



Universiteti i Prishtinës “Hasan Prishtina”

Tema: Aplikimi i algoritmit të klasifikimit Random Forest në
predikimin e rezultatit të video-lojës Dota 2

Mentorë: Prof. Dr. Ing. Lule Ahmed

Studenti: Labinot Vila

Prishtinë, Dhjetorë 2019

Abstrakti

IntelliDota është një projekt i realizuar me anë të gjuhëve programuese Scala dhe Flutter, gërshetimi i së cilave sjell një aplikacion të mençur, të qëndrueshëm, të shpejtë dhe ndihmues, me anë të së cilit mund të analizojmë burime të të dhënave, vizualizojmë dhe të aplikojmë metrika dhe algoritme të ndryshme mbi to. Ky projekt është i ndarë në dy pjesë, ky dokumentin ka të bëjë me pjesën IntelliDota Classification.

IntelliDota Classification paraqet pjesën e parë të projektit IntelliDota në të cilin përdoret një data set i vetë krijuar në të cilin aplikohet algoritmi Random Forest. Ma fjalët data set i vetë krijuar duam të themi që data seti nuk merret i gatshëm, por krijohet me ndihmesën e Steam Public API format i të cilit është Json, filtrohet në bazë të disa kolonave të caktuara dhe përgatitet për algoritëm.

Pas krijimit të algoritmit, e ruajmë modelin e trajnuar në mënyrë që të ripërdoret kurdo që duam të kryejmë një klasifikim të një rreshti të ri. Kjo është realizuar nëpërmjet kontejnerit të izoluar Docker.

Pasi ky aplikacion përmban dy pjesë, ne kemi krijuar një *backend* aplikacion në të cilin ofrojmë një ndërfaqe programuese për shërbimet e pjesës IntelliDota Clustering. Këto shërbime i kemi krijuar duke përdorur gjuhën programuese Scala, saktësisht strukturën Play.

Scala njihet si një gjuhë e fuqishme programuese ndërsa Play është një strukturë që ofron modelin MVC për programim në ueb. Bashkë me Play, është përdorur edhe struktura Spark, që është mënyra kryesore e realizimit dhe zgjidhjeve të problemeve tona. Spark ofron manovrim me data sete, programim 'të shpërndarë' dhe një numër të madh algoritmesh të të mësuarit të makinës. Scala është e shkruar në gjuhën programuese Spark, andaj kombinohet në mënyrë të mahnitshme.

Në mënyrë që të ekzistojë qasja për në pikat fundore pa ndalesa, projekti është konvertuar në një kontejner dhe është hostuar në Cloud nëpërmjet Google Cloud Platform. Konvertimi në kontejner është realizuar nëpërmjet Docker. Lidhja për të kaluar në API realizohet nëpërmjet <https://serverfinal3-qm4ka2ucaq-ew.a.run.app>.

Falenderimet

Si fillim, falënderoj familjen përgjatë këtyre viteve për mbështetjen dhe motivin, duke vlerësuar përkushtimin dhe vështirësitë, e më pas Prof. Dr. Lule Ahmedi për kontributin, seriozitetin dhe nxitjen që na ka ofruar si mua, ashtu edhe çdo studenti që ka pasur nderin të ligjërohet nga ajo përgjatë këtyre viteve.

Po ashtu, një falënderim i veçantë shkon për kompaninë Prime L. L. C, saktësisht mentorit Endrit Bytyqi për gatishmërinë dhe aftësitë që asnjëherë nuk ka nguruar për t'i shpërndarë.

Lista e tabelave

Tab 1: Pika fundore index	47
Tab 2: Pika fundore getColumnns	47
Tab 3: Pika fundore getSample	47
Tab 4: Pika fundore getStages	48
Tab 5: Pika fundore: getCorrelationMatrix	48
Tab 6: Pika fundore getGroupAndCount	48
Tab 7: Pika fundore getStages	48
Tab 8: Pika fundore getSchema	49
Tab 9: Pika fundore getDoubleGroup	49
Tab 10: Pika fundore postPredict	50

Lista e figurave

Fig 1: Algoritmi i një loje tick-tack-toe.....	8
Fig 2: Mësimi me mbikëqyrje i shprehur matematikisht	11
Fig 3: Shembull i Decision Trees	13
Fig 4: Teorema e Bajesit	13
Fig 5: Përmbledhje e Random Forest.....	14
Fig 6: Drejtëza ndarëse në SVM.....	15
Fig 7: Pamja e parë e formacionit të video-lojës Dota 2	17
Fig 8: Organizimi i ekipit	18
Fig 9: Dyshja çelës – domain.....	19
Fig 10: Lidhja për një lojë të caktuar.....	19
Fig 11: Intervali i përzgjedhur për të dhëna	20
Fig 12: Kushtet për anashkalim të video-lojës Dota 2.....	20
Fig 13: Copëzat e data setit me rreshtat përkatës.....	21
Fig 14: Mostra e data setit	22
Fig 15: Kolonat e data setit me tipet përkatëse	22
Fig 16: Popullimi i leaver_status në data set.....	23
Fig 17: Raporti leaver_status me radiant_win ne data set.....	23
Fig 18: getGroupAndCount në 3 ndarje sipas xp_per_min	24
Fig 19: getGroupAndCount në 4 ndarje sipas level.....	24
Fig 20: getGroupAndCount në 3 ndarje sipas hero_healing	24
Fig 21: getGroupAndCount në 4 ndarje sipas denies	25
Fig 22: Atributet e zgjedhura nga struktura.....	25
Fig 23: Struktura e një objekti Json	26
Fig 24: Modeli i klasës Match.....	27
Fig 25: Unaza kryesore e aplikacionit	28
Fig 26: Struktura e metodës fetchGames	29
Fig 27: Konvertimi dhe ruajtja e data setit në makinën lokale.....	29
Fig 28: Pamja e marrjes së të dhënave	30
Fig 29: Leximi i data setit të krijuar.....	30
Fig 30: Renditja e fazave sipas realizimit	31
Fig 31: Vizualizimi i thjeshtë i fazave.....	32
Fig 32: Saktësia e algoritmit Random Forest Classifier	32
Fig 33: Ruajtja e modelit të trajnuar në makinën lokale.....	32
Fig 34: Shembull i gjetjes e një intervali ndër kuartile	33
Fig 35: Struktura e projektit në Scala	34
Fig 36: Paraqitja e objektit Pre.....	34
Fig 37: Paraqitja e objektit Dataset.....	35
Fig 38: Fazat të reprezentuar si JSON objekte – data seti 1.....	36
Fig 39: Fazat të reprezentuar si JSON objekte – data seti 2.....	36
Fig 40: K-Means në Scala	37
Fig 41: Forma e predikimit POST.....	37
Fig 42: Rezultati i POST kërkesës për predikim.....	37
Fig 43: Krijimi i RDD nga lista e attributeve hyrëse	38
Fig 44: Krijimi i data setit nga RDD.....	38

Fig 45: Modeli i trajnuar dhe klasifikimi në kohë reale	38
Fig 46: Pamja e parë e Dockerfile	39
Fig 47: Përmbledhje e hapave të krijimit të imazhit	40
Fig 48: Komandat për ndërtimin e Docker imazhit	41
Fig 49: Pamja e parë e ndërtimit të imazhit në Docker	41
Fig 50: Ndërtimi i suksesshëm i imazhit	42
Fig 51: Listimi i kontejnerëve.....	42
Fig 52: Direktoriumi ku po punojmë në kontejner	42
Fig 53: Përmbajtja e kontejnerit	43
Fig 54: Direktoriumi bazë i kontejnerit	43

Tabela e përmbajtjes

1	Hyrje	8
1.1	Motivimi – Pse inteligjenca artificiale?.....	8
1.2	Përshkrimi i problemit	9
2	Inteligjenca artificiale – Klasifikimi si mësim me mbikëqyrje	11
2.1	Algoritmet e mësimimit me mbikëqyrje.....	12
2.1.1	Pemët vendim-marrëse.....	12
2.1.2	Naïve Bayes	13
2.1.3	Random Forest	14
2.1.4	Support Vector Machine	14
2.1.5	Algoritmi i zgjedhur për aplikacion	15
3	Hyrje në projekt	16
3.1	Loja kompjuterike Dota 2	17
4	Koleksionimi i të dhënave për klasifikim dhe para procesimi.....	19
4.1	Pamja e parë e strukturës Json nga e cila duam të marrim të dhënat	19
4.2	Para procesimi i të dhënave të mbledhura	20
4.3	Metrika gjenerike për data setin e derivuar	21
4.4	Krijimi i data setit – Mënyra e Scalas	26
4.5	Algoritmi i klasifikimi në Scala	30
4.6	Largimi i të veçuarve	33
5	Klasifikimi në kohë reale dhe ruajtja si model i trajnuar	34
5.1	Struktura e aplikacionit	34
5.2	Realizimi i klasifikimi në kohë reale.....	35
6	Përgatitja e imazhit për ekspozimin e shërbimeve	39
6.1	Çka është Docker?	39
6.2	Krijimi dhe shfletimi i kontejnerit	39
7	Konkluzioni	44
8	Lista e pikave fundore	47
8.1	index	47
8.2	getColumnns.....	47
8.3	getSample	47
8.4	getStages.....	48
8.5	getCorrelationMatrix.....	48

8.6	getGroupAndCount	48
8.7	getStages.....	48
8.8	getSchema	49
8.9	getDoubleGroup.....	49
8.10	postPredict.....	50
9	References	51

1 Hyrje

Ashtu siç kureshtja e njeriut nuk shuhet kurrë, ashtu edhe përparimi i teknologjisë nuk ka të ndalur. Megjithëse nuk është e gabuar nëse themi se teknologjia gati ka arritur majat e veta, akoma ka shtigje të pashkelura që mund të themi se paraqesin një botë teknologjike në vete. Ndër to, dhe ndër më të pëlqyerat e kërkuarat nga njerëzit është inteligjenca artificiale (ang. *artificial intelligence*).

1.1 Motivimi – Pse inteligjenca artificiale?

Inteligjenca artificiale, e referuar ndryshe si inteligjenca e makinave (ang. *machine intelligence*) ka të bëjë me stimulimin e inteligjencës të përvetësuar dhe zhvilluar nga makinat [1].

Kjo shkencë ndryshe definohet si fusha e studimit të agjentëve inteligjentë (ang. *intelligent agents*): një pajisje që kupton ambientin ku ndodhet dhe ndërmerr veprime të tilla që mundësia e arritjes së qëllimit të jetë maksimale.

Për një agjent themi se është racional nëse bën gjënë e duhur, pra secili veprim në rrethana të caktuara është i saktë. Por si mund të definojmë se çfarë është e saktë dhe çfarë jo? Nëse agjenti kalon nëpër një sekuencë veprimesh dhe gjendjesh që na kënaqin themi se agjenti ka pasur një performancë të mirë [2].

Pra, makinat tentojnë që të përvetësojnë aftësitë njerëzore të të kuptuarit dhe zgjidhjes së problemeve (ang. *learning and problem solving*).

Qëllimi i një agjenti inteligjentë mund të jetë i thjeshtë, si për shembull luajtja e një loje GO, apo kompleks siç është kryerja e operacioneve matematikore.

Parimi bazë i inteligjencës artificiale është përdorimi i algoritmeve. Algoritmet janë një grumbull instruksionesh që një makinë kompjuterike mund të ekzekutojë. Një algoritm kompleks ndërtohet si bashkësi e algoritmeve të thjeshta (Fig 1).

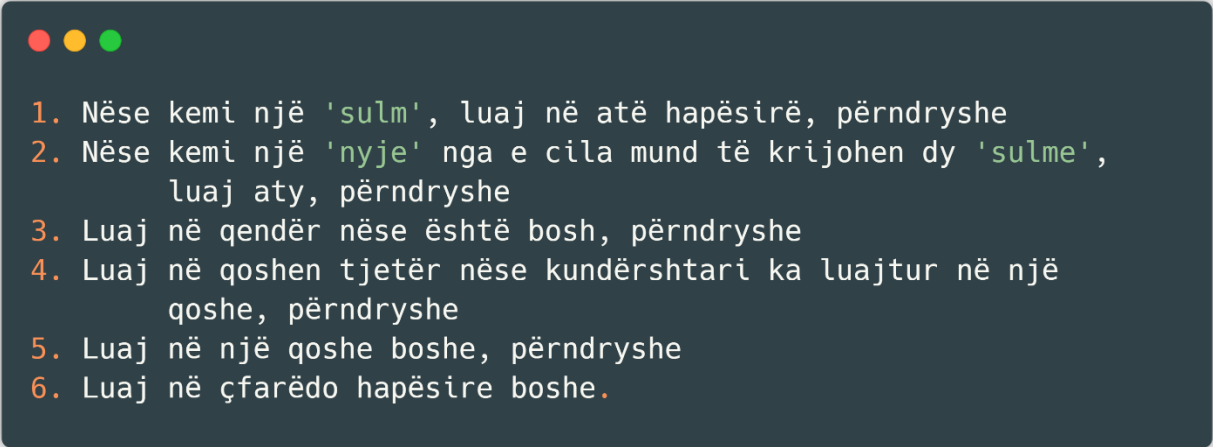
- 
1. Nëse kemi një 'sulm', luaj në atë hapësirë, përndryshe
 2. Nëse kemi një 'nyje' nga e cila mund të krijohen dy 'sulme', luaj aty, përndryshe
 3. Luaj në qendër nëse është bosh, përndryshe
 4. Luaj në qoshen tjetër nëse kundërshtari ka luajtur në një qoshe, përndryshe
 5. Luaj në një qoshe boshe, përndryshe
 6. Luaj në çfarëdo hapësire boshe.

Fig 1: Algoritmi i një loje tick-tack-toe

Disa algoritme, janë të aftë të mësojnë nga një grumbull të dhënash, si në rastin tonë, ku algoritmi mëson një strategji apo një mënyrë të mirë (ang. *rule of thumb*) të cilin e zbaton në të dhëna të reja, e disa algoritme të tjerë mund të vetë shkruajnë algoritme tjera.

