Projet S8 SISN

Intégration d’une méthode d’apprentissage à scikit-learn

19 mai 2017

CENTRALE
MARSEILLE

BODIN Guilherme - GORCE Yoann - SEVY Clément - THOYER Romain

Table des matières

Introduction 1

I. Etude préalable 1

A. Les arbres de décision k-means 1

B. Pré Requis pour contribuer à scikit-learn 1

II. Redéfinition du projet 2

A. Les tests en python 2

B. Réalisation de la documentation 2

C. Intégration et optimisation 3

1. Adaptation des arbres k-means aux arbres de décision sklearn 3

2. Réécriture du code initial en Cython 4

Conclusion 5

Annexe I - Processus visuel de la division des groupes selon les arbres de décision k-means 6

Annexe II - Structure des tests 6

Annexe III – Capture d'écran du résultat du test coverage 7

ANNEXE IV – Extrait de l'analyse du temps d'exécution de chaque partie 8

ANNEXE V - fichier Setup.py 8

ANNEXE VI - Fichier Test_Cython.py 9

Introduction

Scikit-learn est une bibliothèque open-source python dédiée au data-analysis et au data-mining. C'est une bibliothèque libre qui comprend énormément d'outils de régression, de clustering, de réduction de dimension, de sélection de modèle et de classification. Scikit-learn combine l'usage du langage python et du langage Cython.

Dans le cadre de notre projet, nous souhaitons rajouter à cette bibliothèque une méthode d'apprentissage supervisée basée sur une méthode de clustering, les arbres de décision k-means pensés et mis en œuvre par Célia Châtel, François Brucker et Pascal Préa.

I. Etude préalable

A. Les arbres de décision k-means

Les arbres de décision k-means sont basés sur les arbres de décision classiques (ceux de Breiman) et utilisent des critères de distance. Le modèle construit un noeud racine qui contient tous les exemples qu'on veut classifier. À partir de ce noeud l'algorithme réalise la méthode de "clustering" k-means avec k=2, la méthode choisit au hasard parmi le groupe ("cluster") initial (contenant initialement tous les exemples) deux points pour être les noeuds de l'arbre. Après elle associe les exemples aux noeuds à partir du critère de distance euclidienne, c'est-à-dire, les exemples appartiennent au noeud qui est le plus proche. Maintenant on a un arbre avec un noeud initial qui a deux fils, ses fils sont les deux nouveaux noeuds qui contiennent des parties disjointes et complémentaires de la base initiale d'exemples ("clusters"). À chaque fois qu'on crée deux nouveaux noeuds la méthode trace une ligne divisant les deux groupes ("clusters") associés aux noeuds. L'arbre peut continuer ce processus jusqu'à que chaque noeud ne contient qu'un exemple. On peut arrêter le processus de la division en noeuds si toutes les exemples sont de la même classe. On remarque que les deux premiers noeuds sont choisis au hasard mais les suivants sont définis comme le barycentre des exemples qui appartiennent au groupe ("cluster"). À la fin on fait l'union des groupes purs qui contiennent la même classe. Cette méthode permet un gain de précision dans la décision d'après l'analyse de Célia Châtel, François Brucker et Pascal Préa. On trouvera en annexe I le processus visuel de la division des groupes selon les arbres de décision k-means.

B. Pré Requis pour contribuer à scikit-learn

Scikit learn est hébergé sur Github, un service web d'hébergement et de développement de logiciel utilisant un système de versionning. La plateforme permet à plusieurs utilisateurs de contribuer sur un projet. Il faut comprendre que tout code a un coût de maintenance important puisqu'il doit suivre l'avancée des autres outils pour des soucis de compatibilité et que la complexité d'un code dépend de manière non linéaire au nombre de ses fonctionnalités. Les règles d'ajouts dans la bibliothèque sont donc très strictes.

Tout d'abord le code doit suivre une certaine pratique de codage, il doit notamment suivre la

norme PEP8, il doit posséder des docstrings informatives pour chaque méthode public, doit proposer des exemples des fonctionnalités introduites, doit proposer une documentation suivant des règles spécifiques ainsi que des tests unitaires avec un taux de couverture dépassant les 90%. Des outils sont mis à disposition dans la bibliothèque scikit-learn afin de vérifier que notre code respecte les normes.

