toString(), Data.class);
        f_data.addFeature(feature_extractor.extractFeature(d));
}
```

After you have your feature_extracotor_data you should save it in your DB for later using

```
        databasemgt.addFeatureExtractorDataToProblem(feature_extracotor_data,”scholarships”).
```

If you want to add some new features to an existing feature_extractor_data (probably from some new data you just fetched from the web) you can use the method

```
databasemgt.addFeatureToFeatureExtractorDataForProblem(feature,”TDF”,”scholarships”);
```

Writing your own feature extractor

Now we look at how to write your own feature extractor. If you are not interested in this feel free to skip this part. You will first need to choose a feature extraction method. This is a well-studied area for common data domains i.e. textual data or image data. Next create a class and make it extend from FeatureExtractor.

You will need to override just one method which is:

```
        public weka.core.Instance getFeatures(Data d);
```

Note that sometimes to conserve space it is optimal to use weka.core.SparseInstance. Another thing that your feature extractor needs to do is populate the vector attributes with all the possible attributes that can be extracted from a data. The usage of attributes is encapsulated in the FeatureExtractor class and you should not change it. You can do this at construction time or as you are extracting features from a given data. However, your feature extractor should never add values for attributes not declared previously because then it will invalidate all your hardly-trained classifiers you have carefully stored in database. To fully enable the second way of populating your attributes you should also override the method:

```
        public weka.core.Instance getFeatures(Data d,Instances dataset);
```

in which you will only use the attributes already in the dataset and not create any new attributes. It’s up to the client to make sure he never invokes the getFeatures(data) after the attributes have been locked and put in database.

```
attributes.addElement(new Attribute(“numeric attr”));
attributes.addElement(new Attribute(“nominal attr”,{“value1”,”value2”, ”value3”} );
```

Remember what we mentioned above about every feature extractor being paired with a single data type. You should check this in the beginning of your extractFeatures method like so.

```
if ( data instanceof TextData ) {
        //all ok, continue with normal execution
else {
        throw new weka.core.UnssuportedClassTypeException(“Expected data type is
:”+TextData.class.toString()+” , passed was data of type :“+data.getClass().toString()+”.”);
}
```

If you are planning on writing your own feature extractor make sure you check the weka.core.Instance, weka.core.SparseInstance and weka.coreAttribute classes to see how you would use them in your code, or look at how TDF is implemented.

Classifiers

Classifiers are general in their use and are not related to the data domain. This means that the same classifier can work both on images and on text. You will not usually write classifiers yourself, instead you will probably use the ones that came with Weka. So lets just look at a few basic classifiers operation you would typically use in your application. Before a classifier is used it must be trained using a training dataset. A training dataset is simply a set of data represented by their features i.e. FeatureExtractorData for which we have class information, e.g. whether the data belongs in the sports or the politics section as specified on the website from where we originally crawled the data. This is how you would build a decision tree classifier (J48) to classify scholarships listing by their purpose (study or research).

```
Classifier  classifier = new J48(); //decision tree
FeatureExtractorData f_data = databasemgt.getFeatureExtractorDataForProblem(“TDF”,”scholarships);
FeatureExtractorDataHelper helper = new FeatureExtractorDataHelper();
Instances dataset = helper.getInstancesAndAddANewClassAttribute(“scholarships”,"purpose", true, null);
classifier.buildClassifier(dataset);
databasemgt.addClassifierToProblem(classifier, classifier_name, problem_name);
```

This code is rather straightforward except for the helper class, so let’s elaborate on that. Suppose you’ve run a TDF feature extractor on your data and you have all your features stored in your database. Now you want to classify on the “purpose” information which. To train your classifier you must tell him which of the features belong to which field (class). This is done in Weka by declaring a class attribute. This helper class makes your life easier by saving you the trouble of reading through the Weka documentation and adds the class attribute automatically as long as it is specified in your data when you fetched it from the web. You might wonder why didn’t we included this ‘class’ attribute while we were extracting the features. Well the reason is that the features you get from your feature extractor should be reusable for any field not just “purpose”, so as to avoid having to write a different feature extractor for each field.

The last two parameters of the getInstancesAndAddANewClassAttribute are boolean remove_missing_values, which if set to true will only return the features for any data objects that have the information on that particular field. This is what you would normally use for training the classifier, else if the third parameter is false the method will return the feature vectors for all the data – this is what you will want to use later for the actual classifying.

The second parameter is your category map which we will explain in more detail later but for it suffices to say that if it is null this method will find all distinct values of the field (second parameter) and save them in the class attribute.

Once we trained our classifier we want to use it to classify some data:

```
//fetch the classifier and the data from DB
Classifier  classifier = databasemgt.getClassifierForProblem(“purpose”,”scholarship”);
FeatureExtractorData f_data = databasemgt.getFeatureExtractorDataForProblem(“TDF”,”scholarships);

//add the class attribute using your pre-built category map (this is saved automatically when you
saved your classifier)
FeatureExtractorDataHelper helper = new FeatureExtractorDataHelper();
CategoryMap cm = databasemgt.getCategoryMapForProblem(“scholarships”);
```

