Optimizacija modela temeljenih na Gradient Boosting metodi

Markov, Petar

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Petar Markov

Optimizacija modela temeljenjih na Gradient Boosting metodi

DIPLOMSKI RAD

Pula, 2. lipanj 2022. godine

SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE

Petar Markov

Optimizacija modela temeljenjih na Gradient Boosting metodi

DIPLOMSKI RAD

JMBAG: 0165067323, redoviti student
Studijski smjer: Informatika

Kolegij: Izrada Informatičkih Projekata
Znanstveno područje : Društvene znanosti
Znanstveno polje : Informacijske i komunikacijske znanosti
Znanstvena grana : Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Nikola Tanković

Pula, 2. lipanj 2022. godine

Sažetak

Na internetu imamo sve više podataka koji svakim danom eksponencijalno rastu te se povećava potreba za obradom tih podataka. U tome nam uvelike pomaže strojno učenje i modeli koji se koriste za obradu podataka. Svaki od modela dolazi sa unaprijed postavljenim parametrima koji poboljšavaju modele. Ti parametri se nazivaju hiperparemtri modela, a njihova optimizacija je jedan od najvažnijih koraka u izgradnji modela.

Optimizacija hiperparametara se može odraditi ručno, ali to je iscrpan posao koji oduzima puno vremena. Zato su napravljeni algoritmi i tehnike automatske optimizacije hiperparametara od kojih su u ovome diplomskom radu spomenute sljedeće: Pretraživanje po rešetci (*eng. Grid Serach*), Nausumično pretraživanje (*eng. Random Serach*), Bayesova optimziacija (*eng. Bayesian Optimization*), *Tree-structured Parzen Estimator* i *Simmulated Anneal*.

Ključne riječi : Strojno učenje, model, hiperparametri, optimizacija, algoritmi

Abstract

Every day across the Internet the number of data is growing exponentially, and therefore the need for processing that data is growing also. Machine learning and models used in it are helping us in processing that data. Every model comes with preset parameters which improve these models. These parameters are called hyperparameters of model and their optimization is one of the most important steps in building the models.

Optimization of hyperparameters can be done manually, but that is exhausting, and most of the time, time-consuming job. Therefore, algorithms and techniques for automatic optimization of hyperparameters have been developed, of which the following are mentioned in this thesis: Grid Search, Random Search, Bayesian Optimization, Tree-structured Parzen Estimator and Simmulated Anneal.

Keywords : Machine learning, model, hyperparameters, optimization, algorithms

Sadržaj

1 Uvod

Prema podacima iz 2020. godine [1], dnevno generiramo oko 2.5 kvintilijuna bajtova podataka (kvintilijun = 10^{18}). Procjene su da će do 2025. taj broj narasti na 463 exabajtova(10^{18} bajtova). Ta se impresivna količina podataka obradom pretvara u korisne informacije, a pri tome nam uvelike pomaže strojno učenje.
Strojno učenje poznaje veliki broj modela i postupaka učenja, a svaki on njih omogućuje dodatne korisničke parametre koji služe prilagodbi modela u postizanju bolje točnosti prilikom predviđanja. Ti parametri se zovu hiperparametri.
Tema ovog diplomskog rada je automatizacija pronalaska hiperparametara modela zasnovanih na *Gradient Boosting* metodi - jednoj od vrsti algoritama za strojno učenje. To će se provesti pomoću 4 najkorištenije metode optimizacije hiperparametara, a to su: Pretraživanje po rešetci (*eng. Grid Search*), Nasumično pretraživanje (*eng. Random Search*), Bayesova optimizacija (*Bayesian Optimzation*) i algoritme iz programske knjižnice *Hyperopt*: *Tree-structured Parzen Estimator* i *Simulated Anneal*.
Rad opisuje svaku od navedenih metoda te iznosi njihovu usporedbu u pronalaženju optimalnih hiperparametara i krajnje poboljšanje postignute točnosti predikcije modela u odnosu na osnovni model. Za potrebe ovog diplomskog rada koristit ćemo 3 skupa podataka (*eng. Dataset*), koji će biti podijeljeni na 10 skupova za treniranje i 10 skupova za testiranje pomoću metode unakrsne provjere (*eng. Cross Validation*) kako bi se uočila važnost slučajnog odabira skupa za treniranje i skupa za testiranje.

2 Hiperparametri u modelima strojnog učenja

Svaki model u strojnom učenju se bazira na nekoliko čimbenika koji utječu na njegovu točnost i brzinu. Ti čimbenici se nazivaju hiperparametri modela kojeg koristimo za učenje. Oni se prilikom pozivanja modela postavljaju na zadane vrijednosti koje u većini slučaja nisu optimalne, te se njihovom optimizacijom mogu dobiti bolji rezultati. Oni se, za razliku od podataka koje model prima tijekom učenja, postavljaju prije nego što model krene učiti te o njima ovisi na koji način će model učiti prema podacima koje primi. Automatska optimizacija hiperparametra modela je jedan od osnovnih zadataka u automatskom strojnom učenju, ali to može biti vrlo dug i mukotrpan proces, pogotovo ako u skupu podataka ima veliki broj opservacija koji moraju proći kroz model. Ali ako se automatska optimizacija hiperparametara dobro odradi rezultati točne predikcije modela se povećavaju, te dobivamo bolji model koji će nam dati bolje predikcije novih podataka na kojima model nije naučen. Različiti skupovi podataka zahtijevaju različite modele po kojima uče, pa tako kao „znanstvenik o podatcima“ (*eng. Data Scientist*) moramo biti upoznati sa svakim modelom i načinom na koji on radi, te za koju vrstu problema se koristi pojedini model. U nadziranom učenju (*eng. Supervised learning*) najčešće imamo dva problema s kojima se susrećemo [2], a to su problem regresije i problem klasifikacije. Problem regresije se temelji na predikciji vrijednosti, npr. kolika će biti cijena kuće prema danim podacima, dok se problem klasifikacije temelji na predikciji kategorije, npr. nalazi li se mačka ili pas na slici [3]. U ovome diplomskom radu fokusirat ćemo se na problem klasifikacije.

2.1 Proces učenja

Bilo koji problem u strojnom učenju može se podijeliti na 7 [4] osnovnih koraka:

- **Prikupljanje podataka**: Prvi korak u kojemu se skupljaju, po mogućnosti pouzdani podaci, koji će se predati računalu da iz njih nauči neke uzorke. Kvaliteta i točnost podataka je izuzetno bitna za točnost našeg modela. Ako imamo zastarjele ili krive podatke to će utjecati na točnost našeg modela.

- **Priprema podataka**: Nakon što prikupimo podatke, moramo ih pripremiti da iz njih model može naučiti. Ovo je jako važan korak zato što podaci mogu dolazi u svakakvim oblicima. Neke od najvažnijih tehnika za pripremu podataka su: čišćenje podataka od nevažnih podataka, nedostajućih vrijednosti, kolona i redova koji nemaju važnost u učenju,

te skaliranje podataka. Također jedna od bitnih stvari je predočavanje podataka. To nam uvelike pomaže da vidimo strukturiranost podataka, te poveznice između nekih varijabli i klasa u klasifikaciji. Zadnja stvar koju je potrebno napraviti je podijeliti podatke na 2 skupa ali to će biti detaljnije opisano naknadno u radu.

- **Odabir modela**: U ovom koraku potrebno je izabrati odgovarajući model za naše podatke. U strojnom učenju ima dosta modela, te je važno izabrati pravi model. Neki od modela su dobri za prepoznavanje zvuka i glazbe, neki za slike, a neki za predikcije. U sljedećem potpoglavlju su opisani modeli u klasifikaciji.
- **Treniranje modela**: Treniranje modela je najvažniji korak u strojnom učenju. U ovom koraku model uči na podacima koje smo pripremili te postepeno pronalazi uzorke među podacima kako bi napravio predikcije. Ovo će rezultirati poboljšanim predikcijama tijekom učenja modela.
- **Evaluacija modela**: Nakon što se završi treniranje moramo provjeriti kako se naš model ponaša na dosad neviđenim podacima. Ti podaci se nazivaju set za testiranje, ali o tome više naknadno.
- **Optimizacija parametara**: Nakon što smo napravili i evaluirali model, trebamo pokušati poboljšati model na bilo koji način, a to je optimizacija hiperparametara, što je i cilj ovog diplomskog rada te je ovaj korak opisan u prijašnjem potpoglavlju.
- **Predviđanje podataka**: Ovo je završni korak u kojemu naš model može raditi predviđanje novih podataka koji nisu bili u početnom setu podataka.

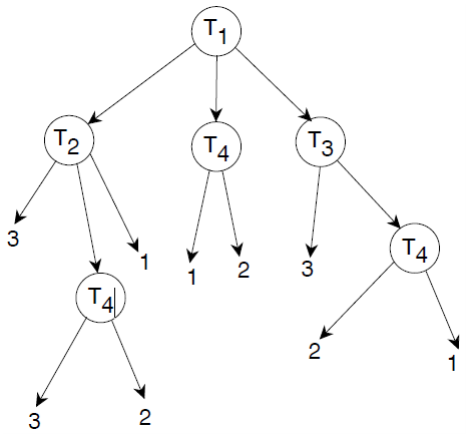
2.2 Modeli u klasifikaciji

Kod klasifikacije postoji nekoliko modela po kojima se klasificiraju podaci. To su:

- Logistička regresija (*eng. Logistic Regression*)
- Naïve Bayes
- K-najbližih susjeda (*eng. K-Nearest Neighbours*)
- Stablo odluke (*eng. Decision Tree*)
- Slučajna šuma (*eng. Random Forrest*)
- Stroj potpornih vektora (*eng. Support Vector Machine*)

2.2.1 Stablo odluke (*eng. Decision Tree*)

Fokusirat ćemo se na model stabla odluke zato što se algoritam *Gradient Boosting*, koji ćemo koristiti u ovome diplomskom radu, temelji na ovome modelu. Stablo odluke se generalno definira kao stablo čiji unutarnji čvorovi su testovi, a listovi su kategorije, tj. kategorije koje želimo predvidjeti.[5]



Slika 1: Stablo odluke

Na slici 1 vidimo jednostavno stablo odluke s tri ishoda (listovi drveta). Unutarnji čvorovi su označeni s T1 do T4, te oni predstavljaju testove koji imaju ishode, i na osnovu tih ishoda se generira stablo odluke. Model stabla odluke radi tako da se opservacija koju želimo klasificirati ispita kroz testove, počevši od T1 , te se zavisno od ishoda testa spušta kroz stablo sve dok ne dođe do jednog od listova koji klasificiraju tu opservaciju.

2.2.2 Hiperparametri stabla odluke

Model stabla odluke u programskom jeziku *Python* se implementira pomoću programske knjižnice *Scikit-Learn*[6] [7]. *Scikit-Learn* je jedna od najpopularnijih programskih knjižnica za strojno učenje koja sadrži sve najpotrebnije alate za klasifikaciju, regresiju i grupiranje. Klasa koja predstavlja model stabla odluke u *scikit-learn* knjižnici naziva se *DecisionTreeClassifier* te dolazi s mnoštvom hiperparametara od kojih ćemo spomenuti najvažnije koji utječu na performanse modela[8]:

- *criterion*: Funkcija koja mjeri kvalitetu podjele čvorova stabla. Možemo birati između dvije moguće funkcije, a to su *Gini* koja mjeri frekvenciju na kojoj će bilo koji element skupa podataka biti pogrešno označen kada je nasumično označen [9] te *Entropy* koja je mjera poremećaja ili mjera čistoće listova [10]

- *splitter*: Označava strategiju koja se koristi kako bi se podijelili čvorovi. Postoje dvije vrste, a to su *Best* - označava najbolju podjelu, te *Random* koja označava najbolju nasumičnu podjelu
- *max_depth*: Broj koji određuje maksimalnu dubinu stabla
- *min_samples_split*: Minimalni broj uzoraka potrebnih za podjelu unutarnjih čvorova
- *min_samples_split*: Minimalan broj uzoraka koji se moraju nalaziti u listu
- *max_features*: Broj značajki koje treba uzeti u obzir pri traženje najbolje podjele

Prilikom traženja optimalnih hiperparametara može doći do problema prenaučenosti *(eng. Overfitting)* modela.

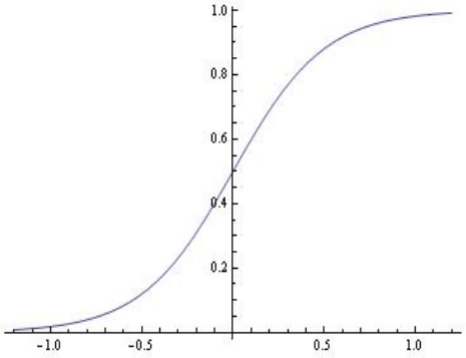
2.3 Funkcija gubitka (*eng. Cost Function*)

Prije nego što objasnim prenaučenost modela, moram spomenuti funkciju gubitka o kojoj ovisi koliko je model dobro naučen. Prilikom odabira našeg modela, cilj nam je pronaći funkciju koja je „dobar“ prediktor ulaznih značajki (*eng. Input Features*) za odgovarajuće izlazne značajke (*eng. Output Features*). Tu funkciju nazivamo hipotezom. Funkciju gubitka objasnit ću na jednostavnom primjeru logističke regresije. Također radi lakšeg objašnjenja primjer će biti na binarnoj klasifikaciji. Binarna klasifikacija ima 2 izlazne značajke, 0 i 1. 0 u klasifikaciji predstavlja klasu da se nešto neće dogoditi, npr. ako želimo klasificirati poruke kao spam, klasa 1 predstavlja da je mail spam, a klasa 0 da nije spam. Kod logističke regresije hipoteza izgleda ovako:

$$h_\theta(x) = g(\theta^T x) \tag{1}$$

gdje $\theta^T x$ predstavlja vektor ulaznih značajki s parametrima θ. Naš cilj je pronaći vrijednosti parametara θ koji će nam dati funkciju u ovisnosti o x koja dobro opisuje naše podatke. Ideja je da izaberemo parametre θ tako da hipoteza, $h_\theta(x)$, daje rezultat blizu izlazne značajke (y) za par podataka iz kojeg naš model uči (x,y). Kada u ovu jednadžbu ubacimo logističku funkciju dobivamo sljedeću formulu:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}} \tag{2}$$



Slika 2: Logistička Funkcija [11]

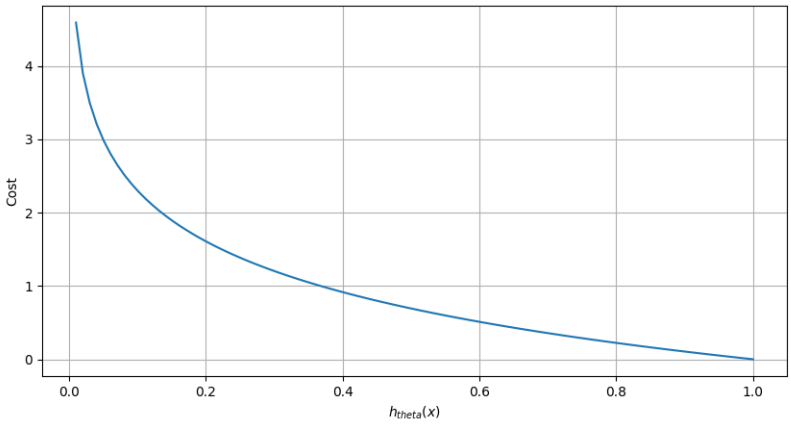
Kao što vidimo sa slike 2, logistička funkcija odlično opisuje problem binarne klasifikacije, zato što će mapirati bilo koji realan broj u rasponu od 0 do 1. Važno je za napomenuti će nam hipoteza dati vjerojatnost da je izlazna značajka 1. To znači da ako je $h_\theta(x) = 0.7$, vjerojatnost da je mail spam je jednaka 70%, dok je vjerojatnost da nije spam jednaka 1-$h_\theta(x)$, u ovome slučaju 0.3, ili 30%. Funkciju gubitka računamo sljedećom formulom:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^i)), y^i) \tag{3}$$

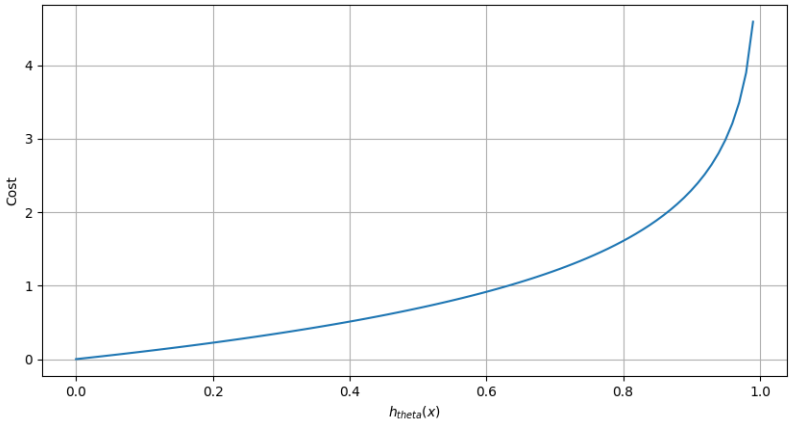
gdje $J(\theta)$ predstavlja ukupni gubitak u ovisnosti o parametrima a m je broj opservacija. U slučaju logističke regresije $Cost(h_\theta(x^i)), y^i)$ računa se pomoću sljedeće dvije formule:

$$Cost(h_\theta(x^i)), y^i) = \begin{cases} -\log(h_\theta), \text{if } y = 1 \\ -\log(1 - h_\theta)), \text{if } y = 0 \end{cases} \tag{4}$$

Grafovi ove dvije funkcije izgledaju:



Slika 3: $-\log(h_\theta)$



Slika 4: $-\log(1-h_\theta))$

Prema slici 3 vidimo da funkcija $-\log(h_\theta)$ na y osi ide u beskonačnost, dok na x osi prolazi kroz 1. Ova funkcija je samo za y = 1, te nam govori da ako naš model, na osnovu ulaznih značajki i parametara, predvidi klasu 1(da će se nešto dogodit) naš gubitak će biti 0, što je i logično zato što smo to i htjeli predvidjeti. Ali što se više približavamo 0, tj. ako je naša pretpostavka bliža 0, gubitak je sve veći, na samom kraju beskonačan. U drugom slučaju (slika 4), je obrnuta situacija. Ova funkcija je samo kada je izlazna značajka y=0, te ako naš model predvidi klasu 0 gubitak je 0, a što se više približavamo 1, tj. što je naš model netočniji, to je gubitak veći. Ovo možemo predočiti sljedećim formulama:

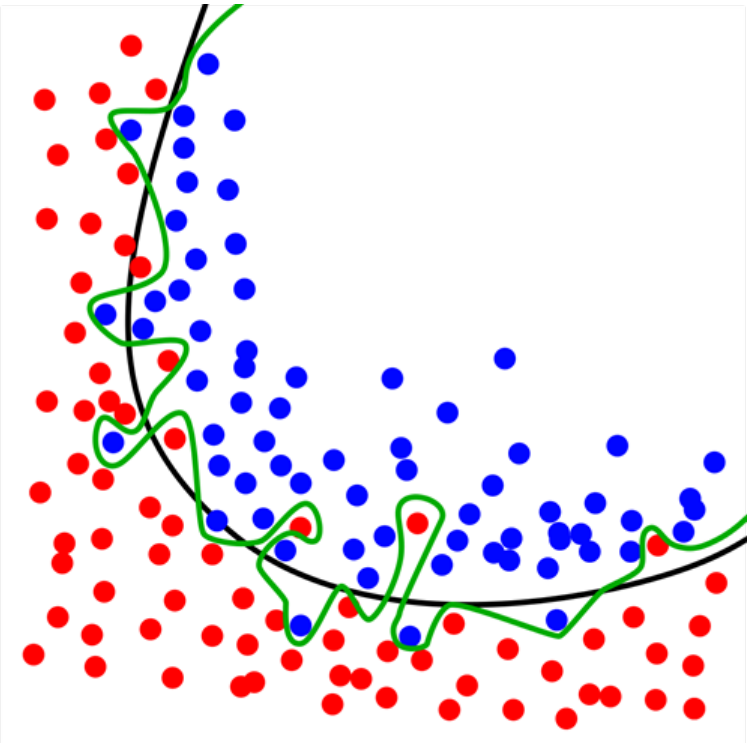
$$Cost(h_\theta(x)), y) = 0, \text{if } h_\theta(x) = y \tag{5}$$

$$Cost(h_\theta(x)), y) \to \infty, \text{if } y = 0 \text{ and } h_\theta(x) \to 1 \tag{6}$$

$$Cost(h_\theta(x)), y) \to \infty, \text{if } y = 1 \text{ and } h_\theta(x) \to 0 \tag{7}$$

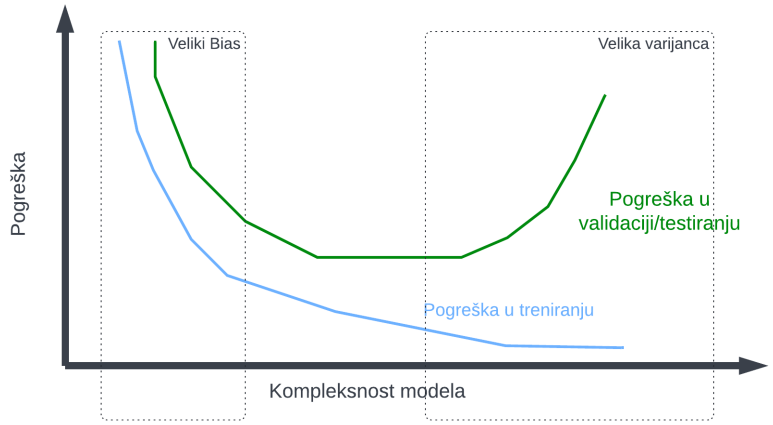
2.4 Prenaučenost (*eng. Overfitting*)

Prenaučenost modela je česta pojava koja se događa prilikom učenja modela. Do nje dolazi kada naš model previše „odgovara“ skupu za treniranje. Kada se to dogodi, model više ne može točno predviđati nove podatke koje nije vidio, zato što je previše dobro naučio predviđati samo prema podacima iz skupa za treniranje. Ako model predugo uči na istim podacima, ili je model pre kompleksan, može početi učiti nevažne podatke iz skupa podataka. Takav model je također prenaučen i nije u mogućnosti generalizirati se prema novi podacima, te ne može izvesti klasifikacijske zadatke za koje je namijenjen[12].