Cependant avant de pouvoir inclure de nouvelles fonctionnalités, l'algorithme doit avoir été présenté dans une publication depuis plus de 3 ans, avec plus de 200 citations et doit avoir une large utilité. On s'attend également à ce que l'algorithme proposé dépasse les méthodes déjà mises en oeuvre dans scikit-learn. La méthode proposée par l'équipe d'informatique de Centrale Marseille n'a été présentée que dans l'année 2016, aucune publication n'a encore été faite, il est donc impossible pour nous d'inclure l'algorithme dans la bibliothèque. Nous avons donc révisé nos objectifs afin d'optimiser au mieux le code et de proposer une documentation ainsi qu'une couverture du code par les tests.

II. Redéfinition du projet

Comme nous l'avons vu il n'est pas possible à l'état actuel de proposer une contribution à la bibliothèque scikit-learn, nous avons tout de même travaillé dans ce sens afin de proposer des tests, une documentation et une optimisation du code.

A. Les tests en python

Le but est de créer un fichier test en python qui permettra de vérifier que nos fonctions fonctionnent de manière correcte. Ces tests de fonctions sont appelés tests unitaires, ou tests du programmeur, en effet ils sont au centre de l'activité du programmeur et permet à ce dernier de vérifier que chaque fonction réalise correctement l'action qui lui est dédiée.

Ces tests sont à écrire au fur et à mesure du code, certaines méthodes de développement comme le Test-Driven Development disent qu'ils doivent être écrits avant la fonction en question.

Nous avons utilisé le module unittest de la bibliothèque standard de Python afin d'effectuer nos tests unitaire. Nous devons définir une fonction de test pour chaque fonction présente dans le code. Nous détaillerons en annexe II la structure des tests.

Une fois les tests effectués, il faut veiller à ce que nous avons couvert tout le programme. Pour cela il existe un module de nose, nosetests qui permet de vérifier que les tests ont couverts la majorité du programme, c'est à dire qu'on appelle entièrement notre fichier dans notre fichier test. Au-delà de 90%, on peut considérer que la couverture par test unitaire est bonne. Ici nous obtenons un taux de couverture de 99%. Une capture d'écran de l'analyse de couverture est disponible annexe III.

B. Réalisation de la documentation

Comme nous l'avons affirmé, pour qu'un algorithme puisse être ajouté à scikit-learn, celui-ci doit avoir une documentation détaillée des classes, méthodes et fonctions qui peuvent être utiles à l'utilisateur mais aussi les cas dans lesquels utiliser le code, des exemples et des explications

mathématiques ou des modules pré requis. Bien que notre code ne puisse être intégré dans l'immédiat, nous avons décidé de réaliser cette documentation, en anglais bien sûr, de façon à ce code puisse tout de même être compris, utilisé et partagé.

Cette documentation a été réalisée grâce à l'outil Sphinx (http://www.sphinx-doc.org/en/stable/). Il permet, à partir de fichiers python d'un projet la création rapide de fichiers reStructuredText (.rst) et une exportation en html donc de construire une documentation interactive, hiérarchisée et facilement hébergeable. La documentation que nous avons réalisée pour notre méthode d'apprentissage se trouve à l'adresse rthoyer.perso.ec-m.fr. Sa réalisation a été grandement facilitée par l'importation et l'utilisation du module autodoc de Sphinx qui parcourt les fichiers python à la recherche des docstrings et crée à partir de celles-ci la documentation des différentes classes et fonctions. Il a donc été nécessaire d'agrémenter le code des différents fichiers python de commentaires, d'explicitation des arguments et des outputs des fonctions afin que la documentation soit entièrement réalisée automatiquement.

En complément de la simple explication du code, il est nécessaire de montrer dans la documentation les apports de celui-ci. Nous avons donc inclus des tableaux comparatifs présentants les apports de l'algorithme par rapport à une méthode classique d'arbres de décision et l'apport du cython à cet algorithme.

C. Intégration et optimisation

1. Adaptation des arbres k-means aux arbres de décision sklearn

Adapter le code des arbres de décision k-means dans la bibliothèque scikit-learn n'est pas une tâche facile. Le code a été développé par plusieurs personnes avec un background de programmation très fort ce qui rend le code difficile à comprendre. La bibliothèque étant entièrement optimisée, il est très difficile de changer des parties importantes sans casser le module.

