Mangaki : comment coder un système de recommandation en Python !

Introduction

Avec Netflix, Quora, Amazon, Mangaki, les systèmes de recommandation sont omniprésents dans nos vies. Mais comment fonctionnent-ils ? Cet article présente les algorithmes principaux qui permettent de les concevoir. Nous en profitons pour décrire notre tout dernier algorithme utilisant les posters des œuvres pour améliorer les recommandations, présenté au workshop MANPU le 10 novembre 2017 à Kyoto, à l'occasion de la conférence *International Conference on Document Analysis and Recognition* (ICDAR).

Tous les modèles présentés dans cet article sont disponibles sur notre GitHub, à l'adresse : https://github.com/mangaki/mangaki/
Vous y trouverez des notebooks Jupyter pour pouvoir reproduire les expériences.

Mangaki.fr : recommandation d'anime et de mangas

Mangaki.fr est un site de recommandation de culture pop japonaise développé par une association d'otakus invétérés. Le principe : vous notez quelques anime, film ou mangas (ou donnez votre identifiant myAnimeList), et le système vous en recommande d'autres à voir ou lire, cf. Figure 1. Il est également possible de demander à Mangaki de prioriser sa watchlist (si l'on hésite entre plusieurs œuvres), de trouver des gens qui ont un profil proche au vôtre,

etc. L'accent est davantage mis sur les perles rares de la culture japonaise que sur le fait de simplement vous recommander le dernier blockbuster en date.

Figure 1. Mangaki permet de noter des œuvres J'adore / J'aime / Neutre / Je n'aime pas / J'ai envie de voir / Je n'ai pas envie de voir.

Notre choix : Python

Mangaki est codé avec le célibrissime framework web Django : en effet, puisque la communauté scientifique en machine learning aime tant utiliser Python (NumPy, SciPy, scikit-learn, TensorFlow), autant coder tout dans un même langage ! Un des auteurs de cet article se souvient avec émoi avoir codé des produits de matrices en PHP/MySQL, et en conserve un goût amer.

Python est particulièrement adapté pour nous permettre d'écrire et de lancer des tests unitaires :

```
./manage.py test mangaki
```

Django est célèbre pour son *scaffolding* (obtenir le panneau admin automatiquement après avoir conçu les modèles), avoir une API REST qui permet de faire le lien entre les algorithmes de machine learning et les vues.

De plus, Python étant présent sur la plupart des distributions Linux, il est d'autant plus facile de déployer Mangaki à l'aide de notebooks Ansible, un outil d'orchestration de déploiement écrit en… Python !

Formaliser le problème

Comme vous aurez eu l'occasion de le lire dans ce fascicule ou sur Internet, lorsqu'on souhaite faire du machine learning, la partie la plus importante est la **formalisation du problème** : que cherche-t-on à prédire, et avec quelles données à disposition ? Ensuite, il est naturel de choisir un modèle parmi la panoplie d'algorithmes existants pour résoudre ce problème, puis enfin d'optimiser les hyperparamètres, période quelque peu agaçante du data scientist :-)

En l'occurrence, on cherche à prédire les notes d'utilisateurs sur des films, à partir de notes existantes, et éventuellement d'informations supplémentaires (des posters, par exemple). Donc nos données d'entraînement *X* seront rassemblées dans un tableau de paires (`user_id`, `work_id`) où `user_id` est un ID d'utilisateur et `work_id` un ID d'œuvre (manga, anime ou film). Pour chacune de ces paires (`user_id`, `work_id`), la note qu'a donnée `user_id` à l'œuvre `work_id` (un nombre positif ou négatif, peu importe) se trouvera dans un tableau *y* à une colonne. Pour entraîner un modèle, on fera : `algo.fit(X, y)`.

Nos données de test seront un autre tableau *X* de paires pour lesquelles on souhaitera prédire la colonne correspondante de notes. Pour déterminer le tableau de notes `y_pred`, on fera simplement : `y_pred = algo.predict(X)`.

On sépare données d'entraînement et données de test afin de pouvoir mesurer la performance du modèle, autrement dit l'algorithme pourrait apprendre parfaitement les données d'entraînement, mais pourrait se tromper sur un nouveau jeu de données : cette incapacité à généraliser est appelée **surapprentissage** (ou *overfitting*).

Données à disposition

Pour nos expériences, on pourra supposer que les notes des gens sont dans une matrice SciPy `ratings` où `ratings[i, j]` contient la note qu'a attribuée l'utilisateur numéro *i* à l'œuvre numéro *j*. Ce format est pratique parce qu'il permet de faire du *slicing* (aussi appelé *tranchage* par mon boucher du coin de la rue) facilement : pour avoir la sous-matrice des gens indexés par user_ids et des œuvres indexées par work_ids, on pourra faire : `ratings[user_ids, work_ids]`.

