Domain Specific Language

Rapport ClassifAI

DSL to setup and compare AI classification algorithm

Fernandes William Demonchaux Marine Gross Paul Kherraf Taha

Sommaire

Liens et Informations

Vous trouverez sur le répertoire Github de Fernandes William les informations suivantes :

- le projet dans le dossier **classifIA**
- La **BNF** et le **domain model** sous forme de diagramme de classe dans le dossier **rapport**

Le projet peut être exécuté en exécutant le fichier Main.groovy. Ce script lit tous les scripts contenus dans le répertoire "scripts" et génère les fichiers correspondants en python ou jupyter (selon le choix d'exportation) dans le dossier "results".

Description du langage proposé

Après avoir étudié les Kaggles fournis et les besoins de Frédéric Precioso, nous avons compris que le cœur du métier du traitement des données était de pouvoir évaluer et classer rapidement les différents classifiers, ainsi que de pouvoir visualiser les résultats, sous forme graphique par exemple, ou bien des tableaux.

Cependant, pour arriver à cette étape, plusieurs phases sont nécessaires, telles que l'importation des fichiers et le prétraitement de ces derniers pour éliminer tout ce qui est superflue, ainsi que les transformations qui doivent être appliquées aux données qui sont souvent similaires, car l'idéal est de pouvoir les comparer dans les mêmes conditions. Seulement après ces étapes inévitables, il est possible de comparer et d'analyser les résultats. Notre objectif est donc de faciliter la comparaison des algorithmes et de rendre modulables les transformations appliquées, afin de pouvoir, avec très peu de code, comparer des algorithmes similaires, avec des transformations différentes, ou bien plusieurs algorithmes avec les mêmes transformations.

Nous avons ainsi identifié 4 phases, et avons décidé d'imposer au développeur de notre DSL de les spécifier. L'utilisateur est conscient des phases obligatoires, et celles-ci vont permettre de structurer les scripts.

```
preparation {}

transformation {}

training {}

comparison {}
```

La préparation

La phase de préparation des données est importante pour nettoyer les données d'entrée, mais peut être fastidieuse. Pour faciliter cette étape, nous avons créé un outil permettant aux développeurs d'importer soit un dataset unifié, soit deux datasets séparés. Lors de l'importation unique, il est possible de spécifier ou non comment répartir les données. Si aucune spécification n'est demandée, alors la répartition sera de 80 % pour le training et 20% pour le testing.

La phase de preprocessing permet également de nettoyer les données avec des méthodes telles que **rmNull** pour supprimer les éléments non définis dans toutes les colonnes, et **rmOutliers** pour supprimer les éléments ayant des valeurs trop éloignées.

L'ordre des éléments saisis n'a aucun impact sur la génération du code Python, mais certains paramètres peuvent générer des erreurs si les entrées sont invalides.

```
preparation {
    fetchAll "../input/digit-recognizer/testTrain.csv"
    splitDataset {
        training_size 70
        seed 2394882
    }
    preprocessing {
        rmNull
        rmOutliers 0.01, 0.8
    }
}
```

```
preparation {
    fetchTrain "../input/digit-recognizer/train.csv"
    fetchTest "../input/digit-recognizer/test.csv"

    preprocessing {
        rmNull
        rmOutliers 0.01, 0.8
    }
}
```

La transformation

Nous avons proposé d'introduire une phase de transformation pour permettre aux développeurs de déclarer les transformations apportées aux données. Nous avons également identifié que certaines transformations peuvent prendre beaucoup de temps à s'exécuter, c'est pourquoi nous avons créé la notion de "pipe" qui regroupe plusieurs transformations à exécuter ensemble.
Ce mot clé permet de détecter et regrouper des transformations similaires en amont et d'optimiser les pipelines de transformation.
La phase de transformation donnera aux développeurs une vision plus claire des transformations appliquées aux données et leur permettra de les factoriser facilement.

