eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Gradu Amaierako Lana

Informatika Ingeniaritzako Gradua

Konputazioa

Zuhaitz kontsolidatu partzialik hoberena garatzeko irizpide desberdinen azterketa

Josué Cabezas Regoyo

Zuzendaria:

Txus Pérez de la Fuente

2023.eko irailaren 17

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Gradu Amaierako Lana

Informatika Ingeniaritzako Gradua

Konputazioa

Zuhaitz kontsolidatu partzialik hoberena garatzeko irizpide desberdinen azterketa

Josué Cabezas Regoyo

Zuzendaria:

Txus Pérez de la Fuente

2023.eko irailaren 17

ESKERRAK

Eskerrak gurasoei, nire anai-arrebei eta lagunei.

Eta eskerrik asko Txus-i bere laguntzagatik eta nirekin izan duen pazientzia guztiagatik. Bera gabe ezin izango nuen lan hau aurrera eraman. Eskerrik asko!

Argibidea

1 LABURPENA

Proiektu honen helburua ALDAPA taldeak diseinatutako PCTBagging (*Partially Consolidated Tree Bagging*) algoritmoaren aldaera berri bat WEKA plataforman inplementatzea da.

PCTBagging gidatua izeneko algoritmo berri honek zuhaitza kontsolidatu partziala hainbat modutan eraikitzea bilatzen du, ahalik eta errendimendurik onena lortzeko. Horretarako, parametro batzuk gehitzen dira, eta, horien bidez, erabiltzaileari aukera ematen zaio zuhaitz kontsolidatua garatzeko modua aukeratzeko.

PCTBagging algoritmoan, zuhaitza osoa garatzen da eta gero adabegi txikienak kentzen dira. PCTBagging gidatuan, ordea, erabiltzaileak hainbat irizpideen artean aukeratzeko aukera izango du garatu beharreko hurrengo nodoa hautatzeko, irizpide diskriminatzailearekiko (*gain ratio*) edo preordenarekiko, besteak beste. Era berean, erabiltzaileak zuhaitz kontsolidatuaren nodo-kopuru edo maila-kopuru maximoa hautatzeko aukera izango du ere. Horrela, erabilitako irizpidearen arabera, sortutako zuhaitz kontsolidatu partziala desberdina izango da eta beraz, honek ematen dituen emaitzak ere.

Esperimentazioa 33 datu-baserekin egin da eta hainbat algoritmo alderatu dira. CTC algoritmoa, PCTBagging algoritmoa eta beste algoritmo lehiakide batzuk PCTBagging gidatua algoritmo berriaren bertsio desberdinekin konparaketa egin da. Probak eta esperimentazioa WEKAn egin dira.

Gako hitzak: Ikasketa automatikoa, sailkapen zuhaitzak, zuhaitz kontsolidatu partziala, PCTBagging gidatua, PCTBagging algoritmoa, C4.5 algoritmoa, CTC algoritmoa, Bagging algoritmoa.

2 SARRERA

PCTBagging, *Partially Consolidated Tree Bagging*, ALDAPA ikerketa taldeak diseinatutako sailkapen algoritmoa da. PCTBagging algoritmoa zuhaitz kontsolidatua eraikitzeko algoritmoaren (*CTC*) eta *Bagging* sailkatzaile anitzaren arteko hibridoa da. Ikuspegi honetan zuhaitz kontsolidatu bat sortzen da lagin-talde bat erabiltzen eta ondoren Bagging metodoa aplikatzen da gainerako zuhaitzak modu independentean garatzeko.

CTC algoritmoarekiko, PCTBagging algoritmoak *Kontsolidazio portzentaia* izeneko parametro bat dauka, zuhaitz kontsolidatuaren zenbat barne-nodo mantendu behar diren zuhaitz osoaren nodo kopuruen aldean adierazten duena. Horrela, lehenengo zuhaitza eraikitzen da eta gero inausketa prozesu bat hasten da non bakarrik barne nodo handienak uzten diren, errotik hasiz parametroak ematen duen balioraino.

Proiektu honetan PCTBagging-aren aldaera bat proposatzen da, **PCTBagging gidatua** algoritmoa. Honek inausketa-prozesua eta zuhaitz kontsolidatu osoaren eraikuntza saltatu eta zuhaitz partziala zuzenean eraikitzea bilatzen du parametroak emandako balioraino, guztiz eraiki behar izan gabe. Horrela, zuhaitz kontsolidatu partzial bat lortzen da eta Bagging-a aplika daiteke. Gainera, zuhaitz partziala eraikitzeko irizpide desberdinak azter daitezke sailkapen-gaitasunaren ikuspegitik ahalik eta zuhaitz partzial onena lortzeko. Aldaera berriaren kasuan, kontsolidazio-ehunekoa portzentaia baten ordez balio bat izan ahalko da. Ehunekoa bada, zuhaitz osoa eraiki behar da zuhaitz osoak izango lituzkeen nodoak jakiteko, eta hori oso zamatsua da; horregatik, erabiltzaileak balio finko bat aukeratu ahal izango du.

Proiektua garatzeko, WEKA erabili da. Kode irekiko software-tresna da, eta datu-meatzaritzaren eta ikaskuntza automatikoaren esparruan erabiltzen da. Algoritmo eta teknika sorta zabala eskaintzen du datuak aztertzeko, sailkatzeko, erregresiorako, taldekatzeko eta bistaratzeko. Proiektua garatu ahal izateko, lehenengo zatian, WEKA plataforma eta bertan inplementatuta dagoen PCTBagging inplementazioa, J48PartiallyConsolidated izenekoa, ulertu eta aztertu beharko dira, eta, ondoren, proposatutako aldaera integratuko da.

33 datu-base erabili dira WEKAk berak eskainitako tresnaren bidez inplementazioaren egokitasuna eta kalitatea aztertzeko. Datu-base horiek dira ALDAPA taldeak erabili ohi dituenetako batzuk.

3 PROIEKTUAREN HELBURUA DOKUMENTUA

Proiektuaren helburua *PCTBagging* algoritmoaren egokitzapen bat WEKAn inplementatzea eta integratzea da. WEKA plataforma prestatuta dago lehendik dagoen kodea aldatzeko beharrik gabe algoritmo berriak gehitzeko.

Proiektuaren helburu nagusiak hauek dira:

1. C4.5, CTC eta PCTBagging algoritmoen WEKAko inplementazioak aztertu.

2. WEKA plataforma ezagutzea, aplikazioaren atal desberdinak nola egituratzen diren ezagutuz eta WEKAn algoritmo berri bat integratzeko modua bilatuz, aplikazioaren interfaze grafikoaren bidez erabilera ahalbidetzeko eta inplementatzeko. WEKAk ezartzen dituen betebeharrak aztertu eta herentzia egitura erabaki.

3. Algoritmo errekurtsiboak iteratibo bihurtzeko modua aztertu eta PCTBagging gidatua algoritmo berria inplementatu.

4. Irizpide berriak inplementatu, lehenengo *J48* klasean (C4.5 algoritmoa) eta gero *J48PartiallyConsolidated* klasean (CTC algoritmoa).

5. Azterketa esperimental bat egitea WEKAn inplementatu berri den algoritmoaren baliozkotasuna egiaztatzeko helburuarekin. Azterketa honek datu-base multzo bat erabiltzea dakar, non proba zehatz batzuk egingo diren. Lortutako emaitzak bere lehiakide zuzenenekin alderatuko dira, hau da, antzekotasun batzuk dituzten sailkatzaileekin, eta kontsolidazio-irizpide ezberdinetatik ebaluatuko dira, haien errendimendua ikuspuntu ezberdinetatik behatu eta aztertzeko aukera emanez.

4 ABIAPUNTUA

Proiektu honen abiapuntua *PCTBagging* algoritmoa da. *PCTBagging* algoritmoa ALDAPA ikerketa taldeak sortu zuen *CTC* algoritmoan oinarrituta *Bagging* sailkatzaile anitzarengana azalpena mantenduz hurbiltzeko.

Proiektu honen helburua *PCTBagging*-en oinarritutako algoritmo berri bat diseinatzea, integratzea eta probatzea da, zuhaitz partzial kontsolidatu bat zuzenean eraikitzea eta eraikuntzarako irizpide desberdinak aukeratzea ahalbidetzen duena.

WEKA Java programazio-lengoaian inplementatuta dago eta kode irekikoa da; horri esker, edozein garatzailek bere algoritmoak integratu eta probatu ditzake. WEKAren erabiltzailearen gidan, bere funtzionamendua eta kodea nola aldatu azaltzen da [1].

Proiektu honetan erabilitako informazio-iturri nagusia ALDAPA taldearen *PCTBagging Algorithm* [2] artikulua da. Gainera, Jesús Ma Pérez de la Fuente-ren doktore-tesia kontsultatu da, *Consolidated Trees: Building a Classification Tree Based on Multiple Subsamples without Sacrificing Explanation* [3] izenekoa, non CTC algoritmoa zehatz-mehatz azaltzen den.

Garrantzitsua da nabarmentzea bai CTC algoritmoa bai PCTBagging algoritmoa Waikato-ko Unibertsitateak onartu zituela eta WEKAren iturburu-kodean daudela inplementatuta, *J48Consolidated* eta *J48PartiallyConsolidated* izenpean, hurrenez hurren.

Hori guztia jakinda, algoritmo berria diseinatu eta inplementatuko da.

5 KONTZEPTU TEORIKOAK

Atal honetan proiektua ulertzeko jakin behar diren kontzeptu garrantzitsuenak azalduko dira.

5.1 IKASKETA AUTOMATIKOA

Ikasketa Automatikoa (*Machine Learning*, ingelesez) Adimen Artifizialaren adar bat da, datuetatik abiatuta ikasteko eta erabakiak hartzeko gai diren ereduak eta algoritmoak garatzera bideratzen dena. Ikasketa Automatikoari esker, ordenagailuek datuetatik 'ikasten' dute eta haien arteko ereduak edo erlazioak aurkitzen dituzte. Ordenagailuak atazak autonomiaz egiteko gai dira, zeregin zehatz bakoitzerako berariaz programatu beharrik gabe.

Ikasketa Automatikoan eredu bat entrenatzen da entrenamenduko datu-multzo bat erabiliz; datu-multzo hauek ezaugarri eta emaitza ezagunak dituzten adibideak edo instantziak dira. Ereduak datu horietatik abiatuta ikasten du, eta orokortu daitezkeen ereduak edo arauak sortzen ditu, aurretik ikusi gabeko datu berriei buruzko iragarpenak egiteko edo erabakiak hartzeko.

Mota desberdineko ikasketa automatikoak daude, baina garrantzitsuenak Ikasketa Gainbegiratua eta Ikasketa Ez-Gainbegiratua dira. Ikasketa Gainbegiratuan, eredua etiketatutako datuak erabiliz entrenatzen da, hau da, datuen ezaugarriak eta espero den erantzuna ematen zaizkio. Adibidez, autoen eta hegazkinen irudiak eman daitezke dagozkien etiketekin batera (autoa edo hegazkina da), eta ereduak bakoitza bereizten duten ezaugarriak ezagutzen ikasten ditu. Etiketa hauei **klase** deitzen zaie. Kasu honetan, klaseak autoa eta hegazkina izango lirateke. Proiektu hau Ikasketa Gainbegiratuan zentratzen da.

Beste aldetik, Ikasketa Ez-Gainbegiratuan, eredua etiketatu gabeko datuak erabiliz entrenatzen da, horrek esan nahi du datuen ezaugarriak ematen direla soilik, espero den erantzunik gabe. Honen helburua patroiak, egiturak edo multzoak aurkitzea da.

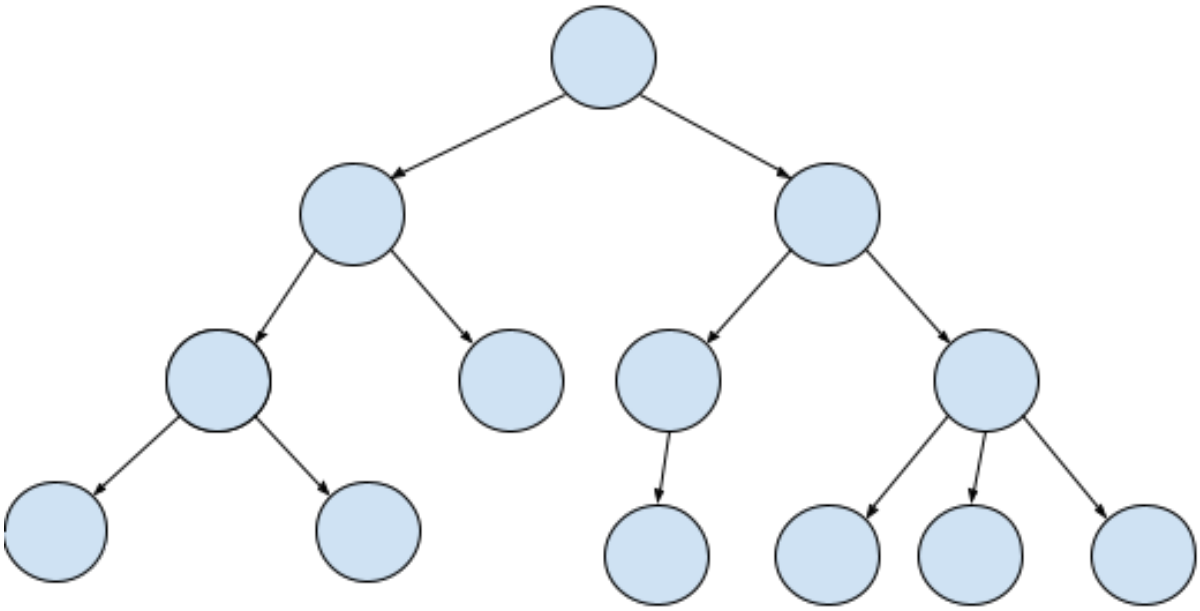
5.2 ERABAKI-ZUHAITZAK

Erabaki-zuhaitzak Ikasketa Gainbegiratuko metodo bat dira, eta zuhaitz-egitura batean erabakiak hartzeko eredu baten irudikapen grafikoan oinarritzen dira.

Erabaki-zuhaitz bat barne-nodoz eta hostoz osatuta dago. Barne-nodoak amaierakoak ez direnak dira, hau da, ume bat edo gehiago dutenak eta sarrerako datuen ezaugarrietan egiten diren baldintzak adierazten dituztenak. Bestalde, hostoak bukaerako nodoak

dira, umerik ez duten nodoak; hosto hauek emaitzak edo irteerak adierazten dituzte. Hosto bakoitza azken erabaki edo sailkapen bati dagokio ebaluatutako baldintzetan eta irizpideetan oinarrituta. Hostoek erabaki-zuhaitz baten erabakiak hartzeko prozesurako funtsezkoak dira, ereduaren azken erantzuna edo aurreikuspena ematen baitute.

Erabaki-zuhaitzak bitarrak edo ez bitarrak izan daitezke. Bitarrak barne nodo bakoitzak zehazki bi ume dituztenak dira, baldintza bakoitzak bi emaitza posible izango ditu. Bestalde, bitarrak ez diren erabaki-zuhaitzetan, barne-nodo bakoitzak bi nodo ume baino gehiago izan ditzake. Horrek esan nahi du egindako probek bi emaitza posible baino gehiago izan ditzaketela. Jarraian, 1. irudian, zuhaitz baten adibide bat dago, nodo mota desberdinekin (bitarrak edo ez-bitarrak).



Irudia 1: Sailkapen zuhaitz baten adibidea

Zuhaitz bat eraikitzeko prozesuan, sarrerako datuak talde txikiagoetan banatzen dira, ezaugarri edo atributu jakin batzuen arabera. Kasu guztiak erro izeneko nodo bakarrean kokatzen dira, hau da, hasierako nodoan. Atributu bat hautatzen da eta populazioa iterazio bakoitzean zatitzen da algoritmoak lortutako baldintza onenaren arabera, eta zuhaitza sortzen da. Zatiketa gehiago egiteko baldintzak betetzen ez badira, nodoa hosto bihurtzen da.

Sailkapen-zuhaitzen abantailetako bat azalpen-gaitasuna da, interpretatzeko egitura intuitibo eta errazari esker. Erabakiak hartzeko prozesua argi eta garbi ikusteko aukera ematen dute. Horrek erraztu egiten du emaitzak nola lortzen diren ulertzea.

5.2.1 INAUSITAKO ETA KOLAPSATUTAKO ZUHAITZAK

Erabaki-zuhaitzei bi prozesu aplika dakizkieke zuhaitzen errendimendua hobetzeko; prozesu hauek inausketa eta kolapsatzea dira.

5.2.1.1 KOLAPSOA

Kolapsatzeko prozesua (ingelesez *collapsed*) erabaki-zuhaitza sinplifikatzen saiatzen da. Horrek esan nahi du informazio gehigarririk ematen ez duten eta beharrezkoak ez diren tarteko nodoak kentzea, eta, horrela, zuhaitza sinplifikatu egin daiteke, sailkatzeko gaitasuna mantenduz.

5.2.1.2 INAUSKETA

Zuhaitza kolapsatu ondoren, askotan, zuhaitza gehiago sinplifikatu daiteke. Erabaki-zuhaitz bat inaustea haren tamaina murrizteko prozesua da, haren errendimenduan laguntzen ez duten adar edo nodo batzuk kenduz. Inausketaren helburua zuhaitza sarrera-datuetara gehiegi egokitzen denean eta orokortzeko gaitasuna galtzen duenean gertatzen den gaindoikuntza saihestea da. Zuhaitzaren konplexutasuna murriztean, eredu sinpleagoa eta errazagoa lortzen da ikusi gabeko datuetan jarduna hobetzen laguntzen duena interpretatzeko eta entrenatzeko. Eredua aplikatzeko behar den denbora murrizten lagun dezake.

Inausketarako bi ikuspegi nagusi daude:

- **Inausketa beherantz (pre-pruning):** Ikuspegi honetan zuhaitza osoa garatu aurretik eraikuntza gelditzeko irizpide batzuk ezartzen dira. Eraikuntzan zehar irizpide horiek ebaluatzen dira eta zuhaitzaren eraikuntza gelditzen da betetzen direnean.
- **Inausketa gorantz (post-pruning):** Ikuspegi honetan zuhaitza osoa eraikitzen da eta ondoren inausketa prozesua hasten da. Zuhaitzaren errendimendua ebaluatzen da eta errendimendua nabarmen hobetzen ez duten adarrak edo nodoak ezabatzen dira.

5.3 ERLAZIONATUTAKO ALGORITMOAK

Jarraian, lan honetan garatu den PCTBagging gidatuarekin lotuta dauden algoritmoen funtzionamendua azalduko da, garrantzitsua da horiek ezagutzea lana behar bezala ulertzeko.

5.3.1 C4.5 algoritmoa

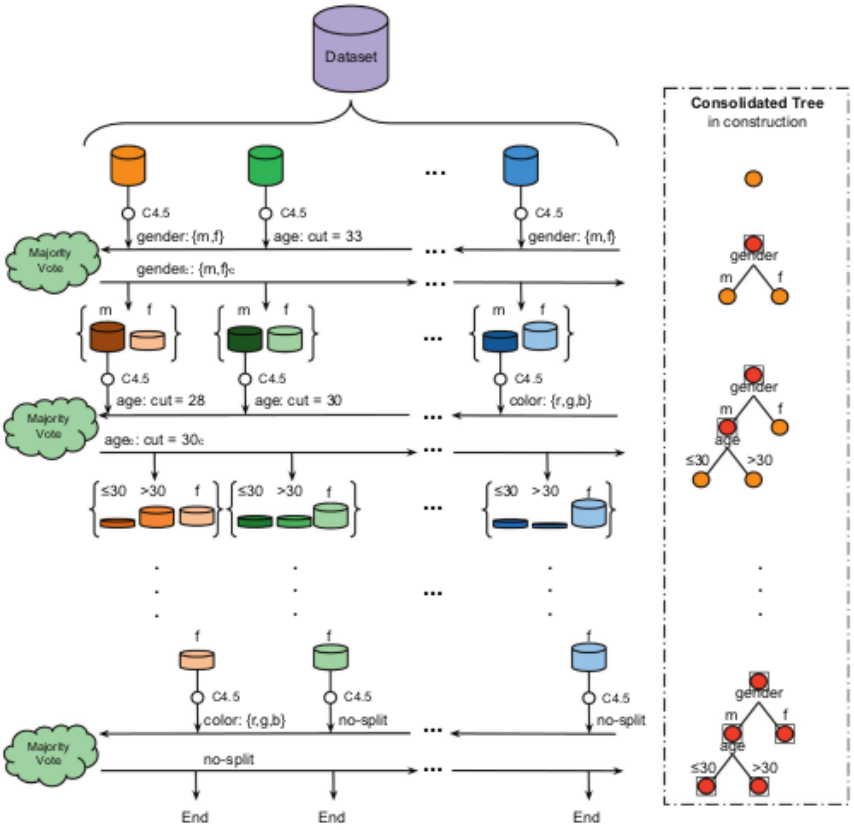
C4.5 ikasketa automatikoko algoritmoa da, Ross Quinlan-ek garatua [4], eta ID3 algoritmoaren hedapen bat da. Entrenamenduko datu-multzoetatik abiatuta erabaki-zuhaitza eraikitzeko erabiltzen da. Algoritmoa hau entropian oinarritzen da atributua hautatzeko eta entropia klaseko etiketen banaketatik abiatuta kalkulatzen da. Algoritmoak informazio normalizatuaren irabazia maximizatzen duen atributua bilatzen du, eta informazio gehien duen atributua hautatzen du. Hautatutako atribututik abiatuta, nodoak eta adarrak sortzen dira, datuak azpimultzo txikiagoetan zatituz, modu errekurtsiboan, zuhaitz osoa eraiki arte.

5.3.2 Bagging

Bagging (Bootstrap Aggregating) [5] mihiztatze-algoritmo bat da (hainbat eredu konbinatzen dituen teknika) eta algoritmo sinpleak batzen ditu konplexuagoak eta eraginkorragoak diren beste batzuk lortzeko. Helburua ereduaren bariantza murriztea eta gaindoikuntza saihestea da. Bagging prozesua entrenamendu-lagin ugari sortzean datza, jatorrizko datu-multzotik abiatuta. Ondoren, eredu bat (C4.5 gure kasuan) entrenatzen da lagin horietako bakoitzarekin modu independentean. Sailkapenerako, banakako iragarpenak konbinatzen dira bozketa-estrategia bat erabiliz. Bozketa-estrategia ohikoena 'gehiengoz bozkatzea' da; bertan, oinarrizko eredu bakoitzak bere botoa ematen du (adibidez, 'autoa' edo 'hegazkina') eta botoen gehiengoa jasotzen duen klasea azken iragarpena da.

5.3.3 Consolidated Tree Classification (CTC)

CTC algoritmoa (*Consolidated Trees Construction*) ALDAPA ikerketa-taldeak diseinatutako algoritmoa da [3], C4.5 algoritmoan oinarrituta. Algoritmoak lagin multzo bat sortzen du eta lagin bakoitzean oinarrituta zuhaitz bat sortzen du C4.5 algoritmoa erabiliz. Zuhaitz hauek ez dira sortzen era independentean, baizik eta zuhaitz guztiak kontuan hartuta bozketa bat eginda, nodoz nodo. Nodo bakoitza zatitzeko aldagairik bozkatuena aukeratzen da; aldagai horri aldagai kontsolidatua deritzo. Lagin guztiak aldagai kontsolidatu berarekin zatitzen dira, eta horrela zuhaitz kontsolidatua lortzen da 2. irudian ikusten den bezala.

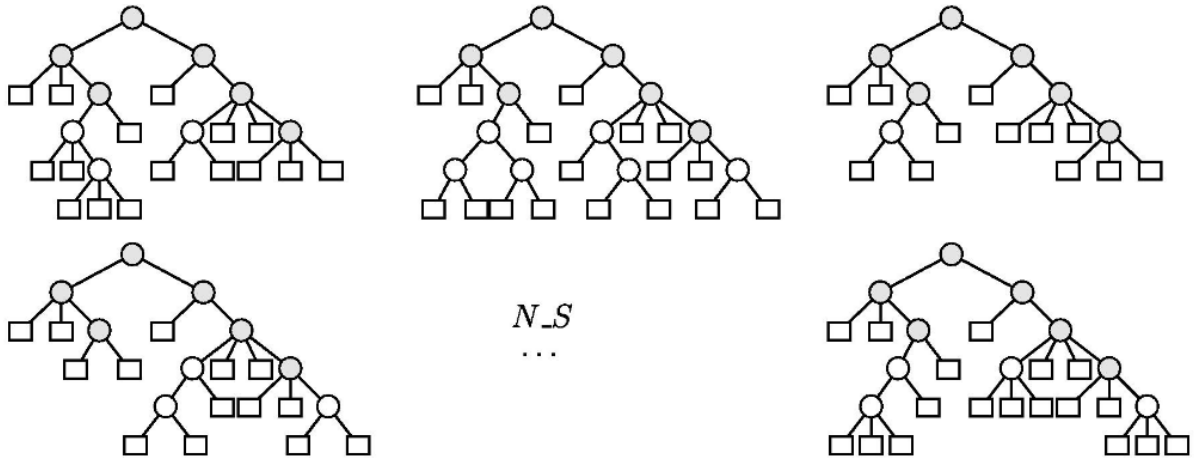


Irudia 2: CTC zuhaitza [3]

Zuhaitz guztiak, egituraren aldetik, berdinak dira eta beraz, sailkapenaren azalpena ematen du, C4.5 algoritmoa bezala, eta ez Bagging-en bezala, non denak desberdinak dira eta, beraz, ez du gaitasun hori.

5.3.4 PCTBagging (Partially Consolidated Tree Bagging)

PCTBagging algoritmoa ALDAPAko ikerketa-taldeak diseinatu zuen [2]. CTC-aren eta Bagging-aren arteko hibridoa da, CTC-ren azalpena eustea du helburu, eta, aldi berean, Bagging-aren diskriminazio-gaitasuna handitzea aprobetxatzea. Erabiltzaileak zuhaitzaren zenbat ehuneko kontsolidatu nahi duen aukeratzen du, eta zuhaitz guztiek egitura komuna izango dute. Egitura horrek sailkapenaren azalpena emango du. CTC algoritmoa aplikatzen da zuhaitz kontsolidatua lortzeko, eta, ondoren, inausi egiten da, erabiltzaileak aukeratutako nodoen ehunekoa utzita. Gero, azpizuhaitz bakoitza bere aldetik garatzen da, Bagging bat eratuz, beti egitura kontsolidatu berarekin 3. irudian ikusten den bezala (atzealdea grisa duten barne-adabegiak). Irudian 'N_S' *Number of Samples* da, hau da, lagin kopurua.



Irudia 3: PCTBagging sailkatzailea [2]

5.4 BALIOZKOTZE-TEKNIKAK

Baliozkotze-teknikak ikasketa-eredu automatiko baten zehaztasuna eta errendimendua ebaluatzeko erabiltzen diren metodoak dira. Oso garrantzitsuak dira ereduaren orokortasuna ebaluatzeko eta aurrez ikusi gabeko datuetan iragarpen zehatzak egiteko gai den aztertzeko.

Gehien erabiltzen den baliozkotze-tekniketako bat baliozkotze gurutzatua da (**cross-validation**). Datu-multzoa *fold* izeneko k zati berdinetan banatzen da. Ondoren, eredua k aldiz entrenatzen da, iterazio bakoitzean *fold* bakoitza proba-multzo gisa erabiliz, eta gainerakoak entrenamendu-multzo gisa erabilita. Segidan, iterazioen emaitzak konbinatu egiten dira, eta emaitza ebaluatzeko batez bestekoa erabiltzen da.

Hasierako datu multzoa bi multzo nagusitan banatzen da: entrenamendu multzoa eta proba multzoa. Entrenamendu multzoa (***training set***) eredua sortzeko erabiltzen den lagina da, datuetan agertzen diren patroiak ikasten dituena; eta proba multzoa, eredua ebaluatzeko erabiltzen da.

5.5 EBALUAZIO IRIZPIDEAK

Ebaluazio-irizpideak ereduen errendimendua eta zehaztasuna ebaluatzeko erabiltzen diren neurriak dira. Irizpide horiekin ereduak zein onak diren ikus daiteke.

Jarraian, 9 atalean proiektu honen PCTBagging gidatua algoritmoa aztertzeko erabili diren irizpideak azalduko dira. Irizpideak hiru taldetan banatzen dira: sailkatzeko gaitasuna, azalpen-gaitasuna eta kostu konputazionala.

5.5.1 SAILKATZEKO GAITASUNA DUTEN IRIZPIDEAK

Sailkatzeko gaitasuna duten irizpideek ereduak aurreikuspen zehatzak egiteko duen gaitasunari buruzko informazioa ematen dute. Irizpide horiek *Nahasmen Matrizea* izeneko taula erabiltzen dute. 1. taulan ikus daitekeen taula honek klase bakoitzerako iragarpen zuzen eta okerren kopurua erakusten du, eta lau koadrantetan banatzen da:

- **Benetako positiboak (TP):** ereduak behar bezala sailkatu dituen kasu positiboen kopurua da (*True Positives*, ingelesez).
- **Positibo faltsuak (FP):** ereduak oker sailkatu dituen kasu negatiboen kopurua da (*False Positives*, ingelesez).
- **Benetako negatiboak (TN):** ereduak zuzen sailkatu dituen kasu negatiboen kopurua da (*True Negatives*, ingelesez).
- **Faltsu negatiboak (FN):** ereduak oker sailkatu dituen kasu negatiboen kopurua da (*False Negatives*, ingelesez).

	Iragarpen Positiboa	**Iragarpen Negatiboa**
Klase Positiboa	TP	TN
Klase Negatiboa	FP	FN

Taula 1: Nahasmen Matrizea

Balio horietatik abiatuta, irizpide desberdinak kalkula daitezke, aurrerago azalduko dugun bezala.

Accuracy

Asmatze-tasak edo *Accuracy*-k iragarpen zuzenen proportzioa kalkulatzen du, iragarpen guztien arabera. Kalkulua egiteko, iragarpen zuzenak zati iragarpen guztien kopurua egin behar da, honako formula hau erabiliz:

$$\textbf{Accuracy} = (\text{TP} + \text{TN}) \;/\; (\text{TP} + \text{TN} + \text{FP} + \text{FN})$$

Oso neurri sinplea da, baina baliteke ez oso erabilgarria izatea datu multzo desorekatuetan, non klase batek beste batzuek baino errepresentazio askoz handiagoa duen. Sailkatzaile batek kasu guztiak klaserik ohikoenekoak balira bezala etiketatzen baditu, emaitza aipagarriak lortuko ditu datu-lagin desorekatu batean; hala ere, ez da behar bezala sailkatzen ari.

Balanced Accuracy

Asmatze-tasa orekatua edo *Balanced Accuracy* datu multzo batean klaseen desoreka kontuan hartzen duen neurria da. Asmatze-tasa kalifikatzen du positiboen eta negatiboen asmatutako batez bestekoa modu independentean kalkulatzen.

Lehenik, positiboen batez besteko asmatze-tasa kalkulatzen da, horri sentsibilitatea deitzen zaio, eta benetako positiboen kopurua benetako positiboen eta negatibo faltsuen baturarekin zatitzen da. Hau da formula:

$$\textbf{Sentsibilitatea} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$$

Bestalde, negatiboen batez besteko asmatze-tasa kalkulatzen da, hau da, espezifikotasuna, eta benetako negatiboen kopurua zatitzen da benetako negatibo eta positibo faltsuen baturarekin. Hau da formula:

$$\textbf{Espezifikotasuna} = \mathrm{TN}/(\mathrm{TN} + \mathrm{FP})$$

Asmatze-tasa orekatuak sentsibilitatearen eta espezifikotasunaren batez besteko aritmetikoa kalkulatzen du:

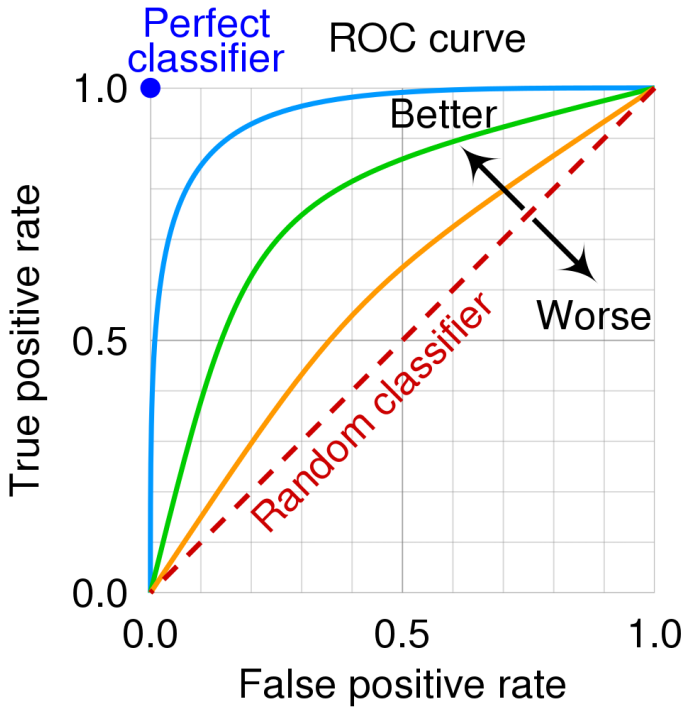
$$\textbf{Balanced Accuracy} = (\text{Sentsibilitatea} + \text{Espezifikotasuna})/2$$

Neurri hau egokiagoa da datu-base desorekatuetarako, kasu positiboen eta negatiboen artean alde handia dagoen kasuetan arazoak saihesten baititu.