Disa nga algoritmet që mësojnë, siç janë *fqinji më i afërt* (ang. *nearest-neighbor*), *pemët me vendime* (ang. *Decision Trees*), apo *rrjeti Bajesian* (ang. *Bayesian network*) munden teorikisht të përafrojnë çfarëdo të dhënash në një funksion të caktuar (nëse kanë memorie dhe kohë të pafundme). Që në kohët e hershme është stimuluar inteligjenca artificiale, e ndër qasjet më të njohura janë:

- Simbolizmi, njohur ndryshe si logjika formale apo implikacioni: nëse personi ka të ftohtë, atëherë ai ka grip.
- Interference Bajesiane (ang. *Bayesian interference*): nëse personi ka të ftohtë, atëherë ekziston një probabilitet që ai të ketë edhe grip.
- *Vektorit Mbështetës i Makinës* (ang. *Support Vector Machine*) apo *Nearest-neighbor*: pas shqyrtimit të të dhënave të personave që kanë të ftohtë, duke përfshirë moshën, simptomat dhe faktorët tjerë, dhe këto faktorë përkasin me pacientin aktual, themi se pacienti ka grip.

Për një makinë thuhet se është inteligjente nëse kalon testin e Turingut, test ky i cili u dizajnuar në kohët e hershme që të sjell një përkufizim të kënaqshëm për inteligjencën. Ky test kalohet nëse është e pamundur të tregohet se a vijnë përgjigjet nga një njeri apo nga një makinë pas disa pyetjeve nga njeriu [3].

1.2 Përshkrimi i problemit

Dihet se sot po thuajse në çdo fushë të jetës gjejmë aplikim të inteligjencës artificiale e sidomos në fushën e kompjuterikes, duke filluar nga sistemet rekomanduese, sistemet për predikim e shumë të tjera.

Duke pasur parasysh këto aplikime, ne kemi tentuar që të ndërtojmë një asistent për video-lojën e njohur Dota 2 pasi jo vetëm që do t'u ndihmonte njerëzve përgjatë lojës, por edhe do të pasuronte komunitetin me një data set unik (pasi po e krijojmë vet një të tillë nëpërmjet thirrjeve përkatëse) dhe kod burimorë që paraqet ecurinë e projektit dhe një shembull për të realizuar diçka të ngjashme si nga ana e inteligjencës artificiale, edhe nga ana e realizimit të thirrjeve dhe filtrimit të strukturës Json.

Duhet cekur se ekzistojnë shumë data sete të lidhura me këtë temë në internet, por për arsye të teknikave dhe metodologjive që dëshirojmë të implementojmë, kemi zgjedhur që ta krijojmë vet një të tillë.

Problemi jonë mund të përshkruhet nëpër disa faza, ku më të rëndësishmet janë:

- Krijimi i data setit – duke kryer këtë hap me sukses, neve na jepet mundësia të aplikojmë metrika, krahasime e edhe algoritme për inteligjencë artificiale në burimin e të dhënave përkatës. Si çështje kryesore paraqitet mënyra e realizimit të kërkesave dhe filtrimi i tyre, në mënyrë që të marrim një data set të shëndoshë.
- Aplikimi i algoritmit – na krijohet një model i trajnuar, qoftë ai lokal apo jo me anë të së cilit mund të aplikojmë klasifikimin në kohë reale. Lokalisht, kjo kryhet lehtë pasi nuk ka nevojë për masa tjera për të ruajtur modelin, për derisa nëse dëshirojmë të publikojmë në një server nikoqir (ang. *host*) duhet ruajtur modelin si një imazh.
- Zgjedhja e gjuhës programuese – për të realizuar kërkesat e më sipërme na nevojitet një gjuhë programuese që ofron fuqi dhe metoda të shumta në procesim dhe para procesim të të dhënave, e edhe opsione për inteligjencë artificiale.
- Krijimi i pikave fundore – u jep thirrjeve që realizohen nga jashtë një adresë të caktuar se ku duhet referuar për të marrë rezultatet e dëshiruara.

Pra, për aplikacionin tonë themi se ka kryer punën me sukses nëse është në gjendje të klasifikoj dhe kllasteroj të dhënat që i japim si hyrje në kohë reale, në mënyrë të shpejtë dhe të organizuar.

Pjesa inteligjente e aplikacionit mund të kryhet në shumë mënyra, duke filluar nga gjuha programuese Python me libraritë përkatëse që mundësojnë krijimin e një aplikacioni të ngjashëm me atë se çfarë duam të bëjmë ne e deri te shkrimi i një algoritmi nga ana jonë. Por meqë nuk është parë e arsyeshme të 'ri zbulojmë rrotën' ne kemi vendosur të përdorim gjuhën programuese Scala bashkë me Spark që ofron një arsenal të fuqishëm për manovrim me të dhëna dhe inteligjencë artificiale. Një përshkrim më në detaje se përse kemi zgjedhur këtë gjuhë me libraritë përkatëse do të shpjegohet më vonë.

E sa i përket pjesës së krijimit të të dhënave, ekzistojnë mënyra të tjera (siç janë përdorimi i një gjuhe programuese për të krijuar thirrjet) por jo metodologji, që do të thotë se data seti që kemi krijuar ne mund të merret vetëm nëpërmjet ueb-faqes zyrtare Steam nëpërmjet thirrjeve dhe filtrimeve përkatëse.

2 Inteligjenca artificiale – Klasifikimi si mësim me mbikëqyrje

Në përgjithësi ekzistojnë forma të ndryshme të të mësuarit të makinës, por tri më konkretisht përfaqësojnë pothuajse çdo formë të të mësuarit. Kemi:

- Mësimin pa mbikëqyrje (ang. *unsupervised learning*) – agjenti mëson mënyra për të zgjidhur apo kalkuluar problemin e caktuar varësisht hyrjes edhe pse nuk ka reagime dalëse nga burime të caktuara (pra, algoritmi nuk shpërblehet e as nuk dënohet për veprimin e zgjedhur). Teknika më e zakonshme dhe e përhapur është kllasterimi, me anë të së cilës bëhet grupimi potencialisht i saktë nga të dhënat hyrëse. Për shembull, një taksi gradualisht zhvillon konceptin e një 'dite të mirë trafiku' apo 'një dite jo të mirë trafiku' pa i dhënë asnjëherë shembuj të mëparshëm nga një 'mësues'.
- Mësimi i sforcuar (ang. *reinforcement learning*) – agjenti mëson nga një seri e veprimeve dalëse, qofshin ato veprime dënuese apo shpërblyese. Për shembull, një bakshish në fund të vozitjes me taksi i tregon algoritmit se ka vepruar mirë, përndryshe ka vepruar keq. I takon algoritmit të vendos se cilat pjesë të tij kanë bërë që të kryejë veprimin mirë.
- Mësimi me mbikëqyrje (ang. *supervised learning*) – agjenti mëson nga çiftet hyrëse – dalëse, dhe mëson një funksion me anë të së cilit krijon një skemë ku orientohen vlerat e ardhshme hyrëse në ato dalëse.

Në këtë aplikacion do të përdoret saktësisht të mësuarit me mbikëqyrje, teknikë kjo që do të përshkruhet më poshtë saktësisht me algoritmin përkatës!

Qëllimi i mësimit me mbikëqyrje është përshkruar si më poshtë (Fig 2) ku x dhe y mund të jenë vlera të çfarëdoshme dhe funksioni h është hipoteza.

Dhënë një bashkësi çiftesh me N elemente trajnuese:
 $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ ku secila dalje
 y_j ($j = 1, 2, \dots, N$) është gjeneruar nga një funksion
i panjohur $y = f(x)$, atëherë gjej një funksion h që
përafron funksionin f .

Fig 2: Mësimi me mbikëqyrje i shprehur matematikisht

Për të masur saktësinë e hipotezës, ne japim një bashkësi testuese të dhënash të ndryshme nga të dhënat trajnuese. Për hipotezën themi se ka gjeneralizuar mirë nëse me saktësi ka predikuar vlerën y për llojin e ri të të dhënave [4]. Kur dalja y është një interval i vlerave të fundme, themi se problemi i të mësuarit është klasifikimi dhe quhet binarë apo Bulean nëse intervali ka vetëm dy vlera. Ndërsa kur y është një numër i vazhdueshëm themi se problemi i të mësuarit është regresioni. Teknikisht në regresion tentojmë të gjejmë një vlerë të afërt, sepse probabiliteti për t'ia qëlluar saktë vlerës është 0.

2.1 Algoritmet e mësimit me mbikëqyrje

Ekzistojnë shumë algoritme të të mësuarit me mbikëqyrje, megjithatë, edhe pse punojnë në parimin e njëjtë të gjithë kanë përparësitë, të metat dhe rastet ku një algoritëm është më i përshtatshëm se tjetri. Nuk ka ndonjë algoritëm që punon më së miri në të gjitha problemet e mundshme. Ndër parametrat kryesorë që bëjnë që një algoritëm të ketë përparësi ndaj një algoritmi tjetër janë:

- Ndryshueshmëria në të dhëna – nëse të dhënat hyrëse kanë llojllojshmëri të lartë (të dhëna diskrete, të vazhdueshme, të vazhdueshme rritëse).
- Përsëritja në të dhëna – nëse të dhënat hyrëse kanë përsëritje të mëdha, kjo rezulton në një matrice korrelacionesh të lartë që do të thotë se disa algoritme si Regresioni Linearë apo Regresioni Logjistik do të performojnë dobët.
- Prezenca e pa varësisë dhe jo linearitetit – nëse secili atribut i të dhënave hyrëse ka një ndikim të pa varur në rezultat, atëherë algoritmet lineare performojnë në përgjithësi më mirë se sa të tjerët.

Më poshtë do të listojmë disa nga algoritmet më të përhapur e më të përshtatshëm pavarësisht data setit.

2.1.1 Pemët vendim-marrëse

Pemët vendim-marrëse (ang. *Decision Trees*) përdoren për të klasifikuar një instancë duke e kaluar nga maja e pemës e deri tek një nyje gjethe, që jep rezultatin e instancës së klasifikuar. Qëllimi është të krijohet një model që predikon vlerën e një variable varësisht vlerave hyrëse të tjera.

Pra, ndërtohet një pemë duke ndarë bashkësinë e attributeve hyrëse në grupe të caktuara, ky proces, i quajtur ndarje rekursive (ang. *recursive splitting*), përsëritet në mënyrë rekursive në secilën nën bashkësi të përfituar. Ky proces mbaron atëherë kur atributi i përfituar është i vetëm, që do të thotë se ndarjet e më tutjeshme nuk ndikojnë në vlerën përfundimtare [5].

Pavarësisht të metave dhe përparësive, bazuar në përdorimin dhe dokumentimin masiv që ka ky algoritëm, lehtë themi se është ndër algoritmet më të fuqishme që po përdoren aktualisht.

Një shembull se si funksionon Decision Trees është i përshkruar më poshtë (Fig 3).

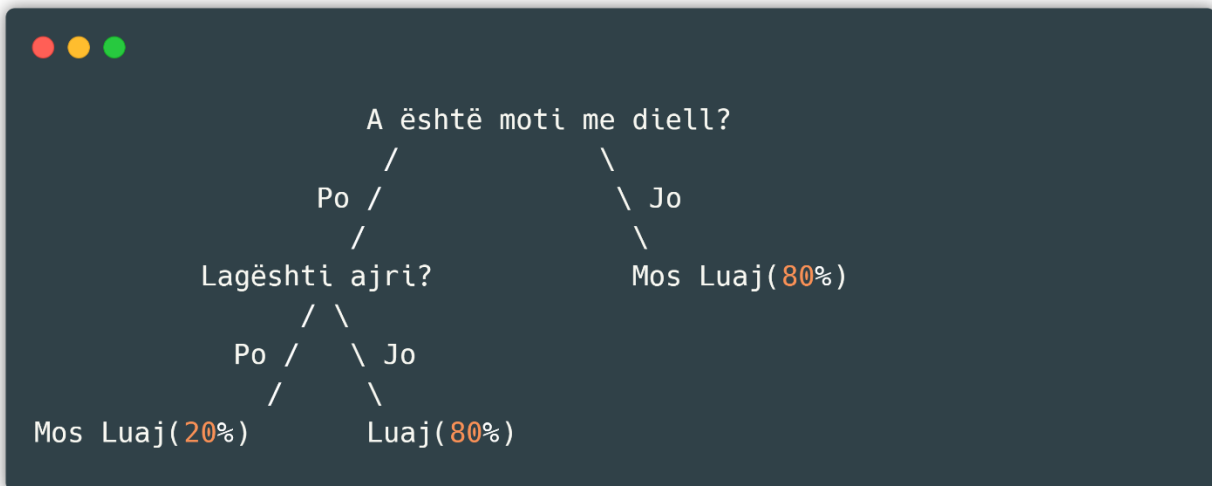


Fig 3: Shembull i Decision Trees

Përparësitë dhe të metat e qasjes *Decision Trees*

- + Janë të afta të gjenerojnë rregulla të kuptueshme.
- + Janë të afta të kryejnë klasifikimin pa llogaritje të mëdha.
- + Kryejnë edhe klasifikimin edhe regresionin.
- Janë të dobëta kur duhet predikuar vlerën e një atributi të vazhdueshëm.
- Nëse kemi pak të dhëna trajnuese, atëherë mazha e gabimit është e madhe.
- Për të trajnuar të dhëna, duhet fuqi e madhe kompjuterike.