Comme en moyenne, chaque utilisateur ne note qu'1 % des films, la matrice `ratings` est pleine de zéros : on dit qu'elle est **creuse**. Dans la bibliothèque SciPy, il existe de nombreux types de matrices creuses pour économiser de la mémoire et du temps de parcours. Selon l'opération qu'on cherche à faire, on préférera les types suivants :

- `coo_matrix` (*coordinate*) si l'on dispose des triplets (i, j, m_{ij}), c'est-à-dire nos paires `user_id`-`work-id` et les notes correspondantes ;
- `csr_matrix` (*compressed sparse row*) si l'on souhaite facilement accéder aux éléments non nuls d'une ligne donnée, c'est-à-dire les œuvres vues par un certain utilisateur ; c'est également un format conseillé pour la multiplication de matrices ;
- `csc_matrix` (*compressed sparse column*) si l'on souhaite facilement accéder aux éléments non nuls d'une colonne donnée, c'est-à-dire les utilisateurs ayant vu une certaine œuvre.

Évaluation des performances : fonction de perte et objectif

Afin de déterminer l'algorithme de recommandation le plus performant, il faut choisir une **fonction de perte**. Par exemple, est-ce que l'algorithme reconstruit bien les notes existantes ? Est-ce que les filles, présentes en minorité sur Mangaki, obtiennent tout de même des recommandations satisfaisantes ? Tout cela permettra de définir la **fonction**

objectif à optimiser, ce qui est crucial pour choisir l'algorithme ensuite. La fonction objectif n'est pas forcément exactement la fonction de perte, comme nous allons le voir.

Enfin, comment vérifier que le modèle généralise bien, c'est-à-dire qu'il ne surapprend pas ? La **méthode de validation croisée** consiste à entraîner les modèles avec 80 % des données (l'ensemble d'entraînement) et tenter de reconstruire les 20 % restants (l'ensemble de test). Les modèles sont comparés avec la valeur de la fonction de perte.

La fonction de perte que nous choisissons est la **RMSE** (*root mean squared error*), la racine carrée de la moyenne des erreurs de prédiction au carré. Par exemple, si un algorithme prédit un 4 au lieu d'un –2, il recevra une pénalité de $(4 – (–2))^2 = 6^2 = 36$ lors du calcul de la somme. S'il prédit correctement, il ne recevra aucune pénalité. Pour des notes comprises entre 1 et 5, la RMSE des algorithmes de recommandation de la littérature est généralement de l'ordre de 1, mais cela dépend du jeu de données.

Prédire les notes

Un algorithme naïf (KNN, *k-nearest neighbors*)

Tout d'abord, un constat à l'amiable : vous avez tendance à bien vous entendre avec ceux qui partagent vos passions. Lorsqu'un de vos amis vous conseille de voir un film, soit vous savez qu'il a des goûts proches aux vôtres et vous considérez sa recommandation, soit vous savez qu'il a des goûts diamétralement opposés aux vôtres, et vous lui répondez : « Oui oui, bien sûr » tout en prenant soin d'ignorer sa recommandation.

Le principe consiste donc, pour chaque paire `user_id`-`work_id` pour laquelle il faut prédire une note :

- Identifier les *k* plus proches voisins de `user_id`, parmi les gens ayant noté l'œuvre `work_id`.
- Récupérer leurs notes et en calculer la moyenne. Ce sera la prédiction renvoyée.

C'est tout simple, et déjà performant. En code, ça donne ça :

```
import numpy as np
from scipy.sparse import coo_matrix
from sklearn.metrics.pairwise import cosine_similarity


class MangakiKNN:
    def __init__(self, nb_users, nb_works, nb_neighbors=20):
        self.nb_neighbors = nb_neighbors
        self.nb_users = nb_users
        self.nb_works = nb_works
```

```
    def fit(self, X, y):
        user_ids = X[:, 0]
        work_ids = X[:, 1]
        self.ratings = coo_matrix((y, (user_ids, work_ids)),
                                  shape=(self.nb_users,
self.nb_works))
        self.ratings_by_user = self.ratings.tocsr()
        self.ratings_by_work = self.ratings.tocsc()
        self.user_similarity =
cosine_similarity(self.ratings_by_user)

    def predict(self, X):
        y = []
        for user_id, work_id in X:
            closest_raters = list(self.ratings_by_work[:,
work_id].indices)
            closest_raters.sort(
                key=lambda rater_id: self.user_similarity[user_id,
rater_id],
                reverse=True)
            neighbor_ids = closest_raters[:self.nb_neighbors]
            rating = self.ratings_by_work[neighbor_ids,
work_id].mean()
            y.append(rating)
        return np.array(y)
```

Notez l'emploi des matrices creuses de SciPy (cf. la section *Données à disposition* plus haut) :

- d'abord on insère les triplets `user_id`-`work_id`-`rating` dans une `coo_matrix` ;
- on la convertit en `csr_matrix` via la méthode `tocsr()` pour pouvoir facilement accéder à la liste des œuvres vues par un utilisateur ;
- et en `csc_matrix` pour pouvoir facilement accéder à la liste des utilisateurs ayant vu une œuvre.

Le *slicing* joue un rôle important dans ce code. Par exemple, `X[:, 0]` contient la première colonne de la matrice NumPy *X*, donc la liste des `user_id`. En ce qui concerne SciPy, `self.ratings_by_work[:, work_id].indices` contient la liste des ID d'utilisateurs ayant noté l'œuvre d'ID `work_id`.

