Integration of heterogeneous datasets

Arthur Imbert

September 2017

The internship took place in the Parietal team at INRIA (https://team.inria.fr/parietal/). My supervisors was Gaël Varoquaux.

1 Summary

2 Introduction

2.1 Heterogeneous data, producers and value

Nowadays, the production of data has considerably grown. Each organization exploits and issues data, using its own schema. Most of the time, these files are produced for specific purpose, taking into account some internal rules or constraints. That leads to a vast amount of data available online, often sharing the same file formats, but still deeply heterogeneous by their structure. Indeed, this heterogeneity complicates the integration of different files.

By the same time, the current enthusiasm around data science and its predictive models shows a high potential for crossing data.

2.2 A theoritical framework

Related problems

Our strategy

3 An application to Open Data

3.1 Open and heterogeneous data

In order to collect a vast amount of heterogeneous data we exploit the Open Data available online.

A French platform exists, gathering all the information needed to use the files issued by the French public organizations: data.gouv.fr. This website hosts and indexes multiple sources

3.2 How can we reuse the data

3.3 An academic angle

3.4 Internship's goal

4 From a bunch of heterogeneous data to a recommendation system

Before any analyze, we build our dataset. From the platform data.gouv.fr, we collect quantities of metadata. It generally includes an URL address where the file is stored. We then directly download as many files as possible. For each file downloaded, we try to clean it such as we can display its content in a tabular form.

At that point, we start the analysis. Firstly by preprocessing files' textual content. The result consists in a Term Frequency - Inverse Document Frequency (TFIDF) matrix. It allows us to weight word occurrences within our corpus according to their importance for each file. Secondly by building a vectorial topic space to represent every files (a file embedding) and based on the TFIDF matrix. This also includes an optimization step where we compare two topic extraction algorithms - Non-negative Matrix Factorization (NMF) and Singular Value Decomposition (SVD) - along with different configurations. Thirdly by looking for closest neighbors in this ”topic space”, in order to interpret distances within the embedding. Using a pretext task (the prediction of a reuse between pairs of files) and a metric learning algorithm, Least Square-residual Metric Learning (LSML), we also try to strengthen our file embedding and the interpretation of its clusters.

4.1 Build a database

The first step is to store a sufficient amount of dirty data in order to keep a statistically significant volume of data throughout all our process.

Collect metadata As described above, the platform data.gouv.fr offers us the opportunity to easily collect a numerous files from different French public organizations. A RESTful API exists in the website. It allows us to get, from a HTTP request, several metadata about more than 25000 pages. These include producer’s identity, a description of the page’s content, some related tags, the different files belonging to the page and, the most important, an URL for each file[1].

Some URL lead us to another website which host files, some directly launch the download of file. We only focus on the latters. It already represents around 60000 URL.

Download data We use DRAGO, an INRIA’s server, and a parallelized process to efficiently download as many files as possible. We use DRAGOSTORAGE, another server, to store the results of our downloads. By the end it represents around 350 Gb of data for more than 55000 files.

4.2 Clean and filter data

Among these files, there are several format: text, JSON, XML, spreadsheet, zip... Indeed, the second step consists in inferring the nature of files in order to filter them. The right process could then be applied in a attempt to clean files and reshape their content in a tabular form.

Filter the formats According to the supposed format of the file, we apply a different strategy to clean it. That involves to avoid the empty files and detect its MIME type (using *magic* library). If we meet a zip file, we repeat the procedure for every zipped items. We mainly focus on three types of documents:

- Text files, from which we try to detect a potential tabular form like in a CSV file.
- Spreadsheets, generally with an obvious tabular form that could make the inference of irregularities more difficult.
- Semi-structured files, generally containing geographical data like in a GEOJSON, easy to read, but difficult to flatten within a table.

CSV files For every file detected as text file, the very first operation executed is a test to determine if it could be a semi-structured file (JSON or GEOJSON). This test simply consists in reading the file, using the loaders from *json* and *geopandas* libraries. If an error occurs, the test fails and we process the file as a proper text file and a potential CSV or TSV file.