Slika 5: Prenaučenost

Na slici 5 je prikazan primjer prenaučenosti. Vidimo da imamo klasifikacijski problem s dva ishoda, crvene i plave točke. Zelenom crtom je označena hipoteza modela koji je prenaučen zato što uzima u obzir sve crvene točke, dok je crnom crtom označena hipoteza modela koji je optimalno naučen. Uz pojam prenaučenosti usko se vežu još dva pojma : varijanca (*eng. Variance*) i bias/pristranost/pomak (*eng. Bias*) (nadalje bias). Bias je nemogućnost modela da nauči dovoljno dobro hipotezu iz skupa za treniranje. Još kažemo da je to razlika između prosječne predikcije našeg modela i točne vrijednosti koju želimo predvidjeti[13]. Modeli s velikim bias-om vode do velike pogreške i na skupu za treniranje i na skupu za testiranje. To se također naziva podnaučenost (*eng. Underfitting*) modela. Varijanca pokazuje koliko će se procijenjeni iznos hipoteze razlikovati ako se koriste drugi skupovi za treniranje. To znači da se opisuje koliko se slučajna varijabla razlikuje od očekivane vrijednosti[13]. Velika varijanca dovodi do prenaučenosti.



Slika 6: Bias vs Varijanca

Na slici 6 vidimo razliku između biasa i varijance na osnovu kompleksnosti modela i greške modela. Model ima velik bias, tj podnaučen je kada mu je velika greška i na skupu za treniranje i na skupu za validaciju/testiranje, dok model ima veliku varijancu, tj prenaučne je kada mu je greška mala na skupu za testiranje a velika na skupu za validaciju/testiranje. Ovo nas dovodi do pojma unakrsne provjere koja sprječava pojavu prenaučenosti i podnaučenosti.

2.5 K-struka unakrsna provjera (*eng. K-folded cross validation*)

U modelima strojnog učenja dijelimo naš skup podataka na skup podataka za treniranje i skup podataka za testiranje kako bi mogli na skupu podataka za testiranje provjeriti točnost modela koji je naučen na skupu za treniranje. Ta podjela se obično vrši tako što 80% slučajno izabranih uzoraka iz skupa podataka završi u skupu za treniranje a 20% u skupu za testiranje. Ovu podjelu radimo kako bi testirali model i njegovu točnost prije nego što predvidi neke nove podatke. Ovo je dobar način da ispitamo točnost modela, ali postavlja se pitanje kojih 80% podataka uzeti u skup za treniranje zato što uzimamo slučajne uzorke za skup za treniranje i testiranje. Može se dogoditi da u jednoj podjeli dobijemo točnost modela od 85%, a u drugoj podjeli točnost modela 90% zbog različitih skupova za treniranje i testiranje. Zbog toga se uvodi postupak K-struka unakrsna provjera. U ovom postupku cilj je podijeliti skup podataka na više skupova za treniranje i testiranje tako da svi podaci budu i u jednom skupu i u drugom skupu kroz više iteracija. Prvo odabiremo na koliko preklopa (*eng. Folds*) želimo podijeliti skup podataka. U praksi je najčešće 10 preklopa[14]. Zatim se cijeli skup podataka dijeli na 10 skupova od kojih je prvih 10% skup za testiranje a ostalih 90% skup za treniranje. Nakon toga se model nauči na osnovu tih podataka, te se izračuna njegova točnost i ta vrijednost se spremi kao točnost prvog modela. Zatim se uzima drugih 10% skupa podataka kao skup za testiranje, a prvih 10% i zadnjih 80% kao skup za treniranje. Ponovo se model trenira i računa njegova točnost i ta vrijednost se sprema kao točnost drugog modela. Postupak se ponavlja dok se ne prođe kroz cijeli skup podataka tako da svaki od 10 preklopa jednom ne bude skup za testiranje. Na kraju se izračuna srednja vrijednost i standardna devijacija svih točnosti modela. Taj podatak nam govori koliko je naš model prosječno točan i koliko je srednje odstupanje od prosječne točnosti. Ovaj pristup je računski zahtjevniji, ali nam daje bolju sliku modela. Formula za računanje k-struka unakrsne provjere je :

$$CV_{(n)} = \frac{1}{n} * \sum_{i=1}^{n} Err_i \tag{8}$$

gdje je CV prosječna točnost modela s n preklopa, n je broj preklopa, Err predstavlja broj krivo klasificiranih opservacija [14].

Slika 7: K-struka unakrsna provjera

Drugi problem, osim što je računski zahtjevniji, je ako naprimjer imamo problem binarne klasifikacije. Zamislimo da se u prvoj podjeli u skupu za testiranje nalaze samo jedinice. To je problem zato što onda podjela nije balansirana, tj. nisu sve klase jednako zastupljene. U tome slučaju se koristi stratificirana unakrsna provjera. Ovaj postupak je sličan kao i obična unakrsna provjera, samo što osigurava da je u svakom preklopu svaka klasa približno jednako zastupljena, što povećava reprezentativnost pojedinog preklopa.
Također moram spomenuti još jednu podjelu, koja nije korištena u ovome radu, ali je iznimno bitna, a to je podjela na skup za treniranje, skup za validiranje i skup za testiranje [15]. U toj podjeli se originalni skup podataka dijeli na 75% podataka u skup za treniranje, na kojemu model uči, 15% podataka u skup za validaciju u kojem se obično odvija nepristrana evaluacija modela na skupu za treniranje i to najčešće dok se radi optimizacija hiperparametara, te ostalih 15% podataka ide u skup za testiranje u kojem se nalaze modelu neviđeni podaci do toga trenutka u vremenu te se na njima rad završna provjera točnosti modela.

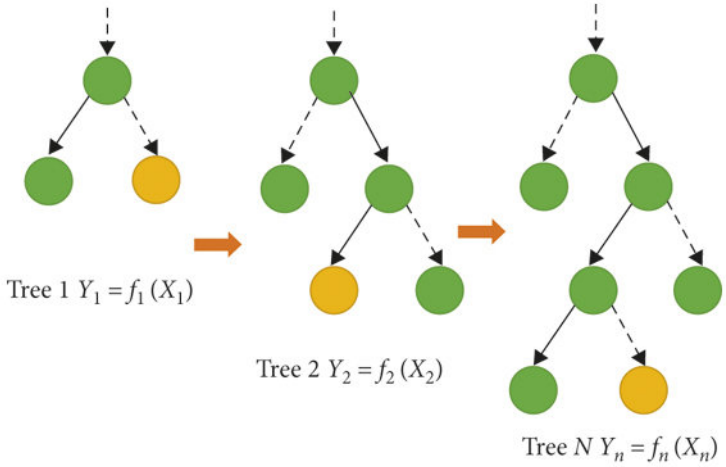
3 Gradient Boosting Machine (GBM)

GBM je tehnika strojnog učenja za regresijske, klasifikacijske i ostale probleme koja se temelji na ansamblu slabih modela predikcije, najčešće stabala odluke. Ovu je ideju prvi razvio Jerome H. Friedman u svome djelu *„Greedy function approximation: A gradient boosting machine“*[16]. Temelji se na tehnici *boosting* koja radi tako da algoritam uči na pogreškama prethodnih algoritama, te s puno slabih klasifikatora postupno dobivamo jedan jaki klasifikator[17]. Jedan od prvih i najpopularnijih *boosting* modela je *AdaBoost*. *AdaBoost* se temelji na stablima odluke koja obično imaju jedan čvor i dva lista. Takva stabla se nazivaju panjevi (*eng. Stump*). *AdaBoost* stvara šumu takvih stabala. Panjevi sami po sebi ne donose precizne klasifikacije, ali kada se napravi šuma panjeva u kojoj svaki panj uči na pogreškama svojih prethodnika, dobije se vrlo dobar klasifikator. Svaki novi panj koji se stvara prilagođava svoje težine na temelju pogreški prethodnih panjeva. GBM koristi drugačiji pristup nego *AdaBoost*. Iako se, isto kao i *AdaBoost*, temelji na netočnim predikcijama prethodnika, GBM podešava svako novo stablo na temelju pogreški predikcije prethodnog stabla. To znači da GBM gleda pogreške, i onda gradi novo drvo na osnovu tih pogrešaka, a predikcije koje su točne nemaju utjecaj na novo drvo. GBM koristi ostatke (*eng. Residual*) čija je vrijednost razlika između predviđenih vrijednosti modela i stvarnih vrijednosti. Generalna ideja GBM: „ Gradient boosting računa ostatke previđanja svakog drveta i zbraja ih kako bi ocijenio model“[18]. GBM algoritam počinje tako što se u prvom koraku napravi list, za razliku od *AdaBoosta* gdje se napravi panj. Još jedna razlika između *AdaBoosta* i GBM je što GBM pravi drveća koja mogu imati više unutarnjih čvorova, ali i taj broj je ograničen algoritmom, tj jednim od hiperparametara. Početni list predstavlja inicijalnu pretpostavku za izlaznu značajku svih uzoraka. Ako se GBM koristi za regresiju za inicijalnu pretpostavku se uzima srednja vrijednost izlazne značajke. U slučaju klasifikacije, za koju ću opisati algoritam GBM-a, za inicijalnu pretpostavku uzima se logaritam omjera šansi[19] (*eng. log(odds)*) svih izlaznih značajki. Nakon toga računamo vjerojatnost inicijalne pretpostavke pomoću logističke funkcije prema formuli $\frac{e^{log(odds)}}{1+e^{log(odds)}}$. Zatim se računaju pseduo ostaci (*eng. „Pseudo“-residuals*) tako što, za svaku opservaciju, oduzimamo vjerojatnost njene izlazne značajke (primjera radi ako je klasificirana kao „da“ uzimamo 1, a ako je klasificirana kao „ne“ uzimamo 0) s vrijednošću inicijalne pretpostavke. Te vrijednosti, odnosno ostatke spremamo u novi stupac i one će nam služiti za kreiranje ostalih stabala. Nakon toga GBM kreira prvo stablo na osnovu ostataka i ulaznih značajki za svaku opservaciju. Početno stablo naziva se bazni učenik (*eng. Base learner*). Ovo stablo je formirano na osnovu vjerojatnosti, dok je naša inicijalna predikcija formirana

na osnovu *log(odds)*, te zbog toga moramo napraviti transformaciju izlaznih vrijednosti koje smo dobili u listovima stabla nazad u *log(odds)*. Najčešća formula za transformaciju koja se koristi u GBM-u je:

$$\frac{\sum \text{Ostatak u listu}_i}{\sum[\text{Prethodna Vjerojatnost}_i * (1 - \text{Prethodna Vjerojatnost}_i)]} \tag{9}$$

Nakon što dobijemo transformirane izlazne vrijednosti za svaki list, ažuriramo *log(odds)* za izlazne značajke tako što zbrajamo inicijalnu pretpostavku i izlaznu vrijednost lista u novonastalom stablu za svaku opservaciju u skupu podataka. Prije nego što obavimo zbrajanje, moramo odrediti parametar stope učenja (*eng. Learning Rate*). Taj parametar množi svako novonastalo stablo u algoritmu. Stopa učenja je bitna zato što kada ovog parametra ne bi bilo, i odradimo zbrajanje dobili bi istu vrijednost izlazne značajke koju smo imali na početku. Vrijednost stope učenja je između 0 i 1. Sada se može izvršiti ažuriranje, te se dobivaju nove vrijednosti *log(odds)* za svaku izlaznu vrijednost koje moramo opet pretvoriti u vjerojatnosti. Te vjerojatnosti spremamo u novi stupac, koji zovemo predviđene vjerojatnosti, te će nam one služiti za ponovno računanje ostataka u drugoj iteraciju algoritma. U drugoj iteraciji algoritam uzima predviđene vjerojatnosti i vjerojatnosti izlaznih značajki, te isto kao u prvom koraku oduzima vjerojatnost izlazne značajke (1 ili 0) s novom predviđenom vjerojatnosti i sprema ih u stupac za ostatke. Na osnovu tih ostataka i ulaznih značajki formira se novo stablo, s listovima koji imaju neke izlazne vrijednosti. Te vrijednosti se opet pomoću jednadžbe 9 transformiraju u *log(odds)* vrijednosti. Sada se pomoću inicijalne predikcije, stope učenja koja utječe sada i na prvo i na drugo stablo, te izlaznih vrijednosti lista, ažurira stupac s previđenim vjerojatnostima. Ovaj postupak se ponavlja sve dok nismo napravili maksimalni broj stabala koji smo postavili prije, ili dok ostaci ne budu jako mali. U praksi se najčešće rade stabla koja imaju 8-32 listova[20].



Slika 8: GBM algoritam [21]

3.1 Hiperparametri *Gradient Boosting* klasifikatora

Gradient Boosting klasifikator implementira se pomoću klase u već prije spomenutoj knjižnici *Scikit-Learn*. Ova klasa naziva se *GradientBoostingClassifier*. Hiperparametri s kojima dolazi ovaj klasifikator možemo podijeliti na tri vrste:

- Parametri za pojedino stablo
- *Boosting* parametri
- Ostali parametri

3.1.1 Parametri za pojedino stablo

Sljedeći parametri utječu na svako individualno stablo :

- *min_samples_split*: Određuje minimalni broj opservacija koje su potrebne u čvoru da bi bio razmatran za dijeljenje. Najčešće se koristi za sprječavanje prenaučenosti. Veće vrijednosti sprečavaju model da uči odnose koji bi mogli biti previše specifični. Prevelike vrijednosti mogu dovesti do podnaučenosti – potrebno je optimizirati koristeći unakrsnu provjeru.
- *min_samples_leaf*: Određuje minimalni broj opservacija u listu. Koristi se u slične svrhe kao i *min_samples_split*.

- *min_weight_fraction_leaf*: Slično kao *min_samples_leaf*, ali definiran kao razlomak ukupnog broja opservacija umjesto cijelog broja.

- *max_depth*: Maksimalna dubina stabla. Što je stablo dublje, veća vjerojatnost prenaučenosti. Preporuka da se podešava koristeći unakrsnu provjeru.

- *max_leaf_nodes*: Maksimalni broj listova u stablu. Ako je ovaj parametar definiran, GBM će ignorirati *max_depth*, zato što se kreiraju binarna stabla, a ako imamo stablo dubine „n“ kreirat će se „2n“ listova.

- *max_features*: Broj značajki koje ulaze u obzir kada se traži najbolja podjela. Veće vrijednosti mogu dovesti do prenaučenosti.

3.1.2 Parametri za *boosting*

- *learning_rate*: Kontrolira stopu učenja. Niže vrijednosti su generalno bolje zato što rade robusniji model za specifične karakteristike stabla, ali u tome slučaju zahtjeva se veći broj stabala kako bi se pronašle sve relacije što dovodi do računski zahtjevnijih operacija.

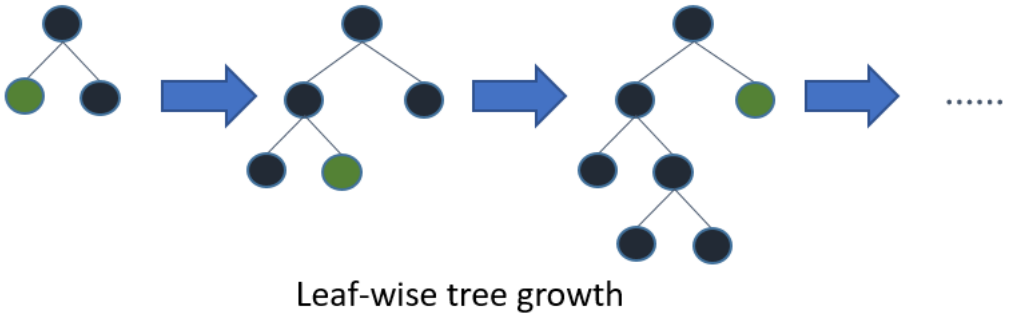
- *n_estimators*: Broj drveća u modelu. Treba se podešavati koristeći unakrsnu provjeru za pojedinu stopu učenja.

- *subsample*: Koliki dio opservacija koji će biti odabran za svako stablo. Vrijednosti malo manje od 1 (obično oko 0.8) smanjuju varijancu u modelu.

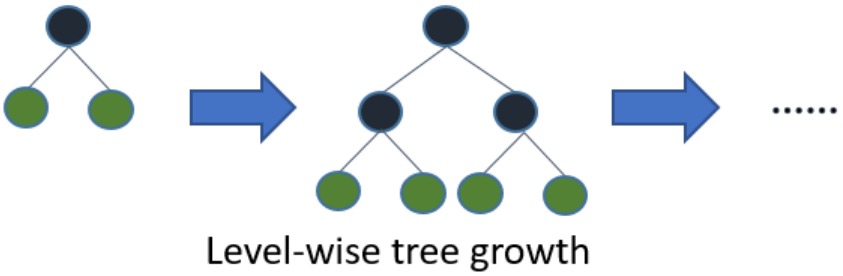
Ostali parametri nisu toliko bitni za automatizaciju hiperparametara pa ih neću navoditi u ovom diplomskom radu.[22]

3.2 *LightGBM* klasifikator

LightGBM je *gradient boosting framework* koji koristi algoritme učenja koji su osnovani na stablima učenja. Odlikuju ga velika brzina učenja i efikasnost, mala potrošnja memorije te velika preciznost. Također prema istraživanju napravljenom u članku "*LightGBM: A Highly Efficient Gradient Boosting Decision Tree*"[23] dokazano je na velikom broju javnom dostupnih skupova podataka da je *LightGBM* do 20 puta brži od ostalih implementacija istih algoritama kao što je *XGBoost*[24]. *LightGBM* koristi tzv. *leaf wise tree growth*, dok ostale implementacije koriste tzv. *level wise tree growth*. Glavna razlika između ova dva načina je što se u *level wise tree growth-u* drveća šire



Slika 9: leaf wise tree growth



Slika 10: level wise tree growth

tako da se odabere čvor s najvećim delta gubitkom, dok se u *level wise tree growth-u* drveća šire po nivoima, tj dubinama drveća.

LightGBM klasifikator dolazi s mnoštvo hiperparametara od kojih ću navesti samo one koje ću koristiti u istraživanju. Većinu ovih parametara sam spominjao u prethodnim poglavljima.

- *learning_rate*: Parmetar koji kontrolira stopu učenja.
- *max_depth*: Ograničava koliko duboko će jedno drvo rasti.
- *num_leaves*: Ovaj parametar kontrolira maksimalan broj čvorova u drvetu, te je najvažniji parametar koji kontrolira kompleksnost modela. Zato što *LightGBM* koristi *leaf wise tree growth* ovaj parametar bi se trebao postaviti na vrijednost manju od 2^{max_depth} te ako je ovaj parametar veći od tog broja može doći do prenaučenosti.
- *n_estimators*: Broj drveća koji će se napraviti u modelu.

- *min_child_samples*: Određuje minimalan broj podataka u jednom listu. Također važan parametar koji sprječava prenaučenost. Najčešće se postavlja na stotine ili tisuće za veliki skup podataka.
- *reg_lambda* i *reg_alpha*: Parametri koji kontroliraju L1 i L2 regularizaciju koja pomaže da se smanji prenaučenost.

