UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
ODDELEK ZA FIZIKO

Matematično-fizikalni praktikum

12. naloga: Strojno učenje

Žiga Šinigoj, 28191058

Ljubljana, marec 2022

1 Uvod

Dandanes je uporaba različnih algoritmov strojnega učenja (Machine Learning, ML) v znanosti že rutinsko opravilo. Poznamo tri osnovne vrste stojnega učenja:

- Nadzorovano učenje (Supervised learning):
  - Klasifikacija (Classification): sortiranje v različne kategorije.
  - Regresija (Regression): modeliranje oz. 'fitanje' napovedi.
- Nenadzorovano učenje (npr. sam najdi kategorije).
- Stimulirano učenje (Artificial Intelligence v ožjem pomenu besede).

V fiziki (in tej nalogi), se tipično ukvarjamo s prvo kategorijo, bodisi za identifikacijo novih pojavov/delcev/... ali pa za ekstrakcijo napovedi (netrivialnih funkcijskih odvisnosti etc).
ML algoritmi imajo prednost pred klasičnim pristopom, da lahko učinkovito razdrobijo kompleksen problem na enostavne elemente in ga ustrezno opišejo:

- pomisli na primer, kako bi bilo težko kar predpostaviti/uganiti pravo analitično funkcijo v več dimenzijah (in je npr. uporaba zlepkov (spline interpolacija) mnogo lažja in boljša).
- Pri izbiri/filtriranju velike količine podatkov z mnogo lastnostmi (npr dogodki pri trkih na LHC) je zelo težko najti količine, ki optimalno ločijo signal od ozadnja, upoštevati vse korelacije in najti optimalno kombinacijo le-teh...

Če dodamo malce matematičnega formalizma strojnega učenja: Predpostavi, da imamo na voljo nabor primerov $\mathcal{D} = \{(\mathbf{x}_k, y_k)\}_{k=1.N}$, kjer je $\mathbf{x}_k = \left(x_k^1, \ldots, x_k^M\right)$ naključno izbrani vektor M lastnosti (karakteristik) in je $\mathbf{y}_k = \left(y_k^1, \ldots, y_k^Q\right)$ vektor Q ciljnih vrednosti, ki so lahko bodisi binarne ali pa realna števila[1]. Vrednosti $(\mathbf{x}_k, \mathbf{y}_k)$ so neodvisne in porazdeljene po neki neznani porazdelitvi $P(\cdot, \cdot)$. Cilj ML metode je določiti (priučiti) funkcijo $h : \mathbb{R}^Q \to \mathbb{R}$, ki minimizira pričakovano vrednost *funkcije izgube (expected loss)*

$$\mathcal{L}(h) = \mathbb{E}\, L(\mathbf{y}, \mathbf{h}(\mathbf{x})) = \frac{1}{N} \sum_{k=1}^{N} L(\mathbf{y_k}, \mathbf{h}(\mathbf{x_k})).$$

Tu je $L(\cdot, \cdot)$ gladka funkcija, ki opisuje oceno za kvaliteto napovedi, pri čemer so vrednosti $(\mathbf{x}, \mathbf{y})$ neodvisno vzorčene iz nabora $\mathcal{D}$ po porazdelitvi P. Po koncu učenja imamo torej na voljo funkcijo $\mathbf{h}(\mathbf{x})$, ki nam za nek vhodni nabor vrednosti $\hat{\mathbf{x}}$ poda napoved $\hat{\mathbf{y}} = \mathbf{h}(\hat{\mathbf{x}})$, ki ustrezno kategorizira ta nabor vrednosti.

Funkcije $\mathbf{h}$ so v praksi sestavljene iz (množice) preprostih funkcij z (nekaj) prostimi parametri, kar na koncu seveda pomeni velik skupni nabor neznanih parametrov in zahteven postopek minimizacije funkcije izgube.

Osnovni gradnik odločitvenih dreves je tako kar stopničasta funkcija $H(x_i - t_i) = 0, 1$, ki je enaka ena za $x_i > t_i$ in nič drugače in kjer je x_i ena izmed karakteristik in t_i neznani parameter. Iz skupine takšnih funkcij, ki predstavljajo binarne odločitve lahko skonstruiramo končno uteženo funkcijo

$$\mathbf{h}(\mathbf{x}) = \sum_{i=1}^{J} \mathbf{a}_i\, H(x_i - t_i),$$

kjer so $\mathbf{a}_i$ vektorji neznanih uteži. Tako t_i kot $\mathbf{b}_i$, lahko določimo v procesu učenja. Nadgradnjo predstavljajo nato *pospešena* odločitvena drevesa (BDT), kjer nadomestimo napoved enega drevesa z uteženo množico le-teh, tipično dobljeno v ustreznih iterativnih postopkih (npr. AdaBoost, Gradient Boost ipd.).

