Catégorisez automatiquement des questions

-

Rapport de Projet

I. Présentation des données et de la problématique

II. Statistiques sur les tags associées aux questions

III. Nettoyage, tokenisation et vectorisation du corpus de questions

IV. Prédiction de tags : approche non supervisée

V. Prédiction de tags : approche supervisée

I. Présentation des données et de la problématique.

Stack Overflow est un site de questions-réponses liées au développement informatique. A chaque question posée sur le site sont associés un ou plusieurs tags qui décrivent les différents thèmes de la question. L'objectif du présent projet est le suivant : développer un système de suggestion de tags lorsque l'utilisateur pose une question. C'est-à-dire développer un modèle qui à partir d'une question posée puisse prédise un certain nombre de tags adéquats. Pour construire le modèle nous utilisons un corpus d'environ 50000 questions récupérées sur le site de Stack Overflow, ainsi que la liste des tags associés à ces questions.

Nous allons utiliser 2 approches différentes. Une approche non supervisée ou seules les questions permettent de dégager un certain nombre de thèmes ainsi que les mots clés associés à ces thèmes.

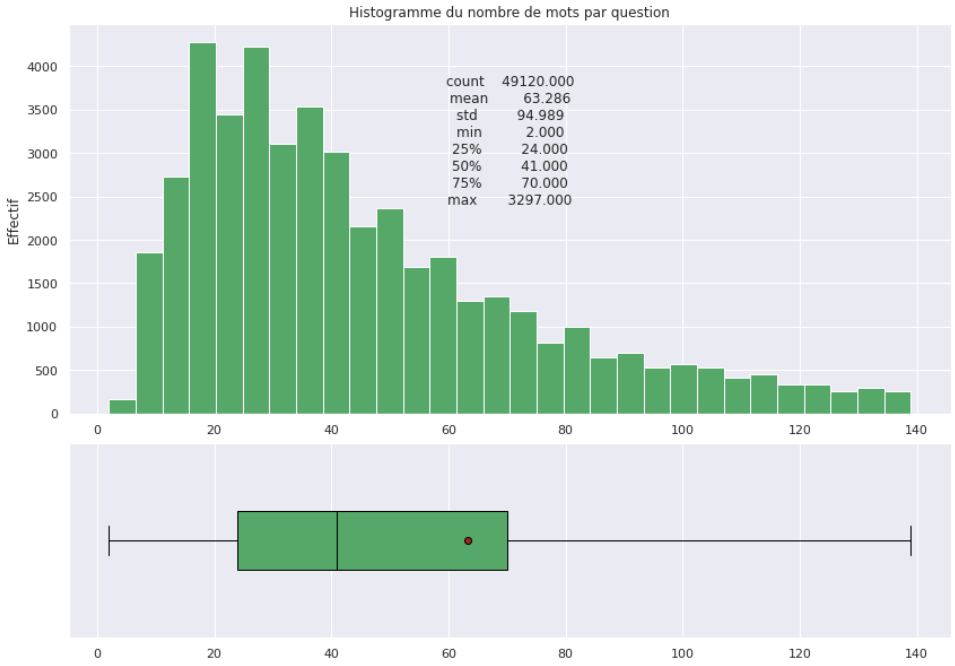
Une approche supervisée où nous utilisons les tags associés aux questions en tant que cible pour pouvoir effectuer une classification multi-labels des questions. Après avoir entrainé le modèle, il est capable de prédire les tags associés à de nouvelles questions.

Pour commencer nous donnons quelques statistiques sur le corpus et les tags récupérés, puis nous décrivons le nettoyage et la tokenisation du corpus de questions, ainsi que la vectorisation des données. Enfin nous décrivons les modèles avec les approches non supervisée et supervisée.

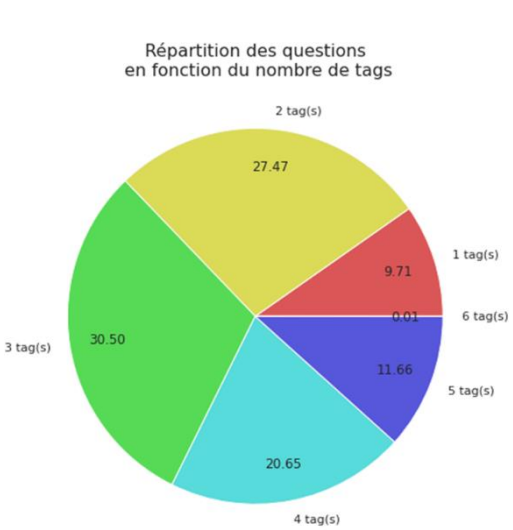
II. Quelques statistiques sur le corpus de question et les tags associés

