TC4 - Recherche et Extraction de données
Moteurs de recherche
Rapport de projet

Jonathan CROUZET, Kevin PASINI,
Matthieu RÉ et Amal TARGHI

10 Novembre 2016

Abstract

Nous allons présenter dans le présent rapport les moteurs de recherche dont les caractéristiques sont présentées dans l'énoncé du projet et seront rappelées ensuite. Nous nous intéresserons en particulier au moteur de recherche temporel (sujet D) dans la seconde partie.

1 Introduction

Les moteurs de recherche sont une partie fondamentale de la navigation internet, et si de nombreuses méthodes sont généralisées pour leur fonctionnement, la recherche est toujours nécessaire devant l'augmentation des volumes de données, et de la diversité des données.

Dans ce rapport, nous nous intéressons dans un premier temps à un moteur de recherche de texte "classique", qui sur la donnée d'une requête devra rendre par ordre de pertinence une liste d'articles, provenant d'une base de données d'articles journalistiques répartis sur plusieurs années.

Dans la seconde partie, nous proposerons une version "temporelle" de ce moteur de recherche : sur la donnée d'une requête et d'un intervalle de temps, nous cherchons à afficher une frise chronologique représentant les principaux articles émanents d'évènements différents sur cet intervalle.

2 Partie 1 : Moteur de recherche

Une première problématique est soulevée pour le développement d'un moteur de recherche: comment connaître l'emplacement de l'information?

Afin de répondre à ce problème, le moteur de recherche doit traiter l'ensemble de l'information disponible (processus d'indexation). Ce processus localise l'information. En effet c'est le même principe de l'index d'un livre. Le moteur de recherche permet d'interroger l'index quand l'utilisateur envoie sa requête. En général,

en lisant un document, l'individu est capable de savoir si ça correspond à ses attentes, également évaluer sa pertinence. Le moteur de recherche ne pas faire ceci d'une manière intuitive, mais par une mesure de la similarité entre chaque document et la requête envoyée.

Suivant cette mesure le moteur est capable de retourner les résultats qui se rapprochent du besoin de l'utilisateur (ou en tout cas de la perception humaine).

La suite va être répartie en deux parties. La première décrit les étapes du développement d'un moteur d'une manière générale et la deuxième explique notre travail.

2.1 Etapes

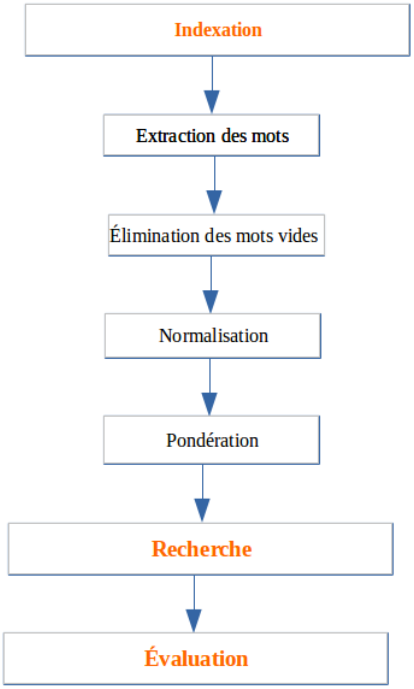


Figure 1: Les étapes de développement d'un moteur de recherche

2.1.1 Indexation

Cette étape consiste à identifier pour chaque document les termes importants, puis à exploiter ces termes comme index pour accéder rapidement aux documents. L'un des objectifs majeurs de l'indexation est de retrouver les documents

contenant les termes de la requête. Cette étape est déroulée hors ligne, il n'est pas nécessaire de l'effectuer plus qu'une seule fois si le corpus n'est pas modifié.

Extraction des mots Cette phase consiste à extraire/segmenter le texte du document en mots. Cette étape est appliquée au corpus et à la requête. Dans cette étape on détermine la façon de traiter des caractères spéciaux tels que les traits d'union et les ponctuations. Par exemple: Devrions nous traiter Charlie la même façon que charlie ? Est ce que "en ligne" représente deux mots ou un seul mot ? Est ce que l'apostrophe dans O'Jason doit être traitée de la même façon que celle l'incendie ? L'observation dans certaines langues, la tokenization devient encore plus complexe, par exemple dans la langue coréenne n'a pas les mêmes séparateurs que la langue française également la langue arabe ou l'hébreu s'écrivent de droite... Donc, une analyse lexicale est nécessaire pour identifier les "tokens" en reconnaissant tout ce qui est des séparateurs, des caractères spéciaux, des chiffres, les ponctuations etc...

Mots vides Les textes contiennent souvent des termes non significatifs (pronoms personnels, prépositions, alphabets). Ce traitement a pour but d'enlever les dépassant d'un certain nombre d'occurrences dans la collection dans le but de réduire la taille de l'index.

Normalisation L'une des méthodes de normalisation est la racinisation il s'agit d'un traitement permettant de regrouper les variantes d'un mot. Par exemple biologie, biologiste, biologique par: biologie).

Pondération La pondération est une phase primordiale puisqu'elle traduit l'importance de chaque terme dans chaque document. Estimer l'importance d'un terme est une tâche difficile. Par exemple un mot qui apparaît dans tous les documents ne sera pas utile pour l'index par contre un mot rare à une grande valeur. Pour calculer les poids en applique la formule suivante:

$$TfIDf = Tf.IDf \tag{1}$$

Avec Tf est le nombre d'occurences de chaque terme dans le document et IDf L'inverse de la fréquence des documents qui contiennent le terme.

2.1.2 Recherche

Un modèle de recherche d'informations représente les relations entre les documents et les requêtes, et définit une stratégie d'ordonnancement des documents. Il existe plusieurs modèles de recherche: modèles booléens, modèles probabilistes modèles vectoriels. Dans notre cas nous avons utilisé ce dernier pour calculer une mesure de similarité entre la requête et les documents qui sont représentés par des vecteurs. Nous avons utilisé la mesure cosinus.

2.1.3 Évaluation

L'évaluation des systèmes de REI est nécessaire pour mesurer la performance et l'efficacité. Un système est dit performant et efficace s'il est capable à réponde aux besoins de l'utilisateur, on peut savoir ceci grâce aux pages visitées, les notes des utilisateurs etc... également il doit être rapide et facile à utiliser.

2.2 Simulation

2.2.1 Introduction

Le but est de développer un moteur de recherche capable d'indexer environ 10000 documents du corpus publiés entre janvier 2015 et avril 2015 et retourner les résultats les plus pertinents par rapport à la requête envoyée tout en respectant les contraintes: le fichier ne doit pas consommer plus que 1Go de la RAM et la taille d'index doit être inférieur à 60 % de la taille du corpus.