AUC (Area Under the Curve)

AUCk (*Area Under the Curve* edo Kurbaren Azpiko Eremua) ROC (*Receiver Operating Characteristics*, Hargailuaren funtzionamendu-ezaugarriak euskaraz) kurbaren azpiko area irudikatzen du. Kurba horrek benetako positiboen tasaren eta positibo faltsuen tasaren arteko erlazioa erakusten du. 1 inguruko AUC balioa ereduak errendimendu ona duela adierazten du.

4. irudian ROC kurba ikusten da. Kurba gorria oso kurba txarra izango litzateke, eta berdea, berriz, ona.



Irudia 4: ROC kurba [6]

Kappa

Kappa koefizienteak (k) ereduaren iragarpenen eta benetako klaseen arteko komunztadura ebaluatzen du. Baliagarria da datuen sailkapeneko ebaluatzaileen arteko adostasun-maila neurtzeko. Kapparen balio altu batek akordio fidagarri bat adierazten du, eta negatibo batek edo 0tik hurbil dagoenak, berriz, adostasun falta edo zoriaren antzeko akordio bat.

Kappa koefizienterako erabili ohi den interpretazio bat hurrengoa da:
Kappa $<$ 0.20: Oso pobrea
0.20 $<=$ Kappa $<$ 0.40: Pobrea
0.40 $<=$ Kappa $<$ 0.60: Ertaina
0.60 $<=$ Kappa $<$ 0.80: Ona
Kappa $>=$ 0.80: Oso ona

Eta formula hau erabiliz kalkulatzen da:

$$\textbf{Kappa} = (P_b - P_e)/1 - P_e$$

Non:

P_b behatutako proportzioa da. Kalkulua egiteko, behaketa-akordioen kopurua zati

ebaluatutako kasu guztiak egin behar da. Hau da:

$$P_b = \frac{TP + TN}{TP + FP + FN + TN}$$

eta

P_e espero den proportzioa da. Ebaluatzaileen artean kasualitatez espero diren akordioen proportzioa adierazten du. Kalkulua egiteko, ebaluatzaile bakoitzerako adostasun-proportzioak biderkatu behar dira, bakoitza bere aldetik, eta, ondoren, proportzio horiek batu behar dira.

$$P_e = \frac{(TP + FP) \cdot (TP + FN) + (FN + TN) \cdot (FP + TN)}{(TP + FP + TN + FN)^2}$$

Kappa egokia da datu-multzo orekatuetarako eta desorekatuetarako.

5.5.2 AZALTZEKO GAITASUNA DUTEN IRIZPIDEAK

Azalpen-gaitasuna duten irizpideak ereduak datu-multzo baten klaseak azaltzeko eta iragartzeko duen gaitasuna ebaluatzeko erabiltzen diren metrikak dira. Erabiltzaileari azalpenak emateko eta kasu bat era batera edo bestera sailkatzera eraman duten arrazoiak ulertzeko erabiltzen dira. Proiektu honetan bi erabiliko ditugu: nodo kopurua eta hosto kopurua:

Nodo kopurua

Nodo kopurua zuhaitz batean egiten diren barne-zatiketen kopuru totala da (nodo finalak ez direnak). Nodo bakoitzak galdera bat adierazten du, datuak azpimultzo txikiagoetan banatzeko erabiltzen dena. Datu multzoaren karakteristiketan oinarritutako erabakiak hartzeko arduradunak dira; beraz, nodo kopurua zenbat eta handiagoa izan, orduan eta zuhaitz konplexuagoa izango da.

Hosto kopurua

Hostoak umerik ez duten nodoak dira eta ereduak aukeratutako azken kategoriak adierazten dituzte. Instantzia bati dagokion klaseari buruzko azken erabakiaren arduradunak dira. Zenbat eta txikiagoa izan hosto kopurua, orduan eta argigarriagoa izango da, eta errazago ulertuko da zuhaitza.

5.5.3 KOSTU KONPUTAZIONALAREN IRIZPIDEAK

Kostu konputazionalaren irizpideak eredua entrenatzeko eta erabiltzeko baliabide konputazionalen kopurua ebaluatzeko erabiltzen diren metrikak dira. Garrantzitsuak dira ereduaren eraginkortasuna ebaluatzeko, batez ere datu-multzo handiekin. Proiektu honetan entrenamendu-denbora erabili da neurri gisa:

Entrenamendu-denbora

Entrenamendu-denbora eredua entrenamendu-datuen multzo batean entrenatzeko behar den denbora da. Garrantzitsua da ereduaren azkartasuna ebaluatzeko.

6 PCTBAGGING GIDATUA

Jarraian, PCTBagging Gidatua (edo PCTgidatua) algoritmoa azalduko da. Algoritmo hau ALDAPA ikerketa-taldeak inplementatutako PCTBagging-ean oinarritzen da, zeina berriki onartu baita WEKAren pakete ofizial gisa, *j48PartiallyConsolidated* izenarekin ([7]). Algoritmo hau CTC eraikuntzako algoritmoaren eta sailkatzaile anizkoitzaren (*Bagging*) arteko hibridoa da. Algoritmo honen helburua algoritmo batek sailkatzeko gaitasuna izan dezan irtenbide bat bilatzea da, baina baita sailkapenari buruzko azalpenak ematea ere.

Gaur egun WEKAn dagoen inplementazioak *Kontsolidazio-ehunekoa* izeneko parametro bat du, zuhaitz osoaren nodo-kopuru osoari dagokionez mantendu beharreko barne-nodo kontsolidatuen kopurua adierazten duena. Horrela, lehenik zuhaitz kontsolidatu osoa (CTC) eraikitzen da, eta gero inausi egiten da, tamaina handieneko nodoak (populazioa) bakarrik utziz, erro-nodotik parametroak emandako balioraino. Puntu honetatik aurrera, zuhaitz guztiak garatzen dira, *Bagging*-ean bezala, modu independentean. Beraz, zuhaitz guztiek oinarri bera izango dute, eta oinarri horretatik aurrera, zuhaitzak modu desberdinean garatuko dira.

Hala ere, ikuspegi praktiko batetik interesgarria izango litzateke nahi den nodo kopuruaren zuhaitz partziala zuzenean eraikitzea, zuhaitz osoa eraiki behar izan gabe. Hori da PCTgidatua algoritmoak bilatzen duena. Zuhaitza kontsolidatzeko prozesua lehenago gelditzea bilatzen da, irizpide baten arabera, eta, ondoren, algoritmoa eraikitzeko prozesuarekin jarraitzea. Horrela, ez litzateke beharrezkoa izango zuhaitz osoa eraikitzea, ondoren kimatzeko, PCTBag algoritmoak egiten duen bezala, baizik eta beharrezkoa den zuhaitz partziala bakarrik eraikitzen da. Ondoren, Bagging-a aplikatu ahal izango da, PCTBag-ak egiten duen bezala.

Bestalde, zuhaitz partzial bat eraikitzen ari denez, ez litzateke beharrezkoa izango populazioaren tamainaren arabera garatzea; aitzitik, zuhaitz kontsolidatu partzialera iristeko hainbat irizpide azter litezke. PCTBag algoritmoan populazio handiena duten nodoak uzten dira bakarrik, erabiltzaileak erabakitako nodo kopurua utziz eta besteak kimatu egiten dira.

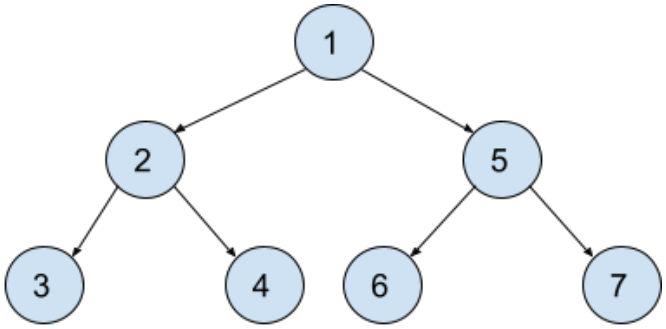
PCTBagging gidatua algoritmoa modu iteratiboan eraikitzeak garatu beharreko hurrengo nodoa aukeratzeko beste irizpide bat hautatzeko aukera ematen du. Proiektu honetan honako irizpide hauek garatu dira, egokienak ikusi direnak, nahiz eta askoz gehiago egon daitezkeen. Irizpideak bi taldetan banatzen dira:

- Noiz gelditu zuhaitz kontsolidatuaren garapena:
  - **Nodo kopuru jakin bat garatu ondoren**. PCTBagging algoritmoak *Kontsolidazio-ehunekoa* parametroarekin duen funtzio bera, baina, portzen-

tajearen ordez nodo-kopuru zehatz bat aukeratuko da.

- **Nodo guztiak maila edo sakonera bateraino garatu ondoren**. Horrek esan nahi du nodoak maila jakin bateraino eraikitzen direla. Adibidez, 2. mailaraino eraiki nahi bada, erro nodoaren ume guztiak eta horien umeak eraikiko dira. Nodoekin bezala, portzentajearen ordez maila-kopuru zehatz bat emango da.

- Zein da garatu beharreko hurrengo nodoa:
  - **Hurrengo nodoa *preorden*-ean aukeratzen da**, CTC algoritmoan edo C4.5-ean bezala. Hau da, lehenengo erroa garatzen da, gero ezkerreko azpi-zuhaitza eta gero eskuineko azpi-zuhaitza nodo bakoitzean, 5. irudian ikusten den bezala.



Irudia 5: Zuhaitz bitar baten preorder garapena

  - **Nodo ume pisutsuena**, PCTBagging-aren egungo bertsioan egiten den bezala.
  - **Nodo ume esanguratsuena** irizpide diskriminatzailean oinarrituta: *Gain ratio*-a
  - Nodo ume esanguratsuenaren eta tamainaren arteko konbinazio bat, ***Gain ratio normalizatua*** *deitu duguna.*

Algoritmo honen inplementazioa aurrerago azalduko da, xehetasunez, 8. atalean. Gainera, 9. atalean algoritmo honekin egindako esperimentazioa dago lortutako emaitzekin. Baina inplementaziora joan aurretik, WEKAn egina dagoenez, WEKA zer den eta nola banatzen den azalduko dugu hurrengo atalean.

7 WEKA

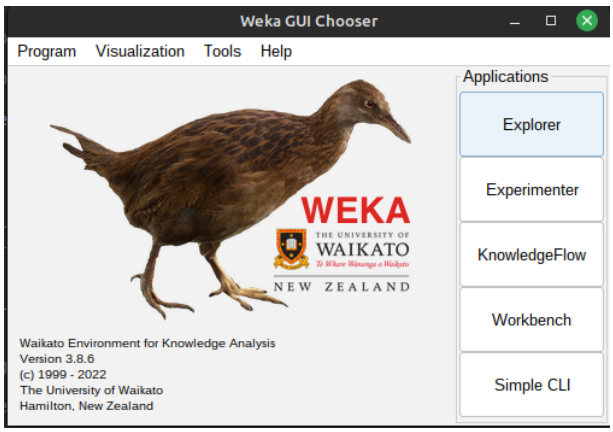
WEKA, *Waikato Environment for Knowledge Analysis*, Javan programatutako software bat da eta asko erabiltzen da ikasketa automatikorako eta datu-meatzaritzarako. Waikato Unibertsitateak garatu zuen eta GNU-GLP lizentziapean banatutako software librea da (erabiltzaileei softwarea erabiltzeko, ikasteko, partekatzeko eta aldatzeko askatasuna bermatzen die).

Softwarea 1993an hasi zen garatzen TCL/TK eta C programazio-lengoaietan, eta 1997an kodea Javan berridaztea erabaki zen. WEKAk algoritmo bilduma zabala du datuen analisiak egiteko eta eredu prediktiboak sortzeko. GNU-GLP lizentzia duenez, kodean algoritmo berriak eransteko aukera ematen du oso erraz, aurrerago azalduko den bezala.

Interfaze grafikoari esker, oso erraza da erabiltzaile hasiberriek eta erabiltzaile aurreratuagoek erabiltzea, baina Ikasketa Automatikoari eta Datu Meatzaritzari buruzko oinarrizko ezagutza eskatzen du. Jarraian, interfaze grafikoak erabiltzaileari ematen dizkion aukerak azalduko dira. Atal honen WEKAri buruz informazio guztia Waikatoko unibertsitateko erabiltzailearen eskuliburutik [1] atera da.

7.1 INTERFAZE GRAFIKOA

Proiektua WEKAren 3.8.6 bertsioan dago eginda, egun dagoen berriena. Aplikazioa exekutatzean, WEKAren interfaze-hautagailua irekitzen da, bost aukera ematen dituena, bakoitzak funtzionalitate desberdin batekin. 6. irudian agertzen den bezala hauek dira bost aukerak: Explorer, Experimenter, KnowledgeFlow, Workbench eta SimpleCLI. Proiektu hau egiteko Explorer-a eta Experimenter-a bakarrik erabili dira, baina guztiak zertarako balio duten azalduko dugu jarraian.



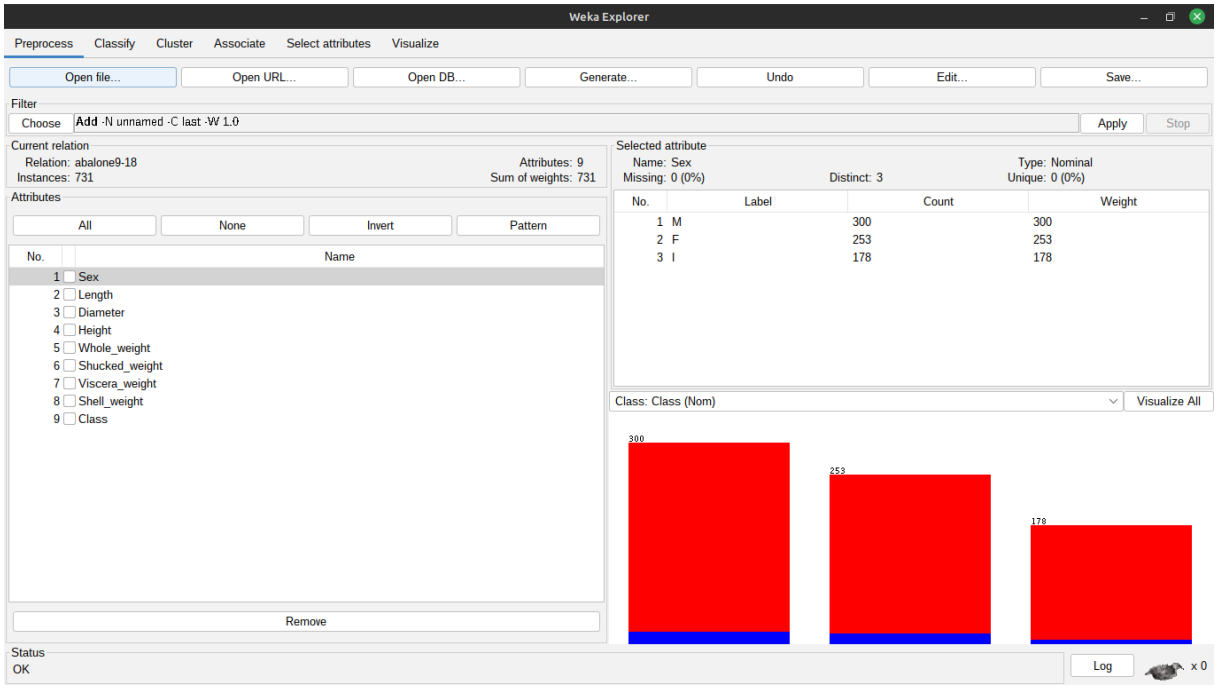
Irudia 6: WEKA GUI Chooser

7.1.1 EXPLORER

WEKAren aukera nagusia da, edo, gutxienez, erabiltzaileek gehien erabiltzen dutena. Aukera ematen du datu-base bat soilik erabiliz algoritmoak probatzeko, eta, beraz, asko laguntzen du algoritmoen funtzionamendu egokia ikusten, datu-base asko erabiliz esperimentazio zehatzagoa egin aurretik. Proiektu honetan gehien erabili den aukera izan da. Atal honen barruan beste 6 aukera daude, baina erabili direnak azalduko ditugu, *Preprocess* eta *Classify*.

7.1.1.1 PREPROCESS

Atal honek hainbat eratako datu-baseak inportatzeko aukera ematen du, hala nola fitxategiak (*.csv, .json, .arff, etab.*), URL batetik edo SQL datu-base batetik. Halaber, *Generate* aukeraren bidez, datu-base bat sor daiteke automatikoki. Bada *Filter* izeneko beste aukera bat, analisia egin aurretik datuei iragazkiak aplikatzeko balio duena, adibidez, atributuak ezaba daitezke, atributuak normalizatu, etab. 7. irudian Preprocess-aren interfazea ikusi daiteke.

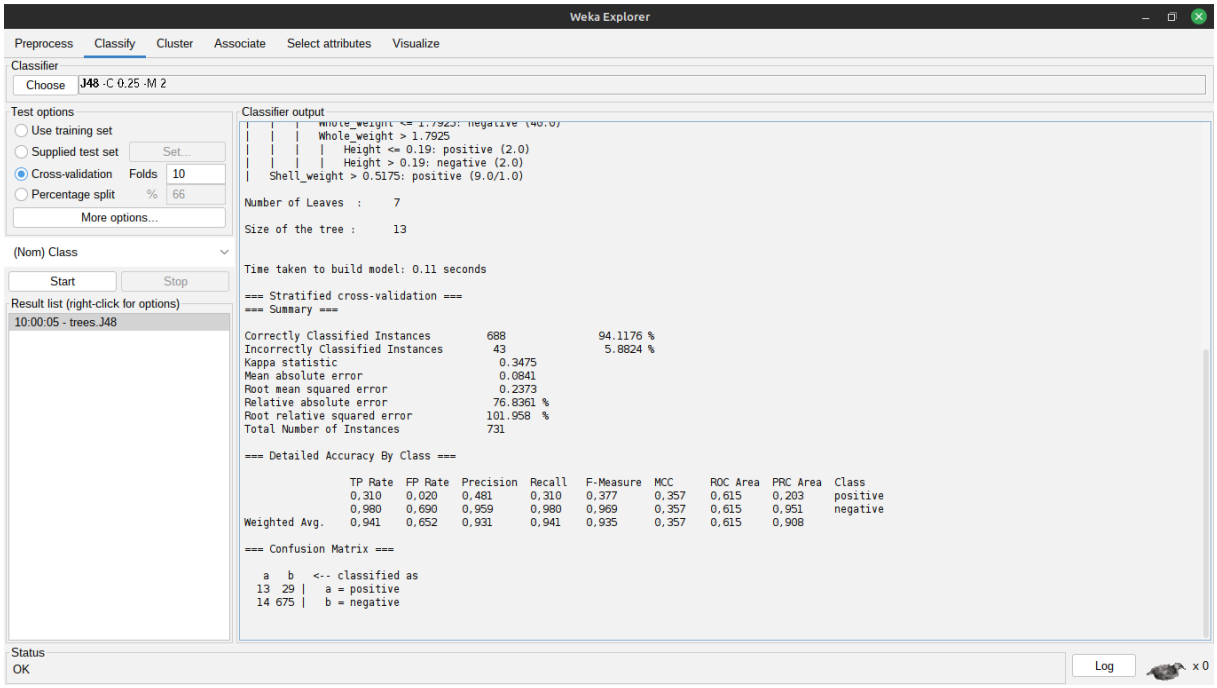


Irudia 7: Explorer atalaren Preprocess

7.1.1.2 CLASSIFY

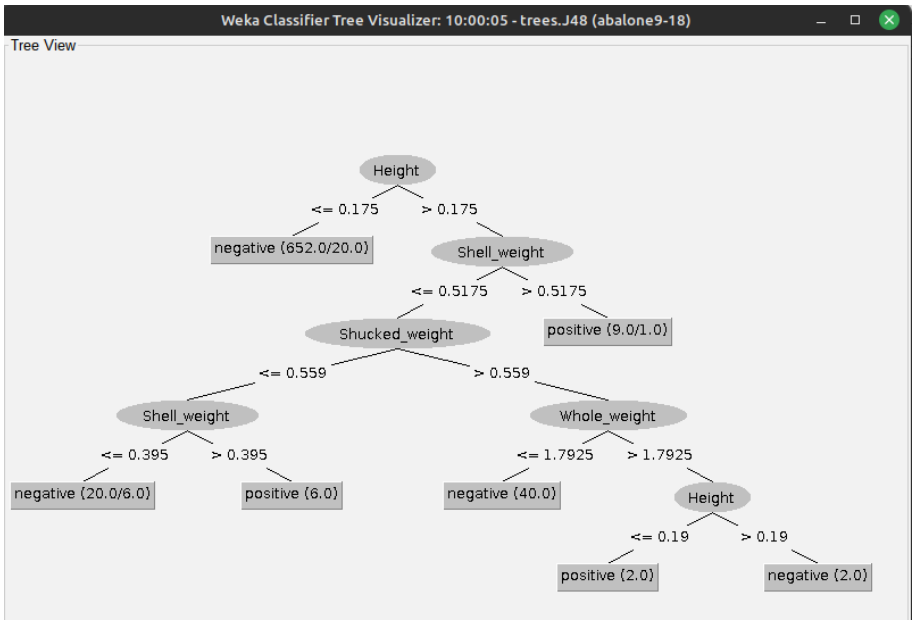
Classify panelean, aurreprozesamenduan inportatutako datuak aztertzeko bertan aukeratutako algoritmoa (*Choose*) aplikatzen da. Sailkatzaile bat sortzeko eta aldi berean ebaluatzeko 'hasi' (*Start*) aukeratu behar da. Ebaluatzeko aukeren artean (*Test Options*) erabilienak *Training set* eta *Cross-validation* dira (5.4. atalean azaldu direnak). Algoritmoa

exekutatu ondoren, sailkapenaren emaitzei buruzko informazioa duen irteera bat agertuko da. 8. irudian ikusi ahal dira dauden aukerak.



Irudia 8: Explorer atalaren Classify

Azkenik, algoritmoak ahalbidetzen badu, eredua modu grafikoan ikus daiteke. Esaterako, erabaki-zuhaitzen kasuan, zuhaitzak duen egitura ikus daiteke. Aukera hau oso erabilia izan da proiektua egiteko; algoritmoa zehazki zer egiten ari den eta zuhaitza behar bezala korritzen ari dela ikusteko balio izan du. 9. irudia emaitzen zerrendan eskuineko klik egitean agertzen den *Visualize tree* aukeraren adibide bat da. Horrelako irudiak behin eta berriz agertuko dira dokumentu honetan, PCTBag gidatuaren funtzionamendua azaltzeko.



Irudia 9: J48 algoritmoarekin sortutako erabaki-zuhaitzaren adibide bat

7.1.1.3 BESTE AUKERAK

Beste aukerak ez dira proiektuan erabili, baina honetarako balio dute labur-labur:

- **Cluster:** gainbegiratu gabeko taldekatze-atazak egiteko aukera ematen du, datuak automatikoki banatzen dira antzeko ezaugarriak dituzten taldeetan edo *klusterretan*. Patroiak identifikatzen dira datuetan, etiketarik (klaserik) behar izan gabe.
- **Associate:** asoziazio-algoritmoak exekutatzen ditu, datuetatik informazioa ateratzeko. Atributuen arteko erlazioak identifikatzeko algoritmoak ditu.
- **Selected attributes:** datu multzo batean atributu iragargarrienak identifikatzeko algoritmoak ematen ditu.
- **Visualize:** datuak eta emaitzak grafikoki ikusteko eta arakatzeko aukera ematen du.

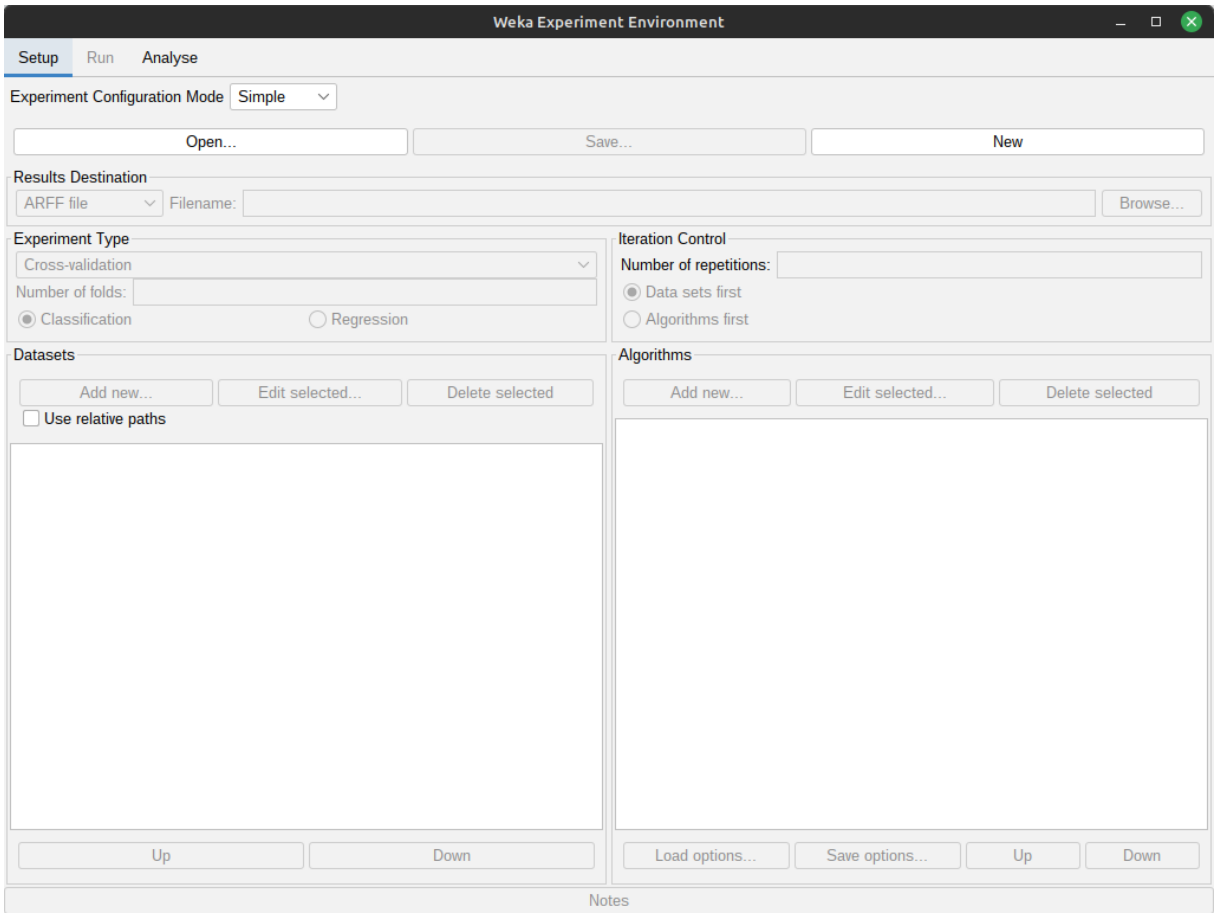
7.1.2 EXPERIMENTER

Atal hau oso erabilgarria da algoritmo baten esperimentazio-faserako, algoritmo asko datu-base askorekin exekutatzeko aukera ematen baitu aldi berean, ondoren emaitzak alderatzeko.

7.1.2.1 SETUP

Hasteko, *Setup* atala prestatu behar da. Hemen esperimentaziorako konfigurazio guztiak egin behar dira, ondoren azalduko dira garrantzitsuenak. 10. irudian ikusten den bezala hainbat aukera daude. *Datasets*-en eta *Algorithms*-en exekuzioan erabili nahi

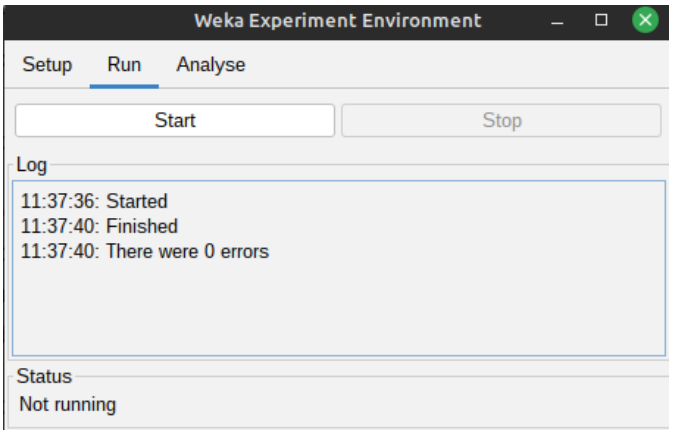
diren datu-baseak eta algoritmo guztiak aukeratu behar dira, hurrenez hurren. Baita egin nahi den esperimentu mota aukera daiteke ere (*Cross-validation* edo *Train/Test Percentage Split*). Zenbat kontrol-iterazio egin nahi diren aukera daiteke ere, iterazio hauek esperimentu baten errepikapen kopurua kontrolatzeko balio dute. Hau baliagarria da emaitza sendoagoak lortzeko eta emaitzen aldakortasuna ebaluatzeko. Azkenik, *Result Destination*-en, emaitzen fitxategia zein motatakoa izatea nahi den aukera daiteke (.arff edo .csv) eta, bukatzeko, non gorde nahi den fitxategia.



Irudia 10: WEKA Experiment Envorment

7.1.2.2 RUN

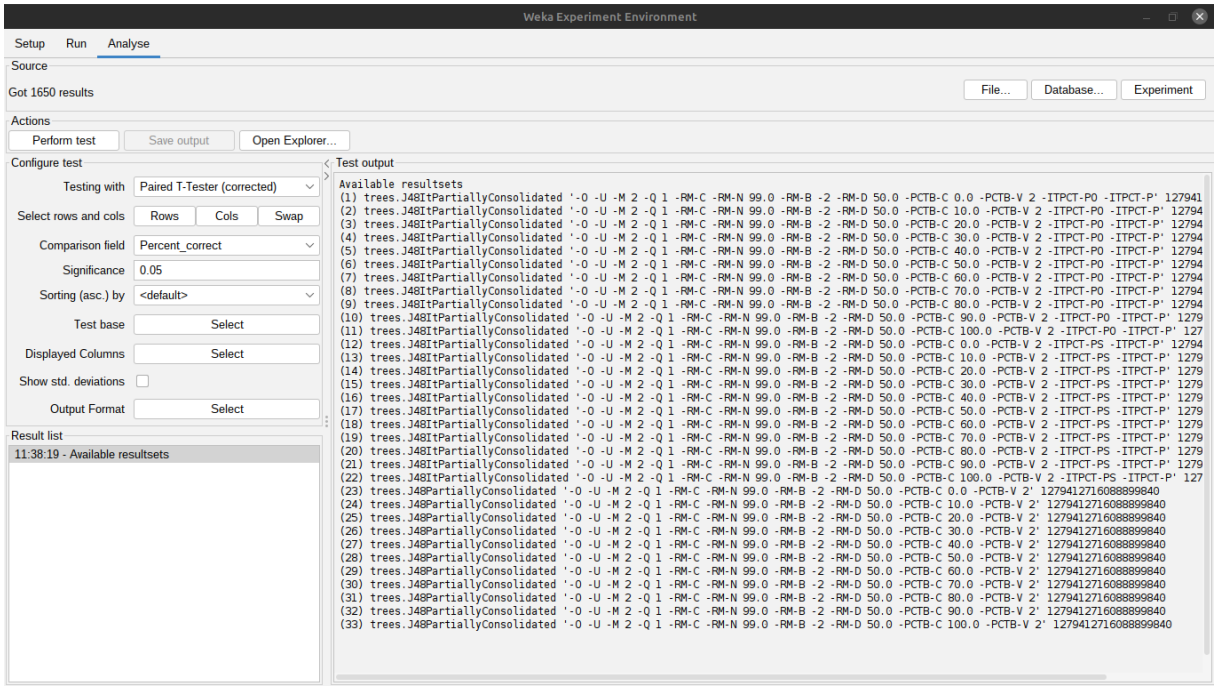
11. irudian ikusten denez oso atal sinplea da, aurreko atalean konfiguratu dena exekutatzeko balio duena. Bi botoi baino ez ditu, hasi eta gelditu. Irteeran, exekuzioa zein ordutan hasi den agertzen da, eta amaitzean amaiera-ordua agertzen da. Horrez gain, akatsen bat egon den edo dena ondo joan den adierazten da. Exekuzio oso erraza eta azkarra bada, ez du luze joko.