2.1.2 Naïve Bayes

Hyn në familjen e klasifikuesve probabilistik (ang. *probabilistic classification*) ku si bazë përdoret teorema e Bajesit (Fig 4) nën supozimin se asnjë atribut nuk është i varur nga tjetri, nga edhe e ka marrë emrin *naïve* [6].

$$P(A/B) = P(B/A)P(A) / P(B) \text{ ku:}$$

$P(A/B)$ – probabiliteti i ngjarjes A kur ka ndodhur ngjarja B
 $P(B/A)$ – probabiliteti i ngjarjes B kur ka ndodhur ngjarja A
 $P(B)$ – probabiliteti i ngjarjes B

Fig 4: Teorema e Bajesit

Ndërsa më së miri shpjegohet me një shembull, nëse kanceri varet nga mosha, atëherë me teoremën Bajesiane mund të përdoret ky informacion që ti jap peshë vartësisë së kancerit nga mosha në predikimin se a ka personi përkatës kancer apo jo. Kjo metodë është e aplikueshme vetëm në të dhëna diskrete.

2.1.3 Random Forest

Është një algoritëm ansambël (ang. *ensemble algorithm*), që do të thotë se kombinon një apo më shumë algoritme të të njëjtit lloj apo të ndryshëm për të klasifikuar objektin, pra është një lloj sikur algoritmi të kalonte nëpër *SVM*, *Naïve Bayes* apo *Decision Tree* e në fund të votohej se cili algoritëm të merret si bazë. Ky lloj algoritmi krijon një bashkësi pemësh në mënyrë të rastësishme të zgjedhur nga të dhënat trajnuese, grumbullon votat nga bashkësi pemësh të ndryshme të votuara si vendim-marrëse dhe caktohet klasa finale e objektit [7]. Një përmbledhje se si funksionon Random Forest është përshkruar më poshtë (Fig 5).

Supozojmë se të dhënat trajnuese janë: [X1, X2, X3, X4] me attribute korresponduese [L1, L2, L3, L4]. Atëherë Random Forest mund të krijojë tri pemë që marrin si vlera trajnuese nënbashkësitë e të dhënave trajnuese, pra [X1, X2, X3], [X1, X2, X4], [X2, X3, X4].

Në fund predikimi bazohet në votat më të shumta prej pemëve përkatëse.

Fig 5: Përmbledhje e Random Forest

Algoritmet e tilla, pra ansambël algoritmet, janë shumë më të sakta se sa llojet e algoritmeve tjera për arsye se grupet e pemëve mbrojnë njëra tjetrën nga gabimet individuale, përveç rastit kur të gjitha grupet e pemëve dështojnë në të njëjtin mënyrë. Për shembull, kur disa grupe pemësh janë gabim, shumica tjetër janë saktë, andaj në këtë mënyrë pemët janë të afta të lëvizin të gjitha në një drejtim të saktë së bashku.

2.1.4 Support Vector Machine

Support Vector Machines në të mësuarit e makinës janë modele të mbikëqyrura me anë të së cilëve analizohen të dhënat të përdorura për klasifikim dhe regresion. Me një bashkësi të të dhënave trajnuese, ku secila i përket një kategorie, *SVM* mundet të ndërtoj një model që shembujve të ardhshëm t'ua shoqëroj kategorive përkatëse, duke e bërë këtë zgjedhje në mënyrë jo-probabilistike.

Konkretisht, një *SVM* është një reprezentim i pikave në hapësirë, të organizuara ashtu që secila pikë në hapësirë është e ndarë me një hapësirë që tentohet të jetë sa më e madhe e mundshme. Më pas, shembujt e ri të predikuar ndjekin po këtë mënyrë dhe kategoria të cilën i përkasin varet nga hapësira në të cilën bien. Si shembull kemi pjesën më poshtë ku pikat me ngjyrë të gjelbër reprezentojnë grupin e parë G1 ndërsa pikat me ngjyrë të verdhë reprezentojnë grupin e dytë

G2. Çfarëdo e dhëne e re që, siç u tha edhe në definicionin më lartë i takon njërit ndaj këtyre grupeve, i takon grupit, atributet e së cilit përafrojnë më shumë (Fig 6).

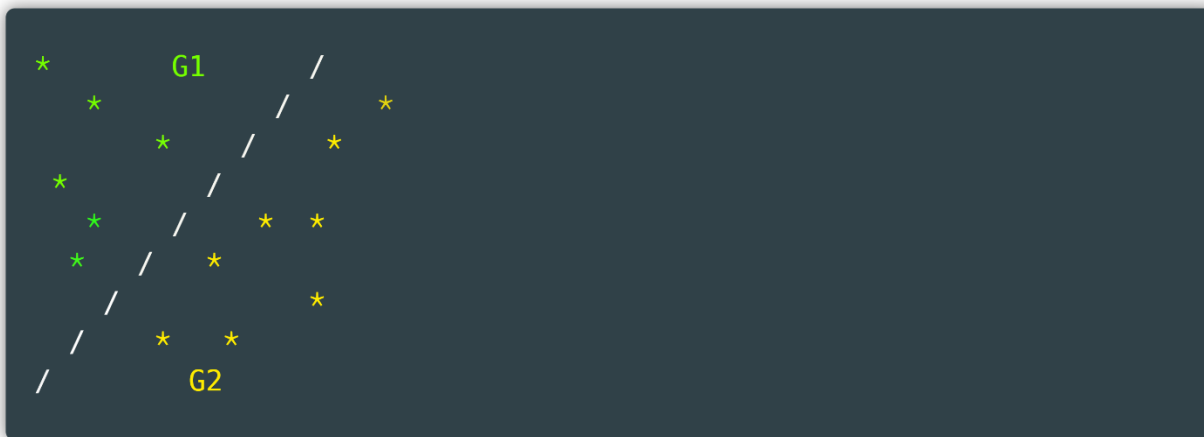


Fig 6: Drejtëza ndarëse në SVM

2.1.5 Algoritmi i zgjedhur për aplikacion

Duke pasur parasysh shumicën e algoritmeve, mënyrën e funksionimit si dhe rastet konkrete për aplikim, ne kemi zgjedhur algoritmin Random Forest Classification. Arsyet se përse kemi zgjedhur këto algoritme janë dhënë më poshtë:

- Me caktimin e thellësisë së pemës, ne nuk jemi pronë e mbi vendosjes (ang. *overfitting*) për arsye se modeli nuk trajnohet aq shumë sa që të jetë shumë dinamik, e edhe nuk është pak dinamik sa të ketë një saktësi të ulët (nëse data seti ka një numër të konsiderueshëm të rreshtave).
- Data seti jonë përbëhet nga vlera numerike (siç do të shihet më poshtë) dhe algoritmi në fjalë është i saktë në predikim pasi gjykohet nga një numër pemësh vendim-marrëse e jo vetëm një soj.
- Nga të gjitha provat e algoritmeve tjera në data setin tonë, ky algoritëm ka rezultuar më i sakti dukshëm (me rreth 5% diferencë).

Pra në vazhdim do të përdoret ky algoritëm.

3 Hyrje në projekt

Me të përshkruar inteligjencën artificiale dhe algoritmet kryesore të klasifikimit si përfaqësues të të mësuarit me mbikëqyrje të makinës, tani mund të kalohet në përshkrimin e zgjidhjes së ofruar të problemit, si dhe të rezultateve të fituara.

Po fillojmë njëherë me përshkrimin e veglave softuerike që janë përdorur, në mënyrë që të kuptohet më vonë përdorimi i tyre varësisht rastit.

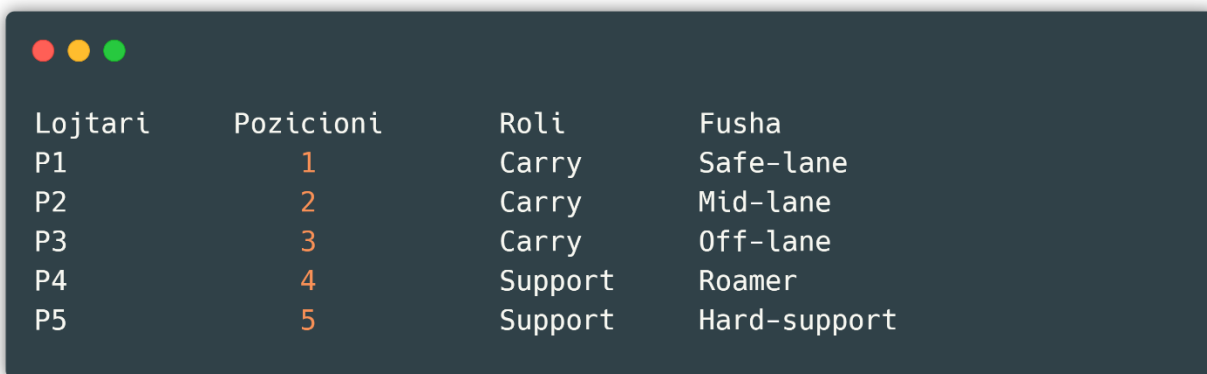
- *IntelliJ* – vegla për ndërtim (ang. *Integrated Development Kit*, IDE) në të cilin është shkruar aplikacioni, është ndërtuar (ang. *build*) dhe provuar (ang. *test*). Kjo vegël është e shkruar në gjuhën programuese Java dhe është zgjedhur ndër shumë tjera për arsye të organizimit të lartë, mundësive të shumta që ofron si dhe paketave të gatshme si për Scala ashtu edhe për Play.
- *Git* – platforma në të cilën është koordinuar ecuria e punës. Pasi kjo platformë dominon në tregun e punës, është parë e arsyeshme që të praktikohet edhe më shumë kjo mënyrë e punës. Funksionon nëpërmjet ndërfaqes komanduese (ang. *Command Line Interface*, *CL*).
- *Github* – platforma në të cilën *Git* vendos dosjet burimore. Kjo platformë në përgjithësi mbledh të gjitha projektet publike të shkruara nga çdo programues në botë dhe ia ofron kërkuarit në mënyrë që ato të kenë qasje edhe në kode burimore edhe në aplikacione që ndërtohen si ndihmesa, siç është edhe vegla e përdorur më poshtë *Ngrok*.
- *Postman* – vegla nëpërmjet së cilës janë kryer HTTP kërkesat (ang. *requests*) si *GET* dhe *POST* me atributet përkatëse. Falë kësaj vegle, ne kemi mundur të stimulojmë kërkesat në mënyrë lokale.
- *Ngrok* – vegla nëpërmjet së cilës janë publikuar qasjet e mundshme të pikave fundore. Kjo është nevojitur për arsye se nuk ka pasur nevojë që të kopjohet projekti në çdo ambient punues por shërbimet në të cilat projekti po ekzekutohet lokalisht ekspozohen në një URL publike nëpërmjet së cilës është realizuar pjesa e dytë e projektit. Për shembull, pika fundore, qasja e së cilës realizohet nëpërmjet lidhjes localhost:9000/getColumns?kind=steam ekspozohet në lidhjen <http://338ab558.ngrok.io/getColumns?kind=steam> dhe kjo lidhje mund të thërritet nga çfarë do burimi.
- *Sublime Text* – vegla me anë të së cilës janë shqyrtuar të dhënat me përmbajtje të madhe të strukturës JSON, nga edhe është bërë përzgjedhja e kolonave për klasifikim. Po ashtu, vegël e dobishme për kontrollin e vlefshmërisë së strukturës JSON (nëpërmjet tasteve CTRL + ALT + J).
- *Excel* – po ashtu vegël për të shikuar të dhënat e formatit *CSV*, në të cilin mund të aplikohen filtrime dhe kontrole të ndryshme.

- *Powershell* – vegla me anë të së cilës është menaxhuar imazhi Docker. Ndryshe nga CLI (Command Line Interface) të tjera, kjo vegël është më funksionale, më e pasur me funksione si dhe ka një ndërfaqe më të përdorshme.
- *Docker* – vegla me anë të së cilës është krijuar imazhi në mënyrë që ti publikojmë shërbimet e tona në internet të qasshme për këdo.

IntelliDota është një aplikacion i ndërtuar në Scala dhe Flutter që tenton të *klasteroj* (ang. *cluster*) dhe *klasifikoj* (ang. *cluster*) rezultatin e një video-loje që në rastin tonë është video-loja *Dota 2*. Do të emërtojmë me IntelliDota Classification pjesën e aplikacionit që kryen klasifikimin ndërsa IntelliDota Clustering pjesën e aplikacionit që kryen kllasterimin. Përpara se të vazhdojmë më tutje me aplikacionin, do të bëjmë një përshkrim të shkurtër se çfarë është kjo video-lojë dhe çfarë duam të predikojmë ne.

3.1 Loja kompjuterike Dota 2

Dota 2 qëndron për Mbrojtjen e Kullave (ang. *Defense of the Ancients*), pra dy ekipe prej pesë lojtarësh tentojnë të shkatërrojnë bazën kryesore të armikut. Një ekip, që përbëhet prej pesë lojtarëve, posedon një ndarje të një niveli më të lartë, pra mbajtësit (ang. *carries*) dhe mbështetësit (ang. *supports*). Në esencë, çdo *support* mbështet një *carry* (Fig 7).



Lojtari	Pozicioni	Roli	Fusha
P1	1	Carry	Safe-lane
P2	2	Carry	Mid-lane
P3	3	Carry	Off-lane
P4	4	Support	Roamer
P5	5	Support	Hard-support

Fig 7: Pamja e parë e formacionit të video-lojës Dota 2

Le të përmendim disa nga karakteristikat e secilit prej këtyre roleve. Pozicioni 1 shquhet për vrasje, asiste, nivel të lartë të parave, eksperiencës si dhe aktivitetit në lojë, por jo gjithmonë. Pozicioni 2 shquhet për shtyrje kullash, vrasje dhe krijues hapësire për pozicionin 1. Pozicioni 3 shquhet për aftësitë për të qëndruar i pari në linjë, nuk shkakton dëme të mëdha, por i gjithë sulmi i armikut bie mbi të. Dhe në fund janë pozicionit 4 dhe 5, ku të dy janë mbështetës por pozicioni 5 njihet për mjekim, për derisa pozicioni 4 është më mbështetës i lehtë. Secili lojtarë, pavarësisht rolit ka qindra statistika, por ndër më të thepisurat dhe më të rëndësishmet janë *leaver_status*, *gold_per_min*, *leaver_status*, *xp_per_min*, *deaths*, *tower_damage* etj. Lista e plotë mund të gjendet në kapitullin e mëposhtëm në të cilin realizohet predikimi postPredict, ku përshkruhet saktë funksioni dhe jepet përshkrimi se a është atributi i derivuar apo jo.