2.2.2 Indexation

Nous avons construit deux index inversés. Chaque terme est associé à la liste des documents auquel il appartient nous avons également fusionné les poids associés à chaque document dans l'index. Le premier index est non normalisé le deuxième est normalisé en utilisant le stemming.

Taille de l'index Pour obtenir un index moins volumineux nous avons pensé à deux méthodes. La première consiste à changer les noms des fichiers (Donner un identifiant à chaque fichier). La deuxième consiste à tronquer les poids à deux chiffres après la virgule par exemple: 0.12356856548 va devenir 0.12.

Pour la première méthode nous sommes passés par 4 étapes:
1. Numéroter les fichiers: à la place d'avoir 20150310_d9440e54e6f72470f0a99e9337b1dca5.txt on aura un numéro. Ceci n'était pas efficace car nous n'avons pu réduire que 18% pour l'index non normalisé et 26% pour l'index normalisé.
2. Encodage: changer la base du numéro de nos fichiers à la base 36. Ceci n'a pas beacoup servi car nous avons éliminé juste 25% pour l'index non normalisé et 33% pour l'index normalisé.
3. Encodage: A la place de la base 36 nous avons pensé à la base 62. Ceci n'a pas été suffisant car nous avons optimisé juste 28% pour le premier corpus et 36%.
4. Suppression de quelques caractères tel que [ ] generés par les keySet. Nous avons atteint notre objectif ¡ 52% pour l'index normalisé et ¡ 60% pour le deuxième index.
Fixer un seuil sur l'Idf (lorsqu'il est trop petit, et que le terme est présent dans presque tous les documents) permet de faire baisser encore la taille des index.

Calcul des poids: Pour calculer les poids nous avons utilisé TfIdf (voir section indexation). Cette méthode permet de donner plus d'importance aux termes rares qui sont discrimants (voir section indexation).

2.2.3 Recherche d'après une requête

Le projet d'indexation 1 demandait d'adapter le principe de calcul de similarité avec le fichier d'index inverse avec poids.

Le principe de la recherche par requête consiste à considérer la requête comme un texte du corpus avec ses propres poids, et de calculer la similarité de cette requête avec tous les autres textes du corpus à l'aide de la formule de similarité.

Il faut arriver à extraire de l'index, en parallèle pour chaque texte, les informations de poids pour tous les mots. Cependant, Il n'est pas possible de calculer une similarité partielle portant sur un certain nombre de mots, nous pouvons en revanche calculer de manière indépendante les sommes partielles qui constituent la formule, puis calculer la similarité en recombinant les sommes 1, 2, 3.

$$sim(d_j, d_k) = \frac{\vec{d_j} \cdot \vec{d_k}}{|\vec{d_j}|.|\vec{d_k}|} = \frac{\overset{①}{\sum_{i=1}^{n} w_{i,j}.w_{i,k}}}{\underset{②}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2}}\underset{③}{\sqrt{\sum_{i=1}^{n} w_{i,k}^2}}}$$

i : *requête.*
k : *documents du corpus.*
i : *mots de l'index.*

Nous disposons d'un index sous la forme d'un TreeMap(String1, TreeMap(String2, Double)). Avec String1: mot de l'index i, String: id des docs k contenant i, Double: $w_{i,k}$ pour tous les k

Nous avons choisi de stocker les résultats partiels permettant le calcul de la similarité dans un **HashMap(String, Pair)** avec String : id des docs k, Pair: $(\Sigma 1, \Sigma 3)$ somme partielle de k. Initialement nous avons $(\Sigma 10, \Sigma 30)$. La $\Sigma 2$ ne dépend que de la requête, elle est donc la même pour tous les textes du corpus, et peu donc être stockée dans une variable appropriée.

La requête est transformée en un **HashMap(String, Integer)** qui contient les mots et leurs occurrences.

La combinaison de ces trois types de données va nous permettre de ne considérer que les poids significatifs présents dans l'index. Ainsi, pour un mot de l'index, seules les similarités des textes contenant ce mot seront actualisées.

L'actualisation des sommes partielles contenues dans le HashMap se réalise ainsi.

- Pour un mot i, nous disposons donc d'un TreeMap T1 contenant l'ensemble des documents qui le contienne et des poids liées.

    1. Calcul de $w_{i,j}$ avec la taille du TreeMap, et l'occurrence de i dans la requête.

2. Pour tous les documents du Treemap :
   (a) Récupération des $w_{i,k}$ correspond dans le TreeMap
   (b) Actualisation $\Sigma 1 = \Sigma 1 + w_i j * w_i k$
   (c) Actualisation $\Sigma 3 = \Sigma 3 + w_{i,k}^2$
3. Actualisation $\Sigma 2 = \Sigma 2 + w_{i,j}^2$

- Ce procédé est repété pour tous les mots de l'index.

Nous obtenons après le parcours de l'index :

1. une variable contenant $\Sigma 2$ et d'un HashMap contenant pour tous
2. les documents k les valeurs $(\Sigma 1, \Sigma 3)_k$)

Ainsi on a :

$$sim(d_j, d_k) = \frac{\Sigma 1_k}{\sqrt{\Sigma 2} * \sqrt{\Sigma 3_k}}$$

2.2.4 Evaluation

Pour évaluer la pertinence de notre système nous avons noté les 20 premièrs documents renvoyés par le moteur pour chaque requête. La note varie entre 0 et 10:

Pertinence

- Charlie Hebdo
- volcan
- playoffs NBA
- accidents d'avion
- laïcité
- élections législatives
- Sepp Blatter
- budget de la défense
- Galaxy S6
- Kurdes

Par exemple pour le cas de **Galaxy S6** qui se compose de deux termes:
Index normalisé (Stemming): Nous avons évalué le premier résultat retourné par l'index Stemmer: 6.0 car le sujet parle des Samsung galaxy cependant Galaxy s5 même chose pour le 2ème document retourné il parle de Galaxy Note4.Par contre pour le 3ème résultat la note était 9 car le sujet correspond à la requête il décrit Galaxy 6 et sa place sur le marché des smartphones.
Index non normalisé: La note du premier résultat est 8.5 car le sujet parle de la sortie de Samsung Galaxy 6 dans le marché ainsi ils comparent les ventes par rapport à la gamme Iphone, mais il parle également des télévision Samsung ce qui ne nous intéresse pas.
Le résulat est logique car avec le Stemming on perd le mot 6 par exemple le premier résultat renvoyé peut être lié au nombre d'occurences du mot galaxy au document qui est élevé.
Par exemple pour le cas de **Kurde** il se compose d'un seul terme.
Index normalisé (Stemming): la note du premier document retourné est 9 car le sujet est intéressant également il parle de l'hitoire des kurdes en Turquie.
Index non normalisé: La note du premier résultat est 8.5 le sujet est intéressant un sujet d'actualité il parle de la situation des kurdes après la guerre en Syrie. ce qui est très intéressant en 2015 même chose pour le 2ème document la note est 8.5 car il parle des kurdes en Iraq après les attaques. La note n'est pas 10/10 car il parle pas de kurdes de manière explicite par exemple: leurs origines, leur histoire ect...