Irudia 11: WEKA Experiment-aren RUN atala

7.1.2.3 ANALYSE

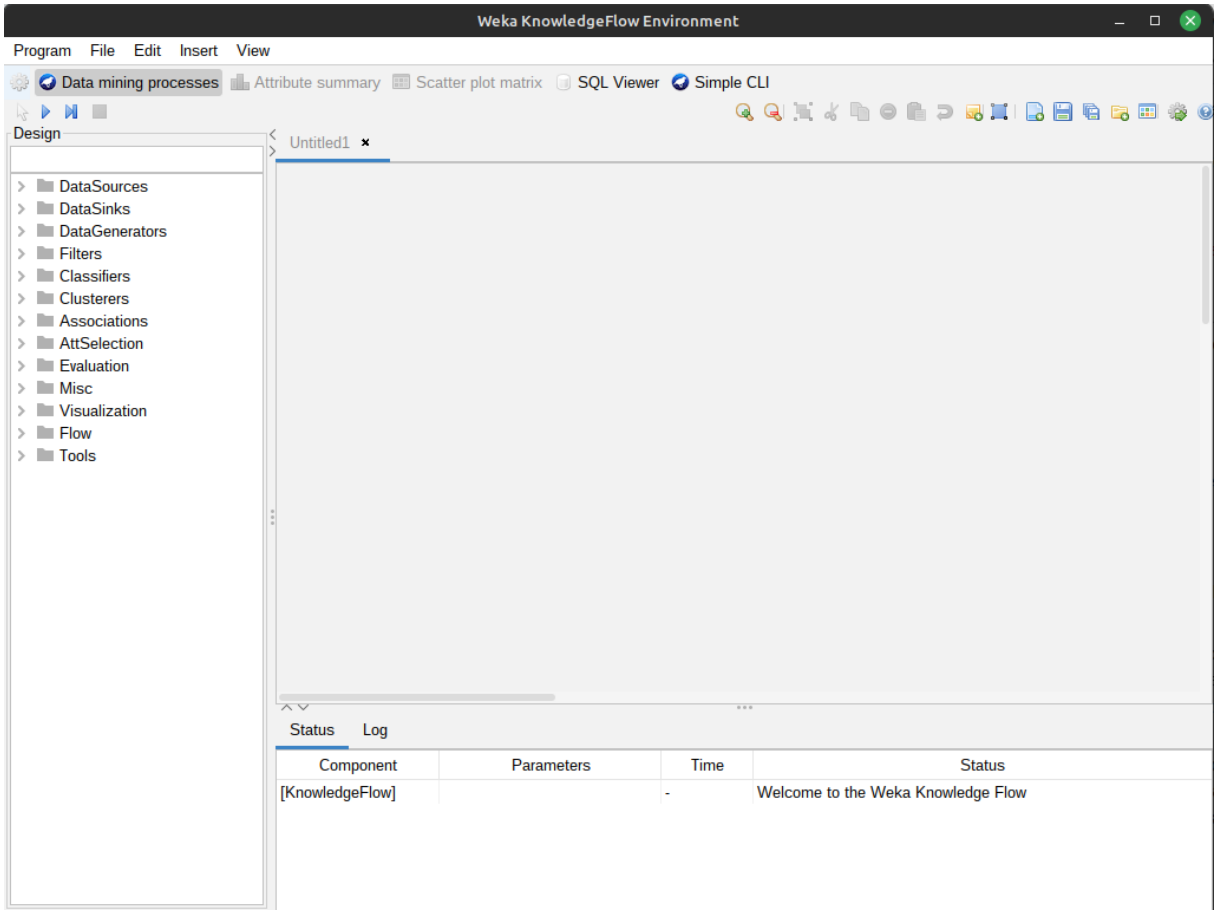
Bukatzeko, izena adierazten duen moduan, *Analyse* atala exekuzioa analizatzeko balio du. Zer konparatu nahi den eta zer formatuan aukeratu behar da. Gero, *Perform test* botoiari emanez konparaketa egiten da. Irteeran konparaketaren informazio guztia agertzen da, eta hau kalkulu-taula batera eramanez grafikoak egin ahal dira konparazioak hobeto ikusteko. 12. irudian, adibide gisa, PCTBag eta PCTgidatua algoritmoak erabiliz atera den irteera erakusten da.



Irudia 12: WEKA Experiment-aren Analyse atala

7.1.3 KNOWLEDGE FLOW

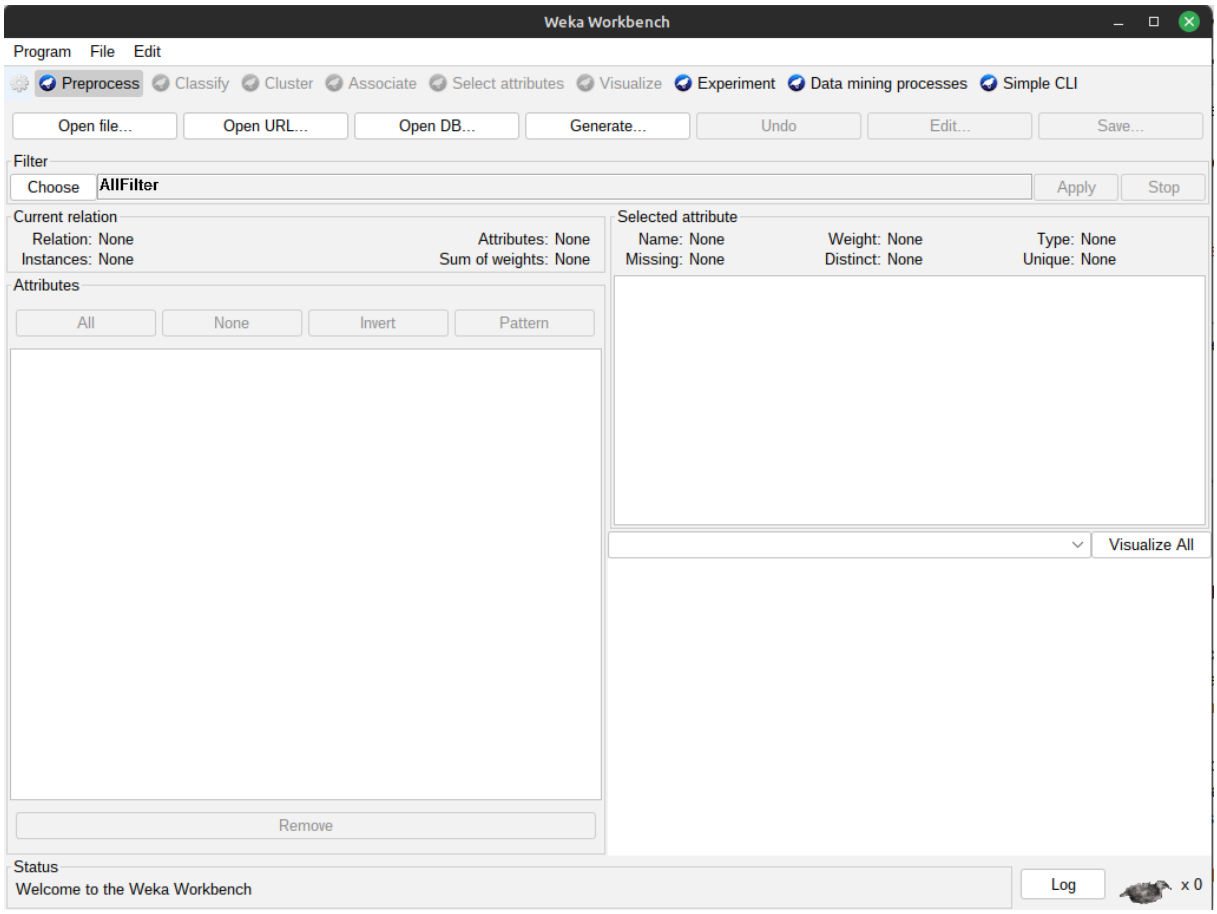
Aurreko atalaren oso antzekoa da, baina, kasu honetan, lan-fluxuak modu bisualean gauzatzeko aukera ematen duen interfaze grafiko interaktiboa da (13. irudia). Nahi diren algoritmoak eta sailkatzaileak aukeratu edo arrastaka eraman daitezke, eta, gero, osagai bakoitzaren parametroak konfiguratu eta doitu daitezke. Elkarrekin konektatzen dira lan-fluxu pertsonalizatuak eraikitzeko eta horrek esplorazioa eta esperimentazioa errazten ditu, hainbat konfigurazio eta analisi-aukerarekin.



Irudia 13: WEKAren KnowledgeFlow atala

7.1.4 WORKBENCH

Workbench-a Explorer-en antzekoa da, baina funtzionalitate zabalagoa eta osoagoa eskaintzen du. Explorer-en ezaugarriak ditu, baina datuen aurreprozesamendurako, eredu sofistikatuagoen eraikuntzarako eta ebaluaziorako, talde-esperimentuetarako eta lan-fluxuen automatizaziorako aukera aurreratuak ere baditu. Egokiagoa da azterketa zehatzagoa behar duten erabiltzaileentzat, baita ereduak modu sakonagoan konfiguratzeko eta ebaluatzeko. 14. irudian agertzen da interfazea.



Irudia 14: WEKAren Workbench atala

7.1.5 Simple CLI

Azkenik, Simple CLI atala (*Simple Command-Line Interface*-ren laburdura, komandolerroko interfaze sinplea, euskaraz). Atal hau bisualki sinpleena da (15. irudia), komandoak idazteko terminal bat baino ez baita. Erabiltzeko zailena da, konfigurazio guztiak eskuz jarri behar direlako, euskarri grafikorik gabe. Nahikoa ezagutza izan behar da erabili ahal izateko. *Help* komandoa erabiliz komando guztiak ikus daitezke:

`capabilities <classname> <args>`: Zehaztutako klasearen gaitasunak erakusten ditu.

`cls`: Garbitu irteera gunea.

`echo msg`: Mezu bat erakusten du.

`exit`: Simple CLI programa itxi.

`help`: Laguntza erakusten du.

`history`: Aurretik erabilitako komando guztiak erakusten ditu.

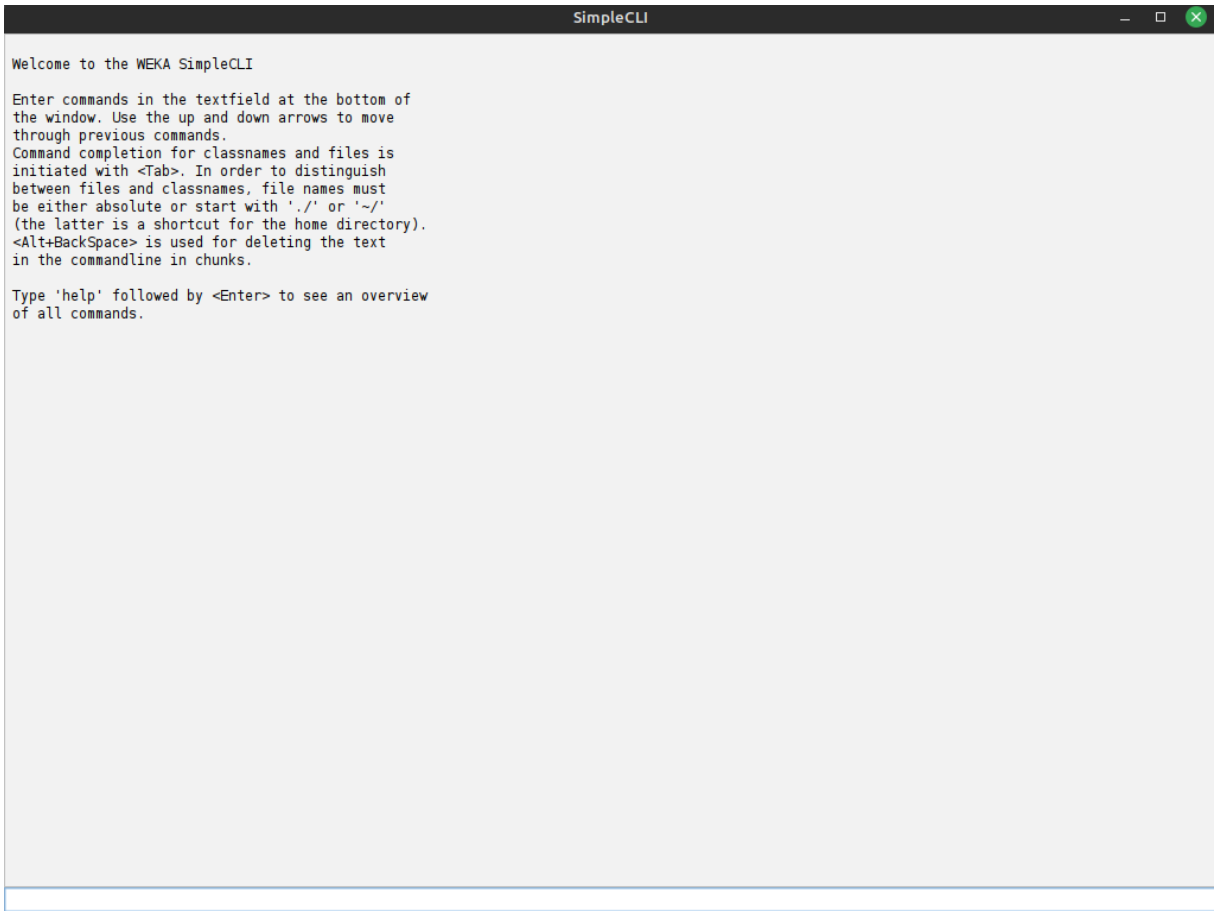
`java <classname> <args>`: Java klase baten gaitasunak erakusten ditu.

`kill`: Egiten ari den lana gelditzen du, baldin badago.

`script <script_file>`: Script fitxategi batetik komandoak exekutatzen ditu.

`set [name = value]`: Ezarri aldagai bat edo erakutsi egungo aldagai guztiak.

`unset name`: Aldagai bat ezabatzen du.



Irudia 15: WEKAren SimpleCli atala

7.2 DATUEN FITXATEGIAK

WEKAko datuak *.json*, *.csv*, *.data* edo fitxategi-mota askotan egon daitezke, baina lehenetsita, eta gehien erabiltzen direnak *.arff* motako fitxategiak dira. ARFF (Atributu-Harremana fitxategiaren formatua, euskaraz) datu-meatzaritzan eta ikaskuntza automatikoan erabili ohi den fitxategi-formatua da, datu-multzo baten instantziei (errenkadak) eta atributuei (zutabeak) buruzko informazio egituratua duena.

Osagaiak azaltzeko adibide sinple bat erabiliko da (2. taula). Adibide honek euskal hiri eta probintziei buruzko informazioa irudikatzen du:

```
@relation EH_HiriakEtaHerrialdeak

@attribute class {hiria, herrialdea}
@attribute izena string
@attribute populazioa numeric
@attribute etxebizitzak numeric

@data
hiria, Bilbo, 345821, 15000
hiria, Donostia, 186409, 12000
hiria, Vitoria-Gasteiz, 252571, 18000
herrialdea, Gipuzkoa, 720000, 98000
herrialdea, Bizkaia, 1162000, 84000
herrialdea, Araba, 333800, 52000
```

Taula 2: *.arff* motako fitxategi sinple baten adibidea

7.2.1 GOIBURUA

Goiburukoak datuen multzoari eta egiturari buruzko informazioa ematen du. Aurrean **@relation** adierazpena du, eta, ondoren, datu-multzoaren izena. Izen deskribatzailea izan behar du zein daturekin ari garen jakiteko. Kasu honetan, izena EH_HiriakEtaHerrialdeak da, aurkituko ditugun datuak deskribatzen ditu: Euskal Herriko Hiriak eta Herrialdeak. Horrela geratuko litzateke:
@relation $EH\_HiriakEtaHerrialdeak$

7.2.2 ATRIBUTUAK

Datuen ezaugarriak edo atributuak definitzen ditu. Atributu bakoitzak izen bat eta lotutako datu-mota bat ditu. Definizioa honelakoa izan behar da: **@attribute** (*atributuaren izena*) (*atributuaren mota*). Artxibo mota honek onartzen dituen atributu mota ohikoenak hauek dira:

- **numeric:** Balio errealak har ditzakeen zenbakizko atributua.
- **integer:** Balio osoak bakarrik har ditzakeen zenbakizko atributua.
- **real:** numeric-ren sinonimoa, zenbakizko atributu jarraitua.
- **string:** karaktere-kate motako atributua.
- **date:** data motako atributuak.

- **Definitutako balioak:** Aurretik definitutako multzo baten balioak bakarrik har ditzakeen atributu nominala. Adibide honetan, hiria edo herrialdea izan daiteke bakarrik class atributua.

Horrela geratuko litzateke numeric atributu baten definizioa:
@attribute populazioa numeric

Eta horrela definitutako balioa duen atributu baten definizioa:
@attribute class hiria, herrialdea

7.3 DATUAK

Amaitzeko, datuak. @data jartzen da eta, hortik aurrera, datuak ilaretan/lerrotan banatzen dira. Ilara bakoitzak atributu bakoitzari dagozkion balioak ditu, komen bidez bereizita. Hemen dago datuen informazio guztia. Honelako datuak dira:

@data
hiria, Bilbo, 345821, 15000
hiria, Donostia, 186409, 12000
...

7.4 WEKAREN EGITURA

WEKA Javan idatzita dago eta elkarrekin lan egiten duten paketeetan egituratuta dago, funtzionalitate guztiak eskaintzeko. Hauek dira garrantzitsuenak:

- **weka.core:** Pakete honek sistema osoan erabiltzen diren WEKAren oinarrizko klaseak biltzen ditu. Besteak beste, datu-instantziak, atributuak, datu-multzoak, funtzio matematikoak eta komando-lerroko aukerak irudikatzeko klaseak biltzen ditu.

- **weka.classifiers:** Pakete honek WEKAko sailkapen-algoritmoen inplementazioak ditu. Algoritmo famatuentzako klaseak barne hartzen ditu, hala nola Erabaki-Zuhaitzak, Naive Bayes, K-Nearest Neighbors. Pakete hau aldatu behar da algoritmo berri bat gehitu nahi badugu. Hurrengo atalean azalduko da nola.

- **weka.clusterers:** Pakete honek multzokatze-algoritmoen (clustering) inplementazioak ditu. Besteak beste, K-Means eta EM algoritmoetarako klaseak sartzen dira.

- **weka.associations:** Pakete honek WEKAko elkartze-algoritmoen inplementazioak biltzen ditu.

- **weka.filters:** Pakete honek datuen aurreprozesamendurako filtroen inplementazioak jasotzen ditu.
- **weka.attributeSelection:** Pakete honek atributuak hautatzeko algoritmoen inplementazioak biltzen ditu.
- **weka.datagenerators:** Pakete honek datu-multzoak sortzeko klaseak biltzen ditu. Ausazko datuak sortzeko klaseak barne hartzen ditu, banaketa eta ezaugarri desberdinen arabera.
- **weka.experiment:** Pakete honek esperimentuak eta ebaluazioak egiteko klaseak biltzen ditu. Hainbat sailkatzaile, filtro eta datu-multzorekin esperimentuak konfiguratu eta exekutatzeko klaseak barne hartzen ditu.
- **weka.estimators:** Pakete honek estimatzaileak ditu, sarrerako datuen arabera balioak aurreikusteko erabiltzen diren algoritmoei edo ereduei dagozkienak.
- **weka.knowledgeflow:** Pakete honek 7.1.3. atalean azaldu den ezagutza-fluxua exekutatzeko behar den guztia dauka.
- **weka.gui:** Pakete honek WEKAren erabiltzaile-interfaze grafikorako (GUI) klaseak biltzen ditu. Pakete hau exekutatuz GUI Chooser APP-a irekitzen da eta bertatik programa erabil daiteke.

7.5 SAILKATZAILE BERRI BAT TXERTATU

Lehen esan den bezala, WEKAk GNU-GLP lizentzia du, eta horrek esan nahi du kodea denetarako erabil daitekeela. Edozein garatzailek algoritmo berriak inplementatzeko eta probatzeko aukera izan dezan, WEKAk hori oso modu errazean egiteko aukerak eskaintzen ditu. Era berean, WEKAk kontuan hartu beharreko gauzei eta gomendioei buruzko dokumentazio zabala du algoritmo berri bat ezartzean. Jarraian, zehatz-mehatz azalduko dugu zer egin behar den algoritmo berri bat sartzeko eta WEKAko GUItik bertatik erabili ahal izateko. Informazio guztia WEKAren Eskuliburutik atera da [1].

7.5.1 SAILKATZAILE MOTAK

Lehenik eta behin, gehitu nahi dugun sailkatzailea zein motakoa den aukeratu behar da. WEKAn bi sailkatzaile mota nagusi daude: sailkatzaile sinpleak eta sailkatzaile anitzak. WEKAn sailkatzaileen klaseak *weka.classifier* paketean daude. Ondoren, haien arteko ezberdintasunak azalduko dira:

7.5.1.1 SAILKATZAILE SINPLEAK

Sailkatzaile hauek ez dute beste sailkatzailerik erabiltzen oinarri gisa eta entrenamendu datuekin entrenatzen dira zuzenean. Adibide batzuk *Naive Bayes, k-NN edo C4.5* dira. Sailkatzaile mota hau beste bi kategoriatan banatzen da ere:

- **Sailkatzaile Abstraktuak:** Sailkatzaile hauek ez dute datuen gaineko suposizio espezifikorik egiten, besterik gabe sailkapena egiteko instantzia bat klase jakin batekoa izateko probabilitatea erabiltzen dute. Azkarra da eta egokia da datu multzo handietarako. Sailkatzaile mota honen adibide bat *C4.5* da. Sailkatzaile abstraktuen klasea WEKAn *AbstractClassifier* izenarekin agertzen da.

- **Sailkatzailek ausazko portaerarekin:** Sailkatzaile hauek zoria sartzen dute beren entrenamendu edo sailkapen prozesuan. Eredu desberdinak sortzen dituzte exekuzio desberdinetan, doikuntza-arazoak saihesteko eta sendotasuna ugaltzeko. Adibide bat *RandomTree* algoritmoa da, ausazko erabaki-zuhaitz multzo bat eraikitzen duena, nodo bakoitzean atributuak eta atalaseak hautatzeko ausazko estrategiak erabiliz. Sailkatzaile azkar bat da eta baita egokia da datu handien multzoetarako ere. Sailkatzailearen klasea *RandomizableClassifier* izenarekin agertzen da.

Sailkatzaile egokiena aukeratzeko orduan, kontuan izan behar da ausazko portaera duten sailkatzaileak abstraktuak baino eraginkorragoak izan daitezkeela modelo baten orokortzea hobetzeko, baina, era berean, konputazionalki garestiagoak izan daitezkeela ausazkotasunaren ondorioz.

7.5.1.2 META SAILKATZAILEAK

Sailkatzaile hauek oinarrizko sailkatzaileen (sinpleak) errendimendua hobetzeko erabiltzen dira. Mota honetako beste bi sailkatzaile mota daude, sailkatzaile baten konbinazioan oinarritzen direnak, edo bat baino gehiagoren konbinazioan oinarritzen direnak. *weka.classifier.meta* paketearen barruan daude sailkatzaile hauek:

- **Sailkatzaile bakarra:** Oinarrizko sailkatzaile bakarra erabiltzen da sailkapena egiteko. WEKAn eskuragarri dagoen edozein sailkapen-algoritmo erabil daiteke. Beste lau kategoriatan banatzen da:

  - **SingleClassifierEnhancer:** Mota hau oinarrizko algoritmo bakarra hobetzeko edo aldatzeko erabiltzen da. Honen adibide bat *Bagging* algoritmoa da. Lehenago 5.3.2. atalean azaldu den bezala, *bootstrap* birlaginketa metodoa erabiltzen du entrenamendu-datuen multzo ugari sortzeko, eta lagin bakoitzean sailkatzaile bat entrenatzen du. Ondoren, emaitza guztiak erabiltzen ditu konbinatzeko eta azken sailkapena sortzeko.

  - **RandomizableSingleClassifierEnhancer:** Aurrekoaren antzekoa da, baina errandomizagarria ere bada, hau da, ausazkotasuna gehitu daiteke.
  - **IteratedSingleClassifierEnhancer:** Oinarrizko sailkatzaileak modu iteratiboan entrenatzen du, iterazio anitzen bidez. Aurreko iterazioen emaitzak erabiltzen ditu errendimendua hobetzeko.
  - **RandomizableIteratedSingleClassifierEnhancer:** Iteratibitatea eta ausazkotasuna nahasten ditu. Aurrekoa bezalakoa da, baina ausazkotasunarekin. Modu iteratiboan entrenatzen da eta ausazkotasuna kontrola daiteke.

- **Hainbat sailkatzaile:** Hainbat sailkatzaile erabiltzen dira prozesuan, sailkatzaile bakar baten menpe egon beharrean. Kasu honetan, azken sailkapen batera iristeko hainbat sailkatzaileen aurreikuspenak konbinatzen dira. Horretarako, hainbat teknika daude, baina zabalduenetako bat da gehien bozkatzen den botoarena. Sailkatzaile mota honekin zehaztapena hobetzen da, sailkatzaile ezberdinen konbinazioa bakoitzaren indar eta ahuleziak kontrajartzen baitira. Bi kategorietan banatzen dira sailkatzaile hauek:
  - **MultipleClassifiersCombiner:** Azken sailkapena egiteko oinarrizko sailkatzaile anitzen aurreikuspenak konbinatzen ditu.
  - **RandomizableMultipleClassifiersCombiner:** Aurrekoa bezala baina ausazkotasunarekin.

7.5.2 INPLEMENTAZIOA

Inplementazioak *weka.classifier* paketearen azpipaketeetako batean egon behar dira. Zuhaitz bat bada, *weka.classifiers.trees*-en joan behar da, bayes motakoa bada *weka.classifiers.bayes*-en eta horrela guztiekin. Honako azpipakete hauek daude:

- **bayes**
- **evaluation**
- **functions**
- **lazy**
- **meta**
- **mi**
- **misc**
- **rules**

- **trees**

7.5.2.1 METODOAK

Sailkapen-algoritmo berri bat gehitu nahi izanez gero, zer metodo inplementatu behar diren azalduko dugu jarraian. Meta sailkatzaileei beste metodo batzuk gehitu behar zaie, atal honen bukaeran azalduko den bezala. Azalpenean laguntzeko, eredu gisa C4.5 algoritmoa erabiliko da WEKAn lehenetsita datorrena, eta, CTC algoritmoaren, PCTBagging-aren eta PCTBagging gidatuaren oinarri izan dena. Lerro batzuk kendu dira, orain ez baitira azalpenerako garrantzitsuak. WEKAn C4.5 algoritmoaren inplementazioa J48 klasean agertzen da. Ondoren azaltzen dira nahitaezko metodoak:

globalInfo()

Deskribapen labur bat itzultzen du, gero GUIn agertuko dena. Ez dago luzera zehatz espezifikorik, baina egokiena da azalpen nahikoa izatea algoritmoak egiten duena ulertzeko.

```
1 public String globalInfo() {
2     return "Class for generating a pruned or unpruned C4.5 decision tree. For more "
3             + "information, see\n\n" + getTechnicalInformation().toString();
4 }
```

Kodea 1: *globalInfo()* metodoaren inplementazioa

Kasu honetan, J48k weka.core.TechnicalInformationHandler interfazea inplementatzen du, beraz, 3. lerroko *getTechnicalInformation()*-rekin lor daiteke informazioa.

listOptions()

Zerrenda bat sailkatzailearen aukera guztiekin itzultzen du. Komando-lerroaren laguntzarako balio du.

```
1   public Enumeration<Option> listOptions() {
2     Vector<Option> newVector = new Vector<Option>(13);
3     newVector.addElement(new Option("\tUse unpruned tree.", "U", 0, "-U"));
4     newVector.addElement(new Option("\tDo not collapse tree.", "O", 0, "-O"));
5     newVector.addAll(Collections.list(super.listOptions()));
6     return newVector.elements();
7   }
```

Kodea 2: *listOptions()* metodoaren inplementazioa

5. lerroan ikusten denez, superklasearen aukerak ere hartu behar ditu.

setOptions(String[])

Metodo hau komando-lerrotik aurreko metodoaren bidez zerrendatutako parametroak definitzeko da, aurretik gidoi bat doa, eta, jarraian, balioa, adibidez, *-M 2*.

```
1 public void setOptions(String[] options) throws Exception {
2     // Other options
3     String minNumString = Utils.getOption('M', options);
4     if (minNumString.length() != 0) {
5       m_minNumObj = Integer.parseInt(minNumString);
6     } else {
7       m_minNumObj = 2;
8     }
9     m_binarySplits = Utils.getFlag('B', options);
10    super.setOptions(options);
11  }
```

Kodea 3: *setOptions()* metodoaren inplementazioa

listOptions()-en bezala, superklaseari ere deitu egiten dio (ikusi 10. lerroa).

getOptions()

Erabiltzaileak aukeratutako parametroen konfigurazioa lortzeko balio du. *listOptions()-en* egitura bera izan behar du. WEKAren erabiltzaile-interfaze grafikoan ere erabiltzen da sailkatzaile baten konfigurazioa sistemaren memoriara kopiatzeko. Horri esker, konfigurazioa beste sailkatzaile batean itsatsi edo fitxategi batean gorde daiteke, ondoren erabiltzeko.

```
1 public String[] getOptions() {
2     Vector<String> options = new Vector<String>();
3     if (m_binarySplits) {
4         options.add("-B");
5     }
6     options.add("-M");
7     options.add("" + m_minNumObj);
8     Collections.addAll(options, super.getOptions());
9     return options.toArray(new String[0]);
10  }
```

Kodea 4: *getOptions()* metodoaren inplementazioa

getCapabilities()

Metodo hau erabiltzen da sailkatzaile baten gaitasunak lortzeko. Gaitasunak sailkatzaile batek erabil ditzakeen ezaugarrien zerrenda bat dira, hala nola ea zein atributu motarekin lan egin dezakeen (nominalak, zenbakikoak, etab.) edo zein klase motarekin (nominala, zenbakikoak, etab.).

```
1   public Capabilities getCapabilities() {
2     Capabilities result;
3     result = new Capabilities(this);
4     result.disableAll();
5     // attributes
6     result.enable(Capability.NOMINAL_ATTRIBUTES);
7     // class
```

```
8     result.enable(Capability.NOMINAL_CLASS);
9     // instances
10    result.setMinimumNumberInstances(0);
11    return result;
12  }
```

Kodea 5: *getCapabilities()* metodoaren inplementazioa

buildClassifier()

Sailkatzaileentzako funtsezko metodoa da, erabiltzaileak aukeratutako datuetatik abiatuta sailkapen-eredua eraikitzeko erabiltzen dena. Nahiko desberdina da sailkatzaile bakoitzarentzat, eredua nola eraiki behar den araberakoa baita. Metodo honek salbuespen bat abiaraz dezake prozesuan zehar errore bat gertatzen bada. Metodoak entrenamendu-datuen multzoa hartzen du argumentu gisa. *getCapabilities()*-rekin egiaztatu egiten da ea sailkatzaileak emandako datu-multzoa erabil dezakeen.

```
1   public void buildClassifier(Instances instances) throws Exception {
2     if ((m_unpruned) && (!m_subtreeRaising)) {
3       throw new Exception("Subtree raising does not need to be unset for unpruned trees
            !");
4     }
5     ...
6     getCapabilities().testWithFail(instances);
7     ModelSelection modSelection;
8     if (m_binarySplits) {
9       modSelection = new BinC45ModelSelection(m_minNumObj, instances,
10        m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
11    } else {
12      modSelection = new C45ModelSelection(m_minNumObj, instances,
13        m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
14    }
15    ...
16    m_root.buildClassifier(instances);
17    ...
18  }
```

Kodea 6: *buildClassifier()* metodoaren inplementazioa

toString()

Eraikitako sailkapen-ereduaren irudikapena, testu moduan, lortzeko erabiltzen da. Ez da derrigorrezkoa, baina bai oso gomendagarria. Zuhaitzek zuhaitzaren egitura itzultzen dute.

```
1   public String toString() {
2     if (m_root == null) {
3       return "No classifier built";
4     }
5     if (m_unpruned) {
6       return "J48 unpruned tree\n------------------\n" + m_root.toString();
7     } else {
```

```
8       return "J48 pruned tree\n------------------\n" + m_root.toString();
9     }
10  }}
```

Kodea 7: *toString()* metodoaren inplementazioa

distributionForInstance(Instance)

Instantzia jakin baterako klase-probabilitateen banaketa lortzeko erabiltzen da. Emandako instantziarako klase-probabilitateak jasotzen dituen *double* motako array bat itzultzen du. Array-aren elementu bakoitzak klase jakin batekoa izateko probabilitatea adierazten du. Klase nominaleko atributuak erabiltzen dituzten sailkatzaileetan erabiltzen da batez ere.

```
1   public double[] distributionForInstance(Instance instance)
2     throws Exception {
3     return m_root.distributionForInstance(instance,m_useLaplace);
4   }
```

Kodea 8: *distributionForInstance()* metodoaren inplementazioa

classifyInstance(Instance)

Instantzia jakin bat sailkatzeko erabiltzen da, aldez aurretik eraikitako sailkapen-eredua erabilita. Emandako eskaerarako aurreikusitako klasea irudikatzen duen balio bat itzultzen du. Klase-atributu nominalen kasuan, balio hori aurrez adierazitako klase-etiketaren indizea da. Zenbakizko atributuen kasuan, balio hori aurreikusitako erregresio-balioa da.

```
1  public double classifyInstance(Instance instance) throws Exception {
2     return m_root.classifyInstance(instance);}
```

Kodea 9: *classifyInstance()* metodoaren inplementazioa

main(String[])

Sailkatzailea komando-lerrotik exekutatzeko erabiltzen da. Sailkatzaile motaren arabera alda daiteke eta programara pasatzen diren komando-lerroko argumentuak irudikatzen dituen array bat hartzen du argumentu gisa.

```
1   public static void main(String[] argv) {
2     runClassifier(new J48(), argv);
3   }
```

Kodea 10: *main()* metodoaren inplementazioa

META SAILKATZAILEEN BESTE METODOAK

Meta sailkatzaileen kasuan beste metodo batzuk inplementatu behar dira. Hauek dira batzuk:
SingleClassifierEnhancer-en kasuan:

- **defaultClassifierString():** Meta-sailkatzaile honetarako lehenetsitako sailkatzaile gisa erabiltzen den klasearen izena itzultzen du.
- **setClassifier(Classifier):** Sailkatzailea ezartzen du. Ez du kopia bat sortzen.
- **getClassifier():** Ezarrita dagoen sailkapen-objektua itzultzen du. Kontuan izan metodo honek barneko objektua itzultzen duela, eta ez kopia bat.