Pri nevronskih mrežah je osnovni gradnik t.i. *perceptron*, ki ga opisuje preprosta funkcija

$$h_{w,b}(\mathbf{X}) = \vartheta\left(\mathbf{w}^T \cdot \mathbf{X} + b\right),$$

[1] ...ali pa še kaj, prevedljive na te možnosti, npr barve...

kjer je $\mathbf{X}$ nabor vhodnih vrednosti, $\mathbf{w}$ vektor vrednosti uteži, s katerimi tvorimo uteženo vsoto ter b dodatni konstatni premik (bias). Funkcija ϑ je preprosta gladka funkcija (npr. arctan), ki lahko vpelje nelinearnost v odzivu perceptrona. Nevronska mreža je nato sestavljena iz (poljubne) topologije takšnih perceptronov, ki na začetku sprejme karakteristiko dogodka $\mathbf{x}$ v končni fazi rezultirajo v napovedi $\hat{\mathbf{y}}$, ki mora seveda biti čim bližje ciljni vrednosti $\mathbf{y}$. Z uporabo ustrezne funkcije izgube (npr MSE: $\mathcal{L}(h) = \mathbb{E}\,||\mathbf{y} - \hat{\mathbf{y}}||^2$), se problem znova prevede na minimizacijo, kjer iščemo optimalne vrednosti (velikega) nabora uteži $\mathbf{w}_i$ ter b_i za vse perceptrone v mreži. Globoke nevronske mreže (DNN) niso nič drugega, kot velike nevronske mreže ali skupine le-teh.
Že namizni računalniki so dovolj močni za osnovne računske naloge, obstajajo pa tudi že zelo uporabniku prijazni vmesniki v jeziku Python, na primer:

- Scikit-Learn (*scikit-learn.org*): odprtokodni paket za strojno učenje,
- TensorFlow (*tensorflow.org*): odprtokodni Google-ov sistem za ML, s poudarkom na globokih nevronskih mrežah (Deep Neural Networks, DNN) z uporabo vmesnika Keras. Prilagojen za delo na GPU in TPU.
- Catboost: (*Catboost.ai*) : odprtokodna knjižnica za uporabo pospešenih odločitvenih dreves (Boosted Decision Trees, BDT). Prilagojena za delo na GPU.

Za potrebe naloge lahko uporabimo tudi spletni vmesnik Google Collab (*colab.research.google.com*), ki dopušča omejen dostop do večjih računskih zmogljivosti.

2 Naloga

Na spletni učilnici je na voljo material (koda, vzorci) za ločevanje dogodkov Higgsovega bozona od ostalih procesov ozadja. V naboru simuliranih dogodkov je 18 karakteristik (zveznih kinematičnih lastnosti), katerih vsaka posamezno zelo slabo loči 'signal' od ozadja, z uporabo BDT ali (D)NN, pa lahko tu dosežemo zelo dober uspeh. Na predavanjih smo si ogledali glavne aspekte pomembne pri implementaciji ML, kot so uporaba ustreznih spremenljivk (GIGO), učenje in prekomerno učenje (training/overtraining), vrednotenje uspeha metode kot razmerje med učinkovitostjo (efficiency) in čistostjo (precision) vzorca (Receiver Operating Characteristic, ROC). Določi uspešnost obeh metod (in nariši ROC) za nekaj tipičih konfiguracij BDT in DNN, pri čemer:

- Študiraj vpliv uporabljenih vhodnih spremenljivk - kaj, če vzamemo le nekatere?
- Študiraj BDT in NN in vrednoti uspešnost različnih nastavitev, če spreminjaš nekaj konfiguracijskih parametrov (npr. število perceptronov in plasti nevronskih mrež pri DNN in število dreves pri BDT).

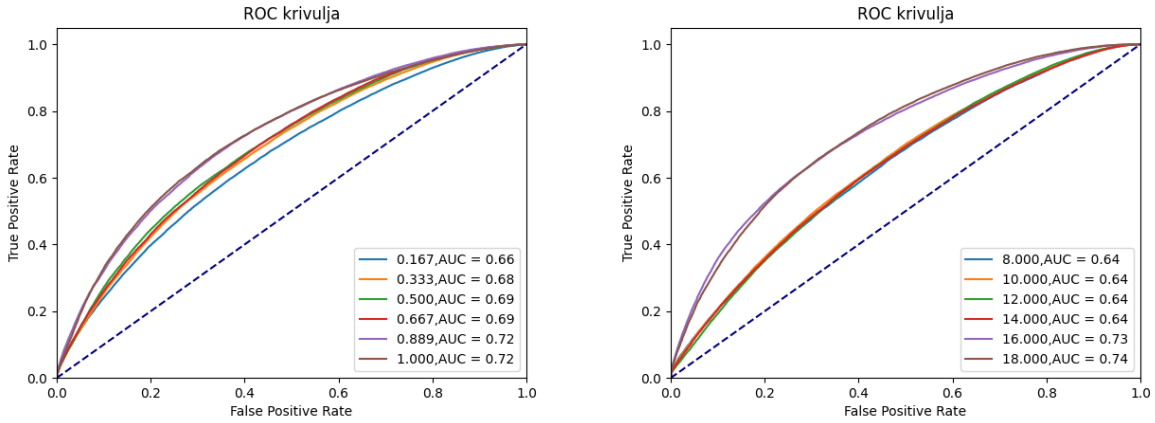
Dodatna naloga: Implementiraj distribucije iz 'playground' zgleda v BDT (lahko tudi RandomForests) in DNN, te distribucije so na voljo v vseh popularnih ML paketih (npr. Scikit...).