Une approche pour intégrer les arbres de décision k-means à la bibliothèque était d'ajouter une nouvelle méthode de split dans la définition de splitter dans _splitter.pyx de la bibliothèque. Cette méthode réalisera un k-means à chaque noeud de l'arbre exactement comment on a défini l'arbre de décision k-means.

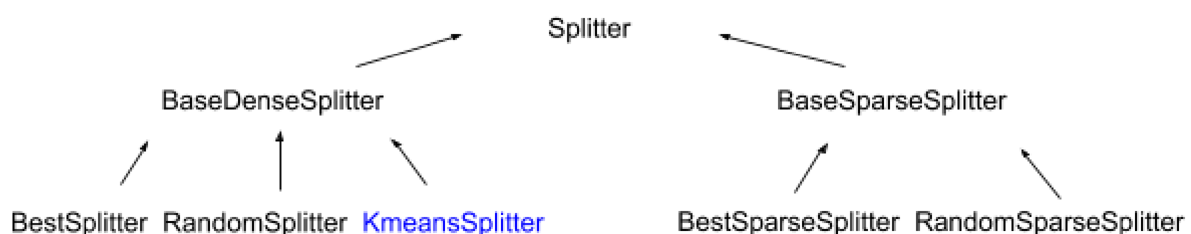


Figure 1 - Nouvelle structure des méthodes dans _splitter.pyx

Le nouveau splitter aura besoin d'un "critérium" (un critère pour définir quel split est le meilleur) pour continuer à fonctionner normalement. Le critérium n'est pas le problème parce que si on lui donne qu'une façon de faire le split, cette façon est forcément la meilleur. Le problème est dans la définition des arbres de décision dans scikit-learn. En effet, les arbres de décision classiques définissent un threshold (seuil) à partir des caractéristiques statiques de l'exemple comme sa pureté selon des critères prédéfinis par rapport à une classe. Si la pureté d'un exemple est plus grande que le threshold déterminé l'exemple va dans un fils de l'arbre et si la pureté est moins grande l'exemple va dans l'autre fils. Or les arbres k-means le threshold est une caractéristique dynamique, c'est la distance entre l'exemple et le nœud auquel il appartient. Il doit donc être calculé à chaque fois qu'on fit un exemple. Le fait que cette caractéristique soit dynamique signifie deux choses donc qu'il faut redéfinir la classe nœud dans la bibliothèque et qu'il faut qu'à chaque itération le k-means nous rend la distance exacte de l'exemple au nœud auquel il appartient. Dû à ces problèmes techniques l'idée n'a pas pu aboutir dans le temps impartit.

2. Réécriture du code initial en Cython

Cython est un langage qui est une extension de Python utilisant le langage C afin de permettre aux programmes d'être plus rapide (notamment en définissant les types de chaque variable et fonction).

Nous voulions utiliser Cython pour optimiser le programme initial car Cython est très utilisé dans la bibliothèque de Scikit-learn. Ayant pour objectif initial de nous intégrer dans cette base de données, la transformation du code en Cython était essentielle.

De plus, les codes déjà présents dans Scikit-Learn sont plus rapides et efficaces que notre code de base. En effet les codes existant traitent presque instantanément la base Digits de Scikit-Learn alors que le nôtre met 3.44 Secondes initialement.

Nous avons ensuite utilisé différentes bibliothèques (Digits qui est en dimension 64 et Iris en dimension 4), en comparant pour chacune de ses bibliothèques l'évolution du temps de réalisation du code en fonction du nombre de données, ce qui nous a permis d'identifier les parties du code qui prennent le plus de temps. Un exemple est disponible en annexe IV.

Nous avons ensuite procédé à la « cythonisation » du code. Pour cela nous avons dû nous appuyer sur différents sites, exemples et recherches afin d'essayer de maîtriser un minimum ce nouveau langage de codage. Nous avons alors rencontré différents problèmes et avons dû nous adapter rapidement à ce nouveau langage. Nous avons donc introduit la définition de chaque type pour les variables et les fonctions, et nous avons choisis quelles fonctions peuvent être appelées à l'extérieur de chaque classe Cython.

Lors de la modification du code en Cython, nous avons dû développer un fichier 'Setup.py' (voir annexe V) dans lequel nous avons enregistré les commandes afin de cythoniser le programme (c'est-à-dire le convertir en un fichier .c et un fichier .pyc). De plus, la conversion du code en Cython implique de lui donner une extension .pyx afin d'informer les utilisateurs et les compilateurs que le code est écrit en Cython et non plus en Python pur.

