Kunne
Stephan

Rapport de stage de recherche du 3 au 28 juillet 2017 au LRI

Introduction

Du 3 au 28 juillet 2017, j'ai effectué un stage au sein de l'équipe GALaC du Laboratoire de Recherche en Informatique, à Orsay. Le Laboratoire de Recherche en Informatique est une unité mixte de recherche de l'Université Paris-Sud et du CNRS. L'équipe GALaC, un acronyme pour Graphes, algorithmes et combinatoire, regroupe une quarantaine de chercheurs du LRI.

Durant la première semaine du stage, j'ai travaillé sur un problème d'analyse de données, en collaboration avec Yoann Demoli, chercheur au département de sciences sociales de l'université de Versailles. Il s'agissait d'étudier une base de données contenant les résultats d'un sondage. Le sondage portait sur les différents déplacements effectués par des individus au cours d'une journée. Environ 11000 individus avaient répondu au sondage, et l'on connaissait pour chacun la liste de ses déplacements lors de cette journée, avec pour chaque déplacement un certain nombre d'informations et de précisions.

L'objectif de l'étude était de partitionner cet ensemble d'individus en différents "clusters", qui regroupent des individus aux journées similaires. J'ai implémenté en Python la distance de Levenshtein pour comparer deux suites de déplacements, puis plusieurs algorithmes de clusterings qui utilisent les distances calculées.

Durant les deux semaines suivantes, j'ai travaillé sur des problèmes de théorie des jeux, et plus précisément sur la prise de décision face à une situation de "bandit manchot". En guise de mise à niveau, j'ai lu plusieurs articles de recherche, ainsi qu'une série de cours de Tim Roughgarden. J'ai ainsi découvert les bases de la théorie des jeux, j'ai découvert des concepts tels que les équilibres de Nash et le prix de l'anarchie dans des jeux à plusieurs joueurs, et j'ai découvert des algorithmes de prise de décision dans des situations de "bandit" ou de "pseudo-bandit".

Enfin, il m'a été demandé d'essayer de démontrer qu'un algorithme, décrit dans un article de Johanne Cohen (LRI-CNRS), Amélie Héliou (AMIB-LIX) et Panayotis Mertikopoulos (LIG-CNRS), possédait la propriété de "non-regret".

I. Algorithmes pour l'analyse de données

On dispose d'une base de données, qui consiste en une liste de déplacements effectués par 11000 individus au cours d'une journée. Pour chaque déplacement, on connaît le numéro de l'individu, l'heure de départ, l'heure d'arrivée, le motif, le moyen de transport, ainsi qu'un certain nombre d'autres informations.
L'objectif de l'étude est de regrouper les individus en clusters, de manière que deux individus qui ont eu une journée similaire se trouvent dans le même cluster.

La première approche appliquée par M. Demoli consistait à ne retenir que les motifs des déplacements et l'ordre dans lequel ils ont été effectués. Un individu est ainsi représenté par un mot, dont chaque lettre représente un déplacement ; l'alphabet contient douze lettres, qui correspondent à douze motifs de déplacements (accompagnement, maison, travail, tournée professionnelle, cours, repas, courses, démarches, sociabilité, promenade, loisirs, autre).

On peut alors comparer deux individus en calculant leur distance de Levenshtein. La distance de Levenshtein entre deux mots est le nombre minimal d'insertions, de suppressions et de remplacements qu'il faudrait effectuer pour passer d'un mot à l'autre. Plus précisément, on attribue un coût à chacune de ces trois opérations ; la distance de Levenshtein entre deux mots est le coût minimal pour transformer un mot en l'autre.

La critique de M. Demoli sur cette méthode est qu'elle ignore complètement les heures et les durées des déplacements. Ainsi, un individu qui a réalisé trois déplacements en dix minutes sera considéré comme proche d'un individu qui a pris une journée pour réaliser trois déplacements de mêmes motif.

J'ai considéré plusieurs manières de représenter un individu, et j'ai finalement opté pour la méthode suivante. Chaque individu est représenté par une suite de paires (motif, durée), où "motif" peut prendre 13 valeurs : l'un des 12 motifs de la base de données, ou la treizième valeur "pause". Chaque paire représente soit un déplacement, soit le temps d'inactivité entre deux déplacements successifs. De cette manière, la suite de symboles qui représente un individu prend en compte les déplacements effectués par cet individu, ainsi que la dimension temporelle de ces déplacements.