3 Rezultati

3.1 DNN

Za konstrukcijo nevronskih mrež sem uporabljal TensorFlow. Globoke nevronske mreže sestavimo tako, da nevronski mreži dodajamo 'skrite' plasti med vhodno in izhodno plastjo. Če na je na sliki označeno število nevronskih plasti npr. 15, je mišljeno 15 plasti + izhodna plast. Prva plasti ima vhodno obliko prilagojeno številu vhodnih lastnosti oz. spremenljivk. V našem primeru je to 18. Število perceptronov v vseh plasteh razen v zadnji je enako. Vse plasti imajo aktivacijsko funkcijo 'ReLu' razen izhodne plastni, ki je sestavljena iz enega perceptrona z aktivcijsko funkcijo 'sigmoid'. Za iskanje minimalne vrednosti 'loss' funkcije je porabljen algoritem 'ADAM'. Vhodni podatki so normirani na interval [0,1]. Naša nevronska mreža je binarni klasifikator, saj loči podatke na signal ali ozadje. Za določanje učinkovitosti nevronske mreže se uporablja ROC krivulja (in tudi njena ploščina), ki je merilo za uspešnost delovanja nevronske

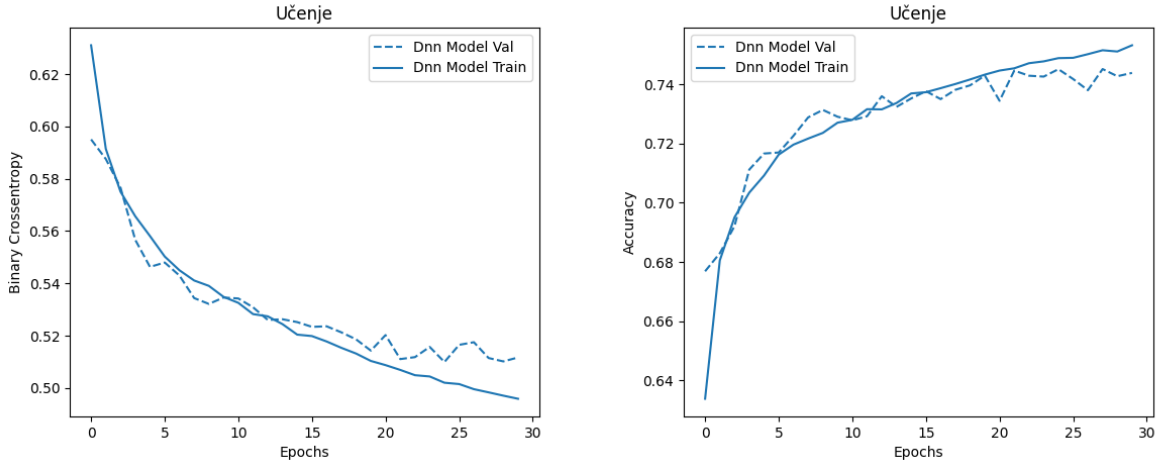
mreže pri ločevanju podatkov v 2 skupini. Perfektni klasifikator bi imel ploščino 1, naključni klasifikator pa 0.5. Za risanje ROC krivulje sem uporabil neodvisne podatke iz mape 'valid'.
Najprej lahko preverimo kako velikost vhodnih podatkov za treniranje mreže in število podanih razpadov oz. vhodnih lastnosti (slika 1) vpliva na ROC krivuljo in njeno ploščino (AUC). Nevronska mreža je sestavljena iz 10 plasti po 32 nevtronov + izhodna plast. V tem primeru je omogočeno zgodnje ustavljanje algoritma, da ne pride do 'over-fitanja'. Velikost podatkov vpliva na učenje nevronske mreže in več podatkov ponavadi pomeni boljšo klasifikacijo. Pri omejitvi števila vhodnih lastnosti (slika 1b) je opaziti precejšnjo razliko, če uporabimo manj kot 16 parametrov od 18. Verjetno je tudi pomembno katere paramtere spustimo, ampak se nisem preveč poglabljal v to.



(a) (b)

Slika 1: a)ROC krivulja pri različnem deležu vhodnih podatkov za treniranje mreže. b)ROC krivulja pri različnem številu vhodnih paramterov (lastnosti).