Fit. Toute l'essence de l'algorithme réside dans la fonction de similarité qui permet de déterminer les plus proches voisins. Pour cet article, on a choisi la **similarité cosinus** (donnée par la fonction `cosine_similarity` de scikit-learn) : si Alice et Bob ont leurs lignes R_a et R_b dans la matrice ratings, alors leur similarité est $R_a \cdot R_b$ / ($||R_a|| \, ||R_b||$) où « · » désigne le produit scalaire canonique réel. Cela représente le cosinus de l'angle formé par leurs vecteurs en dimension *M*, où *M* est le nombre de films. Intuitivement, plus Alice et Bob ont de points communs, plus leur score augmente, et plus ils ont de désaccords, plus leur

score diminue, cf. Figure 2. Ainsi, `self.user_similarity` est une matrice symétrique de taille *N* × *N* dont l'élément `self.user_similarity[user_id, rater_id]` contient la valeur de similarité cosinus de `user_id` et `rater_id`.

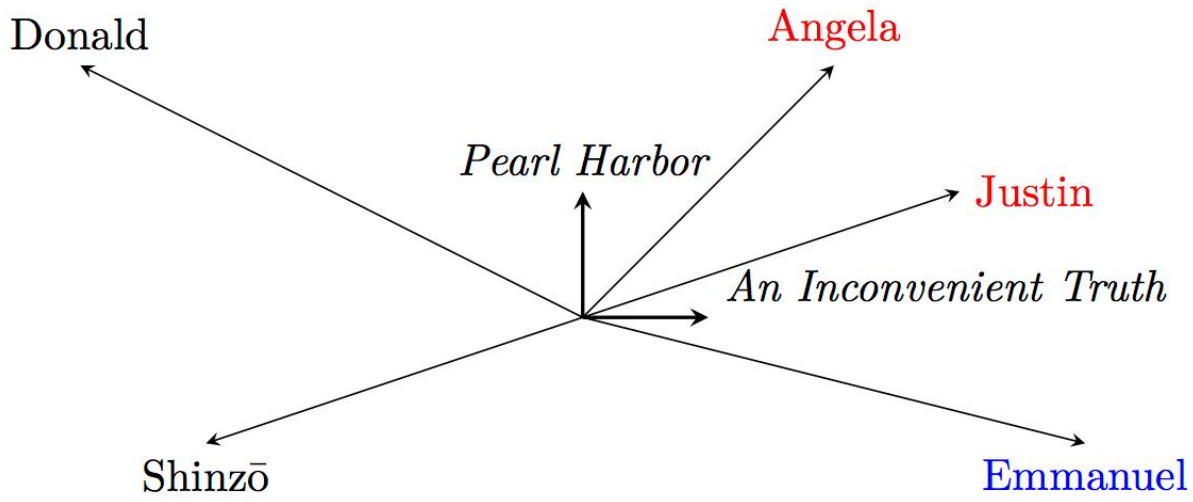

Figure 2. Les 2 plus proches voisins d'Emmanuel pour la similarité cosinus sont Justin et Angela, en supposant que tout le monde n'ait noté que *Pearl Harbor* et *An Inconvenient Truth*. Donald est très peu corrélé à Emmanuel : leur similarité cosinus est presque –1.

Predict. Pour une paire `user_id`-`work_id` dont il faut prédire la note, il ne reste plus qu'à identifier les utilisateurs ayant noté l'œuvre `work_id`, les trier par similarité décroissante à `user_id`, ne conserver que les `nb_neighbors` premiers (`closest_raters[:self.nb_neighbors]`) et enfin calculer la moyenne des valeurs correspondantes dans la matrice `ratings`, cf. Figure 3.

Notes	*Paprika*	*Pearl Harbor*	*An Inconvenient Truth*
Justin	3	1	3
Angela	?	2	2
Donald	−3	2	−4
Emmanuel	?	−1	4
Shinzō	4	−1	−3

Similarité	Justin	Angela	Donald	Emmanuel	Shinzo
Justin	1	0,649	−0,809	0,612	0,090
Angela	0,649	1	−0,263	0,514	−0,555
Donald	−0,809	−0,263	1	−0,811	−0,073
Emmanuel	0,612	0,514	−0,811	1	−0,523
Shinzō	0,090	−0,555	−0,073	−0,523	1

Notes	*Paprika*	*Pearl Harbor*	*An Inconvenient Truth*
Justin	3	1	3
Angela	?	2	2
Donald	−3	2	−4
Emmanuel	3,5	−1	4
Shinzō	4	−1	−3

Figure 3. Emmanuel devrait-il regarder *Paprika* ? C'est la question sur toutes les lèvres. Si l'on exécute l'algorithme des 2 plus proches voisins sur (Emmanuel, Paprika), les plus proches voisins d'Emmanuel sont Justin et Angela, mais comme Angela n'a pas vu Paprika, ses 2 plus proches voisins ayant vu *Paprika* sont Justin et Shinzō. Leur note moyenne est (3 + 4) / 2 = 3,5, qui est renvoyé comme prédiction.

Une technique plus performante (SGD ou ALS, complétion de matrice)

Voici une autre méthode pour prédire les notes manquantes, basée sur de la factorisation de matrices et **l'algorithme de descente de gradient**.

L'idée : si je peux trouver une représentation *U* de mes utilisateurs et *V* de mes films sous la forme d'un vecteur par entité, de façon que la note que le user *i* donne à l'œuvre *j* soit le produit scalaire de U_i avec V_j, alors ça veut dire que je cherche à écrire :

$R = UV$

Où :

- R est la matrice (N utilisateurs × M films) des notes ;
- U la représentation des utilisateurs (N utilisateurs × D composantes) ;
- V la représentation de films (M films × D composantes).