Nous avons fait la même chose pour les 20 premiers résultats retournés pour chaque requête pour chaque index afin de pourvoir évaluer notre système. On a déduit que pour les mots composés une indexation sans normalisation donne des résultats meilleurs.

Performances : **Ram** et **temps d'exécution**

Une indexation sans normalisation occupe plus d'espace dans la mémoire par rapport à une indexation normalisée (Stemming) également pour la recherche. Voir figure 2 3 4 et 5.

On remarque également que temps de recherche dans un index normalisé est inférieur au temps de calcul dans un moteur de recherche non normalisé. Voir figure 6 et 7.

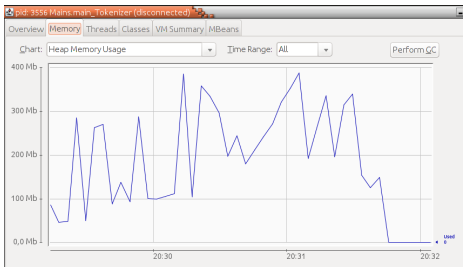


Figure 2: mémoire consomée lors d'une indexation sans normalisation

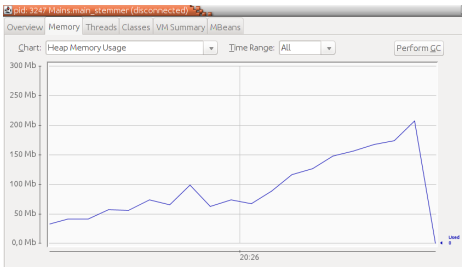


Figure 3: mémoire consomée lors d'une indexation avec du stemming

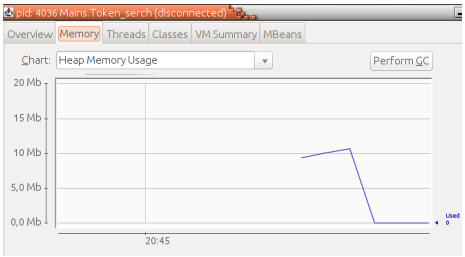


Figure 4: mémoire consomée lors d'une recherche dans un index non normlisée

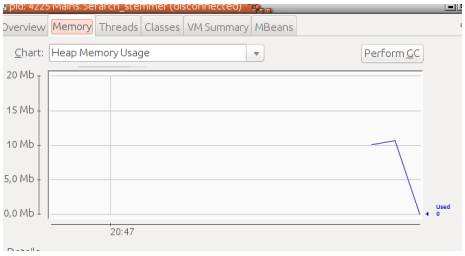


Figure 5: mémoire consomée lors d'une recherche dans un index non normlisée

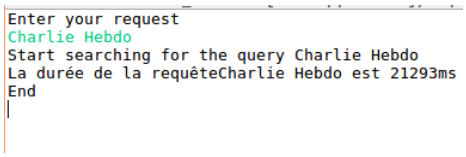


Figure 6: durée de recherhe dans un index non nomalisée

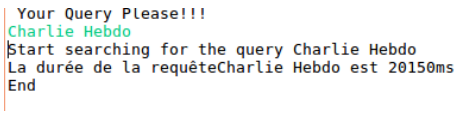


Figure 7: durée de recherhe dans un index nomalisée

2.2.5 Conclusion

Les contraintes exigées ont été respectées : nous n'avons pas dépassé 1Go en RAM et la taille des deux index est inférieure à 60% de la taille du corpus. On pourrait éventuellement diminuer encore cette taille en filtrant les mots avec un faible Idf mais cela pourrait finir par nuire à la qualité des résultats. On remarque que les résultats sont plutôt bons, même avec les requêtes de plusieurs mots alors que l'ordre des mots n'entre pas en compte dans l'indexation.

3 Partie 2 : Moteur de recherche temporel

Dans cette partie nous allons présenter les principes de fonctionnement de notre moteur de recherche temporel, en expliquant les choix, les alternatives et les contraintes auxquels nous sommes confrontés et que nous proposons.

Dans un premier temps nous évoquerons l'indexation dans Elasticsearch, ensuite nous détaillerons les traitements des recherches et nos méthodes de détection d'évènements, pour finir par l'affichage d'une frise de résultats.

3.1 Processus d'indexation et requêtes à Elasticsearch

Comme il nous a été nécessaire de créer dans le premier projet, c'est à partir d'un fichier d'index que les requêtes seront traitées dans ce projet. Cependant, pour gérer ce fichier d'index, les calculs de pertinences, et la gestion de transformations et de calculs tels que le comptage d'éléments ou l'agrégation de résultats, nous avons choisi d'utiliser le moteur d'indexation libre Elasticsearch.

3.1.1 Présentation d'Elasticsearch et de Logstash

Nous allons nous lancer dans ce qui se veut une courte introduction à ce qu'est Elasticsearch. Nous ne reviendrons que sur les informations qui nous ont été nécessaires pour la compréhension et la réalisation du projet.

Elasticsearch est un moteur d'indexation basé sur la librairie Lucene d'Apache, et développé en Java. Il est principalement conçu pour un usage de traitement de gros volumes de données et est prisé par les entreprises notamment pour son côté "scalable", i.e dont on peut facilement étendre l'envergure pour l'adapter au système étudié.

Cette scalabilité est notamment due au fait que l'on peut lancer des instances d'Elasticsearch sur différents serveurs, qui communiquent entre elles via un même cluster, dans lequel les noeuds (et de facto, les instances d'Elasticsearch) se répartissent de façon configurable les index, et les shards (nous y reviendrons juste après). Pour assurer de meilleures performances de recherche, ces sharps peuvent être dupliqués sur différents serveurs, de façon à assurer leur disponibilité et leur accessibilité en cas de chute de serveur.

Ici, nous n'utilisons qu'un seul serveur, et ainsi l'index ne se divise qu'en 5 shards répartis sur un seul serveur local.

Les documents indexés dans Elasticsearch se divisent en index, et chaque index se divise en plusieurs "shards" (instances de Lucene). Les requêtes que l'on fait à Elasticsearch se font principalement à l'aide d'une API HTTP, avec laquelle on intéragit par un langage de requête très bien expliqué dans la documentation officielle [1] d'Elastic, la société à l'origine d'Elasticsearch.

