

WEM - Projet

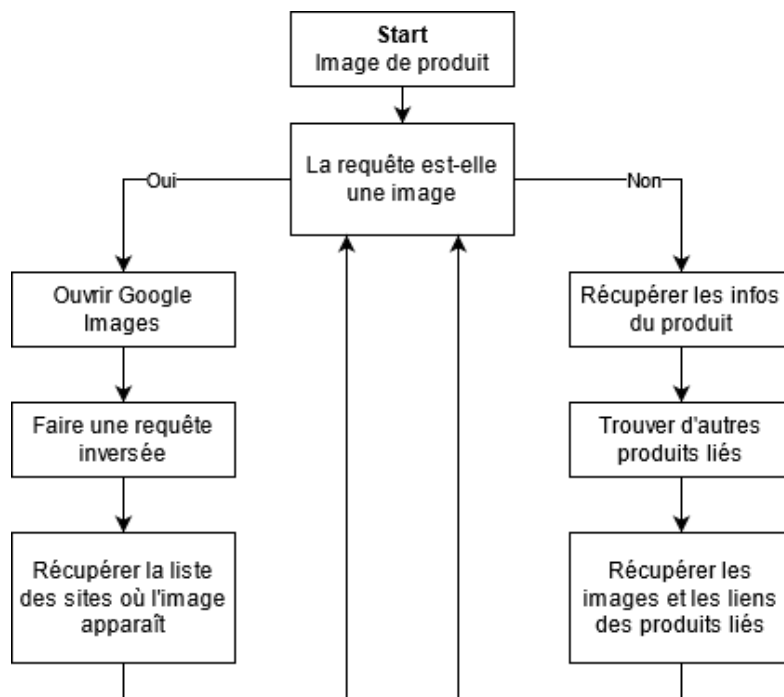
Hochet, Kopp, Silvestri

Dropshipping

La première idée sur laquelle nous étions partis était de réaliser un crawler dont le but aurait été d'identifier les sites de dropshipping, crawler les produits et les prix de vente et finalement offrir une interface de "comparateur" entre les différents sites et des boutiques en ligne de référence comme Amazon ou AliExpress.

Crawler

Notre crawler a été développé en Typescript et utilise la librairie Apify (qui offre également un service en ligne avec proxies).



Le schéma ci-dessus illustre son fonctionnement général. En utilisant une approche très progressive, il tente de récupérer des infos sur un produit sur une page dont la structure est inconnue.

Identifier une page de produit

Si la requête entrante n'est pas une image à rechercher, le crawler part du principe que, si une page est consacrée à un produit et à sa vente, son prix sera affiché en plus grand que d'autres prix potentiels (d'autres produits liés). Pour différencier un prix d'un simple nombre, le crawler commence par identifier toutes les chaînes de caractère contenant un symbole de devise ou le nom d'une devise (dollars, \$, USD, ...). En essayant de parser le reste de la chaîne, on essaie de trouver le prix et, s'il n'est pas trouvé, on parcourt les nœuds du DOM relativement proches pour trouver un nombre.

On retire ensuite les prix “barrés” (en solde ou autre) en testant les règles CSS qui permettent de barrer du texte ou en cherchant un élément du DOM qui permettrait de le barrer également. Cette approche permet d’identifier tous les prix affichés sur une page avec plus ou moins de réussite. Pour identifier le prix “principal” s’il y en a un, on observe la taille du texte de chacun pour sélectionner le plus grand.

Récupérer les produits liés

À partir de la liste de prix obtenue précédemment, on peut partir du principe qu’il s’agit de produits “liés” à celui affiché actuellement. Dans le cas où nous n’avons pas trouvé de produit principal, on suppose que la page en question est de type listing qui propose plusieurs produits (comme une page de catégories).

En partant des éléments de prix, on parcourt le DOM de proximité pour récupérer une image de produit, soit à travers l’élément HTML `` soit à l’aide des règles CSS.

Problèmes

Afin d’analyser et mieux visualiser nos données, nous avons réalisé une petite application web relativement sommaire, qui nous a rapidement montré les limites du système.

Incohérence des données

Malgré les différents essais effectués afin de créer des règles relativement souples mais correctes pour le crawling, nous nous retrouvions avec des données incohérentes pour un certain nombre de sites. Le crawler devait être adapté spécifiquement pour certains sites mais, plus celui-ci avançait, plus le nombre de sites différent augmentait rendant la collecte difficile.

