Pràctica 3
Arbres de decisió i clustering

Ferran Aran Domingo
Oriol Agost Batalla

48253731F
48257094N

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Arbres de decisió

Construcció de l'arbre iterativament

Per a construir el arbre iterativament, hem usat dues llistes que tenen un comportament de cua (LIFO).

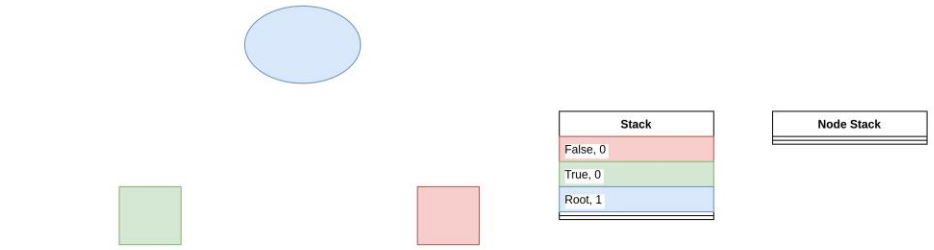
Una d'elles emmagatzemarà estats, per a saber en quin punt de la construcció es troba cada node.

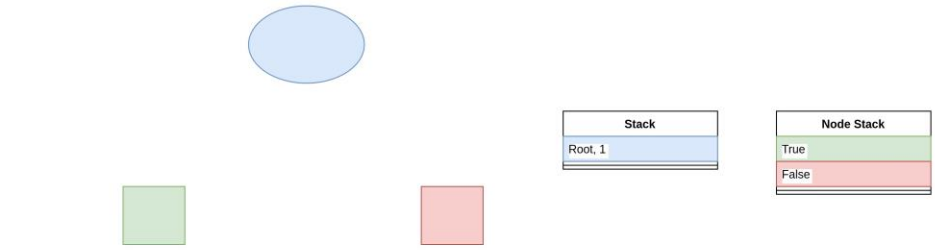
Aquest poden ser 0, que voldrà dir que encara s'han de construir els seus fills, o 1, que voldrà dir que els seus fills ja s'han creat i només hi falta crear el node. Només carregarem en context 1 aquells nodes que no són una fulla.

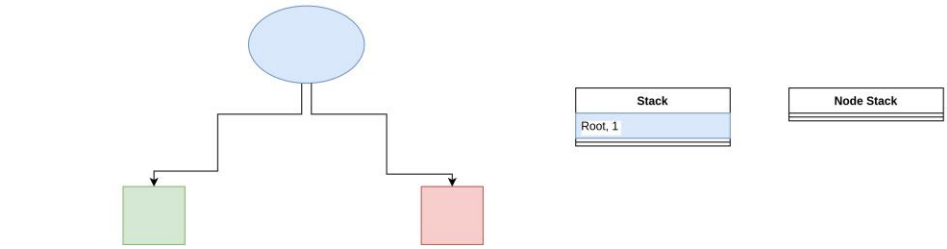
Ens aprofitem doncs de la stack de nodes ja que sabem que si hi ha un node en context 1 a ser tractat, els últims dos nodes que s'han afegit a la cua seran els seus "fills".

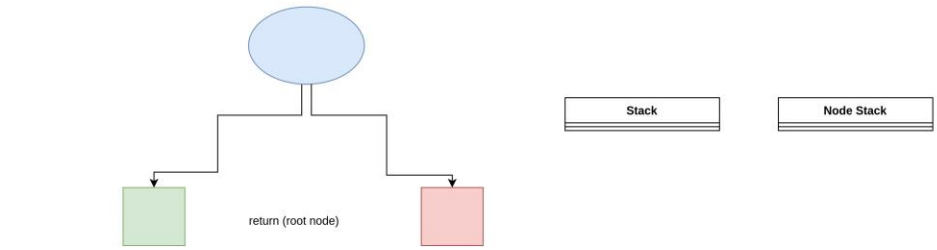
L'estructura, resumida en un arbre de només una pregunta, seria la següent:


Stack
Root, 0
Node Stack


Stack
False, 0
True, 0
Root, 1
Node Stack


Stack
Root, 1
Node Stack
True
False


Stack
Root, 1
Node Stack


Stack
Node Stack
return (root node)

Funció de classificació

En aquesta, anem recorrent el arbre construït fins que trobem una fulla i retornem la predicció d'aquesta fulla.

Si en aquesta hi havia dues prediccions, teníem dues opcions:

- Retornar la classe més comuna en el resultat
- Retornar una classe "aleatòria", amb les probabilitats que tenien les diferents classes. Si per exemple teníem 47 de l'etiqueta "virginica" i una de "versicolor", en cada 47/48 cops fariam la predicció "virginica" i en 1/48 "versicolor"

Ens hem decantat per la classe mes comuna, tot i que si hi ha empat en les claus, retornem una aleatòria d'aquestes, per a no retornar sempre la mateixa de les dues i ser "injustos".