On peut choisir les propriétés de chaque champ indexé par Elasticsearch via le "mapping". Cela va de son type, à la possibilité qu'il soit analysé ou non, si oui par un analyseur français ou d'une autre langue, dans le but d'accélérer et d'optimiser certains types de requêtes.

[1] https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html

Reste maintenant à rentrer tous les article dans un index d'Elasticsearch *spliine*. Pour cela, on utilise Logstash, une solution qui permet de traiter des flux, ou des fichiers, de façon à les transmettre vers d'autres instances comme par exemple Elasticsearch.

Pour cela, Logstash se décompose en 3 "plugins" de traitement de message : le plugin de gestion d'entrée, le plugin de filtre, et le plugin de sortie. Chaque plugin est décrit dans l'annexe 1, tout comme toute l'installation et la configuration d'Elasticsearch et Logstash.

En ce qui concerne le plugin d'entrée, nous avons rencontré le problème que l'indexation et le traitement de messages (donc : des articles) n'était pas simple lorsqu'il contient des retours à la ligne. Nous avons donc deux choix : soit d'indexer un document ligne par ligne, soit de transformer les articles de façon à ce qu'il n'y ait plus de retour à la ligne. La première solution nous imposerait un travail supplémentaire par rapport au calcul de pertinence d'Elasticsearch, qui complexifierait trop pour une faible plus-value.

Ainsi, au lieu d'indexer directement les articles épurés du corpus, on les transforme en remplaçant les '$\backslash n$' en '$\backslash t$'. Ce travail nécessite de créer de nouveaux fichiers, aussi nous en profitons pour créer des fichiers *.csv* pour isoler la date et l'id de l'article, de façon à accélérer pour la suite le temps de traitement de Logstash, et notamment de son plugin filtre.

Les articles indexés sont chacun pourvus notamment d'un champ *date_art* contenant la date de l'article, *id_art* l'id de l'article dans le corpus et *article* le contenu de l'article. Une fois l'indexation réalisée, l'index *spliine*, qui mesure 5.7 *Gb*, est prêt à recevoir ses requêtes.

3.1.2 Requêtes à Elasticsearch et calcul de pertinence

Comme nous l'avons dit, les appels à Elasticsearch se font principalement à l'aide de l'API HTTP, disponible par défaut sur le port 9200 du serveur. Pour faciliter et sécuriser ces appels à Elasticsearch, Elastic propose une bibliothèque Python $elasticsearch - py$, qui facilite ces requêtes.

Le système de requêtes d'Elasticsearch est complet et complexe. Pour réaliser les deux types de requêtes dont on a besoin, il y a plusieurs concepts qu'il convient d'expliquer.

Les requetes à Elasticsearch peuvent être imbriquées. Cette propriété permet par exemple d'ajouter à la recherche d'un terme la contrainte d'une intervalle de date. Dans notre cas, on met en application cela à l'aide d'un plugin *bool* combiné avec un *query_string* et d'un *range*.

Il existe de nombreux types de requêtes appliquées à des chaînes de caractères. Le type choisi permet de faire une requête basée sur la pertinence des documents par rapport à la requête, qui utilise la fonction de scoring optimisé d'Elasticsearch. Il n'est pas nécessaire d'utiliser plus de traitement pour notre application. Une query permet par exemple de minimiser les termes les moins pertinents de la requête, mais cela revient à accentuer les poids donnés par une sorte de tf.idf, qui n'influe que peu sur les résultats pour un temps de traitement plus élevé.

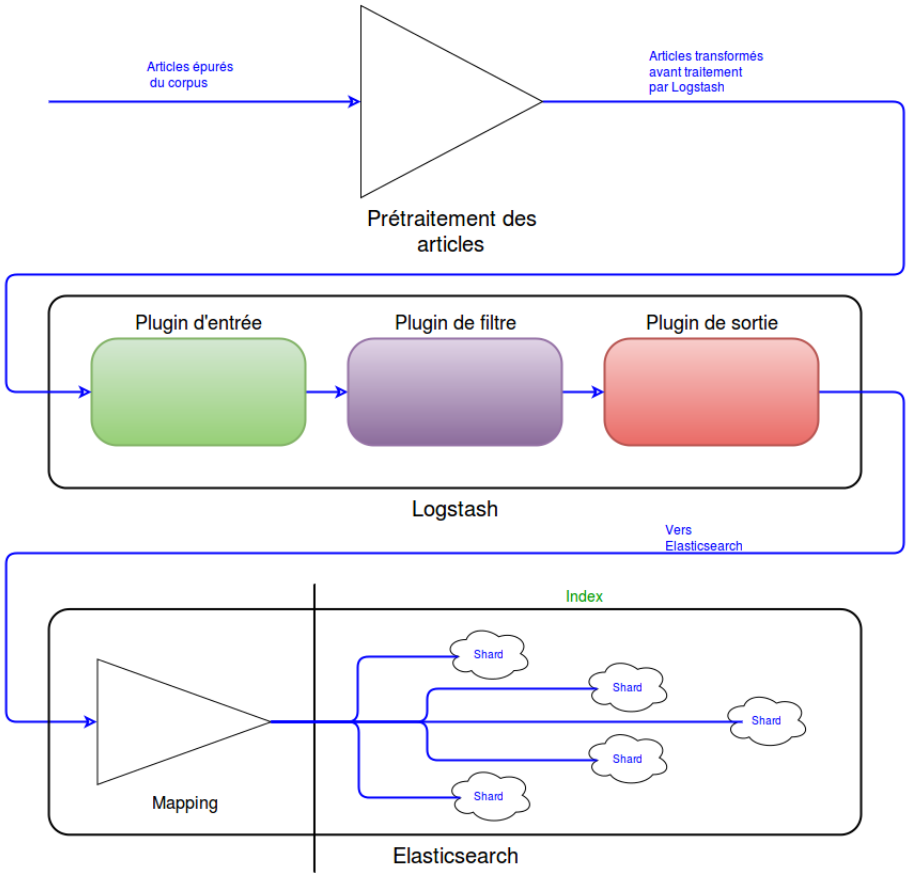


Figure 8: Chemin de la phase d'indexation

Dans son calcul de la pertinence d'un terme par rapport à une requête, Elasticsearch utilise les deux grandes notions vues en cours, le tf.idf et les modèles vecteurs pour le calcul d'une distance cosinus. A cela s'ajoute quelques autres affinités comme la longueur de la chaine de caractère considérée dans le document. [2]

Pour la recherche des pics d'évènements, on a besoin de faire sortir un histogramme du nombre d'article par jour rendu par la requête. Pour cela, on utilise les agrégats, et en particulier les agrégats de type buckets [3]. Ils permettent, avec un peu de configuration supplémentaire de la requête, de faire ressortir le compte d'articles par jour sur la plage temporelle indiquée pour une certaine requête.