Déclarer un type de transformation se fait de cette façon :

```
declare pca55 as pca {
    n_components 0.55
}
```

Pour les unifier dans une même pipeline, il suffit de faire référence à la variable crée juste avant :

```
pipe t1: [pca55, pca55]
```

Il est également possible d'injecter des pipelines dans des pipelines :

```
pipe t2: [t1, pca55, pca62, std]
```

Concernant les transformations prises en charge dans notre DSL , nous avons la possibilité d'utiliser :

- PCA
- MinMax
- StandardScaler

Le Training

Cette phase de comparaison des modèles est cruciale pour les développeurs. Nous utilisons une approche similaire à celle utilisée lors de la phase de transformation, en explicitant tous les classificateurs et leurs paramètres. C'est à cette étape que les transformations à appliquer avant l'entraînement sont spécifiées. Cette méthode permet aux développeurs de rapidement avoir une vue d'ensemble de tous les classificateurs utilisés dans le script et de facilement comparer des classificateurs similaires mais avec des paramètres différents.

Cependant, il est important de noter que la phase d'entraînement peut générer des erreurs si tous les paramètres de distribution ne sont pas spécifiés.

Notre DSL prend en charge plusieurs classificateurs, dont :

- Random Forest Classifier
- Gaussian Naives Bayes Classifier
- KNeighbors Classifier

```
declare gaussian1 as gaussian {
    cv 5
    kfold stratified(2, true)
    distributionParams {
        smoothing logspace(-9, 0, 5)
    }
    transformation t2
}
```

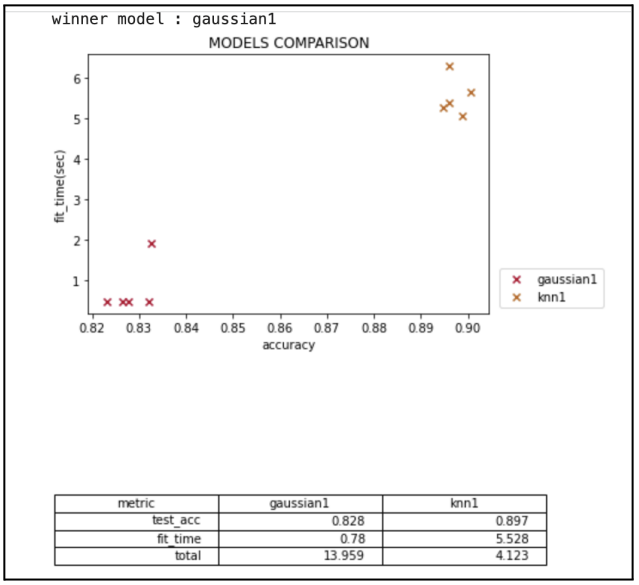
La Comparaison

Enfin, la dernière étape de la comparaison est l'objectif principal du projet, qui consiste à comparer les classificateurs en fonction des transformations, des hyperparamètres ou des différents algorithmes. Les développeurs peuvent spécifier les classificateurs qu'ils souhaitent comparer, un classifier impliquant une transformation et un ensemble d'hyperparamètres. Il est également possible de faire une comparaison avec des

poids. Seuls deux critères de comparaison ont été implémentés : la précision et le temps d'exécution, et il est possible de donner un poids à chacun d'entre eux pour varier les résultats.

```
comparison {
    compare gaussian1, gaussian2,knn1 with test_acc weight 10 and fit_time weight 3
}
```

Nous générons un tableau de comparaison et des graphiques pour représenter les résultats obtenus.
Ces représentations ont pour but d'aider à classifier et comparer les résultats.



metric	gaussian1	knn1
test_acc	0.828	0.897
fit_time	0.78	5.528
total	13.959	4.123

Il est possible de générer 2 types de fichiers, soit un notebook, soit un fichier python :

```
notebook "compare_gaussian_knn"
```

```
python "compare_gaussian_knn"
```

Feature supplémentaire (Killer Feature)

Grâce à un expert dans le domaine des DSL, nous avons ajouté une feature qui permet au développeur d'avoir au début de son jupiter, ou via un pdf exporter, une visualisation du flow qui va être exécuté a posteriori. Cette visualisation donne une vue d'ensemble de ce que le développeur a créé avec notre DSL, il peut également s'apercevoir de tout ce qui n'est pas utilisé et qui peut causer une perte de temps d'exécution, comme les transformations déclarées ou les classificateurs. Cette visualisation est sous forme de graphe allant de l'import des données jusqu'à la comparaison.