Absence d’indexation des sites de dropshipping

Le problème principal est que la plupart des sites de dropshipping ne sont tout simplement pas indexés par Google ou alors pas trouvables directement. Notre système s’appuyant massivement sur la recherche inversée par images ne pouvait du coup pas fonctionner comme on l’attendait.

Blocage des requêtes

Utiliser des services comme Google Images dans le cadre du projet venait avec l’immense avantage que la recherche d’images inversée devenait triviale à réaliser (se résumant à exécuter le bon code JavaScript sur la page). Mais nous avons très rapidement pris la mesure de sa non-viabilité en nous faisant bloquer par Google.

Taux de change

Un problème un peu moindre mais qui mérite tout de même d’être soulevé est que les sites utilisent des devises différentes et il n’existe aucun service totalement gratuit pour obtenir les taux de change actuels. Nous nous sommes donc appuyés sur une vieille compilation partielle de taux et avons dû implémenter notre propre code de conversion.

Conclusion

Le pipeline était fonctionnel, les données étaient proprement insérées en base de données et une petite application web basique fut développée pour les explorer. Néanmoins nous n'avons pas prolongé cette idée, car elle ne menait à rien tant nous nous heurtions à des problèmes très difficilement surmontables. Notre petite application nous a quand même permis d'observer des résultats intéressants en voyant certains sites vendre des produits plus de dix fois le prix proposé sur d'autres boutiques en ligne.

Fake Amazon Reviews Detection

Introduction

Lors d'achats en ligne, l'une des seules façons pour le client d'évaluer la qualité d'un article est de regarder les commentaires des anciens utilisateurs afin de savoir si le produit répond ou non à ses attentes et s'il est de bonne qualité. Le problème avec cette méthode d'évaluation est qu'il est très facile pour un vendeur de créer un compte alternatif qui sera alors utilisé pour faire des "fake-reviews" indiquant que le produit est bon. L'acheteur ne sait donc plus à qui faire confiance pour son évaluation. L'idée de notre projet est, dans un premier temps, d'effectuer une analyse de sentiment sur les commentaires afin de regarder si la note attribuée par un utilisateur reflète ou non son commentaire puis, dans un second temps, essayer de détecter les bots. Un bot serait un utilisateur qui commente énormément de produits de la même entreprise en mettant toujours de très bonnes notes afin de tromper l'utilisateur. Une fois notre analyse faite, les résultats seront exposés sur un site internet présentant diverses statistiques sur des produits et utilisateurs. Notre projet va se concentrer exclusivement sur le site de vente en ligne Amazon.

Contexte et objectifs du projet

Pour notre seconde idée nous nous sommes tournés vers Amazon afin de titiller un peu ses serveurs en le crawlant. L'objectif ici était dual :

- Analyser les avis utilisateurs et exécuter un modèle de Natural Language Processing afin d'évaluer si la note mise par l'utilisateur reflète bien le texte qu'il a rédigé
- Essayer de détecter les bots à la solde des entreprises, c'est-à-dire des utilisateurs qui commenteraient beaucoup de produits de la même entreprise en y mettant toujours d'excellentes notes

Nous nous tournons donc majoritairement vers les avis des utilisateurs pour des catégories de produits données. Le projet s'est déroulé en quatre parties distinctes :

1. Développement d'un crawler adapté à nos besoins et exécutions de ce dernier
2. Prétraitement des données collectées pour les normaliser et trier les éléments incohérents puis analyse à l'aide des librairies Pandas et Hugging Face Transformers pour exécuter l'analyse de sentiments
3. Stockage des données en base de données Postgres et développement d'une application client-serveur
4. Analyse des données et observations à partir des résultats présentés par l'application pour identifier les robots potentiels

Le rapport détaillera chaque partie dans l'ordre, chacune étant relativement indépendante des autres.

Crawler et données

Pour collecter les données utilisateur dont nous avons besoin nous avons mis en place un crawler également basé sur Apify et développé en Typescript, adapté spécifiquement pour Amazon.