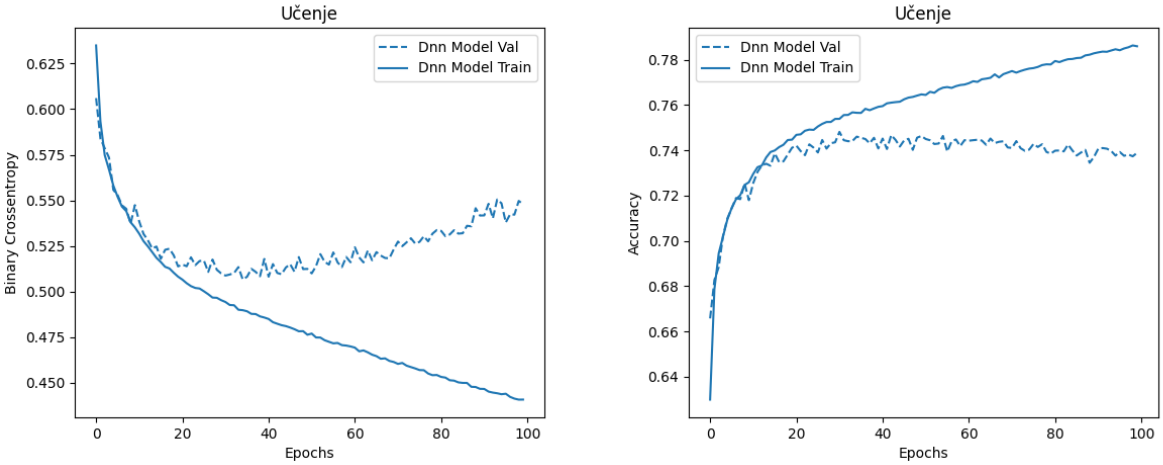
Pri učenju je potrebno paziti na število paramterov, ki jih mora nevronska mreža optimizirati, kar je povezano z velikostjo nevronske mreže. Če je število parametrov preveliko in je večje od števila podatkov oz. v enakem redu kot število podatkov, je nevarno, da bo prišlo do 'over-fitanja' in bo sicer nevronska mreža dosegla odlične rezultate na učnem vzorcu, ampak zato toliko slabše na testnem vzorcu oz. na klasifickaciji drugih vzorcev. Če imamo premalo parametrov nevronska mreža doseže slabše rezultate v klasifikaciji kot bi jih lahko. Učinkovitost je odvisna tudi od števila prehodov (epoch) celotnega nabora podatkov za treniranje skozi mrežo (slika 5a). Za primer over-fitanja sem vzel nevronsko mrežo z 10 plastmi po 100 perceptroni, kar je okvirno 100000 parametrov, če so vsi povezani z vsemi. Z uporabo funkcije 'earlystopping' lahko



(a) (b)

Slika 2: Val-označuje testne podatke, Train-označuje podatke na katerih se mreža uči. a)Ena izmed 'cost' funkcij v odvisnosti od ciklov treniranja (Epochs). b)Natančnost v odvisnosti od ciklov treniranja (Epochs)

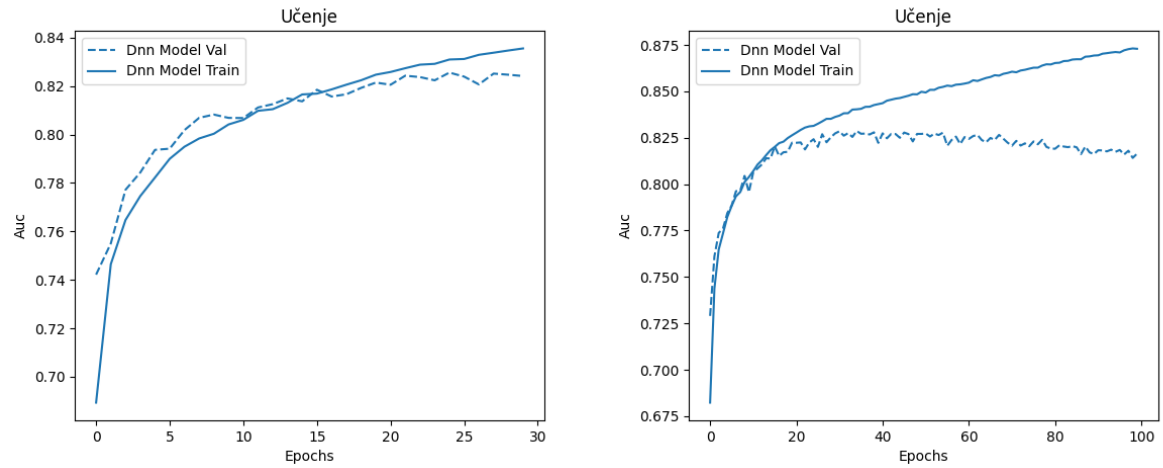
ustavimo algoritem ko doseže 'loss' funkcija minimum, tudi če ne opravi vseh ciklov treniranja (slika 2). Na sliki lahko vidimo potek učenja. 'Binary Crossentropy' (slika 2a) je ena od 'cost' funkciji, ki jo je potrebno minimizirati, da dosežemo najboljše delovanje nevronske mreže, natančnost (slika 2b) pa raste s cikli treniranja na podatkih. Če ne ustavim treniranja ob doseženem minimumu dobimo boljše rezultate za trenirano množico, vendar nevronska mreža deluje slabše za množice, ki niso enake kot trenirane (slika 3), kar pa ni cilj nevronske mreže.