4 Tehnike optimizacije hiperparametara

Hiperparametri modela se postavljaju prije nego što model krene učiti. Taj postupak izbora hiperparametara možemo provesti ručno isprobavajući razne vrijednosti pojedinog hiperparametra i svaki put kada promijenimo hiperparametar pustiti model da ponovo uči. Takav način zahtijeva previše vremena i stalnu izmjenu koda. Zbog ovakvih problema uvedena je automatska optimizacija hiperparametara. U ovom diplomskom radu koristit ćemo 5 metoda automatske optimizacije: Pretraživanje po rešetci (*eng. Grid search*), Nasumično pretraživanje (*eng. Random search*), optimizaciju temeljenu na Bayesovom teoremu (*eng. Bayesian optimization*) (nadalje Bayesova optimizacija) te *Hyperopt* knjižnicu koja sadrži dvije tehnike optimizacije: *Simmulated Anneal* i *Tree-structured Prazen Estimator*.

4.1 Pretraživanje po rešetci (*eng. Grid search*)

Prva metoda koju ću opisati je pretraživanje po rešetci. Pretraživanje po rešetci je iscrpna metoda koja se temelji na pretraživanju podskupa hiperparametarskog prostora[25]. Ova metoda zahtijeva definiranje naziva hiperparametra te vrijednosti za koje želimo provjeriti točnost modela. Nakon što se postave vrijednosti hiperparametara, metoda provjerava točnost modela za svaku kombinaciju hiperparametara. Ovo metoda je računski vrlo zahtjevna, pogotovo ako imamo veliki broj parametara koje želimo optimizirati, te je skup podataka vrlo velik. Iako je računski zahtjevna metoda, pretraživanje po rešetci obično daje bolje rezultate nego potpuno manualna optimizacija kada bi se uspoređivali po utrošenom vremenu. Također pretraživanje po rešetci je pouzdano za mali broj parametara (1-2 parametar)[26]. Pretraživanje po rešetci u *Pythonu* je implementirano u sklopu već spomenute *Scikit-learn* knjižnice. Klasa pomoću koje je implementirana naziva se *GridSearchCV*[7].

```
params_grid_search = {'learning_rate': [0.001, 0.01, 0.5],
          'num_leaves': [10, 20, 150, 300],
          'n_estimators': [500, 1500, 3000],
          'max_depth': [3, 8, 12],
          'min_child_samples': [5, 30, 150],
          'reg_lambda': [1, 50, 99],
          'reg_alpha' : [1, 50, 99]}
grid_search = GridSearchCV(estimator=LGBMClassifier
                           (random_state=42),
                           param_grid=params_grid_search,
                           scoring="accuracy",
                           cv=10,
                           n_jobs=-1,
                           return_train_score=True,
                           verbose=10)
grid_search.fit(X, y)
```

Listing 1: Implementacija pretraživanja po rešetci

Prema 1. listingu vidimo implementaciju pretraživanja po rešetci za LGBM klasifikator. Prvo smo napravili hiperparametarski prostor u kojem smo definirali nazive hiperparametara te njihove vrijednosti koje želimo testirati. Metoda *GridSearchCV* prima nekoliko parametara. Prvi od njih je *estimator* i njemu predajemo model koji smo odabrali. Pod *param_grid* predajemo hiperparametarski prostor kojeg smo prije definirali. *Scoring* parametar prima jednu od metoda po kojoj se procjenjuje model. *Cv* parametar prima broj iteracija unakrsne provjere. Pomoću *.fit* metode treniramo model s parametrima koji su izabrani u trenutnoj iteraciji.

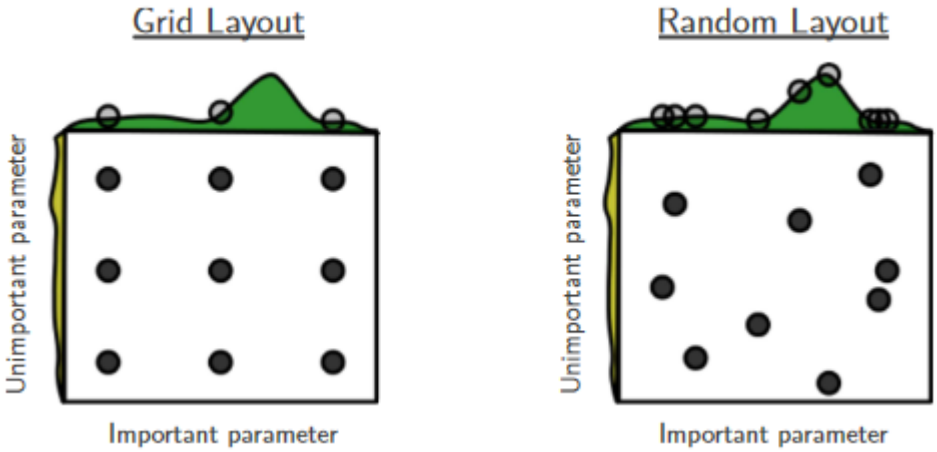
4.1.1 Pseudokod pretraživanja po rešetci

```
for c in [2^-5, 2^-3,...,2^15]:
 for g in [2^-15, 2^-13, ..., 2^5]:
  for train, test in partition:
    model = svm_train(train, c, g)
    score = svm_predict(test, model)
    cv_list.insert (score)
  scores_list.insert(mean(cv_list),c,g)
print max(scores_list)
```

Slika 11: Pseudokod pretraživanja po rešetci[27]

4.2 Nasumično pretraživanje (*eng. Random search*)

Nasumično pretraživanje je slično kao i pretraživanje po rešetci, samo što umjesto manualnog unosa vrijednosti hiperparametara, u hiperparametarskom prostoru definiramo raspon vrijednosti koje hiperparametar može poprimiti. Kao što i ime metode govori, te vrijednosti se nasumično biraju pomoću izabrane distribucije. Ovakav pristup nam omogućava, za razliku od pretraživanja po rešetci, veću fleksibilnost u izboru vrijednosti hiperparametara. Iako su te vrijednosti slučajne, istraživanje koje su proveli James Bergstra i Yoshua Bengio dokazuje da su te vrijednosti efikasnije nego manualno postavljenje vrijednosti i pretraživanje po rešetci[26]. Ovo se pogotovo odnosi na više dimenzionalne hiperparametarske prostore, za koje pretraživanje po rešetci oduzima puno vremena. Najveća prednost kod nasumičnog pretraživanja je odabir broja iteracija koji uvelike smanjuje vrijeme trajanja procesa optimizacije.



Slika 12: Razlika između pretraživanja po rešetci i nasumičnog pretraživanja[26]

Na slici 12 vidimo razliku u pretraživanju prostora između gore spomenute dvije metode. U devet iteracija pretraživanje po rešetki je provjerilo samo 3 stanja za, u ovom slučaju, bitan hiperparametar, dok je nasumično pretraživanje dobilo devet različitih stanja za taj isti hiperparametar i pri tome pronašlo optimalnu vrijednost tog hiperparametra. Nasumično pretraživanje implementira se pomoću *scikit-learn* knjižnice u *Pythonu*. Klasa koja omogućava tu implementaciju naziva se *RandomizedSearchCV*[7].

```
params_random_search = {'learning_rate' : uniform(0.01,0.99),
          'num_leaves': randint(20,3000),
          'n_estimators': randint(500,3000),
          'max_depth': randint(3,12),
          'min_child_samples': randint(2,300),
          'reg_lambda': uniform(0.01,99.99),
          'reg_alpha': uniform(0.01,99.99),
          }
random_search = RandomizedSearchCV(estimator = LGBMClassifier
                                   (random_state=42),
                                   param_distributions =
                                   params_random_search,
                                   n_iter = 1500,
                                   scoring = "accuracy",
                                   cv = 10,
                                   random_state = 42,
                                   n_jobs = -1,
                                   verbose=10
                                   )
random_search.fit(X, y)
```

Listing 2: Implementacija nasumičnog pretraživanja

Ovoj metodi, kao i metodi kod pretraživanja po rešetkom, moramo proslijediti nekoliko parametara. Prosljeđujemo isto naš model, hiperparametarski prostor, koji smo ovaj put definirali malo drugačije. Moramo opet definirati nazive hiperparametara, ali njihove vrijednosti sada definiramo kao raspone koji se nasumično izaberu. U ovom slučaju za hiperparametre *learning_rate*, *reg_lambda* i *reg_alpha* koristimo uniformu distribuciju. Sljedeći parametar *n_iter* označava koliko puta će se provesti nasumična pretraga, te na kraju parametar *cv* označava broj unakrsnih provjera.

4.2.1 Pseudokod nasumičnog pretraživanja

PSEUDO code for Random Search (RS) algorithm

Step-1: Randomly choose an initial value of u_0, where $u_0 \in U$

Step-2: Calculate $f(u_0)$

Step-3: Set k=0

Step-4: Generate a new independent value $u_{new}(k+1) \in U$ according to chosen probability distribution

Step-5: *if* $f(u_{new}(k+1)) < f(u_k)$

Set $u_{k+1} = u_{new}(k+1)$

else $u_{k+1} = u_k$

Step-6: stop if the maximum number of f evaluations has been reached; else return to **Step-1** with the new k set to the former k+1.

Slika 13: Pseudokod nasumičnog pretraživanja[28]

4.3 Bayesova optimizacija (*eng. Bayesian Optimization*)

Bayesova optimizacija je veoma dobra strategija za pronalaženje ekstrema funkcija gubitaka koje su „skupe“ za računanje[29]. Spada u klasu optimizacijskih algoritma koji se nazivaju „Optimizacija temeljena na sekvencijalnim modelima“ (*eng. Sequential Model-Based Optimization*) (SMBO). Ovi algoritmi koriste prethodne vrijednosti funkcije gubitka kako bi odredili sljedeću (optimalnu) točku za novu vrijednost funkcije gubitka. Bayesova optimizacija temelji se na tome da pravi novu funkciju, na osnovu funkcije gubitka, koja se naziva zamjenska/surogat funkcija (*eng. Surrogate function*). Zove se Bayesova optimizacija zato što je temeljena na Bayesovom teoremu koji nam govori da su *posteriorne* (*eng. posterior*) vjerojatnosti modela M s danim podacima E proporcionalne izglednosti (*engl. Likelihood*) od E uvjetovano M pomnoženo sa *prior* vjerojatnošću od M; što je opisano sljedećom formulom [29]

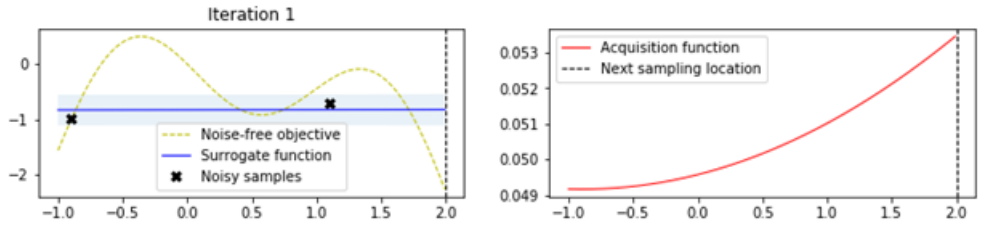
$$P(M|E) \propto P(E|M) * P(M) \tag{10}$$

Pomoću ove jednadžbe možemo optimizirati našu funkciju gubitka. U Bayesovoj optimizaciji, *prior* predstavlja prostor mogućih funkcija gubitaka, tj. predstavlja neko nesigurno područje u kojem se može nalaziti funkcija gubitka prije nego što se neki uvjet uzme u obzir. Našu funkciju gubitka tretiramo kao „crnu kutiju“ (*eng. Black-box*) što znači da nemamo nikakve informacije o njoj, ne znamo formulu funkcije da ju analiziramo, te ne znamo njene derivacije. Također se procjena funkcije zasniva na tome da smo ograničeni

na to da za neku točku „x“ dobijemo (najvjerojatnije s pogreškom) neki odgovor od nje. Iako je funkcija gubitka nepoznata, možemo pretpostaviti da postoji neko prijašnje znanje on nekim njenim svojstvima, kao što je zakrivljenost. Bayesova optimizacija sastoji se o dvije glavne komponente:

- Bayesov statistički model za oblikovanje funkcije gubitka
- Funkcije stjecanja (*eng. Acquisition function*) koja odlučuje sljedeću vrijednost za provjeru maksimuma

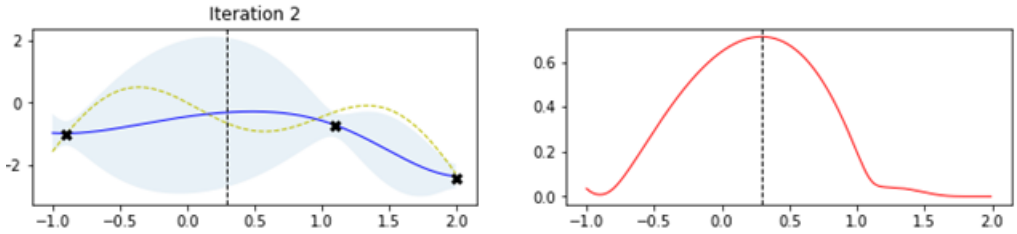
Statistički model, koji se temelji na Gaussovom procesu, pruža Bayesovu *posterior* distribuciju vjerojatnosti koja opisuje moguće vrijednosti funkcije gubitka *f(x)* za odabranog kandidata x. Svaki put kada promatramo funkciju gubitka na novom mjestu, ta distribucija je ažurirana. Funkcija stjecanja računa vrijednost koja bi se dobila iz procjene funkcije gubitka na novom mjestu x, temeljeno na trenutnoj *posteriornoj* distribuciji prema funkciji gubitka. Djelovanje Bayesove optimizacije prikazat ću i objasniti na primjeru.



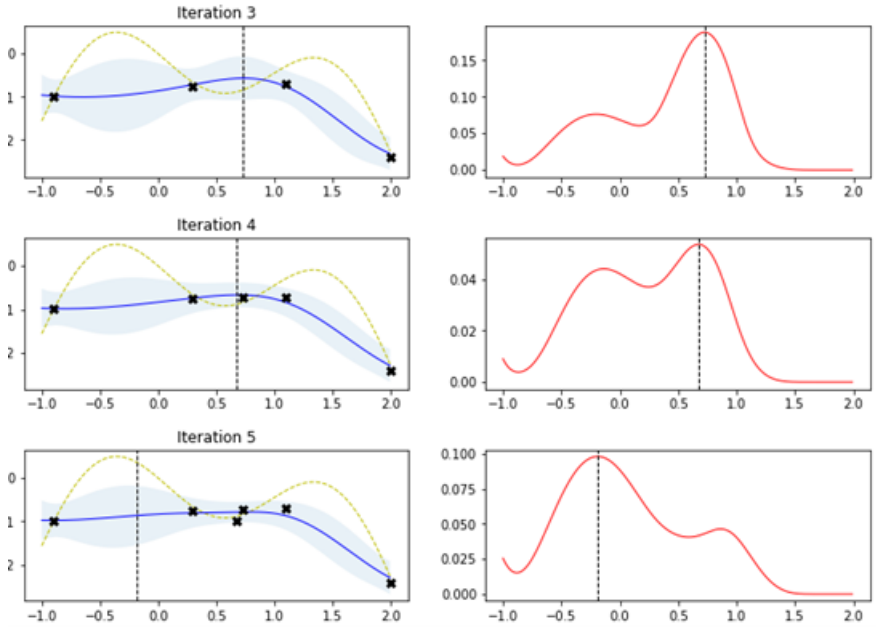
Slika 14: Prva iteracija Bayesove optimizacije

Na slici 14 i na budućim slikama, na lijevom grafu, žutom isprekidanom linijom označena je funkcija gubitka koju želimo pronaći, tj. odrediti njen globalni maksimum, a plavom punom linijom zamjenska funkcija. Svijetlo plavom bojom je označen prostor mogućih vrijednosti funkcije gubitka (*posterior*) koji smo do sada otkrili sa zamjenskom funkcijom. Crnim iksom su označeni kandidati po kojima se određuje izgled sljedeće zamjenske funkcije. Na desnom grafu je funkcija stjecanja prema kojoj se određuje sljedeći kandidat.
U prvoj iteraciji, koja je prikazana na slici 14, zamjenska funkcija je postavljena kao običan pravac, te je određen prostor mogućih vrijednosti. Gledamo maksimum funkcije stjecanja i prema njemu uzimamo sljedećeg kandidata.

Na slici 15 vidimo promjene na zamjenskoj funkciji kada se uzme u obzir novi kandidat, te promjene u prostoru mogućnosti. Također se mijenja i funkcija stjecanja. Sada vidimo da prostor stanja i zamjenska funkcija malo više odgovaraju pravoj funkciji gubitka. Opet uzimamo maksimum funkcije stjecanja za novog kandidata.

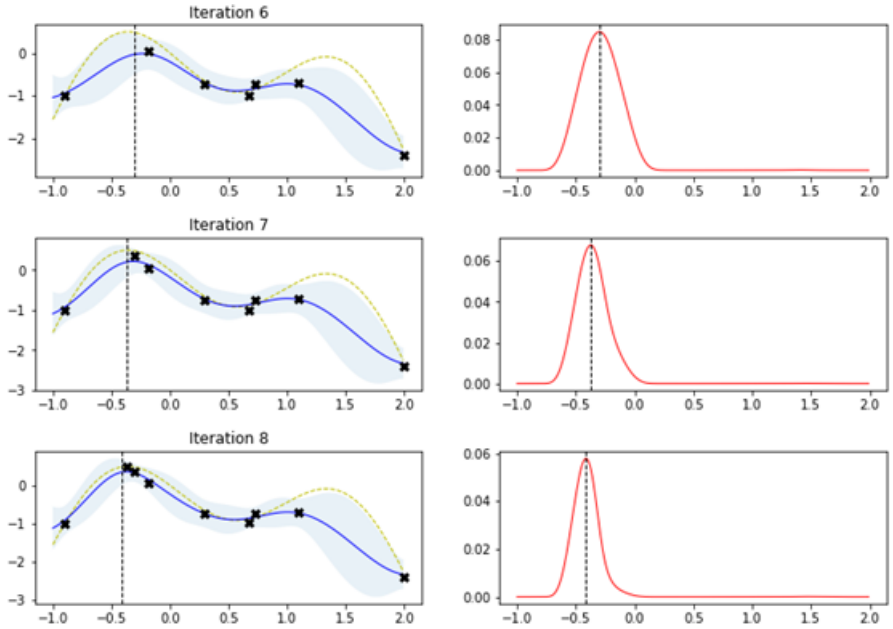


Slika 15: Druga iteracija Bayesove optimizacije



Slika 16: 3., 4. i 5. iteracija Bayesove optimizacije

Na slici 16 vidimo sljedeće 3 iteracije. Vidimo da algoritam konvergira prema lokalnom minimumu funkcije gubitka na osnovu maksimuma funkcije stjecanja, ali se povećao prostor mogućnosti kod globalnog maksimuma funkcije gubitka, te algoritam u sljedećoj iteraciji uzima kandidata iz tog područja.



Slika 17: 6., 7. i 8. iteracija Bayesove optimizacije

Na slici 17 vidimo kako se kroz iteracije izgled zamjenske funkcije lagano približava izgledu prave funkcije gubitka, maksimum funkcije stjecanja se približava vrijednosti maksimuma funkcije gubitka, te dobivamo kandidate sve bliže toj maksimalnoj vrijednosti i naposljetku idealnog kandidata.

Bayesova optimizacija u *Pythonu* se može provesti kroz nekoliko klasa, tj funkcija. U ovom diplomskom radu bit će opisana i u praktičnom dijelu korištena funkcija *gp_minimize* koja je dio *scikit-optimize* programske knjižnice[30]. Ova programska knjižnica sadrži u sebi implementaciju za optimizaciju temeljenu na sekvencijalnim modelima, te omogućava vizualizaciju dobivenih rezultata. Prvo što moramo napraviti za Bayesovu optimizaciju je definirati funkciju gubitka koju želimo minimizirati. U našem primjeru to će biti funkcija koja računa k-struka unakrsnu provjeru.

```
def optimize_bayesian(params, param_names ,x, y):
    params = dict(zip(param_names, params))
    model = LGBMClassifier(**params)
    return -1.0 * np.mean(cross_val_score(model, x, y, cv=10,
    scoring="accuracy", n_jobs=-1))
```

Listing 3: Definiranje funkcije gubitka

Jedan od argumenata *cross_val_score* funkcije(koja računa k-struka unakrsnu provjeru) je i *scoring* parametar. Prema dokumentaciji koja postoji na *scikit-learn* stranici tom argumentu predajemo *string* s metrikom prema kojoj želimo napraviti k-struka unakrsnu provjeru. U ovom diplomskom radu koristit ćemo 3 metrike a to su *"accuracy"*(točnost), ROC_AUC i F1 metrike. Te metrike ću opisati u sljedećem potpoglavlju. Sljedeće što definiramo je hiperparametarski prostor u kojem se nalaze intervali između kojih tražimo vrijednosti parametara. Na kraju pozivamo funkciju *gp_minimize* koja kao argumente prima funkciju koju želimo minimizirati, hiperparametarski prostor (argument *dimensions*), broj puta koliko želimo odraditi Bayesovu optimizaciju (*n_calls*). Argument *n_initial_points* određuje koliko će se puta procijeniti funkcija gubitka prije nego što krene aproksimirati sa procjeniteljem (statistički model temeljen na Gaussovom procesu).

```
param_space = [
                space.Real(0.001,1, prior = "uniform",
                name = "learning_rate"),
                space.Integer(20,3000, name = "num_leaves"),
                space.Integer(500,3000, name = "n_estimators"),
                space.Integer(3,12, name = "max_depth"),
                space.Integer(2,300, name = "min_child_samples"),
                space.Real(0.01,100, prior = "uniform",
                name = "reg_lambda"),
                space.Real(0.01,100, prior = "uniform",
                name = "reg_alpha")
]

param_names=[
             "learning_rate",
             "num_leaves",
             "n_estimators",
             "max_depth",
             "min_child_samples",
             "reg_lambda",
             "reg_alpha",
]

optimization_function_bayesian = partial(optimize_bayesian,
                                         param_names = param_names,
                                         x = X,
                                         y= y)
result_bayesian = gp_minimize(optimization_function_bayesian,
                              dimensions = param_space,
                              n_calls = 200,
                              n_initial_points=10,
                              verbose=10)
```

Listing 4: Implementacija Bayesove optimizacije

4.3.1 Točnost (*accuracy*) metrika

Ova metrika je najjednostavnija od sve tri prije spomenute. Pomoću ove metrike možemo izračunati koliko je naš model točan u predviđanju klasa.