Structure d'Amazon

Amazon a une structure de site relativement logique où les éléments sont bien répartis :

- Chaque produit a un identifiant unique présent dans l'URL
- Chaque produit a un "store" qui vend ce produit, ce dernier possède également une page et son identifiant unique est également présent dans l'URL
- Les URL des pages pour récupérer les avis utilisateurs sont logiquement construites à partir des identifiants de produits

Notre crawler s'est donc massivement appuyé sur ces trois règles pour collecter les données.

Développement du crawler

Le crawler s'appuie sur Apify qui s'occupe de la gestion de la queue de requêtes, le multithreading et offre une couche d'abstraction complète autour des mécanismes plus complexes. Nous avons combiné Apify avec Playwright, une librairie de Microsoft permettant d'utiliser un navigateur web en mode headless (sans rendu visuel) pour l'exécution du JavaScript client, nécessaire sur le site d'Amazon.

Le crawler supporte deux types de pages Amazon en requête entrante (point d'entrée de l'analyse de page), une page de produit ou une page de magasin.

Crawling d'une page de produit

Dans ce cas, le crawler va, dans un premier temps, collecter les informations sur le produit (titre, prix, catégories, etc.). Nous récupérons également le nom du *store* ainsi que son URL pour le visiter plus tard, en plus des produits liés. Pour cela, Amazon affiche plusieurs *sliders* de produits comme les "produits récemment vus" qui ne nous intéressent pas, mais également les "produits basés sur ce que vous avez vu" qui eux nous intéressent beaucoup.

Une fois la récupération terminée, nous changeons de page *sans notifier Apify* (en restant dans le même contexte) pour aller récupérer les avis utilisateurs sur le produit.



Crawling d'une page de magasin

Les pages de magasin offrent la possibilité au magasin en question d'ajouter du contenu comme il l'entend, la structure peut donc varier énormément d'un magasin à un autre, nous rappelant la problématique du crawler de dropshipping. Cependant ils ont tous un point commun, c'est la volonté de vendre des produits, on retrouve donc plein de produits sur chaque page de magasin. Pour les récupérer, nous nous appuyons sur le fait que les URL des produits ont toutes la même structure avec l'identifiant unique à l'intérieur. Ceci nous permet très facilement d'ajouter ces URL à la liste des pages à crawler.

Exécution et collecte des données

Amazon, comme Google, a tendance à bloquer les requêtes des utilisateurs suspects comme nous. Pour résoudre ce problème, nous avons, dans un premier temps, tenté d'exécuter notre crawler en utilisant le service payant d'Apify (et en s'appuyant sur le tier gratuit d'un mois) mais celui-ci était trop limité et n'offrait pas suffisamment de proxies. Nous avons finalement opté pour une exécution parallèle chez chacun de nous. Apify gère, sur une instance de crawler, les pages qui ont déjà été parcourues et nous évite de les revisiter. Ceci ne marche pas lorsqu'on l'exécute chacun de notre côté. Pour éviter les doublons nous avons donc mis en place une solution de stockage partagée sur MongoDB Atlas, dans laquelle nous allons vérifier l'existence de la page à visiter avant de la crawler.

Avant d'être totalement bloqués, nous avons pu collecter 161'575 avis utilisateurs sur 1860 produits.

Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size	Properties
products	1,860	797.4 B	1.4 MB	1	68.0 KB	
reviews	161,575	710.2 B	109.4 MB	1	1.6 MB	

Timeouts et simulation d'utilisateurs

Pour retarder au maximum le moment où nous serions bloqués, nous avons mis en place des timeouts et des déplacements aléatoires de la souris. Nous avons également ajouté des scrolls de la page entière plusieurs fois pour obliger le code JavaScript du site d'amazon qui s'exécute que lorsque l'utilisateur est suffisamment descendu dans la page à s'exécuter, notamment pour charger les différents sliders affichant les produits connexes.

Limite de 300 avis utilisateurs par produit

Le crawling des avis utilisateurs se fait sur une page unique en interagissant avec le code JavaScript de la page. Celle-ci n'affiche que 10 avis à la fois et le temps de chargement successif prend passablement de temps. Nous avons donc posé une limite de 300 avis utilisateurs par produit pour les raisons suivantes :

- Certains produits peuvent avoir plusieurs dizaines de milliers d'avis, ce qui prendrait théoriquement plusieurs heures selon le temps de chargement variable, en supposant que l'on ne soit pas bloqué avant
- Nous souhaitons maximiser la possibilité d'avoir plusieurs avis sur plusieurs produits différents par le même utilisateur pour détecter si c'est un bot, nous avons donc préféré maximiser le nombre de produits

Prétraitement des données et sentiment analysis

Nous avons réalisé une étude préliminaire des données et un prétraitement dans des notebooks jupyter à l'aide de Pandas.