[2] https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.html

[3] https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-datehistogram-aggregation.html

```
GET /spliine/article/_search
{
  "query": {
    "bool": {
      "must": {
        "range": {
          "date_art": {
            "gte": "'+debut+'",
            "lte": "'+fin+'"
          }
        }
      },
      "filter": {
        "query_string": {
          "query": "'+query+'",
          "default_operator": "AND"
        }
      }
    }
  },
  "size": 0,
  "aggregations": {
    "date_hist": {
      "date_histogram": {
        "field": "date_art",
        "interval": "day",
        "format": "yyyy-MM-dd"
      }
    }
  }
}
```

Figure 9: Forme de la requête appliquée pour obtenir les histogrammes de compte d'évènements

3.2 Fonctionnement du script de requête

Après réfléxion, nous avons choisi de coder en python. En effet il était bien plus adapter de réaliser la détection de pics nécéssitant des outils mathématiques et module d'affichage de courbe. De plus une API fonctionnelle nous permettait de communiquer facilement avec Elasticsearch.

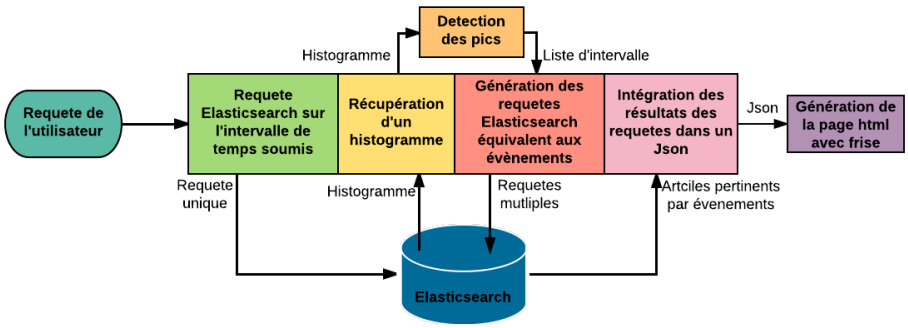
Il est possible de décomposer le script en 4 sections :

1. Le main chargé de les communications entres les autres sections.
2. Elasticsearch qui s'occupe la recherche dans l'index.
3. Une section en python chargée de la détéction des évènements
4. un section en python chargée de la génération de l'affichage.

Le déroulement d'une requête est la suivante :
De la requête utilisateur, le programme génère une requête Elasticsearch soumise à l'aide de l'API. Elle récupère l'histogramme de comptage des articles pertinents par jour, l'adapte et l'envoie à la fonction de détection des pics.

Cette fonction renvoie une liste d'intervalles correspondant à la durée des évènements détectés. Avec cette liste, le programme génère des requetes cherchant les articles les plus pertinents sur la durée de chaque évènement.

Du résultat de ces requêtes, il intègre les informations souhaitées dans l'affichage à l'intérieur d'un json qu'il envoie à la fonction d'affichage. La fonction d'affichage va génerer à l'aide du squelette de la page d'affichage la frise correspondant à la requete utilisateur en y intégrant le json comportant les informations.



3.3 Analyse des pics d'évènements

Pour notre moteur de recherche temporelle, nous voulons pouvoir afficher sur une timeline des résultats répartis le plus uniformément possible sur la plage temporelle considérée. Cependant nous ne voulons pas afficher de résultat pour une période vide où il ne s'est rien passé.

La première étape de la recherche consiste donc à trouver les périodes pour lesquelles *il s'est passé quelque chose* pour une requête donnée. Nous identifions ces periodes par l'augmentation anormal du nombre d'articles pertinents retournés pour la requête.

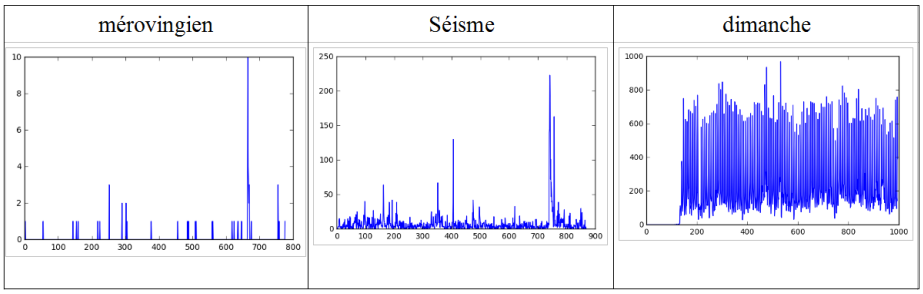
3.3.1 Méthode de détection par seuil fixe

Dans le contexte de recherche de pertinence avec la temporalité, une des problématiques les plus complexes est la détection d'événements.

Nous sommes parties du concept d'histogrammes comptabilisant les articles pertinents par jours. Ainsi, les événements correspond à des pics, l'amplitude du pics correspond à l'importance de éventements et sa variance correspond à la durée. L'extraction de pics d'un histogramme est une question complexe, particulièrement avec des données réelles fortement bruitées.

Dans le cadre de notre projet avec Elasticsearch, il est possible de réaliser une requête qui renvoie un histogramme, avec la particularité qu'il centre histogramme entre le premier et le dernier article pertinent.

Voici quelques exemple histogramme réel:



On constate de très grandes différences entre ces différents histogrammes. Ils n'ont ni la même échelle de temps, ni les mêmes formes. On distingue énormément de bruit, une tendance cyclique, et un palier anormal à 0 avec la requête dimanche, là ou il n'y a aucun bruit et très peu de d'articles pertinents pour la requête mérovingien.

La problématique est d'arriver à détecter des évènements parmi ces histogrammes. Le première problème vient de la définition d'évènements qui ne peuvent pas être définis clairement. Dans le cas de dimanche, on peux considérer chaque dimanche comme un évènement, où uniquement les dimanches "exceptionnels", ce qui reste ambigue. Le deuxième problème vient de la différence extrême de profil que peuvent avoir les histogrammes de documents pertinents.