Accuracy ili točnost nam govori koliko je naš model predvidio točnih predikcija u odnosu na ukupan broj opservacija koji pripadaju toj klasi. Točnost se računa pomoću sljedeće formule

$$\text{Točnost} = \frac{\text{Broj točnih predikcija}}{\text{Ukupan broj predikcija}} \tag{11}$$

Ova metrika je dosta dobar način da vidimo koliko je naš model točan, ali problem dolazi kod veoma osjetljivih predikcija, kao što je predviđanje tumora. Da bih ovo objasnio moram prvo spomenuti matricu konfuzije (*eng. Confusion Matrix*). Matrica konfuzije se, u binarnoj klasifikaciji, sastoji od dva retka i dva stupca.

Matrica konfuzije (confusion matrix)		**Stvarna klasa**	
		G (+) Pozitivni	**NG (-) Negativni**
Predviđeno modelom *h*	**G (+) Pozitivni**	TP	FP
	NG (-) Negativni	FN	TN

Slika 18: Matrica konfuzije

Prema slici 18 vidimo da su u redci ono što je naš model predvidio, a u stupcima ono što je trebao predvidjeti. Matrica konfuzije ima 4 grupe, a to su:

- TP (*True Positive*): U ovoj grupi su nam broj stvarno pozitivnih opservacija i broj stvarno pozitivno predviđenih opservacija. Možemo reći da je to broj opservacija koje je model predvidio da će biti pozitivne.
- FP (*False Positive*): U ovoj grupi se nalaze opservacije koje su negativne, ali je naš model predvidio da će biti pozitivne.
- FN (*False Negative*): U ovoj grupi je broj opservacija koje je naš model predvidio da su negativne a ustvari su pozitivne.
- TN (*True Negative*): U ovoj grupi je broj opservacija koje je model predvidio da su negativne, a ustvari i jesu negativne, tj. model ih je dobro predvidio.

True positive i *true negative* grupe su točne predikcije našeg modela, ali *false positive* i *false negative* su netočne predikcije. Točnost se još može iskazati kao i:

$$\text{Točnost} = \frac{TP + TN}{TP + FN + FP + TN} \tag{12}$$

Kao što sam spomenuo gore u tekstu, problem dolazi kod osjetljivih predikcija. Naš model može imati točnost od 90%, ali imati naprimjer 15 *false negative* predikcija i 5 *false positive* predikcija. Ako se radi o predviđanju tumora, *false negative* grupa ili predikcija je puno gora predikcija nego *false positive* zato što je naš model predvidio da osoba neće imati tumor ali ga u stvarnosti ima, što je puno gore nego da smo predvidjeli da osoba ima tumor, ali ga u stvarnosti nema, pa može napraviti još par testova kod liječnika da se utvrdi obrnuto. Zbog ovakvih slučajeva računamo ROC_AUC i *F1* metrike.

4.3.2 ROC_AUC metrika

AUC stoji za *Area Under Curve* ili površina ispod krivulje i jedna je od najraširenijih metrika koja se koristi u binarnoj klasifikaciji. AUC nekog klasifikatora je jednak vjerojatnosti da će klasifikator predvidjeti neki slučajan uzorak da je pozitivan više nego da će predvidjeti da je negativan. Također moram spomenuti još par formula i pojmova a to su TPR (*eng. True Positive Rate*) ili osjetljivost (*eng. Sensitivity*) koja pokazuje omjer između točno predviđenih pozitivnih uzoraka i svih pozitivnih uzoraka u skupu podataka ,a računa se sljedećom formulom:

$$\text{Osjetljivost} = \frac{TP}{TP + FN} \tag{13}$$

, TNR (*eng True Negative Rate*) ili specifičnost (*eng. Specificity*) koja pokazuje omjer između točno predviđenih negativnih uzoraka i svih negativnih uzoraka u skupu podataka, a računa se sljedećom formulom:

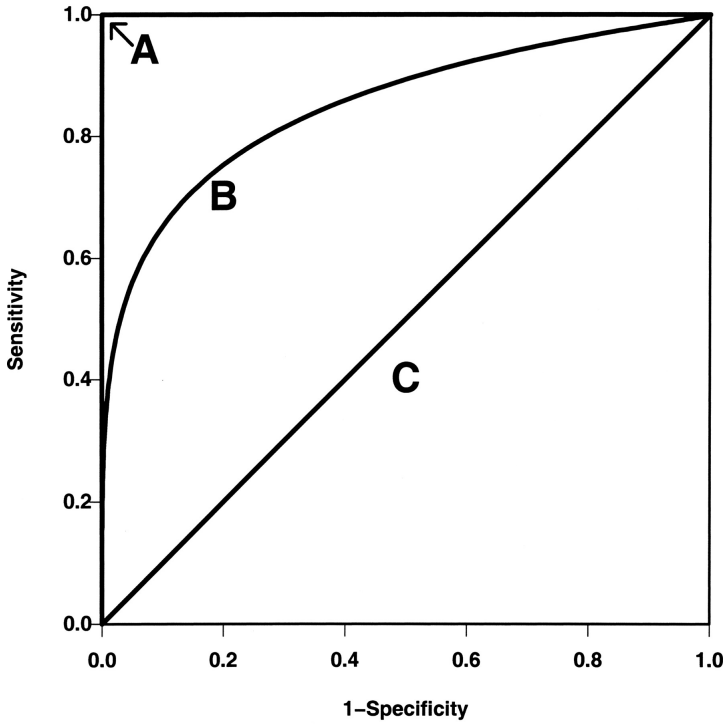
$$\text{Specifičnost} = \frac{TN}{TN + FP} \tag{14}$$

, te FPR (*False Positive Rate*) što predstavlja omjer pozitivno predviđenih uzoraka koji su ustvari negativni i svih negativnih uzoraka u skupu podataka. Formula za FPR izgleda:

$$\textit{False Positive Rate} = \frac{FP}{TN + FP} \tag{15}$$

Također *False Positive Rate* se može dobiti i tako što se od 1 oduzme specifičnost.

ROC (*eng. Receiver Operating Characteristic*) je omjer između *True Positive Ratea* i *False Positive Ratea* i predočava se sljedećim grafom:



Slika 19: ROC krivulja

Na slici 19 slovom A je predočen idealni model čiji ROC_AUC iznos je 1, dok je sa slovom B predočen veoma dobar model. Slovom C je predočen jako loš model[31]. Što je ROC_AUC iznos veći to je i model bolji.

4.3.3 F1 metrika

F1 metrika nam govori koliko je naš model precizan a i u isto vrijeme robustan. Da bi pokazao formulu za F1 moram spomenuti još dva pojma. Prvi je preciznost (*eng. Precision*). Preciznost je omjer između točno predviđenih pozitivnih uzoraka i svih pozitivno predviđenih uzoraka.

$$\text{Preciznost} = \frac{TP}{TP + FP} \tag{16}$$

Sljedeći pojam je opoziv (*eng. Recall*). To je omjer između točno predviđenih pozitivnih uzoraka i svih uzoraka koji su trebali biti točno predviđeni. Formula za opoziv je:

$$\text{Opoziv} = \frac{TP}{TP + FN} \tag{17}$$

F1 pokušava pronaći balans između preciznosti i opoziva pomoću sljedeće formule:

$$\text{F1} = 2 * \frac{1}{\frac{1}{\text{Preciznost}} + \frac{1}{\text{Opoziv}}} \tag{18}$$

4.4 *Hyperopt*

Hyperopt[32] je također, kao i *sckit-optimize*, programska knjižnica za optimizaciju hiperparametara. U *Hyperoptu* su trenutno implementirana 3 algoritma:

- *Tree-structured Prazen Estimator (TPE)*
- *Simulated Anneal (SA)*
- Nasumično pretraživanje

4.4.1 *Tree-structured Prazen Estimator (TPE)*

Tree-structured Prazen Estimator gradi model zamjenske funkcije koristeći Bayesov teorem. Osim, kao što je navedeno u jednadžbi 10, Bayesov teorem se može zapisati na sljedeći način[33]:

$$p(y|x) = \frac{p(x|y) * p(y)}{p(x)} \tag{19}$$

, gdje $p(x|y)$ predstavlja vjerojatnost hiperparametara s obzirom na rezultat funkcije gubitka, što se može iskazati u sljedećem zapisu [34]:

$$p(x|y) = \begin{cases} l(x), \text{if } y < y* \\ g(x), \text{if } y \geq y* \end{cases} \tag{20}$$

Ovu jednadžbu možemo opisati kao dvije različite distribucije hiperparametara. Izraz $y < y*$ predstavlja nižu vrijednost funkcije gubitka od nekog postavljenog praga, a funkcija $l(x)$ je distribucija kada je vrijednost funkcije manja od postavljenog praga, dok funkcija $g(x)$ predstavlja distribuciju kada je vrijednost veća. *Tree-structured Parzen Estimator* algoritam ovisi o $y*$ koji je veći od najbolje promatrane funkcije gubitka da bi se mogla formirati $l(x)$.

4.4.2 *Simulated Anneal (SA)*

Ime ovog algoritma dolazi od tehnike *annealing* u metalurgiji koja uključuje zagrijavanje i kontrolirano hlađenje materijala kako bi se promijenila fizikalna svojstva [35]. *Simmulated Anneal* minimizira funkciju gubitka na sljedeći način. Algoritam generira nekog nasumičnog početnog kandidata $x*$ za kojeg izračuna funkciju gubitka. Nakon toga generira x_i u susjedstvu $x*$ te za njega računa funkciju gubitka. Nakon toga se prema sljedećem pravilu ažuriraju vrijednosti $x*$:

$$\begin{aligned} \text{if: } F(x_i) \leq F(x*) : x* = x_i \\ \text{else: } x* = x_i \text{with probabilty } p = e^{\frac{F(x*)-F(x_i)}{T_i}} \end{aligned} \tag{21}$$

, gdje T_i označava tzv. temperaturu koja se konstantno smanjuje kroz iteracije, od čega dolazi ime algoritma. Ovaj algoritam se ponavlja sve dok se ne pronađe minimalna temperatura. Dok je T_i vrlo visok algoritam provodi puno koraka istraživanja prostora stanja (slično kao nasumično pretraživanje), ali sa smanjenjem T algoritam se fokusira na eksploataciju – svi x_i su blizu jednog od najboljeg dobivenog rezultata.

U *Pythonu* se *Hyperopt* provodi slično kao i Bayesova optimizacija. Prvo moramo definirati funkciju koju želimo minimizirati. U ovom slučaju opet želimo minimizirati funkciju koja vraća vrijednost k-struka unakrsne provjere. Također opet definiramo prostor hiperparametara koji želimo istražiti.

```
def optimize_hp(params, x, y ):
    model = LGBMClassifier(**params)
    return -1.0 * np.mean(cross_val_score(model,
                                          x,
                                          y,
                                          cv=10,
                                          scoring="accuracy",
                                          n_jobs=-1))

space_hyperopt = {
    'learning_rate' : hp.uniform('learning_rate', 0.001 ,1),
    'num_leaves' : scope.int(hp.quniform('num_leaves', 20,3000,1)),
    'n_estimators' : scope.int(hp.quniform('n_estimators', 500,3000,1)),
    'max_depth' : scope.int(hp.quniform('max_depth', 3, 12,1)),
    'min_child_samples' : scope.int(
                        hp.quniform('min_child_samples', 2, 300,1)),
    'reg_lambda' : hp.uniform('reg_lambda', 0.01 ,100),
    'reg_alpha' : hp.uniform('reg_alpha', 0.01 ,100),
}

optimization_function_hp = partial(optimize_hp, x= X, y = y)
```

Listing 5: Definiranje funkcije gubitka

Uz *Hyperopt* se koristi i klasa *Trials* u koju se sprema svaka iteracija tijekom izvođenja optimizacije. Glavna funkcija koja se koristi je *fmin* funkcija iz *hyperopt* paketa. Njoj predajemo funkciju koju želimo minimizirati, hiperparametarski prostor, broj iteracija koje želimo obaviti(*max_evals* argument). Ako želimo koristiti *Tree-structured Prazen Estimator* onda ćemo pod argument *algo* predati *tpe.suggest*, a ako želimo koristiti *Simulated Anneal* onda u taj argument predajemo *anneal.suggest* kao što je prikazano u listingu 6.

```
trials_anneal = Trials()
result_anneal = fmin(fn=optimization_function_hp,
                     space = space_hyperopt,
                     algo = anneal.suggest,
                     max_evals = 1500,
                     trials = trials_anneal)

trials_tpe = Trials()
result_tpe = fmin(fn=optimization_function_hp,
                  space = space_hyperopt,
                  algo = tpe.suggest,
                  max_evals = 1500,
                  trials = trials_tpe)
```

Listing 6: Implementacija TPE i SA algoritma

5 Primjena optimizacije

Za potrebe ovog diplomskog rada odabrali smo 3 skupa podataka na kojima ćemo provesti hiperoptimizaciju parametara. Korištene tehnike optimizacije se navedene prije u radu, a to su pretraživanje po rešetci, nasumično pretraživanje, Bayesova optimizacija te *Hyperopt*. Kao model za klasifikaciju koristit ćemo već spomenuti LGBM klasifikator. Svi skupovi podataka koji su se koristili u praktičnom dijelu preuzeti su s Kagglea. Kaggle je web stranica na kojoj korisnici objavljuju skupove podataka, kreiraju modele i testiraju ih na velikom broju problema u strojnom učenju. Skupovi podataka koje smo odabrali su:

- Skup podataka za predviđanje mogućnosti srčanog udara
- Skup podataka o pitkosti vode
- Skup podataka o dijabetesu

Sav programski kod pisan je u programskom jeziku *Python* 3.10.2 verzija na 64-bitnom sustavu i u *Jupyter Notebooku*. Računalo na kojem je izveden kod pokreće AMD Ryzen 5 3600 procesor, 16 GB radne memorije i NVIDIA GeForce RTX 2060 grafička kartica. Prilikom izvedbe svake od ćelija u *Jupyter Notebooku* smo stavili da se prati vrijeme izvedbe te ćelije, a svaka od ćelija predstavlja po jednu radnju opisanu u nastavku.

5.1 Inicijalne postavke

Prvi korak u svakom programskom kodu u *Pythonu* je uvoz programskih knjižnica koje će se koristiti u programu. *Pandas* i *numpy* su knjižnice koje se koriste za lakši prikaz podataka, te omogućavaju lakše rukovanje s podacima. Iz *sklearn* knjižnice koristimo *cross_val_score* funkciju koja računa k-struka unakrsnu provjeru, *KFold* i *StratifiedKFold* su klase koje koristimo za podjelu podataka u svrhu k-struka unakrsne provjere, *train_test_split* funkciju koja nam služi za podjelu podataka u skup za treniranje i testiranje, *RandomizedSearchCV* i *GridSearchCV* klase pomoću kojih je izvedena implementacija nasumičnog pretraživanja i pretraživanja po rešetci, te funkcije *confusion_matrix* koja omogućava izradu matrice konfuzije. Naš klasifikator smo uvezli iz *lightgbm.sklearn* knjižnice, dok smo funkciju za Bayesovu optimizaciju uvezli iz *skotp* knjižnice. Iz *hyperopt* knjižnice smo uvezli funkcije koje nam trebaju za optimizaciju pomoću *Hyperopta* kao što je ranije spomenuta *fmin* funkcija te algoritmi *tpe* (*Tree-structured Prazen Estimator*) i *anneal* (*Simulated Anneal*).

```
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score,
                                    KFold,
                                    train_test_split,
                                    RandomizedSearchCV,
                                    GridSearchCV,
                                    StratifiedKFold
from lightgbm.sklearn import LGBMClassifier
from sklearn.metrics import confusion_matrix
from scipy.stats import randint,uniform
from hyperopt import fmin, tpe, hp, anneal, Trials
from hyperopt.pyll.base import scope
from functools import partial
import random
from skopt import space
from skopt import gp_minimize
```

Listing 7: Uvoz programskih knjižnica

Sljedeće što smo napravili je postavljanje konstanti kako bi se olakšala izmjena koda u budućnosti, tj. ako bi htjeli nešto promijeniti, promjena se treba napraviti samo na početku koda. Tako smo definirali *RANDOM_STATE* (slučajno stanje) pomoću nasumičnog broja iz intervala od 1 do 10000, broj podjela u k-struka unakrsnoj provjeri (*N_SPLITS*), koji je za sva tri skupa podataka postavljen na 10. Također su napravljene dvije podjele, na običnu k-struka podjelu (*kf*) te stratificiranu podjelu (*skf*). Varijabla *FOLDS* smo postavili na stratificiranu podjelu koja će se koristiti u svim primjerima. *N_JOBS* varijabla je postavljena na -1, a ona nam govori koliko će jezgri procesor koristiti (ako je negativan broj koristi sve jezgre). U varijable *EVALS_IN_HYPEROP*, *EVALS_IN_BAYESIAN* te *EVALS_IN_RANDOM* je smješten broj iteracija za svaki od navedenih načina optimizacije. Većina ovih konstanti je ista za sva tri skupa podataka, tako da neću opisivati za svaki skup podataka opet ove konstante. Navesti ću samo promjene u odnosu na početne vrijednosti navedene na listingu 8. Jedina varijabla koja će se mijenjati je *scoring* (metrika), a isprobat ćemo sve 3 prije spomenute metrike. Za svaki skup podataka naglasit ću koju metriku koristimo. Također svi hiperparametarski prostori spomenuti i postavljeni tijekom obrade podataka za prvi skup podataka koristit će se i u ostala dva skupa podataka.

```
RANDOM_STATE = random.randint(1,10000)
N_SPLITS = 10
kf = KFold(n_splits=N_SPLITS,
          random_state=RANDOM_STATE,
          shuffle=True)
skf = StratifiedKFold(n_splits=N_SPLITS,
                      random_state=RANDOM_STATE,
                      shuffle= True)
SCORING = "roc_auc"
EVALS_IN_HYPEROPT = 3000
EVALS_IN_BAYESIAN = 500
EVALS_IN_RANDOM = 5000
FOLDS = skf
N_JOBS = -1
```

Listing 8: Definiranje konstanti

Također koristimo dvije pomoćne funkcije za ispisivanje trenutnih hiperparametara modela *print_params*, te funkciju koja ispisuje točnosti modela u zavisnosti o prije spomenutim metrikama *print_scores_with_cross_val*. Funkcija *print_params* kao argument prima klasifikator te ispisuje trenutne hiperparametre tog modela, dok funkcija *print_scores_with_cross_val* prima dva argumenta, jedan je model, a drugi je *string* s imenom algoritma za optimizaciju.

```
def print_params(classifier):
  print("Trenutni parametri modela:")
  for param, value in classifier.get_params().items():
    print(f"{param}: {value}")

def print_scores_with_cross_val(classifier, optimization_algo = ""):
  print(f'Točnost modela {optimization_algo} sa unakrsnom provjerom:
  {round(cross_val_score(classifier, X, y, cv=FOLDS,
                         scoring="accuracy",
                         n_jobs=N_JOBS)
                         .mean() * 100, ndigits=2)}%')
  print(f'ROC AUC vrijednost modela {optimization_algo}
  sa unakrsnom provjerom:
    {round(cross_val_score(classifier, X, y,cv=FOLDS,
                           scoring="roc_auc",
                           n_jobs=N_JOBS)
                           .mean() * 100, ndigits=2)} %')
    print(f'F1 vrijednost modela {optimization_algo}
    sa unakrsnom provjerom:
    {round(cross_val_score(classifier,X, y, cv=FOLDS,
                           scoring="f1",
                           n_jobs=N_JOBS)
                           .mean() * 100, ndigits=2)} %')
```

Listing 9: Pomoćne funkcije

5.2 Srčani udar

Prvi skup podataka nad kojim ćemo napraviti optimizaciju hiperparametara je skup podataka koji se sastoji od informacija o pacijentima kao što su njihova starost, spol, kolesterol i slično. Cilj, pomoću ovog skupa podataka, je na osnovu ovih informacija naučiti model da predvidi za nove podatke o pacijentima hoće li imati srčani udar ili ne. Prvo što moramo napraviti je pročitati podatke iz datoteke, te ih spremiti u varijablu.

```
dataset = pd.read_csv('heart.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

Listing 10: Čitanje podataka iz datoteke

Ovaj skup podataka se sastoji od 13 ulaznih značajki i jedne izlaznom značajkom. To vidimo pomoću sljedeće linije u kodu:

```
dataset.describe()
```

Listing 11: Prikaz podataka

Iz ove linije dobijemo sljedeći prikaz:

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	output
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.313531	0.544554
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.612277	0.498835
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.000000	0.000000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.000000	1.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.000000	1.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.000000	1.000000

Slika 20: Prikaz podataka

Na slici 20 su na izlistani osnovni podaci o skupu podataka kao što su: broj opservacija pojedine značajke (*count*), srednja vrijednost (*mean*), standardna devijacija (*std*) i ostali važni statistički podaci. Također vidimo da je zadnja značajka, *output*, naša izlazna značajka, te da prima vrijednosti 0 i 1, što znači da imamo problem binarne klasifikacije. Za ovaj skup podataka koristit ćemo ROC_AUC metriku zato što je ovo osjetljiv skup podataka, za koji ne želimo da imamo puno *false negative* predviđenih opservacija.

5.2.1 Priprema podataka

Sljedeći korak je priprema podataka. Prvo što radimo je razdvajanje skupa podataka na skup za treniranje i skup za testiranje. To radimo sa sljedećom linijom koda:
Kao što se vidi prema parametru *test_size* podijelili smo skup podataka tako da 80% podataka ide u skup za treniranje a 20% ide u skup za testiranje.