Distance de Levenshtein

Principe général

La distance de Levenshtein entre deux mots est le coût minimal pour transformer le premier mot en le second en n'utilisant que les opérations :
- supprimer une lettre ;
- ajouter une lettre ;
- substituer une lettre à une autre ;

le coup de chacune de ces trois opérations ayant été prédéterminé.

Par exemple, si l'on décide que l'ajout et la suppression coûtent 1, et que la substitution coûte 1.5, alors la distance de Levenshtein entre les mots "ALLOUER" et "ALOUETTE" est 4.5, puisque l'on

peut passer de l'un à l'autre en supprimant un 'L', en ajoutant un 'T' et un 'E', et en remplaçant un 'R' par un 'T'.

Sur des exemples moins évidents, calculer la distance de Levenshtein revient à chercher le plus court "chemin" entre deux mots.

Un algorithme efficace pour calculer la distance de Levenshtein entre deux mots est donné par les principes de la programmation dynamique. Etant donnés deux mots a1…am et b1…bn, on va remplir la matrice D à m+1 lignes et n+1 colonnes, dans laquelle la case (i, j) contient la distance de Levenshtein entre les mots a1…ai et b1…bj. En particulier, la case (0, 0) doit contenir la distance entre la chaine vide et la chaine vide, c'est-à-dire 0 ; la case (i, 0) doit contenir la distance entre a1…ai et la chaine vide, c'est-à-dire i fois le coût de suppression ; la case (0, j) doit contenir la distance entre la chaîne vide et b1…bj, c'est-à-dire j fois le coût d'insertion ; et la case (m, n) doit contenir la distance entre a1…am et b1…bn, qui est le résultat recherché.

On peut remplir cette matrice en appliquant la formule de récurrence suivante :

distance(a1…ai, b1…bj) = min(distance(a1…ai-1, b1…bj) + cout_suppression(ai),
distance(a1…ai, b1…bj-1) + cout_insertion(bj),
distance(a1…ai-1, b1…bj-1) + cout_substitution(ai, bj))

L'algorithme s'écrit donc ainsi, étant fixés les paramètres cout_insertion, cout_suppression et cout_substitution :

```
Distance(a, b):
    définir une matrice D[0..m, 0..n] ;
    pour i de 0 à m :
        D[i, 0] = i ;
    pour j de 1 à n :
        D[0, j] = j;
    pour j de 1 à n :
        pour i de 1 à m :
            si a[i] == a[j] :
              D[i, j] = min(d[i-1, j] + cout_suppression(a[i]),
                            d[i, j-1] + cout_insertion(b[j]),
                            d[i-1, j-1])
            sinon :
              D[i, j] = min(d[i-1, j] + cout_suppression(a[i]),
                            d[i, j-1] + cout_insertion(b[j]),
                            d[i-1, j-1] + cout_substitution(a[i], b[j]))
     renvoyer D(m, n).
```

Application

Dans notre cas, l'alphabet qui constitue les mots à comparer est l'ensemble des couples (motif, durée), où motif est pris dans un ensemble à 13 éléments, et durée est une durée en minutes, c'est-à-dire un entier naturel. La distance de Levenshtein entre deux mots va dépendre du choix des coûts d'insertion, de suppression et de substitution. Les clusters obtenus dépendront des distances, donc de ces paramètres.

J'ai choisi, de manière très arbitraire, les paramètres suivants, tout en laissant à M. Démoli la possibilité de les modifier par la suite.

```
cout_suppression((pause, d)) = max(d, 120)
cout_suppression((m, d)) = 80    si m ≠ pause
```

```
cout_insertion(m, d) = cout_suppression(m, d)

cout_substitution((pause, d1), (pause, d2)) = |d1 - d2|
cout_substitution((m1, d1), (m2, d2)) = +∞   si (m1, m2) ≠ (pause, pause)
```

En particulier, le choix cout_insertion = cout_suppression est nécessaire pour que la distance de Levenshtein possède la propriété de symétrie, c'est-à-dire d(x, y) = d(y, x) pour tous mots x et y.