Në këndvështrim të lartë, ekipi organizohet në formacion të tillë (Fig 8):



Fig 8: Organizimi i ekipit

Pra, nëse mblidhen këto të dhëna, analizohen dhe aplikohen metrika në to, ne mund të fitojmë rezultate të ndryshme qoftë rreth lojës, apo edhe lojtarëve që luajnë këtë lojë.

Me algoritme të ndryshme, ne mund të shohim qartë se çfarë po luhet më shumë, çka po pëlqehet e si këto attribute bashkë me shumë të tjera ndikojnë në rezultatin e lojës. Ndër tërë këto aspekte, ne kemi vendos të predikojmë atributin fitorja e radiantëve (ang. *radiant_win*), që është asgjë më shumë se një nga këto kolona të mësipërme që identifikojnë një lojë të posaçme, me dallim e vetëm se kjo kolonë (si dhe disa të tjera) nuk i përket një lojtari konkret, por qëndron si një atribut që identifikon lojën në tërësi.

Nga këto të thëna, ne mund të vërejmë se në këto lojëra që duam të analizojmë, kemi dy lloje attributesh:

- Atributet që i përkasin lojtarëve, si shembull atributet *gold_per_min*, *xp_per_min* etj. apo
- Atributet që i përkasin lojës, që janë si shembull kohëzgjatja e lojës, atributi i së cilës emërohet *duration*, atributi *radiant_win* që aktualisht dëshirojmë të predikojmë, atributi *teamfights* e shumë të tjerë.


Duke bërë kombinimin e këtyre dy attributeve, ne mund të krijojmë një burim të dhënash me të dhëna të shëndosha në të cilin mund të aplikojmë algoritme inteligjente. Një të tillë ne e kemi realizuar bashkë me ndihmesën nga Ndërfaqja Publike e Aplikacionit Programorë Steam (ang. *Steam Public Application Programming Interface*, i njohur si *Steam Public API*).

Valve (kompania që ka zhvilluar këtë ueb-faqe) ofron këto *API*-në mënyrë që zhvilluesit të mund të përdorin të dhënat në mënyra interesante e të dobishme. Pra, zhvilluesit lejohen që të ekzekutojnë kërkesa e të realizojnë pyetësorë të ndryshëm. Për momentin lista aktuale është e kufizuar në vetëm disa video lojëra, por kjo listë pritet të zgjerohet shpejtë [8].

4 Koleksionimi i të dhënave për klasifikim dhe para procesimi

4.1 Pamja e parë e strukturës Json nga e cila duam të marrim të dhënat

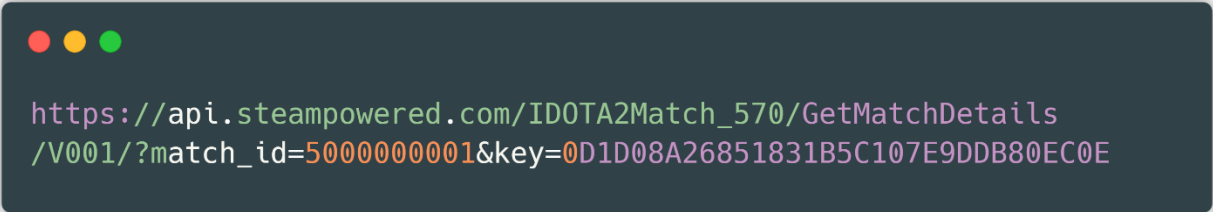
Si lidhje e burimit të të dhënave për klasifikim është përdorur, siç u tha edhe më lartë, ueb-faqja zyrtare e Valve, pra Steam nëpërmjet Steam Public API (<https://steamapi.xpaw.me>) [9]. Përpara se të qasemi në këtë faqe, duhet pasur një llogari në Steam (<https://steampowered.com>) në mënyrë që të fitojmë një çelës (ang. *Steam Key*) dhe një domain emër (ang. *domain name*) nga i cili na lejohej që të marrim të dhëna nga API publik i ofruar nga to [10]. Një dyshe e tyre duket kështu, dhe është vendosur në mënyrë që përdoruesi që përdor të ketë një limit të caktuar në marrjen (ang. *fetching*) e të dhënave (e zgjeruar më poshtë, Fig 9).



```
Key: 0D1D08A26851831B5C107E9DDB80EC0E
Domain Name: tema_diplomes_2019_lv
```

Fig 9: Dyshja çelës – domain

Me marrjen e këtyre kredencialeve ne mund të bëjmë një kërkesë të thjeshtë nëpërmjet lidhjes më poshtë (Fig 10).



```
https://api.steampowered.com/IDOTA2Match\_570/GetMatchDetails/V001/?match\_id=5000000001&key=0D1D08A26851831B5C107E9DDB80EC0E
```

Fig 10: Lidhja për një lojë të caktuar

ku si match id jepet vlera numerike e një loje me numër identifikues përkatës, që në rastin e mësipërm është 5000000001, dhe key është çelësi i gjeneruar më lartë.

Formati i të dhënave i reprezentuar nga lidhja e mësipërme është Json (Javascript Object Notation) dhe ky na lejon ruajtjen dhe transportimin e të dhënave [11]. Një format i vlefshëm i një loje gjendet nga numri 1 deri te numri 6 miliardë, pra, një match id do të ishte e vlefshme nga vlera match id 1 deri tek match id 6000000000.

Dihet që për të grumbulluar kaq shumë të dhëna, duhen vite e vite duke marrë parasysh që një video-lojë zhvillohet për rreth 45 minutash. E për një periudhë kohore kaq të gjatë, edhe struktura e të dhënave, që në rastin tonë është Json ndërron dukshëm, pasi gjithmonë është tentuar të krijohet një strukturë që është më përshkruese dhe më e kuptueshme se ajo paraprakja. Çfarë duam të themi është se struktura Json me match id 1 nuk është e njëjtë me

strukturën Json me match id 5 miliard. Për këtë arsye, ne kemi zgjedhur një interval në të cilin struktura Json nuk ndryshon dhe posedon aq shumë të dhëna sa që ne të mund të krijojmë një data set të mirëfilltë. Intervali që kemi zgjedhur ne është (Fig 11):

```
val START_ID      = 5000000000L
val END_ID        = 5999999999L
val FEEDS         = 50000
```

Fig 11: Intervali i përzgjedhur për të dhëna

Ku start id paraqet fillimin e intervalit që duam të marrim të dhënat dhe end id paraqet mbarimin e intervalit, gjatë këtij intervali, për shkak të automatizimit kemi vendosur që vlera e numrit të të dhënave të jetë dinamike, pra në rastin tonë FEEDS, që është 50.000.

Varësisht numrit FEEDS, aq të dhëna do të ketë data seti jonë. Do i referohemi me fjalën data set burimit tonë të të dhënave të cilin jemi duke e vetë-krijuar.

4.2 Para procesimi i të dhënave të mbledhura

Gjatë marrjes së të dhënave, kemi hasur në game id që nuk janë të përshtatshme apo janë të korruptuara në kuptimin që dëmtojnë algoritmin predikues apo nuk tregojnë të dhëna reale. Attribute të tilla ka vendosur vetë Steam kur ndodhin rrethana të caktuara në lojë (Fig 12).

```
Mos e proceso strukturën Json në rrethana të tilla:
1. Struktura përmban një atribut të quajtur 'error'.
2. Struktura nuk përmban atributin 'radiant_win',
   atribut ky që duam të predikojmë.
3. Atributi 'duration' është 0, do të thotë që loja nuk
   ka filluar fare.
```

Fig 12: Kushtet për anashkalim të video-lojës Dota 2

Me të përjashtuar këto raste, jemi siguruar që secila Json strukturë të jetë e vlefshme dhe të ketë të dhëna përshkruese, në mënyrë që algoritmi të jetë sa më i saktë.

Ky ka qenë rasti i vetëm në të cilin ka qenë nevoja për intervenim nga jashtë, përndryshe këto të dhëna nuk përmbajnë asnjë vlerë asgjëje (ang. *null*), vlerë jo-përshkruese apo të ndonjë lloji tjetër që do ta dëmtonte data setin tonë në çfarëdo mënyre.

Duhet cekur gjithashtu se të gjitha të dhënat e këtij data seti janë të formës së vazhdueshme, për këtë arsye ka qenë më se e nevojshme aplikimi i heqjes së të veçuarve (ang. *outliers*), ndërsa atributet *leaver_status* dhe *level* nuk janë prekur nga kjo procedure, ngase respektivisht njëri atribut është binarë ndërsa tjetri është diskret.

Meqenëse ekziston një kufi prej rreth 11.000 thirrjeve të njëpasnjëshme në lidhjen e mësipërme që është i paraparë ta ndaloj abuzimet dhe sulmet e ndryshme, ne e kemi ndarë procesin e dërgimit të kërkesave në 5 pjesë, ku secila pjesë përmban 10.000 thirrje të njëpasnjëshme që rezultojnë në pothuajse çerekun e data setit përfundimtarë. E më pas këto 5 pjesë i kemi bashkuar për të krijuar një data set të plotë, në të cilin po operojmë tani (Fig 13 Fig 12).

The diagram illustrates the process of combining five data sets into one. It shows five boxes labeled DS1, DS2, DS3, DS4, and DS5, each containing the text '10.000'. These boxes are connected by plus signs (+). Below this sequence, an equals sign (=) is followed by a box labeled 'DS' containing the text '(50.000 rreshta)', indicating the final combined data set.

Fig 13: Copëzat e data setit me rreshtat përkatës

Ky data set i plotë përmban 13 kolona dhe 50.000 rreshta. Se si është realizuar kjo procedurë do të tregohet më vonë pasi të hyjmë në pjesën e realizimit të konceptit përmes gjuhës programuese përkatëse

4.3 Metrika gjenerike për data setin e derivuar

Për të pasur më shumë njohuri rreth data setit, lidhjes dhe të dhënave në përgjithësi, aplikojmë disa nga metrikat e krijuara. Duhet pasur parasysh se të gjitha këto metrika po kryhen në një mostër me 1000 rreshta për arsye se procedimi në data setin e plotë do të kërkonte më shumë kohë.

Fillojmë me thirrjen fundore `getSample` nga e cila shohim një mostër të data setit tonë (Fig 14), nga shohim se kemi vlera numerike të vazhdueshme në shumicën e attributeve, përveç *leaver_status* dhe *level* që mund të konsiderohen vlera diskrete, e njëra konkretisht binare.

Meqë vetëm se kemi pasur njohuri paraprake rreth kësaj loje dhe se si funksionon ajo, përzgjedhja nuk ka qenë pjesa më e vështirë, ku në një strukturë JSON me rreth 65 attribute, ne kemi zgjedhur attribute që i përmban ky JSON file dhe attribute që ne kemi nxjerrë vetë, dhe ka rezultuar të jetë një veprim i suksesshëm.

```
[{
  "gold_per_min": 6355,    "level": 150,
  "leaver_status": 0,     "xp_per_min": 9056,
  "radiant_score": 63,    "gold_spent": 241485,
  "deaths": 61,          "denies": 40,
  "hero_damage": 267225,  "tower_damage": 10333,
  "last_hits": 974,       "hero_healing": 2316,
  "duration": 2991,       "radiant_win": 1
},
{
  "gold_per_min": 3300,    "level": 149,
  "leaver_status": 0,     "xp_per_min": 3810,
  "radiant_score": 46,    "gold_spent": 151995,
  "deaths": 48,          "denies": 49,
  "hero_damage": 219657,  "tower_damage": 25377,
  "last_hits": 1791,      "hero_healing": 375,
  "duration": 3306,       "radiant_win": 1
}]
```

Fig 14: Mostra e data setit

Shohim secilën kolonë dhe tipin përkatës nëpërmjet thirrjes `getSchema` dhe shohim se të gjitha kolonat janë të tipit numër (Fig 15). Pra, na lejojnë metriket numërore si krahasime, filtrime numërore e të tjera. Mos të harrojmë që në klasën *Match* secilit atribut i kemi shoqëruar tipin numër.

```
[{ "column": "gold_per_min", "type": "IntegerType" },
{ "column": "level", "type": "IntegerType" },
{ "column": "leaver_status", "type": "IntegerType" },
{ "column": "xp_per_min", "type": "IntegerType" },
{ "column": "radiant_score", "type": "IntegerType" },
{ "column": "gold_spent", "type": "IntegerType" },
{ "column": "deaths", "type": "IntegerType" },
{ "column": "denies", "type": "IntegerType" },
{ "column": "hero_damage", "type": "IntegerType" },
{ "column": "tower_damage", "type": "IntegerType" },
{ "column": "last_hits", "type": "IntegerType" },
{ "column": "hero_healing", "type": "IntegerType" },
{ "column": "duration", "type": "IntegerType" },
{ "column": "radiant_win", "type": "IntegerType" }]
```

Fig 15: Kolonat e data setit me tipet përkatëse

Nëse aplikojmë grupimin dhe njehsimin në *leaver_status*, nëpërmjet `getGroupAndCount` (parametër nënkuptohet që vendoset *leaver_status*) shohim se ndarja nuk është një ndarje e mirëfilltë, pra rreth 80% e të dhënave janë në *leaver_status* 0 dhe rreth 20% janë në *leaver_status* 1 (Fig 16).

```
[{ "leaver_status": 1, "count": 162 },  
 { "leaver_status": 0, "count": 838 }]
```

Fig 16: Popullimi i *leaver_status* në data set

Nëse aplikojmë metrikën tjetër, e cila është një thirrje e brendshme në të cilën duam të kontrollojmë se si qëndron raporti i *leaver_status* me kolonën të cilën ne duam të predikojmë, pra Tab 8: Pika fundore `getSchema`

`getDoubleGroup`, marrim (Fig 17):

```
[ { "leaver_status": 1, "radiant_win": 0, "count": 100 },  
   { "leaver_status": 1, "radiant_win": 1, "count": 62 },  
   { "leaver_status": 0, "radiant_win": 0, "count": 368 },  
   { "leaver_status": 0, "radiant_win": 1, "count": 470 }]
```

Fig 17: Raporti *leaver_status* me *radiant_win* ne data set

Pra, kur *leaver_status* nuk është zero do të thotë që loja nuk është përfunduar si ekip dhe si rrjedhojë të bardhët kanë humbur pothuajse 65% të lojërave.

