


Recommandation de documents pour moteur de recherche Coveo

Philippe Blais
Philippe Blouin-Leclerc
William Bourget
Stéphane Caron
Samuel Lévesque


21 décembre 2018

Résumé

Les systèmes de recommandation sont des applications classiques en apprentissage machine. Dans ce projet, nous proposons un modèle permettant proposer des documents pertinents basés sur le résultat d'une recherche écrite. Les données utilisées proviennent du site Web des ressources techniques publiques de la compagnie *Coveo*.

L'ensemble du code et des documents qui ont servi à la résolution de cette problématique et à l'écriture de l'article se trouvent dans le répertoire  du projet.

1 Présentation du problème et état de l'art

Dans le cadre de ce projet proposé par la compagnie Coveo , nous devons utiliser un historique de requêtes faites par des utilisateurs afin de développer un modèle de recommandation de documents. L'objectif du modèle est de proposer une série de 5 documents d'intérêt en fonction de la recherche qui est faite par l'utilisateur et de certaines autres caractéristiques.

Il existe plusieurs méthodes pour bâtir des systèmes de recommandation. Parmi ces méthodes, il en existe deux qui sont fréquemment utilisées en pratique. Ces méthodes sont le filtrage collaboratif et les systèmes basés sur le contenu. La première consiste à calculer des associations entre des items (documents) ou des utilisateurs pour ainsi utiliser cette information pour faire une recommandation. Cette méthode a l'avantage d'être relativement simple à implémenter. La deuxième méthode consiste à utiliser les attributs des items ou des utilisateurs pour prédire les documents d'intérêt. Cette méthode a l'avantage d'apprendre des liens entre certains attributs pour raffiner la qualité

des recommandations.

Dans ces différentes approches décrites, le modèle commence par extraire de l'information des documents cible et peut par la suite se définir une mesure de distance entre une requête et chacun des documents pour déterminer quelle serait la meilleure correspondance requête-document. Dans notre cas, nous n'avons pas accès au contenu détaillé des documents que l'on souhaite prédire, mais seulement à un jeu restreint de caractéristiques comme la source du document, son auteur et son titre.

Nous avons également le choix d'attaquer la problématique comme un problème de régression ou comme un problème de classification. Dans le premier cas, il faut attribuer un score à chacun des documents et ainsi recommander ceux dont le score est le plus élevé. Dans le deuxième cas, il faut tenter de prédire directement une classe, qui correspond à un document en particulier.

Nous avons décidé d'utiliser un système basé sur le contenu dans un contexte de classification. Étant donné que nous avons beaucoup d'utilisateurs différents (environ 600) et beaucoup de documents différents (environ 6000), la méthode basée sur le filtrage collaboratif nous apparaissait moins efficace. De plus, certains utilisateurs sont inconnus par le système, ce qui rend plus complexe la tâche d'utiliser cette information.

Le reste du document est structuré de cette façon : la section 2 décrit notre approche de manière plus détaillée, la section 3 présente notre méthodologie expérimentale, la section 4 présente les résultats expérimentaux, la section 5 fait l'analyse de ces résultats. Finalement, la section 6 présente les différents constats et leçons que nous avons tirés de ce projet.

2 Approche proposée

Selon le livre *Introduction to Information Retrieval* de (Schutze, Manning, & Raghavan,), une approche standard en recherche d'information est de se servir du contenu des documents pour créer un jeu d'attributs pour chaque document disponible. Ces attributs sont ensuite utilisés pour faire l'apprentissage d'un algorithme prédictif. Cela correspond à l'approche basée sur le contenu que nous avons choisie.

On peut également utiliser cette approche pour créer des attributs pour chacune de nos requêtes. Une fois que nous avons deux ensembles d'attributs (requêtes et documents), nous pouvons définir une mesure de similarité qui permettra d'associer une requête à un document. En ayant de bons attributs, il est possible de penser que la mesure de similarité permettra de trouver des associations vers des documents pertinents pour l'utilisateur.