Le calcul de la distance entre deux mots de longueurs l_1 et l_2 se fait en $O(l_1 l_2)$; il y a $11000^2/2$ tels calculs à faire. Une fois toutes les distances calculées, on peut les stocker dans une matrice pour ne plus avoir à les recalculer durant l'exécution des algorithmes de clustering.

Algorithmes de clustering

Principe général

L'objectif général du clustering est de classifier des données, par exemple en effectuant une partition de l'espace des données. On espère que cette partition sera représentative du monde réel qui a été observé pour générer les données.

Il existe de nombreux algorithmes de clustering, qui correspondent à des models de clusters différents.

Par exemple, si l'on fait l'hypothèse que les données suivent une combinaison de plusieurs lois normales, alors le but d'un algorithme de clustering peut être de retrouver les paramètres de ces lois normales, chaque cluster étant ainsi représenté par une loi normale.

Dans le cadre de mon stage, il m'a été demandé d'effectuer une partition des données en utilisant des variantes de l'algorithme `k-means`.

L'algorithme k-means

L'algorithme k-means est un algorithme simple de partition. Le paramètre k étant fixé, il s'agit de déterminer k sous-ensembles C1, …, Ck de l'ensemble des données, de manière à minimiser les distances intra-clusters. L'idée de cet algorithme est de sélectionner k points qui seront les centres des clusters, et de ranger chaque point x dans le cluster dont le centre est le plus proche de x.

Initialement, les k centres de clusters sont choisis au hasard. Puis, on raffine encore et encore le clustering en remplaçant à chaque étape le centre de chaque cluster par le barycentre de ce cluster. L'algorithme s'écrit donc ainsi :

```
K-means(données) :
    définir k ensembles C1, …, Ck, initialement vides ;
    choisir au hasard k données d1, …, dk ;
    pour chaque donnée x :
        trouver j dans {1, …, k} qui minimise la distance de x à dj ;
        ajouter x à Cj ;
    répéter jusqu'à ce que le clustering soit satisfaisant :
        pour j de 1 à k :
            dj = barycentre(Cj) ;
        pour chaque donnée x :
```

```
            trouver j dans {1, …, k} qui minimise la distance de x à dj ;
            ajouter x à Cj ;
    renvoyer le clustering {C1, …, Ck}.
```

L'application de l'algorithme k-means pose deux problèmes : il faut déterminer par avance le paramètre k, et il faut être capable de calculer le barycentre d'un ensemble de points. Dans notre situation, ce calcul de barycentre est problématique : les données ne sont pas des points dans un espace semblable à R^n, mais les suites de déplacements effectués par un individu tout au long d'une journée. On sait calculer les distances entre deux suites de déplacements (leur distance de Levenshtein, présentée précédemment), mais on ne sait pas représenter ces mots dans un espace géométrique, ni définir un barycentre.

Il va donc falloir trouver une variante de l'algorithme k-means qui permette de raffiner les clusters sans en connaître les barycentres. J'ai implémenté et testé les trois algorithmes suivants :

L'algorithme k-medoids

Cette première variante est très simple : dans chaque cluster C, au lieu de calculer un barycentre des éléments de C, on va sélectionner l'élément "le plus central" de C, c'est-à-dire l'élément x de C qui minimise la somme des distances de x à tous les éléments de C.

```
K-medoids(données) :
    définir k ensembles C1, …, Ck, initialement vides ;
    choisir au hasard k données d1, …, dk ;
    pour chaque donnée x :
        trouver j dans {1, …, k} qui minimise la distance de x à dj ;
        ajouter x à Cj ;
    répéter jusqu'à ce que le clustering soit satisfaisant :
        pour j de 1 à k :
            trouver x dans Cj qui minimise la somme des distances de x
aux éléments de Cj ;
            dj = x ;
        pour chaque donnée x :
            trouver j dans {1, …, k} qui minimise la distance de x à dj ;
            ajouter x à Cj ;
    renvoyer le clustering {C1, …, Ck}.
```

La condition "jusqu'à ce que le clustering soit satisfaisant" introduit un élément d'arbitraire. On peut attribuer un score à chaque clustering, par exemple la somme des distances des éléments au centre de leur cluster, et interrompre l'algorithme dès que l'amélioration du score ne dépasse plus un certain ε prédéterminé. En pratique, sur notre base de données, l'algorithme se stabilisait toujours très rapidement ; après quelques itérations, les centres (ou "medoids") des clusters devenaient fixes, et l'amélioration du score devenait donc exactement zéro.