MultipleClassifiersCombiner-en kasuan:

- **setClassifiers(Classifier[]):** Sailkatzaile gisa erabiltzeko sailkatzaileen array-a ezartzen du.
- **getClassifiers():** Erabiltzen ari den sailkatzaileen array-a itzultzen du.
- **getClassifier(int):** Emandako indizeak adierazten duen array-aren posizioan dagoen sailkatzailea itzultzen du.

7.5.2.2 PARAMETROAK

WEKAko parametroak algoritmo bat erabiltzean espezifikatu daitezkeen konfigurazioak dira. Parametroekin algoritmoa pertsonalizatu daiteke, erabiltzailearen beharretara egokitu dadin. Algoritmo baten parametroak ezartzeko eta lortzeko bi modu daude:

- **Komando-lerrotik:** WEKAn komando-lerrotik ezar daitezke parametroak. Horretarako, *setOptions(String [] options)* eta *getOptions()* metodoak ezarri behar dira algoritmoaren klasean. Parametroak testu-kate gisa pasatzen dira, *-parametroarenIzena parametroarenBalioa* formatuan. Adibidez, *-C 0.5*.
- **GUItik:** WEKAko GUItik bertatik ere ezar eta konfigura daitezke parametroak. Parametro horiek *GenericObjectEditor* objektu-editorea erabiliz konfigura daitezke; horretarako, hiru metodo inplementatu behar dira:
  - ***public void set<ParametroarenIzena>(<Mota>):*** Egiaztatzen du balioa baliozkoa den eta, hala bada, eguneratzen du balioa.

```
1 public void setMinNumObj(int v) {
2     m_minNumObj = v;
3 }
```

Kodea 11: *setMinNumObj()* metodoaren inplementazioa

– ***public <Mota> get<ParametroarenIzena>():*** Hautatutako parametroaren balioa itzultzen du.

```
1 public int getMinNumObj() {
2     return m_minNumObj;
3 }
```

Kodea 12: *getMinNumObj()* metodoaren inplementazioa

– ***public String <ParametroarenIzena>TipText():*** Parametroaren deskribapen bat itzultzen du, GUItik ikusi ahal izango dena.

```
1 public String minNumObjTipText() {
2 return "The minimum number of instances per leaf.";
3 }
```

Kodea 13: *minNumObjTipText()* metodoaren inplementazioa

8 PCTBAGGING GIDATUAREN INPLEMENTAZIOA

Kapitulu honetan azalduko da nola inplementatu den *PCTBagging gidatua* algoritmoa urratsez urrats WEKAn. Horretarako, gehitu diren argumentu eta parametro berriak azalduko dira, eta metodo batzuen sorkuntza edota aldaketa batzuk ere. Azken emaitzara iristeko egindako urrats guztiak azalduko dira.

Aldaera berria dagoeneko inplementatuta dagoen PCTBagging-ean oinarritzen da, eta, beraz, superklase gisa *J48PartiallyConsolidated* klasea aukeratu da. Baina PCTgidatuaren klasean lanean hasi aurretik, *J48* (C4.5) klasea ondo aztertu da. Klase horretan dago oinarrituta *J48Consolidated* klasea (CTC algoritmoa), aldi berean *J48PartiallyConsolidated*-ren superklasea dena. Aurrerago azalduko denez, PCTgidatuaren inplementazioa iteratiboa da, horregatik, klase berriari *J48ItPartiallyConsolidated* deitu zaio.

8.1 J48 ITERATIBOA

Lehenik eta behin, C4.5 algoritmoan lan egin da, zuhaitzaren sorreraren funtzionamendua berdina baita C4.5-ean eta PCTBagging-ean, eta, beraz, C4.5-en egiten diren aldaketak aldaera berrira pasatu ahal izango dira (aldaketa batzuk eginez, noski). Errazagoa denez C4.5-ekin lan egitea PCTBagging-arekin baino, lehenbizi J48 klasean lan egitea erabaki da.

Jakin behar da WEKAn dagoen J48-ren inplementazioa errekurtsiboa dela. Hau arazo bat da, ezarri nahi diren funtzionalitate berrietarako ez baita posible algoritmoa errekurtsiboa izatea. Zuhaitza modu errekurtsiboan eraikiz, beti preordenean eraikitzen da; hau da, lehenik erro-nodoa prozesatzen da eta gero bere umeak ezkerretik eskuinera. Horrela, zuhaitz osoa eraiki arte. Algoritmo berriaren kasuan horrek ez luke balioko. PCTgidatuan, zuhaitza osoa eraiki behar izatea aurreztu nahi dugu eta gainera, garatu beharreko hurrengo nodoa ez da beti preordenak adierazten duena, baizik eta desberdina da erabiltzaileak aukeratutako irizpidearen arabera. Beraz, egin beharreko lehen urratsa J48 klasea iteratibo bihurtzea da. WEKAn C4.5 algoritmoaren inplementaziorako klase asko daude, baina iteratibitatea sartzeko interesatzen den klasea *C45PruneableClassifierTree* klasea da. Klase honek C4.5 algoritmoa erabiliz zuhaitza eraikitzen eta irudikatzen du, era berean *Pruneable* denez zuhaitzaren inausketa egitea ahalbidetzen du haren orokortzea hobetzeko. Inausketa edo *prunning*-a 5.2 atalean azaldu da. *C45PruneableClassifierTree* erabaki-zuhaitz baten klasea denez, *ClassifierTree* klasea hedatzen du.

Klase honek erabiltzaileak bere beharren arabera aukeratzen dituen zenbait atributu ditu. Atributu hauek aldatu egiten dira kodea exekutatzean, sailkatzailea sortzeko. J48iteratiboa eta J48 atributu berdinak dituzte. Hurrengo hauek dira:

Atributua	**Mota**	**Hasie-raketa**	**Deskripzioa**
$m\_toSelectModel$	ModelSelection		Azpizuhaitza hautatzeko irizpidea zehaztu
$m\_localModel$	ClassifierSplitModel		Eredu lokala nodoan
m_sons	ClassifierTree[]		Nodoaren umeak
m_isLeaf	boolean		Nodoa hostoa da
$m\_isEmpty$	boolean		Nodoa hutsik dago
m_train	Instances		Entrenamendu-instantziak
m_test	Distribution		Inausketa-instantziak
m_id	int		Nodoaren identifikatzailea
$m\_pruneTheTree$	boolean	false	Zuhaitza inausi behar da
$m\_collapseTheTree$	boolean	false	Zuhaitza kolapsatu behar da
m_CF	float	0.25f	Inausketaren konfiantza-faktorea
$m\_subtreeRaising$	boolean	true	subtreeRaising prozedura egin ala ez
$m\_cleanup$	boolean	true	Garbitu zuhaitza eraiki ondoren

Taula 3: J48 klasearen atributuak

8.1.1 ITERATIBOA BIHURTZEA

C45ItPruneableClassifierTree izeneko klase berri bat sortu da algoritmo iteratiboan lan egiteko. Klase honen superklasea *C45PruneableClassifierTree* da. Beste era batera esanda, *C45ItPruneableClassifierTree*-k *C45PruneableClassifierTree* zabaltzen du, beraz

metodo asko euren artean komunak dira. Honek aukera ematen du dagoen funtzionaltasuna aprobetxatzeko eta kode bikoiztuta ez egoteko.

[8] liburuan algoritmo errekurtsibo bat iteratibo batean nola bihurtzen den azaltzen da, algoritmoa eraikitzeko eredu gisa erabili da. Hasteko, zuhaitza sortzen duten metodoak identifikatu behar dira. *C45PruneableClassifier*-ren kasuan, honako hauek dira:

8.1.1.1 *Sailkatzailea eraikitzeko metodoa*

buildClassifier():

Metodo honek zuhaitza eraikitzeko datuak hartzen ditu parametro gisa, eta arbola eraiki egiten du. Hasteko, klaserik gabeko instantziak ezabatzen ditu eta gero, ***buildTree()*** metodora deitzen dio zuhaitza eraikitzeko. Amaitzeko, erabiltzaileak hala nahi badu, zuhaitza kolapsatu, inausi edo garbitzen du (kendu beharrezkoak ez diren edo erredundanteak diren nodoak). Azken urrats hau ez da garrantzitsua algoritmo iteratiboa eraikitzeko. *C45PruneableClassifier*-ren kasuan kodea honako hau da:

```
1 /**
2    * Method for building a pruneable classifier tree.
3    *
4    * @param data the data for building the tree
5    * @throws Exception if something goes wrong
6    */
7   public void buildClassifier(Instances data) throws Exception {
8
9     // remove instances with missing class
10    data = new Instances(data);
11    data.deleteWithMissingClass();
12
13   buildTree(data, m_subtreeRaising || !m_cleanup);
14   if (m_collapseTheTree) {
15     collapse();
16   }
17   if (m_pruneTheTree) {
18     prune();
19   }
20   if (m_cleanup) {
21     cleanup(new Instances(data, 0));
22   }
23  }
```

Kodea 14: *C45PruneableClassifier* klasearen *buildClassifier()* metodoaren inplementazioa

Metodo hau komuna da algoritmo iteratiboarentzat zein errekurtsiboarentzat; beraz, klaseak superklasearen metodoaren inplementazioa heredatzen du. Ez da beharrezkoa metodoa klasean birdefinitzea, zuzenean erabil baitezake superklasetik jasotako inplementazioa.

8.1.1.2 *Zuhaitza eraikitzeko metodoak*

Errekurtsiboaren kasuan, bi metodo erabiltzen dira zuhaitza sortzeko: *buildTree()* eta *getNewTree()*. *buildTree()* ez dago inplementatuta *C45PruneableClassifierTree*-n, *ClassifierTree* superklasetik hartzen da. Lehen ikusi den bezala, *buildClassifier*-a zuhaitzaren eraikitzaileari deitzen dio, eta honek datu-instantzia multzo bat emanda zuhaitza eraikitzen du. Gainera, beste parametro bezala boolear bat hartzen du (keepData), zuhaitza eraiki ondoren jatorrizko datuak mantendu behar diren edo ez adierazten duena. Hau erabilitako entrenamendu multzoaren informazio osoa gordetzeko egiten da, baliagarria izan daiteke analisi gehigarrietarako edo ondorengo ebaluazioetarako.

buildTree():

Metodoak sailkapen lokaleko eredu bat (azpizuhaitz bat) hautatzen du (15. kodearen 19. lerroa), emandako datu-multzoa erabilita. Azpizuhaizak adabegi ume bat baino gehiago baditu (hau da, *m_localModel.numSubsets () > 1*), multzoa azpimultzotan banatuko da eta horietako bakoitzerako zuhaitz bereiziak eraikiko dira, *getNewTree* metodoari deituz eta errekurtsibitatea erabiliz. Azpizuhaitzak azpiatala bakarra badu, horrek esan nahi du ez dagoela beste azpi-banaketarik, eta uneko nodoa hosto gisa markatzen da, zuhaitzaren puntu horretan zatiketa gehiago egingo ez direla adieraziz.

```
1  /**
2  * Builds the tree structure.
3  *
4  * @param data     the data for which the tree structure is to be generated.
5  * @param keepData is training data to be kept?
6  * @throws Exception if something goes wrong
7  */
8
9  public void buildTree(Instances data, boolean keepData) throws Exception {
10
11 Instances[] localInstances;
12 if (keepData) {
13     m_train = data;
14 }
15 m_test = null;
16 m_isLeaf = false;
17 m_isEmpty = false;
18 m_sons = null;
19 m_localModel = m_toSelectModel.selectModel(data);
20 if (m_localModel.numSubsets() > 1) {
21     localInstances = m_localModel.split(data);
22     data = null;
23     m_sons = new ClassifierTree[m_localModel.numSubsets()];
24     for (int i = 0; i < m_sons.length; i++) {
25         m_sons[i] = getNewTree(localInstances[i]);
26         localInstances[i] = null;
27     }
28 } else {
29     m_isLeaf = true;
30     if (Utils.eq(data.sumOfWeights(), 0)) {
```

```
31          m_isEmpty = true;
32      }
33      data = null;
34  }
35  }
```

Kodea 15: j48 errekurtsiboaren (originala) *buildClassifier()* metodoaren inplementazioa

getNewTree():

Metodo hau oso sinplea da. *buildTree()*-tik deitzen zaionean, zuhaitz objektu berri bat sortzen du datu berriekin, eta *buildTree()*-ri deitzen dio zuhaitz berria eraikitzeko. Bukatzeko, zuhaitz berria itzultzen du. Zerbait gaizki badoa, salbuespen bat botako du. Hau da kodea:

```
1  /**
2  * Returns a newly created tree.
3  *
4  * @param data the data to work with
5  * @return the new tree
6  * @throws Exception if something goes wrong
7  */
8  protected ClassifierTree getNewTree(Instances data) throws Exception {
9
10 C45PruneableClassifierTree newTree =
11   new C45PruneableClassifierTree(m_toSelectModel, m_pruneTheTree, m_CF,
12                  m_subtreeRaising, m_cleanup, m_collapseTheTree);
13 newTree.buildTree((Instances)data, m_subtreeRaising || !m_cleanup);
14
15 return newTree;
16 }
```

Kodea 16: j48 errekurtsiboaren (originala) *getNewTree()* metodoaren inplementazioa

Azken metodo honetan dago errekurtsibitatea, *buildTree()*-k *getNewTree()*-ri deitzen baitio eta honek berriro *buildTree()*-ri deitzen dio, zuhaitza osoa eraiki arte. 1 sasikodean ikusten da algoritmoak egiten duena.

Sasikodea 1 Zuhaitza eraiki errekurtsiboki

```
1:  procedure ZuhaitzaEraiki(m_train, keepData)
2:      if keepData is True then
3:          m_train ← data
4:      AtributuakHasieratu()
5:      azpizuhaitza ← azpizuhaitzaHautatu()
6:      if Adabegi ume bat baino gehiago then
7:          ZatituAzpizuhaitza(azpizuhaitza)
8:          for all ume in m_sons do
9:              ZuhaitzBerriaEraikiErrekurtsiboki(ume, keepData)
10:     else
11:         MarkatuNodoaHostoGisa()
```

Behin hau ikusita, kode bera sortzeko modu bat bilatu behar da, baina modu iteratiboan. Hori egiteko zerrenda bat erabiltzea erabaki da. Hasteko, erro nodoa sartu behar da zerrendan eta orduan, algoritmoa begizta batean sartuko da. Zerrendan dagoen lehenengo nodoa hartuko da eta aurreko kode errekurtsiboa egiten duen bera egingo da, baina, errekurtsiboki eraiki beharrean *getNewTree()* erabiltzen, zerrendan gehituko dira nodo umeak eta begiztaren hurrengo iterazioan zerrendaren lehenengo nodoa tratatuko da, hutsik egon arte. Hobeto ulertzeko ondoren dago algoritmo iteratiboaren sasikodea (2. sasikodea).

Sasikodea 2 Zuhaitza eraiki iteratiboki

```
1:  procedure ZuhaitzaEraikiIteratiboa(m_train, keepData)
2:      ZerrendaSortu()
3:      LehenNodoaZerrendanSartu()
4:      while Zerrenda ez dago hutsik do
5:          unekoNodoa ← ZerrendarenLehehengoNodoaHartu()
6:          ZerrendarenLehehengoNodoaEzabatu()
7:          if keepData is True then
8:              m_train ← data
9:          AtributuakHasieratu()
10:         oraingoAzpizuhaitza ← oraingoNodoarenAzpizuhaitzaHautatu()
11:         if Adabegi ume bat baino gehiago then
12:             ZatituOraingoAzpizuhaitza(oraingoAzpizuhaitza)
13:             for all seme in m_sons do
14:                 UmeZuhaitzBerriaZerrendanSartu(seme, keepData)
15:         else
16:             MarkatuNodoaHostoGisa()
```

Ikus daitekeenez, algoritmoa ia bera da, baina desberdintasun bat dago: orain, nodoak modu errekurtsiboan sortu beharrean, zerrendan gehitzen dira eta zerrendako lehena hartzen da beti, zerrenda hutsik egon arte. Horrela, zuhaitza preordenan eraikiko da algoritmo errekurtsiboaren orden berean, baina modu iteratiboan.

Javako kodeaz hitz egiten, lehen esan bezala, kasu honetan ez da beharrezkoa *buildClassifier()* metodoa berriro inplementatzea.

buildTree() metodoan *util.ArrayList* bat erabili da zerrenda bezala nodoak sartzen joateko. Sasikodean ikusi den moduan, jada ez da beharrezkoa *getNewTree()* izeneko metodo bat izatea, honek errekurtsibitatea sartzen zuelako, orain, zuzenean umea gehitzen zaio zerrendari. Ondoren azalduko da *buildTree()* iteratiboaren kodea:

Hasteko, zerrenda sortzen da eta lehenengo nodoa sartzen zaio. Baina zerrendan ez dira nodoak sartzen, objektuak baizik. Objektuak, alde batetik, zuhaitza egiteko datuak izango ditu, eta, bestetik, zuhaitza bera. Erroaren kasuan, *this* da, hau da, uneko zuhaitza. 17. kodean *buildTree()* metodoaren lehenengo lerroak agertzen dira.

```
1 public void buildTree(Instances data, boolean keepData) throws Exception {
2 ArrayList<Object[]> list = new ArrayList<>();
3 list.add(new Object[] {data, this});
```

Kodea 17: j48 iteratiboaren *buildTree()* metodoaren lehenengo zatia

Ondoren, algoritmoa begiztan sartzen da. Lehenengo objektua aldagai batean gordetzen da eta objektua zerrendatik ezabatzen da begiztaren hurrengo iterazioan ez hartzeko. Zuhaitza *currentTree* aldagaian gordetzen da, exekuzio osoan erabiliko baita. Hemendik aurrera, algoritmo errekurtsiboa egiten duen estrategia bera egiten da, gauza bat aldatuz: denbora guztian uneko nodoaz (*currentTree*) hitz egiten da, hau da, zerrendan lehena zegoena. Uneko nodoarentzako eredu bat hautatzen da (20. lerroa), azpimultzo bat baino gehiago dagoen ikusten da (21. lerroa), eta umeen objektuak zerrendari gehitzen zaizkio (27. lerroa).

```
4 Instances[] localInstances;
5 while (list.size() > 0) {
6     Object[] current = list.get(0);
7     list.set(0, null); // Null to free up memory
8     list.remove(0);
9     Instances currentData = (Instances) current[0];
10    C45PruneableClassifierTreeIt currentTree = (C45PruneableClassifierTreeIt) current
          [1];
11
12    Instances[] localInstances;
13    if (keepData) {
14        currentTree.m_train = currentData;
15    }
16    currentTree.m_test = null;
17    currentTree.m_isLeaf = false;
18    currentTree.m_isEmpty = false;
19    currentTree.m_sons = null;
20    currentTree.m_localModel = currentTree.m_toSelectModel.selectModel(currentData);
21    if (currentTree.m_localModel.numSubsets() > 1) {
22        localInstances = currentTree.m_localModel.split(currentData);
23        currentData = null;
24        currentTree.m_sons = new ClassifierTree[currentTree.m_localModel.numSubsets()];
25        for (int i = 0; i < currentTree.m_sons.length; i++) {
26            ClassifierTree newTree = new C45PruneableClassifierTreeIt(currentTree.
                  m_toSelectModel, m_pruneTheTree, m_CF, m_subtreeRaising, m_cleanup,
                  m_collapseTheTree);
27            list.add(new Object[] {localInstances[i], newTree});
28            currentTree.m_sons[i] = newTree;
29        }
30    } else {
31        currentTree.m_isLeaf = true;
32        if (Utils.eq(currentData.sumOfWeights(), 0)) {
33            currentTree.m_isEmpty = true;
34        }
35        currentData = null;
36    }
37 }
38 }
```

Kodea 18: j48 iteratiboaren *buildTree()* metodoaren bigarren zatia

Umeak sortzen dituen begiztan zentratzen bagara, algoritmo errekurtsiboan *getNewTree()* deitzen zen bitartean, kasu honetan, umeari dagozkion instantziak eta zuhaitz berria dituen objektu bat gehitzen du zerrendara. Amaitzeko, ume hori ezartzen da guraso

nodoaren umetzat.

```
25 for (int i = 0; i < currentTree.m_sons.length; i++) {
26     ClassifierTree newTree = new C45PruneableClassifierTreeIt(currentTree.
           m_toSelectModel, m_pruneTheTree, m_CF, m_subtreeRaising, m_cleanup,
           m_collapseTheTree);
27     list.add(new Object[] {localInstances[i], newTree});
28     currentTree.m_sons[i] = newTree;
29 }
```

Kodea 19: j48 iteratiboaren *buildTree()* metodoaren umeen sorketa bakarrik

Hurrengo iterazioan berriz ere lehenengoa hartuko da eta preorden-ean eraikitzen den zuhaitza sortuko da. Errekurtsiboki egindako zuhaitza eta iteratiboki egindako zuhaitza berdin-berdinak izango dira.

8.1.2 IRIZPIDE BERRIAK SARTU

Orain algoritmoa iteratiboa dela, zuhaitza eraikitzeko irizpide berriak sar daitezke. Horretarako, bi parametro berri gehitu dira:

- *m_maximumCriteria*: *int* motako parametro honek eraikitzeko barne nodo (edo maila) kopuru maximo bat aukeratzeko balio du. Erabiltzaileak aukera dezake zein barne nodo edo maila arte nahi duen zuhaitz kontsolidatua eraikitzea. Aukeratutako maximoa zuhaitzarena baino handiagoa bada, zuhaitza osoa eraikiko da. Adibidez, zuhaitzak 5 maila baino ez baditu eta erabiltzaileak gehienez 7 jartzea erabakitzen badu, zuhaitza osorik eraikiko da. Gauza bera nodo kopuruarekin.
- *m_priorityCriteria*: Parametro honekin erabiltzaileak zuhaitzaren nodoak zein ordenatan eraiki nahi dituen aukeratzen du. Sei aukeren artean hauta dezake:
  - **Originala:** Zuhaitza algoritmo errekurtsiboa bezala eraikitzea. Maximorik gabe. Parametro hau erabilgarriagoa izango da PCTBagging gidatua eraikitzeko. Honetan preorder-en berdina egiten du.
  - **LevelByLevel:** Zuhaitza mailaz maila eraikitzea. Hau da, lehenengo 1. maila osoa, gero 2. maila osorik, eta horrela maila maximora iritsi arte.
  - **Preorder:** Zuhaitza aurreordenan eraiki (6. atalan azaldu den bezala).
  - **Size:** Populazio handiena duen nodotik txikienera zuhaitza eraikitzea.
  - **Gainratio:** Irabazi-erlazio (gain-ratio) handiena duen nodotik txikienera zuhaitza eraikitzea.
  - **Gainratio normalizatua:** Irabazi-erlazio normalizatuko (gain-ratio bider populazio tamaina) handienetik txikienera zuhaitza eraikitzea.

m_order izeneko atributu berri bat sartu da ere, *int* motakoa. Bere helburua nodoak garatzen ari diren ordenaren erregistro bat eramatea da. 0an hasten da eta handitu egiten da nodoen garapenak aurrera egin ahala; beraz, uneko nodoaren orden-zenbakia adierazten du.

Inplementazioari dagokionez, gauza batzuk gehitu zaizkio 18. kodeari. Garrantzitsuena da orain zerrendak ez dituela objektuak datuekin eta zuhaitzarekin gordetzen; orain, aukeratutako irizpidearen balioa eta nodo horren maila ere gordetzen direla (0 erroa bada). Aukeratutako irizpidearen balioa (*orderValue* kodean) erabiltzaileak aukeratutakoaren arabera aldatzen da; tamaina aukeratu bada, nodo horren populazio-tamaina gordeko da; gainratioa aukeratu bada, informazio hori gordeko da, eta horrela gainerakoekin. Mailaka eraikitzea aukeratu bada, elementu horrek ez du ezertarako balioko.

```
list.add(new Object[] { data, this, null, 0});//(Data, tree, orderValue, currentLevel)
```

internalNodes izeneko aldagai bat gehitu da. Aldagai honek eraikitzen ari den zuhaitzak orain arte dituen barne-nodoen kopurua gordetzen du eta erabiltzaileak aukeratutako maximora iristen denean (baldin badago) zuhaitzaren eraikuntza gelditzeko balio du. 0tik hasten da, eta handitu egiten da nodoak zerrendara gehitu ahala.

Bestalde, lehen konparazioa egiten zen lekuan, azpizuhaitzean azpimultzo bat baino gehiago zegoen ikusteko ($currentTree.m\_localModel.numSubsets() > 1$, 18. kodearen 21. lerroa), orain, horretaz gain, beste hiru gauza alderatzen dira:

- ($m\_priorityCriteria == J48It.Original$): Erabiltzaileak zuhaitza sortzea inolako murrizketarik gabe erabaki du.
- ($m\_priorityCriteria == J48It.Levelbylevel$) && ($currentLevel < m\_maximumCriteria$): Erabiltzaileak zuhaitza mailaka eraikitzea aukeratu du, eta oraindik ez da iritsi aukeratutako mailara.
- ($m\_priorityCriteria > J48It.Levelbylevel$) && ($internalNodes < m\_maximumCriteria$): Erabiltzaileak zuhaitza nodoen arabera eraikitzea aukeratu du, eta oraindik ez da aukeratutako nodo kopurura iritsi.

Jakina, azpimultzoen konparazioa eta konparazio berrietako bat ere betetzen ez bada, zuhaitza ez da eraikitzen jarraituko, eta aurreko kodearen 31. lerrora igaroko da. 8.2.2. atalean geroago azalduko den bezala, J48It.Original eta J48It.Levelbylevel aukerak int motako *enumeration* bateko elementuak dira.

Amaitzeko, aukeratutako irizpidearen arabera, umeak zerrendan orden batean edo bestean sartuko dira. Bakoitzaren *orderValue* aldagaiaren balioa kalkulatuko da, eta, ondoren, umeak zerrendari gehituko zaizkio *addSonOrderedByValue()* metodoa erabiliz

(20. kodea). Metodo honek ume berriak izan behar duen kokapena bilatzen du zerrenda handitik txikira ordenatuta egon dadin.

```
1 public void addSonOrderedByValue(ArrayList<Object[]> list, Object[] son) {
2 if (list.size() == 0) {
3     list.add(0, son);
4 } else {
5     double sonValue = (double) son[2];
6     for (int i = 0; i < list.size(); i++) {
7
8         double parentValue = (double) list.get(i)[2];
9         if (parentValue < sonValue) {
10            list.add(i, son);
11            break;
12        }
13        if (i == list.size() - 1) {
14            list.add(son);
15            break;
16        }
17    }
18 }
19 }
```

Kodea 20: *addSonOrderedByValue()* metodoaren inplementazioa

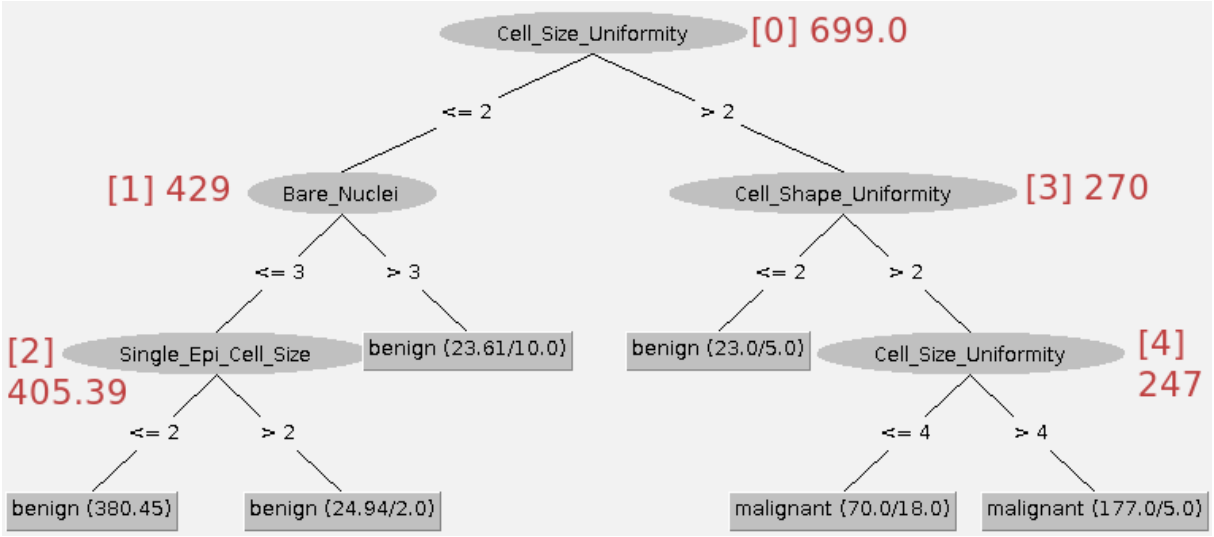
Jarraian, irizpide bakoitzaren inplementazioari eta funtzionamenduari dagokienez dauden aldeak azalduko dira:

Tamaina

Erabiltzaileak aukeratutako irizpidea nodo bakoitzeko populazioaren tamaina bada, *orderValue* nodo horren azpizuhaitzaren banaketaren *perBag*-a izango da. *perBag* erabaki-zuhaitzaren azpizuhaitzean bagging-prozesuan sortutako datu-multzoaren edo poltsaren (*bag*) tamaina edo instantzia-kopurua da. Kasu honetan azpizuhaitzaren tamaina jakiteko erabiltzen da. Honela lortuko da:

```
orderValue = currentTree.m_localModel.distribution().perBag(i);
Object[] son = new Object[] { localInstances[i], newTree, orderValue, currentLevel + 1
    };
addSonOrderedByValue(list, son);
```

Jarraian, zuhaitza baten adibide bat dago, datu-base bat erabiliz eta gehienez 5 barne nodo jarriz, nodoak tamainaz eraikiz. Gorriz idatzita agertzen da nodo bakoitzaren *orderValue*-ren balioa, kasu honetan tamaina (balio hauek *debugger*-etik ikus daitezke) eta nodo hori zer posizioan garatu den.



