Universitat de Lleida
Escola Politècnica Superior

24/01/2021

Pràctica 2

Grau en Enginyeria Informàtica

Joel Aumedes Serrano (48051307Y)
Joel Farré Cortés (78103400T)
ESCOLA POLITÈCNICA SUPERIOR – UNIVERSITAT DE LLEIDA

Introducció

En aquesta pràctica hem implementat diferents algoritmes d'aprenentatge supervisat i no supervisat vists a classe.

Arbres de decisió

Per a provar la nostra implementació dels arbres de decisió amb un *dataset* més extens, hem creat un nou *dataset* utilitzant un script que genera registres de forma aleatòria.

Construcció de l'arbre de forma recursiva

Per a construir l'arbre de forma recursiva simplement busquem la millor partició possible i després construïm el *decisionnode* cridant a la mateixa funció de forma recursiva a la branca *True* i a la branca *False*.

Per a trobar la millor partició, calculem el guany que ens generaria les preguntes possible que en cas de que les dades siguin arbitràries, fem una pregunta per a cada valor possible i en cas de que siguin numèriques, preguntem per una sèrie de números des del mínim trobat fins al màxim, amb una separació entre els números que es passa per paràmetre i que per defecte es 0,1.

Construcció del arbre de forma iterativa

Per a la constricció de l'arbre de forma iterativa primer que tot ens guardem una referència al node arrel. Aleshores utilitzem una llista on guardem els nodes que falta particionar, cada node guardarà els registres que ha de particionar al seu atribut *results* mentre no sigui processat. Per a processar un node trobem la millor partició utilitzant la mateixa funció que de forma recursiva, aleshores es poden donar dos casos. Si el millor guany es més gran que el paràmetre *beta*, vol dir que aquest node no és una fulla, per tant l'hi modifiquem els atributs necessaris i l'hi guardem una referència als seus nodes fills que encara no han sigut processats. Aquests nodes fills s'afegeixen a la llista nodes per a que siguin processats. En canvi, si el millor guany no és més gran que *beta* vol dir que el node es una fulla i per tant no creem més nodes.

L'estructura orientada a objectes de Python ens permet guardar aquestes referències de manera que al acabar tot el procediment, el node arrel tingui referències a tots els altres nodes.

Funció de classificació

La funció de classificació recorre l'arbre fent preguntes als nodes fins a trobar un node fulla. A cada node es realitza la pregunta corresponent i segons si el resultat dóna cert o fals, es continua per la branca *True* o per la branca *False*.

Clustering

Distància total

L'atribut *intertia_* la calculem a dins la funció *fit()*. Degut a que en el proper apartat de les *Restarting policies* l'algoritme es realitza múltiples vegades, la *intertia_* es calcula totes les vegades i escollim la millor ja que acabarà essent la que realitza una millor aproximació.

Distància en funció de *k*

Per a mostrar aquest gràfic, realitzem un número d'execucions amb diferents valors de *k* i guardem totes les *intertia_* en una llista. Aleshores creem un simple gràfic de barres mitjançant la llibreria *matplotlib*.

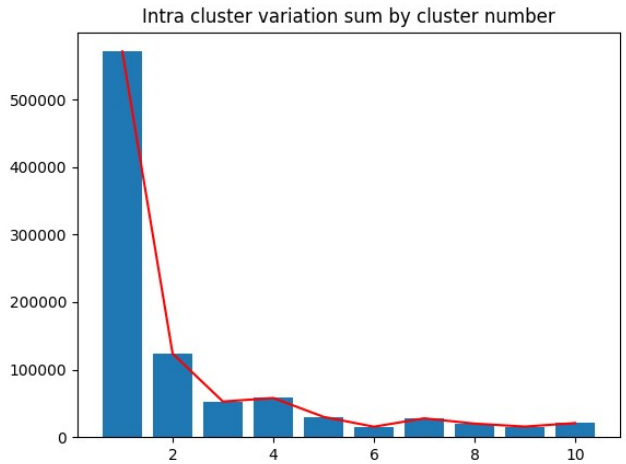
Restarting policies

Les *restarting policies* són parametritzades amb el paràmetre *execution_times* que per defecte s'estableix a 5. S'utilitza dins la funció *fit()* on tot el procés de generar *clusters* aleatòriament i calcular els *clusters* i la *inertia_* es repeteix tantes vegades com dicti *execution_times* i es manté la versió que una *inertia_* més petita i això vol dir que els *clusters* estàn més lligats entre sí.