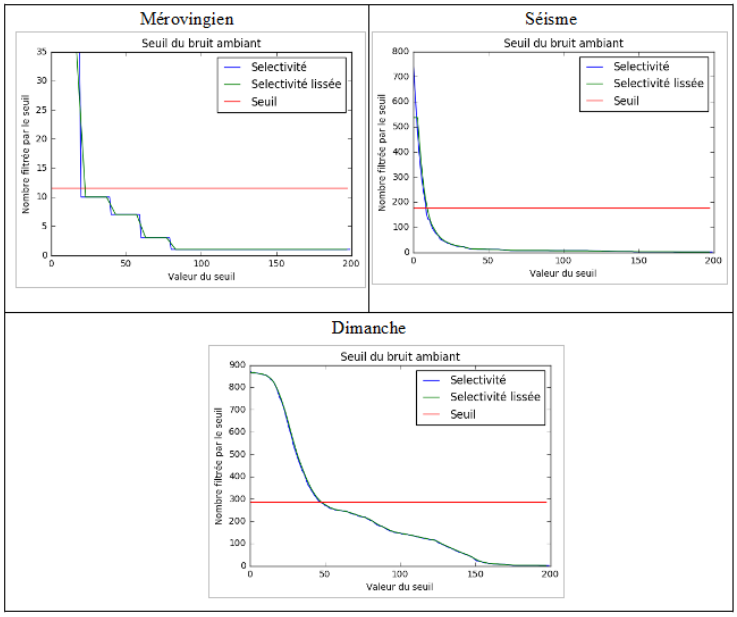
À partir de ce postulat, nous avons décidé d'essayer d'attaquer le problème sous deux aspects différents: un aspect naïf avec une méthodes intuitives, et un aspect plus techniques.

Méthode naïve par seuils Notre idée naïve est de déterminer un seuil, à partir duquel on considère que les pics sont des évènements à part entière. Avec ce concept, il semble simple d'arriver à filtrer ce que l'on considère comme évènement du reste.

Le première réflexe fut de lisser ces histogrammes chaotiques pour simplifier la recherche.

Le première problème est de trouver la valeur de ce seuil qui dépend irrémédiablement du profil de l'histogramme: dans un cas fortement bruité comme "dimanche", cette valeur va être forte, de l'ordre de plusieurs centaines ; au contraire, dans un cas comme mérovingien, cette valeur va être très faible, de l'ordre de l'unité.

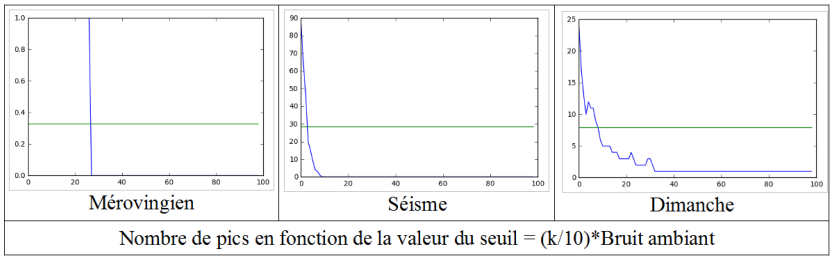
Pour cela, nous avons eu l'idée de observer la chute du nombre points de l'histogramme lorsque l'on fixe un seuil minimum: on obtient des courbes décroissantes qui contiennent de l'information sur le bruit ambiant de l'histogramme.



Pour en extraire un valeur concrète nous avons essayer plusieurs techniques à bases de dérivés, de coefficients directeurs mais au vu du comportement chaotiques de l'histogramme originel, ces méthodes se sont montrées peu concluantes. Nous avons fini par simplement choisir un seuil fixe: on considère qu'une certaine proportion de ces points appartient au bruit ambiant (66 %), et nous avons relevé la valeur du bruit qui y correspondait (l'intersection entre la courbe et la droite : cela nous fournissait ainsi un estimation du bruit ambiant.

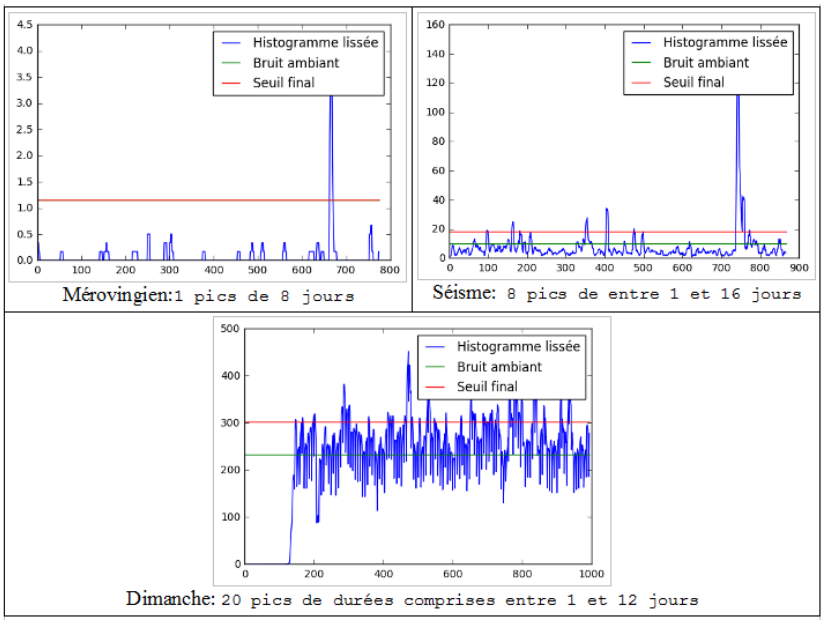
Nous avons ensuite cessé de considérer les points de l'histogramme pour nous consacrer sur les pics avec la définition suivante : un pic est un intervalle de points supérieur (en tout point) à un seuil. Avec l'estimation du bruit ambiant nous possédions un seuil initial avec lequel nous pouvions travailler.

Ce seuil n'est pas forcément adapté à la détection optimale de pics mais il nous permet d'avoir un ordre de grandeur, ainsi nous testons plusieurs seuils à $k * Bruitambiant$, avec $k \in [1; 10]$ pour chaque seuil nous comptons le nombre d'événements distincts. Sur la même base de réflexion que précédemment nous pouvions donc tracer la chute du nombre de pics en fonction du seuil.



Avec la même réflexion que précédemment nous avons considérer que seul un % des pics supérieurs au bruit ambiant était intéressant, nous avons finis par fixer ce seuil à 33 %.

Une fois les coefficients k déterminés il est possible de déterminer les pics pertinents, et ainsi d'extraire la date de début, et la date de fin de l'événement correspondant aux pics.



Cette méthode bien que naïve semble réussir à fournir un résultat intéressant pour les trois profils extrêmes.

De plus nous avons fixé deux paramètres correspondant à l'estimation des points bruités (66 %) et l'estimation des pics significatifs (33 %). Il est tout à fait possible de modifier ces valeurs, de les déterminer en fonction du profil de l'histogramme, voir de les adapter en fonction de l'exigence de l'utilisateur (nombre d'évènements, degré de rareté des événements)

3.3.2 Méthode de détection par seuil variable

Le nombre moyen d'articles récupérés pour une requête peut varier au cours du temps. Ainsi il peut être intéressant de faire varier le niveau du seuil en fonction de ces évolutions.

