

IA pour le génie logiciel

Création d'un classifieur de tickets de bugs

Quentin Perez et Sylvain Vauttier

15 novembre 2021

Objectif de ce TP

L'objectif de ce TP est de montrer un exemple d'utilisation de l'intelligence artificielle dans le domaine du génie logiciel. Nous allons utiliser l'apprentissage supervisé ainsi que des techniques de traitement du langage naturel afin de créer un classifieur visant à catégoriser les tickets décrivant des bugs. Pour cela, nous allons utiliser un jeu de données de tickets issu de la publication scientifique de Herzig *et al.* [1]. Ce jeu de données contient 5591 tickets issus de 3 grands projets open-source : HTTPClient¹, Jackrabbit² et Lucene³. Ces tickets ont été manuellement annotés et ont donc des étiquettes "fiabiles", ce qui permet d'éviter les biais ou les erreurs de classification. Ce TP va se dérouler en plusieurs étapes :

- Traitement du langage naturel :
 - création de n-grammes
 - vectorisation via TF-IDF
- Sélection des *features* les plus représentatives du corpus
- Optimisation et sélection d'un classifieur
- Échantillonnage pour trouver un nombre optimal de *features*
- Étude de l'explicabilité du classifieur

Environnement de travail

Dans ce TP, nous allons travailler avec un environnement Python 3 et avec des *notebooks* Jupyter. Jupyter permet de créer des *notebooks* exécutables via un navigateur web ou des IDE le supportant. L'environnement Jupyter permet l'exécution, la visualisation des résultats et leur sauvegarde. Un carnet peut ainsi être exécuté et sauvegardé avec ses résultats (graphiques ou non) sans avoir à ré-exécuter l'application pour les obtenir. Nous allons également utiliser différents *packages* Python pour :

1. <https://issues.apache.org/jira/projects/HTTPCLIENT/issues/>
2. <https://issues.apache.org/jira/projects/JCR/issue>
3. <https://issues.apache.org/jira/projects/LUCENE/issues>

- Le clonage de projet et la manipulation de données Git, notamment les tags à l'aide de [GitPython](#) ;
- L'ouverture de fichiers CSV et la manipulation de données avec [Pandas](#) ;
- L'apprentissage machine avec la bibliothèque [Scikit-Learn](#) ;
- La visualisation avec [Matplotlib](#).

Mise en place de l'environnement de travail

Avant toute chose vous devez obligatoirement avoir Python 3 d'installé sur votre machine. Si ce n'est pas le cas vous pouvez vous référer à la documentation d'installation [ici](#). L'ensemble du TP est disponible sur GitHub grâce à l'URL suivante : <https://github.com/qperez/TP-Master-MTP-GL-IA4GL>. Clonez le dépôt sur votre machine puis placez-vous dans le répertoire.

Afin de s'abstraire des dépendances Python déjà présentes sur votre machine, nous allons créer dans le répertoire du TP un environnement virtuel Python ou Venv. Cet environnement virtuel vous permettra d'installer les dépendances nécessaires au TP.

Création du Venv

Pour la création de l'environnement virtuel, nous allons utiliser la documentation Python disponible [ici](#). Ouvrez un terminal ou une invite de commande et placez-vous dans le répertoire du TP.

Machine Linux / Mac

```
mkdir venv && python3 -m venv ./venv && source venv/bin/activate
```

Machine Windows

```
mkdir venv ; c:\Python35\python -m venv .\venv ; venv\Scripts\activate.bat
```

Installation des dépendances Python

La liste des dépendances Python nécessaires à la bonne mise en œuvre du TP se trouve dans le fichier `requirements.txt`. Pour installer les dépendances, utilisez la commande suivante à la racine du répertoire du TP : `pip install -r requirements.txt`

Environnement de développement


Pour ce TP, vous êtes libres d'utiliser l'environnement de développement que vous souhaitez. **Cependant**, nous recommandons très fortement l'utilisation de l'IDE [Visual Studio Code](#) pour la suite de ce TP. En effet, cet IDE embarque un serveur Jupyter vous permettant ainsi d'utiliser facilement les *notebooks* Jupyter et d'exécuter le code depuis l'IDE. L'installation du TP et des *plugins* nécessaires dans Visual Studio Code est montrée par la vidéo suivante : <https://youtu.be/kPqv0VMiK0c>

1 Traitement du langage naturel

Contrairement à notre dernier TP nous n'allons pas travailler avec des données numériques mais textuelles. Ces données textuelles ne sont pas intelligibles comme telles par un algorithme d'apprentissage, il va donc falloir effectuer des traitement sur ces données. Cette étape est appelée traitement du langage naturel. Plusieurs traitements sont nécessaire afin d'obtenir une information de qualité :