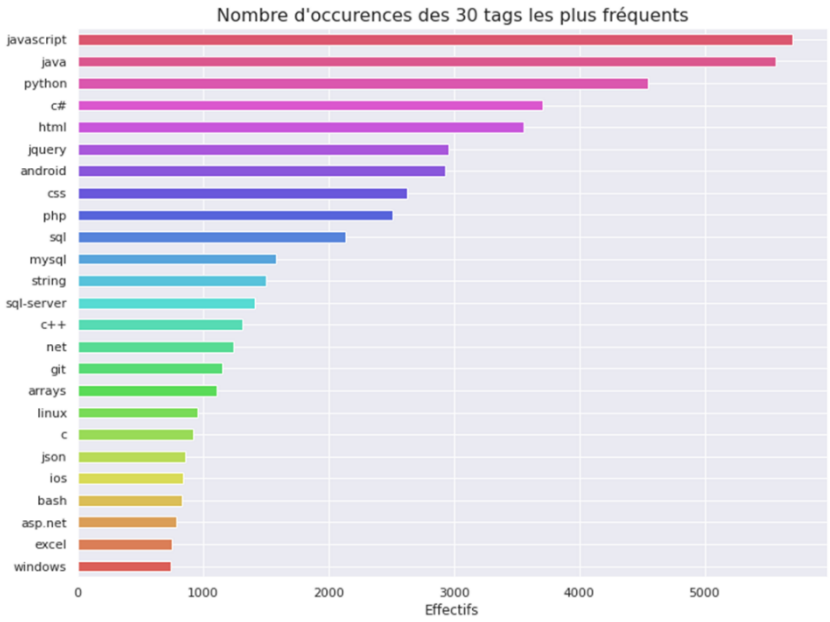
Pour commencer précisons qu'une question sur Stack Overflow est constituée d'un titre et d'un corps, que nous avons concaténé par commodité.

Voici l'histogramme et la boite à moustaches du nombre de mots par questions. Le corpus est constitué de 49120 questions et une question contient en moyenne 63 mots.



Il y a en tout 8637 tags uniques et chaque question du corpus possède de 1 à 6 tags, voici le camembert donnant la répartition des questions en fonctions du nombre de tags, ainsi que le diagramme en barre des tags les plus fréquents.





Pour avoir une vue d'ensemble des fréquences des différents tags, nous pouvons regarder la proportion de tags couverts lorsqu'on sélectionne un nombre de tags parmi les plus fréquents (en abscisse sur le graphique suivant). Sur le même graphique nous affichons également la proportion de questions couvertes par au moins un tag et la proportion de questions dont tous les tags sont couverts.

Par exemple si l'on sélectionne 10 % des tags les plus fréquents, ces 860 tags couvrent plus de 80 % de l'ensemble des tags comptés avec multiplicité, 60 % des questions ont tous leurs tags couverts et plus de 99% des questions possèdent au moins un tag couvert.

III. Nettoyage, tokenisation et vectorisation du corpus de questions

Avant de vectoriser le corpus, il faut d'abord nettoyer puis transformer les questions en une liste de mots, c'est l'étape de la tokenisation.

Voici les différentes étapes effectuées :

- Suppression des balises HTML.
- Tokenisation avec une expression régulière : regex = '[A-Za-z][A-Za-z0-9_+\-#]*'.
- Suppression des « stopwords » anglais basiques auxquels on ajoute quelques mots les plus fréquents du corpus, soit environ 200 mots en tous.
- Racinisation des mots, pour ne garder que le sens du mot, et réduire le nombre de mots. Exemple : expression --> express.
- La suppression des mots peu fréquents est réalisée par l'algorithme de vectorisation.

Après ces étapes, chaque question est décrite par la liste de ces mots racinisés, et qui n'ont pas été supprimés. On peut passer à la vectorisation des données, c'est-à-dire la transformation de chaque question en un vecteur.

Nous avons utilisé deux types de vectorisation : le BOW (Bag Of Words) et le TF-IDF (Term Frequency – Inverse Document Fraquency), implémentées par les fonctions CountVectorizer et TfidfVectorizer de scikit-learn.