Ainsi, j'observe une partie infime des ratings R (1 %, on vous a dit), et je cherche à identifier U et V. Une fois que j'ai U et V, il m'est facile de généraliser à des nouvelles paires utilisateur i et œuvre j car je n'ai plus qu'à calculer $U_i \cdot V_j$.

ALS. Une méthode pour trouver U et V à partir de R s'appelle **l'algorithme des moindres carrés alternés** (ALS, *alternating least squares*) : initialiser U et V au hasard, puis fixer U et optimiser V, puis fixer V et optimiser U, etc. jusqu'à une convergence, qui survient après quelques dizaines d'itérations sur les jeux de données que nous avons testés. En quelques secondes sur un MacBook 1,3 GHz, ALS peut apprendre U et V pour une matrice R de 2000 utilisateurs, 10000 films, 330000 ratings, et c'est le meilleur algorithme que nous ayons à Mangaki ne s'appuyant que sur les ratings (si on y ajoute les posters, on peut faire mieux, cf. ci-dessous). Le code de ALS est disponible sur http://research.mangaki.fr.

SGD. Mais dans cet article, par souci de concision et de généralisation, nous avons préféré présenter **l'algorithme de descente de gradient stochastique** (SGD, *stochastic gradient descent*). Le principe : on cherche à minimiser une certaine fonction d'erreur. Du coup, si on dérive la fonction d'erreur par rapport aux paramètres (ici, U et V), on obtient un gradient, qui est un vecteur ayant pour dimension le nombre de paramètres. En faisant un petit pas dans la direction opposée au gradient, on a de bonnes chances de baisser l'erreur.

Attention, des maths ! Fuyez ! Lorsqu'on observe une nouvelle note R_{ij} de l'utilisateur i à l'œuvre j, on tente d'améliorer notre prédiction $U_i \cdot V_j$. L'erreur relative de reconstruction (de combien on s'est trompé) e_{ij} est $U_i \cdot V_j - R_{ij}$, et comme on souhaite minimiser la RMSE (la racine carrée de la moyenne des carrés des erreurs), cela revient à minimiser e_{ij}^2, ce qui équivaut à minimiser $e_{ij}^2/2$, pour ne pas s'embêter avec un facteur 2.
Or, la quantité correspondant à la dérivée de $e_{ij}^2/2$ par rapport à U_i est en fait simplement le vecteur $e_{ij} V_j$. C'est le **gradient**.
Donc la mise à jour à faire est : $U_i \leftarrow U_i - \gamma e_{ij} V_j$ (avancer d'un petit pas γ dans la direction **opposée** au gradient pour baisser l'erreur).
Similairement, la mise à jour de V_j est : $V_j \leftarrow V_j - \gamma e_{ij} U_i$.

Malheureusement, si vous implémentez l'algorithme ainsi, votre erreur baissera effectivement sur le jeu d'entraînement, mais pas sur le jeu de test : vous ne parviendrez pas à bien prédire de nouvelles paires. C'est du **surapprentissage**. Tan tan tan. Il faut donc choisir une autre fonction objectif à minimiser que directement la fonction de perte (cf. la section *Évaluation des performances*).

Ajouter de la régularisation. Si au lieu de minimiser seulement l'erreur de reconstruction au carré, on ajoute une valeur de régularisation, la quantité à minimiser devient :
$\sum e_{ij}^2/2 + \lambda (\sum ||U_i||^2 + \sum ||V_j||^2)/2$.

Où λ est un paramètre de régularisation, aussi parfois appelé « hyperparamètre à la con ». Les /2 sont toujours là pour faire joli.
L'algorithme qui en découle est laissé en exercice au lecteur (au cas où vous n'auriez pas remarqué jusque-là, j'ai toujours rêvé d'être prof de maths)… Non, en fait, voici :

Pseudo-code. Initialiser tous les U_i et V_j au hasard.
Itérer plusieurs fois :
Pour tout couple (i, j) de X, avec la note R_{ij} correspondante dans y :
$U_i \leftarrow U_i - \gamma\ (e_{ij}\ V_j + \lambda\ U_i)$
$V_j \leftarrow \text{Vi} - \gamma\ (e_{ij}\ U_i + \lambda\ V_j)$