Réalisation du compiler

Séparer par étapes notre code à faciliter l'implémentation du compilateur. En effet, nous avons pu créer des classes Generator qui génèrent des Strings pour chaque étape en fonction des paramètres donnés.

Scénario

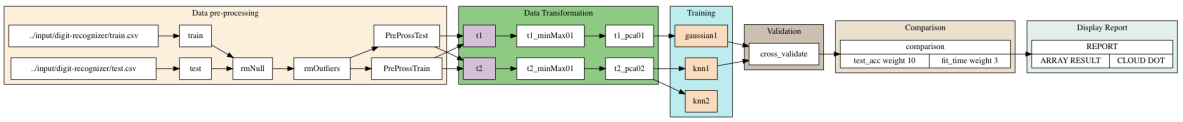
Il existe un grand nombre de scénarios possibles réalisable via notre DSL, tant que l'utilisateur souhaite effectuer des comparaisons entre différents modèles ou transformations. Il est donc possible de :

- Comparer des modèles différents en utilisant les mêmes paramètres de distribution et les mêmes transformations.
- Comparer le même modèle en utilisant les mêmes transformations, mais avec des hyperparamètres différents.
- Comparer l'impact de transformations différentes sur un même modèle.

Comparaison de 2 modèles différents

```
preparation {
   fetchTrain "../input/digit-recognizer/train.csv"
   fetchTest "../input/digit-recognizer/test.csv"

   preprocessing {
       rmNull
       rmOutliers 0.01, 0.8
   }
}
transformation {
   declare pca01 as pca {
       n_components 0.62
   }
   declare pca02 as pca {
       n_components 0.41
   }
   declare minMax01 as minmax {}
   declare standardScaler1 as standardScaler{
   }

   pipe t1: [minMax01, pca01]
   pipe t2: [minMax01, pca02]
}
training {
   declare gaussian1 as gaussian {
       cv 5
       kfold stratified(5, true)
```

```
            distributionParams {
                smoothing logspace(-5, 0, 5)
            }
            transformation t1
        }

        declare knn1 as knn {
            cv 5
            kfold stratified(2, true)
            cv 5
            distributionParams {
                nNeighbors randint(1, 11)
                algorithm 'auto'
            }
            transformation t2
        }

        declare knn2 as knn {
            cv 5
            kfold stratified(2, true)
            cv 5
            distributionParams {
                nNeighbors randint(1, 10)
                algorithm 'auto'
            }
            transformation t2
        }
    }


    comparison {
        compare gaussian1, knn1 with test_acc weight 10 and fit_time weight 3
    }
    notebook "compare_gaussian_knn"
```

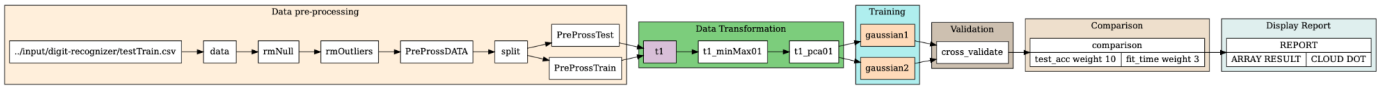


notebook généré :
https://www.kaggle.com/marinedemonchaux/compare-knn-gaussian

Comparaison d'un même modèle avec différents hyper paramètre de distribution

```
preparation {
    fetchAll "../input/digit-recognizer/train.csv"
    splitDataset {
        training_size 70
        seed 2394882
    }
    preprocessing {
        rmNull
```

```
        rmOutliers 0.01, 0.8
    }
}
transformation {
    declare pca01 as pca {
        n_components 0.62
    }
    declare pca02 as pca {
        n_components 0.41
    }
    declare minMax01 as minmax {}
    pipe t1: [minMax01, pca01]
}
training {
    declare gaussian1 as gaussian {
        cv 5
        kfold stratified(5, true)
        distributionParams {
            smoothing logspace(-9, 0, 5)
        }
        transformation t1
    }
    declare gaussian2 as gaussian {
        cv 5
        kfold stratified(5, true)
        distributionParams {
            smoothing logspace(-2, 0, 5)
        }
        transformation t1
    }
}
comparison {
    compare gaussian1, gaussian2 with test_acc weight 10 and fit_time weight 3
}
notebook "compare_gaussian"
```