Ces fonctions commencent par supprimer les mots peu fréquents grâce au paramètre « min_df ». Si par exemple min_df = 10, les mots qui apparaissent moins de 10 fois dans le corpus sont supprimés. Cela permet de réduire la taille des données et de ne garder que l'information essentielle (cf graphique qui suit).

Chaque question du corpus est ensuite codée par un vecteur dont chaque composante représente un mot unique du corpus. Pour le Bag of Words, la valeur d'une composante est égale au nombre de fois que le mot apparait dans la question. Pour le TF-IDF, on multiplie ce nombre d'occurrence d'un mot par l'inverse de la fréquence du mot sur tout le corpus, ceci pour donner plus de poids aux mots qui caractérisent le mieux la question, et qui sont globalement moins présents dans le corpus.

La matrice obtenue une fois le corpus vectorisé est une matrice creuse, qui contient une grande majorité de 0. Les algorithmes qui gèrent ce type de matrice le font différemment, pour des questions de complexité spatiale et temporelle.

Voici un exemple de vectorisation d'une question, en « BOW », puis en « TF-IDF ».

Question initiale :
Recommended Fonts for Programming? What fonts do you use for programming, and for what language/IDE? I use Consolas for all my Visual Studio work, any other recommendations?

Liste des mots après cleaning :
['recommend', 'font', 'program', 'font', 'program', 'languag', 'ide', 'consola', 'visual', 'studio', 'ani', 'recommend']

Vectorisation BOW :
[('recommend', 2), ('font', 2), ('program', 2), ('languag', 1), ('ide', 1), ('visual', 1), ('studio', 1), ('ani', 1), ('consola', 0)]

Vectorisation tfidf :
[('font', 0.55), ('recommend', 0.54), ('program', 0.37), ('ide', 0.28), ('languag', 0.24), ('visual', 0.23), ('studio', 0.22), ('ani', 012), ('consola', 0)]

Le mot consola est supprimé, car peu présent dans le corpus.

Voici le graphique qui nous donne le nombre de mots conservés en fonction du paramètre « min_df ».

IV. Approche non supervisée

Pour l'approche non supervisée, nous utilisons l'algorithme Latent Dirichlet Allocation (LDA). Cet algorithme se base sur les hypothèses suivantes :

- Chaque question aborde un certain nombre de thèmes dans différentes proportions.
- Chaque mot possède une distribution associée à chaque thème.

On choisit le nombre de thèmes (topics) et l'algorithme nous permet de décrire chaque thème par l'ensemble des mots qui le caractérise le plus.

Voici par exemple ce que donne la LDA sur notre corpus lorsqu'on choisit 20 thèmes et que l'on affiche les 10 premiers mots qui caractérisent le mieux chaque thème. On utilise ici une vectorisation « TF-IDF » avec min_df = 20.

```
topic_0                         [http, server, request, connect, get, url, com, file, php, user]
topic_1              [tab, video, play, typescript, html5, observ, control, filter, loop, mistak]
topic_2                     [date, time, file, differ, format, get, datetim, ani, doe, question]
topic_3                     [android, devic, phone, app, sdk, emul, iphon, apk, develop, eclips]
topic_4                  [break, case, xcode, categori, switch, simul, xyz, author, iphon, pool]
topic_5                [git, branch, commit, repositori, push, master, github, remot, merg, plot]
topic_6                   [power, exec, download, file, excel, open, php, distribut, write, link]
topic_7               [selenium, wpf, wait, millisecond, grid, sleep, utc, temp, task, implement]
topic_8                   [tabl, sql, select, queri, mysql, databas, column, server, insert, row]
topic_9                               [div, imag, px, css, width, color, height, td, style, li]
topic_10         [charact, regex, express, match, regular, space, vim, firstnam, pattern, letter]
topic_11                 [self, swift, anim, nil, func, objective-c, super, xcode, section, hide]
topic_12                       [string, array, list, int, object, number, return, convert, b, x]
topic_13                    [echo, bash, php, mongodb, script, perl, this-, dump, easiest, team]
topic_14             [batch, cmd, kill, sleep, command, window, player, dictionari, renam, delay]
topic_15                    [cell, excel, vba, rang, sub, dim, workbook, a1, worksheet, formula]
topic_16               [file, instal, directori, java, command, path, python, project, folder, c]
topic_17              [jqueri, function, input, button, javascript, form, click, html, div, text]
topic_18                    [zoom, xhtml, style, properti, rest, page, none, android, web, catch]
topic_19    [android, activ, view, button, layout, textview, drawabl, edittext, listview, intent]
```

Pour chaque question du corpus, l'algorithme retourne la distribution de probabilité sur chaque thème. Si par exemple la distribution du topic 0 sur notre question vaut 0.56, on peut choisir de retourner les 5 premiers mots du topic 0, qui serviront de tags pour la question.