Dans notre cas, nous n'avons pas accès au contenu des documents. Ainsi, au lieu de créer deux ensembles d'attributs, nous avons créé des attributs pour les requêtes seulement. Ensuite, nous allons utiliser ces attributs pour entraîner un modèle supervisé où les classes à prédire sont les différents documents possibles.

Pour bâtir ce modèle, nous pouvons séparer le travail en 2 grandes étapes : prétraitement des données et modélisation. La première étape consiste essentiellement à créer les attributs, alors que la deuxième consiste à entraîner des modèles en utilisant ces attributs.

Pour ce qui est de la création d'attributs, nous pensons qu'une grande partie de l'information prédictive réside dans le contenu de la requête directement. Pour utiliser cette information, nous allons tester plusieurs méthodes de vectorisation de texte pour transformer ces requêtes textuelles en information numérique. De plus, afin de mieux capter le contexte autour des mots dans les requêtes, nous allons utiliser les plongements de mots, décrits dans le chapitre 6.8 du livre *Speech and Language Processing* de (Jurafsky & Martin,). Pour faciliter l'apprentissage, nous allons également normaliser nos requêtes. Plus de détails sont donnés sur ces éléments dans la prochaine section.

Pour ce qui est de la partie modélisation, on souhaite utiliser les représentations numériques de nos requêtes pour comparer différents modèles d'apprentissage automatique et optimiser leurs hyperparamètres pour augmenter le pouvoir prédictif de notre modèle.

Finalement, puisqu'on s'attaque ici à un problème de classification ayant un très grand nombre de classes,

nous allons également tenter de faire un modèle basé sur de l'apprentissage non-supervisé. Pour ce faire, nous allons créer des attributs pour nos documents pour ainsi regrouper certains d'eux et réduire le nombre de classes possibles. On utiliserait par la suite ces classes agrégées pour entraîner un modèle de classification (qui serait cette fois-ci supervisé) retournant plutôt le groupe de documents duquel on choisirait les 5 plus pertinents (fréquents).

3 Méthodologie expérimentale

Avant même de créer les attributs, il était nécessaire de faire la jonction entre les différents jeux de données entre autres pour indiquer au modèle quelle requête a mené à quel document. D'abord, voici les différents jeux de données (avec le nombre d'observations) à notre disposition :

| Type | E | V | T |
|-----------------|-------|-------|------|
| <i>searches</i> | 52133 | 14895 | 7448 |
| <i>clicks</i> | 24491 | 6920 | ?? |

La colonne E correspond à l'ensemble d'entraînement, la colonne V à l'ensemble de validation et la colonne T correspond à l'ensemble de test. Pour ce dernier, nous avons accès aux recherches seulement et non aux clicks. Comme on peut le voir, il y a plus de recherches que de clicks. Cela veut dire que certaines recherches n'ont mené à aucun click. Une même recherche peut également avoir mené à plusieurs clicks. En bref, il existe plusieurs façons de joindre ces recherches avec ces clicks. Puisqu'elles n'avaient pas de pouvoir prédictif, nous avons retiré les recherches qui n'ont pas mené vers un click. De plus, nous avons gardé seulement le dernier click, en se basant sur le moment du click, pour une même recherche. Notre raisonnement derrière ce traitement est que le dernier click est probablement le "bon" et que les précédents ajoutent du bruit au modèle, car ils ne sont pas vraiment des documents d'intérêt.

3.1 Prétraitement des données

Maintenant que nous avons des observations avec des variables explicatives et des étiquettes, la prochaine étape consiste à effectuer certains prétraitements sur ce jeu de données. Nous avons fait plusieurs combinaisons de pré-traitements de données. Premièrement, nous avons sélectionné les variables que nous voulions inclure dans notre modèle prédictif, c'est à dire : `query_expression`, `search_results`, `user_country` et `user_language`.

Une imputation sur les données manquantes a aussi été effectuée. Lorsqu’une donnée était manquante, nous avons pris la valeur moyenne de cette variable pour en faire l’imputation.