Irudia 16: J48 iteratiboa algoritmoarekin sortutako zuhaitza, nodoak tamainaren arabera garatuz

Ikus daitekeenez, 5 barne-nodo daude (hostoak ez diren nodoak eta ume bat baino gehiago dutenak, borobilak irudian). Atal honen amaieran agertzen den *dumpTree()* metodoari esker (21. kodea), arbola ere agertzen da *Explorer*-aren irteeran. Kortxete artean ([ ]) nodo bakoitza zein ordenatan gehitu den agertzen da:

```
[0] Cell_Size_Uniformity <= 2
|   [1] Bare_Nuclei <= 3
|   |   [2] Single_Epi_Cell_Size <= 2: [3] benign (380.45)
|   |   [2] Single_Epi_Cell_Size > 2: [8] benign (24.94/2.0)
|   [1] Bare_Nuclei > 3: [9] benign (23.61/10.0)
[0] Cell_Size_Uniformity > 2
|   [4] Cell_Shape_Uniformity <= 2: [10] benign (23.0/5.0)
|   [4] Cell_Shape_Uniformity > 2
|   |   [5] Cell_Size_Uniformity <= 4: [7] malignant (70.0/18.0)
|   |   [5] Cell_Size_Uniformity > 4: [6] malignant (177.0/5.0)
```

Irudia 17: 16. irudiaren zuhaitzaren irteera (testu moduan)

16. eta 17. irudietan agertzen diren zenbakiak berdinak dira, zuhaitz berekoak baitira. Lehen esan dugun bezala, kortxeteen zenbakiak nodo bakoitza zein ordenatan gehitu den adierazten dute, 0tik hasita. Kasu honetan, *Cell_Size_Uniformity* nodoa erroa da (0), erantsitako hurrengo nodoa bere umea *Bare_Nuclei* (1), hurrengo handiena *Single_Epl_Cell_Size* izango litzateke (2), gero hirugarrena eta horrela zuhaitzaren nodo guztiekin.

Irabazi-ratioa

Erabiltzaileak irabazi-ratioa aukeratzen badu, *orderValue* irabazi-ratioa bera izango da. Gainratioa erabaki-zuhaitzetako neurria da, eta atributu baten garrantzia ebaluatzen

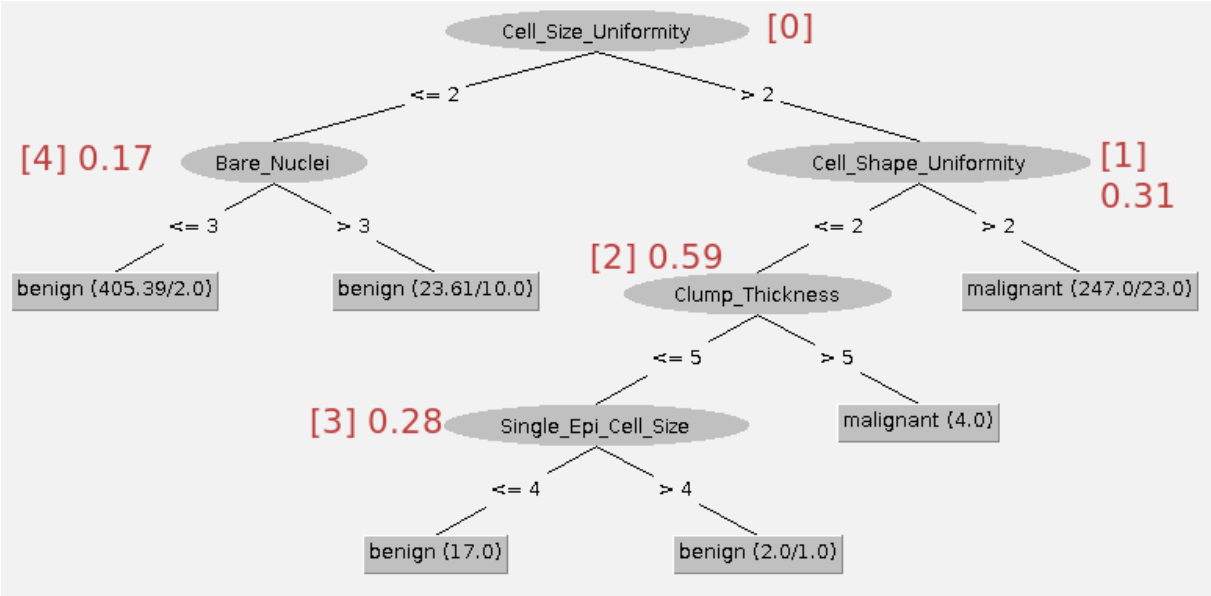
du. Datuak atributuaren berezko informazioarekin zatitzean lortutako informazioaren irabazia konbinatzen du. Informazio-atributuak hautatzen laguntzen du eta balio askoko atributuak nahiago izatea saihesten du.

Umearen azpizuhaitzak azpimultzo bat baino gehiago badu *orderValue .gainRatio()* egiten lortzen da. Bestela, oso balio txiki bat ezarriko zaio (MIN_VALUE) irabazi-ratioari buruzko informazio nahikorik ez dagoelako eta inoiz aukeratua ez izateko. Honela geratuko litzateke:

```
if (sonModel.numSubsets() > 1) {

    orderValue = ((C45Split) sonModel).gainRatio();
} else {
    orderValue = (double) Double.MIN_VALUE;
}
Object[] son = new Object[] { localInstances[i], newTree, orderValue, currentLevel + 1
    };
addSonOrderedByValue(list, son);
```

Hau aurreko datu base bereko zuhaitza da (16. irudia), baina nodoak irabazi-ratioaren arabera eraikiz:



Irudia 18: Zuhaitza irabazi-ratioaren arabera garatuz

Erro nodoa eraikitzen den lehena da (0), dagoen bakarra baita. Ondoren, bi umeen artean zein eraiki aukeratu behar da, *Bare_ Nuclei* eta *Cell_ Shape_ Uniformity*-ren artean irabazi-ratio handiena duena bigarrena da, beraz, hori eraikitzen da (1). Hurrengo iterazioan, erroaren beste umea (*Bare_ Nuclei*) eta *Cell_ Shape_ Uniformity*-aren umeen artean aukeratu behar da eta gehien irabazi-ratio duena *Clump_ Thickness* da, beraz eraikitzen den hurrengoa da (2). Horrela, 5 barne-nodo eraiki arte.

Hurrengo irudian aurreko zuhaitzaren (18. irudia) irteera testu moduan agertzen da.

```
[0] Cell_Size_Uniformity <= 2
|   [4] Bare_Nuclei <= 3: [7] benign (405.39/2.0)
|   [4] Bare_Nuclei > 3: [5] benign (23.61/10.0)
[0] Cell_Size_Uniformity > 2
|   [1] Cell_Shape_Uniformity <= 2
|   |   [2] Clump_Thickness <= 5
|   |   |   [3] Single_Epi_Cell_Size <= 4: [9] benign (17.0)
|   |   |   [3] Single_Epi_Cell_Size > 4: [10] benign (2.0/1.0)
|   |   [2] Clump_Thickness > 5: [8] malignant (4.0)
|   [1] Cell_Shape_Uniformity > 2: [6] malignant (247.0/23.0)
```

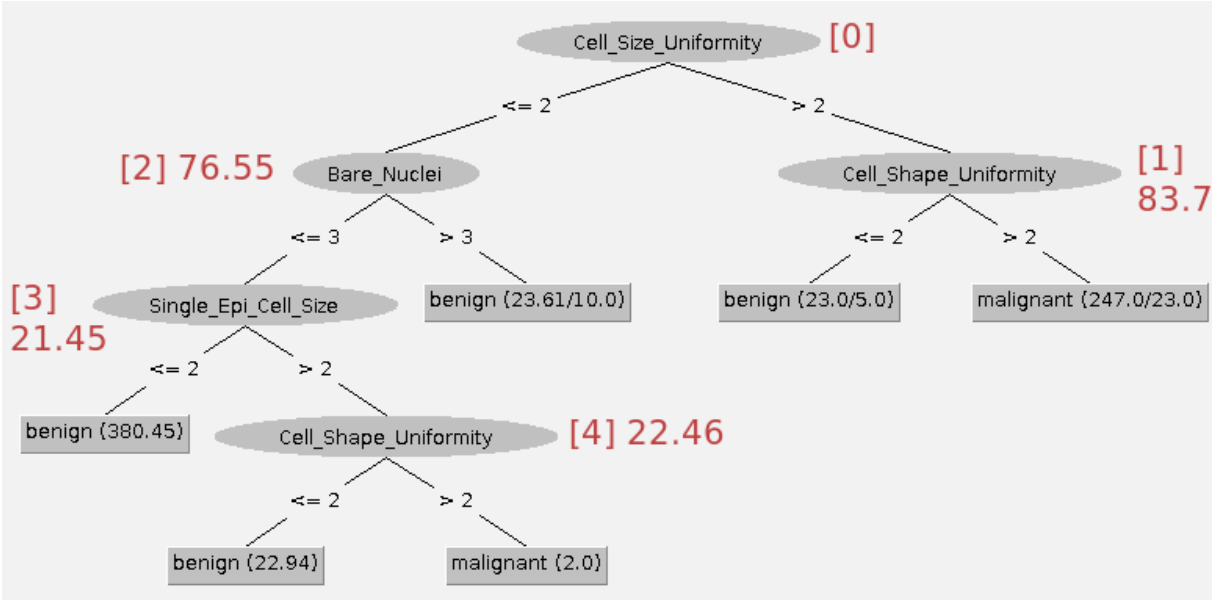
Irudia 19: 18. irudiaren zuhaitzaren irteera (testu moduan)

Irabazi-ratio normalizatua

Erabiltzaileak aukeratutako irizpidea irabazi-ratio normalizatua bada, aurreko bien konbinazioa egingo da, gainratioa tamainaz biderkatuko da. Lehenengo tamaina kalkulatuko da eta irabazi-ratioarekin biderkatuko da:

```
double size = currentTree.m_localModel.distribution().perBag(i);
double gainRatio;
ClassifierSplitModel sonModel = ((C45ItPruneableClassifierTree) newTree).
    m_toSelectModel.selectModel(localInstances[i]);
if (sonModel.numSubsets() > 1) {
    gainRatio = ((C45Split) sonModel).gainRatio();
    orderValue = size * gainRatio;
} else {
    orderValue = (double) Double.MIN_VALUE;
}
Object[] son = new Object[] { localInstances[i], newTree, orderValue, currentLevel + 1
    };
addSonOrderedByValue(list, son);
```

Eta jarraian dago ateratako zuhaitza (Irudia 20). Ikusten denez aurreko bien desberdina da.



Irudia 20: Zuhaitza irabazi-ratio normalizatuaren arabera garatuz

Irabazi-ratioarekin bezala, ikusi diren nodoen artean balio handiena duena aukeratzen da. Hurrengo irudian (Irudia 21) zuhaitza ikusten da testu formatuan.

```
[0] Cell_Size_Uniformity <= 2
|   [2] Bare_Nuclei <= 3
|   |   [3] Single_Epi_Cell_Size <= 2: [8] benign (380.45)
|   |   [3] Single_Epi_Cell_Size > 2
|   |   |   [4] Cell_Shape_Uniformity <= 2: [9] benign (22.94)
|   |   |   [4] Cell_Shape_Uniformity > 2: [10] malignant (2.0)
|   [2] Bare_Nuclei > 3: [7] benign (23.61/10.0)
[0] Cell_Size_Uniformity > 2
|   [1] Cell_Shape_Uniformity <= 2: [6] benign (23.0/5.0)
|   [1] Cell_Shape_Uniformity > 2: [5] malignant (247.0/23.0)
```

Irudia 21: 20. irudiaren zuhaitzaren irteera (testu moduan)

Mailaz maila

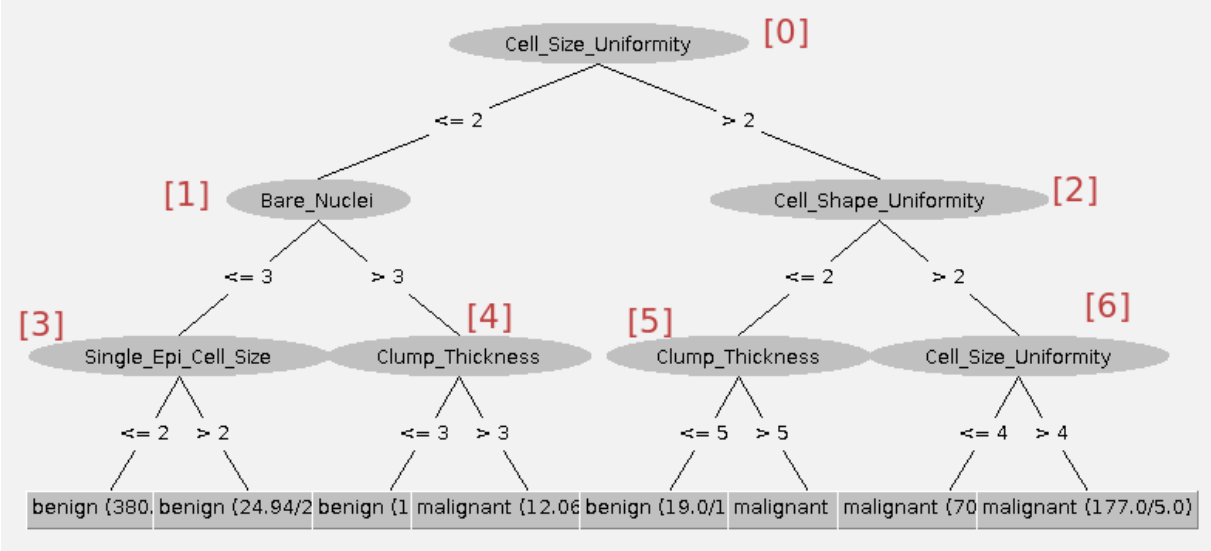
Aukeratutako irizpidea mailaz maila edo preorder (edo Original) bada, umea gehituko da *listSons* umeen-zerrendan. Ez delako beharrezkoa *orderValue* kalkulatzea, umeen zerrenda zein posiziotan gorde behar den ikustea besterik ez dago.

```
listSons.add(new Object[] { localInstances[i], newTree, 0, currentLevel + 1 });
```

Erabiltzaileak aukeratutako irizpidea zuhaitza mailaka sortzea bada, barne nodoen zerrendari umeen zerrenda gehituko zaio amaieran. Horrela, maila bakoitzeko nodoak zerrendara gehitzen dira hurrengo mailara igaro aurretik. Beraz, *list* zerrendak lehenengo mailako nodo guztiak jasoko ditu lehenik, gero bigarren mailako guztiak, eta horrela hurrenez hurren.

```
list.addAll(listSons);
```

Adibide honetan (22. irudia) maximoa 3ra aldatu da, bestela, zuhaitzak 5 maila baino ez dituenez osoa eraikiko zen.



Irudia 22: Zuhaitza mailaz maila garatuz, 3 maximoarekin

Aurreko adibideetan bezala, hurrengo irudian (23. irudia) zuhaitza ikus daiteke testu formatuan. Argi ikus daiteke zuhaitza mailaka eraiki dela.

```
[0] Cell_Size_Uniformity <= 2
|   [1] Bare_Nuclei <= 3
|   |   [3] Single_Epi_Cell_Size <= 2: [7] benign (380.45)
|   |   [3] Single_Epi_Cell_Size > 2: [8] benign (24.94/2.0)
|   [1] Bare_Nuclei > 3
|   |   [4] Clump_Thickness <= 3: [9] benign (11.55)
|   |   [4] Clump_Thickness > 3: [10] malignant (12.06/2.06)
[0] Cell_Size_Uniformity > 2
|   [2] Cell_Shape_Uniformity <= 2
|   |   [5] Clump_Thickness <= 5: [11] benign (19.0/1.0)
|   |   [5] Clump_Thickness > 5: [12] malignant (4.0)
|   [2] Cell_Shape_Uniformity > 2
|   |   [6] Cell_Size_Uniformity <= 4: [13] malignant (70.0/18.0)
|   |   [6] Cell_Size_Uniformity > 4: [14] malignant (177.0/5.0)
```

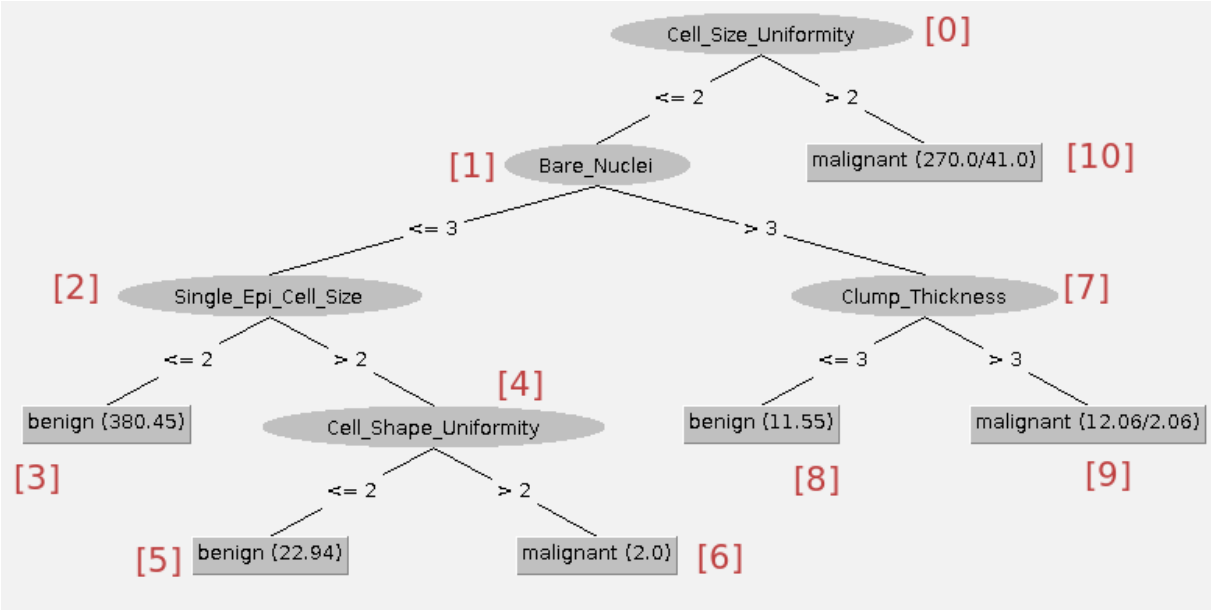
Irudia 23: 22. irudiaren zuhaitzaren irteera (testu moduan)

Preorder

Erabiltzaileak aukeratutako irizpidea preordenan (edo jatorrizkoa) bada, *list* elementuak *listSons*-era gehitzen dira; horrela, nodoak *listSons* zerrendaren hasieran gehitzen dira, eta *list* zerrendan gehitutako lehen nodoa umeen zerrendako lehen nodoa izango da.

Ikuspegi horrekin, preordenean eraikitzea lortzen da. Erabiltzaileak 'Original' aukeratu badu, preordenan eraikiko da ere baina maximorik gabe, beraz, zuhaitz osoa eraikiko da beti.

```
listSons.addAll(list);
list = listSons;
```



Irudia 24: Zuhaitza aurreordenean garatua

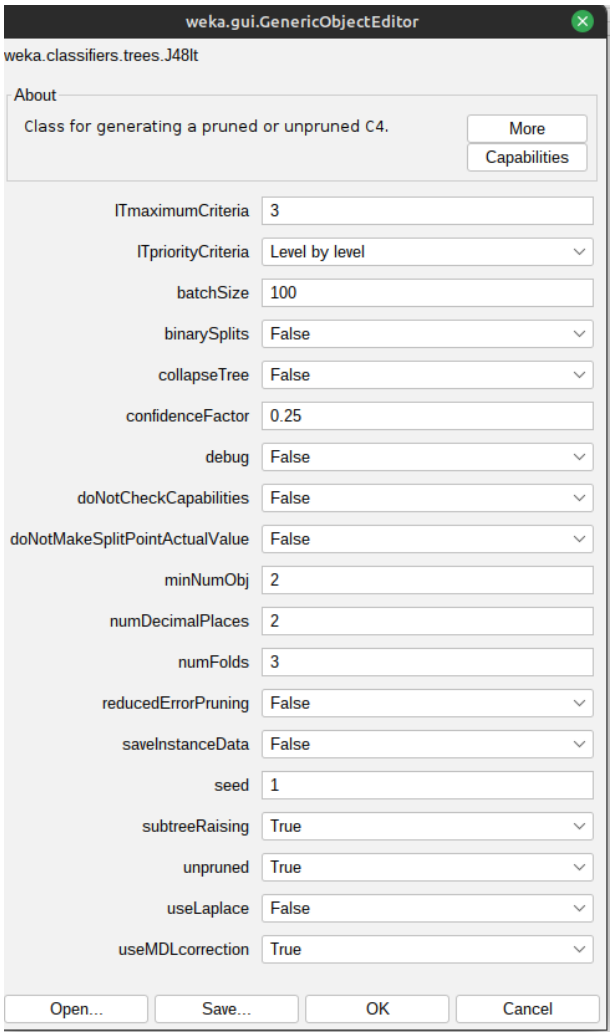
Nodoak zein ordenatan eraiki diren aztertuz gero, zuhaitza aurreordenan eraikita dagoela ikus daiteke, lehenengo ezkerreko aldea eraikitzen baita eta gero eskuinekoa. Barneko nodoak ez direnak (laukiak 24. irudian) eraiki diren ordena dute ere, baina ez dira kontuan hartzen maximorako. Adibideekin amaitzeko, hona hemen zuhaitza testu formatuan (Irudia 25).

```
[0] Cell_Size_Uniformity <= 2
|   [1] Bare_Nuclei <= 3
|   |   [2] Single_Epi_Cell_Size <= 2: [3] benign (380.45)
|   |   [2] Single_Epi_Cell_Size > 2
|   |   |   [4] Cell_Shape_Uniformity <= 2: [5] benign (22.94)
|   |   |   [4] Cell_Shape_Uniformity > 2: [6] malignant (2.0)
|   [1] Bare_Nuclei > 3
|   |   [7] Clump_Thickness <= 3: [8] benign (11.55)
|   |   [7] Clump_Thickness > 3: [9] malignant (12.06/2.06)
[0] Cell_Size_Uniformity > 2: [10] malignant (270.0/41.0)
```

Irudia 25: 24. irudiaren zuhaitzaren irteera (testu moduan)

Adibide hauekin, erabiltzaileak aukeratutako irizpidearen arabera zuhaitzak desberdinak direla ikus daiteke, izan ere, zuhaitza eraikitzeko hurrengo nodoa aukeratzeko modua desberdina da irizpide bakoitzean.

Sailkatzaile iteratibo hau GUI-an erabili ahal izateko, 7.5.2. atalaren urratsak jarraitu dira. Nola egin den zehatz mehatz aurrerago azalduko da 8.2.2. atalean PCTBagging algoritmo gidatuarekin. Interfazea J48-aren berdina da, bi parametro berriekin, parametro hauek 'IT' aurrizkia dute algoritmo iteratiboaren berriak direla jakiteko (ikusi Irudia 26):



Irudia 26: j48 iteratibo algoritmoaren interfazea

Bukatzeko, zuhaitza *Explorer*-aren irteeratik ikus dadin, sailkapen-zuhaitzek WEKAn duten *dumpTree()* funtzioa erabili da, baina berriz definitu da nodo bakoitzaren ordena ager dadin. Funtzioa *ClassifierTree* klaseko berbera da, baina 20. lerroa (ikusi kodea 21) gehitu da nodo bakoitzaren garapen-ordena ager dadin. Hori lortzeko, aurretik definitutako *m_order* atributua erabili da.

```
1 /**
2 * Help method for printing tree structure.
3 *
4 * @param depth the current depth
5 * @param text  for outputting the structure
6 * @throws Exception if something goes wrong
7 */
```

```
8 public void dumpTree(int depth, StringBuffer text) throws Exception {
9 int i, j;
10 for (i = 0; i < m_sons.length; i++) {
11     text.append("\n");
12     for (j = 0; j < depth; j++) {
13         text.append("|   ");
14     }
15     text.append("[" + m_order + "] ");
16     text.append(m_localModel.leftSide(m_train));
17     text.append(m_localModel.rightSide(i, m_train));
18     if (m_sons[i].isLeaf()) {
19         text.append(": ");
20         text.append("[" + ((C45ItPruneableClassifierTree) m_sons[i]).m_order + "] ");
21         text.append(m_localModel.dumpLabel(i, m_train));
22     } else {
23         m_sons[i].dumpTree(depth + 1, text);
24     }
25 }
26 }
```

Kodea 21: *dumpTree()* metodoaren bertsio berria

8.2 PCTBAGGING GIDATUA

PCTBagging gidatua PCTBagging-ean oinarritzen da, aldi berean CTC-n oinarritzen dena, zeina, aldi berean, C4.5 algoritmoan oinarrituta dagoena. Hiru sailkatzaile hauen funtzionamendua 5.3. atalean azaldu da. PCTBgidatuak hiru sailkatzaile horiek dituzten parametroak ere izango ditu. RM aurrizkia dutenak CTC sailkatzailekoak dira, eta PCTB aurrizkikoak PCTBagging-ekoak.

PCTBag gidatua sailkatzailea sortzeko metodoak *C45ItPartiallyConsolidatedPruneableClassifierTree* izeneko klase berri batean doaz; klase hau *weka.classifiers.trees.j48itPartiallyConsolidated* paketean dago. PCTBagging sailkatzailearen metodoak *weka.classifiers.trees.j48PartiallyConsolidated* paketean doaz eta CTC-koak *weka.classifiers.trees.j48Consolidated* paketean.

8.2.1 FUNTZIONALITATE BERRIAK SARTU

PCTBagging gidatua algoritmoan zentratzen, interesatzen den parametroa kontsolidazio ehunekoa da. Lehen azaldu den bezala, zuhaitz partzialean kontsolidatu beharreko nodoak adierazten ditu parametro honek. Parametro honen eta J48 iteratiboaren *maximumCriteria* parametroaren funtzionamendua berdina da (ikusi 8.1. atala) eta, beraz, inplementazioaren zati honetan berrerabil daiteke. Superklasean erabiltzen den parametro bera erabiltzen da; beraz, PCTgidatuaren kasuan, parametroaren izena *consolidationPercent* izango litzateke, hau da, PCTBag inplementazioaren izen bera.

PCTgidatuaren inplementazioan, lehenik eta behin PCTBag-aren funtzionamenduari

erreparatu behar zaio. 22. kodean Java inplementazioa agertzen da. Sailkatzailearen eraikitzaileak (*buildClassifier()*) hiru gauza egiten ditu funtsean:

1. **Eraiki zuhaitz osoa**
2. **Kimatu zuhaitza** erabiltzaileak kontsolidazio-ehunekoaren bidez erabakitako nodo handienak soilik gera daitezen.
3. **Bagging**-a aplikatu

```
1 /**
2 * Method for building a pruneable classifier consolidated tree.
3 *
4 * @param data the data for pruning the consolidated tree
5 * @param samplesVector the vector of samples for building the consolidated tree
6 * @param consolidationPercent the value of consolidation percent
7 * @throws Exception if something goes wrong
8 */
9 public void buildClassifier(Instances data, Instances[] samplesVector, float
      consolidationPercent) throws Exception {
10
11 buildTree(data, samplesVector, m_subtreeRaising || !m_cleanup);
12 if (m_collapseTheTree) {
13     collapse();
14 }
15 if (m_pruneTheTree) {
16     prune();
17 }
18 leavePartiallyConsolidated(consolidationPercent);
19 applyBagging();
20
21 if (m_cleanup)
22     cleanup(new Instances(data, 0));
23 }
```

Kodea 22: PCTBagging algoritmoaren *buildClassifier()* metodoa

PCTBagging aldaera berriaren kasuan, metodo hau aldatu egiten da, ez baita zuhaitza eraiki behar gero kimatzeko, baizik eta zuhaitza pixkanaka eraikitzen joan behar da. Kasu honetan, hauxe egiten du:

1. Kontsolidatu beharreko **nodo kopurua kalkulatzen da**
2. **Zuhaitza modu iteratiboan eraikitzen** da
3. **Bagging-a** aplikatzen da

Sailkatzailea eraginkorragoa da erabiltzaileak zenbaki zehatz bat ematen badu kontsolidatu nahi dituen barne nodo zenbakirako, baina askotan erabiltzaileak ez daki zuhaitz batek zenbat nodo izango lituzkeen osoa eraikitzen bada. Hori dela eta, erabiltzaileak, zenbaki gisa ez ezik, kontsolidatu beharreko barne-nodoak portzentaje gisa erabakitzeko aukera izateko aukera ere gehitu da (PCTBagging-ek egiten duen bezala). PCTBag-en

consolidationPercent izeneko parametroa berrerabili da, hau da, kontsolidatu behar den nodo kopurua. PCTBag-ek ez bezala, algoritmo honek nodoak modu iteratiboan eraikitzen ditu mugara iritsi arte. Beraz, muga nodo kopurua zein den jakiteko modu bakarra zuhaitza osoa eraikiz gero izango lituzkeen nodoak jakitea da. Hau askoz zamatsuagoa da.

Horretarako, beste parametro bat gehitu zaio sailkatzaile honi. Parametro honi *consolidationPercentHowToSet* deitu zaio, eta erabiltzaileari aukera ematen dio aukeratu ahal izateko nodo edo maila kopuruaren kontsolidazioa ehuneko bat izatea (*based on percentage value (%)*) edo, aitzitik, balio jakin bat aukeratzea (*using a numeric value*) j48Iteratibo algoritmoan egiten den bezala. Jakina, erabiltzaileak aukeratu duenaren arabera, *buildClassifier()*-ak gauza bat edo beste bat egingo du; balio jakin bat aukeratuz gero, ez da beharrezkoa zuhaitza aldez aurretik eraikitzea nodo kopurua zein den jakiteko. Berriz, ehuneko bat aukeratuz gero, lehenengo zuhaitz osoa eraikiko da eta gero zuhaitz osoaren nodo kopuruaren arabera kalkulatuko da nodo kopurua. Ehuneko bat zein zenbaki-balio bat aukeratuz gero, parametroa bera izango da (*consolidationPercent*) kodea sinplifikatzeko.

Azkenik, J48Iteratiboan bezala, beste parametro bat gehitu da erabiltzaileak nodoak nola eraiki nahi dituen erabaki dezan. Parametro honek *m_priorityCriteria* du izena.

Parametro hauen inplementazioa aurrerago azalduko da 8.2.2. atalean.

Parametro berriekin PCTgidatuaren *buildClassifier()* aldatu egiten da, erabiltzailearen hautaketaren arabera gauza bat edo beste bat egin beharko baitu. Pixkanaka azalduko da metodo honek zer egiten duen:

Hasteko, PCTgidatutik PCTBagging algoritmoa erabiltzeko aukera ematen zaio erabiltzaileari. Erabiltzaileak *Original* aukeratzen badu lehentasun-irizpidetzat superklaseko *buildClassifier()* metodoa deituko da. Horrela, PCTBag algoritmoa erabiltzeko aukera ematen da, sailkatzailea aldatu behar izan gabe.

```
if (m_priorityCriteria == J48It.Original) {
   super.buildClassifier(data, samplesVector, consolidationPercent);}
```

Kodea 23: Lehentasun-irizpidea 'Original' gisa ezartzen bada *buildClassifer*-en kode zatia

Erabiltzaileak beste edozein aukera aukeratzen badu, beste bi aukera daude: kontsolidatu beharreko nodoak adieraztea balio edo ehuneko baten bidez.