Le code correspondant :

```
import numpy as np
from sklearn.metrics import mean_squared_error


class MangakiSGD:
    def __init__(self, nb_users, nb_works, nb_components=20,
nb_iterations=10,
                 gamma=0.01, lambda_=0.1):
        self.nb_components = nb_components
        self.nb_iterations = nb_iterations
        self.nb_users = nb_users
        self.nb_works = nb_works
        self.gamma = gamma
        self.lambda_ = lambda_
        # self.bias = np.random.random()
        # self.bias_u = np.random.random(self.nb_users)
        # self.bias_v = np.random.random(self.nb_works)
        self.U = np.random.random((self.nb_users,
self.nb_components))
        self.V = np.random.random((self.nb_works,
self.nb_components))

    def fit(self, X, y):
        for epoch in range(self.nb_iterations):
            step = 0
            for (i, j), rating in zip(X, y):
                if step % 100000 == 0:  # Pour afficher l'erreur
de train
                    y_pred = self.predict(X)
                    print('Train RMSE (epoch={}, step={}):
%f'.format(
                        epoch, step, mean_squared_error(y, y_pred)
** 0.5))
```

```
                predicted_rating = self.predict_one(i, j)
                error = predicted_rating - rating
                # self.bias += self.gamma * error
                # self.bias_u[i] -= self.gamma * (error +
                #                                 self.lambda_ *
self.bias_u[i])
                # self.bias_v[j] -= self.gamma * (error +
                #                                 self.lambda_ *
self.bias_v[j])
                self.U[i] -= self.gamma * (error * self.V[j] +
                                           self.lambda_ *
self.U[i])
                self.V[j] -= self.gamma * (error * self.U[i] +
                                           self.lambda_ *
self.V[j])
                step += 1

    def predict_one(self, i, j):
        return (  # self.bias + self.bias_u[i] + self.bias_v[j] +
                self.U[i].dot(self.V[j]))

    def predict(self, X):
        y = []
        for i, j in X:
            y.append(self.predict_one(i, j))
        return np.array(y)
```

Choisir les hyperparamètres. Si λ = 0, on surapprend. Si γ est trop petit, on apprend trop lentement. Si γ est trop grand, l'erreur oscille dans tous les sens. Nous conseillons γ = 0,01 et λ = 0,1 sur le jeu de données de Mangaki et celui de Movielens, mais encore une fois, cela dépendra du jeu de données.

Rendre l'algorithme stochastique. Pour obtenir l'algorithme du gradient stochastique, il suffit de permuter la liste des exemples avant de la traverser, et donc parcourir la paire `zip(X[indices], y[indices])` où indices est une permutation aléatoire de `list(range(len(X)))`. Cela permet d'éviter de tomber dans un cycle qui empêcherait de converger.

Ajouter des biais. Il est possible d'enrichir le modèle avec un paramètre de biais *b* ajouté à tous les ratings, ainsi qu'un biais bu_i pour chaque utilisateur *i* et un biais bv_j pour chaque œuvre *j*. La prédiction pour le couple utilisateur *i* œuvre *j* n'est plus seulement $U_i \cdot V_j$ mais $b + bu_i + bv_j + U_i \cdot V_j$, et tous les paramètres sont mis à jour par descente de gradient. Testez le code ci-dessus en retirant les commentaires pour voir l'impact de ces nouveaux paramètres sur l'apprentissage. Essayez également de voir ce qui se passe lorsque λ vaut 0 (c'est-à-dire, pas de régularisation).

Visualisation. Ce qui est particulièrement intéressant avec ce modèle, c'est de pouvoir visualiser ce qu'il a appris. Si l'on affiche les deux premières composantes des vecteurs V_j, on obtient ce qui est à la Figure 4. On peut également afficher les composantes des vecteurs des utilisateurs, et ils aimeront les films qui sont dans la même direction. Cela permet entre autres de découvrir les profils qui séparent le plus les utilisateurs.