Pour attaquer de façon claire le problème de l’optimisation des pré-traitements et de la sélection des hyperparamètres, nous avons utilisé le module *pipeline* de la librairie Python *sklearn*. Ce dernier nous permet de définir nos différentes étapes de pré-traitement par des classes dont les paramètres sont modifiables. En procédant ainsi, on peut faire une recherche en grille non seulement sur les hyperparamètres de nos modèles, mais aussi sur les différents prétraitements possibles.

En ce qui concerne les transformations sur les variables, trois types de transformation ont été effectués. Les trois transformations qui ont été faites sont une transformation des variables catégorielles en valeurs numériques, une normalisation des données textuelles et une vectorisation des données textuelles pour obtenir des résultats numériques utilisables par les modèles d’apprentissage automatique.

Pour la transformation des variables catégorielles, nous avons simplement créé une variable indicatrice pour chaque modalité de la variable catégorielle.

Pour ce qui est de la normalisation des données textuelles, nous avons testé deux scénarios. Dans le premier cas, nous gardons la donnée textuelle telle qu’elle en ne faisant aucune normalisation. Dans le deuxième cas, on utilise l’objet *PorterStemmer* de la librairie Python *NLTK* pour faire du stemming sur nos requêtes. Cette technique consiste à faire une série de traitements automatiques qui retirent les affixes des mots de la requête pour conserver leur racine. Cela permet de regrouper les différentes conjugaisons d’un mot.

Pour la vectorisation des textes de recherche, nous avons intégré quatre types de vectorisation dans notre *pipeline*. Pour les deux premières méthodes, nous avons utilisé l’objet *CountVectorizer* de la librairie Python *sklearn*. Nous avons premièrement utilisé cet objet pour créer une matrice d’occurrence de chacun des mots pour toutes nos requêtes. Ensuite, avec le même objet, nous avons testé la création d’une matrice de présence de chacun des mots (semblable, mais en n’ayant que des résultats binaires).

Par la suite, nous avons testé la méthode *tf-idf*. Cette méthode permet d’attribuer un poids à chaque mot de la requête en fonction de sa présence dans la requête, mais aussi de sa présence dans les autres requêtes. Un mot présent dans tous les documents aura donc un poids moins élevé qu’un mot présent seulement dans quelques requêtes, mais à une grande fréquence.

On peut ainsi mieux quantifier le pouvoir discriminant des mots de la requête.

Finalement, nous avons aussi testé la vectorisation de nos requêtes selon *Word2Vec* de la librairie Python *gensim*. Avec ce modèle, les mots sont projetés dans un espace vectoriel qui permet de capter les similarités entre des mots partageant un entourage semblable. En effet, l’entraînement du modèle de plongement de mots se fait en prédisant l’entourage d’un mot donné à partir de sa représentation vectorielle. On peut ainsi obtenir de bons résultats en généralisation puisqu’on peut plus facilement construire des classifieurs basés sur la représentation vectorielle et qui s’applique à plusieurs mots contrairement à l’utilisation d’une matrice de compte qui n’applique un coefficient qu’à un seul mot.

Nous avons aussi fait une version de notre code où l’on faisait du prétraitement sur les titres des documents. Nous avons fait du *clustering* avec les titres de documents pour prédire des groupes de documents au lieu de prédire des documents uniques. Pour ce faire, nous avons retiré les mots outils des titres de documents. Par la suite, nous avons fait deux combinaisons de prétraitements. Pour la première, nous avons utilisé l’objet *PorterStemmer* et par la suite nous avons vectorisé les titres des documents à l’aide de la méthode *tf-idf*. La deuxième combinaison était de seulement utiliser *Word2Vec* avec le modèle pré-entraîné de Google disponible sur le Web. Une fois que nous avions les attributs des documents, il était possible de faire du *clustering* en utilisant l’objet *KMeans* de la librairie Python *sklearn*.

3.2 Modélisation

Une fois que nous avons un jeu de données interprétable par un algorithme d’apprentissage supervisé, la prochaine étape consiste à tester différents types d’algorithmes et aussi différentes combinaisons d’hyperparamètres. Voici les différents algorithmes que nous avons testés :

- Classifieur *k*-PPV
- Perceptron multicouche

Pour chaque méthode, le *clustering* de documents a été testé. Les modèles étaient alors entraînés à prédire des *clusters* de documents. Il y avait donc beaucoup moins de classes à prédire. Lorsque le *clustering* n’était pas utilisé, les modèles étaient entraînés à prédire directement le *document_id* associé à la requête. Pour la validation, une fois que l’algorithme prédisait un *cluster* de documents, nous utilisions les 5 documents qui étaient les plus fréquents à l’intérieur du

cluster de document prédits comme étant nos 5 documents retournés.