Gjithsesi, se përzgjedhja e kolonave ka qenë e mirë, mund të vërtetohet nëpërmjet matricës së korrelacioneve, `getCorrelationMatrix`, një matricë e cila paraqet lidhshmërinë e kolonave ndaj njëra tjetrës.

Pra kemi edhe korrelacione, edhe korrelacione inverse, ku një raport i plotë i kësaj matrice është i bashkangjitur në *Github*. Le të shohim tash atributet tjerë, vetëm sa të marrim një parafytyrim se me çfarë të dhënash kemi të bëjmë.

- **Funksioni:** grupimi sipas kolonës *xp_per_min* në 3 ndarje
Rezultati: fitojmë një ndarje relativisht të mirë ku në intervalit 0 ~ 472 i takojnë 828 elemente, intervalit 472 ~ 4755 i takojnë 113 elemente ndërsa intervalit 4755 e tutje i takojnë rreth 60 elemente (Fig 18).

```
[{ "bucket": "1.0", "count": "828", "border": "472.0" },
{ "bucket": "2.0", "count": "113", "border": "4755.0" },
{ "bucket": "3.0", "count": "59", "border": "9038.0" }]
```

Fig 18: getGroupAndCount në 3 ndarje sipas xp_per_min

- **Funksioni:** grupimi sipas kolonës *level* në 4 ndarje
Rezultati: edhe njëherë vërtetohet se data seti jonë është i shëndetshëm, ngase është lehtë e kuptueshme nga lojtarët e zakonshëm të kësaj loje se pjesa dominuese e lojërave mbaron me të gjithë lojtarët me *level* maksimal, që vërtetohet nga ky diagram (Fig 19).

```
[{ "bucket": "1.0", "count": "17", "border": "4.0" },
{ "bucket": "2.0", "count": "26", "border": "42.0" },
{ "bucket": "3.0", "count": "235", "border": "80.0" },
{ "bucket": "4.0", "count": "722", "border": "118.0" }]
```

Fig 19: getGroupAndCount në 4 ndarje sipas level

- **Funksioni:** grupimi sipas kolonës *hero_healing* në 3 ndarje
Rezultati: automatikisht krijohet edhe një ndarje e nivelit 0, ngase ka popullim të shumtë. Përndryshe, kjo ndarje na len të kuptohet se ekzistojnë dy grupe kryesore, ato që shërojnë dhe ato që nuk shërojnë (Fig 20).

```
[{ "bucket": "1.0", "count": "973", "border": "0.0" },
{ "bucket": "2.0", "count": "22", "border": "20812.0" },
{ "bucket": "3.0", "count": "4", "border": "41624.0" },
{ "bucket": "4.0", "count": "1", "border": "62436.0" }]
```

Fig 20: getGroupAndCount në 3 ndarje sipas hero_healing

- **Funksioni:** grupimi sipas kolonës *denies* në 4 ndarje
Rezultati: krijohet një ndarje në 5 grupime, për arsye se edhe niveli 0 konsiderohet. Mund të shihet se intervali kryesorë qëndron në mes numrave 0 – 100. Për derisa një pjesë e konsiderueshme gjendet në tri grupimet e para. Po ashtu këto të dhëna mund të thuhet se janë diskrete (Fig 21).

```
[{ "bucket": "1.0", "count": "215", "border": "0.0" },
{ "bucket": "2.0", "count": "488", "border": "31.0" },
{ "bucket": "3.0", "count": "268", "border": "62.0" },
{ "bucket": "4.0", "count": "28", "border": "93.0" },
{ "bucket": "5.0", "count": "1", "border": "124.0" }]
```

Fig 21: getGroupAndCount në 4 ndarje sipas denies

Grupimet e realizuara më sipër nuk janë praktikë e mirë pasi numri i kategorizimeve është shumë i vogël e nuk mund të nxirren të dhëna të shëndosha por është realizuar në këtë mënyrë për arsye të prezantimit (hapësirës), përndryshe, nëse dëshirojmë të nxjerrim të dhëna analitike rreth kolonave përkatëse është shumë më e arsyeshme të bëhen me dhjetëshe.

Pra, në total kemi zgjedhur 14 attribute nga disa dhjetëra (Fig 22).

```
Atributet e zgjedhura nga struktura Json:
- radiant_score, duration dhe radiant_win.

Atributet e derivuara nga struktura Json:
- gold_per_min, level, leaver_status, xp_per_min,
  gold_spent, deaths, denies, hero_damage,
  tower_damage, last_hits dhe hero_healing.
```

Fig 22: Atributet e zgjedhura nga struktura

Atributet e strukturës janë ato attribute të cilat nuk ka pasur nevojë të mëtutjeshme për përpunim përpara se të futen në data set, ndryshe nga atributet e nxjerra të strukturës, të cilat i kemi përbë nga copëza të vogla.

Duhet cekur se edhe atributet e përbëra janë përsëri attribute të strukturës Json, por jo që mund t'i qasemi në mënyrë të drejtpërdrejtë. Më poshtë po përshkruajmë se çfarë duam të themi me qasje të drejtpërdrejtë.

Forma e strukturimit të një skeme Json paraqitet më poshtë (Fig 23). Shihet se si atribut kryesorë është *result*, q konsiderohet prind për secilin atribut tjetër. Fëmijët e këtij atributi mund të kategorizohen në dy grupe: atributet me nën attribute siç është *players* i cili përbëhet nga 10 objekte, secili objekt me veçori përkatëse për një lojtarë specifik.

```
{  "result": {
    "players": [], //10 lojtarë
    "radiant_win": true,
    "duration": 1469,
    "start_time": 1567328090}}
```

Fig 23: Struktura e një objekti Json

Këto objekte i kemi agreguar me shumë në çelësa përkatës dhe kemi krijuar të dhëna për data setin tonë të ri. Ndryshe nga ky lloj i attributeve, kemi edhe atributet vlerat e të cilëve mund t'i marrin direkt, pa agregime apo përpunime të mëtutjeshme. Operimi me këto attribute është shumë më i lehtë sesa me atributet të cilat duhet derivuar apo përpunuar, por ky numër qëndron në raport 2/8 ndaj attributeve që duhet përpunuar, numër ky i vogël.

Manovrimet dhe manipulime me data setin aktual i kemi kryer për arsye se Scala me libraritë e saj ofron mundësi shumë të mira për të kaluar të dhënat nga skema Json për në klasën tonë Match, përshtatje kjo që realizohet me kod minimal e konciz, pra në njëfarë mënyre na është imponuar një mënyrë e tillë e punës. Le të shohim se si i kemi realizuar dhe organizuar të dhënat dhe kodin në projektin aktual në Scala.

4.4 Krijimi i data setit – Mënyra e Scalas

Përpara se të vazhdojmë me shpjegimin e projektit, duhet zgjeruar në pika të shkurtra konceptin Gson.

Gson [12] është një librari e gjuhës programuese Java që ndër shumë funksione tjera, na lejon konvertimin e Java objekteve në reprezentimin përkatës të strukturës JSON. Meqë Java ekzekutohet në Makinën Virtuale Java (ang. *Java Virtual Machine, JVM*), do të thotë që edhe Scala mund të përdorë po këtë librari [13].

Në rastin tonë, ne kemi përdorur Gson që të mund të kalojmë të dhënat nga struktura JSON në një Scala klasë situate (ang. *Scala case class*). Një *case class* na lejon që të definojmë modelin apo strukturën e data setit tonë, ku një rresht i data setit është një objekt i ri i kësaj klase. Gson na është mundësuar pasi kemi shtuar vartësinë në përbërjen e projektit si *gson* nëpërmjet lidhjes [com.google.code.gson](https://mvnrepository.com/artifact/com.google.code.gson/com.google.code.gson/2.8.5), versioni i të cilit është 2.8.5.

Për të shpjeguar më mirë deklarinimin e më sipërm, përdorim një shembull. Le të themi se kemi një JSON skemë me një atribut të vetëm të quajtur atributi, vlera e të cilit është 10 dhe krijojmë një *case class* me emrin Klasa me një atribut të quajtur atributi dhe e përdorim metodën *fromJson* atëherë na krijohet një objekt i ri i klasës Klasa me variabël atributi që ka vlerën 10.

Kjo është një praktikë shumë e mirë dhe e përdorshme për të krijuar klasa në mënyrë automatike, e një seri e gjatë e këtyre klasave rezulton në një data set.

Modeli jonë, i quajtur Ndeshja (Fig 24) përbëhet nga 14 attribute, e secili nga këto attribute duhet gjithsesi të figurojë edhe në strukturën JSON të marrë nëpërmjet lidhjeve të cekura më sipër (nëpërmjet Steam API). Secilit atributit i përket tipi numër (ang. *Integer, Int*).

```
case class Match
(
  gold_per_min: Int,      level: Int,
  leaver_status: Int,     xp_per_min: Int,
  radiant_score: Int,     gold_spent: Int,
  deaths: Int,            denies: Int,
  hero_damage: Int,       tower_damage: Int,
  last_hits: Int,         hero_healing: Int,
  duration: Int,          radiant_win: Int
)
```

Fig 24: Modeli i klasës Match

Siç u cek edhe më lartë, kjo klasë shërben vetëm për të krijuar një objekt të ri për secilin rekord të sjellë nëpërmjet lidhjeve të realizuara në klasën kryesore në intervalet e caktuara. Objekti i krijuar i klasës Match më pas ruhet në një sekuencë. Sekuenca paraqet një koleksion të Scalas që me importimin e një funksioni të caktuar të librarisë Spark, i lejohet konvertimi në një data set dhe anasjelltas. Meqë data seti është i pa ndryshueshëm, neve na nevojitet që të krijohet data seti komplet në një sekuencë e më pas ta konvertojmë. Gjatë kësaj procedure, ne gjurmojmë shumën e objekteve të sekuencës në mënyrë që nëse arrin një numër të caktuar të ndaloj, pasi është arritur madhësia e data setit të dëshiruar. Ky numër në kodin tonë është konstantja *FEEDS*. Funksionet e Spark i marrim nga vartësitë *spark-core* dhe *spark-sql* nga *org.apache.spark*, versioni i së cilës është në këtë projekt 2.4.4.

Kjo mënyrë e përshkrimit është e nivelit të lartë, ndërsa më poshtë (Fig 25) përshkruhet saktësisht rrjedha e programit, variablat e përdorura dhe logjika kryesore e krijimit të një rekordi të vlefshëm. Fillojmë me krijimin e një lidhje të saktë në mënyrë që të kryejmë kërkesën. Kjo lidhje është krijuar nëpërmjet tre stringjeve: *steam api*, *gameld* dhe *steam key*, ku *steam api* dhe *steam key* janë konstanta ndërsa *gameld* ndryshon varësisht intervalit të mëposhtëm, që fillon me *start id* dhe mbaron me *end id*.

```

var matches = Seq[Match]() // sekuencë e klasës Matches

for (gameId <- START_ID to END_ID) {
    val game = Fetcher.fetchGames
        (steam_api + gameId + steam_key, args)

    if (game != null) {
        matches = matches :+ game
        foundGames += 1

        println(foundGames+". Analyzing game ["+gameId+"]")
    }

    if (foundGames == FEEDS) break
}

```

Fig 25: Unaza kryesore e aplikacionit

Args paraqet një varg që i kemi dhënë vlerat përmes *argumenteve programore* (ang. *program arguments* [14] të IntelliJ , vegla softuerike me të cilin është përpunuar aplikacioni), e ky varg përmban të gjitha vlerat e përfituara të cilat janë *gold_per_min*, *level*, *leaver_status*, *xp_per_min*, *gold_spent*, *deaths*, *denies*, *hero_damage*, *tower_damage*, *last_hits* dhe *hero_healing* (si më lartë)

Krahas këtyre informatave, shihet edhe objekti me metodën përkatëse *Fetcher.fetchGames* (Fig 26), metodë statike (ang. *static method*) kjo e cila na ndihmon që të mbledhim të dhënat. Në këtë metodë definohet kërkesa (ang. *request*) e cila mundëson qasjen në Steam Public API, kontrollohet se a është kërkesa e vlefshme, dhe nëse po, atëherë kthehet (ang. *returns*) një instancë e re e klasës *Match* me të gjitha atributet e specifikuar sipër të marrura nga *Json* struktura në të cilën është bërë kërkesa. Mundësia për kërkesa na është bërë e mundshme pasi të shtojmë *requests* nga *com.lihaoyi*, versioni më i ri i së cilës është **0.1.8**.

Ky objekt më pas shtohet tek sekuenca e quajtur *matches* dhe kështu përsëritet procesi deri sa të formulohet një sekuencë me 500.000 të dhëna.

```

def fetchGames(api: String, args: Array[String]): Match = {
  val responseAsJSON = requests.get(api)
  if (responseAsJSON.statusCode != 200) return null

  val preparedGames = Derivator.prepareGame
    (responseAsJSON.get("players").getAsJSONArray, args)

  gson.fromJson(responseAsJSON, classOf[Match])
}

```

Fig 26: Struktura e metodës fetchGames

Pas këtij procesi, krejt çka mbetet është konvertimi në një *Spark Dataframe* dhe ruajtja e dosjes në mënyrë të pastër (ang. *cleaned-version*) në mënyrë lokale (Fig 27).

```

val gamesDF = matches.toDF


gamesDF.write.format("csv").option("header", true)
  .mode("overwrite")
  .save(System.getenv("fetched_steam_data"))

```

Fig 27: Konvertimi dhe ruajtja e data setit në makinën lokale

Ku krijohet një dosje e re në shtegun (ang. *path*) përkatës, i cili është definuar po ashtu nëpërmjet *IntelliJ* - variablat e mjedisit (ang. *environment variable*). Ky përcaktim lehtëson punën në grup, ku ekipi punues në vend që të ndryshojë vlerat e variablave statike në mjedisin lokal, ndryshon vetëm variablat e mjedisit.

