Pipeline

Explication rapide

En gros le but de la pipeline c'est de prendre un data set pandas, de lui appliquer des transformations (ex : transformer notre query en vecteur numérique). Une fois les transformations appliquées, on peu utiliser un classificateur pour faire nos prédictions.

On peu utiliser un grid search sur toute notre pipeline, autant pour optimiser nos transformations que les paramètres de notre classificateur.

Les méthodes score, predict, fit, etc fonctionne sur une pipeline.

On peut assembler des pipeline l'une à la suite de l'autre.

Pipeline dans notre code à nous

Les trucs importants se passent dans basis_pipeline.py. Notre pipeline est séparée en 2 partie, la partie *Transformer* et la partie *Classifier*.

Voici la pipeline de trnasformation :

```
#Pipeline de toutes les transformations qu'on fait, en ordre
transformation_pipeline=pipeline.Pipeline([

    ("data_extract", FilterColumns(filter_group=["query_expression","search_nresults","user_country","user_language"])),
    ("vectorize_query", VectorizeQuery(vectorize_method="count", freq_min=2)),
    ("categorical_var_to_num", TransformCategoricalVar())

])
```

Chacune des 3 lignes représente une transformation qu'on fait sur notre data frame. par exempel vectorize_query prend la query, la transform en vecteur numérique et retire la query textuelle du data frame. Toutes ces transformations se font une après l'autre. On peut ajouter d'autre *transformer* éventuellement.

Grid search avec pipeline

Pour tester toutes les combinaisons possibles, on se définit une grille de paramètres à explorer :

```
grille_transformer={
    "Transformer__vectorize_query__freq_min": [1,2],
    "Transformer__vectorize_query__vectorize_method": ["count","tf-idf"]
}
estimators={
    "MLP": MLPClassifier(),
    #"XGB":GradientBoostingClassifier(),
    "KNN":KNeighborsClassifier()
}
grille_estimators={
    "MLP":{"Classifier__activation": ["relu", "tanh"]},
    #"XGB":{"Classifier__n_estimators":[10,32]},
    "KNN":{"Classifier__n_neighbors":[1,3,10],"Classifier__weights":["uniform","distance"]}

}
```

Une grille pour les paramètres de notre pipeline de transformation, un dict avec nos modèles et un dict avec nos paramètres de classificateur à tester. Il faut respecter la structure si on veut en ajouter d'autre. Les classificateurs doivent avoir la méthodes .predict_proba() pour pouvoir être utilisés, car on fait le grid search avec le score de coveo

J'ai créé de quoi qui permet de tester tout ça :

```
Make_grid=Make_All_Grid_Search_Models(transformation_pipeline,grille_transformer,estimators,grille_estimators)

if config["Create all grid searchs"]:
    Make_grid.test_best_grid_search(mini_df,mini_y)

if config["Show me all the grids"]:
    Make_grid.show_me_all_grids()

if config["Show me the best grid"]:
    grid_search=Make_grid.return_best_grid_search()
    print("\nBest grid search:")
    print(grid_search.best_params_)
    print("Score en validation:",grid_search.best_score_)
```

1. ça permet de créer un fichier avec pickle qui sauvergarde une liste qui contient un objet grid search pour chacun des classificateur qu'on utilise (le fichier est list_grid_search.p).

2. permet de print toutes les combinaisons testées

3. retourne le meilleur objet grid search qu'on peut ensuite utiliser

Améliorations

Pour l'instant on utilise seulement une partie de nos données, on prend juste les search qui résulte en un click, puis ceux sans click ne sont pas utilisés (donc 52 000-25 000 ish données sont tout simplement pas utilisées).

Je suis pas mal sûr que y'a de quoi à faire avec ces données là. En leur attribuant par exemple un document en regardant les visits antérieurs de l'utilisateur avec le visit id et le click date time. Ex : si un gars cherche "I want to know what is the best restaurant to eat on my first date with this awesome chikita I met on Tinder" et ne click sur rien, puis 2 minutes après il cherche : "good restaurant" et il clique sur de quoi, on peut assumer qu'on aurait pu lui renvoyer ce document trouvé dès sa première querry trop longue.

Je propose qu'une petite équipe travaille sur ça, car je pense qu'on peut vraiment aller chercher de quoi. C'est également simple de travailler de façon indépendante sur ça, pendant qu'une autre équipe travaille sur autre chose. J'ai commencer à faire un frame de fonction dans le fichier import_data.py :

```
#TODO à faire
def sophisticated_merge(df_searchs,df_clicks):
    '''
    Merge les clicks au searchs en se basant sur le search_id.
    Trouver une façon intelligente d'assigner des documents_id au search qui n'ont résulté en un click

    :param df_searchs:
    :param df_clicks:
    :return: Un data frame pandas avec toutes les colones contenus dans df_searchs et df_clicks
    '''
    pass
```

Il resterait à la compléter. C’est quand même une grosse job à faire !