```
X_train, X_test, y_train, y_test = train_test_split(X,
y,
test_size=0.20,
random_state=RANDOM_STATE,
shuffle=True)
```

Listing 12: Razdvajanje na skup za treniranje i testiranje

Zadnja stvar što smo napravili je inicijalizacija običnog LGBM klasifikatora.

```
lgbm_classic= LGBMClassifier(random_state=RANDOM_STATE)
```

Listing 13: Inicijalziacija LGBM klasifikatora

5.2.2 Treniranje modela i predviđanje

Nakon što smo podijeli podatke u skup za treniranje i testiranje krećemo s učenjem i predikcijom LGBM klasifikatora. To se radi pomoću sljedećih naredbi:

```
lgbm_classic.fit(X_train, y_train)
y_pred = lgbm_classic.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
print(f"Točnost LGBM klasifikatora bez unakrsne provjere:
{lgbm_classic.score(X_test, y_test) * 100} \%")
```

Listing 14: Učenje i predikcija klasifikatora

```
[[24  4]
 [ 4 29]]
Točnost LGBM klasifikatora bez unakrsne provjere: 86.88524590163934 %
```

Slika 21: Točnost klasifikatora

Također ovdje smo isprintali matricu konfuzije i točnost klasifikatora bez unakrsne provjere, zato što ju nismo još napravili, a na slici 21 vidimo da je naš klasifikator sa trenutnim parametrima napravio 8 grešaka u predikciji, od čega su 4 *false negative* i 4 *false positive* greške, dok je 24 uzoraka točno predvidio da su *true positive*, a 29 *true negative*. Također vidimo da je točnost modela 86,88%. Ovo su rezultati klasifikatora bez ikakve unakrsne provjere i optimizacije hiperparametara.

5.2.3 *Grid Search*

Sada ćemo odraditi optimizaciju pomoću pretraživanja po rešetci. Ona se radi na sljedeći način:

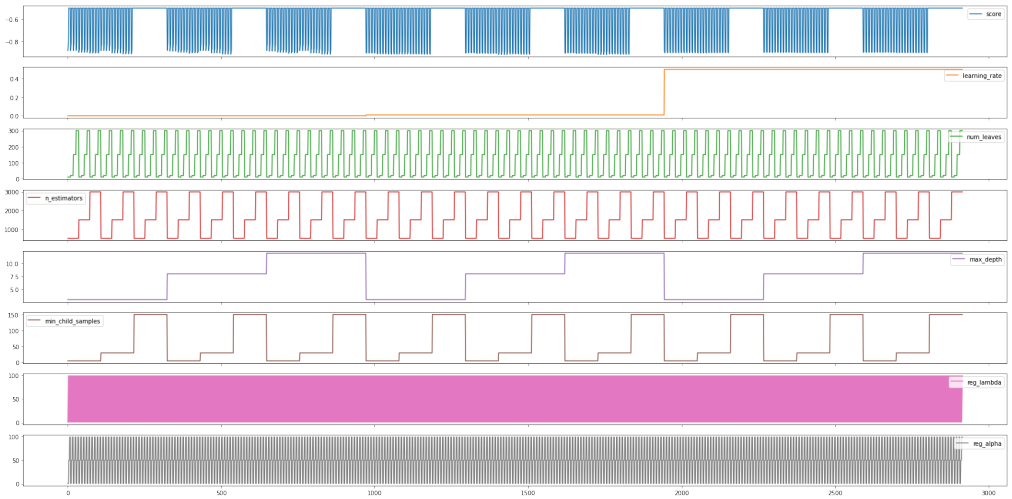
```
params_grid_search = {'learning_rate': [0.001, 0.01, 0.5],
          'num_leaves': [10, 20, 150, 300],
          'n_estimators': [500, 1500, 3000],
          'max_depth': [3, 8, 12],
          'min_child_samples': [5, 30, 150],
          'reg_lambda': [1, 50, 99],
          'reg_alpha' : [1, 50, 99]}
grid_search = GridSearchCV(estimator=LGBMClassifier(
                           random_state=RANDOM_STATE),
                           param_grid=params_grid_search,
                           scoring=SCORING,
                           cv=FOLDS,
                           n_jobs=N_JOBS,
                           return_train_score=True,
                           verbose=10)
grid_search.fit(X, y)
print(f"Najbolja točnost sa Grid Search-om:
{grid_search.best_score_ * 100} %")
print(f"Najbolji parametri sa Grid Search-om: {grid_search.best_params_}")
```

Listing 15: Implemetacija *Grid Search* algoritma

```
Fitting 10 folds for each of 2916 candidates, totalling 2916
0 fits
Najbolja točnost sa Grid Search-om: 91.84510342598578 %
Najbolji parametri sa Grid Search-om: {'learning_rate': 0.0
1, 'max_depth': 8, 'min_child_samples': 30, 'n_estimators':
500, 'num_leaves': 10, 'reg_alpha': 1, 'reg_lambda': 1}
CPU times: total: 21.5 s
Wall time: 5min 37s
```

Slika 22: Rezultati *Grid Search* algortima

Kao što vidimo prema slici 22 parametarski prostor koji smo postavili na listingu 15 imao je 2916 mogućnosti da se sve vrijednosti parametara međusobno isprobaju, ali smo postavili da se odradi 10 struka unakrsna provjera pa je ukupni broj iteracija 29160. Vrijeme trajanja izvedbe ove ćelije je 5 minuta i 37 sekundi, što nije previše, ali nismo ni imali preveliki broj kombinacija hiperparametara, ali ni preveliki broj opservacija (303 opservacije prema slici 20). Također vidimo najbolju točnost modela i najbolje vrijednosti hiperparametara s kojima smo dobili tu točnost (u ovom slučaju ROC_AUC točnost), a to je 91,04% što je poboljšanje u odnosu na točnost bez ikakve optimizacije. Pretraživanje po rešetci je pronašlo da su sljedeći hiperparametri najbolji: *learning_rate* = 0.01, *max_depth* = 8, *min_child_samples* = 30, *n_estimators* = 500, *num_leaves* = 10, *reg_alpha* = 1, *reg_lambda* = 1, a njihovu ovisnosti prema najboljoj točnosti možemo vidjeti na sljedećoj slici:



Slika 23: Vizualizacija rezultata *Grid Search* algortima

Ova slika se dobije pomoću sljedećg koda:

```
grid_search_scores = grid_search.cv_results_
grid_search_results_df=
        pd.DataFrame(np.transpose([-grid_search_scores['mean_test_score'],
        grid_search_scores['param_learning_rate'].data,
        grid_search_scores['param_num_leaves'].data,
        grid_search_scores['param_n_estimators'].data,
        grid_search_scores['param_max_depth'].data,
        grid_search_scores['param_min_child_samples'].data,
        grid_search_scores['param_reg_lambda'].data,
        grid_search_scores['param_reg_alpha'].data]),
        columns=['score', 'learning_rate', 'num_leaves',
        'n_estimators', 'max_depth', 'min_child_samples',
        'reg_lambda', 'reg_alpha'])
grid_search_results_df.plot(subplots=True, figsize=(30, 15))
```

Listing 16: Implementacija prikaza rezultata *Grid Searcha*

Također prema slici 23 vidimo da male razlike u točnosti kada se mijenjaju hiperparametri vezani uz stabla, a to su *max_depth* i *n_estimators*, a kada je parametar *min_child_samples* postavljen na vrijednost 150 dobivamo veoma malu ROC_AUC vrijednost. Iako su ove razlike u vrijednosti metrike veoma male, bilo kakvo poboljšanje metrike je dobro, zato što će se to odraziti na veću točnost kada model dobije neke nove podatke koje nije do sada vidio.

5.2.4 *Random Search*

Sljedeći postupak optimizacije je nasumično pretraživanje koje smo proveli pomoću sljedećih linija koda:

```
%%time
params_random_search = {'learning_rate' : uniform(0.01,0.99),
          'num_leaves': randint(20,3000),
          'n_estimators': randint(500,3000),
          'max_depth': randint(3,12),
          'min_child_samples': randint(2,300),
          'reg_lambda': uniform(0.01,99.99),
          'reg_alpha': uniform(0.01,99.99),
          }
random_search = RandomizedSearchCV(estimator =
                            LGBMClassifier(random_state=RANDOM_STATE),
                            param_distributions = params_random_search,
                            n_iter = EVALS_IN_RANDOM,
                            scoring = SCORING,
                            cv = FOLDS,
                            random_state = RANDOM_STATE,
                            n_jobs = N_JOBS,
                            verbose=10
                            )
random_search.fit(X, y)
print("Najbolja točnost sa Random Search-om: {:.2f}
%".format(random_search.best_score_*100))
print("Najbolji parametri prema Random Search-om:",
random_search.best_params_)
```

Listing 17: Implemetacija *Random Search* algoritma

Prema slici 24 vidimo da smo imali 5000 kandidata (ovaj broj smo postavili na početku programa), a zbog 10 struka unakrsne provjere imamo 50000 ponavljanja. Također vidimo da je vrijeme izvršavanja ove ćelije slično kao i kod *Grid searcha*, ali dobivamo još bolji iznos tražene metrike, koji iznosi 92,30%. Također vidimo i hiperparametre koje je pronašao *Random search*: $learning\_rate = 0.1092752295906971$, $max\_depth = 7$, $min\_child\_samples = 18$, $n\_estimators = 2324$, $num\_leaves = 1568$, $reg\_alpha = 3.4982991494330395$, $reg\_lambda = 62.037627830084354$. Možemo uočiti da se dosta razlikuju od

```
Fitting 10 folds for each of 5000 candidates, totalling 5000
0 fits
Najbolja točnost sa Random Search-om: 92.30 %
Najbolji parametri prema Random Search-om: {'learning_rate':
0.1092752295906971, 'max_depth': 7, 'min_child_samples': 18,
'n_estimators': 2324, 'num_leaves': 1568, 'reg_alpha': 3.498
2991494330395, 'reg_lambda': 62.037627830084354}
CPU times: total: 34.4 s
Wall time: 5min 20s
```

Slika 24: Rezultati *Random Search* algortima

hiperparametara koje je pronašao *Grid search*, pogotovo hiperparametri o kojima ovisi izgled stabla. Imamo puno veći broj stabala (*Grid search* 500 a *Random Search* 2324), te broj čvorova u drveću (*Grid search* 10 a *Random Search* 1568). Iako sam naveo u teorijskom dijelu da broj čvorova u drvetu ne bi trebao biti veći od 2^{max_depth}, u ovom slučaju je provedena unakrsna provjera, a i hiperparametri za regularizaciju su uključeni u optimizaciju.

5.2.5 *Hyperopt* (*Simulated Anneal*)

Sljedeća optimizacija je pomoću *Hyperopt* knjižnice i sa *Simulated Anneal* algoritmom. To smo proveli pomoću sljedećim kodom:

```
def optimize_hp(params, x, y ):
    model = LGBMClassifier(**params)
    return -1.0 * np.mean(cross_val_score(model, x, y,
    cv=FOLDS, scoring=SCORING, n_jobs=N_JOBS))

space_hyperopt = {
    'learning_rate' :
    hp.uniform('learning_rate', 0.001 ,1),
    'num_leaves' :
    scope.int(hp.quniform('num_leaves', 20,3000,1)),
    'n_estimators' :
    scope.int(hp.quniform('n_estimators', 500,3000,1)),
    'max_depth' :
    scope.int(hp.quniform('max_depth', 3, 12,1)),
    'min_child_samples'
    : scope.int(hp.quniform('min_child_samples', 2, 300,1)),
    'reg_lambda'
    : hp.uniform('reg_lambda', 0.01 ,100),
    'reg_alpha'
    : hp.uniform('reg_alpha', 0.01 ,100),
}

optimization_function_hp = partial(optimize_hp, x= X, y = y)
```

Listing 18: Funkcija gubitka i hiperparametarski prostor

Na listingu 18 smo definirali funkciju koju želimo optimizirati, te hiperparametarski prostor. Ova funkcija gubitka će nam služiti i za *Simulated Anneal* i za *Tree-structured Parzen Estimator* algoritme. Pomoću sljedećeg koda odrađujemo *Simulated Anneal* algoritam.

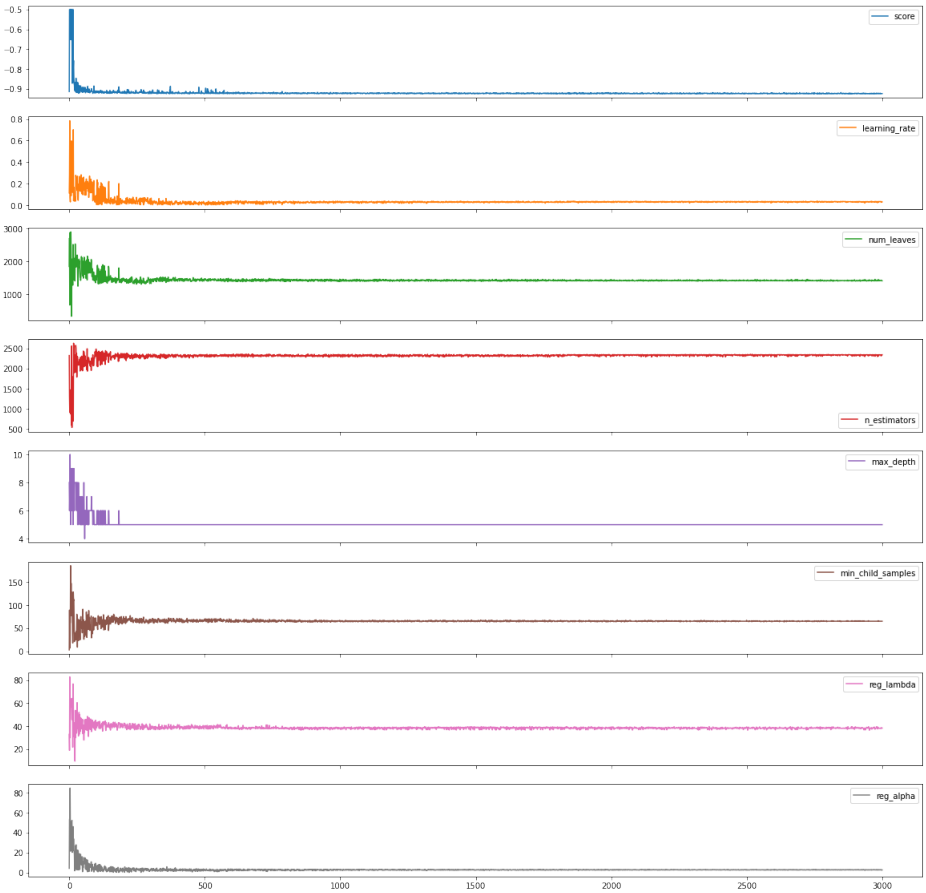
```
%%time
trials_anneal = Trials()
result_anneal = fmin(fn=optimization_function_hp,
space = space_hyperopt, algo = anneal.suggest,
max_evals = EVALS_IN_HYPEROPT,
trials = trials_anneal)
print(result_anneal)
```

Listing 19: Implemetacija *Simulated Anneal* algoritma

```
100%|██████████| 3000/3000 [16:30<00:00,  3.03trial/s, best
loss: -0.925450468648998]
{'learning_rate': 0.03217203120135238, 'max_depth': 5.0, 'mi
n_child_samples': 65.0, 'n_estimators': 2342.0, 'num_leaves
': 1416.0, 'reg_alpha': 2.621615128887446, 'reg_lambda': 38.
18206514665476}
CPU times: total: 1min 38s
Wall time: 16min 31s
```

Slika 25: Rezultati *Simulated Anneal* algoritma

Simulated Anneal algoritam nam je dao sljedeće rezultate kao što možemo vidjeti prema slici 25: *learning_rate* = 0.03217203120135238, *max_depth* = 5.0, *min_child_samples* = 65.0, *n_estimators* = 2342.0, *num_leaves* = 1416.0, *reg_alpha* = 2.621615128887446, *reg_lambda* = 38.18206514665476. Ako usporedimo ove vrijednosti hiperparametra s vrijednostima koje smo pronašli sa *Random Search* algoritmom (slika 24) vidimo da su vrijednosti hiperparametara veoma slični. Vrijeme potrebno za izvršavanje ovog algoritma je bilo 16 minuta i 30 sekundi što je tri puta više nego *Random Searcha* i *Grid Searcha*, dok je vrijednost funkcije gubitka s tim hiperparametrima iznosila 92.54%, što je opet malo poboljšanje u odnosu na *Random Search*. Na slici 26 vidimo vrijednosti hiperparametara prema iteracijama, i kao što smo naveli u teorijskom dijelu ovaj algoritam na početku nasumično generira neku vrijednost hiperparametra, pa istražuje susjedne vrijednosti sve dok ne nađe optimalnu vrijednost. Također možemo vidjeti da se već oko 250-te iteracije pronalazi približno optimalna vrijednost.



Slika 26: Vizualizacija rezultata *Simulated Anneal* algoritma

5.2.6 *Hyperopt* (*Tree-structured Parzen Estimator*)

Tree-structured Parzen Estimator algoritam se implementira, pošto smo već definirali funkciju gubitka i hiperparametarski prostor, na sljedeći način:

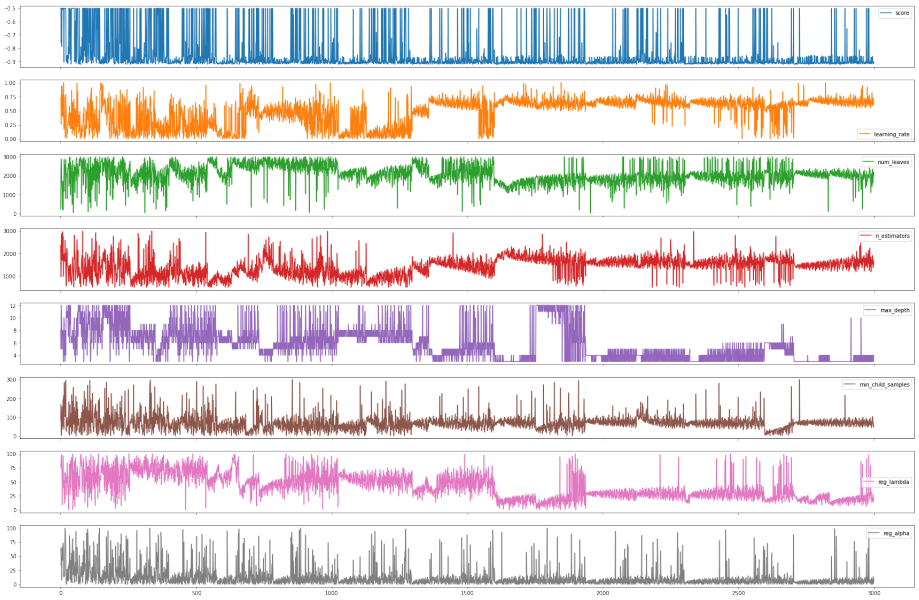
```
%%time
trials_tpe = Trials()
result_tpe = fmin(fn=optimization_function_hp,
space = space_hyperopt,
algo = tpe.suggest,
max_evals = EVALS_IN_HYPEROPT,
trials = trials_tpe)
print(result_tpe)
```

Listing 20: Implemetacija *Tree-structured Parzen Estimator* algoritma

