Recherche d'Information

M2 DAC

TME 6. Combinaison de scores

Au cours de ce TME, nous nous interessons à la manière dont on peut combiner des scores provenant de différents modèles ou mesures, afin d'obtenir un meta-moteur performant. L'hypothèse centrale est que les différents modèles que l'on a pu implémenter tendent à retourner les documents pertinents plutôt vers le haut de la liste mais qu'ils sont rarement d'accord sur l'ordonnancement à proposer. L'idée est alors d'apprendre une combinaison des différents scores, et ainsi de tirer parti conjointement des différentes mesures considérées, afin d'obtenir un modèlede recherche plus performant. Le travail à réaliser est d'implémenter, entrainer et expérimenter un modèle de recherche basé sur une combinaison de scores issues de métriques diverses (orientées requête ou non).

1 Features

La première étape avant de combiner les scores issus de différentes métriques (ci-après appelés $features$) est bien sûr de disposer d'un mécanisme pour calculer et accéder à ces scores efficacement. Nous proposons alors de mettre en place une classe abstraite Featurer qui contiendra à minima :

- Une référence à l'index utilisé;
- Une méthode abstraite $getFeatures(idDoc, query)$ qui retourne une liste de scores (possiblement unique selon les cas) pour le document $idDoc$ et la requête $query$.

Afin d'être efficace et ne pas avoir à recalculer les mêmes choses plusieurs fois, la méthode $getFeatures$ pourra conserver les valeurs calculées pour chaque document et chaque requête dans une table $features$ de la classe $Featurer$ (que l'on pourra rendre persistente en rendant la classe serialisable par exemple).

Différents types de $Featurers$ pourront être définis:

- Certains non dépendant de la requête : longueur du document, nombre de termes différents dans le document, somme des idf des termes du document, importance du document dans le graphe des hyperliens (score $PageRank$), etc...

- D'autres orientés requête mais indépendants du document : somme des idfs de la requête, longueur de la requête, etc...
- Enfin des scores dépendant à la fois de la requête et du document comme ceux attribués par un modèle de recherche donné (modèle vectoriel, probabiliste, de langue, etc... avec différents paramètrages et schémas de pondération des termes).

Pour ce dernier cas, il conviendra de définir une classe $FeaturerModel$ par exemple qui dérivera $Featurer$, contiendra une référence à un modèle de recherche donné et déterminera les scores à attribuer aux documents en fonction d'un appel à la fonction $getRanking$ du modèle considéré. Notons que l'intérêt de la table $features$ est alors évident car la fonction $getRanking$ retourne l'ensemble des scores de tous les documents de l'index pour une requête donnée et que l'on ne veut pas avoir à recalculer cette liste à chaque appel à $getFeatures$ pour les différents documents à considérer.

Enfin, on pourra considérer un $Featurer$ particulier nommé par exemple $FeaturerList$ qui permettera de regrouper tous les scores obtenus sur une liste de $Featurers$ différents.

2 Meta-Modèle de recherche

Une fois les $Featurers$ définis pour notre meta-modèle de recherche, il s'agit de définir la manière de combiner les scores obtenus afin de retourner une liste ordonnée de documents en réponse à une requête utilisateur. Différents types de combinaisons peuvent être envisagés. Dans un premier temps, nous nous limiterons à une combinaison linéaire des scores mais nous pourrons envisager par la suite d'autres types de combinaisons. Dans un soucis de généricité, nous implémenterons donc une classe abstraite générique $MetaModel$ qui contiendra un $FeaturersList$ et qui étendera la classe $IRmodel$.

Pour la définition d'un meta-modèle linéaire, il conviendra alors d'étendre $MetaModel$ et de redéfinir la méthode $getScores$ de manière à ce que le score attribué pour chaque document d corresponde à un produit scalaire de ses $features$ $x_{d,q}$ avec un vecteur de poids θ : $f_\theta(d,q) =< x_{d,q}, \theta >$.

Il s'agit alors d'apprendre un vecteur de poids θ pour ordonner efficacement en réponse à une requête utilisateur nos documents en fonction de leur vecteur de $features$. Dans l'idéal, nous aimerions apprendre à ordonner les documents selon une mesure que l'on utilise pour évaluer nos modèles, par exemple la mesure MAP. Néanmoins, ce genre de mesure n'est pas dérivable et il sera alors difficile d'apprendre un vecteur de poids efficace pour ce genre de mesure. Il est alors d'usage de travailler par paires et d'apprendre à faire en sorte que tous les documents pertinents $d \in P(q)$ obtiennent un score $f_\theta(d,q)$ supérieur à celui des documents non pertinents $d' \in \bar{P}(q)$:

$$d \in P(q) \wedge d' \in \bar{P}(d) \Rightarrow f_\theta(d,q) > f_\theta(d',q)$$

Cela peut se faire relativement simplement en considérant un coût de hinge-loss classique en classification pour toutes les requêtes et toutes les paires de documents (d,d') telles que $d \in P(q) \wedge d' \in \bar{P}(q)$:

$$\ell_{theta} = \sum_{q \in \mathcal{Q}} \frac{1}{|P(q)| * |\bar{P}(q)|} \sum_{d \in P(q), d' \in \bar{P}(q)} \max\big(0, 1 - f_\theta(d,q) + f_\theta(d',q)\big) + \lambda ||\theta||_2^2$$

où $\lambda > 0$ correspond au coefficient de régularisation L2.

Nous proposons de minimiser cette fonction de coût par gradient stochastique. Pendant $tmax$ iterations:

1. Tirage aléatoire d'une requête q;
2. Tirage aléatoire d'un document pertinent d et d'un document non pertinent d' pour cette requête;
3. Si $1 - f_\theta(d,q) + f_\theta(d',q) > 0$ alors on modifie le vecteur θ:
$$\theta \leftarrow \theta + \alpha(x_{d,q} - x_{d',q})$$
4. On régularise θ selon sa norme l2:
$$\theta \leftarrow (1 - 2\alpha\lambda)\theta$$
où $\alpha > 0$ est le pas d'apprentissage (qui peut éventuellement être dégressif).

L'apprentissage du modèle doit se faire sur une base de requêtes d'apprentissage, tel que cela a été fait pour déterminer les paramètres optimaux pour les modèles de langue, Okapi, etc... Notez par ailleurs que, pour être tout à fait rigoureux, le réglage de λ devrait être fait en fonction d'une base de validation et non sur la base de test mais nous pouvons nous contenter ici d'observer les performances obtenues pour différentes valeurs de λ.

Note: Afin d'être à même d'apprendre des pondérations efficaces, il conviendra de faire en sorte que tous les *Featurers* utilisés retournent des scores entre 0 et 1.

Note 2 : Pour être plus efficace, l'apprentissage pourra se faire sur un sous-ensemble de documents obtenus par pooling sur les requêtes d'entraînement, c'est à dire l'union des k premiers documents retournés par un modèle classique pour chacune de ces requêtes.

3 SVMrank

Si le temps le permet, vous pouvez tester l'utilisation de l'implémentation SVMrank de Thorsten Joachims, afin de définir des combinaisons non linéaires des scores. Cet outil est disponible à l'adresse :
`http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html`.
Les instructions de téléchargement, de compilation et d'utilisation sont disponibles à cette même adresse.
Dans ce cas, on pourra définir une classe $MetaModelSVM$ qui travaillera, aussi bien pour l'apprentissage que pour l'inférence de scores de pertinence, en interfaçant SVMrank. Différents paramètrages permetteront d'expérimenter divers types de combinaisons des scores.