Prétraitement

À l'origine nous avons mis en place un début de pipeline dans RapidMiner dans l'objectif de réaliser le sentiment analysis avec. Cependant nous avons été mis en garde quant aux possibilités d'en extraire les résultats et nous nous sommes donc tournés sur des scripts python pour cette partie.

Le prétraitement a notamment inclus l'extraction du pays d'origine du commentaire et de sa date de publication (stocké les deux dans une chaîne), le nettoyage des caractères trop spéciaux et un filtrage des avis sans notes.

Sentiment Analysis

Pour réaliser le sentiment analysis nous avons utilisé le modèle bert-base-multilingual-uncased-sentiment édité par NLPTown et publié sur Hugging Face, dont nous avons utilisé la librairie Transformers pour appliquer le modèle. Celui-ci vient avec l'avantage qu'il gère plusieurs langues (ce qui n'était pas impossible dans le cas d'avis utilisateurs et aucun moyen de connaître la langue dans laquelle ceux-ci ont été rédigés) mais également qu'il note les avis directement de 1 à 5 étoiles, ce qui correspond parfaitement à la notation utilisée par Amazon.

Le modèle est néanmoins annoncé avoir un taux d'exactitude moyen sur leur dataset d'entraînement (~67%) mais bien meilleur lorsqu'on accepte une fourchette d'une étoile de différence (~95%).

Stockage en base de données

Une fois l'analyse de sentiments réalisés et les données traitées et nettoyées, nous les avons insérées dans une base de données Postgres, un peu parce que nous ne sommes pas totalement à l'aise avec Pandas mais surtout, car nous en avons besoin pour mettre en place l'application client-serveur de démonstration. Celle-ci s'est faite en plusieurs étapes successives :

1. Insertion des données initiales depuis Pandas
2. Calcul des différents résultats d'agrégation (étoiles par produit et par magasin, etc.) dans des vues
3. Création de tables à partir des vues SQL

L'objectif de la réalisation de tables était principalement d'accélérer l'interface graphique, certaines requêtes étant particulièrement lourdes et longues.

Développement d'une application client-serveur pour l'observation des données collectées

Application client-serveur

Afin d'avoir un résultat visuel et compréhensible sur les données collectées et travaillées, mais également pour pouvoir prolonger l'étude de celles-ci, nous avons mis en place une application client-serveur.

Backend

Le côté serveur est implémenté à l'aide de Flask, serveur web en python. Celui-ci s'appuie sur psycopg2 comme driver pour Postgres. Son architecture est relativement simple et orientée autour de deux modules python, un premier, *db.py*, s'occupant des requêtes avec Postgres, et un second, *__init__.py*, fichier d'entrée pour Flask, dans lequel nous déclarons les différents endpoints d'API accessibles.

Frontend

Le frontend est développé à l'aide de la librairie React en Typescript, s'appuyant notamment sur Tailwind pour l'apparence, Apex Charts pour l'affichage de graphiques et react-router-dom pour la gestion du routing.

Fondamentalement l'application propose trois fonctionnalités principales différentes :

- Une s'occupant des magasins, les listant et fournissant des détails sur chaque magasin
- Une s'occupant des auteurs de reviews et offrant des détails sur chacun d'eux
- Une dernière permettant la recherche d'un magasin ou d'un auteur

Observation des données d'analyse de sentiment

L'objectif de cette partie était de voir si les commentaires d'avis utilisateurs correspondaient bien aux notes données.



L'illustration ci-dessus est une capture d'écran de notre application montrant les résultats de l'analyse des sentiments sur l'ensemble des produits crawlés (et des avis liés) de l'entreprise CeraVe. Comme on peut le voir, la courbe des avis utilisateurs (en bleu) suit relativement la courbe des avis calculés par l'analyse de sentiments (en vert).

Les statistiques en haut montrent également les moyennes générales, dans ce cas on peut voir que le score moyen est de 3.8 étoiles alors que le score calculé est de 3.5

Dans l'ensemble, nous avons observé que le modèle d'analyse de sentiments a tendance à noter plus sévèrement les commentaires que les notes des utilisateurs.