Pra, aktualisht kemi në makinën tonë lokale një *data set* të gatshëm dhe të pastër, në të cilin mund të aplikojmë metrika të ndryshme e edhe algoritme të mençura. Një mostër e marrjes dhe analizimit të të dhënave duket si më poshtë (Fig 28).



```
8. Analyzing game [5000000018]
9. Analyzing game [5000000019]
10. Analyzing game [5000000020]
```

Fig 28: Pamja e marrjes së të dhënave

4.5 Algoritmi i klasifikimi në Scala

Me krijimin e data setit, ne tashmë jemi në gjendje të bëjmë një predikim në atributin *radiant_win*.

Si hap i parë, normalisht që importohet data seti nëpërmjet librarisë Spark (Fig 29).

```
var dataframe = spark.read
    .option("header", true)
    .option("inferSchema", true)
    .csv(System.getenv("fetched_steam_data"))
```

Fig 29: Leximi i data setit të krijuar

Më pas duhet riemëruar atributi *radiant_win* në etiketë (ang. *label*) pasi mënyra se si algoritmi funksionon është ashtu që si hyrje duhet të ketë një atribut të vetëm, të quajtur cilësitë (ang. *features*) kurse atributi që predikohet duhet quajtur *label*.

Përveç kësaj cilësie, një data seti me të dhëna të vazhdueshme (ang. *continual*) është gjithmonë e arsyeshme të përdoret një shkallëzues (ang. *scaler*) [15] bashkë me një metodë për largimin e të veçuarve, në mënyrë që algoritmi të mos dominohet nga një atribut i vetëm.

Si një veçori e re që i është shtuar Sparkut janë fazat (gypat) (ang. *pipelines*). Por çfarë saktësisht na ofron neve kjo veçori? Në vend që të aplikohen këto tre hapa në mënyrë të veçantë, ne mund t'i vendosim në një fazë dhe të aplikojmë algoritmin në fazën kryesorë e jo në një hap të posaçëm (Fig 30).

```

val assembler = new VectorAssembler()
    .setInputCols(args).setOutputCol("non-scaled")
val scaler = new StandardScaler()
    .setInputCol("non-scaled").setOutputCol("features")
    .setWithStd(true).setWithMean(true)
val algorithm = new RandomForestClassifier()
    .setLabelCol("label").setFeaturesCol("features")
    .setNumTrees(10)
val pipeline = new Pipeline()
    .setStages(Array(assembler, scaler, algorithm))

```

Fig 30: Renditja e fazave sipas realizimit

Shohim se kemi tre funksione kryesore:

- *vecAssembler_id* – qëndron për *VectorAssembler* dhe shërben që një varg të kolonave të grumbullojë në një atribut të vetëm, të quajtur në rastin tonë të pa shkallëzuarat (ang. *non-scaled*). Pra, si kolona hyrëse (ang. *inputColumns*) pranon kolonat që duam t'i trajnojmë dhe si kolonë dalëse (ang. *outputColumn*) është kolona e re e krijuar *non-scaled*.
- *stdScal* – qëndron për *standardScaler* dhe shërben që të kthejë të gjithë numrat relativisht në një interval të caktuar, në rastin tonë ndërmjet 0 dhe 1. Pra, secila kolonë ka vetëm vlera të transformuar në intervalin 0 – 1. Përmban dy metoda statike që janë *withMean* dhe *withStd*. E para përgatitë të dhënat duke ua marrë vlerën mesatare para se ti shkallëzoj, ndërsa e dyta cakton devijimin standard. Gjithsesi, pesha e kolonave nuk ndryshon pavarësisht këtyre mutacioneve.
- *rfc* – qëndron për *Random Forest Classifier* (i përshkruar më lartë), ofron metodat përkatëse me anë të së cilave mund të trajnojmë dhe aplikojmë metrika në të dhëna të reja.

Pra, nëse bëjmë një përmbledhje të shkurtër rreth hapave të një data setit që kalon deri në predikim; atributet që shërbejnë në predikim bëhen një dhe riemërohen si *të pa shkallëzuara* (ang. *non-scaled*), ky atribut më pas shkallëzohet dhe riemërohet në *features*, e në fund aplikohet algoritmi *Random Forest Classifier* ku fitohet një data set i cili përmban një kolonë me shumë se parapraket të quajtur *prediction*, vlera e të cilit është ajo çfarë ne dëshirojmë.

Fazat jo vetëm që lehtësojnë procedurën, por edhe bëjnë kodin më konciz, të lexueshëm dhe më të manovrueshëm. Një vizualizim i gjërave që u thanë më sipër duket kështu (Fig 31).

```
args                => Array("non-scaled")
Array("non-scaled") => scale    => Array("features")
Array("features")   => classify => Array("label")
```

Fig 31: Vizualizimi i thjeshtë i fazave

Saktësia e algoritmit tonë me një ndarje të të dhënave 3/10 aktualisht sillet deri 87%, siç shihet edhe më poshtë (Fig 32):

```
Random forest classifier accuracy: 87.29641693811075.
```

Fig 32: Saktësia e algoritmit Random Forest Classifier

Si metrika për vlerësim kemi përdorur atë të librisë Spark që quhet *accuracy* e cila realizohet nëpërmjet klasë *MulticlassClassificationEvaluator* që përdor dy metoda kryesore që shërbejnë për marrjen e predikimeve aktuale dhe krahasimin me ato reale. Kthen probabilitetin e saktësisë, pra një numër prej 0 deri 1.

Më pas ruajmë modelin e trajnuar lokalisht në këtë mënyrë (Fig 33).

```
val model = pipeline.fit(train)
model.write.overwrite.save
  (System.getenv("classified_model"))
```

Fig 33: Ruajtja e modelit të trajnuar në makinën lokale

Përparësi e madhe e këtij veprimi është se nuk ka nevojë që të trajnohet modeli për secilën të dhënë që duam të predikojmë, por algoritmi ngarkon algoritmin e trajnuar dhe aplikon veprimin e caktuar në të, që kryhet në disa mili sekonda.

Përndryshe, nëse nuk do të ruanim algoritmin e trajnuar, do të duhej ta trajnonim atë për secilin predikim, që për data setin tonë, i duhen përafërsisht 5-6 minuta. E kjo do të rezultonte në eksperiencë të tmerrshme për këdo që tenton të aplikoj metrika në të.

4.6 Largimi i të veçuarve

Siç edhe kemi cekur më lartë, për arsye të vlerave të vazhdueshme, mundësia e paraqitjes të *outliers* është e madhe andaj është mirë që të largohen të dhënat me këto veti përpara aplikimit të algoritmit *Random Forest*.

Për këtë pjesë ne kemi përdorur metodën për intervale ndër kuartile (ang. *interquartile range*). Kjo metodë i është aplikuar secilës kolonë të data setit tonë, e metoda funksionon në atë mënyrë që së pari radhiten të dhënat në varg rritës dhe merren medianat e gjysmës së parë dhe gjysmës së dytë. Le të themi se Q1 është gjysma e medianës së parë ndërsa Q3 gjysma e medianës së dytë. Gjejmë diferencën në mes këtyre dy vlerave, $IQR = Q3 - Q1$, vlerë kjo e quajtur vlera ndër kuartile (nga edhe e ka marrë emrin) dhe ia shtojmë përkatësisht zbrësim vlerës Q3 dhe Q1, pra, $A = Q3 + 1.5 * IQR$ dhe $B = Q1 - 1.5 * IQR$ ku 1.5 është një vlerë rritëse apo zvogëluese e intervalit që duam të marrim. Secilën vlerë në data set më të vogël se B dhe më të madhe se A i largojmë (Fig 34).

```
5   9   2   1   3   7   6   5   8   // vargu i dhënë
1   2   3   5   5   6   7   8   9   // rradhitja rritëse
                        |           // mediana
      (2 + 3) / 2      (7 + 8) / 2   // Q1 dhe Q3
IQR = Q3 - Q1 = 7.5 - 2.5 = 5
A = Q1 - 1.5 * IQR = 2.5 - 1.5 * 5 = -5
B = Q3 + 1.5 * IQR = 7.5 + 1.5 * 5 = 15
```

Fig 34: Shembull i gjetjes e një intervali ndër kuartile

5 Klasifikimi në kohë reale dhe ruajtja si model i trajnuar

5.1 Struktura e aplikacionit

Siç e kemi cekur edhe më lartë, arkitektura në të cilin Play është ndërtuar është MVC, pra *Model*, *View* dhe *Controller*. Në rastin tonë, për arsye të integritimit të aplikacionit me Flutter, kemi çkytur komponentin *View*, ndërsa komponenti *Model* nuk është përdorur pasi nuk kemi nevojë për lidhje me bazën e të dhënave, pasi të dhënat po i ruajmë në mënyrë lokale. Diçka si komponenti *model* mund të konsiderohet klasa *Match* e përmendur më lartë, që paraqet një strukturë në të cilën kalohen informatat JSON.

Pra, ne aktualisht përdorim një kontrollor dhe një pako (ang. *package*) të quajtur veglat (ang. *utilities*) nëpërmjet së cilës ekzekutojmë metoda të shumta (Fig 35).

```
controllers ->
    MainController
utilities    ->
    Dataset
    Pre
    Statistics
```

Fig 35: Struktura e projektit në Scala

MainController paraqet kontrollerin kryesorë, që drejton secilën kërkesë në rrugën e duhur. Pra, secila metodë me rrugën (ang. *routing*) përkatëse menaxhohet nga ky kontrollor.

Pre (Fig 36) paraqet po ashtu një objekt në të cilin ekzekutohen metodat globale për Spark dhe për ngarkim të data seteve *steam* dhe *Kaggle*.

```
object Pre {
    def spark(appName: String, master: String) = {}

    def dataframe(spark: SparkSession, path: String) = {}
}
```

Fig 36: Paraqitja e objektit Pre

Dataset paraqet një objekt të Scalas që duket kështu (Fig 37) në të cilin janë të organizuara të gjitha metodat me parametrat të kontrollërit përkatës *MainController*. Secila rrugë e kontrollërit pra ekzekuton njërin nga metodat e lartë cekura.

```
object Dataset {  
  def getStages(path: String)  
  def getColumns(dataframe: DataFrame)  
  def getSample(dataframe: DataFrame, percentage: Double)  
  def getCorrelationMatrix(dataframe: DataFrame)  
  def getPredictedModel(path: String)  
  def getStats(dataframe: DataFrame)  
  def getRawStats(spark: SparkSession, path: String)  
  
  def predict(spark: SparkSession,  
              dataframe: DataFrame, s: Seq[Int])  
}
```

Fig 37: Paraqitja e objektit Dataset

Dhe në fund *Statistics* përmban metodat e kontrollërit *MainController*, konkretisht metodat *getGroupByAndCount* dhe *getBinary*. Meqë kompleksiteti i kësaj metode është më i madh, atëherë ka qenë nevoja që të krijohet një klasë e re për funksionalizim.

5.2 Realizimi i klasifikimi në kohë reale

Rikujtojmë se ne gjatë programit tonë, kemi ruajtur modelin e klasifikuar. Çfarë do të thotë kjo?

Dihet që data seti duhet trajnuar, e më pas të testohet me të dhëna testuese, në rastin tonë, patëm një saktësi rreth 87% me të dhëna trajnuese. Por për një aplikacion normal, të dhënat trajnuese shërbejnë vetëm si provë (ang. *testing*) për të treguar se a ja vlen algoritmi jonë të sprovohet me të dhëna reale apo jo.

Algoritmi jonë ka rezultuar shumë i suksesshëm me të dhëna trajnuese të të njëjtit burim, por le të provojmë të testojmë me një varg të dhënash krejt të një natyre tjetër.

Në aplikacionin tonë ofrojmë edhe një thirrje fundore të quajtur *getStages*, thirrje kjo që na ndihmon të kujtojmë parametrat dhe funksionet që kemi përdorur në algoritmin tonë, thënë ndryshe, mundësohet zgjatja e fazave të planifikuara varësisht algoritmit (Fig 38).

```

{ "vecAssembler_4d7620a503d8": [
  { "name": "inputCols", "value": "[Ljava.lang]" },
  { "name": "outputCol", "value": "non-scaled" }],
  "stdScal_3eb70c027ef3": [
    { "name": "inputCol", "value": "non-scaled" },
    { "name": "outputCol", "value": "features" },
    { "name": "withMean", "value": "true" },
    { "name": "withStd", "value": "true" }],
  "rfc_ba39ad829e83": [
    { "name": "featuresCol", "value": "features" },
    { "name": "labelCol", "value": "label" },
    { "name": "numTrees", "value": "10" }]]}

```

Fig 38: Fazat të reprezentuar si JSON objekte – data seti 1

ku shihen emrat e klasave (për shembull *vecAssembler_id*) dhe metodave të përdorura me vlerat përkatëse (për shembull *outputCol* me vlerën *non-scaled*).