1. Créer des n -grammes à partir des différentes phrases du corpus. Un n -gramme est une suite contigus d'unités textuelles appelées *tokens*. Formellement, un n -gramme = $\{tok_1, tok_2, ..., tok_n\}$. Dans ce TP nous allons utiliser des uni-grammes, bi-grammes et des tri-grammes. Si l'on considère la phrase suivante

"Le chat est noir"

- les uni-grammes créés sont : "le", "chat", "est", "noir"
- les bi-grammes créés sont : "le chat", "est noir" 
- les ~~bi~~-grammes créés sont : "le chat est", "chat est noir"

2. Supprimer les mots trop/peu fréquents dans le corpus. En effet, ces mots par leur fréquence viennent introduire du bruit dans les données
3. La vectorisation de ces n -grammes via la méthode *Term Frequency - Inverse Document Frequency* (TF-IDF). Les n -grammes sont transformés en informations mathématiques (vecteurs) appelées *features*.

L'ensemble de ces étapes est réalisé à l'aide d'un seul objet présent dans Scikit-Learn : [TfidfVectorizer](#). La manière d'utiliser le vectorizer vous sera donné dans le carnet Jupyter de ce TP.

2 Sélection des *features* les plus représentatives du corpus

La seconde étapes consiste à sélectionner les *features* les plus représentatives du corpus de tickets. Vous allez devoir compléter la fonction `feature_computing`, pour cela :

1. Utilisez le vectorizer passé en paramètre afin de transformer les informations textuelles en informations mathématiques (vecteurs) utilisables par des classifieurs. Ces vecteurs sont créés à l'aide de la méthode TF-IDF du [TfidfVectorizer](#). Pour cela référez vous à la fonction [fit.transform](#).
2. La seconde étape consiste à sélectionner les *features* les plus représentatives du corpus. Un moyen de faire cela est d'utiliser la méthode du Chi-deux. La méthode du Chi-deux va mesurer la dépendance entre une feature donnée et la classe (BUG ou NBUG) et ainsi vous permettre de sélectionner k *features* représentatives du corpus. Pour faire cela vous devez créer un objet [SelectKBest](#) que vous affecterez à la variable `ch2`. Vous pouvez vous inspirer de l'exemple donné dans la [documentation](#).

3. Enfin, utilisez la méthode `fit_transform` avec en paramètres `X` et `labels` pour calculer les *features* représentatives. Affectez le retour de cette méthode à la variable `X`

3 Comparaison de classifieurs via Grid-Search

Dans cette fonction nous allons utiliser un algorithme nommé Grid-Search (algorithme de type brute-force utilisant la puissance cartésienne) permettant d'optimiser les paramètres d'un ou plusieurs classifieur(s) puis d'en comparer les performances. Nous allons comparer 5 types de classifieurs disponibles dans Scikit Learn :

- `Software Vector Machine` (`SVC` ([documentation](#))). Optimisation du paramètre `kernel` : `['linear', 'rbf']`
- `LogisticRegression` ([documentation](#)). Optimisation du paramètre `C` : `[0.5, 0.75, 1]`
- `MultinomialNB` ([documentation](#)). Pas d'optimisation de paramètre.
- `RandomForestClassifier` ([documentation](#)). Optimisation du paramètre `max_depth` : `[5, 10, 15]`
- `RidgeClassifier` ([documentation](#)) Optimisation du paramètre `alpha`: `[0.5, 0.75, 1]`

Attention, certains de ces classifieurs utilisent de l'aléatoire. Il est donc nécessaire de fixer les graines à l'aide du paramètre `random_state`, présent dans les constructeurs

Nous allons utiliser un Pipeline dans lequel nous allons ajouter les dictionnaires contenant les classifieurs et leurs paramètres à ajuster. Pour cela vous allez vous pouvez vous inspirer du code suivant :

```
pipeline = Pipeline([
    ('clf', UnClassifieur()),
])

parameters = [
    {
        'clf': [Classif1(random_state=0)],
        'clf__myParamClassifieur1': ['linear', 'rbf']
    },
    {
        'clf': [Classif2(random_state=0)],
        'clf__myParamClassifieur2': ['linear', 'rbf']
    }
]

grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1)
```

Dans le code plus haut le Pipeline est initialisé avec un classifieur. C'est un *workaround* car on ne peut pas créer de Pipeline vide. Nous allons utiliser ce Pipeline avec une liste (variable `parameters`) de dictionnaires contenant le classifieur à tester et ses paramètres. Dans le dictionnaire présenté plus haut :

- `'clf'` : est une liste contenant 1 seul élément, le classifieur à tester.