Les deux algorithmes suivant effectuent un clustering en résolvant le problème des k centres.

Le problème des k centres

Etant donné un ensemble fini E de cardinal n, une distance sur E, et un entier $k \leq n$, on veut trouver un sous-ensemble S_0 de E qui minimise la distance maximale des éléments de E à l'ensemble S_0.

Le problème des k centres est naturellement lié au problème des k clusters : étant donné une solution $S = \{d_1, \ldots, d_k\}$ au problème des k centres, on peut construire des clusters de la manière suivante :

```
pour chaque donnée x :
    trouver j dans {1, …, k} qui minimise la distance de x à dⱼ ;
    ajouter x à Cⱼ
```

Par définition de S, la distance maximale d'un élément à un centre de son cluster est optimale. Si de plus la distance vérifie l'inégalité triangulaire, on peut en déduire que les diamètres des clusters sont optimaux.

Le problème des k centres est malheureusement NP-complet. On ne peut pas espérer trouver une solution optimale S_0 en un temps raisonnable. Toutefois, il existe des algorithmes polynomiaux pour trouver une 2-approximation, c'est-à-dire un ensemble S tel que la distance maximale des éléments de E à l'ensemble S soit au pire deux fois la distance maximale des éléments de E à l'ensemble optimal S_0.

J'ai implémenté deux algorithmes qui trouvent ainsi une 2-approximation du problème des k centres.

Le premier algorithme est présenté par Vijay V. Vazirani dans son livre "Approximations algorithms", dans lequel on peut également trouver une preuve que le problème des k centres est NP-complet, une preuve que cet algorithme fournit une 2-approximation, et une preuve que quel que soit $\varepsilon > 0$, il n'existe pas d'algorithme polynomial pour obtenir une $(2-\varepsilon)$ approximation.

Le second algorithme est présenté par Teofilo F. Gonzalez dans son article "Clustering to minimize the maximum intercluster distance", là encore avec une preuve que l'algorithme fournit une 2-approximation.

L'algorithme k-center de Vazirani

Vazirani approche le problème des k centres comme s'il s'agissait d'un problème de graphes. On considère le graphe complet G dont les sommets sont les éléments de l'ensemble E, et dont les arêtes sont pondérées par les distances des éléments de E.

L'algorithme k-center de Vazirani permet de se ramener au problème de trouver un sous-ensemble dominant minimal de l'ensemble des sommets d'un graphe.

Etant donné un graphe H, on appelle ensemble dominant M un sous-ensemble des sommets de H tel que tout sommet de H est soit dans M, soit voisin dans H d'un élément de M. Un ensemble dominant minimal est un ensemble dominant M tel qu'aucun sous-ensemble de M n'est dominant.

Etant donné un graphe H, on appelle ensemble indépendant I un sous-ensemble des sommets de H tel qu'aucune paire de sommets de I ne soit adjacente dans H. Un ensemble indépendant maximal est un ensemble indépendant I tel que pour tout ensemble J vérifiant $I \subseteq J \subseteq H$, l'ensemble J n'est pas indépendant.

Par définition, tout ensemble indépendant maximal I est aussi un ensemble dominant minimal. Or il est très facile de construire un ensemble indépendant maximal :

```
Ensemble_indépendant_maximal(H) :
    définir un ensemble I, initialement I = Ø ;
    pour chaque sommet x de H :
        si x n'a pas de voisins dans I :
            I = I ∪ {x} ;
```

```
    renvoyer I.
```

Pour résoudre le problème k-center, on commence par trier les arêtes de G dans l'ordre croissant de leur poids, de manière à obtenir une suite a_1, a_2, …, a_m d'arêtes de poids croissants. On définit ensuite une suite croissante de sous-graphes de G : pour i entre 0 et m, on définit le graphe G_i comme étant le graphe dont les sommets sont les sommets de G, et dont les arêtes sont les i arêtes a_1, …, a_i.