E nëse krahasojmë me funksionin origjinal, gjithçka merr kuptim, pasi emrat e mësipërm përkasin komplet me metodat e shkruara më lartë. Si shembull po ilustrojmë edhe një rast tjetër, ku është përdorur algoritmi K-Means me parametra të caktuar (Fig 39).

```

{ "vecAssembler_025011705475": [
  { "name": "inputCols", "value": "[Ljava.lang]" },
  { "name": "outputCol", "value": "features" }],
  "kmeans_dda0042efb74": [
    { "name": "k", "value": "6" },
    { "name": "maxIter", "value": "25" }]]}

```

Fig 39: Fazat të reprezentuar si JSON objekte – data seti 2

Ku në bazë të këtij rasti, mund të themi se është përdorur klasa *vecAssembler* dhe klasa *kmeans* me metodat statike me metodat e tyre përkatëse (për shembull *name* me vlerën 6). Kjo është realizuar si në vijim (Fig 40).

```

val assembler = new VectorAssembler()
    .setInputCols(elements)
    .setOutputCol("features")
val kmeans = new KMeans()
    .setK(6)
    .setMaxIter(25)

```

Fig 40: K-Means në Scala

Kalojmë në pjesën e predikimit ku duhet të krijojmë një kërkesë për aplikacionin tonë, përgjigjja e të cilit do të na tregoj rezultatin në formatin JSON (Fig 41).

```

localhost:9000/postPredict?
gold_per_min=2585& level=139& leaver_status=0&
xp_per_min=3658& radiant_score=34& gold_spent=89220&
deaths=65& denies=21& hero_damage=170629& tower_damage=2732&
last_hits=901& hero_healing=10700& duration=2549

```

Fig 41: Forma e predikimit POST

Nga mund të shihet se attributeve gold_per_min, level, leaver_status, xp_per_min, radiant_score, gold_spent, deaths, denies, hero_damage, tower_damage, last_hits, hero_healing dhe duration ua kemi shoqëruar vlerat përkatëse numerike. Dihet që numri i attributeve në lidhje është i njëjtë me numrin e attributeve që pranon pika fundore. Ekzekutojmë thirrjen dhe na vjen përgjigjja përkatëse (Fig 42).

```

"probability": {
  "type": 1,
  "values": [
    0.9838030841661979,
    0.016196915833802155
  ]
},
"prediction": 0.0

```

Fig 42: Rezultati i POST kërkesës për predikim

Nëse analizojmë këtë përgjigje, shohim se na kthehet një çelës (ang. *key*) me një tip (ang. *type*) që paraqet llojin e algoritmit që në rastin tonë është klasifikim, një listë vlerash që paraqesin probabilitetin dhe vetë predikimi.

Pra, algoritmi ka klasifikuar vlerat hyrëse nga ne me 0 me një probabilitet 0.983 që i bie 98%, një saktësi shumë e lartë.

Ndërsa se çfarë ndodhë në prapa skenë kur realizohet thirrja nëpërmjet lidhjes më sipër është se krijohet një Spark RDD – që paraqesin një formë tjetër të koleksioneve më primitive sesa data setet (Fig 43).

```
val RDD = spark.sparkContext.makeRDD(list)
```

Fig 43: Krijimi i RDD nga lista e attributeve hyrëse

e më pas simulohet krijimi i një data seti me kolonat hyrëse që janë dhënë nëpërmjet pjesës hyrëse (*frontend*) nga RDD (Fig 44)

```
val df = spark.createDataFrame(RDD, StructType(columns))
```

Fig 44: Krijimi i data setit nga RDD

tani, *df* paraqet data setin tonë me një rresht. Me të krijuar këtë data set, mundemi që ta aplikojmë në modelin e trajnuar më herët (Fig 45).

```
getPredictedModel(sys.props.get("classified_model").get)  
    .transform(df)
```

Fig 45: Modeli i trajnuar dhe klasifikimi në kohë reale

Ku *getPredictedModel* kthen modelin e trajnuar në shtegun e caktuar të quajtur *classified_model* dhe përmes metodës *transform*, realizohet klasifikimi në kohë reale dhe kthehet përgjigjja në formën JSON.

6 Përgatitja e imazhit për ekspozimin e shërbimeve

6.1 Çka është Docker?

Përpara se të vazhdojmë më tutje, duhet të përshkruajmë se çka është Docker dhe si funksionon ai. Docker është një vegël që ndihmon në krijimin, zhvillimin dhe publikimin e aplikacionit në ueb. Këtë e bën duke paketuar softuerin tonë në paketa të quajtur kontejnerë. Kontejnerët janë të izoluar nga njëri tjetri, që do të thotë se posedojnë pavarësi të plotë. Motivimi për Docker, ndër shumë arsye tjera, ka ardhur nga ajo se për tu ekzekutuar programi nga dy sisteme të ndryshme është shumë më e lehtë të krijohet një Docker imazh që mund të ekzekutohet nga pala tjetër sesa pasimi i kodit burimorë. Arsyeja tjetër dhe kryesore është ajo se për të publikuar një sistem në ueb, që në rastin ka qenë publikimi i API të krijuara nga pjesa backend, Docker është vegla më e përshtatshme, sidomos kur aplikacioni është i shpërndarë në disa pjesë, si në rastin tonë, pjesa e data seteve, modelet e trajnuara për klasifikim dhe kllasterim e të tjera.

Po përshkruajmë më poshtë hapat kryesorë që kemi marrë deri në krijim e një imazhi si të lartcekur.

6.2 Krijimi dhe shfletimi i kontejnerit

Si hap i parë që duhet të kryejmë është të instalojmë Docker nga faqja zyrtare. Çdoherë kur krijojmë një imazh Docker, duhet të ekzistojë një tekst i quajtur Dockerfile në direktoriumin e projektit kryesorë. Në rastin tonë, ne e kemi krijuar këtë dosje në direktoriumin rrënjë të projektit (ang. *root directory*).

Duhet konfiguruar Dockerfile ashtu që ekzekutimi i tij, të mundësoj krijimin e imazhit tonë. Përmbajtja e Dockerfile tonë është si vijon (Fig 46):

```
1 FROM aa8y/sbt:0.13.18
2
3 USER root
4
5 RUN apk update && apk add --no-cache libc6-compat
6
7 COPY ./target/universal/scalatra-1.0.zip .
8
9 RUN unzip scalatra-1.0.zip
10
11 COPY ./main_route .
12
13 CMD scalatra-1.0/bin/scalatra -Dhttp.port=${PORT}
```

Fig 46: Pamja e parë e Dockerfile

ku rreshti numër 1 paraqet imazhin kryesorë dhe ambientin në të cilin do të ekzekutohet aplikacioni. Meqë projekti jonë është i ndërtuar në Scala dhe si bazë për ndërtim (ang. *build*) ka SBT-në, atëherë ne përdorim po të njëjtin imazh me po të njëjtin version si projekti, pra 0.13.18. Më pas, caktojmë përdoruesin si *root* (hapi 3) në mënyrë që të kemi qasje administrative. Duhet theksuar se ky imazh nuk është një imazh zyrtarë nga Docker por i krijuar nga një person apo persona të caktuar, që në rastin tonë është zhvilluesi *aa8y*. Pra, ne po krijojmë një imazh nga një imazh tjetër që po përdorim si mjedis zhvillues në imazhin tonë.

Përditësojmë ambienton tonë dhe shtojmë *libc6-compat* që mundëson që libraritë të mos kenë konflikte dhe siguron që aplikacioni të mos ketë gabime gjatë ekzekutimit (hapi 5).

Kopjojmë shtegun `./target/universal/scalatra-1.0.zip` në direktoriumin bazë të imazhit tonë që nuk e kemi krijuar akoma (hapi 7). Shtegu i lartcekur mundësohet nga *sbt shell* me anë të komandës *dist* që krijon një aplikacion të kompresuar gati për publikim, pra me librari dhe vartësi të gjitha të gatshme dhe të paketuara në të.

Pas kopjimit të dosjes së projektit në direktoriumin bazë të imazhit, e dekompresojmë atë (hapi 9). Ky hap duhet kryer në mënyrë që të kemi dosjen e hapur për ekzekutim.

Më në fund kopjojmë dosjen tonë lokale *main_route* dhe e vendosim po ashtu në direktoriumin bazë të imazhit Docker (hapi 11). Rikujtojmë se *main_route* paraqet dosjen në të cilën janë të vendosura të gjitha data setet e nevojitur për aplikacion bashkë me modelet e trajnuara, pra shtegu `./main_route/fetched_steam_data` paraqet data setin e krijuar si më lartë.

Hapi 13 paraqet një aksion abstrakt, që do të thotë se pas ekzekutimit të hapave më lartë, duhet vizituar shtegu `scalatra-1.0/bin/` dhe ekzekutuar BIN fajlli `scalatra`. Prapashtesa `-Dhttp.port` paraqet një aksion që detyron aplikacionin të ekzekutohet në portin `${PORT}`. Ky port vjen nga *GCP (Google Cloud Platform)*, pasi si rregull i GCP është që porti në të cilën do të publikohet ueb-faqja të jetë dinamike, e jo të caktohet nga zhvilluesit e aplikacionit (në rastin tonë ne).

Një përmbledhje e të gjithë hapave më sipër është përshkruar më poshtë (Fig 47)

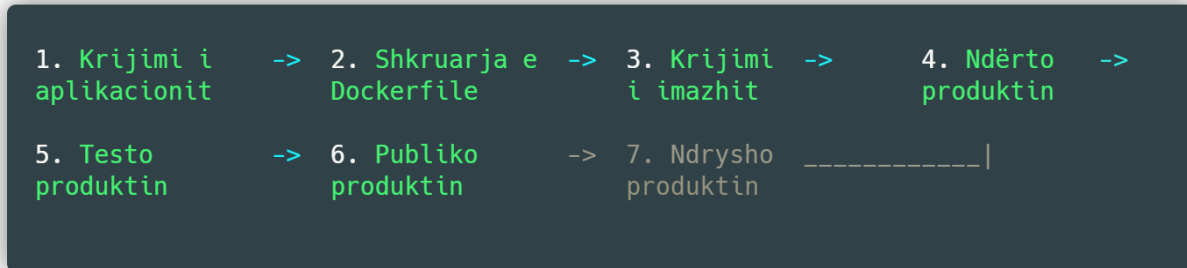


Fig 47: Përmbledhje e hapave të krijimit të imazhit

Me të kryer këto hapa, imazhi jonë është gati për tu krijuar. Për të vazhduar më tutje, hapim një *CLI (Command Line Interface)* në mënyrë që të ndërtojmë imazhin dhe përdorim komandat (Fig 48)


```
cd Documents/Github/ScalaTry/  
docker build -t intellidota .
```

Fig 48: Komandat për ndërtimin e Docker imazhit

Komanda e parë mundëson kalimin te direktoriumi ku gjendet projekti, ndërsa komanda e dytë ndërton imazhin. Parashtesa *-t* mundëson qëndron për etiketues (ang. *tag*) dhe lejon që imazhi i krijuar në këtë rast të njihet si intellidota, ndërsa pika tregon që shtegu që duhet përdorur për të krijuar imazhin është ky ku gjendemi aktualisht. Pas ekzekutimit të asaj komande, marrim rezultatet si në vijim (Fig 49):

```
PS C:\Users\Labinot\Documents\Github\intellidota> docker build -t intellidota .  
    Sending build context to Docker daemon  235.3MB  
Step 1/7 : FROM aa8y/sbt:0.13.18  
----> 5760094a84cb  
Step 2/7 : USER root  
----> Using cache  
----> 66fc68e29875  
Step 3/7 : RUN apk update && apk add --no-cache libc6-compat  
----> Using cache  
----> a89b58f78e6e  
Step 4/7 : COPY ./target/universal/scalatra-1.0.zip .  
----> Using cache  
----> d9b718abb4c1  
Step 5/7 : RUN unzip scalatra-1.0.zip  
----> Using cache  
----> bd79cc99cad8  
Step 6/7 : COPY ./main_route .  
----> Using cache  
----> fba5375c8e10  
Step 7/7 : CMD scalatra-1.0/bin/scalatra -Dhttp.port=${PORT}  
----> Using cache  
----> a3a823eff3c5  
Successfully built a3a823eff3c5  
Successfully tagged intellidota:latest  
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and  
directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and  
reset permissions for sensitive files and directories.  
PS C:\Users\Labinot\Documents\Github\intellidota>
```

Fig 49: Pamja e parë e ndërtimit të imazhit në Docker

Dhe nëse duam që të shfaqim imazhet e krijuara deri më tani, përdorim komandën *docker images*. Pas krijimit të imazhit, ekzekutojmë atë me anë të komandës: *docker run -it intellidota* ku parashtesa *-it* qëndron për ndërfaqen (ang. *interface*) dhe marrim rezultatet si më poshtë (Fig 50):

```

PS C:\Users\Labinot\Documents\Github\intellidota> docker run -it intellidota
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/src/app/scalatra-1.0/lib/ch.qos.logback.logback-classic-1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/src/app/scalatra-1.0/lib/org.slf4j.slf4j-log4j12-1.7.16.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]
[warn] o.a.h.u.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
[info] play.api.Play - Application started (Prod) (no global state)
[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0.0.0.0:1234

```

Fig 50: Ndërtimi i suksesshëm i imazhit

Pra, serveri është i ngritur në 0.0.0.0 me portin 1234, pra vizitimi i *localhost:1234* na shpjen te aplikacioni jonë. Me këtë rezultat, edhe publikimi në çfarëdo platforme të madhe Cloud nuk do të na shfaq problem. Për fund, le të shohim përmbajtjen e kontejnerit tonë.