Poda de l'arbre

El primer que hem fet ha set guardar el guany, goodness, canvi d'entropia en els Decision nodes a l'hora de construir l'arbre.

Després, per a fer la poda de l'arbre, primerament mirem si el fill esquerre del node passat és una fulla i si no és així, fem una crida recursiva a la funció. Després, fem el mateix amb el fill dret.

Això ens permet implementar la poda de baix a dalt, començant pels nodes no fulla més baixos.

Després, el que fem és mirar si els dos fills del node són fulles, i si és així, cas en el que podem fer merge; mirem si el guany d'aquest node és menor que el threshold passat per paràmetre, moment en el que haurem de fer el prunning d'aquest.

Per a fer aquest prunning, "resetejem el node", posant la columna a –1, el valor a None, el goodness a 0 i el fill esquerra i fill dret a None. Finalment, pose'm en el camp results d'aquest node el resultat de ajuntar els dos camps results dels fills.

Test performance

Per a fer el test_performance, el que fem es que donada una llista, per cada item calculem la etiqueta predita i mirem si aquesta és la que realment té aquest item. Si és així, sumem 1 a un comptador.

Finalment retornem el comptador entre la longitud de la llista que ens ha entrat.

Per tant, aquest valor final oscil·larà entre 0 i 1.

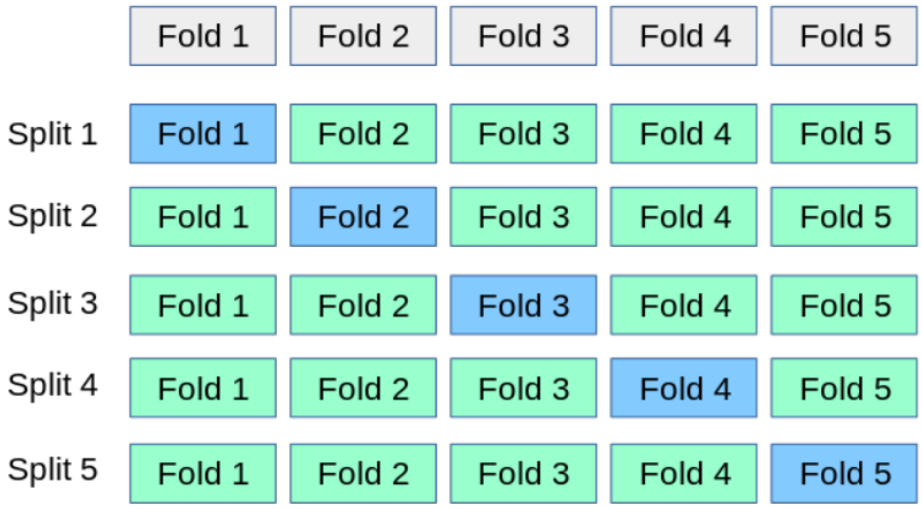
Cross-validation

Per a implementar el mètode de cross-validation, hem seguit el pseudo-codi que se'ns proporcionava en els apunts.

En aquest , primerament settejem la seed si en tenim. Després, randomitzem el dataset entrat, fem les k particions (valor que se'ns entra per paràmetre de les dades) i després, entrem en un bucle, que tindrà tantes iteracions com particions.

En aquest, separem les dades en train i test, on les dades de train seran totes les particions menys aquella que té com índex l'índex d'iteració actual del bucle (imatge a continuació). Enterenem el model, li apliquem el prunning, calculem l'accuracy amb les dades de test i finalment afegim aquest valor en una llista.

Finalment, apliquem la funció d'agregació (mitja en aquest cas) a la llista d'accuracy's i retornem aquest valor.



Millor paràmetre de threshold

Per a trobat el millor paràmetre de threshold, primerament partim les dades en train i test, usant en el nostre cas un 20% de les dades com test.

A continuació, calculem diferents scores, fent increments. Si per exemple indiquem que volem 10 iteracions, calcularem una sèrie de scores amb threshold de 1/10 * nombre d'iteració.

D'aquesta manera, parametritzem el nombre d'iteracions que volem fer.

Si el resultat obtingut és major que el que teniam emmagatzemat, emmagatzemem aquest en comptes del que teníem.

Finalment, entrenem un nou arbre amb el conjunt de dades training, el prune'm amb el threshold que ens ha reportat millor score i calculem el seu accuracy amb les dades de test, retornant aquest valor.

Tot i així, ens hem adonat de que aquest càlcul no sembla tenir molt efecte, i que el accuracy abans i després de prunar el arbre amb les dades de test és el mateix. Això segurament sigui degut a que tenim un conjunt de dades "petit".