On définit le carré d'un graphe : étant donné un graphe H, le graphe H^2 possède les mêmes sommets que H, et deux sommets x et y sont reliés par une arête dans H^2 si et seulement si il existe un chemin de longueur au plus 2 de x à y dans H.

Pour tout j, on appelle M_j un ensemble indépendant maximal de G_j^2. Enfin, on appelle i le plus petit entier tel que le cardinal de M_i soit inférieur ou égal à k.

Avec ces définitions, les k sommets de l'ensemble M_i forment une 2-approximation du problème k-center pour le graphe G.

L'algorithme proposé par Vazirani consiste à construire la matrice d'adjacence du graphe G_0^2, puis celle du graphe G_1^2, etc., et de les utiliser pour construire les ensembles indépendant maximaux M_0, M_1, etc., jusqu'à obtenir un ensemble de cardinal k. La construction de G_{j+1}^2 en fonction de G_j^2 est linéaire en le nombre de points.

Malheureusement, le plus petit entier i tel que $|M_i| = k$ peut être très élevé, car i n'est a priori majoré que par le nombre d'arêtes du graphe G, c'est-à-dire par le carré du nombre de points. Dans ces conditions, la complexité totale de l'algorithme peut être d'ordre cubique.

Toutefois, la construction directe d'un graphe G_j^2 sans passer par le graphe précédent n'est que quadratique ; on peut donc rechercher l'entier i par dichotomie en construisant les graphes G_j^2 dans un ordre approprié plutôt que dans l'ordre croissant.

L'algorithme k-center de Gonzalez

On part d'un unique cluster C1, qui contient tous les éléments. On choisit au hasard un élément, qu'on désigne arbitrairement comme centre d_1 du cluster C1.
A chaque étape *i* de l'algorithme, on sélectionne l'élément *x* le plus étranger à son propre cluster, c'est-à-dire l'élément qui maximise la distance au centre de son propre cluster.
On crée un nouveau cluster C_{i+1} de centre $d_{i+1} = x$, puis on réassigne chaque élément de E au cluster de centre le plus proche.
L'algorithme s'arrête lorsque l'on a atteint le nombre de clusters demandé.

L'algorithme de Gonzalez s'écrit donc ainsi :

```
Gonzalez_k-center(E) :
    choisir au hasard un élément d1 ;
    pour i de 1 à k :
        pour tout j de 1 à k :
            Cj = Ø
        pour chaque donnée x :
            trouver j dans {1, …, i} qui minimise la distance de x à dj ;
            ajouter x à Cj
        trouver l'élément x le plus éloigné du centre de son cluster ;
        di+1 = x ;
    renvoyer {d1, …, dk}
```

Conclusions

Sur notre base de données de 11000 individus, les trois algorithmes aboutissent à des clusters très différents.

Un gros avantage de l'algorithme k-center de Gonzalez est sa construction des clusters un par un. On n'est plus obligé de fournir k comme un paramètre ; on peut lancer l'algorithme sans borner le nombre de clusters, et garder une copie des clusters formés à chaque étape.
On peut par exemple tracer une courbe donnant la distance maximale d'un élément au centre de son cluster, en fonction du nombre de clusters. L'étude de cette courbe permet de choisir une valeur de k a posteriori.

Les deux algorithmes de k-center ont tendance à créer des clusters de tailles très inégales. Pour la grande majorité, les clusters ne contiennent qu'un très petit nombre d'individus chacun, comme s'il s'agissait d'individus "atypiques" qui n'avaient pas pu être regroupés dans un plus gros cluster ; et un très gros cluster contient à lui tout seul plus de 10000 individus.

L'algorithme k-medoids, au contraire, produit des clusters de tailles relativement équilibrées.

Malgré tout, le comportement de ces trois algorithmes dépend très fortement des paramètres très arbitraires qui ont été choisis ; d'une part, le nombre de clusters k ; d'autre part, et surtout, du choix de la représentation des individus en suite de couples (motif, durée), et des distances qui découlent de cette représentation.

Comme les données ne peuvent pas être représentées dans un espace euclidien, il est très difficile de visualiser les clusters obtenus par ces algorithmes. Durant la semaine de stage consacrée à ce problème, je n'ai trouvé aucun moyen d'exercer une critique sur les résultats obtenus, sur leur utilité, ou sur les paramètres choisis.