```
100%|██████████| 3000/3000 [17:43<00:00,  2.82trial/s, best
loss: -0.9245192307692307]
{'learning_rate': 0.6242690529591417, 'max_depth': 3.0, 'min
_child_samples': 70.0, 'n_estimators': 1439.0, 'num_leaves':
2117.0, 'reg_alpha': 2.5127573893173416, 'reg_lambda': 24.07
8511185424517}
CPU times: total: 12min
Wall time: 17min 43s
```

Slika 27: Rezultati *Tree-structured Parzen Estimator* algoritma

Pomoću *Tree-structured Parzen Estimator* algoritma smo dobili sljedeće vrijednosti hiperparametara: *learning_rate* = 0.6242690529591417, *max_depth* = 3.0, *min_child_samples* = 70.0, *n_estimators* = 1439.0, *num_leaves* = 2117.0, *reg_alpha* = 2.5127573893173416, *reg_lambda* = 24.078511185424517 s kojima model ima 92,45% ROC_AUC vrijednost. Najveća razlika između vrijednosti hiperparametara kod *Tree-structured Parzen Estimator* algoritma i *Simulated Anneal* algoritma je kod hiperparametra stope učenje (*learning_rate*), ali je zato broj stabala (*n_estimators*) puno manji.



Slika 28: Vizualizacija rezultata *Tree-structured Parzen Estimator* algoritma

5.2.7 Bayesova optimzacija

Zadnji algoritam koji ćemo implementirati je pomoću Bayesove optimziacije. Prvo moramo napraviti funkciju gubitka koju želimo minimizirati, te definirati hiperparametarski prostor. To se radi na sljedeći način:

```
def optimize_bayesian(params, param_names ,x, y):
    params = dict(zip(param_names, params))
    model = LGBMClassifier(**params)
    return -1.0 * np.mean(cross_val_score(model, x, y, cv=FOLDS,
    scoring=SCORING, n_jobs=N_JOBS))
%%time
param_space = [
space.Real(0.001,1, prior = "uniform", name = "learning_rate"),
space.Integer(20,3000, name = "num_leaves"),
space.Integer(500,3000, name = "n_estimators"),
space.Integer(3,12, name = "max_depth"),
space.Integer(2,300, name = "min_child_samples"),
space.Real(0.01,100, prior = "uniform", name = "reg_lambda"),
space.Real(0.01,100, prior = "uniform", name = "reg_alpha")
]
param_names=[
             "learning_rate",
             "num_leaves",
             "n_estimators",
             "max_depth",
             "min_child_samples",
             "reg_lambda",
             "reg_alpha",
]
```

Listing 21: Funkcija gubitka i hiperparametarski prostor

A implementacija se provodi na sljedeći način:

```
optimization_function_bayesian = partial(optimize_bayesian,
param_names = param_names,
x = X,
y= y)
result_bayesian = gp_minimize(optimization_function_bayesian,
dimensions = param_space,
n_calls = EVALS_IN_BAYESIAN,
n_initial_points=10,
verbose=10,
n_jobs=N_JOBS)

bayesian_parameters = dict(zip(param_names, result_bayesian.x))
print(bayesian_parameters)
```

Listing 22: Implementacija Bayesove optimizacije

Kod Bayesove optimizacije možemo pratiti koju vrijednost je poprimila funkcija gubitka kod svake iteracije. Prikazat ću samo prve tri i zadnje 2 iteracije.

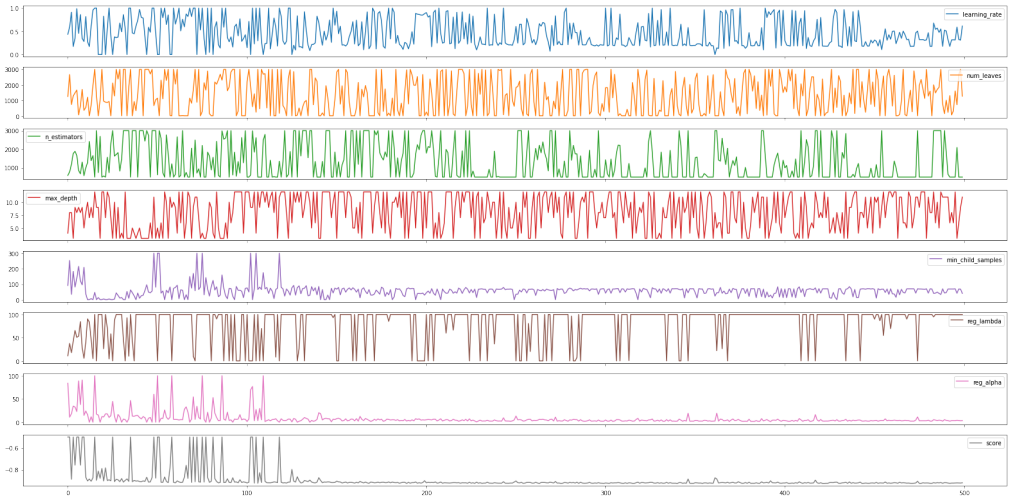
```
Iteration No: 1 started. Evaluating function at random poin
t.
Iteration No: 1 ended. Evaluation done at random point.
Time taken: 0.0410
Function value obtained: -0.5000
Current minimum: -0.5000
Iteration No: 2 started. Evaluating function at random poin
t.
Iteration No: 2 ended. Evaluation done at random point.
Time taken: 0.0900
Function value obtained: -0.5000
Current minimum: -0.5000
Iteration No: 3 started. Evaluating function at random poin
t.
Iteration No: 3 ended. Evaluation done at random point.
Time taken: 0.0690
Function value obtained: -0.8865
Current minimum: -0.8865
```

Slika 29: Prve 3 iteracije Bayesove optimizacije

```
Iteration No: 499 started. Searching for the next optimal po
int.
Iteration No: 499 ended. Search finished for the next optima
l point.
Time taken: 20.5711
Function value obtained: -0.9215
Current minimum: -0.9252
Iteration No: 500 started. Searching for the next optimal po
int.
Iteration No: 500 ended. Search finished for the next optima
l point.
Time taken: 20.7980
Function value obtained: -0.9194
Current minimum: -0.9252
{'learning_rate': 0.6638741310539517, 'num_leaves': 20, 'n_e
stimators': 2210, 'max_depth': 12, 'min_child_samples': 62,
'reg_lambda': 0.01, 'reg_alpha': 5.889141882002036}
CPU times: total: 1h 5min 41s
Wall time: 1h 3min 9s
```

Slika 30: Zadnje 2 iteracije Bayesove optimizacije

Prema slikama 29 i 30 vidimo da nam se ispisuje koja je trenutna iteracija, koliko vremena je oduzela ta iteracija, koliku vrijednost je imala funkcija gubitka u toj iteraciji i koliki je trenutni minimum funkcije gubitka. Također na kraju zapisa na slici 30 vidimo vrijednosti hiperparametara koje smo dobili, a to su: *learning_rate* = 0.6638741310539517, *max_depth* = 12, *min_child_samples* = 62, *n_estimators* = 2210, *num_leaves* = 20, *reg_alpha* = 0.01, *reg_lambda* = 5.889141882002036. Također vidimo da je minimum funkcije gubitka -0.9252 što nam daje ROC_AUC vrijednost od 92.52%. Još jedna vrlo bitna stvar za primijetiti je vrijeme izvođenja Bayes-ove optimizacije, a ono iznosi 1:03:09 sati, što dosta duže u odnosu na ostale algoritme.



Slika 31: Vizualizacija rezultata Bayesove optimizacije

5.2.8 Usporedba rezultata

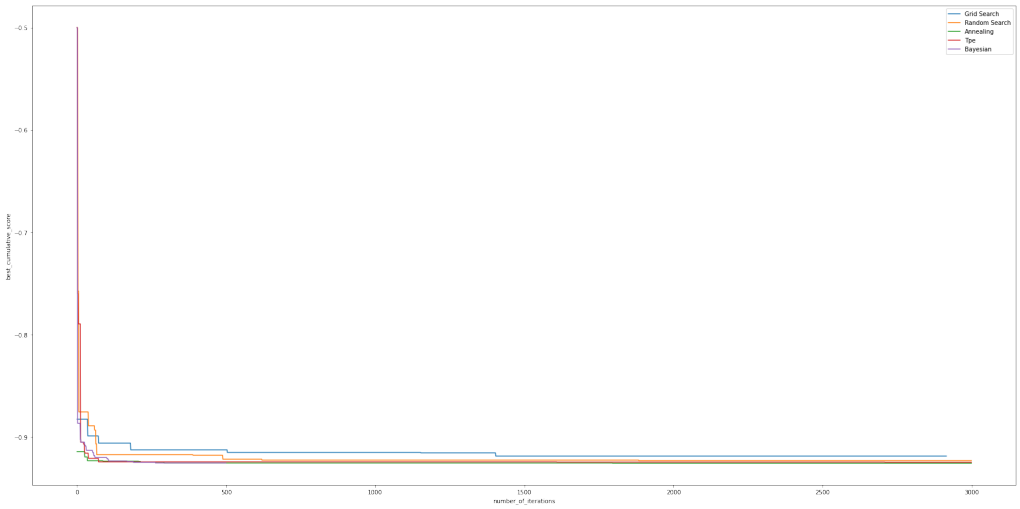
Na kraju nam je ostalo samo da usporedimo dobivene rezultate svih algoritama optimizacije, a to smo napravili pomoću već prije spomenute funkcije na listingu 9 *print_scores_with_cross_val*. Također smo napravili vizualnu usporedbu svi optimizacijskih algoritama pomoću sljedećeg koda.

```
scores_df=pd.DataFrame(index=range(3000))
scores_df['Grid Search']=grid_search_results_df['score'].cummin()
scores_df['Random Search']=random_search_results_df['score'].cummin()
scores_df['Annealing']=anneal_results_df['score'].cummin()
scores_df['Tpe'] = tpe_results_df['score'].cummin()
scores_df['Bayesian'] = bayesian_results_df['score'].cummin()

ax = scores_df.plot(figsize=(30, 15))

ax.set_xlabel("number_of_iterations")
ax.set_ylabel("best_cumulative_score")
```

Listing 23: Implementacija vizualne usporedbe rezultata



Slika 32: Usporedba svi rezultata

Kao što možemo vidjeti sa slika 33 i 34 svaki od modela koje smo napravili u nekoj od metrika je dobar, na primjer model koji smo dobili pomoću hiperparametara koje je pronašla Bayes-ova optimizacija ima vrlo dobru točnost (najveća točnost od svih modela) , ROC_AUC vrijednost i F1 vrijednost (najveća F1 vrijednost od svih), ali je i standardna devijacija najveća. Također Bayesova optimizacija je zahtijevala i najduže vremena (4 puta duže od ostalih), ali je dala i najbolje rezultate. Najbolje hiperparametre za ROC_AUC metriku, koju smo i tražili, dobili smo pomoću *Hyperopt* knjižnice i pomoću *Simulated Anneal* algoritma. Model napravljen sa hiperparametrima dobivenih pomoću pretraživanja po rešetci je dobio najgoru vrijednost metrike, dok je nasumično pretraživanje dobilo drugu najgoru vrijednost. Algoritam *Tree-structured Parzen Estimator* je dobio veoma sličnu vrijednost kao i nasumično pretraživanje, ali je trajao 3 puta duže nego nasumično pretraživanje. Na kraju možemo reći da za ovaj skup podataka, kada uzmemo u obzir brzinu algoritma i najbolju vrijednost metrike koju smo izabrali, najbolji je *Simulated Anneal* algoritam iz *Hyperopt* knjižnice.

```
Točnost modela bez optimizacije sa unakrsnom provjerom: 80.1
8%
Standardna devijacija modela bez optimizacije sa unakrnsom p
rovjerom: 6.11 %
ROC AUC vrijednost modela bez optimizacije : 88.26 %
F1 vrijednost modela bez optimizacije : 82.02 %


Točnost modela s Grid Search-om sa unakrsnom provjerom: 84.
8%
Standardna devijacija modela s Grid Search-om sa unakrnsom p
rovjerom: 5.45 %
ROC AUC vrijednost modela s Grid Search-om : 91.85 %
F1 vrijednost modela s Grid Search-om : 86.68 %


Točnost modela s Random Searchom-om sa unakrsnom provjerom:
84.14%
Standardna devijacija modela s Random Searchom-om sa unakrns
om provjerom: 5.11 %
ROC AUC vrijednost modela s Random Searchom-om : 92.3 %
F1 vrijednost modela s Random Searchom-om : 85.87 %
```

Slika 33: Metrike svih modela

```
Točnost modela s Hyperopt-Anneal sa unakrsnom provjerom: 84.
13%
Standardna devijacija modela s Hyperopt-Anneal sa unakrnsom
provjerom: 6.34 %
ROC AUC vrijednost modela s Hyperopt-Anneal : 92.55 %
F1 vrijednost modela s Hyperopt-Anneal : 85.47 %


Točnost modela s Hyperopt-Tpe sa unakrsnom provjerom: 83.8%
Standardna devijacija modela s Hyperopt-Tpe sa unakrnsom pro
vjerom: 5.53 %
ROC AUC vrijednost modela s Hyperopt-Tpe : 92.45 %
F1 vrijednost modela s Hyperopt-Tpe : 85.37 %


Točnost modela s Bayesian sa unakrsnom provjerom: 87.08%
Standardna devijacija modela s Bayesian sa unakrnsom provjer
om: 6.46 %
ROC AUC vrijednost modela s Bayesian : 92.52 %
F1 vrijednost modela s Bayesian : 88.25 %
```

Slika 34: Metrike svih modela

5.3 Pitkost vode

Drugi skup podataka koji smo izabrali je skup podataka o pitkosti vode. Za ovaj skup ćemo koristiti točnost kao metriku prema kojoj će se raditi optimizacija hiperparametara.

5.3.1 Priprema podataka

Kao i kod prvog skupa podataka prvo što moramo napraviti je pogledati kakve podatke imamo. To radimo na isti način kao i kod prvog skupa podataka (listing 11).

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.000000	3276.000000	3114.000000	3276.000000	3276.000000
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.205111	14.284970	66.396293	3.966786	0.390110
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.824064	3.308162	16.175008	0.780382	0.487849
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.483754	2.200000	0.738000	1.450000	0.000000
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.734414	12.065801	55.844536	3.439711	0.000000
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.884968	14.218338	66.622485	3.955028	0.000000
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.792304	16.557652	77.337473	4.500320	1.000000
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.342620	28.300000	124.000000	6.739000	1.000000

Slika 35: Opis podataka

Kao što vidimo na slici 36 imamo 9 ulaznih značajki koje uključuju osnovne informacije o kvaliteti vode kao što su pH vrijednost, količina klora, sulfata itd. Također vidimo da imamo izlaznu značajku pitkost, koja ima dva stanja 0 i 1. Nadalje, vidimo da imamo 3198 opservacija, ali u nekih stupcima kao što je ph vrijednost vidimo da imamo praznih (*null*) vrijednosti, pa to moramo popuniti. Ima više načina za popunjavanje takvih vrijednosti, ali odlučili smo da ćemo nedostajuće vrijednosti popuniti s prosječnom vrijednošću u tome stupcu (*mean* vrijednost). To radimo na sljedeći način:

```
1 dataset.ph = dataset.ph.fillna(dataset.ph.mean())
2 dataset.Sulfate = dataset.Sulfate.fillna(dataset.Sulfate.mean())
3 dataset.Trihalomethanes = dataset.Trihalomethanes.
4                 fillna(dataset.Trihalomethanes.mean())
```

Listing 24: Popunjavanje nedostajućih vrijednosti

Ako sada ponovo izvedemo naredbu s listinga 11 vidimo da sada svi stupci imaju isti broj opservacija. Također nakon ovog koraka, dijelimo skup podataka na skup za treniranje i skup za testiranje isto kao i kod prvog skupa podataka (listing 12).

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
count	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000	3198.000000
mean	7.085299	195.966292	21932.729508	7.157705	333.914847	426.248249	14.290948	66.388968	3.960728	0.375235
std	1.451207	33.288296	8694.697529	1.565831	35.465996	81.730425	3.310196	15.780525	0.781109	0.484259
min	0.000000	47.432000	320.942611	0.530351	187.170714	181.483754	2.200000	0.738000	1.450000	0.000000
25%	6.284536	175.931744	15621.042015	6.163502	317.285500	365.695713	12.084591	56.743869	3.438465	0.000000
50%	7.080795	196.928061	20882.160702	7.145835	333.775777	421.136657	14.252898	66.396293	3.950530	0.000000
75%	7.871417	216.757733	27196.306175	8.137345	350.305293	483.784905	16.585797	76.762017	4.485903	1.000000
max	14.000000	323.124000	61227.196008	12.912187	475.737460	753.342620	28.300000	124.000000	6.739000	1.000000

Slika 36: Opis podataka

5.3.2 Treniranje modela

Nakon što smo odradili pripremu podataka puštamo model da uči nad podacima (listing 14) te dobivamo sljedeće rezultate:

```
[[363  54]
 [ 76 147]]
Točnost LGBM klasifikatora bez unakrsne provjere: 79.6875 %
```

Slika 37: Točnost modela

Prema 37 vidimo da smo dobili veoma dobru točnost od 79,6875%. Uz pomoć optimizacije hiperparametara probat ćemo taj postotak povećati.

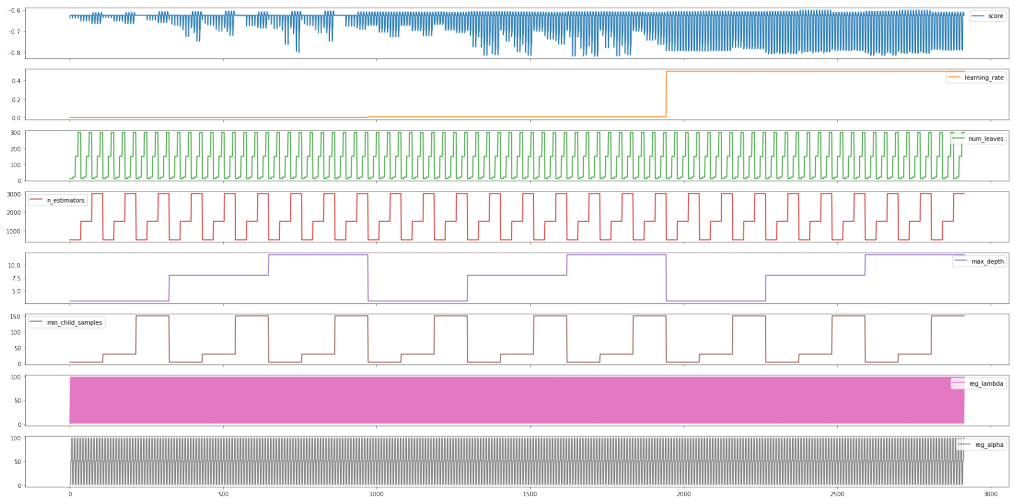
5.3.3 *Grid Search*

Uz pomoć već prije definiranog hiperparametarskog *grida* (listing 15 dobivamo sljedeće rezultate:

```
Fitting 10 folds for each of 2916 candidates, totalling 29160 fits
Najbolja točnost sa Grid Search-om: 81.8639302507837 %
Najbolji parametri sa Grid Search-om: {'learning_rate': 0.01, 'max_depth': 12, 'min_
child_samples': 5, 'n_estimators': 3000, 'num_leaves': 300, 'reg_alpha': 1, 'reg_lam
bda': 50}
CPU times: total: 2min 37s
Wall time: 47min 7s
```

Slika 38: Rezultati *Grid Searcha*

Kao što možemo vidjeti sa slike 38, algoritmu je bilo potrebno 47 minuta da završi što je dosta više u odnosu na prvi skup podataka, ali ovaj put smo imali i puno više opservacija. Također vidimo da nam se poboljšala točnost na 81,86 %, a to smo dobili sa sljedećim hiperparametrima: $learning\_rate = 0.01$, $max\_depth = 12$, $min\_child\_samples = 5$, $n\_estimators = 3000$, $num\_leaves = 300$, $reg\_alpha = 1$, $reg\_lambda = 50$. Također na sljedećoj slici je prikazana vizualizacija traženja hiperparametara po svakoj iteraciji.



Slika 39: Vizualizacija rezultata *Grid Searcha*

5.3.4 Nasumično pretraživanje

Sljedeća optimizacijska tehnika je nasumično pretraživanje. Pomoću nje smo dobili sljedeće rezultate:

```
Fitting 10 folds for each of 5000 candidates, totalling 50000 fits
Najbolja točnost sa Random Search-om: 80.83 %
Najbolji parametri prema Random Search-om: {'learning_rate': 0.6683814440851101, 'ma
x_depth': 10, 'min_child_samples': 6, 'n_estimators': 2659, 'num_leaves': 292, 'reg_
alpha': 1.3542699579017912, 'reg_lambda': 20.470501001176398}
CPU times: total: 1min 25s
Wall time: 23min 51s
```

Slika 40: Rezultati nasumičnog pretraživanja

Na slici 40 vidimo da nam se poboljšala točnost (80,83%) u odnosu na prije optimizacije, ali ako usporedimo rezultate s pretraživanjem po rešetci vidimo da smo dobili lošiju točnost nego spomenuti algoritam. Također brzina izvede algoritma za ovaj skup podataka iznosi skoro 24 minute, što je

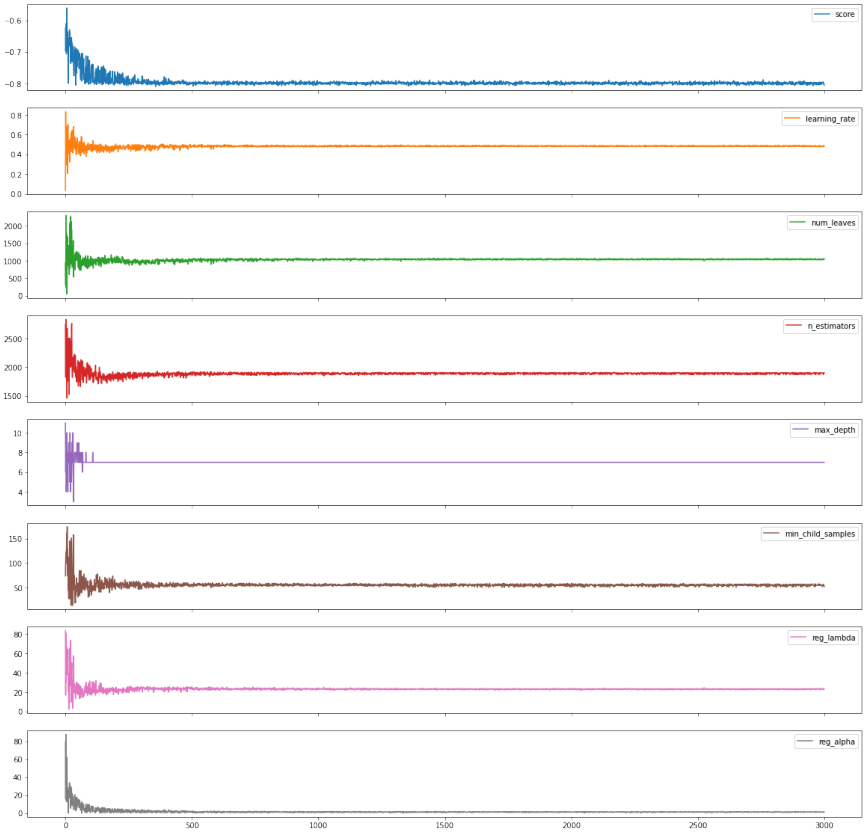
duplo brže nego pretraživanje po rešetci. Hiperparametre koje smo dobili koristeći ovaj algoritam su: *learning_rate* = 0.6683814440851101, *max_depth* = 10, *min_child_samples* = 6, *n_estimators* = 2659, *num_leaves* = 292, *reg_alpha* = 1.3542699579017912, *reg_lambda* = 20.470501001176398.

5.3.5 Hyperopt (*Simulated Anneal*)

Rezultati dobiveni koristeći ovaj algoritam su sljedeći:

```
100%|████████████████████████████████████| 3000/3000 [34:41<00:00,  1.44tria
l/s, best loss: -0.809264302507837]
{'learning_rate': 0.48036624733907357, 'max_depth': 7.0, 'min_child_samples': 55.0,
'n_estimators': 1897.0, 'num_leaves': 1031.0, 'reg_alpha': 1.0083963134106373, 'reg_
lambda': 23.445116446536673}
CPU times: total: 13.5 s
Wall time: 34min 41s
```

Slika 41: Rezultati *Simulated Anneala*



Slika 42: Vizualizacija rezultata *Simulated Anneala*

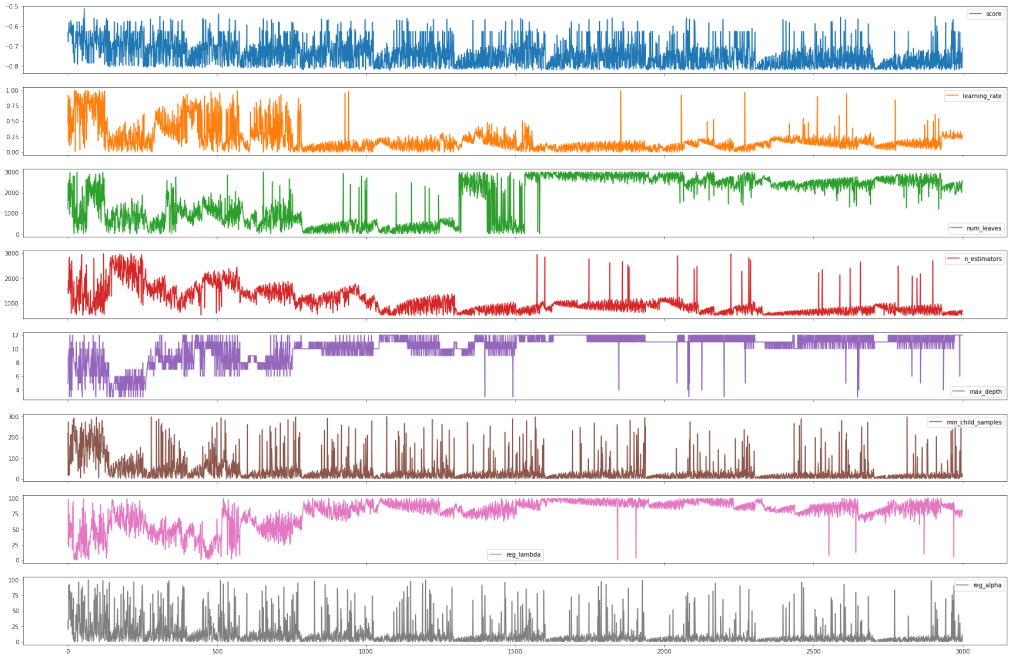
Kao što možemo vidjeti na slici 41 dobili smo 80,9264302507837% točnost sa sljedećim hiperparametrima: *learning_rate* = 0.48036624733907357, *max_depth* = 7.0, *min_child_samples* = 55.0, *n_estimators* = 1897.0, *num_leaves* = 1031.0, *reg_alpha* = 1.0083963134106373, *reg_lambda* = 23.445116446536673. Algoritmu je trebalo skoro 35 minuta da napravi optimizaciju od 3000 iteracija. Kada usporedimo rezultate sa *Random Searchom* i *Grid Searchom* vidimo da je *Grid Search* i dalje pronašao najbolje hiperparametre koji daju najbolju točnost. Također prema slici 42 vidimo da je algoritam, opet kod negdje 500-te iteracije, počeo konvergirati prema najboljoj vrijednosti hiperparametara.

5.3.6 *Hyperopt* (*Tree-structured Parzen Estimator*)

Naredni algortiam optimizacije je *Tree-structured Parzen Estimator*, a pomoću njega smo dobili sljedeće rezultate za ovaj skup podataka:

```
100%|████████████████████████████████████| 3000/3000 [1:28:00<00:00,  1.76s/tri
al, best loss: -0.8211451802507836]
{'learning_rate': 0.23117425625068863, 'max_depth': 11.0, 'min_child_samples': 2.0,
'n_estimators': 511.0, 'num_leaves': 2450.0, 'reg_alpha': 0.17852541067531755, 'reg_
lambda': 78.89269924528269}
CPU times: total: 8min 42s
Wall time: 1h 28min
```

Slika 43: Rezultati *Tree-structured Parzen Estimator*



Slika 44: Vizualizacija rezultata *Tree-structured Parzen Estimator*

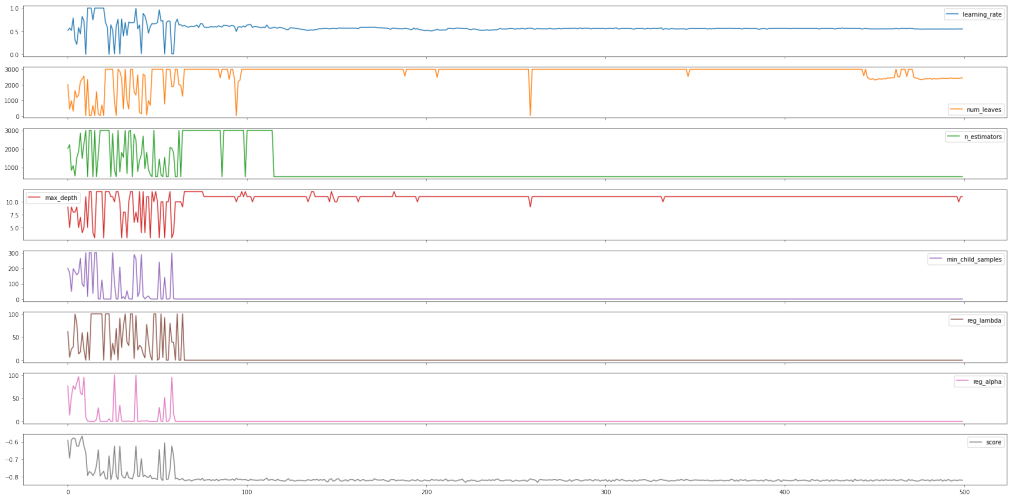
Algoritam *Tree-structured Parzen Estimator* je pronašao sljedeće hiperparametre kao najbolje za danu metriku: $learning\_rate = 0.23117425625068863$, $max\_depth = 11.0$, $min\_child\_samples = 2.0$, $n\_estimators = 511.0$, $num\_leaves = 2450.0$, $reg\_alpha = 0.17852541067531755$, $reg\_lambda = 78.89269924528269$. Pomoću njih trenirani model ima točnost od 82,11% što je najveća točnost do sada. Iako je dobio najbolju točnost, vrijeme izvođenja ovog algoritma je također bilo dugačko, čak 1 sat i 28 minuta, što je skoro 3 puta duže od *Random Searcha*.

5.3.7 Bayesova optimizacija

Zadnju optimizaciju koju provodimo je Bayesova optimizacija, a ona nam je dala sljedeće rezultate. Prikazat ću samo zadnje dvije iteracije i završne rezultate.

```
Iteration No: 499 started. Searching for the next optimal point.
Iteration No: 499 ended. Search finished for the next optimal point.
Time taken: 18.7072
Function value obtained: -0.8187
Current minimum: -0.8327
Iteration No: 500 started. Searching for the next optimal point.
Iteration No: 500 ended. Search finished for the next optimal point.
Time taken: 18.0649
Function value obtained: -0.8205
Current minimum: -0.8327
{'learning_rate': 0.5592623472278075, 'num_leaves': 3000, 'n_estimators': 500, 'max_
depth': 11, 'min_child_samples': 2, 'reg_lambda': 0.01, 'reg_alpha': 0.01}
CPU times: total: 41min 22s
Wall time: 1h 1min
```

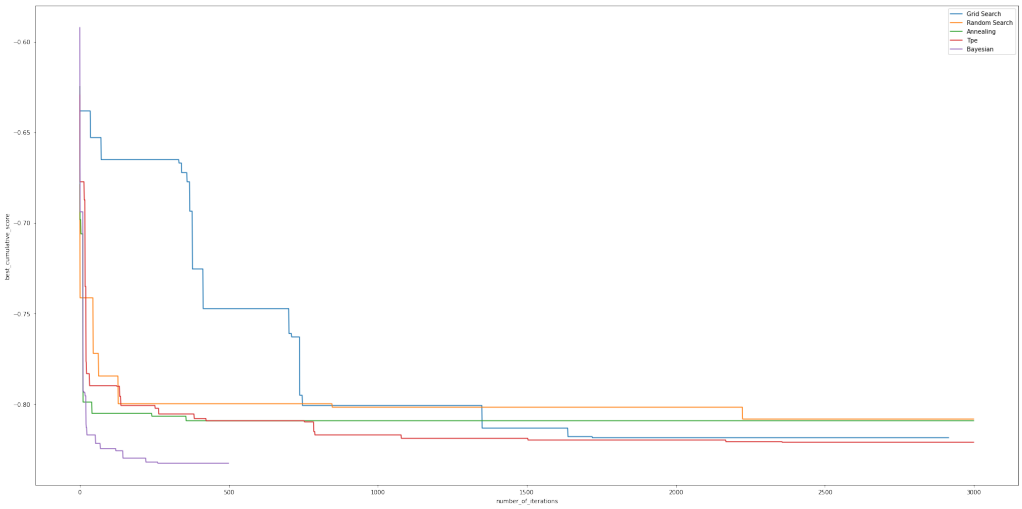
Slika 45: Rezultati Bayesove optimizacije



Slika 46: Vizualizacija rezultata Bayesove optimizacije

Prema slici 45 vidimo da je algoritmu trebalo otprilike jedan sat da završi, a dobili smo točnost od 83,27% koju smo postigli sa sljedećim hiperparametrima: *learning_rate* = 0.5592623472278075, *max_depth* = 11, *min_child_samples* = 2, *n_estimators* = 500, *num_leaves* = 3000, *reg_alpha* = 0.01, *reg_lambda* = 0.01.

5.3.8 Usporedba rezultata



Slika 47: Usporedba svi rezultata

```
Točnost modela bez optimizacije sa unakrsnom provjerom: 79.39380877742947%
Standardna devijacija modela bez optimizacije sa unakrnsom provjerom: 1.181958978115
5915 %
ROC AUC vrijednost modela bez optimizacije : 83.5367043551089 %
F1 vrijednost modela bez optimizacije : 70.73023975459594 %


Točnost modela s Grid Search-om sa unakrsnom provjerom: 81.863930250783 7%
Standardna devijacija modela s Grid Search-om sa unakrnsom provjerom: 1.330021961343
3866 %
ROC AUC vrijednost modela s Grid Search-om : 84.98621649916247 %
F1 vrijednost modela s Grid Search-om : 73.69921855230514 %


Točnost modela s Random Searchom-om sa unakrsnom provjerom: 80.8317985893417%
Standardna devijacija modela s Random Searchom-om sa unakrnsom provjerom: 1.68975878
00188894 %
ROC AUC vrijednost modela s Random Searchom-om : 84.85496021775545 %
F1 vrijednost modela s Random Searchom-om : 73.96553467669015 %
```

Slika 48: Metrike svih modela

```
Točnost modela s Hyperopt-Anneal sa unakrsnom provjerom: 80.9264302507837%
Standardna devijacija modela s Hyperopt-Anneal sa unakrnsom provjerom: 1.36552344117
9755 %
ROC AUC vrijednost modela s Hyperopt-Anneal : 84.2700418760469 %
F1 vrijednost modela s Hyperopt-Anneal : 73.36771968973308 %


Točnost modela s Hyperopt-Tpe sa unakrsnom provjerom: 82.11451802507837%
Standardna devijacija modela s Hyperopt-Tpe sa unakrnsom provjerom: 1.87289238111421
3 %
ROC AUC vrijednost modela s Hyperopt-Tpe : 85.6447487437186 %
F1 vrijednost modela s Hyperopt-Tpe : 74.54300320139113 %


Točnost modela s Bayesian sa unakrsnom provjerom: 83.27125783699059%
Standardna devijacija modela s Bayesian sa unakrnsom provjerom: 1.710908006107913 %
ROC AUC vrijednost modela s Bayesian : 85.41765912897823 %
F1 vrijednost modela s Bayesian : 75.12948207808272 %
```

Slika 49: Metrike svih modela

Kao što možemo vidjeti sa slika 47, 48 i 49 za skup podataka o pitkosti vode i izabranom metrikom točnosti Bayesova optimizacija se pokazala kao najbolji algoritam optimizacije. Iako je postupak optimizacije trajao sat vremena, točnost je za 1% bolja od modela koji smo dobili pomoću *Tree-structured Parzen Estimator* algoritma iz *Hyperopt* knjižnice kojemu je bilo potrebno i duže vremena, dok je sljedećem najboljem algoritmu (*Grid Search*) bilo potrebno 47 minuta a dobili smo za 1,5% lošiju točnost. Na kraju možemo zanemariti vremensku trajnost algoritma zbog boljih rezultata u odnosu na druge algoritme.

5.4 Dijabetes

Treći, i posljednji, skup podataka koji smo izabrali je skup podataka o pacijentima pomoću kojega ćemo predvidjeti ima li pacijent dijabetes ili ne. Za ovaj skup ćemo koristiti F1 točnost kao metriku prema kojoj će se raditi optimizacija hiperparametara.

5.4.1 Priprema podataka

Kao i kod ostalih skupova podataka prvo što moramo napraviti je pogledati kakve podatke imamo. To radimo na isti način kao i kod prvog i drugog skupa podataka (listing 11).

	6	148	72	35	0	33.6	0.627	50	1
count	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000	767.000000
mean	3.842243	120.859192	69.101695	20.517601	79.903520	31.990482	0.471674	33.219035	0.348110
std	3.370877	31.978468	19.368155	15.954059	115.283105	7.889091	0.331497	11.752296	0.476682
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243500	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	32.000000	32.000000	0.371000	29.000000	0.000000
75%	6.000000	140.000000	80.000000	32.000000	127.500000	36.600000	0.625000	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Slika 50: Opis podataka

Kao što vidimo na slici 50 imamo 8 ulaznih značajki koje označavaju osnovne dijagnostičke mjere provedene u istraživanju. Također vidimo da imamo izlaznu značajku, koja ima dva stanja 0 i 1. Vidimo da imamo 767 opservacija što nije toliko puno, pa bi optimizacija trebala biti kraća nego za drugi skup podataka. U ovom skupu podataka vidimo da nemamo nedostajućih vrijednosti pa možemo odmah prijeći na podjelu na skup za treniranje i skup za testiranje. To radimo isto kao i kod prvog i drugog skupa podataka (listing 12).

5.4.2 Treniranje modela

Nakon što smo odradili pripremu podataka puštamo model da uči nad podacima (listing 14) te dobivamo sljedeće rezultate:

```
[[78 19]
 [13 44]]
Točnost LGBM klasifikatora bez unakrsne provjere: 79.22077922077922 %
```

Slika 51: Točnost modela

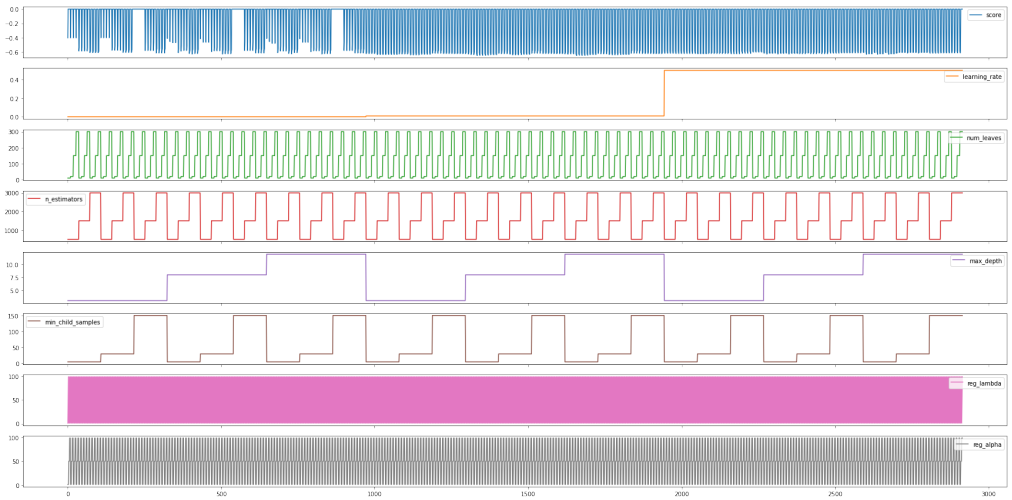
Prema 51 vidimo da smo dobili točnost od 79,22%. Moram da napomenem da je ovo točnost modela, a ne F1 točnost koju ćemo koristiti kao metriku za optimizaciju hiperparametara.

5.4.3 *Grid Search*

Uz pomoć već prije definiranog hiperparametarskog *grid-a* (listing 15 dobivamo sljedeće rezultate:

```
Fitting 10 folds for each of 2916 candidates, totalling 29160 fits
Najbolja točnost sa Grid Search-om: 65.22410911365317 %
Najbolji parametri sa Grid Search-om: {'learning_rate': 0.01, 'max_depth': 8, 'min_c
hild_samples': 5, 'n_estimators': 1500, 'num_leaves': 10, 'reg_alpha': 1, 'reg_lambd
a': 99}
CPU times: total: 33.5 s
Wall time: 12min 27s
```

Slika 52: Rezultati *Grid Searcha*



Slika 53: Vizualizacija rezultata *Grid Searcha*

Kao što možemo vidjeti sa slike 52, algoritmu je bilo potrebno 12 minuta da završi, a dobili smo prvu F1 točnost za ovaj skup podataka (65.22%). Hiperparametri s kojima smo dobili ove rezultate su: *learning_rate* = 0.01,

max_depth = 8, *min_child_samples* = 5, *n_estimators* = 1500, *num_leaves* = 10, *reg_alpha* = 1, *reg_lambda* = 99. Također na slici 53 je prikazana vizualizacija pretrage hiperparametara po svakoj iteraciji.

5.4.4 Nasumično pretraživanje

Sljedeća optimizacijska tehnika je nasumično pretraživanje. Pomoću nje smo dobili sljedeće rezultate:

```
Fitting 10 folds for each of 3000 candidates, totalling 30000 fits
Najbolja točnost sa Random Search-om: 65.35 %
Najbolji parametri prema Random Search-om: {'learning_rate': 0.8898590516456748, 'ma
x_depth': 10, 'min_child_samples': 10, 'n_estimators': 870, 'num_leaves': 1462, 'reg
_alpha': 5.602075749321284, 'reg_lambda': 14.226715824971288}
CPU times: total: 28.6 s
Wall time: 5min 18s
```

Slika 54: Rezultati nasumičnog pretraživanja

Pomoću *Random Searcha* smo dobili F1 točnost od 65.35%, što možemo vidjeti na slici 54, što je malo bolje od *Grid Searcha*, ali brzina algoritma je bila duplo brža(*Random Search*: 5 minuta, *Grid Search*: 12 minuta). Hiperparametre koje smo dobili koristeći ovaj algoritam su: *learning_rate* = 0.8898590516456748, *max_depth* = 10, *min_child_samples* = 10, *n_estimators* = 870, *num_leaves* = 1462, *reg_alpha* = 5.602075749321284, *reg_lambda* = 14.226715824971288.

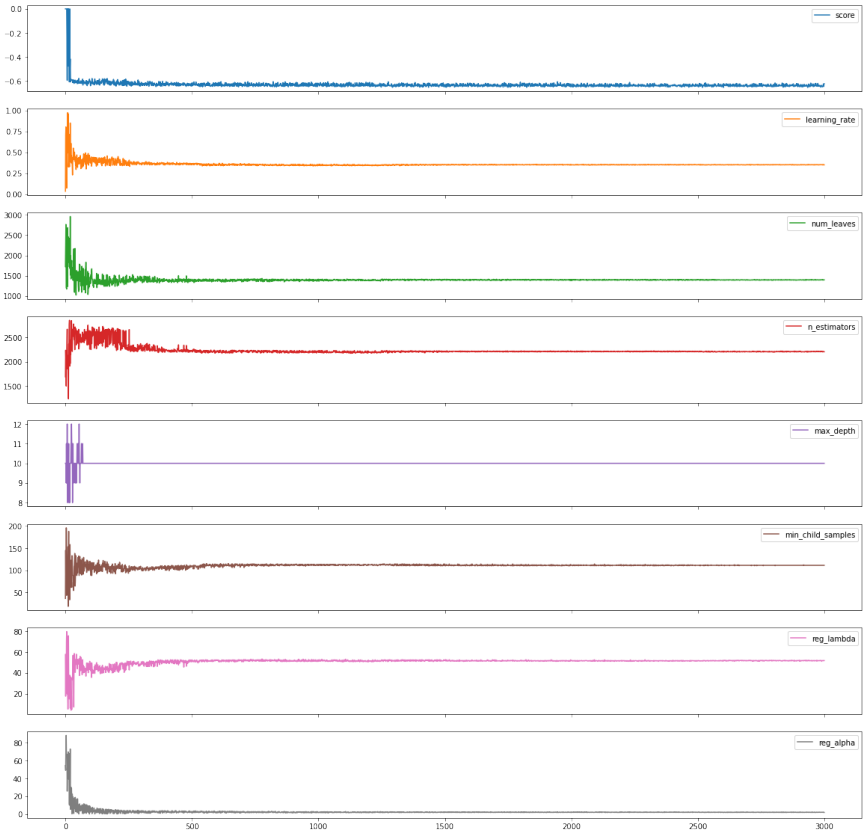
5.4.5 *Hyperopt* (*Simulated Anneal*)

Rezultati dobiveni koristeći ovaj algoritam su sljedeći:

```
100%|████████████████████████████████████| 3000/3000 [12:34<00:00,  3.98trial
/s, best loss: -0.6531427484072164]
{'learning_rate': 0.35127324841339935, 'max_depth': 10.0, 'min_child_samples': 111.
0, 'n_estimators': 2209.0, 'num_leaves': 1395.0, 'reg_alpha': 1.8914110321208635, 'r
eg_lambda': 51.7555052227802}
CPU times: total: 17.7 s
Wall time: 12min 34s
```

Slika 55: Rezultati *Simulated Anneala*

Kao što možemo vidjeti na slici 55 dobili smo 65.31% F1 točnosti sa sljedećim hiperparametrima: *learning_rate* = 0.35127324841339935, *max_depth* = 10.0, *min_child_samples* = 111.0, *n_estimators* = 2209.0, *num_leaves* = 1395.0, *reg_alpha* = 1.8914110321208635, *reg_lambda* = 51.7555052227802. Algoritma je napravio optimizaciju u 12 minuta i 34 sekunde. Kada usporedimo rezultate sa *Random Search-om* i *Grid Search-om* ovaj put vidimo



Slika 56: Vizualizacija rezultata *Simulated Anneala*

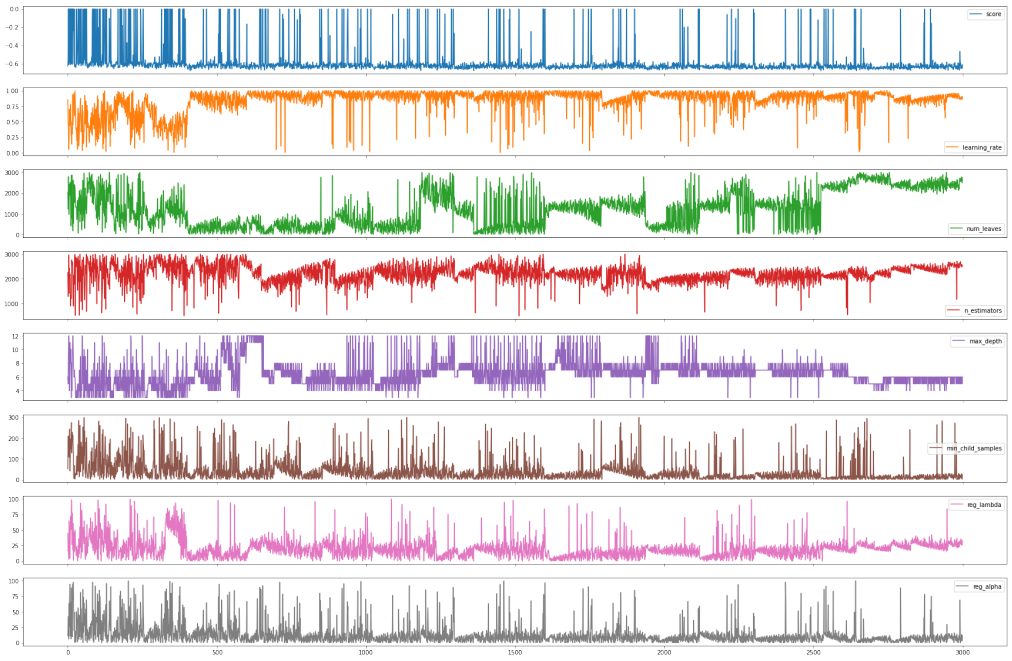
da je *Random Search* bio uspješniji u pronalasku bolje F1 točnosti. Također prema slici 56 vidimo da je algoritam, opet negdje kod 500-te iteracije, počeo konvergirati prema najboljoj vrijednosti hiperparametara.

5.4.6 *Hyperopt* (*Tree-structured Parzen Estimator*)

Naredni algoritam optimizacije je *Tree-structured Parzen Estimator*, a pomoću njega smo dobili sljedeće rezultate za ovaj skup podataka:

```
100%|████████████████████████████████████| 3000/3000 [19:53<00:00,  2.51trial
/s, best loss: -0.6786142980133753]
{'learning_rate': 0.8529784441999736, 'max_depth': 7.0, 'min_child_samples': 2.0, 'n
_estimators': 2128.0, 'num_leaves': 2281.0, 'reg_alpha': 5.036940577536111, 'reg_lam
bda': 17.50011765032669}
CPU times: total: 7min 56s
Wall time: 19min 53s
```

Slika 57: Rezultati *Tree-structured Parzen Estimator*



Slika 58: Vizualizacija rezultata *Tree-structured Parzen Estimator*

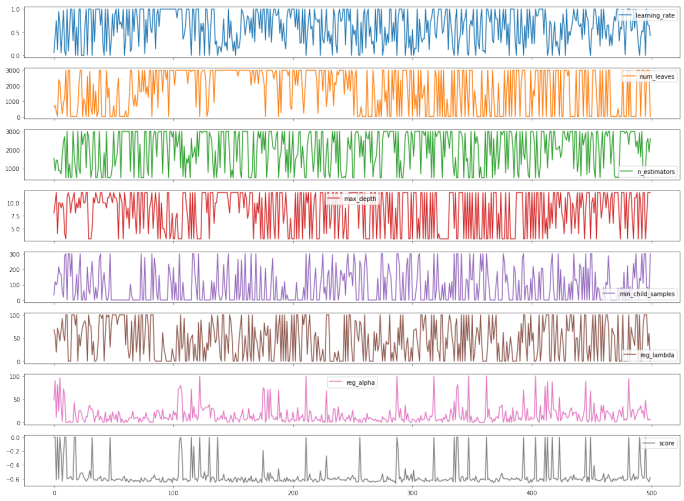
Algoritam *Tree-structured Parzen Estimator* je pronašao sljedeće hiperparametre kao najbolje za danu metriku: $learning\_rate = 0.8529784441999736$, $max\_depth = 7.0$, $min\_child\_samples = 2.0$, $n\_estimators = 2128.0$, $num\_leaves = 2281.0$, $reg\_alpha = 5.036940577536111$, $reg\_lambda = 17.50011765032669$. Model s ovim hiperparametrima je dobio 67.86% F1 točnost, što je za sada najbolji rezultat. Također kao i u prethodna 2 slučaja ovaj algoritam je nešto vremenski duži od ostalih algoritama, s vremenom izvođenja od skoro 20 minuta.

5.4.7 Bayesova optimizacija

Zadnju optimizaciju koju provodimo je Bayesova optimizacija, a ona nam je dala sljedeće rezultate. Prikazat ću samo zadnje dvije iteracije i završne rezultate.

```
Iteration No: 499 started. Searching for the next optimal point.
Iteration No: 499 ended. Search finished for the next optimal point.
Time taken: 23.1053
Function value obtained: -0.6451
Current minimum: -0.6688
Iteration No: 500 started. Searching for the next optimal point.
Iteration No: 500 ended. Search finished for the next optimal point.
Time taken: 22.8289
Function value obtained: -0.5814
Current minimum: -0.6688
{'learning_rate': 1.0, 'num_leaves': 3000, 'n_estimators': 3000, 'max_depth': 9, 'mi
n_child_samples': 2, 'reg_lambda': 17.32012616551272, 'reg_alpha': 5.00924487949756
2}
CPU times: total: 43min 22s
Wall time: 1h 5min 6s
```

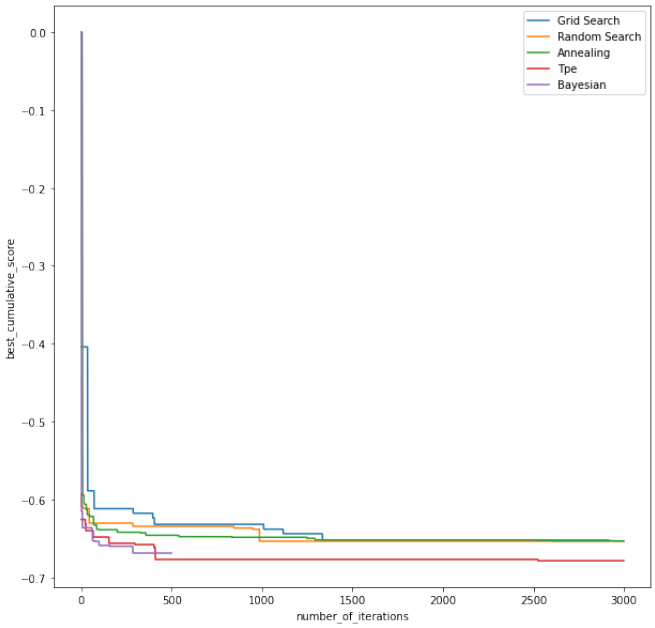
Slika 59: Rezultati Bayesove optimizacije



Slika 60: Vizualizacija rezultata Bayesove optimizacije

Prema slici 59 vidimo da je algoritmu trebalo otprilike, kao i u prethodnim skupovima podataka, jedan sat da završi, a dobili smo F1 točnost od 66,88% koju smo postigli sa sljedećim hiperparametrima: $learning\_rate = 1.0$, $max\_depth = 9$, $min\_child\_samples = 2$, $n\_estimators = 3000$, $num\_leaves = 3000$, $reg\_alpha = 5.009244879497562$, $reg\_lambda = 17.32012616551272$.

5.4.8 Usporedba rezultata



Slika 61: Usporedba svi rezultata

```
Točnost modela bez optimizacije sa unakrsnom provjerom: 75.35885167464116%
Standardna devijacija modela bez optimizacije sa unakrnsom provjerom: 3.043297941086
206 %
ROC AUC vrijednost modela bez optimizacije : 80.11623931623933 %
F1 vrijednost modela bez optimizacije : 63.93473367218954 %


Točnost modela s Grid Search-om sa unakrsnom provjerom: 77.19412166780589%
Standardna devijacija modela s Grid Search-om sa unakrnsom provjerom: 4.030123530753
03 %
ROC AUC vrijednost modela s Grid Search-om : 82.56467236467235 %
F1 vrijednost modela s Grid Search-om : 65.22410911365317 %


Točnost modela s Random Searchom-om sa unakrsnom provjerom: 76.67634996582365%
Standardna devijacija modela s Random Searchom-om sa unakrnsom provjerom: 5.45201402
9189117 %
ROC AUC vrijednost modela s Random Searchom-om : 82.43675213675215 %
F1 vrijednost modela s Random Searchom-om : 65.35122041919423 %
```

Slika 62: Metrike svih modela

```
Točnost modela s Hyperopt-Anneal sa unakrsnom provjerom: 76.79767600820233%
Standardna devijacija modela s Hyperopt-Anneal sa unakrnsom provjerom: 4.62955717876
5428 %
ROC AUC vrijednost modela s Hyperopt-Anneal : 80.24729344729344 %
F1 vrijednost modela s Hyperopt-Anneal : 65.31427484072164 %


Točnost modela s Hyperopt-Tpe sa unakrsnom provjerom: 78.36637047163364%
Standardna devijacija modela s Hyperopt-Tpe sa unakrnsom provjerom: 5.14759254796743
3 %
ROC AUC vrijednost modela s Hyperopt-Tpe : 82.7219373219373 %
F1 vrijednost modela s Hyperopt-Tpe : 67.86142980133752 %


Točnost modela s Bayesian sa unakrsnom provjerom: 77.7153110047847%
Standardna devijacija modela s Bayesian sa unakrnsom provjerom: 4.8852844881398685 %
ROC AUC vrijednost modela s Bayesian : 82.85327635327636 %
F1 vrijednost modela s Bayesian : 66.8779279610601 %
```

Slika 63: Metrike svih modela

Kao što možemo vidjeti sa slika 61, 62 i 63 za skup podataka o dijabetesu i izabranom metrikom F1 točnost *Tree-structured Parzen Estimator* algoritam iz *Hyperopt* knjižnice se pokazao kao najbolji algoritam optimizacije. Također vidimo da model izgrađen sa hiperparametrima dobivenim iz ovog algoritma ima najbolju točnost, ROC_AUC vrijednost i F1 vrijednost u odnosu na sve ostale modele izgrađene sa hiperparametrima dobivenima od ostalih optimizacijskih tehnika.

6 Zaključak

Prilikom ovog istraživanja isprobali smo i koristili veoma dobar klasifikator, *LightGBM* klasifikator, koji je i bez optimizacije dobro klasificirao podatke. Uz optimizacijske tehnike smo još više povećali njegovu točnost koristeći određene metrike za svaki od 3 skupa podataka. Kao što smo vidjeli brzina izvođenja algoritma dosta ovisi o broju opservacija koje imamo u skupu podataka, ali i o hiperparametrima o kojima ovisi izgled stabala odluke. Također smo vidjeli da je pretraživanje po rešetci veoma iscrpno što se tiče vremena, i u većini slučaja smo pomoću njega dobili najlošije rezultate. Nasumično pretraživanje je u većini slučaja bilo drugo najlošije što se tiče rezultata, ali je svaki put trajalo dosta kraće od ostalih algoritama. Bayesova optimizacija je uvijek bila jedna od najboljih što se tiče rezultata, ali je također svaki put trajala duže od ostalih, u nekim slučajima i tri puta duže. Algoritmi iz knjižnice *Hyperopt*, *Tree-structured Prazen Estimator* i *Simmulated Anneal*, su u sva tri slučaja davali veoma dobre rezultate u prihvatljivom vremenskom razdoblju. Na kraju mogu reći da je svaki od algoritama optimizacije dobar za neku stvar: Pretraživanje po rešetci je dobro za mali broj hiperparametara, Nasumično pretraživanje je dobro za brzu optimizaciju s dobrim, ali ne idealnim, rezultatima, dok su ostala tri algoritma dobra za pronalazak idealnih hiperparametara, uz to da se mora paziti na vrijeme koje je potrebno da se izvrše algoritmi.

Literatura

[1] Domo. Data never sleeps 6.0, 2018.

[2] Vladimir Nasteski. An overview of the supervised machine learning methods. *HORIZONS.B*, 4:51–62, 12 2017.

[3] A. Thakur. *Approaching (Almost) Any Machine Learning Problem*. Abhishek Thakur, 2020.

[4] Yufeng G. The 7 steps of machine learning, 2017.

[5] Nils J. Nilsson. Introduction to machine learning: An early draft of a proposed textbook. pages 175-188. http://robotics.stanford.edu/people/nilsson/mlbook.ht ml, 1996.

[6] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[9] Pablo Aznar. Decision trees: Gini vs entropy, 2020. Resource available on https://quantdare.com/decision-trees-gini-vs-entropy/.

[10] Sam T. Entropy: How decision trees make decisions, 2019.

[11] Nikolay Kyurkchiev and Svetoslav Markov. *Sigmoid Functions Some Approximation and Modelling Aspects: Some Moduli in Programming Environment MATHEMATICA*. 08 2015.

[12] IBM. Overfitting, 2021.

[13] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.

[15] Ade Nurhopipah and Uswatun Hasanah. Dataset splitting techniques comparison for face classification on cctv images. *Indonesian Journal of Computing and Cybernetics Systems*, 14:341–352, 2020.

[16] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[17] Robert E Schapire. A brief introduction to boosting. In *Ijcai*, volume 99, pages 1401–1406. Citeseer, 1999.

[18] C. Wade and K. Glynn. *Hands-On Gradient Boosting with XGBoost and scikit-learn: Perform accessible machine learning and extreme gradient boosting with Python*. Packt Publishing, 2020.

[19] K. Jome. *Gradient Boosting Trees: A Beginner's Guide For Gradient Boosting: Decision Tree Machine Learning Projects*. Independently Published, 2021.

[20] Cory Maklin. Gradient boosting decision tree algorithm explained, 2019.

[21] Lu Ye, Saadya Fahad, Musaddak Maher, and Mou Leong Tan. Bayesian regularized neural network model development for predicting daily rainfall from sea level pressure data: Investigation on solving complex hydrology problem. *Complexity*, 2021:1–14, 04 2021.

[22] Aarshay. Complete machine learning guide to parameter tuning in gradient boosting (gbm) in python, 2016.

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.

[24] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[25] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019.

[26] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[27] Alex Frid, Hananel Hazan, Ester Koilis, Larry Manevitz, Maayan Merhav, and Gal Star. *The Existence of Two Variant Processes in Human Declarative Memory: Evidence Using Machine Learning Classification Techniques in Retrieval Tasks*, volume 9770, pages 117–133. 01 2016.

[28] Vikram Kamboj, S.K. Bath, and J.S. Dhillon. A novel hybrid de–random search approach for unit commitment problem. *Neural Computing and Applications*, 28, 07 2017.

[29] Eric Brochu, Vlad Cora, and Nando Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 12 2010.

[30] Tim Head, Manoj Kumar, Holger Nahrstaedt, Gilles Louppe, and Iaroslav Shcherbatyi. scikit-optimize/scikit-optimize, October 2021.

[31] Jayawant N. Mandrekar. Receiver operating characteristic curve in diagnostic test assessment. *Journal of Thoracic Oncology*, 5(9):1315–1316, 2010.

[32] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[33] Fulvio De Santis and Fulvio Spezzaferri. Alternative bayes factors for model selection. *The Canadian Journal of Statistics / La Revue Canadienne de Statistique*, 25(4):503–515, 1997.

[34] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

Popis slika

List of Listings