Ehunekoa aukeratuz gero, lehen esan bezala, zuhaitza osoa eraiki behar da, aukeratutako ehunekoari dagozkion barne nodo edo maila kopurua jakiteko, hau egiteko superklasearen *buildTree()* metodoa erabiltzen da (ikusi Kodea 24). Berriz ere, bi aukera daude kodearen parte honetan: nodo edo mailak kontsolidatu nahi izatea. Bi aukeren funtzionamendua ia berdina da: batek zuhaitzaren mailak kalkulatzen ditu eta erabiltzaileak

emandako portzentajearekin biderkatzen du; besteak, berriz, zuhaitzaren barne-nodoen kopurua kalkulatzen du eta portzentajearekin biderkatzen du.

```
super.buildTree(data, samplesVector, m_subtreeRaising || !m_cleanup); // build the tree
     without restrictions
if (m_priorityCriteria == J48It.Levelbylevel) {

    // Number of levels of the consolidated tree
    int treeLevels = numLevels();

    // Number of levels of the consolidated tree to leave as consolidated based on
    // given consolidationPercent
    int numberLevelsConso = (int) (((treeLevels * consolidationPercent) / 100) + 0.5);
    m_maximumCriteria = numberLevelsConso;
    System.out.println(
            "Number of levels to leave as consolidated: " + numberLevelsConso + " of "
                + treeLevels);

} else {

    // Number of internal nodes of the consolidated tree
    int innerNodes = numNodes() - numLeaves();

    // Number of nodes of the consolidated tree to leave as consolidated based on
    // given consolidationPercent
    int numberNodesConso = (int) (((innerNodes * consolidationPercent) / 100) + 0.5);
    m_maximumCriteria = numberNodesConso;
    System.out.println(
            "Number of nodes to leave as consolidated: " + numberNodesConso + " of " +
                innerNodes);

}
```

Kodea 24: Ehunekoa aukeratzen bada *buildClassifer*-en kode zatia

Balio bat aukeratu bada, parametroak emandako balioa maximoa izango da zuzenean eta ez da beharrezkoa izango superklaseko buildTree()-ra deitzea zuhaitza osoa eraikitzeko:

```
else // consolidationPercentHowToSet == // J48ItPartiallyConsolidated.
    consolidationPercent_Value
{
    m_maximumCriteria = (int) consolidationPercent;
    System.out.println("Number of nodes or levels to leave as consolidated: " +
        m_maximumCriteria);
}
```

Kodea 25: Zenbakia aukeratzen bada *buildClassifer*-en kode zatia

Amaitzeko, zuhaitza sortzen da lortutako maximoekin, ondoren zuhaitza kolapsatu edota inausi egiten da, eta, bukatzeko, Bagging-a aplikatzen da.

```
buildTree(data, samplesVector, m_subtreeRaising || !m_cleanup);
if (m_collapseTheTree) {
    collapse();
}
if (m_pruneTheTree) {
    prune();
```

```
}
applyBagging();
```

Kodea 26: *buildClassifer*-en kodearen azken zatia

Ondoren, 27. kodean, *buildClassifier()* metodoaren kode osoa ikus daiteke:

```
1 public void buildClassifier(Instances data, Instances[] samplesVector, float
      consolidationPercent, int consolidationNumberHowToSet) throws Exception {
2 if (m_priorityCriteria == J48It.Original) {
3
4     super.buildClassifier(data, samplesVector, consolidationPercent);
5
6 } else {
7
8     System.out.println("it buildClassifer");
9
10    if (consolidationNumberHowToSet == J48ItPartiallyConsolidated.
          ConsolidationNumber_Percentage) {
11
12        super.buildTree(data, samplesVector, m_subtreeRaising || !m_cleanup); // build
              the tree without restrictions
13
14        if (m_collapseTheTree) {
15            collapse();
16        }
17        if (m_pruneTheTree) {
18            prune();
19        }
20
21        if (m_priorityCriteria == J48It.Levelbylevel) {
22
23            // Number of levels of the consolidated tree
24            int treeLevels = numLevels();
25
26            // Number of levels of the consolidated tree to leave as consolidated based
                  on
27            // given consolidationPercent
28            int numberLevelsConso = (int) (((treeLevels * consolidationPercent) / 100)
                  + 0.5);
29            m_maximumCriteria = numberLevelsConso;
30            System.out.println(
31                    "Number of levels to leave as consolidated: " + numberLevelsConso +
                          " of " + treeLevels);
32        } else {
33
34            // Number of internal nodes of the consolidated tree
35            int innerNodes = numNodes() - numLeaves();
36
37            // Number of nodes of the consolidated tree to leave as consolidated based
                  on
38            // given consolidationPercent
39            int numberNodesConso = (int) (((innerNodes * consolidationPercent) / 100) +
                  0.5);
40            m_maximumCriteria = numberNodesConso;
41            System.out.println(
42                    "Number of nodes to leave as consolidated: " + numberNodesConso + "
                          of " + innerNodes);
```

```
43
44          }
45
46      } else // consolidationNumberHowToSet ==
47              // J48ItPartiallyConsolidated.ConsolidationNumber_Value
48      {
49          m_maximumCriteria = (int) consolidationPercent;
50          System.out.println("Number of nodes or levels to leave as consolidated: " +
                m_maximumCriteria);
51      }
52
53      // buildTree
54      buildTree(data, samplesVector, m_subtreeRaising || !m_cleanup);
55      if (m_collapseTheTree) {
56      collapse();
57      }
58      if (m_pruneTheTree) {
59      prune();
60      }
61
62      applyBagging();
63
64      if (m_cleanup)
65          cleanup(new Instances(data, 0));
66  }
67  }
```

Kodea 27: PCTgidatuaren *buildClassifier()* metodo osoaren inplementazioa

Zuhaitza eraikitzeko metodoari dagokionez, PCTBag-aren *buildTree()* erabili da, baina J48Iteratiboan (8.1.1. atala) azaldutako aldaketa berdinekin iteratiboa egiteko eta maximoak gehitzeko.

8.2.2 J48ITPARTIALLYCONSOLIDATEDTREE KLASEA

Sailkatzailearentzako klasea sortu ondoren, WEKAri gehitu behar zaio, programatik erabili ahal izateko. Horretarako, 7.5.2. atalean deskribatutako urratsak jarraitu behar dira. Aldaketa hauek *J48ItPartiallyConsolidated* izeneko klase berri batean doaz, eta *weka.classifiers.trees* paketean joan behar du. Klase honek PCTBagging-aren klasea zabaltzen du, hau da, *J48PartiallyConsolidated* klasea.

Lehenik eta behin, parametro berriak gehitu behar dira, erabiltzaileak GUItik edo komando-lerrotik aukera ditzan. Parametro berriei 'ITPC' (Partially Consolidated Tree iteratiboa) aurrizki gisa jartzea erabaki da. Parametro berriak honako hauek dira:

Izena	Mota	Definizioa	Deskripzioa
PCTBconsolidationPercent	float	-PCTB-C (zbk)	PCTBagging-ean dago ere, eta erabiltzaileak kontsolidatu nahi duen nodo kopurua adierazten du.
ITPCTpriorityCriteria	int	-ITPCT-PO	Eraiki zuhaitza PCTBagging algoritmoa erabiliz
		-ITPCT-PL	Eraiki zuhaitza mailaz maila
		-ITPCT-PP	Eraiki zuhaitza preorder-ean
		-ITPCT-PS	Eraiki tamainaz ordenatua
		-ITPCT-PG	Eraiki irabazi-ratioz ordenatua
		-ITPCT-PGN	Eraiki zuhaitza irabazi-ratio normalizatuaz ordenatua
ITPCTconsolidationPercent-HowToSet	int	-ITPCT-P	Ezarri zenbat nodo edo maila sortuko diren portzentaje gisa
		-ITPCT-V	Ezarri zenbat nodo edo maila sortuko diren balio baten arabera

Taula 4: PCTgidatuaren parametro berriak

Bai *priorityCriteria* bai *consolidationPercentHowToSet* mota osokoak dira, aukera bakoitzari zenbaki batekin lotu baitzaio:

```
/** Ways to set the priority criteria option */
public static final int Original = 0;
public static final int Levelbylevel = 1;
public static final int Preorder = 2;
public static final int Size = 3;
public static final int Gainratio = 4;
public static final int Gainratio_normalized = 5;

/** Ways to set the consolidationPercent option */
public static final int consolidationPercent_Value = 1;
public static final int consolidationPercent_Percentage = 2;
```

Kodea 28: Aukera bakoitzeko zenbakizko balioak

Orain, bai laguntza-metodoak bai metodoak inplementatu behar dira parametroak gehitzeko. Hau da *listOptions()* metodoaren inplemenazioa parametro berrien aukerak terminaletik ikusteko balio duena:

```
1 public Enumeration<Option> listOptions() {
2     Vector<Option> newVector = new Vector<Option>(1);
3
4     newVector.addElement(new Option("\tBuild the tree as it was originally built.", "
          ITPCT-PO", 0, "-ITPCT-PO"));
5     newVector.addElement(new Option("\tBuild the tree level by level.", "ITPCT-PL", 0,
          "-ITPCT-PL"));
6     newVector.addElement(new Option("\tBuild the tree in preorder.", "ITPCT-PP", 0, "-
          ITPCT-PP"));
7     newVector.addElement(new Option("\tBuild the tree ordered by size.", "ITPCT-PS", 0,
           "-ITPCT-PS"));
8     newVector.addElement(new Option("\tBuild the tree ordered by gainratio.", "ITPCT-PG
          ", 0, "-ITPCT-PG"));
9     newVector.addElement(new Option("\tBuild the tree ordered by normalized gainratio."
          , "ITPCT-PGN", 0, "-ITPCT-PGN"));
10    newVector.addElement(new Option("\tSet the number of nodes or levels to be
          generated based on a value\\n\" +\n" + "                          \"\\tas a
          percentage (by default)", "ITPCT-P", 0, "-ITPCT-P"));
11    newVector.addElement(new Option("\tSet the number of nodes or levels to be
          generated based on a numeric value", "ITPCT-V", 0, "-ITPCT-V"));
12
13    newVector.addAll(Collections.list(super.listOptions()));
14    return newVector.elements();
15 }
```

Kodea 29: PCTgidatuaren *listOptions()* metodoa

setOptions() metodoak aukera bakoitzaren konfigurazioak ezartzen ditu, erabiltzaileak aukeratutakoaren arabera:

```
1 public void setOptions(String[] options) throws Exception {
2
3     if (Utils.getFlag("ITPCT-P", options))
4         setITPCTconsolidationPercentHowToSet(new SelectedTag(
              consolidationPercent_Percentage, TAGS_WAYS_TO_SET_CONSOLIDATION_NUMBER));
5     else if (Utils.getFlag("ITPCT-V", options))
6         setITPCTconsolidationPercentHowToSet(new SelectedTag(consolidationPercent_Value
              , TAGS_WAYS_TO_SET_CONSOLIDATION_NUMBER));
7
8
9     if (Utils.getFlag("ITPCT-PO", options))
10        setITPCTpriorityCriteria(new SelectedTag(Original,
              TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
11    else if (Utils.getFlag("ITPCT-PL", options))
12        setITPCTpriorityCriteria(new SelectedTag(Levelbylevel,
              TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
13    else if (Utils.getFlag("ITPCT-PP", options))
14        setITPCTpriorityCriteria(new SelectedTag(Preorder,
              TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
15    else if (Utils.getFlag("ITPCT-PS", options))
16        setITPCTpriorityCriteria(new SelectedTag(Size,
              TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
17    else if (Utils.getFlag("ITPCT-PG", options))
```

```
18          setITPCTpriorityCriteria(new SelectedTag(Gainratio,
                TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
19      else if (Utils.getFlag("ITPCT-PGN", options))
20          setITPCTpriorityCriteria(new SelectedTag(Gainratio_normalized,
                TAGS_WAYS_TO_SET_PRIORITY_CRITERIA));
21
22      super.setOptions(options);
23  }
```

Kodea 30: PCTgidatuaren *setOptions()* metodoa

getOptions() metodoak *string* bat itzultzen du erabiltzaileak aukeratutako uneko konfigurazioarekin:

```
1  public String[] getOptions() {
2
3      Vector<String> options = new Vector<String>();
4      Collections.addAll(options, super.getOptions());
5
6      if (m_ITPCTpriorityCriteria == 0) options.add("-ITPCT-PO");
7      else if (m_ITPCTpriorityCriteria == 1) options.add("-ITPCT-PL");
8      else if (m_ITPCTpriorityCriteria == 2) options.add("-ITPCT-PP");
9      else if (m_ITPCTpriorityCriteria == 3) options.add("-ITPCT-PS");
10     else if (m_ITPCTpriorityCriteria == 4) options.add("-ITPCT-PG");
11     else if (m_ITPCTpriorityCriteria == 5) options.add("-ITPCT-PGN");
12
13     if (m_ITPCTconsolidationPercentHowToSet == consolidationPercent_Value) options.add(
           "-ITPCT-V");
14     else options.add("-ITPCT-P");
15
16
17     return options.toArray(new String[0]);
18 }
```

Kodea 31: PCTgidatuaren *getOptions()* metodoa

Kontsolidazio-ehunekorako laguntza-testua berdefinitu da, orain ehuneko bat edo zenbaki bat izan baitaiteke:

```
1 public String PCTBconsolidationPercentTipText() {
2     return "Consolidation percent or number for use after the consolidation process";
3 }
```

Kodea 32: PCTgidatuaren *PCTBconsolidationPercentTipText()* metodoaren berdefnizioa

priorityCriteria parametro berriari dagokionez, honako hauek dira behar dituen metodo berriak. Honela geratuko litzateke laguntza testua:

```
1 public String ITPCTpriorityCriteriaTipText() {
2     return "Build the tree ordered by a criteria: Original (recursive), LevelByLevel,
          Preorder, Size, Gainratio, Normalized Gainratio";
3 }
```

Kodea 33: J48ItPartiallyConsolidated klasearen *ITPCTpriorityCriteriaTipText()* metodoa

Erabiltzaileak zabalgarrian aukeratutako balioa lortzeko metodoa:

```
1 public SelectedTag getITPCTpriorityCriteria() {
2     return new SelectedTag(m_ITPCTpriorityCriteria,
3             TAGS_WAYS_TO_SET_PRIORITY_CRITERIA);
4 }
```

Kodea 34: J48ItPartiallyConsolidated klasearen *getITPCTpriorityCriteria()* metodoa

Erabiltzaileak aukeratutako balioa ezartzeko metodoak egiaztapen gehigarri bat du, erabiltzaileak bere konfigurazioa aukeratzeko duen aukeretan akatsik dagoen ikusteko.

```
1 public void setITPCTpriorityCriteria(SelectedTag newPriorityCriteria) throws Exception
      {
2     if (newPriorityCriteria.getTags() == TAGS_WAYS_TO_SET_PRIORITY_CRITERIA)
3     {
4         int newPriority = newPriorityCriteria.getSelectedTag().getID();
5
6         if (newPriority == Original || newPriority == Levelbylevel || newPriority ==
              Preorder || newPriority == Size || newPriority == Gainratio || newPriority
              == Gainratio_normalized)
7             m_ITPCTpriorityCriteria = newPriority;
8         else
9             throw new IllegalArgumentException("Wrong selection type, value should be:
                  "
10                    + "between 0 and 5");
11    }
12 }
```

Kodea 35: J48ItPartiallyConsolidated klasearen *setITPCTpriorityCriteria()* metodoa

consolidationPercentHowToSetTipText delakoari dagokionez, aurrekoaren oso antzekoa da. Zabalgarri batean bi aukera ere badaude: edo ehunekoa edo zenbakia. Parametroa deskribatzeko metodoa hau izango litzateke metodoaren deskribapenarekin:

```
1 public String ITPCTconsolidationPercentHowToSetTipText() {
2     return "Way to set the consolidation number to be generated:\n" +
3             " * using a fixed value which directly indicates the number nodes to
                  consolidate\n" + " * based on a value as a percentage (by default)\n";
4 }
```

Kodea 36: J48ItPartiallyConsolidated klasearen *ITPCTconsolidationPercentHowToSetTipText()* metodoa

Erabiltzaileak aukeratzen duena lortzeko metodoa:

```
1 public SelectedTag getITPCTconsolidationPercentHowToSet() {
2     return new SelectedTag(m_ITPCTconsolidationPercentHowToSet,
          TAGS_WAYS_TO_SET_CONSOLIDATION_NUMBER);
3 }
```

Kodea 37: J48ItPartiallyConsolidated klasearen *getITPCTconsolidationPercentHowToSet()* metodoa

priorityCriteria-rekin bezala, erabiltzaileak zer erabaki duen zehazteko, aldez aurreko egiaztapen bat egiten da eta akats bat gertatzen bada errore bat aterako da:

```
1 public void setITPCTconsolidationPercentHowToSet(SelectedTag
      newWayToSetconsolidationPercent) throws Exception {
2     if (newWayToSetconsolidationPercent.getTags() ==
          TAGS_WAYS_TO_SET_CONSOLIDATION_NUMBER)
3     {
4         int newEvWay = newWayToSetconsolidationPercent.getSelectedTag().getID();
5
6         if (newEvWay == consolidationPercent_Percentage || newEvWay ==
              consolidationPercent_Value)
7             m_ITPCTconsolidationPercentHowToSet = newEvWay;
8         else
9             throw new IllegalArgumentException("Wrong selection type, value should be:
                  " + "between 1 and 2");
10    }
11 }
```

Kodea 38: J48ItPartiallyConsolidated klasearen *setITPCTconsolidationPercentHowToSet()* metodoa

Inplementatu beharreko metodo garrantzitsuenetako bat *buildClassifier()* da, sailkatzailea sortzeko aukera ematen duena, ez nahastu zuhaitz kontsolidatua eraikiko duen sailkatzailearen klaseko *buildClassifier()*-arekin. Metodo hau superklasearen oso antzekoa da, baina aldaketa batekin: sortu nahi den sailkatzaile mota aldatu behar da; kasu honetan *C45ItPartiallyConsolidatedPruneableClassifierTree* motakoa dela adierazi behar da, sailkatzaile berriaren eraiketa bertan baitago (39. kodearen 26. lerroa). Eraikitzailean parametro gisa gehitu behar da erabiltzaileak zuhaitz kontsolidatua sortzeko aukeratu duen irizpidea.

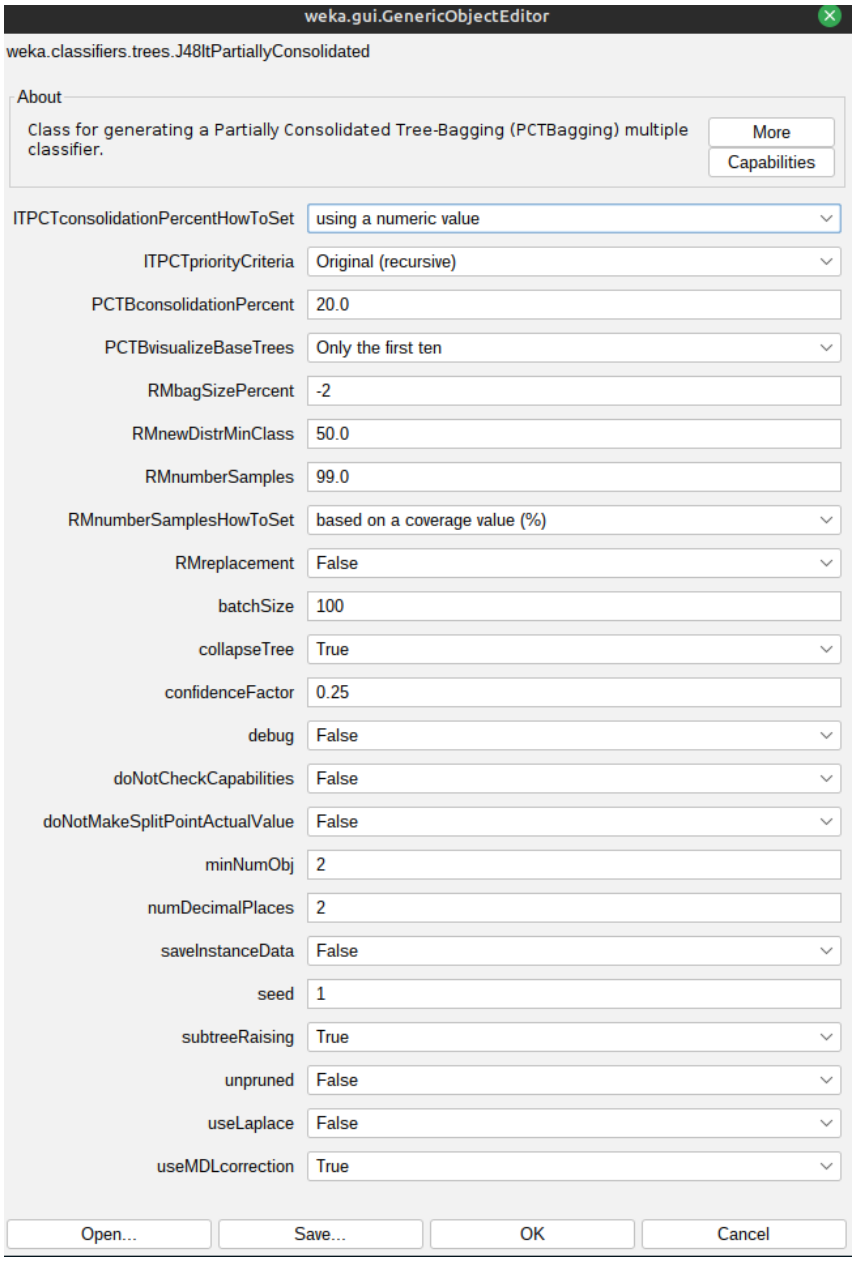
Gainera, sailkatzailearen *buildClassifier()*-era deitzerakoan, ez dira soilik instantziak eta laginak parametrotzat eman behar, erabiltzaileak aukeratutako kontsolidazio-ehunekoa ere pasatu behar zaio (hau *J48PartiallyConsolidated*-en bezala) eta ehuneko bat edo zenbaki bat erabili nahi den esaten duen parametroa ere.

```
1 /**
2  * Generates the classifier.
3  * (based on buildClassifier() function of J48Consolidated class and of
       J48PartiallyConsolidated class)
4  *
5  */
6 public void buildClassifier(Instances instances)
7         throws Exception {
8
9     // can classifier tree handle the data?
10    getCapabilities().testWithFail(instances);
11
12    // remove instances with missing class before generate samples
13    instances = new Instances(instances);
14    instances.deleteWithMissingClass();
15
```

```
16      //Generate as many samples as the number of samples with the given instances
17      Instances[] samplesVector = generateSamples(instances);
18
19      /** Set the model selection method to determine the consolidated decisions */
20      ModelSelection modSelection;
21      modSelection = new C45ConsolidatedModelSelection(m_minNumObj, instances,
            m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
22      /** Set the model selection method to force the consolidated decision in each base
            tree*/
23      C45ModelSelectionExtended baseModelToForceDecision = new C45ModelSelectionExtended(
            m_minNumObj, instances, m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
24
25      C45ItPartiallyConsolidatedPruneableClassifierTree localClassifier =
26              new C45ItPartiallyConsolidatedPruneableClassifierTree(modSelection,
                    baseModelToForceDecision, !m_unpruned, m_CF, m_subtreeRaising, !
                    m_noCleanup, m_collapseTree, samplesVector.length,
                    m_ITPCTpriorityCriteria);
27
28      localClassifier.buildClassifier(instances, samplesVector,
            m_PCTBconsolidationPercent, m_ITPCTconsolidationPercentHowToSet);
29
30      m_root = localClassifier;
31      m_Classifiers = localClassifier.getSampleTreeVector();
32
33      ((C45ModelSelection) modSelection).cleanup();
34      ((C45ModelSelection) baseModelToForceDecision).cleanup();
35  }
```

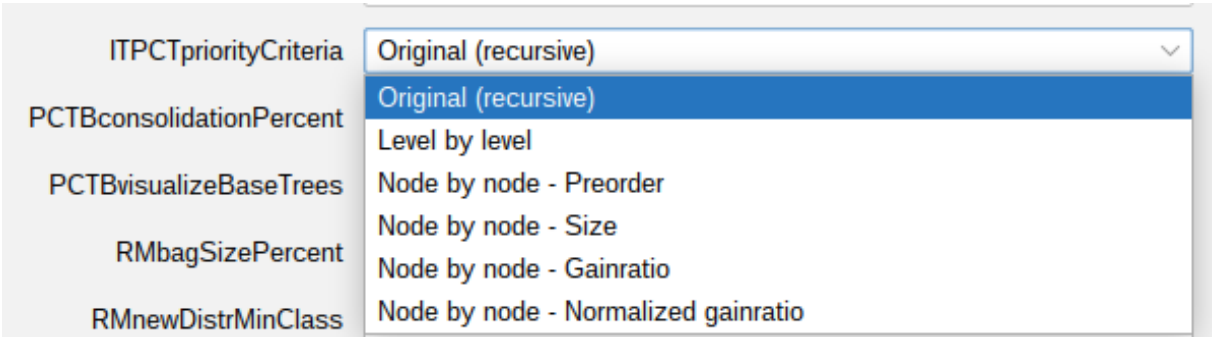
Kodea 39: J48ItPartiallyConsolidated klasearen *buildClassifier()* metodoa

WEKA exekutatuz gero, PCTBagging algoritmo gidatua interfazetik horrela geratuko litzateke.

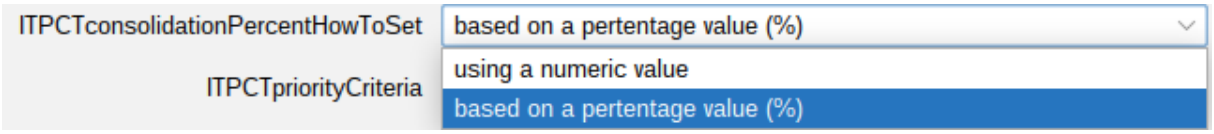


Irudia 27: PCTgidatuaren parametro (*options*) interfazea

Eta hemen daude bi parametro berrien hedagarriaren aukerak, erabiltzaileak errazago izan dezan aukeran dauden artean aukeratzea.



Irudia 28: priorityCriteria parametroaren hedagarria



Irudia 29: consolidationPercentHowToSet parametroaren hedagarria

Inplementazioaren kodea Github-eko biltegi honetan dago: `https://github.com/JosueCab/wekatfg`

9 ESPERIMENTAZIOA

PCTBagging gidatua algoritmoa WEKAn inplementatu ondoren, esperimentaziora pasatu behar da, emaitzak eta algoritmoaren errendimendua aztertzeko hainbat datu-multzorekin.

Ezaugarri desberdinak dituzten 33 datu-base erabili dira kasu guztietan funtzionatzen duela ikusteko. Datu-base horiek dira, adibidez, ALDAPA taldeak PCTBagging algoritmoaren esperimentazioan [2] erabili zituen kategorietako batekoak, datu-multzo desorekatuena. Esperimentaziorako, *5-fold Cross-Validation* 10 aldiz egin da ebaluazio-irizpideen zenbatespenak kalkulatzeko. Prozesu hau 10 aldiz errepikatzeak batez besteko errendimenduaren eta algoritmoaren aldakortasunaren batez besteko zehatzagoak lortzea ahalbidetzen du, datuen partizio desberdinekin.

PCTgidatuaren irizpideak hurbilen dituen algoritmoekin alderatu ditugu. Grafikoen ondoan legenda bat ageri da lerro bakoitzak aipatzen duen algoritmoa adieraziz.

Konparazioak egiteko erabili diren algoritmoak honako hauek dira:

- C4.5
- CTC lehenetsitako balioekin: azpilagin orekatuak = %99
- Bagging C4.5 oinarrizko algoritmo gisarekin
- PCTBagging-a
- PCTgidatua, 'LevelByLevel' lehentasun-irizpidearekin
- PCTgidatua, 'Preorder' lehentasun-irizpidearekin
- PCTgidatua, 'Size' lehentasun-irizpidearekin
- PCTgidatua, 'Gainratio' lehentasun-irizpidearekin
- PCTgidatua, 'Gainratio_Normalized' lehentasun-irizpidearekin

PCTBagging algoritmoan eta PCTgidatua algoritmo guztietan esperimentazioa egin da hurrengo kontsolidazio portzentajeekin = **%0, %10, %20, %30, %40, %50, %60, %70, %80, %90 eta %100.**

Eta ebaluazio-irizpide hauek aukeratu dira:

- Sailkatzeko gaitasuna duten irizpideak:
  - Accuracy

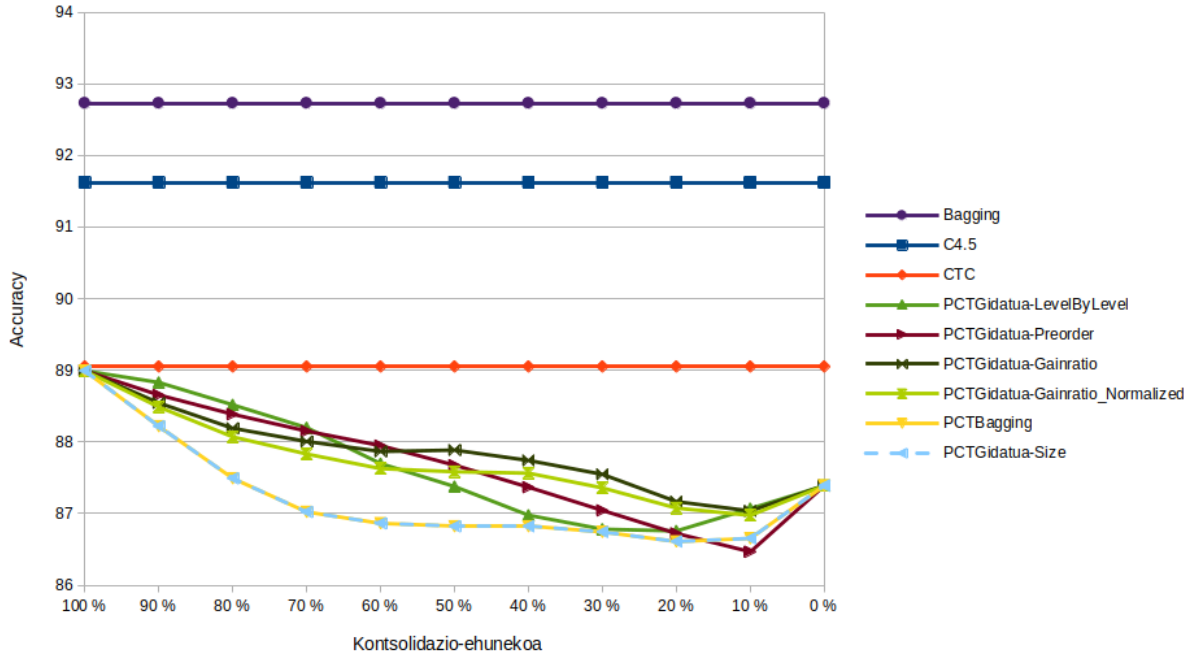
  - Balanced Accuracy
  - AUC
  - Kappa
- Azaltzeko gaitasuna duten irizpideak:
  - Nodo kopurua
  - Hosto kopurua
- Kostu konputazionalaren irizpideak:
  - Elapsed_Time_Training

Azalpenak arintzeko, laburdurak erabiliko ditugu termino jakin batzuk aipatzeko. Adibidez, PCTGidatua-Size laburtua izango da PCT-Size bezala, PCTGidatua-LevelByLevel PCT-LevelByLevel bezala, eta horrela hurrenez hurren.

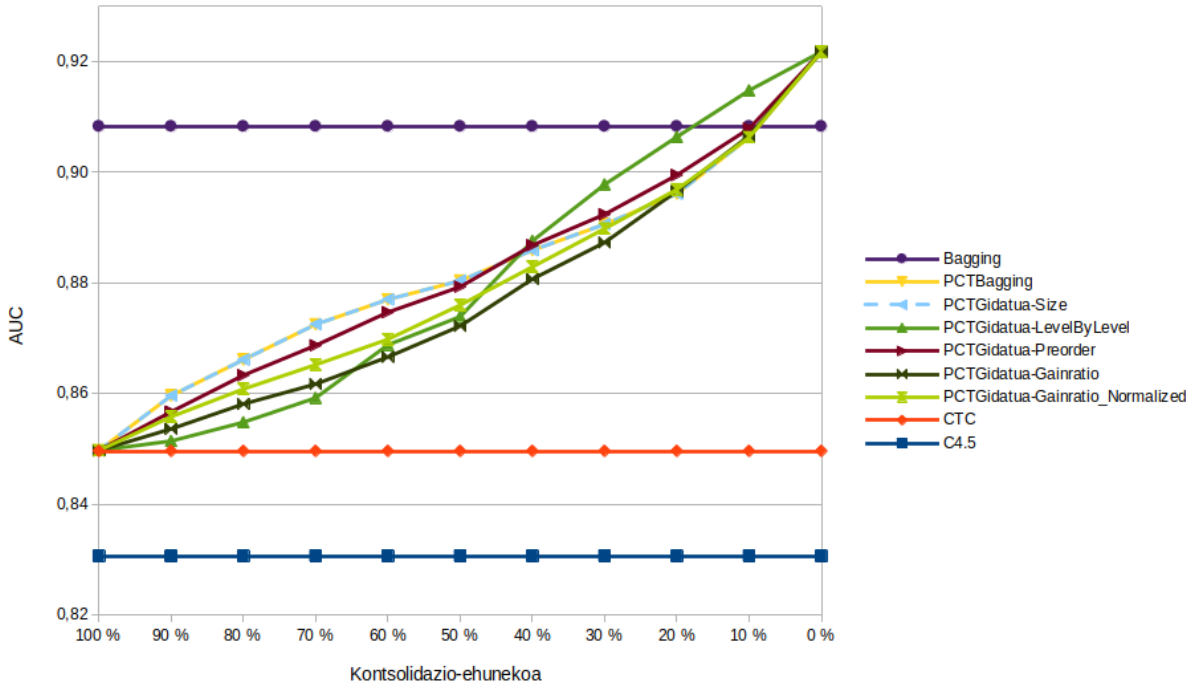
Esperimentazioa azaltzen hasi aurretik, kontuan izan behar da kasu guztietan zuhaitz kolapsatuak eta inausiak erabili direla (*collapsed* eta *pruned*). Horrek eragina izango du emaitzetan; algoritmoaren inplementazioan (8. atala) azaldu den bezala, PCTgidatuak zuhaitz kontsolidatua osorik eraikitzen du, kolapsatu edo inausi egiten du beharrezkoa bada (gure esperimentazioan hala da), eta, ondoren, nodo (edo maila) kopuru totala kalkulatzen du, gero zuhaitz kontsolidatua erabiltzaileak jarritako mugaraino berriro eraikitzen du eta, amaitzeko, kolapsatzen eta inausten du. PCTBagging-en kasuan, zuhaitz kontsolidatua eraiki eta kolapsatu edo inausi egiten da, eta, ondoren, erabiltzaileak aukeratutako nodo-kopurua egoteko beharrezkoak diren nodoak ezabatzen dira.