Fillojmë me komandën *docker container ps --all*. Kjo mundëson që të listohen të gjithë kontejnerët bashkë me numrin identifikues përkatës dhe rezultati duket si më poshtë (Fig 51):

```

PS C:\Users\Labinot> docker container ps --all

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c26ebae69ab5	intellidota	"/bin/sh -c 'scalatr...'"	26 seconds ago	Up 25 seconds
	vigorous_shockley			

Fig 51: Listimi i kontejnerëve

Që do të thotë se kontejneri jonë ka numrin identifikues c26ebae69ab5 dhe ka emrin intellidota. Me të pranuar këto informacione, ekzekutojmë komandën *docker exec -it c26ebae69ab5 bash* për të kaluar në përmbajtje të kontejnerit. Fjala *exec* qëndron për ekzekutues (ang. *ekzekutues*).

Tanimë jemi brenda kontejnerit tonë. Fillojmë me komandën PWD (Point Working Directory) për të parë shtegun në të cilin jemi aktualisht (Fig 52):

```

bash-4.4# pwd
/usr/src/app

```

Fig 52: Direktori ku po punojmë në kontejner

Shikojmë se çfarë ka brenda direktoriumit aktual si më poshtë (Fig 53):

```
bash-4.4# ls
classified_model  fetched_steam_data  logs
scalatry-1.0.zip clustered_model      kaggle_data         scalatry-1.0
```

Fig 53: Përmbajtja e kontejnerit

Pra, kemi modelet e trajnuara, projektin e kompresuar dhe të dekompresuar (scalatry-1.0.zip dhe scalatry-1.0 respektivisht). Nëse kthehemi tek direktoriumi bazë, pra root, shohim se kemi (Fig 54):

```
bash-4.4# cd ~
bash-4.4# ls
bin      lib      libexec  local   sbin     share   src
```

Fig 54: Direktoriumi bazë i kontejnerit

Duke u bazuar në këtë rezultat, shohim se brenda imazhit aa8y/sbt:0.13.18 është i përdorur një sistem operativ Unix, nga se ne nuk kemi përdorur një imazh të tillë në mënyrë të drejtpërdrejtë.

Kjo është edhe një arsye se pse Docker është aktualisht një ndër veglat më të përdorura aktualisht, pra ofron fleksibilitet dhe është lehtë i përdorshëm dhe menaxhueshëm.

7 Konkluzioni

Projekti është i ndarë në dy pjesë, pjesa *frontend* dhe pjesa *backend* nga edhe kanë ardhur vështirësitë më të shumta, pasi vizualizimi dhe përshkrimi i funksioneve kryesore është dashur të bëhet në koordinim të plotë. Gjithsesi, eksperiencia e krijimit të një platforme të tillë në të cilën koordinohen pjesa *frontend* dhe pjesa *backend* ka qenë e veçantë dhe do të hyjë në përdorim shumë në të ardhmen, pasi në ditët e sotme rrallë herë punohet me *frontend* të integruar në skeletin (ang. *framework*) përkatës.

Megjithëse në industri ekzistojnë projekte të ngjashme, asnjë nuk është i tillë. Ndër projektet më të spikatura që edhe nga jemi inspiruar janë Opendota dhe Dotabuff, që integrojnë grafike me statistika në kohë reale dhe japin një ndërfaqe të këndshme në të cilën lehtë mund të arrihet improvizimi, megjithatë asnjëri nuk integron në to të mësuarit e makinës, qoftë ai kllasterim apo klasifikim, apo ndonjë lloj të mësuarit tjetër.

Pjesa e parë e projektit, e quajtur IntelliDota Classification fillon me krijimin e data setit tonë nga Steam Public API. Me krijim të data setit, mendohet dërgimi i kërkesave GET më shumë se 50000 herë deri të marrim Json dokumentet e nevojshme, në mënyrë që data seti jonë të ketë një pamje fillestare, megjithëse gjatë këtyre GET kërkesave, bëjnë përjashtim disa. Pra, ne marrim vetëm data setet që janë përshkruese dhe që nuk përmbajnë vlera të pa parashikuara. Një plus gjatë kësaj procedure është se nuk merremi me vlera të zbrazëta pasi vetëm se i filtrojmë ato gjatë marrjes së data setit. Pra, nuk kemi nevojë për para-procesim konkret.

Pason zhvillimi i algoritmit Random Forest në data setin e krijuar, në të cilin është fituar një saktësi rreth 87%. Kemi përdorur Random Forest me një thellësi prej 10 njësish në mënyrë që të shmanget *overfitting*. Modelin e trajnuar e kemi ruajtur me anë të Docker imazhit në mënyrë që të ri-përdoret më vonë, pra kurdo që të jetë e nevojshme e të mos ketë nevojë për ri-trajnim.

Më pas është vazhduar me krijimin e ueb shërbimeve nëpërmjet gjuhës programuese Scala, konkretisht strukturës Play. Play paraqet një model MVC (model, view, controller) dhe është një aset i fuqishëm në zhvillimin e ueb faqeve. Si ueb shërbime, kemi krijuar disa, ku kryesoret janë mundësia për të klasifikuar dhe kllasteruar në kohë reale, duke mos lënë anash mundësitë për analiza grafike apo vizualizime.

Të gjithë projektin tonë e kemi ndërthurur në një imazh Docker. Këtë e kemi realizuar në mënyrë që projekti të mund të ekzekutohet kudo, pavarësisht ambientit të cilit ndodhet. Imazhet janë edhe një mënyrë e re e hostimit e përdorur shumë kohëve të fundit, e për të krijuar imazhe Docker është vegla kryesore. Imazhi ndërtohet kur brenda strukturës së projektit krijohet një Dockerfile që përdoret si dhënës instruksionesh për veglën Docker. Imazhi jonë Docker përfshinë data setet Steam dhe Kaggle bashkë me modelet trajnuese të secilit. Imazhi jonë është i izoluar, nga edhe vjen përparësia kryesore.

Ndërkaq, projekti jonë aktualisht nuk mund të zgjerohet më shumë por mund të përdoret si një aplikacion bazë për statistika interesante. Megjithatë, gjithsesi se do të vazhdohet të punohet me të mësuarit e makinës dhe të nxirren analiza më të thella si për video loja, edhe për raste konkrete. Në video lojëra, mund të hidhet edhe një hap i ardhshëm si për shembull, nxjerrja e analizave se cilat ishin arsyet kryesore që një ekip fitoi. Niveli i të mësuarit të makinës që duhet

zhvilluar për këtë aplikacion nuk është më i avancuar se sa niveli që po përdorim aktualisht, por aftësitë analitike duhet rritur, pasi gjithçka përsëri nxirret nga grafet.

Elemente tjera me rëndësi edhe për projektin edhe për të ardhmen kanë qenë përdorimi i strukturës Json, pra manipulimi i attributeve nëpërmjet Scalas, përdorimi i Gson si librari ndërmjetësuese e Json dhe Scalas dhe marrja e të dhënave nëpërmjet vartësive të Scalas, si *lihaoyi* me anë të së cilës kemi realizuar kërkesat. Ja vlen të ceket edhe njëherë aplikacioni *ngrok* me anë të së cilit kemi mundur të publikojmë pikat fundore tonat ashtu që të mund të realizohet pjesa e dytë, pra ajo *frontend*.

Të gjitha pjesët e këtij aplikacioni janë publike, duke filluar nga data seti i krijuar, tek Scala, lidhja ndërmjet tyre me REST modelin si dhe algoritmi i klasifikuar dhe kllasteruar në mënyrë që zhvilluesit të ndihmohen sado pak.

8 Shtojcat

Më poshtë janë listuar kodet burimore dhe data seti:

- Backend: <https://github.com/LabinotVila/IntelliDota>
- Modelet e trajnuara: <https://github.com/noraibrahimi/IntelliDotaIC>
- Frontend: <https://github.com/noraibrahimi/IntelliDota-mobile>
- Steam data seti: <https://www.kaggle.com/labinotvila/dota-2-steam-api-fetched-dataset>

9 Lista e pikave fundore

9.1 index

host / index

Parametrat:

Përshkrimi: Kthen përmbajtjen e faqes filluese, që është lista e pikave fundore të ofruara nga ne.

Tab 1: Pika fundore index

9.2 getColumns

host / getColumns

Parametrat: **kind** – lloji i datasetit [steam / Kaggle]

Përshkrimi: kthen një listë të kolonave të datasetit përkatës

Shembull: /getColumns?**kind**=steam

Tab 2: Pika fundore getColumns

9.3 getSample

host / getSample

Parametrat: **kind** – lloji i datasetit [steam / Kaggle]

percentage – përqindja e monstrës [0 deri 100]

Përshkrimi: kthen një monstër të datasetit përkatës, varësisht përqindjes

Shembull: /getSample?**kind**=steam&**percentage**=10

Tab 3: Pika fundore getSample

9.4 getStages

host / getStages

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
Përshkrimi:	kthen një varg të fazave nëpër të cilët kanë kaluar të dhënat	
Shembull:	/getStages? kind =kaggle	

Tab 4: Pika fundore getStages

9.5 getCorrelationMatrix

host / getCorrelationMatrix

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
Përshkrimi:	kthen një varg numrash që tregojnë ndërlidhjen mes kolonave të datasetit përkatës	
Shembull:	/getCorrelationMatrix? kind =kaggle	

Tab 5: Pika fundore: getCorrelationMatrix

9.6 getGroupAndCount

host / getGroupAndCount

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
	attribute	- grupimi ndodhë sipas këtij atributi
	partitions	- numri i ndarjeve të të dhënave numerike
Përshkrimi:	kthen një varg grupesh dhe intervalin përkatës që të dhënat i përkasin	
Shembull:	/getGroupAndCount? attribute =xp_per_min& partitions =3	

Tab 6: Pika fundore getGroupAndCount

9.7 getStats

host / getStats

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
Përshkrimi:	kthen sasinë e rreshtave dhe kolonave të datasetit përkatës	
Shembull:	/getStats? kind =steam	

Tab 7: Pika fundore getStats

9.8 getSchema

host / getSchema

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
Përshkrimi:	kthen kolonat dhe tipin përkatës të datasetit të caktuar	
Shembull:	/getSchema? kind =steam	

Tab 8: Pika fundore getSchema

9.9 getDoubleGroup

host / getDoubleGroup

Parametrat:	kind	- lloji i datasetit [steam / Kaggle]
	col1	- kolona e parë
	col2	- kolona e dytë
Përshkrimi:	kthen grupimin dhe njehsimin sipas kolonave përkatëse	
Shembull:	/getDoubleGroup? kind =steam& col1 =leaver_status& col2 =radiant_win	

Tab 9: Pika fundore getDoubleGroup

9.10 postPredict

host / postPredict

Parametrat:	gold_per_min – paratë për minutë	- derivuar
	level – nivelet	- derivuar
	leaver_status – statusi i përfundimit të ndeshjes	- derivuar
	xp_per_min – eksperiencia për minutë	- derivuar
	radiant_score – pikët e të bardhëve	- drejtpërdrejtë
	gold_spent – paratë e harxhuara	- derivuar
	deaths – vdekjet gjatë lojës	- derivuar
	denies – mohimet e vrasjeve	- derivuar
	hero_damage – dëmtimet ndaj kundërshtarëve	- derivuar
	tower_damage – dëmtimet ndaj kullave	- derivuar
	last_hits – goditja përfundimtare për vrasje	- derivuar
	hero_healing – ndihmesa ndaj ekipit	- derivuar
	duration – gjatësia e lojës	- drejtpërdrejtë
Përshkrimi:	Kthen dy attribute të reja në data set që tregojnë se a kanë fituar të bardhët apo jo dhe saktësisht përqindjen e përgjigjes së dhënë	
Shembull:	/postPredict? gold_per_min=2585&level=139&leaver_status=0&xp_per_min=3658 &radiant_score=34&gold_spent=89220&deaths=65&denies=21 &hero_damage=170629&tower_damage=2732&last_hits=901 &hero_healing=10700&duration=2549	

Tab 10: Pika fundore postPredict

10 References

- [1] P. Norvig dhe S. J. Russell, *Artificial Intelligence: A Modern Approach*, Harlow, United Kingdom: Prentice Hall, 2009.
- [2] P. Norvig dhe S. J. Russell, «Artificial Intelligence: A Modern Approach,» në *Artificial Intelligence: A Modern Approach*, Harlow, United Kingdom, Prentice Hall, 2019, p. 37.
- [3] P. Norvig dhe S. J. Russell, «Artificial Intelligence: A Modern Approach,» në *Artificial Intelligence: A Modern Approach*, Harlow, United Kingdom, Prentice Hall, 2009, p. 2.
- [4] P. R. S. J. Norvig, «Artificial Intelligence: A Modern Approach,» në *Artificial Intelligence: A Modern Approach*, Harlow, United Kingdom, Prentice Hall, 2009, p. 695.
- [5] J. R. Quinlan, «Wikipedia,» University of Sydney, 1975. [Në linjë]. Available: https://en.wikipedia.org/wiki/Decision_tree_learning. [Qasja 6 October 2019].
- [6] Bayes, «Wikipedia,» [Në linjë]. Available: https://en.wikipedia.org/wiki/Naive_Bayes_classifier. [Qasja 6 October 2019].
- [7] Ho dhe . L. Breiman, «Wikipedia,» 1995. [Në linjë]. Available: https://en.wikipedia.org/wiki/Random_forest. [Qasja 11 September 2019].
- [8] Valve, «Steam,» 8 October 2016. [Në linjë]. Available: <https://steamcommunity.com/dev>. [Qasja 4 November 2019].
- [9] neilatsyracuse, «Reddit,» Reddit, 2018. [Në linjë]. Available: https://www.reddit.com/r/DotA2/comments/7yzlu4/dota2_api/. [Qasja 2 October 2019].
- [10] S. Steam Developers, «Steam,» 12 September 2003. [Në linjë]. Available: <https://steamcommunity.com/dev>. [Qasja 5 October 2019].
- [11] «w3school,» w3, 2019. [Në linjë]. Available: https://www.w3schools.com/js/js_JSON_intro.asp. [Qasja 1 October 2019].
- [12] «Github,» Google, 22 May 2008. [Në linjë]. Available: <https://github.com/google/gson>. [Qasja 11 October 2019].
- [13] Oracle, «Wikipedia,» OpenJDK, 8 May 2007. [Në linjë]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine. [Qasja 28 September 2019].

- [14] «Jetbrains,» JetBrains, 1 January 2001. [Në linjë]. Available: <https://www.jetbrains.com/help/idea/2016.1/intellij-idea-help.pdf>. [Qasja 1 October 2019].
- [15] M. Zaharia, «Apache Spark,» 9 September 2019. [Në linjë]. Available: <https://spark.apache.org/docs/latest/ml-features>. [Qasja 16 October 2019].
- [16] P. Bugnion, P. R. Nicolas dhe A. Kozlov, Scala: Applied Machine Learning, Packt Publishing, 2017.
- [17] Prime, «Prime,» Prime, November 2016. [Në linjë]. Available: <https://goprime.io/>. [Qasja June 2019].
- [18] Databricks, The Data Scientist's Guide to Apache Spark™, Berkley: Apache Spark, 2019.