Pour le classifieur k -PPV nous avons considéré deux hyperparamètres différents. En premier lieu, nous avons évidemment testé plusieurs valeurs pour le nombre de voisins (k). Nous avons testé les valeurs suivantes : 1, 3, 9, 11, 15, 25 et 50. Nous avons également testé différentes fonctions de poids : **uniform** et **distance**.

Pour le perceptron multicouche, nous avons seulement testé la fonction d'activation *ReLU*. Cependant, nous avons testé plusieurs topologies de réseaux : 1 couche cachée avec 100 neurones, 2 couches cachées avec 100 neurones chacune et 3 couches cachées avec également 100 neurones par couche.

Pour faire l'entraînement, la validation et le test des différents types d'algorithmes mentionnés, l'approche classique consiste à séparer le jeu de données utilisé en trois partitions, chacune servant justement à faire l'entraînement, la validation et le test. Comme mentionné au début de cette section, les données mises à disposition par Coveo sont déjà regroupées selon cette approche. Il s'avère toutefois que le jeu de données de test ne comprend pas les cibles, soit les clicks, ce qui le rend inutilisable pour présenter une performance finale en test (ce jeu sera utilisé par Coveo pour faire leur propre calcul de la performance du modèle soumis). Pour pallier à ce problème, le jeu de validation a donc été utilisé comme jeu pour rapporter la performance en test, et le jeu d'entraînement pour entraîner et valider les modèles testés. Étant donné une réduction des données utilisées pour entraîner l'ensemble des différents modèles testés, une validation croisée à 2 plis a été effectuée sur le jeu d'entraînement d'origine. Il est plutôt courant d'utiliser au moins 3 plis pour effectuer des validations croisées, mais ici, 2 plis ont été choisis afin de réduire le temps de calcul pour l'entraînement des modèles. Après avoir entraîné les modèles avec toutes les différentes configurations choisies, le jeu de données de validation soumis par Coveo a été utilisé pour rapporter la performance en test. Finalement, afin d'utiliser un maximum de données pour entraîner le modèle final, les deux jeux de données d'entraînement et de validation ont été exploités.

Enfin, pour évaluer la performance de notre modèle et ainsi choisir la configuration optimale, il faut une mesure de performance. Coveo a défini ce qui constitue une bonne prédiction. Chaque document cliqué par un utilisateur après avoir effectué une recherche est considéré comme pertinent. Ce sont donc les cibles

que le modèle doit prédire pour une recherche donnée. Une bonne prédiction doit prédire un ensemble d'au plus 5 documents parmi lesquels on doit retrouver au moins 1 document pertinent. Cette condition est formellement donnée par la fonction de perte suivante, qui indique qu'une perte de 1 résulte de l'absence de documents communs entre l'ensemble de 5 documents renvoyé par notre modèle de prédiction, et l'ensemble des documents cliqués pour la recherche en question :

$$l(y_i, \hat{y}_i) := \begin{cases} 1 & \text{si } y_i \cap \hat{y}_i = \emptyset; \\ 0 & \text{autrement.} \end{cases} \quad (3.1)$$

4 Résultats expérimentaux

Dans un premier temps, nous avons commencé par joindre les recherches et les clicks de la manière expliquée dans la section précédente. Voici le nombre d'observations que nous avons pour les ensembles d'entraînement et (E) de validation (V) après ces traitements :

| Type | E | V |
|------------------------|-------|------|
| <i>searches/clicks</i> | 18571 | 6920 |

Afin d'avoir une idée plus claire des performances reliées à nos modèles, nous avons commencé par nous bâtir un modèle *baseline*. Ce modèle consiste simplement à assigner les 5 documents les plus fréquents, selon les documents dans le jeu de données d'entraînement. Pour ce modèle, nous avons obtenu un score de 0.0359.