Une fois tout cela réalisé, nous avons créé un fichier 'test_cython.py' (voir annexe VI) qui nous permet de voir le nouveau profil de temps d'execution de notre programme Cython.

Base de données	Temps Code python	Temps Code Cython
DIGIT Complet	3.436 secondes	3.224 secondes
DIGIT 100 données	0.336 secondes	0.313 secondes
IRIS Complet	0.249 secondes	0.216 secondes
IRIS 100 données	0.182 secondes	0.171 secondes

Ces données ont été calculées en faisant une moyenne sur 15 valeurs obtenues avec le code, et chacune de ces données possèdent une **erreur de 0.001 secondes** qui correspond à l'erreur indiquée sur la fonction utilisée pour obtenir ces temps (cProfile).

Finalement, l'utilisation de Cython n'a pas augmenté de manière significative le temps de calcul de l'algorithme, cependant Cython permet d'avoir un programme plus clair pour des utilisateurs extérieurs, en effet, nous voyons que chaque variable et fonctions sont définies de manière claire et servent de commentaires au code initial. De plus, on remarque que ce sont les distances euclidiennes et les K-means qui prennent le plus de temps pour réaliser ce programme, et ce quel que soit le nombre de données ou de dimensions de la bibliothèque utilisée, or nous faisons appel à des fonctions déjà optimisée, car présente dans la bibliothèque scikit-learn, pour les calculer et ne pouvons donc pas diminuer le temps de calcul.

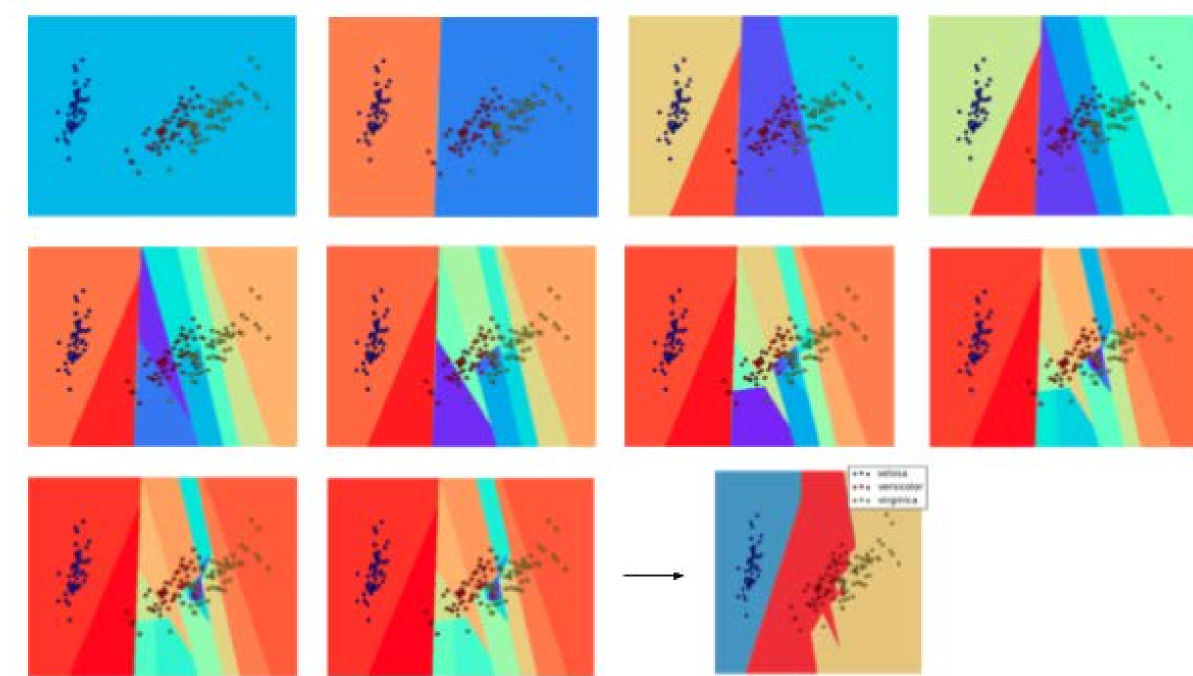
Conclusion

Ce projet, qui avait pour but l'intégration d'une méthode d'apprentissage a été un véritable apprentissage pour notre groupe. Cela a en effet été l'occasion de découvrir les arbres de décision k-means mais aussi les enjeux et les exigences de la rédaction d'un code plus professionnel que ce à quoi nous avions jusque-là étés confrontés.