notebook généré : https://www.kaggle.com/marinedemonchaux/compare-gaussians

Prise de recul sur le DSL et la syntaxe

En résumé, notre DSL répond aux exigences souhaitées et comporte des fonctionnalités de gestion des erreurs pour les entrées invalides. Il y a de nombreuses fonctionnalités exploitables et il est facile d'ajouter de nouvelles fonctionnalités au DSL grâce aux modèles créés pour faciliter l'implémentation de nouvelles parties du code. De plus, nous sommes assez permissifs avec l'utilisateur en utilisant des valeurs par défaut pour certaines structures, ce qui rend le code plus souple à écrire dans certaines situations. Le code transformé en Jupyter ou en Python présente une visualisation graphique des résultats obtenus lors de la comparaison. En outre, nous avons une structure de visualisation du code

pratique pour que l'utilisateur et les personnes qui lisent ce code aient une idée globale de la façon dont le code est structuré.

Cependant, malgré notre généricité, le projet aurait besoin d'un refactor pour structurer encore mieux les différentes implémentations, notamment les dernières. Il n'est pas possible pour l'utilisateur d'ajouter du contenu tel que de nouvelles transformations ou classificateurs via notre DSL. Cela est dû à un manque de temps et devrait être amélioré à l'avenir en créant une syntaxe permettant à l'utilisateur d'ajouter dynamiquement de nouvelles bibliothèques. Il convient également de noter que tout le projet ne supporte pas les informations erronées. Cela est dû à la gestion des erreurs qui a été faite après le support de la syntaxe, ce qui a créé une dette technique. Malgré ces points négatifs, la mission semble respectée et la syntaxe nous semble cohérente et facile à prendre en main pour un utilisateur souhaitant faire des comparaisons de classificateurs.

Concernant la technologie choisie, nous avons choisi d'utiliser Groovy pour construire notre DSL interne. Malgré nos connaissances limitées du langage, nous pensons que c'est un bon choix, car Groovy est très permissif et permet de créer une syntaxe impressionnante. La prise en main est similaire à celle de Python et le fait de ne pas avoir à typer les variables est un vrai gain de temps. Nous avons également apprécié les closures de Groovy qui ont été d'une grande aide pour notre système de phases. Cependant, il y a une courbe d'apprentissage importante pour comprendre les concepts spécifiques à Groovy, et il est simple de causer des fuites de mémoire lors de la création de DSL et de la gestion des données.

Répartition de groupe

Marine DEMONCHAUX :

- Création d’un exemple notebook correct sur lequel nous nous sommes basés
- L’ajout de la fonctionnalité de génération du notebook ou python,
- Génération du tableau des scores dans l’étape comparaison,
- Fixer les erreurs de syntaxe ou de kernel en testant le résultat python.

Taha KHERRAF :

- La visualisation en graph du flow
- Etape de Préparation + réjection d'erreur sur la phase de préparation
- Le support de la syntaxe de comparaison + la génération de la comparaison en python

William FERNANDES :

- Prise en charge de tout le support de la syntaxe excepté la phase de comparaison
- Mise en place de la phase transformation et des pipelines associés
- Mise en place de la phase de training et du traitement de tous les hyper paramètres
- Réjection d'erreur sur la phase de transformation et training
- Mise en place du kernel et des patterns pour faciliter l'implémentation

Marine DEMONCHAUX Taha KHERRAF William FERNANDES :

- Création de la syntaxe
- Réalisation de la BNF
- Conception du domain model
- Réalisation du rapport

Il est malheureux de constater que Paul Gross n'a pas donné de nouvelles ni participé au projet depuis le début. Il a commencé à implémenter la BNF lors de la dernière séance de cours le 16 janvier 2023, mais ne la jamais finie, malgré l'avoir contacté pour obtenir le BNF finie.