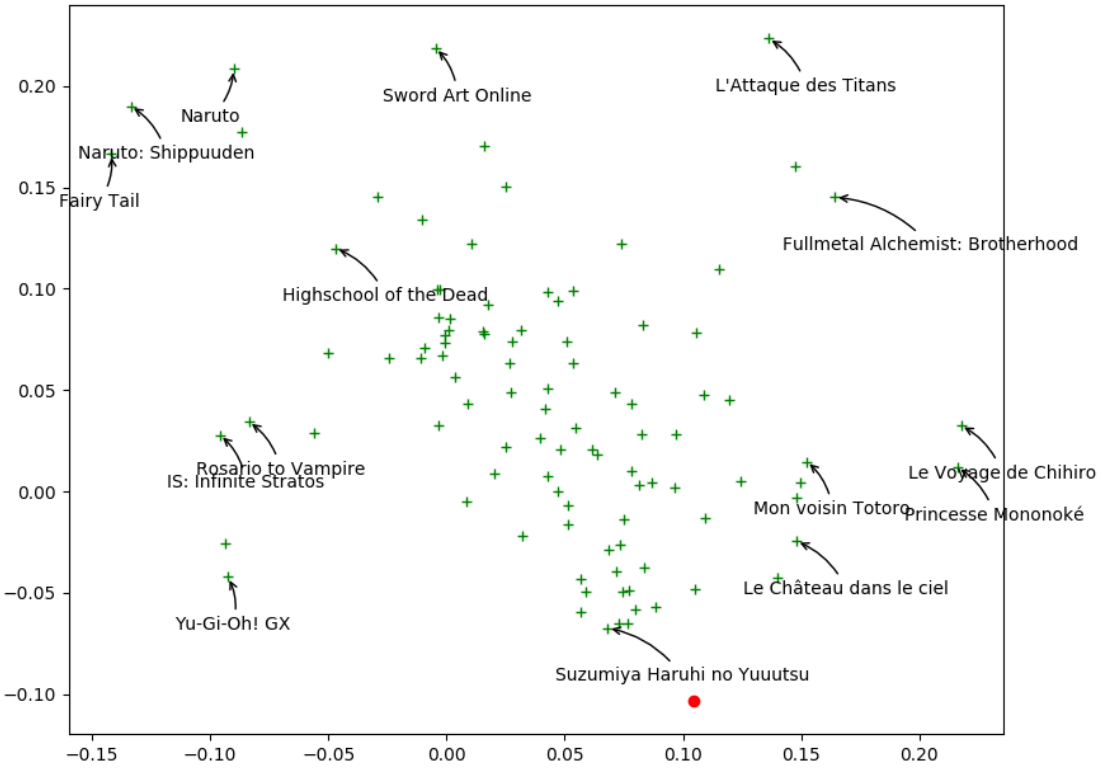

Figure 4. Les deux premières composantes des représentations des œuvres V_j pour les 100 œuvres les plus populaires sur Mangaki. Le point rouge correspond aux deux premières composantes du vecteur U_i où *i* désigne l'un des auteurs de cet article. Il aime *Haruhi Suzumiya* et *Princesse Mononoké* mais pas *Fairy Tail* ou *Naruto*.

Utiliser les posters pour améliorer les recommandations (deep learning)

Le problème principal avec la méthode que nous venons de voir est que lorsqu'une nouvelle œuvre vient de sortir, on ne dispose pas encore de notes (problème de *cold-start*), et donc on ne peut pas calculer de représentation correspondante.

Nous allons maintenant présenter une méthode qui illustre que le machine learning permet de faire des inférences à partir de données d'origine variée ! En se servant des posters, on peut améliorer les recommandations.

Illustration2Vec est un réseau de neurones convolutionnel (CNN, *convolutional neural network*) développé par Saito et Matsui en 2015 qui permet d'extraire des tags à partir d'illustrations (principalement de manga). Vous lui donnez un poster en entrée, et il vous renvoie une liste de tags qu'il reconnaît sur les photos, avec des valeurs de confiance, cf. Figure 5 pour un exemple. Ce CNN a été entraîné sur 1,5 millions d'illustrations du site communautaire Danbooru, qui contient énormément de tags, à partir d'un VGG-16 (type de CNN) déjà pré-entraîné sur ImageNet. Les 502 tags les plus fréquents ont été conservés.

Popularisation du deep learning

« *En 2005, nous étions sans doute 3–4 groupes dans le monde à continuer à croire aux réseaux de neurones artificiels* » — Yoshua Bengio
2007. Le trio **Hinton-LeCun-Bengio** organise à Vancouver un workshop non officiel de deep learning concurrent à NIPS 2007.
2012. Une équipe de Krizhevsky, Sutskever & Hinton remporte le challenge de **reconnaissance d'images** ImageNet 2012 en réalisant un taux d'erreur de seulement 15,3 % avec un max-pooling CNN (la 2e meilleure équipe était à 26,2 %).
2014. Goodfellow et al. (dont Courville et Bengio) inventent les *Generative Adversarial Networks* (GANs), capables de **générer des photographies réalistes** pour des observateurs humains.
Mars 2016. AlphaGo de DeepMind **bat le champion du monde de go** Lee Sedol 4 à 1 avec des CNN couplés à la méthode de *Monte Carlo Tree Search*.
Septembre 2016. WaveNet de DeepMind permet de **synthétiser la parole** de façon réaliste (par exemple, faire dire un texte à une voix à partir d'échantillons).

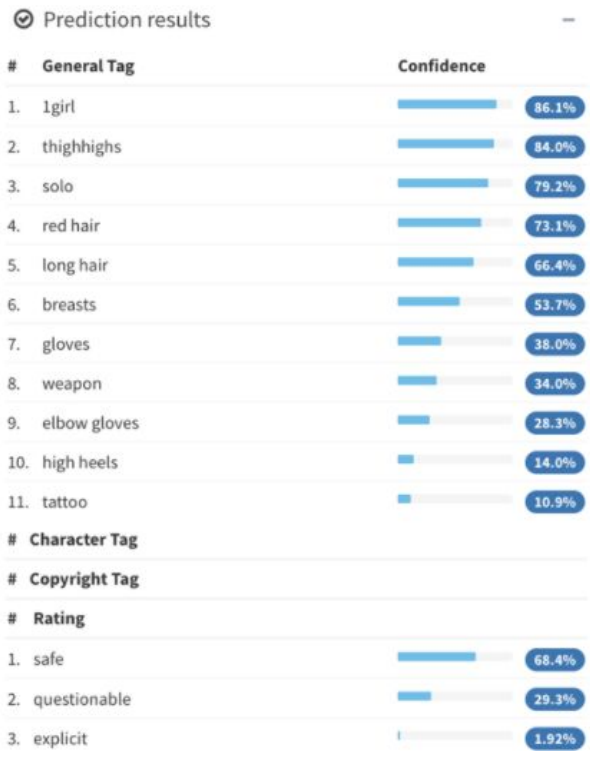

Figure 5. Un exemple de sortie d'Illustration2Vec sur une illustration de manga, à gauche. Le vecteur de tags correspondant à cette image a toutes ses composantes à 0 sauf *1girl* à 0,861 (confiance 86,1 %), *red hair* à 0,731, etc.

En passant tous nos posters à ce modèle, déjà entraîné sur ImageNet et librement téléchargeable sur http://illustration2vec.net, on obtient une matrice T de taille M œuvres × 502 tags telle que pour toute œuvre j et tout tag k, $T_{jk} \geq 0$ vaut une valeur d'autant plus grande que le tag k semble apparaître dans l'œuvre j. Montrons à présent comment s'en servir pour faire des recommandations.