L'objectif de l'intégration à Scikit-learn ne pouvant être atteint, nous nous sommes concentrés sur l'approfondissement des tests du programme, sa documentation ainsi que sur son optimisation grâce au cython. Ce dernier point est à nuancer car, n'ayant pas appris le langage cython avant ce projet, nous avons optimisé le code en étant toujours dépendant du temps de calcul des k-means à chaque étape.

Enfin cette conclusion est aussi l'occasion de remercier Célia Châtel qui nous a encadrés tout au long de ce projet.

Annexe I - Processus visuel de la division des groupes selon les arbres de décision k-means.



Source: KMeans Decision Tree - Celia Châtel

Annexe II - Structure des tests

Nous utilisons le module unittest de python pour effectuer nos tests unitaires.

Voici un exemple de test :

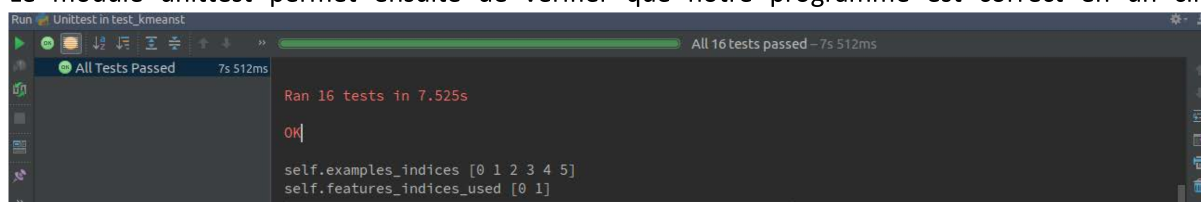
```
def test_maxfeature(self):
    kmdt = KMeansDecisionTree(1)
    assert kmdt.n_features_to_use(0) == 1
    kmdt = KMeansDecisionTree(0.75)
    assert kmdt.n_features_to_use(0) == 1
    assert kmdt.n_features_to_use(1) == 1
```

Le nom de la fonction de test doit commencer par le mot “test”. On appelle ensuite notre fonction et on vérifie que le résultat est bien celui attendu avec la commande “assert”.

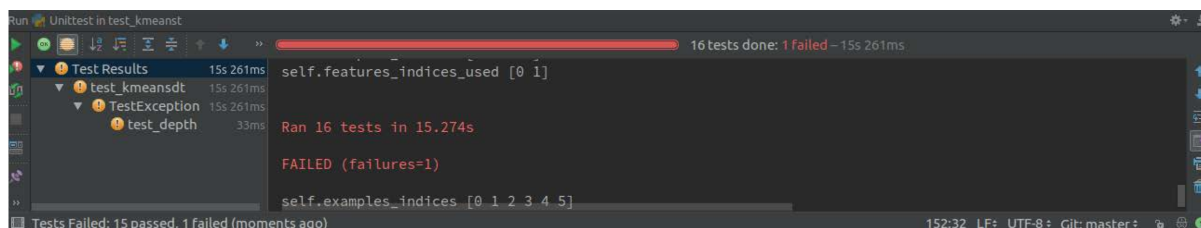
Nous pouvons aussi tester les erreurs avec la méthode assertRaises comme suivant :

```python
def test_raise_maxfeature(self):
    kmdt = KMeansDecisionTree("feature")
    with self.assertRaises(ValueError):
        kmdt.n_features_to_use(1)
```