(a) (b)

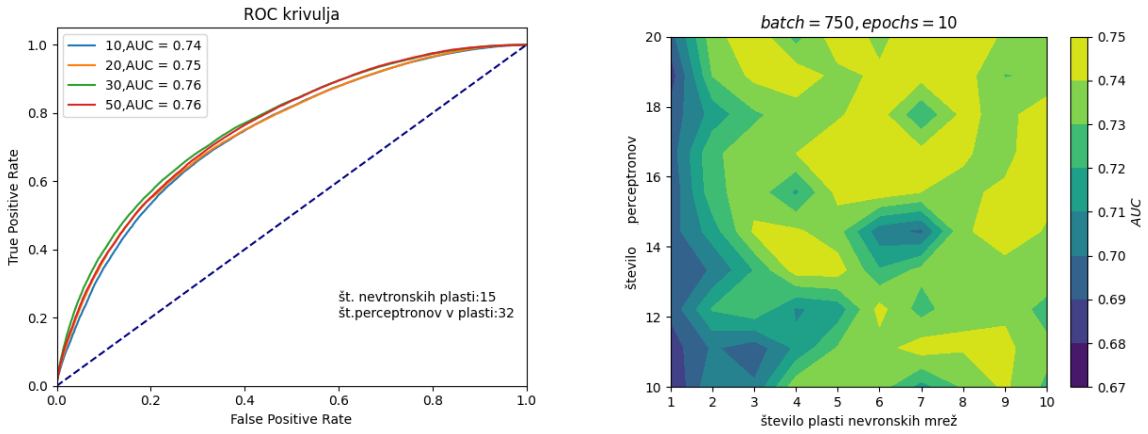
Slika 3: Val-označuje testne podatke, Train-označuje podatke na katerih se mreža uči. a)Ena izmed 'cost' funkcij v odvisnosti od ciklov treniranja (Epochs). b)Natančnost v odvisnosti od ciklov treniranja (Epochs)

Pogledamo lahko še odvisnost ploščine (AUC) od treniranja in pretiranega treniranja nevronske mreže (slika 4 a in b). Če s treniranjem ne dosežemo minimuma zaradi premajhnega števila iteracij se, kot pričakovano, nevronska mreža izboljšuje (slika 5a). Zanimivo je tudi pogledati ploščino pod ROC krivuljo v odvisnosti od števila perceptronov in nevronskih plasti (slika 5b). Izgleda da je v splošnem dobro vzeti več plasti, ampak ni pa mogoče razbrati neke odvisnosti, ki bi povedala kakšno je optimalno število perceptronov in plasti.



(a) (b)

Slika 4: Val-označuje testne podatke, Train-označuje podatke na katerih se mreža uči. a)AUC v odvisnosti od ciklov treniranja (Epochs). b)AUC v odvisnosti od ciklov treniranja (Epochs), kjer je onemogočena funkcija 'earlystopping'.

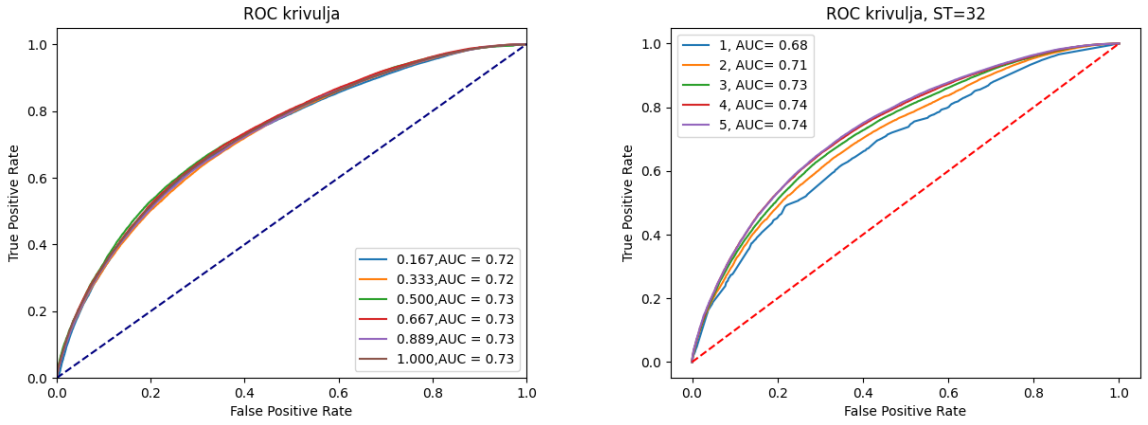


(a) (b)

Slika 5: a)ROC v odvisnosti od števila Epoch. b)AUC v odvisnosti od števila perceptronov in števila plasti.