L'idée est de regarder sur une fenêtre passé la moyenne et l'écart type de la série temporelle. La taille de la fenêtre, lag, est le premier paramètre de l'algorithme. On en déduit un intervalle :$[avg - thresh * std, avg + thresh * std]$, où $thresh$ est le second paramètre. Si on sort de cet intervalle on considère qu'on a un événement.

Cependant si on a un événement important, au prochain point la moyenne et l'écart type vont beaucoup changer et fausser la methode. C'est pourquoi on calcul en même temps une série temporelle filtrée. Cette série est :

$$x_{filtered}(i) = \begin{cases} x(i) \text{ si } i \text{ n'est pas un événement} \\ x_{filtered}(i-1) * (1 - influence) + x(i) * influence \text{ sinon} \end{cases}$$

On calcule la moyenne et l'écart type sur cette série. Généralement on veut prendre $influence$, le troisième paramètre, très petit.

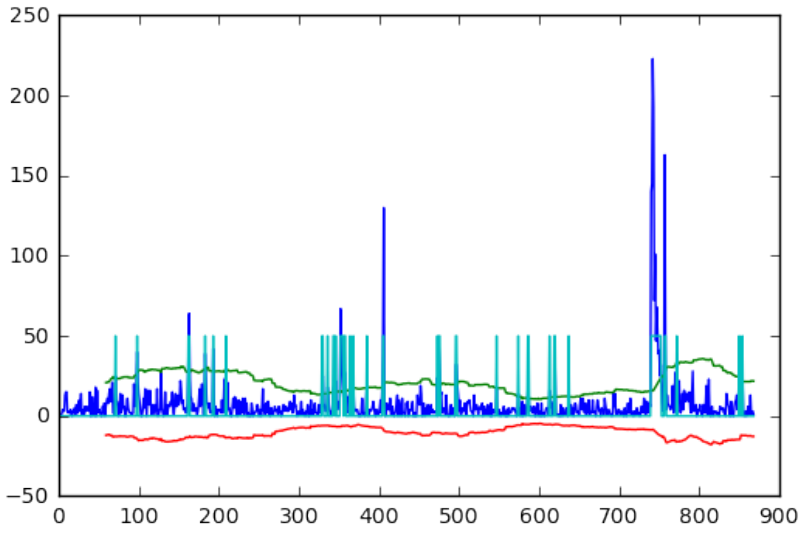


Figure 10: Série : bleu, seuil haut : vert, seuil bas : rouge, évènements : bleu clair

3.3.3 Améliorations : décompositon de la série temporelle

Il est possible que dans les résultats de certaines requêtes apparaissent une certaine périodicité à cause d'un événement qui a lieu tous les n pas de temps.

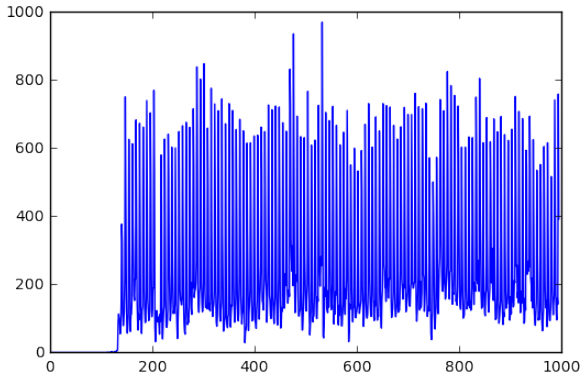


Figure 11: Exemple de série périodique : `dimanche`

Pour ces requêtes, on souhaiterait ne pas récupérer toutes les instances de l'évènement mais seulement les plus importantes. Une solution est de décomposer la série temporelle en une composante tendancielle, périodique et un bruit.

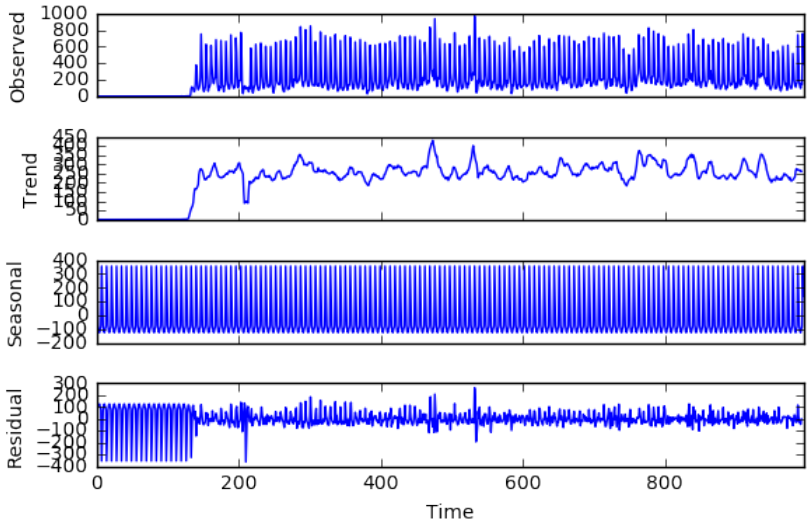


Figure 12: Décomposition

On remarque que les périodes les plus importantes correspondent aux périodes avec un bruit plus fort. Néanmoins travailler sur le bruit ne s'est pas avéré

fructueux. Utiliser les algorithmes vus précédement sur la tendance de la série donne de meilleurs résultats.

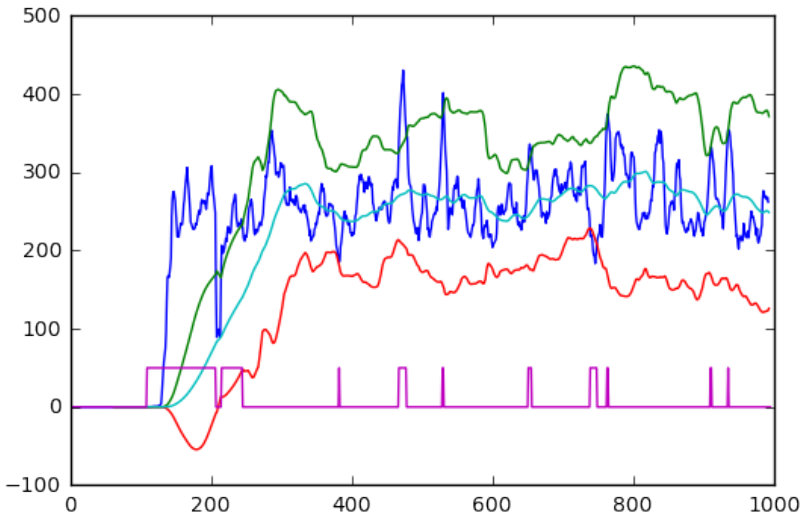


Figure 13: Résultats sur la composante tendancielle

Cette méthode a un gros défaut : il faut avoir un moyen pour deviner la périodicité de la série. Ce paramètre dépend des requêtes et est même inexistant pour certaines.