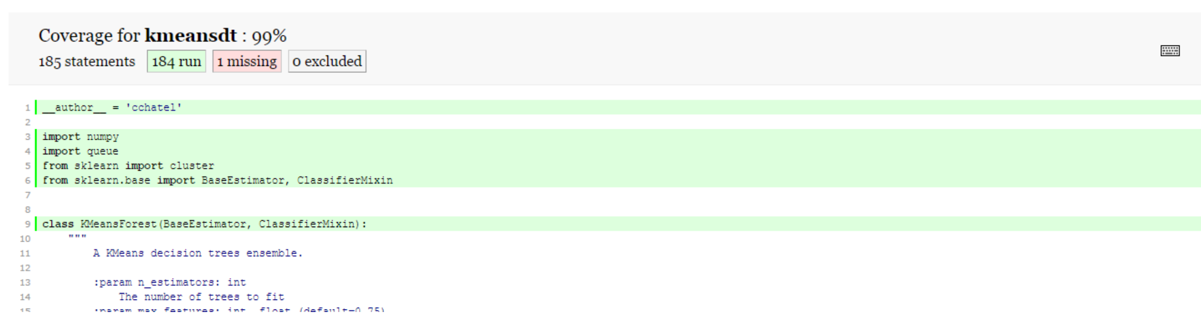
Le module unittest permet ensuite de vérifier que notre programme est correct en un clic.



En cas d'erreur, il indique la fonction qui a posé problème afin de retrouver au mieux l'endroit du code qui pose problème.



Annexe III – Capture d'écran du résultat du test coverage



ANNEXE IV – Extrait de l’analyse du temps d’exécution de chaque partie

1257035 function calls in 3.224 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	3.224	3.224	<string>:1(<module>)
706	0.001	0.000	0.001	0.000	_internal.py:227(__init__)
706	0.000	0.000	0.000	0.000	_internal.py:252(get_data)
5913	**0.253**	**0.000**	**0.253**	**0.000**	**_k_means.pyx:275(__pyx_fuse_1_centers_dense)**
5913	0.062	0.000	0.062	0.000	_k_means.pyx:275(_centers_dense)
1760	**0.593**	**0.000**	**2.123**	**0.001**	**_k_means_elkan.pyx:107(__pyx_fuse_1k_means_elkan)**
1760	0.019	0.000	0.019	0.000	_k_means_elkan.pyx:107(k_means_elkan)
1760	0.019	0.000	0.019	0.000	_k_means_elkan.pyx:32(__pyx_fuse_1update_labels_distances_inplace)
35515	0.022	0.000	0.249	0.000	_methods.py:31(_sum)
…					
5913	0.007	0.000	0.033	0.000	numeric.py:414(asarray)
44141	0.041	0.000	0.051	0.000	numeric.py:484(asanyarray)
1760	0.002	0.000	0.008	0.000	numeric.py:535(ascontiguousarray)
11193	**0.309**	**0.000**	**1.328**	**0.000**	**pairwise.py:162(euclidean_distances)**
11193	0.040	0.000	0.055	0.000	pairwise.py:33(_return_float_dtype)
11193	0.043	0.000	0.804	0.000	pairwise.py:57(check_pairwise_arrays)
176	0.001	0.000	0.003	0.000	queue.py:115(put)
176	0.001	0.000	0.002	0.000	queue.py:147(get)
1	0.000	0.000	0.000	0.000	queue.py:199(_init)
….					
1760	0.012	0.000	0.012	0.000	{method 'random_sample' of 'mtrand.RandomState' objects}
37278	0.240	0.000	0.240	0.000	{method 'reduce' of 'numpy.ufunc' objects}
1	0.000	0.000	0.000	0.000	{method 'remove' of 'set' objects}
1760	0.005	0.000	0.005	0.000	{method 'searchsorted' of 'numpy.ndarray' objects}
21929	0.018	0.000	0.168	0.000	{method 'sum' of 'numpy.ndarray' objects}

ANNEXE V - fichier Setup.py

```
from distutils.core import setup

from distutils.extension import Extension

from Cython.Build import cythonize
```

```python
from Cython.Distutils import build_ext

extensions = [
    Extension("kmeansdt", ["kmeansdt.pyx"])
]
setup(
    cmdclass = {'build_ext':build_ext},
    ext_modules = cythonize(extensions),
)
```

ANNEXE VI - Fichier Test_Cython.py

```python
import pyximport
pyximport.install()
import cProfile
from kmeansdt import KMeansDecisionTree
from sklearn.datasets import load_iris
iris = load_iris()
from sklearn.model_selection import train_test_split
iris_train, iris_test, iris_train_target, iris_test_target = train_test_split(iris.data, iris.target, test_size=0.33)
kmdt=KMeansDecisionTree()
cProfile.run('kmdt.fit(iris_train,iris_train_target)')
```