- `'clf_myParamClassifieur'` : permet à l'algorithme de tester les paramètres `myParamClassifieur` du classifieur testé `'clf'`

Un dictionnaire par classifieur/paramètre testé doit être ajouté dans le tableau `parameters`

4 Bloc principal d'exécution de la fonction de Grid-Search

1. Chargement du jeu de données de Herzig et al. avec la méthode `load_dataset("dataset_herzig_etal.json")`
2. Multiplication du titre des tickets du *dataset* par un facteur 3 via la méthode `multiply_title`
3. Extraction du corpus et des labels via la méthode `get_corpus_labels`
4. Vectorisation à l'aide de TF-IDF. Pour cela, créez un "vectoriseur" à l'aide de : `TfidfVectorizer(max_df=0.5, min_df=2, ngram_range=(1, 3), sublinear_tf=True, stop_words='english')`
5. Sélection des *features* représentatives grâce à la méthode `feature_computing`
6. Recherche d'un classifieur via la méthode `grid_search_classifiers`

5 Échantillonnage pour trouver un nombre de *features* optimal

1. Chargement du jeu de données de Herzig et al. avec la méthode `load_dataset("dataset_herzig_etal.json")`
2. Multiplication du titre des tickets du *dataset* par un facteur 3 via la méthode `multiply_title`
3. Extraction du corpus et des labels via la méthode `get_corpus_labels`
4. Vectorisation à l'aide de TF-IDF. Pour cela, créez un "vectoriseur" à l'aide de : `TfidfVectorizer(max_df=0.5, min_df=2, ngram_range=(1, 3), sublinear_tf=True, stop_words='english')`
5. Création du classifieur, sélectionner le meilleur couple classifieur/paramètres retourné par Grid-Search précédemment
6. Création du classifieur, sélectionner le meilleur couple classifieur/paramètres retourné par Grid-Search précédemment
7. Échantillonnage du nombre *k* de *features* par pas de 5000 entre 30000 et 60000, dans la boucle faire :
 - sélection des *k features* représentatives grâce à la méthode `feature_computing`
 - le scoring à l'aide de la méthode `make_scoring_train_test_split`

En cas d'égalité des scores F1 entre différents nombre de *features*, vous conserverez la valeur du plus grand nombre de *features* parmi les meilleurs scores.

6 Création de la matrice de confusion

Nous allons maintenant créer la matrice de confusion correspondant à notre classifieur. Cette matrice permet de visualiser graphiquement la qualité de la classification effectuée par notre classifieur. La matrice de confusion recense le nombre de :

- vrais positifs (VP)
- vrais négatifs (VN)
- faux positifs (FP)
- faux négatifs (FN)

Cette matrice est un indicateur de la qualité de votre classifieur. Plus le nombre de FP et FN est réduit meilleure est la classification.

Pour créer cette matrice il vous faut :

1. Comme dans les blocs de code précédent charger le jeu, multiplier le titre, extraire le corpus et les labels puis binariser les labels
2. Extraire les k meilleures *features* en fonction de l'échantillonnage fait plus haut (méthode `feature_computing`)
3. Utiliser la meilleure configuration de classifieur calculée avec Grid-Search
4. Entraîner ce classifieur à l'aide de la méthode `fit` avec `X_train` et `y_train`
5. Faire des prédictions à l'aide de la méthode `predict` de votre classifieur

Pour cela, vous pouvez vous inspirer de l'exemple donné ici : https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

7 Explicabilité du classifieur

Un des défis lié à l'IA et aux algorithmes à trait à leur explicabilité. L'explicabilité se définit comme le fait de pouvoir comprendre les mécanismes internes du classifieur qui fondent une ou plusieurs prédictions. Cette explicabilité peut se faire de manière globale (mécanismes internes du classifieur qui conduisent à la classification) ou de manière locale (mécanisme qui conduisent à la classification d'une instance).

Nous allons expliquer la classification de 4 tickets :

- 2 tickets sont des faux positifs (indices 3997 et 5098 dans le *dataset* de tickets)
- 2 tickets sont des faux négatifs (indices 2656 et 3479 dans le *dataset* de tickets)

Pour expliquer les mots impactant la classification des tickets nous allons utiliser une méthode d'explication se nommant Lime située dans le package Python [eli5](https://eli5.readthedocs.io/en/latest/tutorials/black-box-text-classifiers.html#textexplainer). Pour cela vous pouvez vous inspirer de l'exemple donné dans la documentation de eli5 : <https://eli5.readthedocs.io/en/latest/tutorials/black-box-text-classifiers.html#textexplainer>.

Attention, avant d'exécuter le code suivant, veuillez à avoir exécuté le code du bloc précédent (code de matrice de confusion).