De manière générale, nous n'avons pas observé de différences notables entre les évaluations des utilisateurs et les notes calculées par notre modèle d'analyse de sentiments.

Détection de robots

Pour identifier les robots, nous avons tenté plusieurs approches.

Proximité des chaînes de caractères

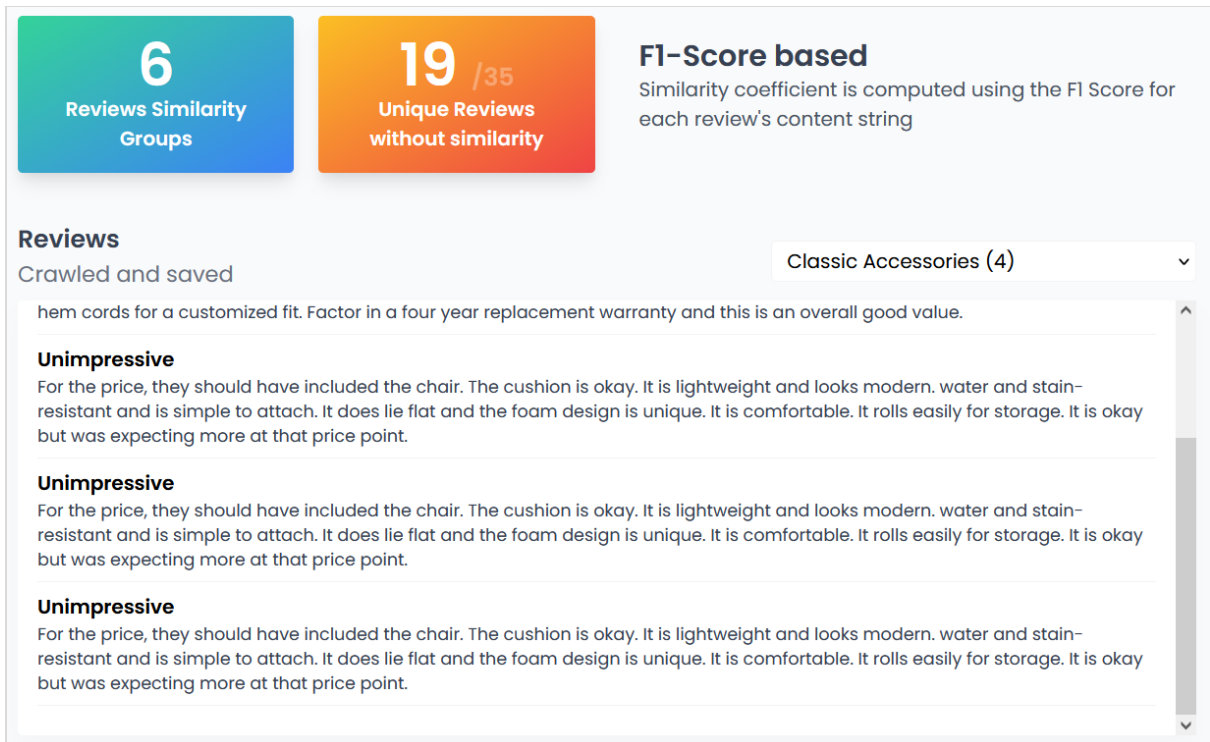
Dans un premier temps nous avons calculé la proximité des chaînes de caractères des commentaires d'un même auteur. Nous sommes partis du principe qu'un robot est un seul compte et qu'il commente des textes relativement similaires ou identiques. Cette approche implique que nous nécessitions relativement beaucoup de commentaires d'un même utilisateur, nous les avons donc triés par tel.



Cette capture représente le groupement de commentaires de l'utilisateur dont nous avons crawlé le plus d'avis. Nous avons enregistré 48 commentaires en tout, en avons trouvé 37 relativement distincts et 11 répartis dans 4 groupes très proches ou similaires selon le calcul du score F1, dont la formule est $2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$.

L'observation initiale des résultats était moyennement concluante, celle-ci dépend massivement du nombre de commentaires que nous avons par utilisateur, hors celui-ci descend très rapidement. 48 commentaires pour l'utilisateur dont nous en avons le plus est trop peu pour conclure à un robot.