Several operations are executed in order to extract some data in a tabular form. Using *chardet* library, the encoding is inferred. If the latter is still undetermined, we read the file by default with an ”utf-8” encoding. If the file is long enough, a sample of rows is randomly extracted to speed up the next steps. Using *Sniffer* class from *csv* library, we try to detect a column delimiter. For example, in a CSV file, it’s a comma. This is a character which should occur the same number of times on each row. *Sniffer* doesn’t use a ”all or nothing” approach and then allows some irregularities in the data. Once we have a delimiter, the number of columns can be easily computed. In order to avoid rows with commentary or titles, we only keep those with the right number of columns (or delimiter occurrences). The last step, and the more difficult, before saving data within a *pandas* Dataframe, consists in inferring a pertinent header row. We test the consistency of data types for each row,

[1]A more detailed description of metadata is given in annexes

excepted the targeted one (which should be a string). If every columns nearly keep the same data type along file, we consider the targeted row as a potential header. The operation is repeated for the first rows until a proper header row is found. If nothing return, values by default are assigned as column names.

Excel files When a file is detected as a spreadsheet, it is read and edited using *xlrd* and *xlutils* libraries, respectively. The main difficulty is to skip the wrong rows (title, commentaries) and clearly delimited the pertinent data. Contrary to the previous cases, we can't infer this information based on a wrong number of delimiter: in a spreadsheet, the tabular format is already given. Rows have the same number of columns. But sometimes a row has too many empty cells or the latter are merged. So, before parsing the data through a dataframe and looking for the header, we need to preprocess the cells.

Cell by cell, we first replace characters that could mistake us while reading the document as a text file (for example "\n" and "\r"). Then, we fill merged cells by simply duplicating the information. We collect the potential name of its different sheets. For each sheet, we avoid the rows with an unexpected number of empty cells. The sheet is then parsed within a *pandas* Dataframe and saved as a text file. That allows us to apply the same operation executed with the CSV files in order to infer a header row. By the end, a single spreadsheet should return one cleaned table per sheet.

JSON, GEOJSON and XML files When a text file is detected, as previously explained, we try to read it as a JSON, then as a GEOJSON. The same process is then applied for both of them. Concerning the XML files, we just load it as a JSON file before, using the *xmljson* library.

All these files can be read as a tree. The idea is to recursively explore the tree structure of the file in order to find a pertinent branch to flatten in a dataframe. The pattern we are looking for is a list of dictionaries, where each item of the list is an observation and the dictionaries' keys are features. For performance reasons we just explore the first layers of the tree. By the end, we flatten the part of the tree with the largest number of items and a consistent number of features. This involves *json_normalize* function from *pandas* library. Generally, the header automatically inferred is pertinent.

Errors and extradata Through our cleaning process, numerous files are skipped because of their detected MIME type (PDF, images, etc.) or due to an error. For those which return a cleaned table, we also collect extradata. This are parts of the original file we don't include in the final table. While cleaning the text and Excel files, we still save separately rows with the wrong number of columns. It often represents a commentary, a source or the title. With the semi-structured files, we keep the part of the tree we don't flatten as extradata.

4.3 Build a file embedding

Once we have enough tables, with extradata for some of them, the third step consist in building a file embedding. That means we want to represent our files within a vectorial space in order to easily compute distances between files, and group them by cluster. We define a pretext task, predict a reuse between two files, to validate our process and determine if our embedding represent the corpus efficiently: two files reused together should be close in our space.

Text data preprocessing We mainly use the textual content of the files to determine if it worths to reused them together or not. Thus a preprocessing step is decisive. As every cleaned table is saved as a text file, the goal is to adequately count and weight words occurrences.

We distinguish three parts for a cleaned file: the content of table, its header row and some extradata (in a separate file). Thus, the first time-consuming task includes creating three different $N_{files} \times N_{words}$ occurrence count matrices with N_{files} the number of files and N_{words} the number of different words. This tokenization process is executed with *CountVectorizer* function of *sklearn* library. For each matrix, we normalize its rows, so that the sum of each row returns the same value. This penalizes large files with a lot of textual content. Normalizing the three matrices separately also emphasizes header and extradata, two parts with generally a lower but nonetheless quite pertinent textual content. Ultimately we mix the three matrices such that,

$$D = 0.5 * Content + 0.25 * Header + 0.25 * Extradata \tag{1}$$

The next task is a stemming algorithm from *nltk* library. During this process, we reduce inflected words to their root form (or stem). For example, such cuts could happen,

$$different \rightarrow differ$$
$$differ \rightarrow differ$$
$$differently \rightarrow differ$$

That makes our occurrence count matrix D more consistent with spelling mistakes, grammatical changes and conjugation. During the stemming process we keep an history of changes in order to determine, for each stem, the most frequent inflected word. We can then substitute the stem by its most frequent origin in order to have a list of existing words as a result. In the previous example, our matrix D could have a columns $different$ instead of $differ$. This is called a lemmatization process.