```
Instances dataset = helper.getInstancesAndAddANewClassAttribute(“scholarships”,"purpose", false,
cm);
//classify any data object who don’t have class information
List<String> classes = new ArrayList<String>();
List<Integer> indices = new ArrayList<Integer>();
for ( int i = dataset.numInstances()-1 ; i >= 0 ; --i) {
        //if an instance is already classified, skip it
        if ( ! dataset.instance(i).classIsMissing() ) continue;
        //this is where the actual classification happens
        double class_value = classifier.classifyInstance(dataset.instance(i));
        //look up the class name, from its index
        String class_name = dataset.classAttribute().value((int)class_value);
        classes.add(class_name);
        indices.add(i);
}
//update the class information in DB
for ( int i = 0 ; i < classes.size() ; ++i ) {
        Feature f = f_data.getFeatures().get(indices(i));
        databasemgt.setCategoryInformationForDataForProblem(“purpose”,classes.get(i),f.getData_id
(),”scholarships”);
}
```

If you want to do this only for a specific set of data can run a custom query on the database. Another thing you might want to do is to compare the performance of your classifiers. Weka comes with a large pool of classifiers so you need to choose the right one for the job and the only way to do this is test their performance. This can be done using the Evalution class from Weka

```
Instances train = ...   //from somewhere
Instances test = ...    //from somewhere
// train classifier
Classifier cls = new J48();
cls.buildClassifier(train);
// evaluate classifier and print some statistics
Evaluation eval = new Evaluation(train);
eval.evaluateModel(cls, test);
```

Or if you can’t obtain a large amount of data to divide it into training and test partitions you can perform a cross validation.

```
Evaluation eval = new Evaluation(data);
eval.crossValidateModel(tree, data, 10, new Random(1));
```

This can also be used to adjust the parameters for your classifier. More about this can be found at http://weka.wikispaces.com/Use+WEKA+in+your+Java+code.

Clustering

Clustering is used to identify identical or near duplicate of your data that would frequently appear on different websites. You can’t automatically evaluate your clusterers and you will have to manually inspect your data and see if produced clusters are appropriate. A good choice is to use a hierarchical clusterer where you have more direct control over the clusters creation.

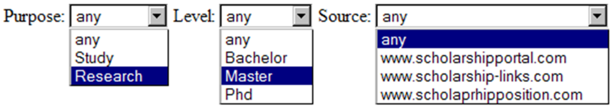
Using clusterer is pretty straightforward.

```
//create the dataset (list of attributes)
Instances dataset = feature_extractor_data.getInstances();
Map<Integer,ClusterTeaser> cluster_teasers = new HashMap<Integer,ClusterTeaser>();
//build the clusterer
clusterer.buildClusterer(dataset);
//set the cluster for each data object
for ( Feature f : feature_extractor_data.getFeatures() ) {
        //merge your list of attibutes (dataset) with the actual values (instance)
        Instance instance = f.getInstance();
        instance.setDataset(dataset);
        //do the actual clustering
        int cluster_number = clusterer.clusterInstance(instance);
        //create a ClusterTeaser for each cluster
        ClusterTeaser cluster_teaser = cluster_teasers.get(cluster_number);
        if ( cluster_teaser == null ) {
                Data data = databasemgt.getDataForProblem(f.getData_id(),”scholarships”);
                cluster_teaser = new ClusterTeaser(UUID.randomUUID(), data.getTitle());
                cluster_teasers.put(cluster_number, cluster_teaser);
        }
        databasemgt.setClusterTeaserForDataForProblem(cluster_teaser, f.getData_id(),”scholarships”);
}
```

The cluster teaser can be used later when displaying the data on your website.

Querying

One of the main purposes of an aggregator is to allow easier access to relevant data. Instead of using a key word search system we will use a more domain specific search using the category map. We will allow the user to choose a value for any of the categories Figure


Purpose: any
any
Study
Research
Level: any
any
Bachelor
Master
Phd
Source: any
any
www.scholarshipportal.com
www.scholarship-links.com
www.scholaprhipposition.com

Getting the values that go into the combo boxes is simple

```
//load the category map from database
CategoryMap cm = databasemgt.getCategoryMapForProblem(“scholarships”);
//find a category that is named purpose and then get all its children
List<<Category> purpose_sub_categories = cm.getCategory(“purpose”).getChildren();
//copy over the names of the subcategories
List<String> purpose_values = new ArrayList<String>(purpose_sub_categories.size()+1); For (
Category c : purpose_sub_categories ) purpose_values.add(c.getname
```

You can use this procedure for any category even if it is not directly connected to the root category like the Bachelor category in the example above.

After the user has chosen specific values for the categories you can run a query on the database to get the appropriate data, Let us suppose that the user choose purpose:study, source:any level:bachelor -> last-year students.

```
//construct the query
Map<String,String> query = new HashMap<String,String>(); query.put("level", "bachelor");
query.put("bachelor ", "last-year students");
query.put("purpose ", "study");
//fetch the respective data sorted by time of publication
List<Data> result = databasemgt.getDataForQueryForProblemSorted (query,“scholarships”,"time");
```

You can sort by any value stored in the database just make sure you are familiar with the Jackson naming policy, or inspect the database to see what your field is named. By default the query returns the data that satisfies all given user specifications. It is easy to run a query using $or, $any, $in if you want more custom queries.

Best of luck with your aggregator!