Escollir un valor de *k*

Per a escollir un valor de *k* hem utilitzat dos mètodes diferents: el *Elbow Method* i el *Silouette Method*.

L'*Elbow Method* consisteix en calcular la suma de variació *intra-cluster* de tots els clusters per un rang de K's i representar-la en una gràfica. Per calcular la suma de *variació intra-cluster* d'un *cluster*, calculem les distàncies de tots els punts d'un cluster entre ells. Aleshores, en aquesta gràfica localitzar el colze (*elbow*), que és on la funció decrementa més fortament.



Com podem veure, el colze es troba a 2, que és el número òptim de *clusters* d'aquest *dataset*.

El següent mètode, el *Silhouette Method*, consisteix en calcular la mitjana de la Silhouette dels punts un cop s'ha realitzat l'execució de l'algoritme en un rang de K's.

A partir del número de *clusters*, podem calcular la *Silhouette* mitjana i la *Silhouette* d'un punt amb les següents fórmules:

$$S(k)=\frac{\sum_{i=0}^{Número\ de\ punts} s(i)}{Número\ de\ punts} \quad s(i)=\frac{b(i)-a(i)}{max(a(i),b(i))}$$

El paràmetre *a* calcula la mitjana de la distància del punt que estem calculant a tots els altres punts del mateix cluster. Com més proper a 0 sigui aquest número, millor. Es calcula de la següent forma:

$$a(i)=\frac{\sum_{j=0,\,j\neq i}^{j=|C_i|} euclideanSquared(i,j)}{|C_i|-1}$$

Dividim per $|C_i|-1$ja que no calculem la distància$scoref(i,i)$. Si el punt *i* és l'únic punt del cluster, el paràmetre *a* és 0.

El paràmetre *b* calcula la mitjana de la distància del punt que estem calculant a tots els punts del *cluster* que està més aprop seu, però del que no forma part. Com més gran sigui, millor. Es calcula amb aquesta fórmula:

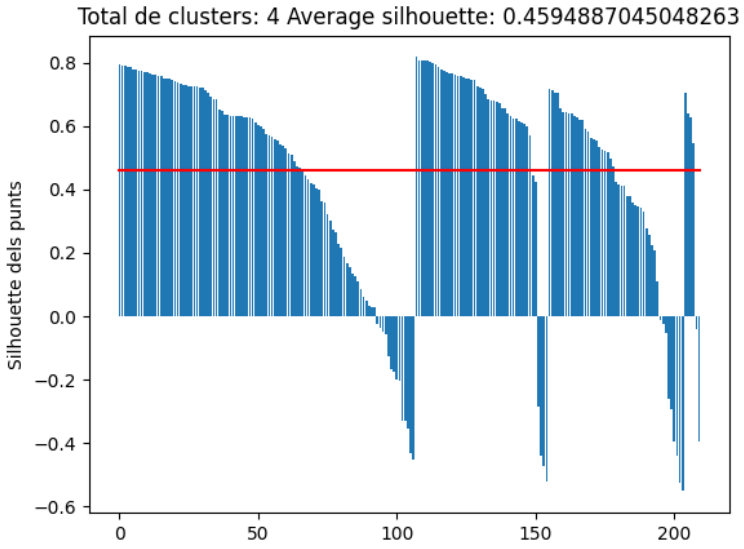
$$b(i)=min\left(d\ \underset{j=0}{\overset{j=k,\,j\neq i}{(C_j)}}\right)$$