Comme pour chaque utilisateur, on sait quelles œuvres il (ou elle) a aimées / pas aimées et les tags des posters correspondants, on peut identifier ses tags préférés. On modélise les préférences de l'utilisateur i par un vecteur P_i, paramètre à apprendre, et on prédit la note qu'il va donner à l'œuvre j par la valeur $P_i \cdot T_j$. Intuitivement, pour chaque tag k, on a :

- Si $P_{ik} = 0$, alors l'utilisateur i se fiche du tag k : ça n'a pas d'incidence sur ses notes.
- Si $P_{ik} > 0$, alors les œuvres impliquant le tag k ont un bonus dans les notes de l'utilisateur i (car toutes les entrées T_{jk} sont positives).
- Si $P_{ik} < 0$, les œuvres impliquant le tag k ont un malus dans les notes de l'utilisateur i.

C'est exactement comme le modèle précédent, sauf qu'on connaît déjà la représentation des œuvres ($V_j = T_j$, $U_i = P_i$). On veut pouvoir interpréter les goûts des utilisateurs, donc on ajoute la contrainte supplémentaire que P_i doit avoir beaucoup de composantes à 0. Le problème que l'on cherche à résoudre, trouver P_i sachant T s'appelle une régression linéaire creuse (*sparse linear regression*), déjà implémentée dans scikit-learn sous le nom LASSO (*Least Absolute Shrinkage and Selection Operator*) :

```
from sklearn.linear_model import Lasso
```

En reprenant la structure vue à l'algorithme `MangakiKNN` (cf. section *Un algorithme naïf*), si `self.ratings_by_user` est une matrice SciPy qui contient les œuvres vues par chaque utilisateur, on n'a qu'à entraîner un LASSO par utilisateur sur les lignes de la matrice de tags correspondant aux œuvres notées par cet utilisateur.

```
import numpy as np
from scipy.sparse import coo_matrix
from sklearn.linear_model import Lasso


class MangakiLASSO:
    def __init__(self, nb_users, nb_works, T):
        self.nb_users = nb_users
        self.nb_works = nb_works
        self.T = T

    def fit(self, X, y):
        user_ids = X[:, 0]
        work_ids = X[:, 1]
```

```
        self.ratings = coo_matrix((y, (user_ids, work_ids)),
                                  shape=(self.nb_users,
self.nb_works))
        self.ratings_by_user = self.ratings.tocsr()
        self.lasso = {}
        for user_id in range(self.nb_users):
            rated_work_ids = self.ratings_by_user[user_id].indices
            user_ratings = self.ratings_by_user[user_id].data
            if len(rated_work_ids):
                self.lasso[user_id] = Lasso(alpha=0.01,
fit_intercept=True)
                self.lasso[user_id].fit(self.T[rated_work_ids],
user_ratings)

    def predict(self, X):
        y = []
        for user_id, work_id in X:
            if user_id in self.lasso:
                y.append(self.lasso[user_id]
                            .predict(self.T[work_id].reshape(1,
-1)))
        return np.array(y)
```

L'avantage, c'est que lorsqu'un modèle LASSO a été entraîné, on peut consulter les coefficients P_i appris en faisant `self.lasso[i].coef_`, cf. **Figure 6**. On peut également expliquer aux utilisateurs : « *On vous recommande ce film parce que vous semblez aimer les personnages avec lunettes, mais pas celui-là parce que vous n'aimez pas les kimonos.* »

```
20 petals 0.062048928418
22 butterfly -0.00575388224009
27 necktie -0.0441011232299
29 kimono 0.0108843921645
33 table 0.0609460759477
48 instrument -0.00326064812932
60 from behind -0.00583659841878
69 traditional media 0.0147549217731
73 magical girl 0.0312992390526
76 younger -0.00337236166619
92 child -0.010405530536
102 3boys -0.0148483643512
130 blonde hair -0.0562550649455
146 dutch angle 0.00639426176897
180 white hair -0.0311996357131
228 mole -0.00851666561641
229 blue hair -0.0126566933755
244 zettai ryouiki -0.00325367435046
282 backpack -0.0220788327809
284 twin drills 0.0149344607109
369 facial mark -0.0585771795163
416 short twintails 0.00372983158992
451 sky 0.0121431534467
476 frog 0.039292267754
```

Figure 6. Un exemple de vecteur P_i appris (seules les composantes non nulles sont représentées, par leur indice k, le nom du tag correspondant et la valeur P_{ik}). Cet utilisateur semble aimer les *magical girls*, les… *tables* mais pas les *facial marks* et les *neckties*.

Donc quel est le meilleur modèle ? Réponse section suivante.

Combiner des modèles

Lorsque vous avez plusieurs modèles pour prédire les notes, lequel choisir ? Réponse : tous ! En théorie, combiner des modèles permet de pallier les limitations de chacun en s'appuyant sur les prédictions des autres. En pratique, ce qui risque de se passer, c'est d'obtenir un monstre qui surapprend vos données (*overfitting*).