3.2 Boosted Decision Trees in Gradient Boosting

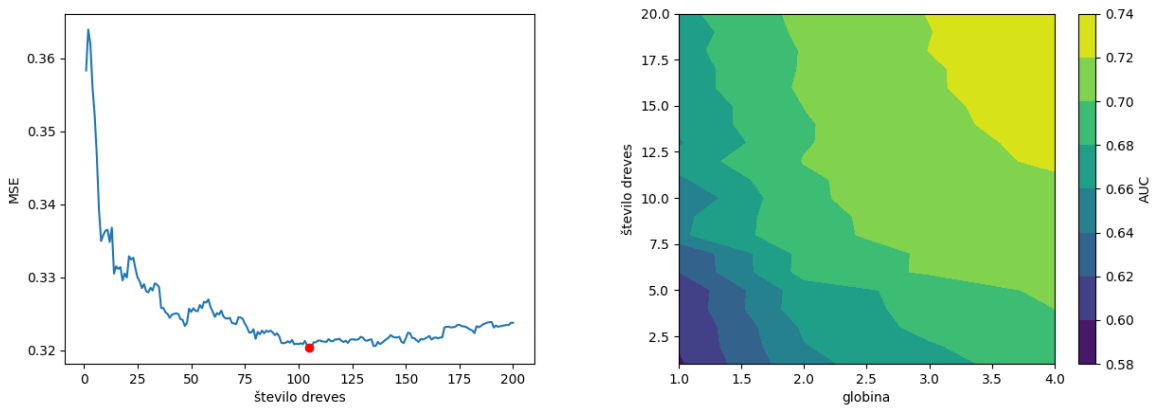
Za implementacijo pospešenih določitvenih dreves sem uporbil Scikit-Learn-ov klasifikator AdaBoost. Algoritem je zasnovan tako, da najprej določi uteži na podatkih za treniranje, nato pa se osredotoči na več podmnožic podatkov, ki jih je prejšnja iteracija premalo utežila in tako iterativno popravlja vrednosti uteži s treniranjem na določenih podmnožicah. Tako kot pri DNN lahko najprej pogledamo, kako se odziva algoritem na manjše število podatkov (slika 6a). Na sliki je videti, da so BDT precej manj občutljiva na količino podatkov kot DNN, in že samo 16 % delež vseh podatkov nam da skoraj enake rezultate kot če uporabimo celotno količino podatkov za treniranje algoritma. Slika 6b prikazuje ROC krivuljo BDT v odvisnosti od globine. Parameter globine pomeni, skozi koliko pogojev se klasificirajo podatki v enem drevesu. Če je globina 1, pomeni da imamo samo en pogoj in 2 možni podmnožici, v kateri se razvrstijo podatki.



(a) (b)

Slika 6: a)ROC krivulja v odvisnosti od deleža uporabljenih podatkov za treniranje, število dreves: 32, globina: 3. b)ROC krivulja v odvisnosti od globine, ST označuje število dreves.

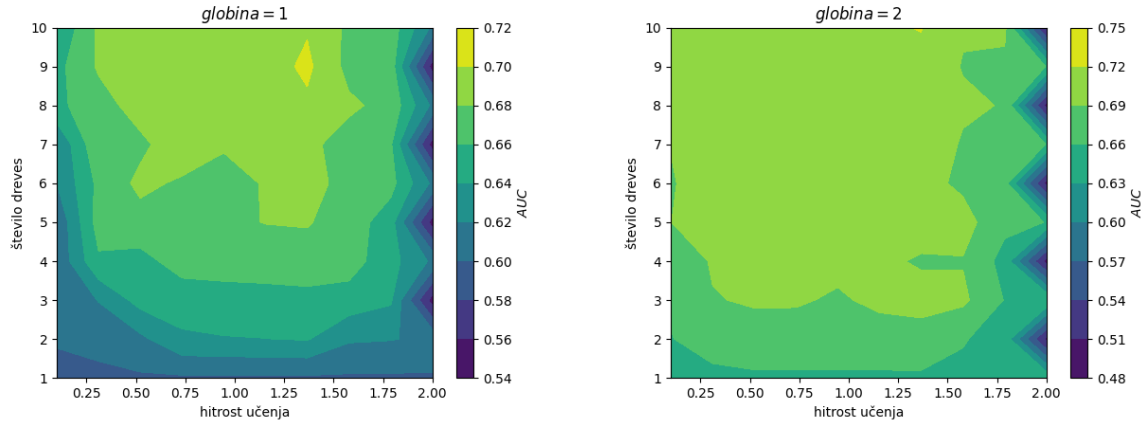
Tudi pri DBT lahko pretirano natreniramo algoritem. Če določimo preveliko število dreves in s tem preveč razdrobimo podmnožice bo prišlo do 'over-fitanja' kar prikazuje slika 7. Minimum 'loss' funkcije, v tem primeru MSE označuje rdeča pika. Težavo se da odpraviti na več načinov. Eden izmed njih je ta, da DBT določimo veliko število dreves in potem poiščemo pri kateri vrednosti doseže 'loss' funkcija minimum. Tako uporabimo optimalno število dreves za dani problem. Odvisnost AUC od globine in števila dreves prikazuje slika 7b. Pričakovano se vidi, da se z večanjem globine in števila dreves algoritem izboljšuje.



(a) (b)

Slika 7: a)MSE v odvisnosti od števila dreves. Rdeča pika označuje minimum, globina: 3 b)AUC v odvisnosti od globine in števila dreves.