For the last preprocessing task we compute a TFIDF matrix from D. On the one hand, some words are frequent in every file (for example "the", "a", "is"). They usually appear to be less informative. Some of these words could even be regarded as noise and directly removed from the matrix D, using a predefined *stop words* list. On the other hand, rarer terms may often be more interesting and would deserve a bigger weight. In order to emphasize them, we use *TfidfTransformer* function from *sklearn* library. This transformation weights each word frequency by its inverse document frequency. We have

$$TFIDF(t,d) = TF(t,d) \times IDF(t) \tag{2}$$

with

$$IDF(t) = log(\frac{N_{files}}{1 + DF(t)}) \tag{3}$$

where term-frequency $TF(t,d)$ is the number of occurrences of term t in document d and document-frequency $DF(t)$ is the number of documents that contain term t. Each row is then normalize with a L_2 norm. Ultimately, the most discriminant words for a specific file (or row) have a higher floating score.

Topic modeling At that point, with TFIDF matrix we already have a representation of files within a very high-dimensional vectorial space. Topic modeling can be defined as a dimensionality reduction technique. We use it to reveal a hidden semantic structure in our corpus. We then expect that the topics produced are clusters of similar words. After dimensionality reduction, a file should be interpreted as a mixture of topics. Our aim is to compute a $N_{files} \times N_{topics}$ matrix W, with N_{topics} the number of topics (in our case, an integer to choose between 5 and 100). During the internship we used two different topic modeling algorithms: NMF and SVD. They basically factorize the TFIDF matrix into the product of W, which represents files within topic space, and a $N_{topics} \times N_{words}$ matrix H, which represents topics within word space.

Non-negative Matrix Factorization (NMF) NMF can be used if data matrix doesn't contain negative values. Indeed, it perfectly fit with textual data and more precisely occurrence count. It's also widely used in document clustering, recommender systems or computer vision.

Let V be our TFIDF matrix. The algorithm minimizing a distance between V and the matrix product $W.H$, with W and H containing non-negative elements. We use *sklearn*'s implementation where the distance minimized is a squared Frobenius norm, such as

$$d_{\text{Fro}}(A,B) = \frac{1}{2}||A - B||^2_{\text{Fro}} = \frac{1}{2}\sum_{i,j}(A_{ij} - B_{ij})^2 \tag{4}$$

The objective function to minimize is

$$\frac{1}{2}||V - WH||^2_{\text{Fro}} + \alpha\lambda||W||_1 + \alpha\lambda||H||_1 + \frac{1}{2}\alpha(1-\lambda)||W||^2_{\text{Fro}} + \frac{1}{2}\alpha(1-\lambda)||H||^2_{\text{Fro}} \tag{5}$$

with α a constant that multiplies the regularization terms, and so controls the sparsity of W and H, and λ a constant to balance L_1 and L_2 elementwise penalizations. Ultimately, we have

$$V \approx W.H \tag{6}$$

under the constraints $W \succeq 0$ and $H \succeq 0$.

Singular Value Decomposition (SVD) It approaches a spectral decomposition. We use *sklearn*'s implementation which is a truncated version (called Truncated SVD). We approximate V such as

$$V \approx V_k = U_k \Sigma_k P_k^\top \tag{7}$$

The diagonal entries σ_i of Σ are M's singular values. The columns of U and V are called the left-singular vectors and right-singular vectors of M, respectively. This implementation is called "truncated" because we only compute the k column vectors of U and row vectors of P corresponding to the k largest singular values Σ. The topic space is defined by $U_k \Sigma_k^\top$, with k features (or topics). In relation with our framework, we have

$$k = N_{topics}$$

$$W = U_k.\Sigma_k$$

$$H = P_k^\top$$

Finally, as this algorithm doesn't center the data before computing the singular value decomposition, it's still efficient with sparse matrix from *scipy* library. That perfectly fits with our TFIDF matrix.

Metric learning In practice, the previous unsupervised step is enough to build an efficient file embedding. For a certain N_{topics}, words clusters could form homogeneous topics which are even interpretable. Any Euclidean distance between two files would be synonymous of similarity. Two files close in W would have the same semantic structure, the same textual content. This is a first step toward reuse prediction. Indeed, it's pertinent to reuse similar files in order to increase data volume. Our goal is to go further in the analysis, to see if metric learning could yet improve the way we interpret distances. We want to base reuse prediction on more than just similarity. With a supervised metric learning algorithm, the idea is to learn from the existing reuses the potential of crossing two files weighted in two strictly different, but yet complementary, topics.