Kolapsatzeko edo inausteko aukera aukeratzen ez bada, PCTBagging-aren eta PCT-Size-ren emaitzak BERDIN-BERDINAK izan beharko lirateke kasu guztietan (kostu konputazionalaren kasuan izan ezik). Hau logikoa da, izan ere, PCTBagging-ek tamaina handiagoko nodoak uzten ditu eta PCT-Size-ek tamaina handiagoko nodoak garatzen ditu, eta, beraz, normala da emaitzak berdinak izatea.

Hori dela eta, eta esperimentazioan sakonki sartu aurretik, 2 adibide jarri ditugu hori frogatzeko, zuhaitz kontsolidatua kolapsatu edo inausi ezean PCTBagging-ak eta PCT-Size-k emaitza berberak ematen dituztela.



Irudia 30: Accuracy grafikoa unpruned



Irudia 31: AUC grafikoa unpruned

Ez gara grafiko hauen azalpenean sartuko, aurrerago azalduko baitugu *pruned* zuhaitzen bidez. 30. eta 31. grafikoetan, PCTBagging-eko eta PCTSize-ko lerroek (horia eta urdin argia, hurrenez hurren) ibilbide bera jarraitzen dute, hau da, emaitzak BERDIN-BERDINAK dira.

Esperimentazio honetarako, *pruned* zuhaitzak erabiltzea erabaki da, hau C4.5 algoritmoaren aukera lehenetsia baita. Algoritmo desberdinekin egindako esperimentuetan ikusi da *pruned*-etan emaitza hobeak lortzen direla normalean. Gainera, zuhaitz txikiagoak ateratzen dira, eta, beraz, ulergarriagoak, eta horrela sailkatzaileen azalpena sinpleagoa da.

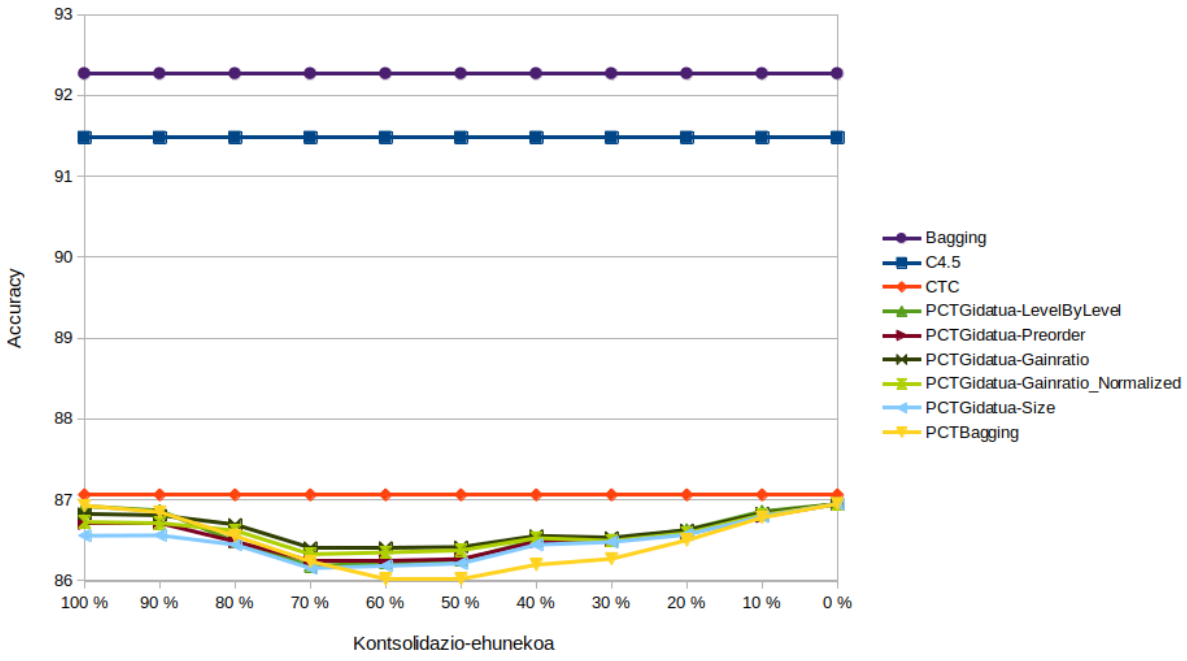
9.1 SAILKATZEKO GAITASUNA

Atal honetan, sailkatzeko gaitasuna duten neurriekin esperimentatuko da.

9.1.1 ACCURACY

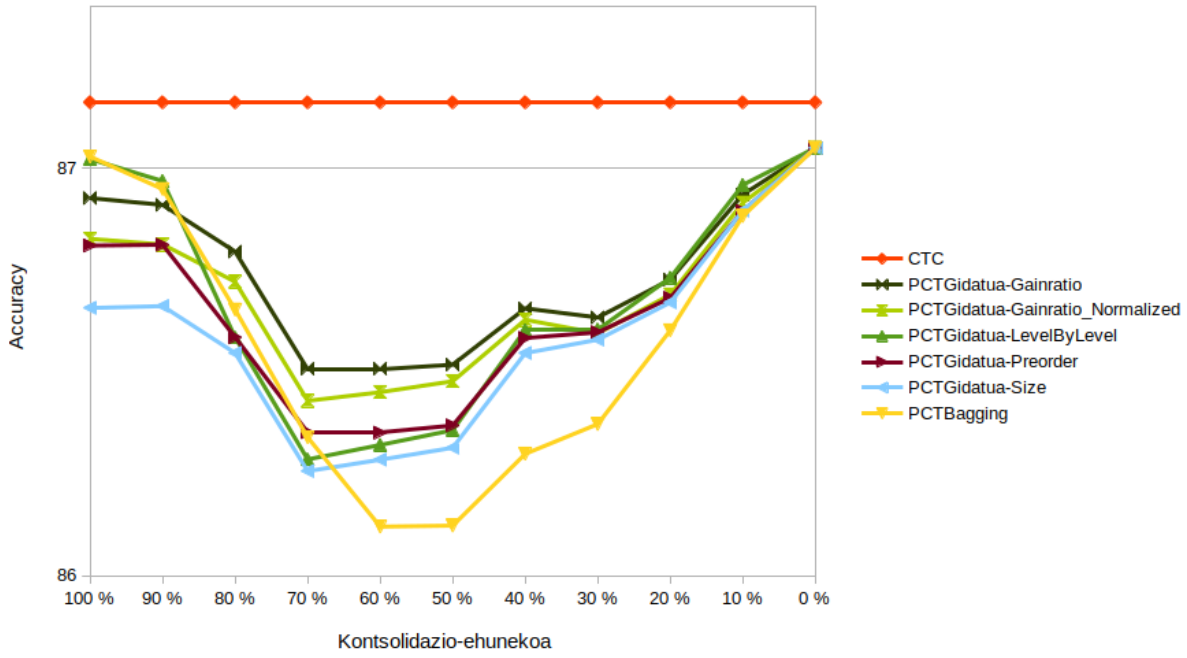
32. irudian ikus daitekeenez, Y ardatzean une bakoitzeko algoritmoen zehaztasun (*accuracy*) balioak agertzen dira, eta X ardatzean, berriz, kontsolidazio-ehunekoa. C4.5, CTC eta Bagging-en kasuan, beti da balio bera, ez baitago kontsolidazio parametrorik algoritmo horietarako.

Balioei dagokienez, ikus daiteke PCT algoritmoek datu-base horiekin ez dituztela CTC, Bagging eta C4.5 gainditzen. Izan ere, Bagging-ak Boostrap laginak erabiltzen ditu, eta gure PCTBag algoritmoa, PCTgidatua eta CTC-a azpilagin orekatu oso txikiak erabiltzen ari dira. Azpi-lagin horiek, bereiz ikusten baditugu, oso informazio gutxi ematen digute, eta arazoak sortzen dira sailkatzeko orduan. Beraz, datu-base horiek ebaluatzeko, hobe da bi klasetarako asmatze-tasa erabiltzen dituzten metrikak erabiltzea (AUC edo Balanced Accuracy, adibidez).



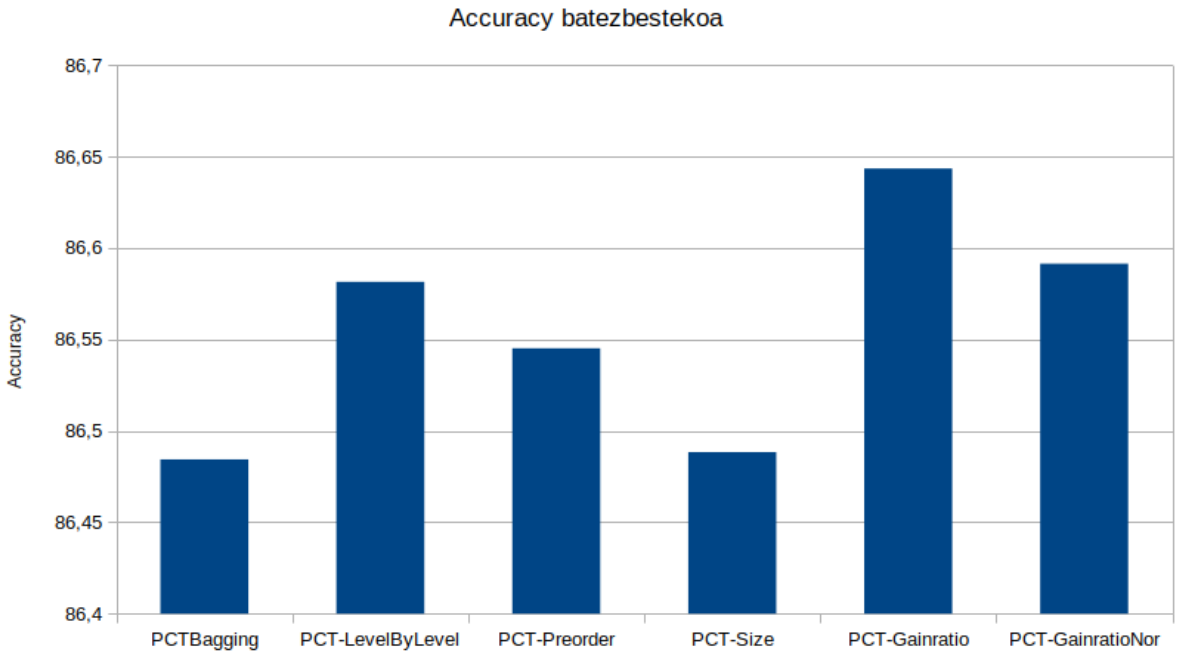
Irudia 32: Accuracy grafikoa

Orain zooma egingo dugu aurreko grafikoan, PCTBagging-arekiko desberdintasunak ikusteko. 33. irudiari erreparatzen badiogu, hainbat gauza ikus daitezke. Hasteko, lehen esan dugun bezala, emaitza guztiak CTC-koak baino okerragoak dira. PCTgidatuaren aldagaien emaitzak trukatzen dira exekuzio osoan zehar, baina ikus daiteke ia denbora guztian emaitza onenak ematen dituena gainratioa dela.



Irudia 33: Accuracy grafikoa hurbilduta

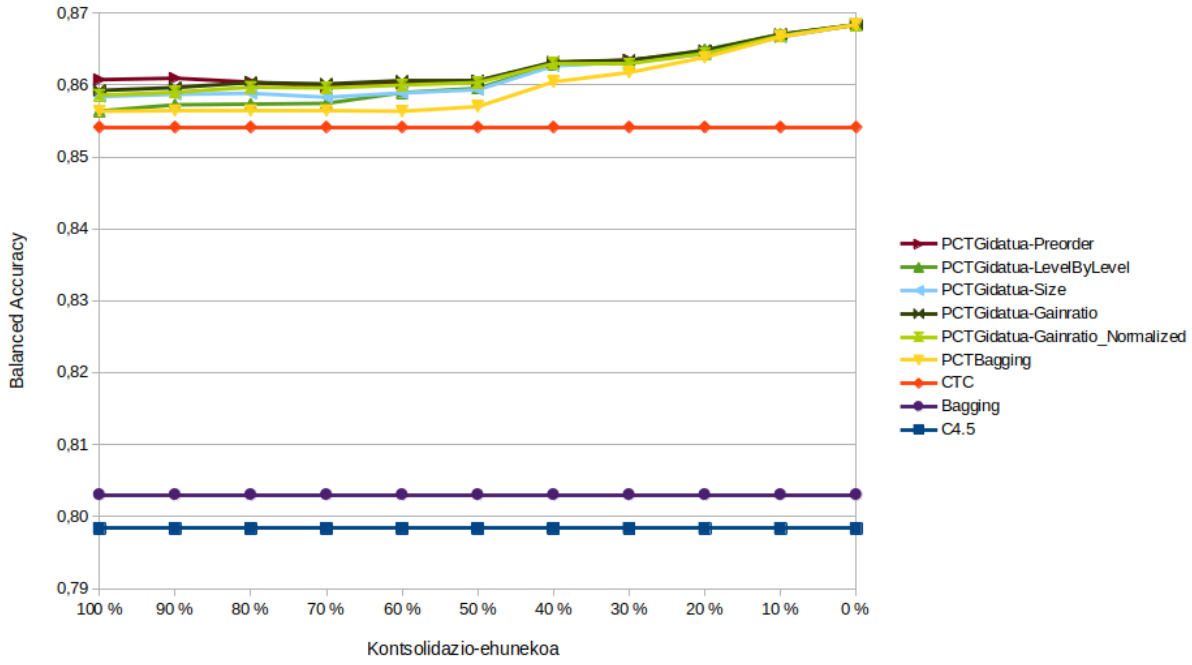
Grafikoan gauza batzuk argi ikusten dira. Lehenik eta behin, PCTBagging-a eta PCT-Size beti emaitza txarrenak ematen dituztenak direla; beraz, esan liteke kasu honetarako egokitzapen berriak (PCTgidatua) emaitzak hobetzen dituela. Esperimentazio osoaren emaitzen batezbestekoa egiten badugu, hau da, aldaera bakoitzaren kontsolidazio-portzentaje guztien batezbestekoa egiten badugu (%0, %10, %20, ... %100), 34. irudian ikusten den bezala, onena PCT-Gainratio da. Hau gainratioak ezaugarri garrantzitsuenak hautatzeko duen gaitasunaren ondorio izan daiteke.



Irudia 34: Accuracy batezbesteko grafikoa

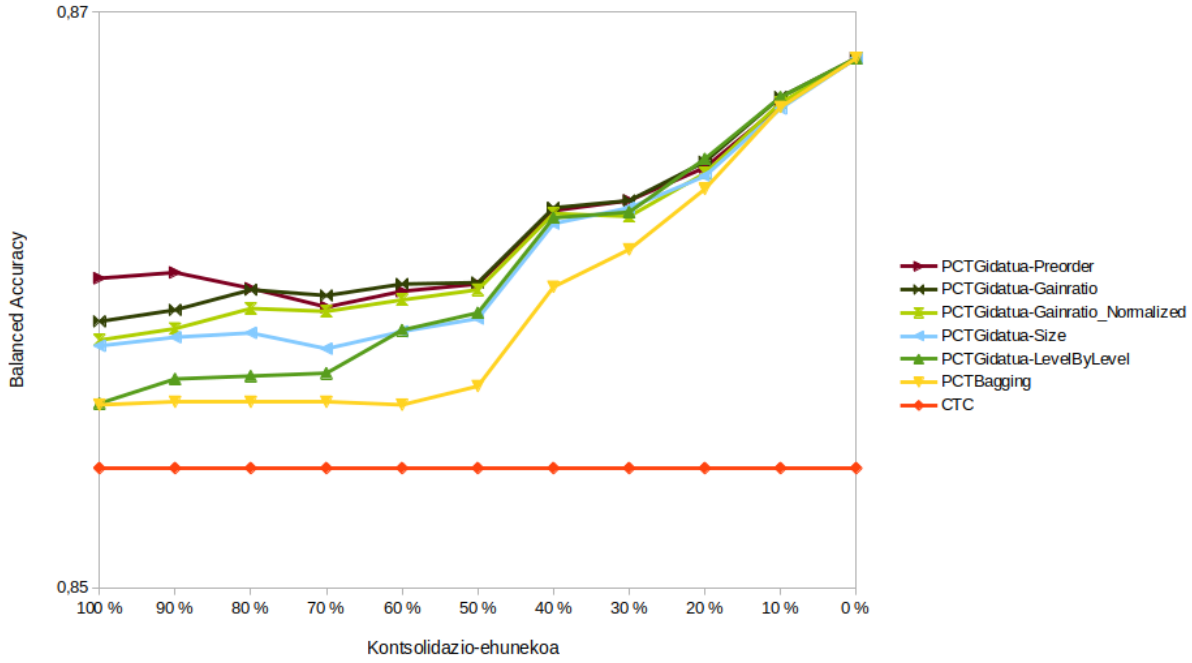
9.1.2 BALANCED ACCURACY

Balanced accuracy-ari dagokionez (zehaztapen orekatua), lehen azaldu dugun bezala *accuracy*-arekin, kasu honetan emaitzak askoz hobeak dira, klase positiboak zein negatiboak kontuan hartzen baitira. 35. irudian ikus daitekeenez, PCT guztien emaitzak CTC, C4.5 eta Bagging-enak baino askoz hobeak dira.



Irudia 35: Balanced Accuracy grafikoa

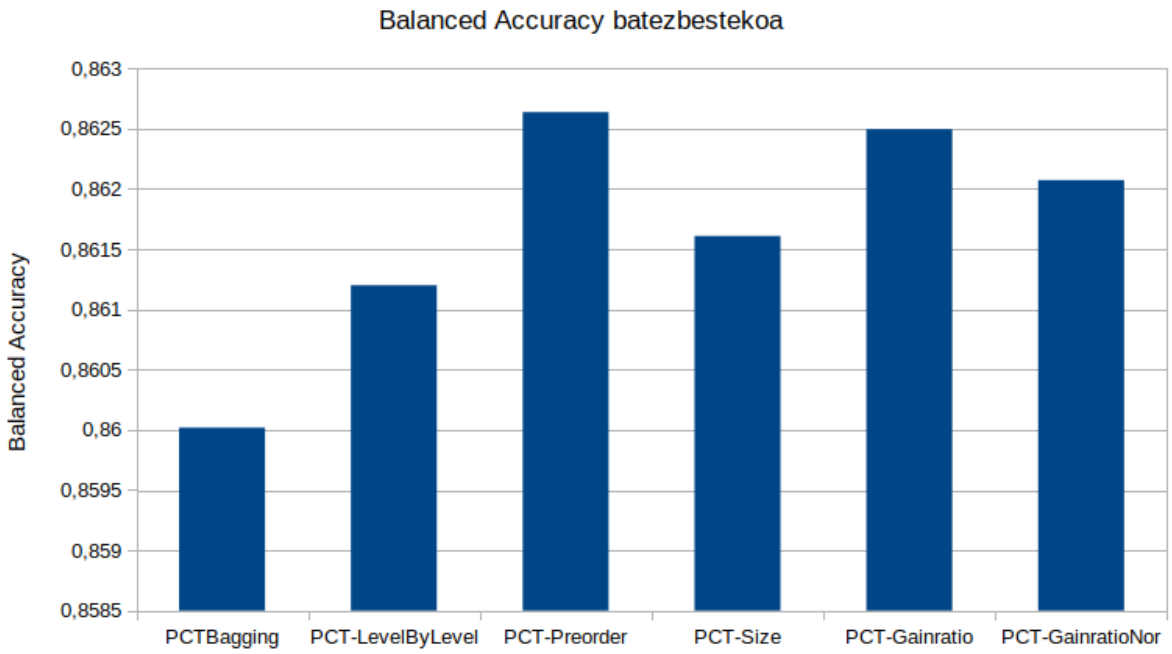
Grafikora hurbilduz gero (36. irudia), PCT-Preorder da denbora gehienez lehenengoa dagoena, eta, bigarrenik, PCT-Gainratio. Emaitza txarrenak ematen dituztenak (edo hori dirudi) PCTBagging eta PCT-LevelByLevel dira; beraz, esan daiteke kasu honetan ere PCTBagging algoritmoa hobetzen dela.



Irudia 36: Balanced Accuracy hurbilduta

Ikusten den moduan, zenbat eta kontsolidazio-ehunekoa txikiagoa izan, orduan eta emaitzak hobeak ematen ditu eta beraien artean dagoen aldea txikiagoa da.

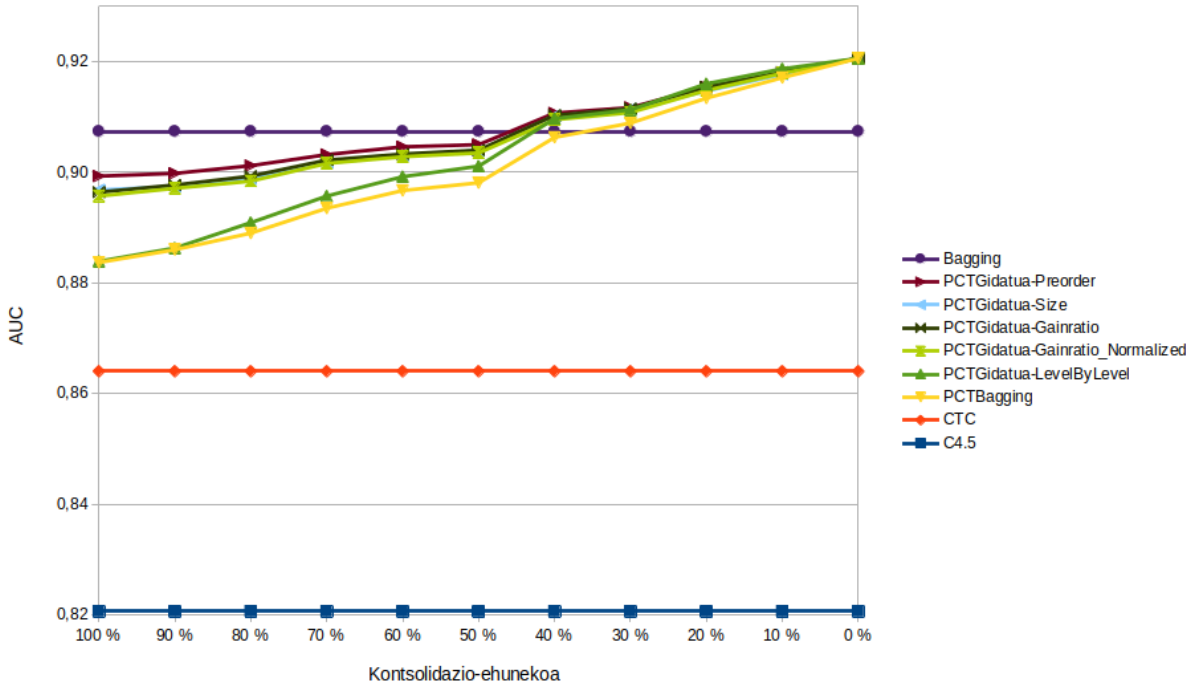
Batezbestekoa egiten badugu *accuracy*-rekin egin den bezala, 37. irudian, aurreko grafikoan ikusten den moduan, preorder da emaitzarik onena ematen duena, gainratiotik gertu. Hori hainbat faktoreren ondorio izan daiteke. Lehenik eta behin, argi dago preorder datuen egitura modu eraginkorragoan aprobetxatzen ari dela, hau da, informazio garrantzitsuena garatzen diren lehen nodoetan (baina ez handienetan) dagoela aprobetxatzen ari du algoritmoak. Gainratioaren kasuan, accuracy-rekin gertatzen zen moduan, ezaugarri garrantzitsuenak hautatzeko duen gaitasunaren ondorio da.



Irudia 37: Balanced Accuracy batezbesteko grafikoa

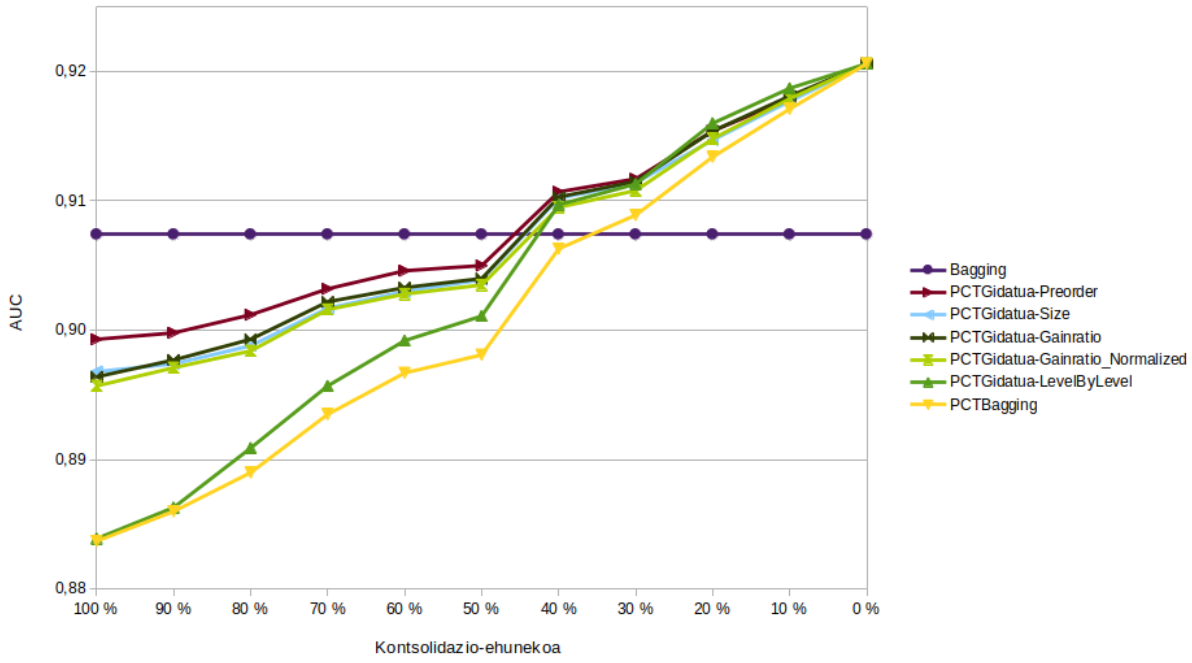
9.1.3 AUC

AUCren kasuan (*Area Under Curve*) emaitzak nahiko antzekoak dira 38. irudian ikus daitekeen bezala. PCT algoritmo guztien balio gehienak CTC-ren eta Bagging-aren artean daude, eta Bagging-a gainditzen dute %40 inguruko (edo baxuago) kontsolidazio-ehunekoa dagoenean.



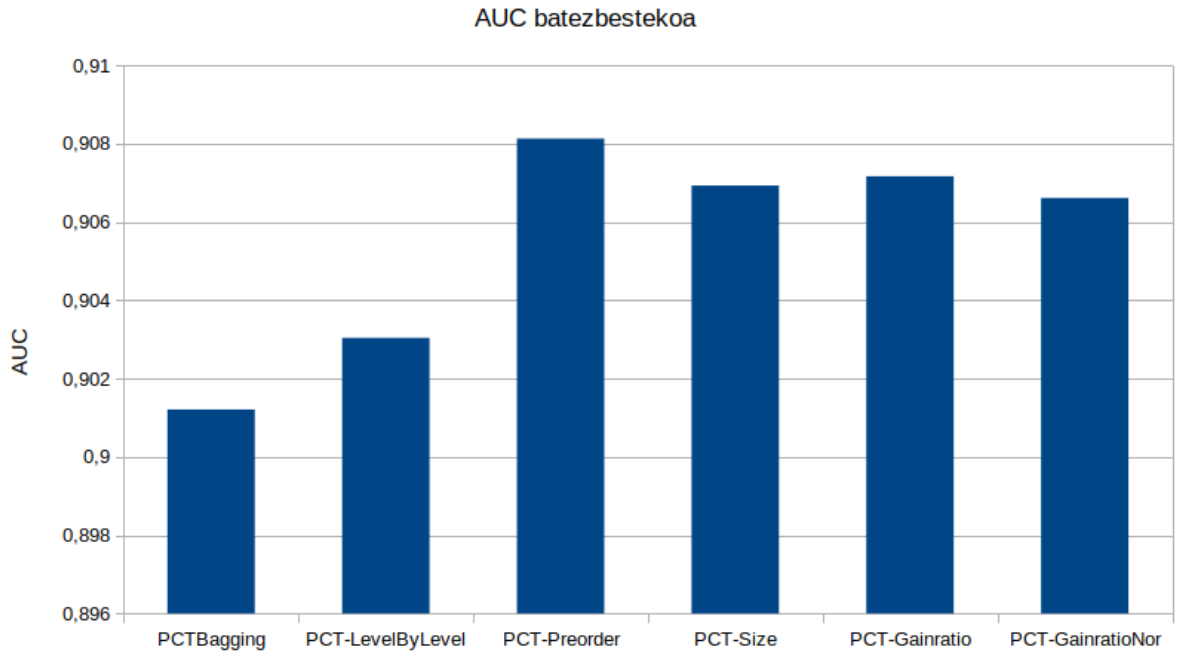
Irudia 38: AUC grafikoa

Grafikoari hurbiltzen bagara, 39. irudian ikusten den moduan, kontsolidazio-ehuneko handiagoarekin (%100etik %45era) preorder araberako ordenamendua eraginkorragoa izan daiteke. Bestalde, %45etik %0ra, kontsolidatutako informazio kopurua txikiagoa denez, badirudi eraginkorragoa dela modu sekuentzialagoan kontsolidatzea, hau da, mailaz maila, baina diferentzia txikia da, izan ere, ez da ia ikusten.



Irudia 39: AUC grafikoa hurbilduta

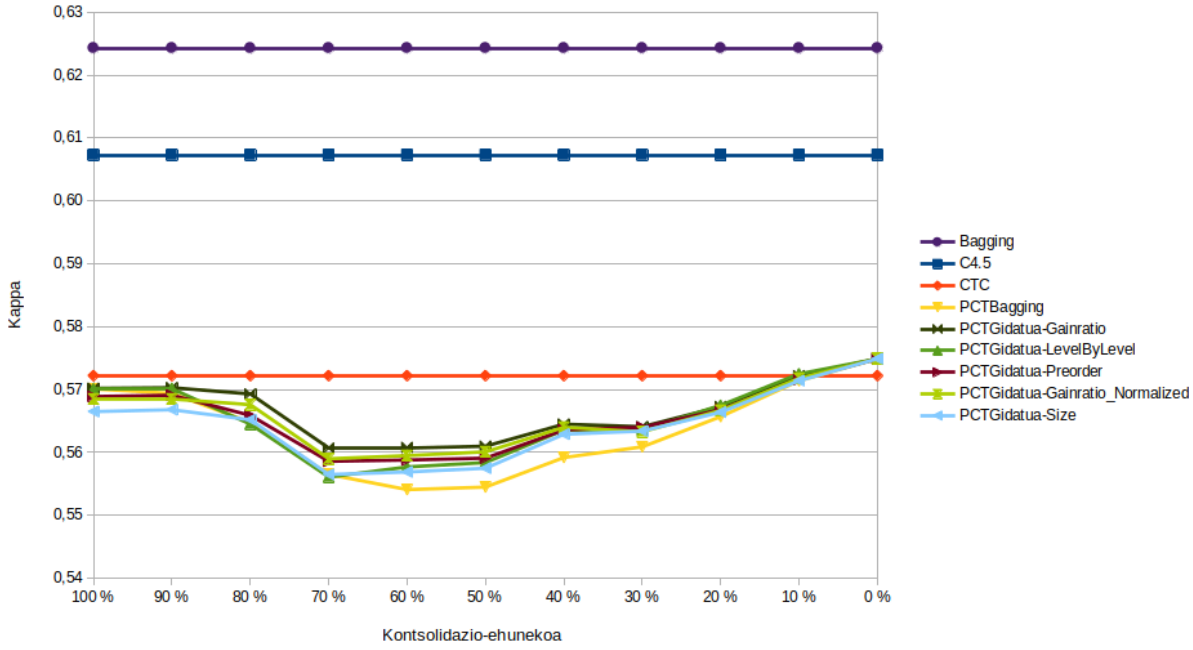
Batezbestekoa berriz egiten badugu, 40. irudian ikusiko dugu kasu honetan emaitzarik onenak ematen dituena PCT-Preorder dela. Beste neurriekin gertatu den bezala, okerrenak PCTBagging eta PCT-LevelByLevel dira. Beraz, kasu honetan egokitzapenek jatorrizko algoritmoa gainditzen dute ere.