Exemple :

Question : *Remove last commit from remote git repository \n Possible Duplicate:\n Rolling back local and …*

Distributions des thèmes : topic_5 : 0.76, les autres inférieurs à 0.1.

Tags prédits : [git, branch, commit, repositori, push, master, github]

Cette approche présente quelques inconvénients :

- Les thèmes ne sont pas tous pertinents.
- Certains thèmes qui ressortent ne sont pas propres à un langage informatique particulier.
- Beaucoup de thèmes qui nous intéressent ne sont pas présents.
- Les mots qui ressortent en premier ne sont pas toujours les plus pertinents.

Si l'on veut prédire des tags pertinents, il est beaucoup plus aisé d'utiliser une approche supervisée. L'approche non supervisée a néanmoins le mérite de faire ressortir quelques grands thèmes du corpus, uniquement grâce à son vocabulaire.

V. Approche supervisée

L'approche supervisée consiste à entrainer un modèle qui prédise un ensemble de tags lorsqu'on lui donne de nouvelles questions. Il s'agit d'une classification multi-labels : pour chaque nouvelle question et chaque tag existant, on associe ou non le tag à la question. A chaque nouvelle question est donc assigné 0, 1 ou plusieurs tags.

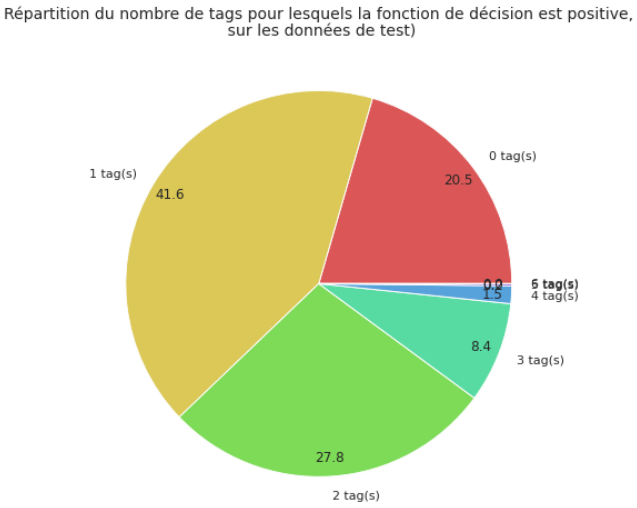
Pour évaluer la qualité de notre modèle et choisir les meilleurs paramètres, on sépare les données en données d'entrainement et de test (75 – 25%), et on regarde les métriques moyennes sur les données de test. Le choix final des paramètres à optimiser est effectué par validation croisée sur les données d'entrainement.

Le modèle comprend :

- La vectorisation du corpus de question que l'on a vu en partie 3, avec comme paramètres importants le type de vectorisation (BOW ou TF-IDF) et la borne min_df.
- Les tags cibles sont codés en one hot encoding grâce à la fonction MultiLabelBinarizer de scikit-learn.
- On entraine un classifieur multi-labels, c'est-à-dire un classifieur binaire pour chaque tag sélectionné. Pour cela on utilise la fonction MultiOutputClassifier de scikit-learn et un classifieur binaire au choix, comme une régression logistique ou un SVC.
- Pour réduire la complexité, on limite le nombre de tags pour lesquels on réalise la classification, on choisit de prendre les n tags les plus fréquents, ce choix constitue un paramètre de notre modèle.

Amélioration du modèle :

- Le modèle retourne les tags dont les fonctions de décision sont positives. On peut vouloir « forcer » le modèle à retourner un nombre minimum de tags, en allant chercher les tags non retournés par le modèle dont les fonctions de décisions négatives sont les plus grandes.



Métriques utilisées :

La métrique la plus naturelle pour évaluer la prédiction d'un ensemble de tags est l'indice de Jaccard, qui est la proportion de tags correctement prédits par rapport à l'union des tags réels et des tags prédits.