3.3.4 Limites de ces méthodes

De nombreux paramètres à optimiser et la plupart dépendent des requêtes.

Avec ces méthodes nous sommes incapables de détecter des événements qui se recouvrent les uns les autres. Par exemple, si on recherche `séisme` et que deux séismes ont eu lieu en même temps à deux endroits différents. Les deux pics se recouvreront dans l'histogramme et nos algorithmes ne détecteront qu'un événement. Pourtant, les vocabulaires des deux types d'articles seraient différents. Les entités nommées comme les noms de pays, régions et villes ne seraient pas les mêmes et on pourrait s'en servir pour reconnaitre les deux événements superposés.

3.4 Affichage des résultats

L'affichage est la dernière brique du projet. Elle peux paraître anecdotique mais c'est pourtant la partie la plus parlante qui va permettre d'observer le résultat pratique de l'entièreté du projet, nous ne l'avons donc pas négligée.

Nous avons convenus que les informations essentielles à afficher sur un frise chronologique défilante était prioritairement l'emplacement des évènements dans

le temps, et le positionnement des articles les plus pertinents pour chaque événement. Il fallait pouvoir distinguer les différents éléments entre eux, tout en les associant avec leurs articles correspondant.

Les contraintes techniques étaient claires: il fallait pouvoir générer une page HTML contenant la frise depuis notre script, et la page HTML ne devait pas contenir de PHP pour être directement interprétable depuis un navigateur de recherche. Avec ce cahier des charges en tête nous nous sommes attelés à la recherche d'une solution de frise chronologique compatible.

Un certain nombre de solutions trouvées n'étaient pas directement intégrable dans le script : il s'agissait d'application, ou de création et d'hébergement en ligne. Nous avons donc cherché du coté des scripts de frise en javascript. Plusieurs choix s'offraient à nous comme Timelinr et Simile Timeline. Cette deuxième solution est celle qui était la plus proche du visuel attendu.

Son principal défaut est que l'implémentation de base utilise des Json pour renseigner les informations à localiser sur la frise, ce qui ne fonctionne pas en html/js . Pour contourner le problème, nous avons décidés d'inscrire directement le Json à l'intérieur de la page html.

Les résultats d'affichage Timline.js est une application js qui permet de générer une frise chronologique défilante en y inscrivant des événements ponctuelles où avec une durée. L'application propose aussi de superposer deux frises d'échelles différentes en les synchronisant, permettant ainsi d'avoir un visuel global, et une vision plus détaillées de la durée des évènements.

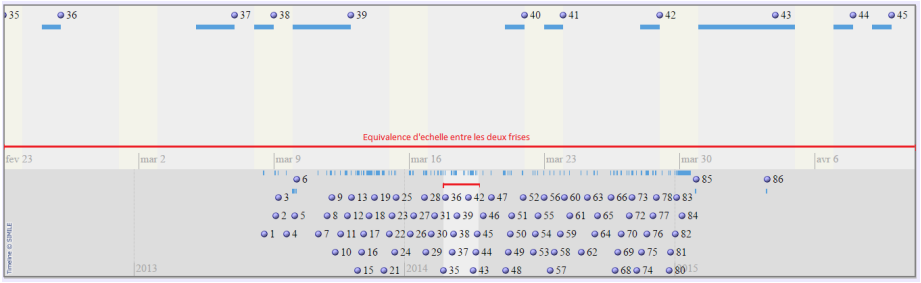
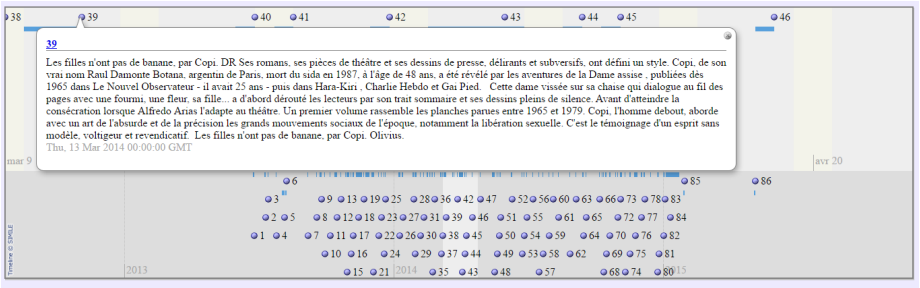


Figure 14: La frise inférieure nous donne la vision global, la supérieure renseigne sur les détails. Chaque point représente un article pertinent. Les zones bleues représentent les évènements et leur étendue temporelle.

Il est possible de se déplacer sur chacune des deux frises. Les déplacements sont proportionnels à l'échelle de la frise, mais s'appliquent sur les deux frises car elles sont entièrement synchronisées. Il est possible de sélectionner un point correspondant à un article pour avoir des informations complémentaires, nous avons pour l'instant choisi d'y insérer le texte.



4 Conclusion

Dans la première partie du projet nous avons respecté les contraintes qui nous étaient imposées. Les résultats obtenus lors des recherches sont, dans l'ensemble, pertinents.

Dans la seconde partie du projet nous sommes arrivés à un moteur de recherche temporelle qui répond en grande partie à nos attentes même si les limites des méthodes que nous avons employées pour détecter les événements importants se font ressentir sur certains types de requêtes.

Nous sommes contents de notre travail sur ce projet dans sa globalité et satisfaits des résultats obtenus. Avec plus de temps nous aurions sans doute pu étoffer notre travail sur le moteur de recherche temporelle. Par exemple trouver un titre pour les événements ou s'intéresser au problème des évenements qui se superposent.

5 Répartition du travail

Jonathan CROUZET En charge de la détection des pics dans les histogrammes. J'ai mis en place et testé différentes méthodes dans des notebooks avant d'en faire un programme Python.

Kévin PASINI En charge de l'affichage et de la détection naïve de pics dans les histogrammes, j'ai également contribué à l'implémentation de la fonction de recherche du projet 1.

Matthieu RÉ En charge de la partie indexation et requête à Elasticsearch dans le second projet. J'ai également organisé les différentes fonctions du moteur de recherche entre elles.

Amal TARGHI En charge de la partie Moteur de recherche classique dans le premier projet, et participé à la rédaction du rapport.