Nous avons néanmoins observé des résultats intéressants comme illustrés dans la capture ci-dessous :



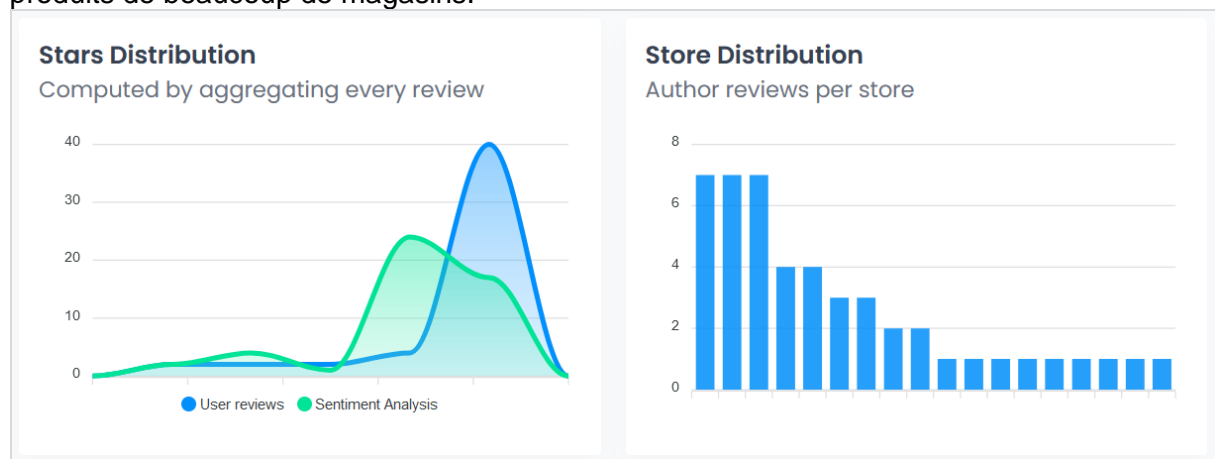
Cet utilisateur par exemple a 16 avis répartis dans 6 groupes de texte proche selon le score F1, et après observation on peut voir que les avis sont identiques. En conclure qu'il s'agit d'un robot est peut-être prématuré mais c'est une bonne piste. Plusieurs utilisateurs suivent ce schéma de commentaires identiques.

Autres indices de distance et approches

Nous avons également tenté de mesurer la distance entre les deux textes à l'aide de la distance de Levenshtein mais les résultats étaient encore moins concluants. Nous avons donc souhaité tenter une approche avec modèle de machine learning et avons notamment identifié un modèle basé sur DistillRoberta (modèle de base Roberta avec knowledge distillation), cependant nous n'avons pas réussi à le faire fonctionner dans le cadre du projet.

Commentaires par magasin

Une autre hypothèse était celle du nombre de commentaires par magasin. Théoriquement nous pourrions penser qu'un robot, affilié à un magasin, aura tendance à largement commenter les articles de celui-ci en y mettant des notes favorables sans pour autant s'attarder sur d'autres magasins. Nous avons donc cherché à regarder si les auteurs dont nous avons le plus de commentaires avaient commenté dans plusieurs magasins. Les résultats que nous avons obtenus ne sont pas encourageants, car nos utilisateurs ont commenté des produits de beaucoup de magasins.



Comme on peut le voir sur la capture ci-dessus, l'utilisateur dont nous avons le plus de commentaires en a mis (pour ceux que nous avons crawlé) pour 18 magasins, sur 48 commentaires. Le graphique de droite montre par contre que cet utilisateur a des avis notés plus sévèrement par notre modèle d'analyse de sentiments, mais restant globalement dans la marge d'erreur acceptée.

Groupement par magasin

Suite aux résultats précédents, nous avons tenté l'approche inverse, en groupant les avis par magasins puis par utilisateurs, afin de maximiser le nombre d'avis d'un même magasin par utilisateur. Cependant dans ce cas de figure les maximums d'avis ne dépassent pas 5-6 par magasin et ne sont donc pas suffisants pour conclure à un robot.