Si on veut forcer le modèle à prédire plus de tag, il est plus logique de regarder la proportion de tags correctement prédits par rapport l'ensemble des tags réels, j'ai appelé cette métrique « true_predict/true ».

$$\textbf{Indice de Jaccard} = \frac{\#(y_{true} \cap y_{pred})}{\#(y_{true} \cup y_{pred})}$$

$$\textbf{true\_predict/true} = \frac{\#(y_{true} \cap y_{pred})}{\#(y_{true})}$$

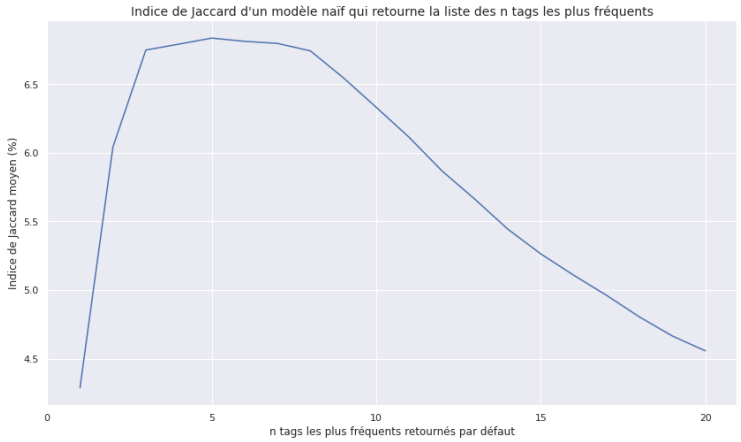
Voici un exemple de calcul de métriques :

Tags réels = y_true = ['python', 'pandas']
Tags prédits = y_pred = ['python', 'dataframe', 'numpy']

Score de Jaccard = 1 tag en commun sur 4 = 25.00 %.
true_predict/true = 1 tag en commun sur 2 possibles = 50.0 %.

On peut remarquer que lorsque le nombre de tags prédits est choisi, optimiser l'indice de Jaccard revient à optimiser le true_predict/true.

Avant même de construire un modèle élaboré il est intéressant de regarder ce que donne les métriques sur un modèle naïf, qui retourne systématiquement les n tags les plus présents :

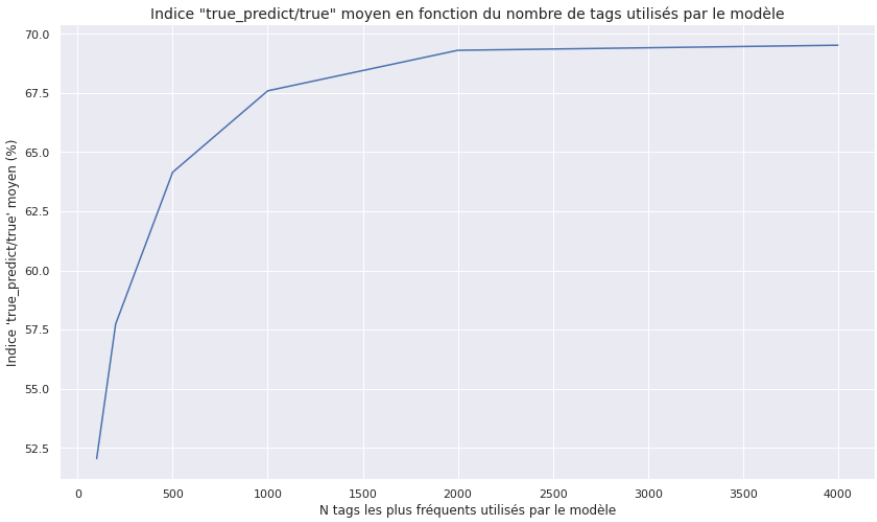


Gardons en tête que si le modèle naïf retourne les 5 tags les plus fréquents, il prédit correctement environ 17% des tags réels.

Pour résumer voici les principaux paramètres de notre modèle :