Clustering

En aquesta segona part de la pràctica se'ns demana que afegim funcionalitat a la funció que aplica l'algoritme K-Means, usant sempre la distància euclídea al quadrat normalitzada (on 1 representa dos vectors idèntics).

Cal destacar que per tal de fer mes entenedor l'algoritme hem decidit crear una classe que el representi, de forma que segons es va afegint funcionalitat van apareixent nous mètodes que la implementen. D'aquesta manera des del *main* podem anar mostrant els resultats obtinguts amb l'algoritme en les diferents etapes de millora.

Total distance

La responsabilitat d'obtenir les distàncies de cada *item* al seu centroide cau sobre el mètode *_find_closest_centroids()* que, apart de retornar els *clusters* formats (per cada centroide, quins *items* li han estat assignats) també retornarà una llista amb les distàncies.

Per saber quins son els valors dels centroides, usem el mètode *_move_centroids()*, que ens actualitza el valor dels atributs de tots els centroides.

Per tant, des de *find_clusters()*, un cop han retornat els dos mètodes esmentats, ja podem sumar tots els valors de la llista de distàncies i retornar-ho juntament amb la llista de centroides.

Restarting policies

Se'ns demana que afegim la funcionalitat de poder parametritzar un determinat nombre de reinicis de l'algoritme, de manera que ens quedem el millor resultat basant-nos en la distància total obtinguda (busquem maximitzar-la).

Aquesta funcionalitat queda implementada en el mètode *find_best_start_config()*, que farà tantes iteracions (on en cada iteració aplica k-means) com vinguin donades per paràmetre, quedant-se sempre amb el millor resultat obtingut, un cop acabat el bucle retornarà una tupla amb el mateix format que en l'apartat anterior, però essent aquesta la que ha obtingut la major distància total.

Distància en funció de k

Per a poder donar el valor de la distància total donada una *restarting policy* i parametritzant el valor de la k, hem implementat el mètode *test_different_kvalues()*, que rep un rang de valors que anirà prenent la k, i el nombre de reinicis (iteracions de la *restarting policy*) que volem que es facin per cada valor de k.

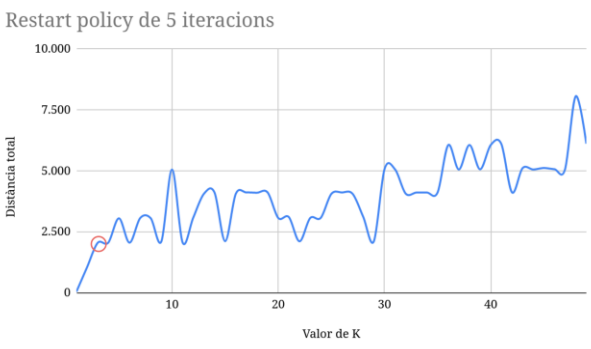
Un cop acabats els càlculs ens retorna una llista amb totes les distàncies màximes obtingudes.

Escollir valor de k

El mètode del colze és una heurística usada en tècniques de *clustering* per determinar en quin punt deixa de valer la pena incrementar el nombre de *clusters*. Aquesta decisió vindrà donada per l'anàlisi d'un gràfic, on es vegi la millora dels resultats obtinguts en funció del nombre de *clusters*.

En el nostre cas, la millora dels resultats obtinguts la podem representar amb la distància total obtinguda, així doncs, buscarem en la corba del gràfic un 'colze' que representi el moment en que l'increment de qualitat dels *clusters* obtinguts no es prou significatiu com per a que valgui la pena afegir un nou *cluster*.

Després de provar amb tota mena de combinacions de valors, hem escollit dos gràfics com a subjectes de l'anàlisi;



En el de l'esquerra tenim u rang de valors de k que va des de 1 fins a 19 amb 100 iteracions de *restart policy*. Al de la dreta la k va des de 1 fins a 50 i tenim nomes 5 iteracions de *restart policy.*

En el primer, el fet de tenir tants reinicis ens permet allunyar-nos una mica de l'indeterminisme que suposa la inicialització dels centroides, aconseguint d'aquesta forma uns resultats més sòlids en els que basar la cerca del 'colze', determinant així el nombre de centroides més òptim per el *dataset*.

En el segon, hem prescindit pràcticament dels reinicis, fent-ne només 5 per cada valor de la k. I perquè ens interessa aquest segon gràfic? Doncs ens serveix per reafirmar la decisió d'escollir com a 'colze' el valor k=4, ja que ens permet atribuir l'inestable increment que precedeix als valors superiors a k=4 al sobre entrenament del model.

Aquesta última conclusió la deduïm arrel de que en el *dataset* només hi ha 100 Blogs, i tot i que el gràfic ens diu que pel valor de k=49 hem obtingut el millor resultat, la lògica ens diu que no te cap sentit classificar 100 blogs en 49 categories diferents. Per tant, atribuïm aquest fet al sobre entrenament del model i prenem com a millor valor el de k=4.