Let's consider two files represented in a topic space by two vectors x and y. A distance can be computed between them, using for example the quadratic norm

$$\begin{aligned} ||x-y||_2^2 &= \sum_{i=1}^{N_{topics}} (x_i - y_i)^2 \\ &= (x-y)^T(x-y) \end{aligned} \tag{8}$$

However, we can assign a different weight for each dimension, such that

$$\begin{aligned} ||x-y||_w^2 &= \sum_{i=1}^{N_{topics}} w_i \times (x_i - y_i)^2, \quad w_i \geq 0 \, \forall i \\ &= (x-y)^T diag(w)(x-y) \end{aligned} \tag{9}$$

With a more generally form we have

$$||x-y||^2 = (x-y)^T M (x-y), \quad M \succeq 0 \tag{10}$$

For $z = x - y$ and we define $\tilde{z} = M^{\frac{1}{2}} z$, so

$$\begin{aligned} ||\tilde{z}||_2^2 &= z^T (M^{\frac{1}{2}})^T M^{\frac{1}{2}} z \\ &= z^T M z \\ &= ||z||^2 \end{aligned} \tag{11}$$

The matrix M is called Mahalanobis matrix, such that $M = L^T L$ with $L = M^{\frac{1}{2}}$. The aim of a metric learning algorithm is to compute M and its factor L. Thus we can apply the latter as a transformation matrix to convert any multidimensional difference between two files $(x - y)$. Standard Euclidean distances can then be easily computed from the new transformed vector $L(x - y)$. It amounts to weight each feature (or dimension) and combination of features.

One big advantage of this method is the possibility to use a supervised framework. Thus, the Mahalanobis matrix is computed such that the transformation matrix L moves reused files closer within our learned metric space. To do that, we use a LSML algorithm from the *metric_learn* library.

Least Square-residual Metric Learning (LSML) This algorithm uses pairs' label to learned the Mahalanobis matrix M from constraints. The latters are defined as relative comparisons. For example, the distance between two reused files A and B is supposed to be smaller than the distance between two non reused files C and D. Contrary to a usual classification problem, we don't want to class a pair as reusable or not, as this objective seems meaningless. The results we are looking for are only relative.

Let $x_i \in \mathcal{R}^{N_{topics}}$, $\forall i$, we define a set of relative comparison as

$$\mathcal{C} = \{(x_a, x_b, x_c, x_d) : d(x_a, x_b) < d(x_c, x_d)\} \tag{12}$$

and the distance function as a M-transformed quadratic norm

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^T M (x_i - x_j)} \tag{13}$$

with M the Mahalanobis matrix we want to learn. We also define a Hinge loss

$$L(d(x_a, x_b) < d(x_c, x_d)) = H(d_M(x_a, x_b) - d_M(x_c, x_d)) \tag{14}$$

where a Hinge function is defined as

$$H(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x^2 & \text{if } x > 0 \end{cases} \tag{15}$$

The loss function is zero when the constraint $\mathcal{C}$ is satisfied, otherwise it's the difference of the new learned distance (the Euclidean distance in the L-transformed space). This loss penalizes any learned metric which moves reused pairs away within the vectorial space (and breaks the relative constraint). We also note that the shape of the Hinge function doesn't penalize the matrix M for moving reused pairs too close. Thus the closeness between reused files may be excessive sometimes.

Once a proper Mahalanobis matrix is learned from a set of pairs file (reused and not reused), we can directly transform the file embedding W with the transformation matrix L. It modifies the weigh of each file through the topics and so its vectorial representation within the topic space.

4.4 Find neighborhood

Once we have a file embedding where distances can be interpreted as a similarity score between files and potential reuse, the fourth step consists in implementing neighbor searches. We look for pertinent groups of files in order to validate the distances between actual reused files. Yet, it would allow us to find files which belong to a cluster of reused files, but no one reused yet.

We use the *NearestNeighbors* function from the *sklearn* library to implement neighbor searches. We don't specify the number of neighbors we want. It relies on the local density of points. It's a radius-based neighborhood. Every file within a fixed radius from the targeted file is counted as neighbor. We choose a basic L2 distance, since we already transform our topic space with our transformation matrix L. The critical parameter of this step is the radius. Too small, very few neighbors would be assigned to our target. Too big, the algorithm would class noisy files as neighbors. The choice of a pertinent radius is detailed in the next section, along with the optimization of several other parameters.

5 Results and validation

5.1 Metrics and experimental framework

5.2 Results

5.2.1 Presentation of the data

5.2.2 Metric comparison

5.2.3 Topics and Wordclouds

5.2.4 Reuse prediction

6 Discussion

6.1 Limitations

6.2 Problems

6.3 Future work

7 Conclusion

8 Annexes