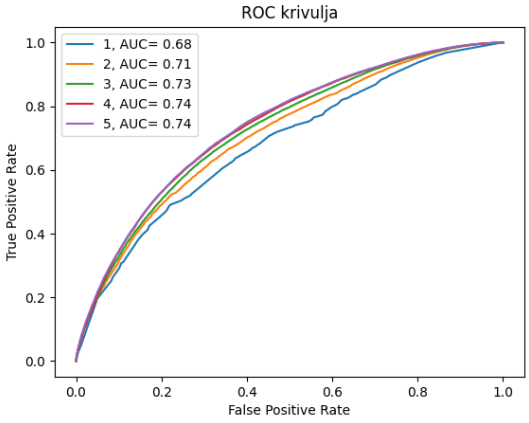
Zanimivo je pogledati tudi odvisnost AUC od hitrosti učenja in števila dreves, pri dani globini (slika 8). Privzeta hitrost učenja algoritma AdaBoost je 0.1. Pri globini 1 očitno privzeta hitrost ni optimalna. Paziti je potrebno tudi na zgornjo mejo hitrosti, saj lahko pri prevelikih korakih preskočimo minimum 'loss' funkcije.



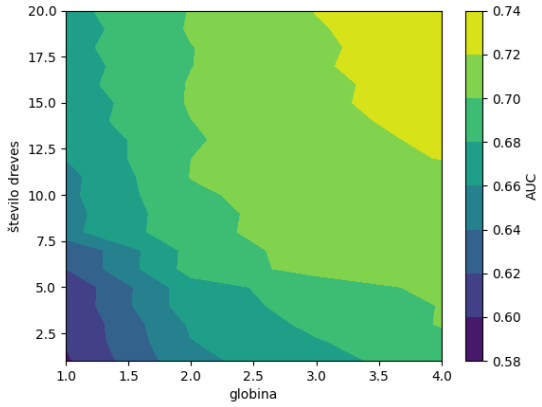
(a) (b)

Slika 8: a)AUC v odvisnosti od števila dreves in hitrosti učenja pri globini 1.b) AUC v odvisnosti od števila dreves in hitrosti učenja pri globini 2

Algoritem Gradient Boosting je podoben kot algoritem AdaBoost. Za razliko od AdaBoost, ki po vsaki iteraciji popravi uteži, Gradient Boosting algoritem poskuša 'fitati' nove uteži k ostankom napak, ki jih podajo prejšnje uteži. Pri implementaciji sem uporabil Scikit-Learn-ov algoritem 'GradientBoostingClassifier'. Rezultati so zelo podobni kot pri AdaBoost algoritmu (slika 9, 10).

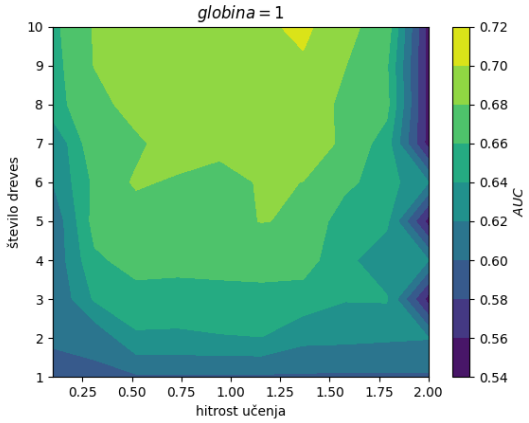


(a)



(b)

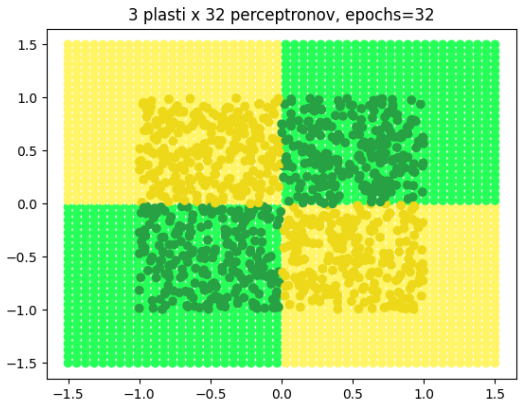
Slika 9: a)ROC krivulja pri različnih globinah, število dreves: 32 b)AUC v odvisnosti od števila dreves in globine.



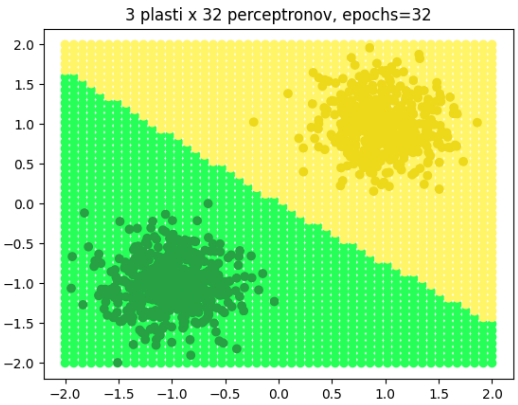
Slika 10: AUC v odvisnosti od hitrosti učenja in števila dreves.

3.3 Dodatna naloga

Na slikah so narisani testni podatki in predvideno ozadje.