Irudia 40: AUC batezbesteko grafikoa

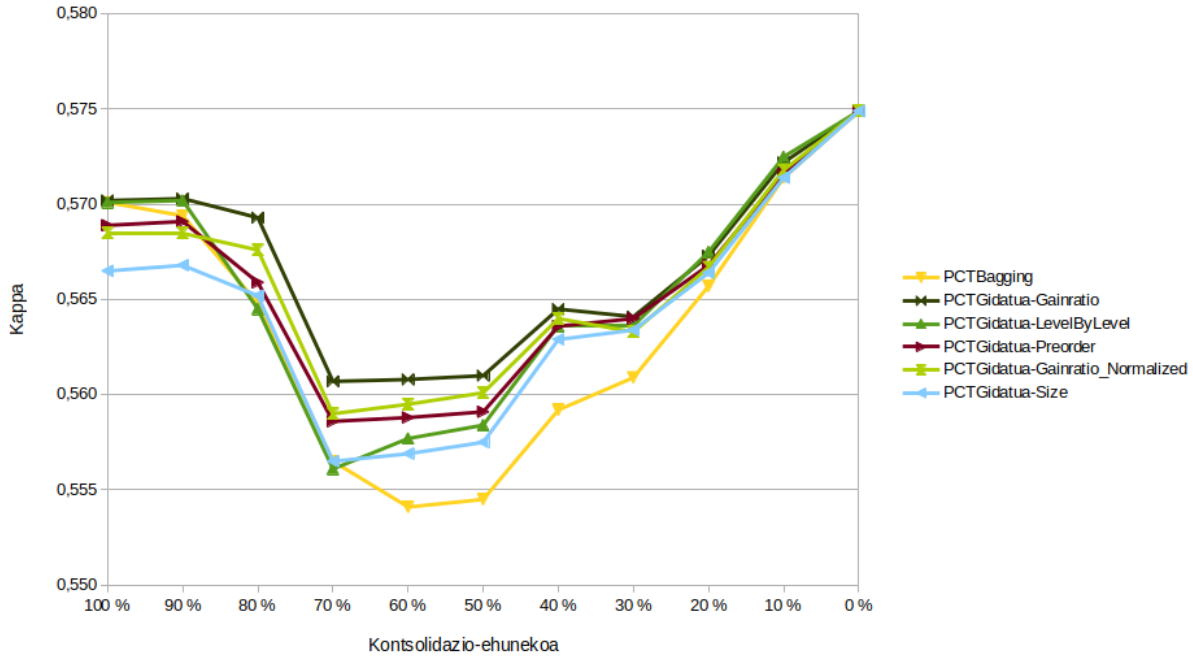
9.1.4 KAPPA

Kapparen kasuan, emaitzak asmatze-tasaren (*Accuracy*) oso antzekoak dira, 41. irudian agertzen den moduan. *Accuracy*-n bezala, kasu honetan PCT algoritmo guztiak dira emaitza txarrenak ematen dituztenak, azpilagin orekatu oso txikiak erabiltzen ari ditugulako.



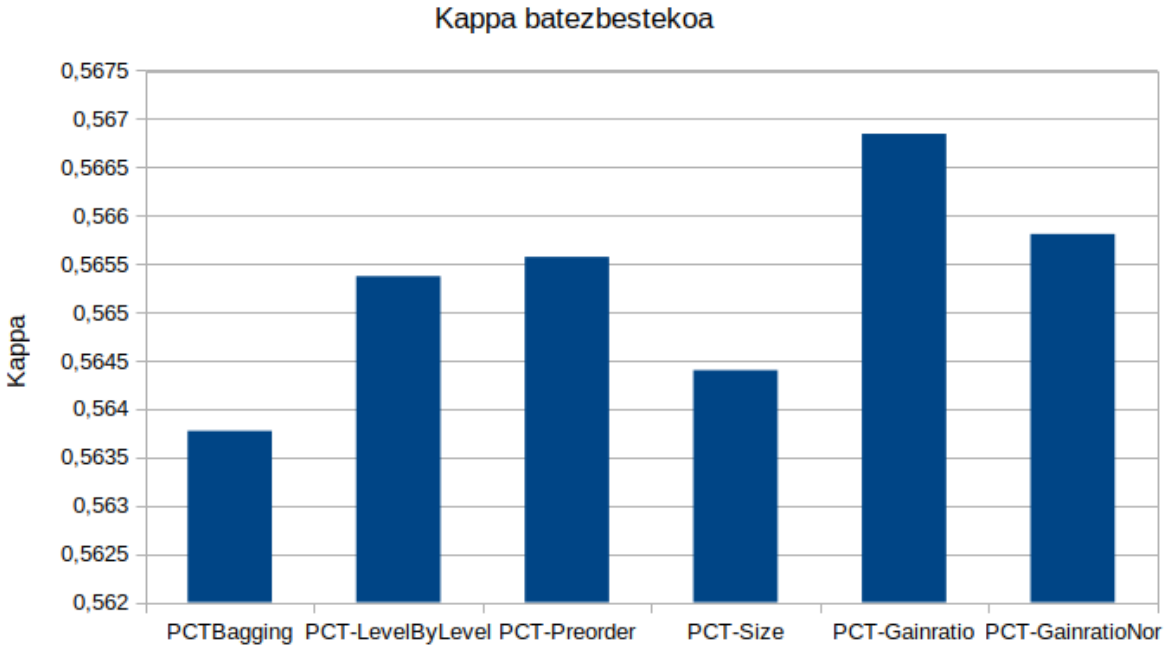
Irudia 41: Kappa grafikoa

PCT algoritmoetara hurbiltzen bagara onena zein den ikusteko, 42. grafikoan ikus daiteke PCTBagging eta PCT-Size direla okerrenak; beraz, esan daiteke kasu honetan ere algoritmo berriak hobetu egiten duela PCTBagging. Irizpide desberdinen emaitzak nahiko konstanteak dira grafiko osoan, eta onena gainratioa dela ikus daiteke.



Irudia 42: Kappa grafikoa hurbilduta

Batezbestekoa egiten badugu, 43. irudia, emaitzak hauek eta asmatze-tasarenak ia berdinak direla ikusten da. Kappak, espero daitekeen akordioaz gain, kontuan hartzen baitu aurrezikusitako sailkapenen eta sailkapen errealen arteko akordioa, *accuracy* egiten duen bezala. Kasu honetan emaitzarik onenak gainratioa ere ematen ditu.



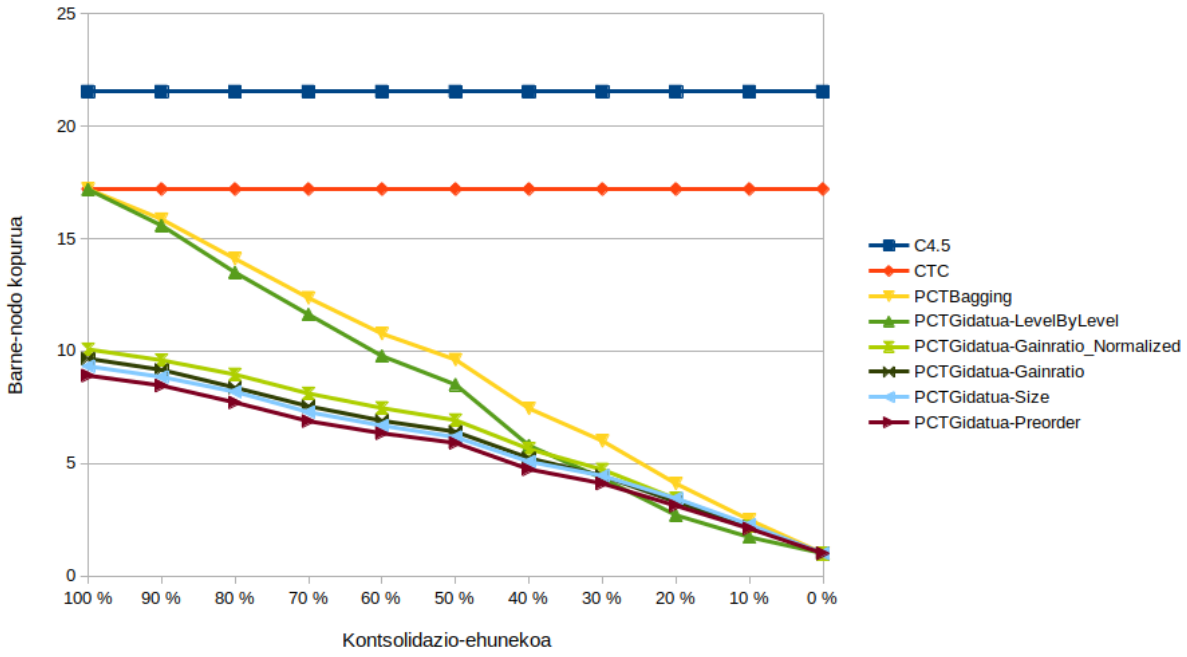
Irudia 43: Kappa batezbesteko grafikoa

9.2 AZALTZEKO GAITASUNA

Orain ikusiko dugu zeintzuk diren algoritmo onenak azalpen-gaitasunari dagokionez. Nodo kopuru txikiena behar duten algoritmoak onenak izango dira, helburua sailkapenari azalpen erraz bat ematea baita. Irudikapen zehatzagoa eta argiagoa lortzean, interpretazio ulergarriagoa eskaini ahal izango dugu, hala nola, sailkapena nola egin den eta datu multzoaren prozesuan zein ezaugarri izan diren garrantzitsuenak.

9.2.1 Barne-nodo kopurua

44. irudiaren grafikoan, PCTgidatuaren irizpide desberdinak eta aukeratutako lehiakideak alderatzen ari dira. PCTBagging-a da nodo gehien behar dituena, beraz, kasu honetan eraginkortasun gutxien duena, txarrena. PCT-ren aldagai guztiek hobetzen dute PCTBagging-a, baina onena PCTGainratioa da.



Irudia 44: Barne-nodo kopuruen grafikoa

Grafikoan ikus daitekeenez, zuhaitzaren mailetan maximo bat jarriz gero, alde handia dago gainerakoekin; argi dago maila kopuru desberdinekin barneko-nodo kopuru ezberdina egongo dela.

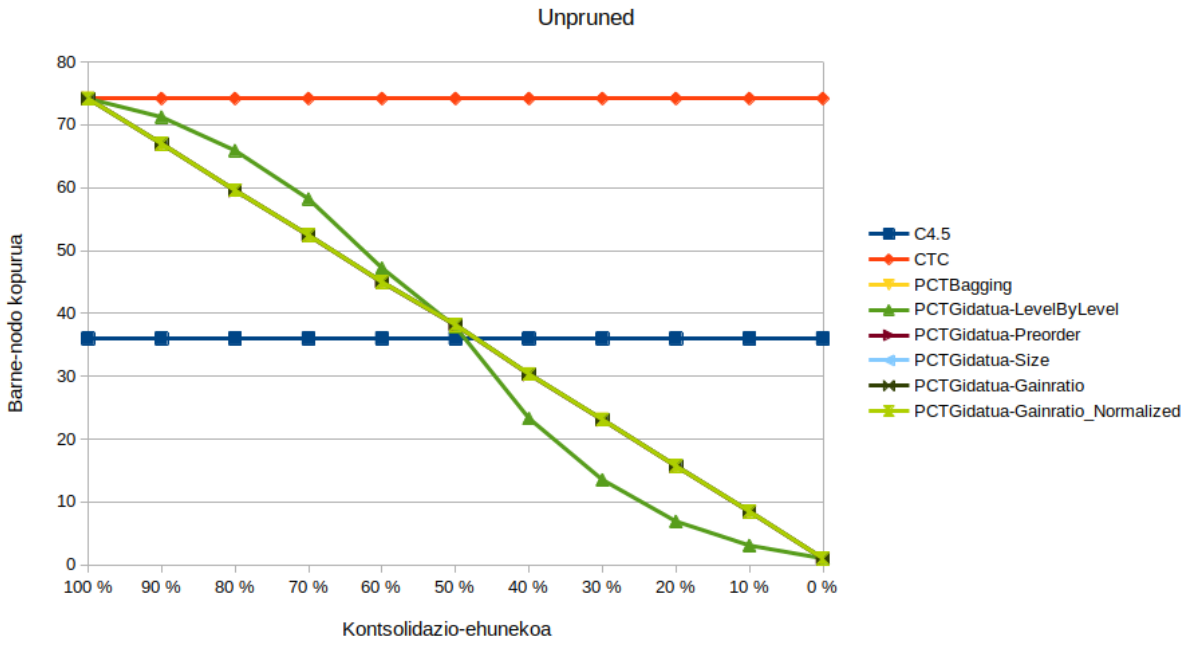
Ehuneko bat kontsolidazio-parametro gisa erabiltzen denean, pentsa liteke barne-nodoen kopurua berbera izan beharko litzatekeela nodo-kopuruan muga bat ezarri den kasu guztietan. Printzipioz, horrela izan beharko luke. Orduan, zergatik dago desberdintasun hori? Inausitako zuhaitz kontsolidatuaren nodo-kopurua jakin ondoren, zuhaitza berriro garatuko dugu erabiltzaileak emandako parametroetara iritsi arte (puntu horretara arte, nodo-kopuruak berdin-berdina izan beharko luke) eta gero zuhaitza berriro kolapsatu edo inausi egiten dugu. Eta batzuetan (gutxitan) aldaketa bat gertatzen da adarren bat kolapsatu edo inausteari dagokionez. Hori dela eta, aldaketa txiki bat dago PCTgidatuaren aldaeren artean nodo kopurua mugatzea erabakitzen dugunean.

Azkenik, PCTBagging-aren eta PCTgidatuaren arteko ezadostasunaren arrazoia da, aurretik azaldu bezala, PCTBagging-ean zuhaitza garatzen dela, kolapsatu edo inausi egiten dela (beharrezkoa bada) eta gero nodo batzuk ezabatzen direla. Gidatutako PCTgidatuaren kasuan, berriz, zuhaitza eraikitzen da ezarritako mugara iritsi arte, eta, ondoren, inausi egiten da. Horregatik, emaitzetan ikusitako alde hori gertatzen da.

Gainerako algoritmoekin alderatuta, Bagging-a emaitzarik txarrena lortzen duena da, lagin bakoitzerako zuhaitz desberdin bat sortzean ez baitu emaitzaren azalpenik ematen, eta horregatik ez da grafikoan ere agertzen. CTC da nodo-kopuru handiena eta, beraz,

emaitza txarrenak ematen dituena, eta PCT-en berdina da kontsolidazioa %100ekoa denean. Azkenik, C4.5 grafikoaren erdian mantentzen da prozesu guztian. Emaitzarik onenak PCT aldaerak ematen dituzte.

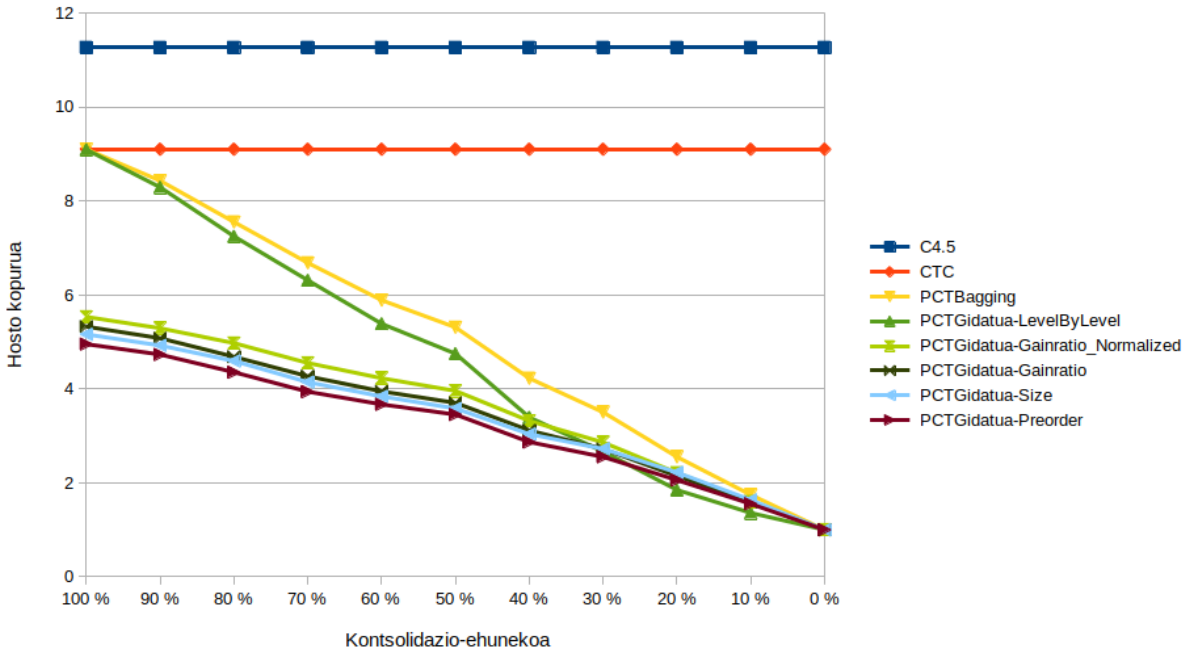
Unpruned zuhaitzen kasuan, zuhaitzak ez direnez inausi edo kolapsatu behar, nodo-kopurua PCTren aldaera guztietan, PCT-LevelByLevel-en izan ezik, berdin-berdinak dira. 45. irudian unpruned zuhaitzak erabili dira eta, ikus daitekeenez, guztiek nodo kopuru bera dute.



Irudia 45: Unpruned zuhaitzekin egindako barne-nodo kopuruen grafikoa

9.2.2 Hosto kopurua

Hosto kopuruari dagokionez, 46. irudian ikus daitekeen bezala, emaitzak barne-nodoen berdin-berdinak dira, alde bakar batekin: kasu honetan, hosto kopurua txikiagoa da, normala denez.

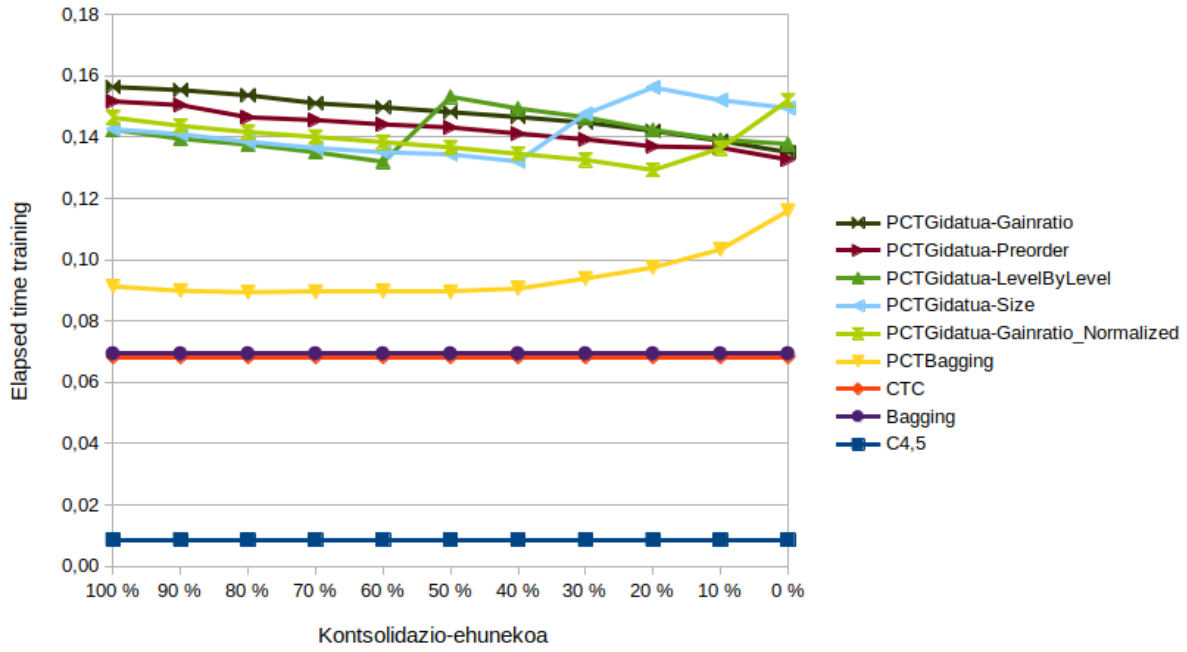


Irudia 46: Hosto kopuruen grafikoa

9.3 KOSTU KONPUTAZIONALA

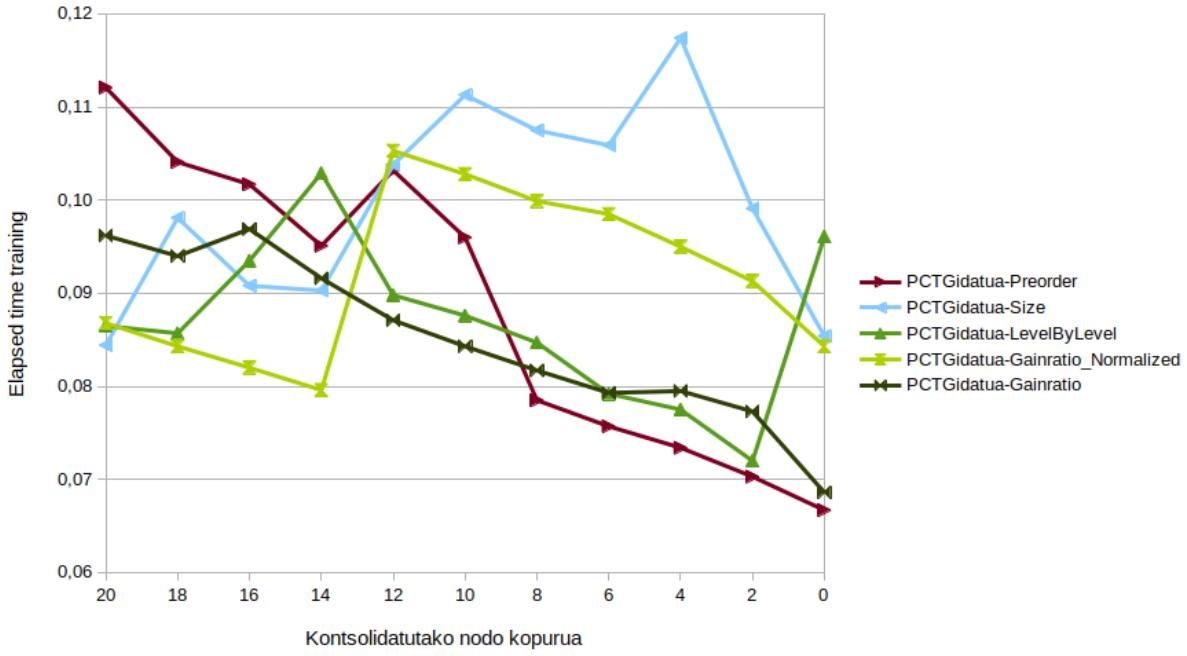
Atal honetan algoritmo bakoitzaren kostu konputazionala alderatuko dugu. Kasu honetan, algoritmo bakoitzak datu-multzoa entrenatzeko behar izan duen denbora alderatuko dugu.

47. irudiaren grafikoari erreparatzen badiogu, ikus daiteke kontsolidazio-ehunekoa erabilita PCTgidatuaren irizpideak direla denbora gehien behar dutenak. Hau logikoa da; izan ere, ehunekoa erabiliz gero, lehenik eta behin zuhaitz osoa eraiki behar da zuhaitzaren nodo guztien (edo mailen) kopurua jakiteko, erabiltzaileak emandako ehunekoari dagokion nodo-kopurua kalkulatzeko, eta, ondoren, berriz eraiki behar da zuhaitza, erabiltzaileak aukeratutako nodo-kopurua (edo maila) zein den jakinda. Horregatik, logikoa da PCTBagging-a baino denbora gehiago behar izatea.



Irudia 47: Igarotako denbora kontsolidazio-ehunekoekin grafikoa

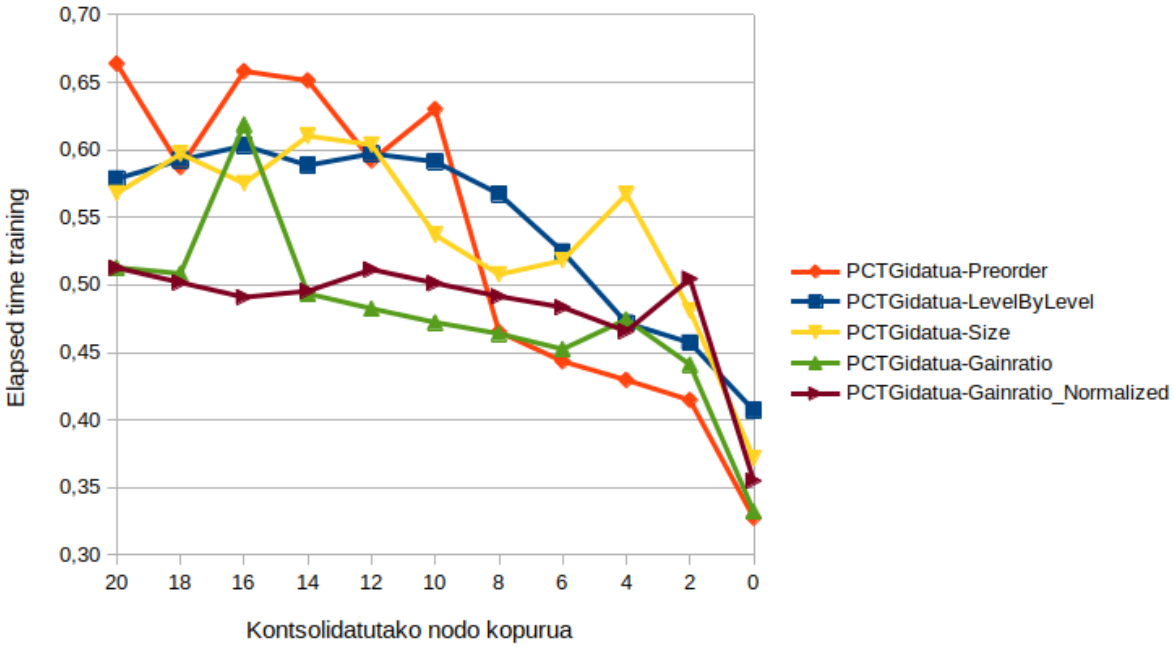
Jarraian, 48. irudiaren grafikoan PCTgidatuaren irizpideak alderatzen dira, kontsolidatu beharreko nodo (edo maila) kopururako balio numerikoak erabiliz (eta ez ehunekoak). Horrela, algoritmoak zuhaitz kontsolidatua garatzeko behar duen denbora erakusten da, erabiltzaileak esaten duen maximora arte (aldez aurretik osorik eraikitzeko beharrik gabe).



Irudia 48: Igarotako denbora balio zehatzekin grafikoa

Aurreko grafikoan ikus daitekeenez, nahasketa asko dago eta ezin da ezer behar bezala ikusi. Hau gertatzen da datu-baseek oso tamaina desberdinak dituztelako, eta, beraz, ez da egokia konparazioa horrela egitea.

Balio zehatzekin behar den entrenamendu-denbora hobeto ikusteko, zamatsua den datu-base bat aukeratu da. Kasu honetan, *abalone19* datu-basea aukeratu da PCTgidatuaren aldaera desberdinen denborak alderatzeko.



Irudia 49: *abalone19* datu-basea igarotako denbora balio zehatzekin grafikoa

49 irudian ikus daitekeenez, gorabeherak daude, baina gutxi gorabehera joera bat ikus daiteke. Zenbat eta nodo gutxiago kontsolidatu denbora txikiagoa da, hori logikoa da, algoritmoa lehenago gelditzuko baita.

9.4 ESPERIMENTAZIOAREN ONDORIOAK

Amaitzeko, hainbat neurriekin esperimentazioa ikusi ondoren, PCTgidatua algoritmoak PCTBagging algoritmoa hobetzen duela ondoriozta daiteke. PCTgidatuaren ia edozein aldaera PCTBagging baino hobea da, baina onena dirudiena PCTGainratioa da. Horren arrazoia izan daiteke, beste aldaeretan ez bezala, PCTGainratioak ezaugarri adimentsuagoak hautatzen dituela, informazioaren irabazia ez ezik, erabaki-zuhaitzeko nodo bakoitzean dauden zatiketa posibleen kopurua ere kontuan hartzen duelako, eta horrek klaseen edo helburuen arteko diskriminazio hobea ahalbidetu lezake.

Esperimentazioa algoritmoaren aldaera egokia aukeratzearen garrantzia nabarmentzen du, baita aldaera aukeratzeak algoritmoen errendimenduan nola eragin dezakeen ere.

Hala ere, zuhurra izango litzateke datu-multzo gehiagorekin probatzen jarraitzea, joera horri eusten ote zaion egiaztatzeko.

10 ONDORIOAK ETA LERRO IREKIAK

Proiektu honetan PCTBagging gidatua algoritmoa diseinatu eta integratu da WEKA plataforma librean. PCTBagging (*Partially Consolidated Tree Bagging* edo zuhaitz partzialki kontsolidatu Bagging-a) algoritmoaren helburua CTC-ren eta Bagging-aren arteko hibrido bat sortzea da. PCTBagging-ak beti modu berean kontsolidatzen du zuhaitz partziala, eta hori da PCTBagging gidatua aldatzen saiatzen dena. PCTgidatuak aukera ematen dio erabiltzaileari zuhaitza kontsolidatuak nola garatu nahi dituen aukeratzeko. Erabiltzaileak aukera dezake garatu beharreko hurrengo nodoa zein izango den; adibidez, garatzeko hurrengo nodoa populazio gehien duen hurrengoa izan daiteke, edo irabazi-indize handiena duen hurrengoa. Gainera, erabiltzaileak nodo edo maila kopuru maximo bat aukera dezake kontsolidatzeko.

Esperimentazio bat egin da inplementazioaren baliozkotasuna egiaztatzeko eta PCTBagging gidatuaren aldaera desberdinen emaitzak PCTBagging-ekin eta beste algoritmo lehiakide batzuekin alderatzeko.

Esperimentazioa egin ondoren oso emaitza interesgarriak lortu dira. PCTgidatuaren emaitzak eta PCTBagging-enak antzekoak dira, baina badirudi algoritmo berri honek hobekuntza txiki bat lortzen duela PCTBagging-aren aurrean, erabilitako ia irizpide guztietan, sailkatzeko gaitasunerako, azaltzeko gaitasunerako edo kostu konputazionalerako.

Pertsonalki, asko gustatu zait proiektu hau egitea eta oso esperientzia aberasgarria iruditu zait. Ezaugarri hauetako proiektu bat nolakoa den jakiteko eta ikerketaren munduan murgiltzeko aukera eman dit. Javan berriro programatzeko lagundu dit ere, aspaldi ez nuelako horrekin lan egin. Azkenik, Eclipse eta Github-en funtzionamendua ikasteko eta Waikato unibertsitatearen WEKAko ezarpena ezagutzeko aukera eman dit.

10.1 LERRO IREKIAK

Proiektua egin ondoren, lerro batzuk irekita geratu dira:

- Zuhaitz kontsolidatuaren garapena gelditzeko irizpideetan beste aukera bat gehitu daiteke: Garapena gelditu hurrengo nodoa erroaren populazioaren tamainaren ehuneko bat baino txikiagoa denean, adibidez, zuhaitz kontsolidatua garatzen jarrai genezake garatu beharreko hurrengo nodoak kasuen %10 ez duen bitartean.
- Beste irizpide bat gehitu daiteke. Errore-tasan oinarrituta etorkizun handiena duen ume nodoa garatu hurrengoa, *NBTree* algoritmoan (WEKAn ere inplementatua) egiten den bezala.
- Gainera, esperimentazioa beste datu-base batzuetara ere zabaldu ahal izango litzate-

ke.

- Inplementatu den PCTBagging gidatua beste Ensembletan probatzea, hau da, beste PCTEnsembletan: Boosting, Random Subspace Method, Random Forest... eta ikusi irizpideek eragina bera duten edo aldatzen den.
- Edozein irizpide erabilita ere, zuhaitza inausi egin behar bada, hurrengo nodoa garatu baino lehen jada jakitea adabegi hori inausiko den ala ez. Estrategia honen atzean dagoen ideia da, kasu batzuetan, hasieratik argi dagoela zuhaitzaren adar batzuk ez direla oso baliagarriak izango sailkatzeko edo erabakiak hartzeko.

Erreferentziak

[1] Remco R. Bouckaert, Eibe Frank, Mark Hall, Richard Kirkby, Peter Reutermann, Alex Seewald, and David Scuse. *Weka Manual for Version 3-6-10*. The University of Waikato, 2013.

[2] Igor Ibarguren, Jesús M. Pérez, Javier Muguerza, Olatz Arbelaitz, and Ainhoa Yera. Pctbagging: From inner ensembles to ensembles. a trade-off between discriminatin capacity and interpretability. 2022.

[3] Jesús M. Pérez de la Fuente. Árboles consolidados: Construcción de un árbol de clasificación basado en múltiples submuestras sin renunciar a la explicación. 2006.

[4] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., 1993.

[5] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[6] CMG Lee. Roc kurba. `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`. Lizentzia: CC BY-SA 4.0.

[7] WEKAren iturburu-kodea deskargatzeko URL-a. `https://www.cs.waikato.ac.nz/~ml/weka/`.

[8] Javier Campos Laclaustra. *Estructuras de datos y algoritmos*. Prensas Universitarias de Zaragoza, 1995.