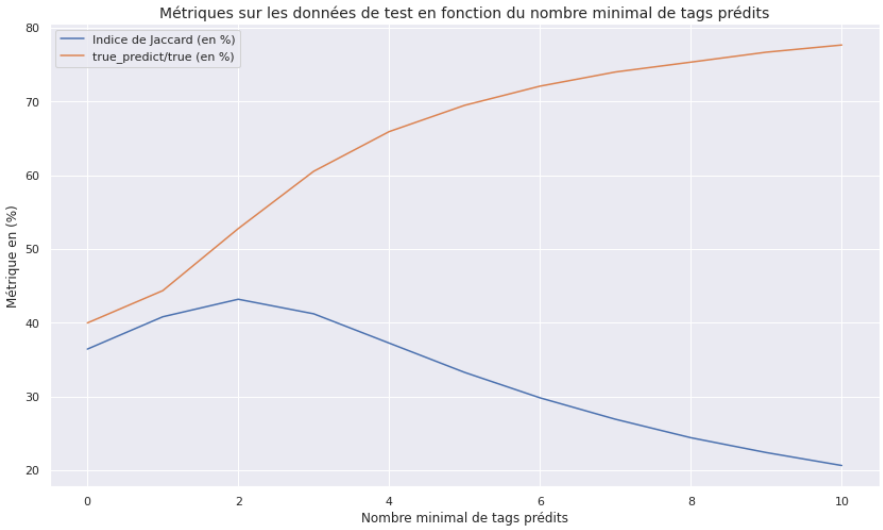
- Le **Nombre de tags sélectionnés** pour la classification, il s'agit d'un compromis entre performance et complexité du modèle.
- Le **Nombre de tags minimal prédit,** il s'agit plus d'un choix cohérent que d'un paramètre à optimiser. Plus ce nombre est élevé meilleure est la métrique « true_predict/true », mais il n'y a pas d'intérêt à retourner par exemple 100 tags à chaque fois.
- Paramètres de vectorisation du corpus : le **type de vectorisation** (bow ou tf-idf) et la **borne inf 'min_df'** pour les mots peu fréquents. On optimise ces paramètres par validation croisée.
- Paramètres du **classifieur binaire**, notamment les paramètres de régularisation pour éviter le surapprentissage. Le choix final de ces paramètres se fait par validation croisée.

Commençons par regarder l'effet du nombre de tags sélectionnés sur la métrique « true_predict/true », sur les données de test, les autres paramètres ayant été optimisés.



On peut voir qu'il est inutile d'utiliser plus de 2000 tags (soit environ 23% des tags), le modèle ne s'en trouvant que très peu amélioré.

Regardons maintenant les métriques en fonction du nombre minimal de tags prédit :



Comme le montre le graphique c'est une bonne idée d'aller chercher les tags dont les fonctions de décision sont négatives. Etonnamment l'indice de Jaccard est meilleur lorsqu'on force le modèle à retourner au moins 2 tags.

Plus le nombre de tags prédits est important meilleure est la proportion de tags réels prédits par rapport aux tags réels (métrique true_predict/true) ce qui est logique, mais moins bonne est la proportion de tags réels prédits par rapport aux tags réels et proposés (indice de Jaccard). Puisque le but ici est de suggérer des tags il fait sens de proposer plus de tags quitte à obtenir un indice de Jaccard plus faible. Renvoyer 5 tags semble un bon compromis, le modèle retourne alors en moyenne un peu moins de 70% des tags réels.

Après choix et optimisations des différents paramètres, voici les paramètres du modèle final :

- **Nombre de tags sélectionnés** = 2000
- **Nombre de tags minimal prédit** = 5
- **Vectorisation** = « TF-IDF » et **min_df** = 4
- **Classifieur** = régression logistique avec régularisation l2 et paramètre de régularisation C = 4.

A noter que le classifieur SVC donne des résultats similaires mais le modèle est beaucoup plus lourd en complexité temporelle.

Les métriques du modèle final choisit sont respectivement de 33% et 69% pour l'indice de Jaccard et le « true_predict/true ».

Pour finir voici quelques exemples concrets du fonctionnement de notre modèle final :

question_text	tags_true	tags_predict_fct_decision >= 0	tags_predict	jaccard_indice	true_predict/true
How do you round a floating point number in Perl? How can I round a decimal number (floating point) to the nearest integer? \n\ne.g.\n\n1.2 = 1\n1.7 = 2\n\n	[perl, floating-point, rounding]	[rounding, floating-point]	[rounding, floating-point, math, numbers, perl]	60.0	100.0
How do you change the size of figures drawn with matplotlib? How do you change the size of figure drawn with matplotlib?\n	[python, graph, matplotlib, plot, visualization]	[matplotlib, python]	[matplotlib, python, java, android, c#]	25.0	40.0
Removing numbers from string How can I remove digits from a string?\n	[python, string]	[string]	[string, numbers, python, c#, java]	40.0	100.0
insert datetime value in sql database with c# How do I insert a datetime value into a SQL database table where the type of the column is datetime?\n	[c#, sql, datetime]	[c#, sql, sql-server, datetime]	[c#, sql, sql-server, datetime, sql-server-2005]	60.0	100.0