Pour choisir notre modèle, nous avons testé 272 combinaisons de prétraitements et d'hyperparamètres différents (47 avec un modèle perceptron multicouche et 225 avec un K plus proches voisins), pendant une durée approximative de 20 heures.

Voici des tableaux synthèses de ces expérimentations présentant le score moyen en validation selon certains paramètres pour nos deux familles de modèles testés (perceptron multicouche (MLP) et k plus proches voisins (KNN)) :

| Performance moyenne | |
|---------------------|-------|
| MLP | KNN |
| 0.246 | 0.131 |

| Normalisation des requêtes | MLP | K-NN |
|----------------------------|-------|-------|
| Aucune | 0.242 | 0.130 |
| PorterStemmer | 0.250 | 0.132 |

| Type de vecteur | MLP | KNN |
|-----------------|-------|-------|
| Word2Vec | 0.094 | 0.101 |
| Compte | 0.298 | 0.142 |
| Compte binaire | 0.302 | 0.142 |
| tf-idf | 0.290 | 0.134 |

| Fréquence minimum | MLP | KNN |
|-------------------|-------|-------|
| 1 | 0.247 | 0.134 |
| 2 | 0.245 | 0.129 |

| Nombre de couches cachées | MLP |
|---------------------------|-------|
| 1 | 0.285 |
| 2 | 0.239 |
| 3 | 0.214 |

| Nombre de voisins | KNN |
|-------------------|-------|
| 1 | 0.115 |
| 3 | 0.140 |
| 8 | 0.140 |
| 11 | 0.135 |
| 15 | 0.132 |
| 25 | 0.130 |
| 50 | 0.128 |

Pour ce qui est du modèle avec le *clustering* des titres de documents, nous nous sommes aperçus, après quelques tests, que le modèle était beaucoup moins performant que le modèle sans *clustering*. Nous avons des résultats aux alentours de 0.21 avec 180 *clusters*. Nous n'avons donc pas pousser plus loin notre analyse.

Voici les 2 combinaisons les plus performantes pour chacun des modèles :

- **Modele MLP** (score validation : **0.360**) : Une couche cachée de 100 neurones avec une normalisation des requêtes et vecteur de compte traditionnel qui considère tous les termes de requête étant présents au moins une fois.
- **Modele KNN** (score validation : **0.166**) : 15 voisins sont considérés et des poids basés sur la distance avec une normalisation des requêtes et vecteur de compte binaire qui considère tous les termes de requête étant présents au moins une fois.

Score du meilleur modèle (MLP) sur le jeu de données de test : 0.356.

5 Analyse des résultats

Les résultats montrent que le meilleur score pour le modèle MLP est plus de deux fois supérieur que le meilleur score pour le modèle KNN (0.360 vs 0.166).

Ceci montre que la problématique de recommandation de documents est probablement trop complexe pour être modélisée par un algorithme basé sur le voisinage comme le KNN. Les deux méthodes fonctionnent bien avec un grand nombre de données et ont généralement un biais faible. Dans le cas du MLP, ce grand nombre de données a probablement permis l'entraînement d'un nombre important de neurones, ce qui est directement lié avec la capacité de notre classifieur à mieux modéliser le phénomène complexe de recommandation de documents. Par rapport à la problématique, le modèle KNN présente aussi un désavantage si on le compare au MLP en terme de temps d'évaluation lors de la présentation d'une nouvelle donnée. KNN doit stocker et utiliser les données pour déterminer le voisinage d'une nouvelle donnée en entrée. Cela peut représenter un obstacle avec le nombre important de données utilisées pour bâtir le modèle, ce qui n'est pas souhaitable pour un engin de recherche qui doit idéalement donner une réponse rapidement pour satisfaire les exigences techniques d'un tel outil. En revanche, le MLP permet de ne conserver que les paramètres du modèle et permet ainsi un temps d'évaluation nettement réduit.