En aquesta fórmula, utilitzem el mínim per a agafar solament la mitjana de distàncies més propera, que calculem amb *d(j)* que és la distància mitja des del punt i a tots els punts del cluster j i que calculem així:

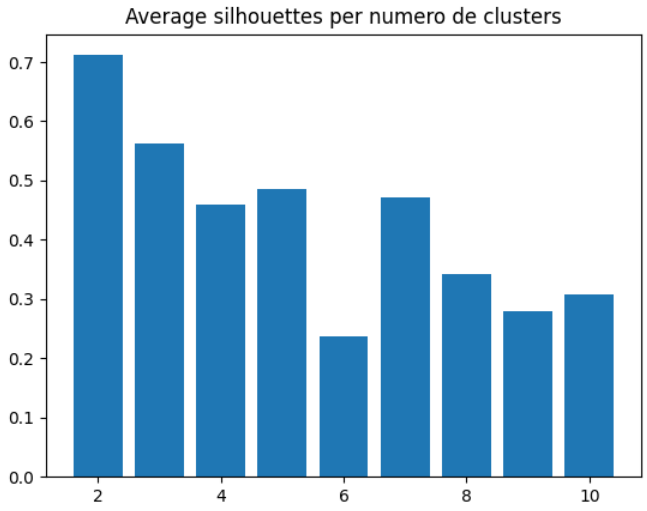
$$d(i,C_j)=\frac{\sum_{l=0}^{l=|C_j|} euclideanSquared(i,l)}{|C_j|}$$

Aleshores, tornant a la fórmula de la Silhouette *s(i)*, veiem que prendrà un valor entre 1 i -1. Com més proper a 1 millor, ja que vol dir que aquest punt està molt aprop dels punts del seu cluster, i molt lluny de qualsevol altre cluster. En canvi, si el resultat és negatiu, vol dir que aquest punt no ha estat assignat correctament, ja que està més aprop d'un altre cluster que del seu.

Per saber quin valor de *k* és millor, hem de calcular la mitjana de *Silhouettes* i agafar la màxima. També podem crear una gràfica de les S*ilhouettes* del *cluster* per veure les S*ilhouettes* de cada punt. Si hi ha algun punt negatiu vol dir que està mal assignat.



Aquí podem veure que amb 4 *clusters* hi ha punts que estan mal assignats.



Finalment podem veure que el valor màxim es troba al 2, que és el mateix resultat que hem obtingut amb el *Elbow Method*.

Scikit-learn

Missing data

Primer de tot, llegim els fitxers *.csv* mitjançant la funció de *pandas: read_csv*. Imprimint les cinc primeres dades de cada *set* ja podem veure que hi han dades que falten.

Per a computar aquestes dades que falten *sklearn* disposa de la llibreria *sklearn.impute* que proporciona diferents mètodes per a calcular aquestes dades que falten. Les classes que hem utilitzat en aquesta pràctica són: *SimpleImputer* i *KNNImputer*.

La classe *SimpleImputer* consta de les següents estratègies, *constant*, *mean*, *median* i *most_frequent*. L'estratègia constant substitueix tots els valors que falten per un valor que es passa per paràmetre. En el nostre cas aquest valor ha sigut 0. L'estratègia *mean* substitueix els valors que falten per la mitjana d'aquella columna. L'estratègia *median* les canvia per la mediana de la columna. Finalment, l'estratègia *most_frequent* les canvia per la dada que més es repeteix en la columna.

La classe *KNNImputer* rep per paràmetre un número de veïns i una estratègia de dues possibles, *uniform* o *distance*. El que fa aquesta *Imputer* és calcular les dades que falten a partir dels valors dels seus veïns, calculats amb la distància *Euclidean*. *KNNImputer* calcula la mitja dels veïns si l'estratègia és *uniform*. Si l'estratègia és *distance* calcularà la mitjana ponderada donant més pes als veïns que estan més prop dels veïns que ens interessen.