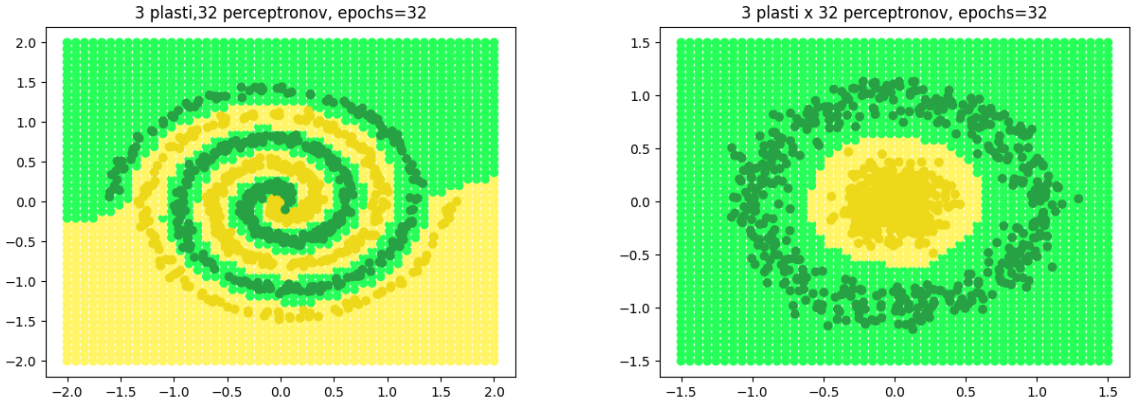


(a)



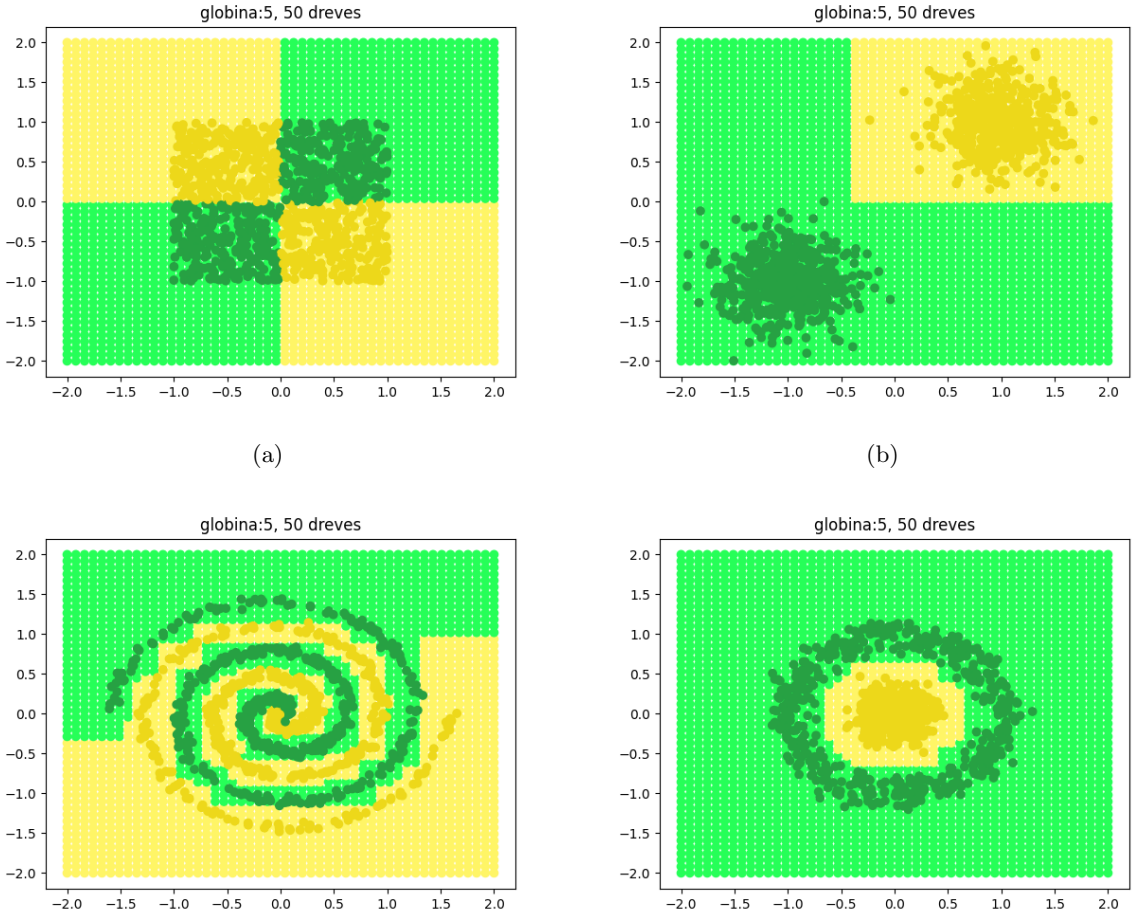
(b)

Slika 11: Implementacija distribucij z DNN.



(a) (b)

Slika 12: Implementacija distribucij z DNN.



(a) (b)

(c) (d)

Slika 13: Implementacija distribucij z AdaBoost.

4 Zaključek

Strojno učenje je lahko uporabno.