Lorsque le *clustering* de documents était utilisé pour se créer des étiquettes de données, les modèles étaient moins performants. Nous croyons qu'avec seulement le titre des documents, nous n'avions pas assez d'information pour faire des clusters significativement différents. De plus, une fois que le cluster était prédit, ce n'était peut-être pas optimal de simplement proposer les 5 documents les plus fréquents. Il est donc préférable de laisser les algorithmes apprendre directement les documents à prédire plutôt que des groupes de ceux-ci si on utilise une approche trop simpliste comme nous avons fait.

Également, on remarque que, malgré la grande notoriété des plongements de mots en traitement de la langue naturelle, l'utilisation des *embeddings* de mots ne donne pas de bons résultats dans notre contexte. On explique cela par le fait que le vocabulaire de nos requêtes est assez particulier et contient des mots qui ne font pas partie de la langue usuelle. Pour cette raison, le modèle de Google que nous avons utilisé ne permet pas bien de capter les ressemblances entre nos requêtes. Il aurait donc fallu procéder à un entraînement du modèle sur nos requêtes pour qu'il s'adapte mieux à notre contexte, mais puisque les plongements de mots sont basés sur l'entourage des mots et que nos requêtes sont très courtes, l'entraînement que nous avons n'a pas du tout amélioré les résultats.

Maintenant, au-delà de la comparaison des deux algorithmes testés, il reste tout de même qu'un score final de 0.36 n'est pas une performance très bonne. Plusieurs raisons peuvent être derrière ce faible taux de classement. D'une part, le contenu des documents était une donnée qu'on ne pouvait utiliser pour bâtir notre modèle puisque non présente. Or, comme présenté au début de cet article, la littérature montre que c'est principalement sur cette information que se basent les techniques actuelles en recommandation de documents. L'extraction d'information pertinente pour la présente problématique a donc dû être faite principalement sur le texte des recherches. Ce type de texte est pour sa part beaucoup plus bruyé (plus de mots inconnus/ne faisant pas partie de la langue usuelle) que le texte de documents ce qui rend plus difficile l'utilisation de modèles standards entraînés.

6 Conclusion

En conclusion, si nous avions eu plus de temps ou de ressources, nous aurions aimé faire un entraînement plus complet des hyperparamètres de nos modèles, tester d'autres modèles prédictifs et incorporer des informations sur l'historique de recherche de l'utilisateur pour personnaliser les résultats retournés. Nous aurions également été curieux de voir les gains de performance que nous aurions pu obtenir avec le contenu des documents puisque la plupart des approches classiques en recherche d'information s'appuient sur cela.

Finalement, ce projet nous a permis de nous initier à certains concepts du traitement de la langue naturelle, domaine que nous avons survolé brièvement

dans le cadre du cours. De plus, cela nous a permis de réaliser l'importance de bien comprendre et d'analyser les données que nous utilisons pour construire un modèle, car elles sont la base des choix de modélisation à faire. Nous avons également pu voir l'importance de décortiquer le problème en plusieurs petits problèmes afin de procéder incrémentalement sans perdre trop de temps à tester des caractéristiques inutiles. Ce projet a été une très bonne pratique pour voir un type de problématique réel en modélisation prédictive.

Références

- Ai, Q., Bi, K., Guo, J., Croft, W. B. (2018). Learning a deep listwise context model for ranking refinement. doi: 10.1145/3209978.3209985
- Alpaydin, E. (2010). *Introduction to machine learning* (2nd éd.). The MIT Press.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep learning*. MIT Press. (<http://www.deeplearningbook.org>)
- Hastie, T., Tibshirani, R., Friedman, J. (2001). *The elements of statistical learning*. New York, NY, USA : Springer New York Inc.
- Jurafsky, D., Martin, J. H. (2014). *Speech and language processing* (Vol. 3). Pearson London.
- Pang, L., Lan, Y., Guo, J., Xu, J., Xu, J., Cheng, X. (2017). DeepRank : A new deep architecture for relevance ranking in information retrieval. doi: 10.1145/3132847.3132914
- Schutze, H., Manning, C. D., Raghavan, P. (2008). *Introduction to information retrieval* (Vol. 39). Cambridge University Press.