Model stacking, ensemble methods

Les méthodes usuelles consistent à soit combiner plein de modèles très simples et peu performants pour obtenir un meilleur modèle (*boosting*), soit combiner plein de modèles complexes mais qui surapprennent pour obtenir un modèle qui généralise mieux (*bagging*). En fait, tout le champ lexical y passe : *blending*, *stacking*, *averaging*, etc. En particulier, **xgboost** (*extreme gradient boosting trees*) et **LightGBM** (*gradient boosting machine*) ont rapidement gagné en popularité, étant utilisés par la plupart des gagnants aux compétitions de sciences des données de type Kaggle.

Une manière simple de combiner 2 modèles consiste à considérer la moyenne de leurs prédictions, éventuellement pondérée davantage d'un côté ou de l'autre. Si je combine k modèles faisant des prédictions y_1, …, y_k, je peux apprendre les coefficients p_1, …, p_k d'une combinaison linéaire $y = p_1y_1 + \ldots + p_ky_k$.

BALSE. À Mangaki, le modèle réalisant la RMSE la plus faible est une combinaison non linéaire de ALS (la complétion de matrice vue plus haut s'appuyant sur les notes seulement) et LASSO, qui s'appuie sur les posters et les notes. Nous l'avons appelé *Blended Alternate Least Squares with Explanation* (BALSE), en hommage à un certain anime que le lecteur (la lectrice) reconnaîtra peut-être. Sinon il (elle) peut toujours chercher sur Google.

Pour aller plus loin

Techniques

Algorithmes en ligne. L'avantage d'un modèle comme l'algorithme du gradient stochastique, c'est qu'on met à jour les paramètres au fur et à mesure qu'on observe des notes. Cette approche est adaptée lorsqu'on a des grands volumes de données qu'on observe petit à petit : pas besoin de devoir réentraîner tout le modèle sur toutes les notes à chaque nouvelle note. Il existe de nombreuses variantes de cet algorithme que vous pouvez consulter à l'adresse suivante :
http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

Méthodes à noyaux. Pour déterminer les plus proches voisins, on peut choisir une autre fonction de similarité que le cosinus. En voici une liste (le noyau gaussien RBF est populaire, par exemple) :
http://scikit-learn.org/stable/modules/metrics.html

TensorFlow. L'avantage principal de ce framework, c'est qu'il permet de tester une multitude d'algorithmes d'apprentissage de type SGD (cf. *Une technique plus performante*). On a juste à formuler la fonction objectif, et il calcule les dérivées tout seul pour faire les mises à jour des paramètres. Ce que nous vous conseillons, c'est de prendre un algorithme que vous connaissez (régression linéaire, analyse de composantes principales (PCA), etc.), et d'essayer de le réimplémenter en TensorFlow. Recherchez par exemple PCA in TensorFlow, ou le modèle de complétion de matrice décrit dans cet article mais entraîné avec Adam, une variante de SGD : https://github.com/songgc/TF-recomm
Découvrez les super exemples de Aymeric Damien :
https://github.com/aymericdamien/TensorFlow-Examples

Concepts

Feedback implicite. Parfois il est difficile d'obtenir des notes de la part des utilisateurs. Alors soit on essaie d'extraire des informations sur les œuvres, soit on cherche des signaux faibles comme : « Cette personne est restée longtemps sur cette page », qui sont d'autres

manières pour l'utilisateur d'exprimer son intérêt. À Mangaki, on préfère la première méthode (s'appuyer sur les posters), moins intrusive !

Zone de confort. Ce que l'on reproche souvent aux systèmes de recommandation, c'est de laisser les utilisateurs dans leur « zone de confort », où ils ne consomment que des valeurs sûres, ce qui laisse moins de place au risque et à la diversité. Heureusement il existe aussi des façons de modéliser la diversité pour pouvoir présenter des œuvres variées. Encore une fois, tout dépend de la fonction que l'on cherche à optimiser. Nous estimons qu'il s'agit d'un faux problème, lié à une mauvaise conception du système qui résulte en une mauvaise expérience utilisateur, contrairement au vrai problème qui arrive dans le paragraphe suivant.

Discrimination involontaire. Un modèle de machine learning reproduit les biais des données avec lesquelles il a été entraîné. Si 90 % des utilisateurs sur Mangaki sont des garçons, les filles risquent de ne pas être satisfaites par leurs recommandations, et on risque de ne pas s'en apercevoir car elles ne contribueront qu'à 10 % de l'erreur. Pour pouvoir y remédier, il faut d'abord savoir qui est une fille, ce qui pose des problèmes de confidentialité. Ceci est un domaine actif de recherche, voir l'émission *La Faute à l'algo* http://fautealgo.fr/revue-de-presse/ et le workshop *Fairness, Accountability, and Transparency in Machine Learning* (FATML) qui est devenu une conférence : https://fatml.org https://fatconference.org

Conclusion

Ce vaste article est à présent terminé ! N'hésitez pas à tester Mangaki.fr, rejouer les codes présentés ici et contribuer à une *good first issue*, absolument tout est sur GitHub : https://github.com/mangaki/mangaki
Garanti sans polyphosphates. Toutes vos remarques seront les bienvenues :-)

Pour se perfectionner en machine learning, rien de tel que le *project-based learning*. Choisissez un problème, un jeu de données, et c'est parti ! Il existe une pléthore d'open data sur le site Kaggle (ou même, https://data.gouv.fr, ou le site de l'OCDE, ou de la NASA), avec des notebooks pour reproduire les expériences.