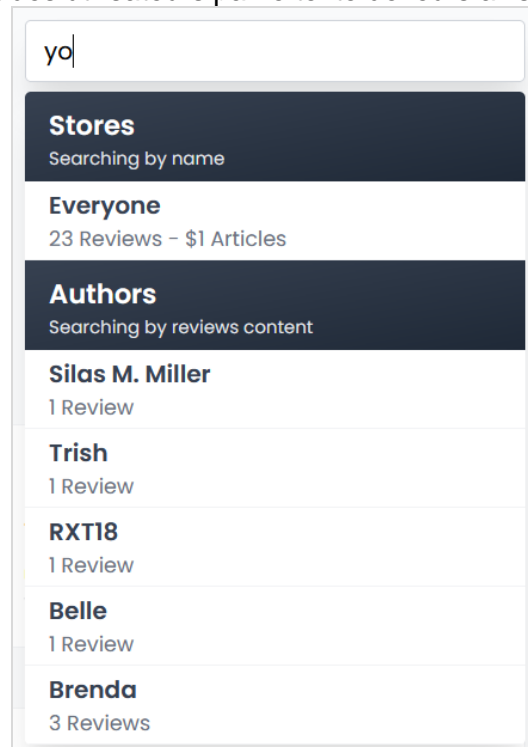
La capture suivante montre le résultat obtenu pour l'utilisateur ayant le plus commenté CeraVe, le magasin le plus évalué que nous ayons. Nous n'y avons pas détecté de commentaires similaires à part pour deux parmi tous ceux qu'il a rédigés.

Travail supplémentaire sur l'application

Afin de rendre l'application plus attractive et agréable à utiliser, nous avons implémenté quelques fonctionnalités supplémentaires, non liées à l'analyse de données.

Recherche de magasins et d'auteurs d'avis

Pour faciliter la recherche d'auteurs et de magasins, nous avons implémenté une barre de recherche. Dans un premier temps nous souhaitions la développer en nous appuyant sur TXAi mais le temps étant compté et les difficultés rencontrées lors de l'implémentation nous a empêché de finaliser son implémentation. Nous nous sommes donc rabattus sur une méthode de recherche plus classique en nous appuyant sur Postgres. Celle-ci permet de trouver des magasins par leur nom et des utilisateurs par le texte de leurs avis sur des produits.



The screenshot shows a search bar with the text 'yo' entered. Below the search bar, there are two main sections: 'Stores' and 'Authors'. The 'Stores' section is titled 'Stores' and has a subtitle 'Searching by name'. It shows a result for 'Everyone' with '23 Reviews - \$1 Articles'. The 'Authors' section is titled 'Authors' and has a subtitle 'Searching by reviews content'. It shows a list of authors: 'Silas M. Miller' (1 Review), 'Trish' (1 Review), 'RXT18' (1 Review), 'Belle' (1 Review), and 'Brenda' (3 Reviews).

Stores
Everyone 23 Reviews - \$1 Articles

Authors
Silas M. Miller 1 Review
Trish 1 Review
RXT18 1 Review
Belle 1 Review
Brenda 3 Reviews

La capture d'écran ci-dessus illustre la barre de recherche retournant des résultats pour le mot clé savamment choisi "yo".

Pagination

La pagination, souvent oubliée mais néanmoins importante, permet de parcourir une liste de résultats de manière progressive sans avoir à charger l'entièreté du dataset en mémoire. Celle-ci est proprement implémentée à chaque fois qu'une liste d'objets est affichée.

Conclusion

Comme nous l'avons présenté dans ce rapport, nous n'avons pas obtenu de résultats relativement concluants concernant nos deux sujets d'analyse. Nous pensons que la raison principale est le manque de données, les nôtres étant trop éparses et réparties sur trop de magasins et de produits. Si l'on veut établir des conclusions à partir de notre projet, il serait intéressant de se limiter à un seul magasin et de crawler un nombre de review par produit beaucoup plus important. En effet, vu que seulement 300 évaluations sont récupérées sur l'ensemble, il est très probable que notre système soit passé à côté de plusieurs commentaires provenant de bots. De plus, les commentaires sélectionnés étant les 300 premiers, le résultat est dépendant de l'algorithme d'Amazon qui présente les commentaires qui lui semblent les plus pertinents. Il n'est donc pas impossible qu'un tri ait déjà été effectué avant l'affichage. Malgré ces défauts, notre projet permet de représenter de façon claire les distributions de commentaires par produit, par utilisateur ou par magasin et montre que les commentaires laissés par un utilisateur ne reflètent pas forcément la note attribuée.