Poročilo za 1. Domačo nalogo pri predmetu NSU

Matija Kerkoč (27192017)

28. marec, 2021

1 Uvod

Za domačo nalogo sem izbral drugo možnost, torej AutoML.

Nalogo sem se odločil reševati v `Jupyter` notebook-u, saj ponuja poganjanje posameznih celic kar je zelo priročno pri delu s podatki. Na primer, podatke lahko naložimo le enkrat in potem kličemo le kar potrebujemo.

Opomba. Pri uporabi metode `seaborn.distplot()` dobimo opozorila, da je metoda zastarela in naj raje uporabljamo druge metode za izris grafov. Ker je bil primer predstavljen s to specifično metodo in metoda še vedno deluje, sem se jo vseeno odločil uporabljati. V ta namen izklopimo opozorila, kar je v splošnem zelo nevarno početje in se ga poslužujemo le če smo prepričani kaj počnemo.

Vedno ko se ukvarjamo s podatkovnimi množicami, si je smiselno najprej pogledati, s kakšnimi podatki se v resnici ukvarjamo. Naša podatkovna množica ima 1203 primerov. Imamo 30 značilk, tipa `float`, ki so označene kot `x1, x2, ... , x30` in eno binarno ciljno spremenljivko `y`, ki zavzame možni vrednosti `"teamEdward"` ali `"teamJacob"`. Vsi podatki so popolni in nimamo `null` ali `NaN` vrednosti.

1.1 Prvi problem

Prvi "problem" s katerim se srečamo v naših podatkih je neenakomerna zastopanost obeh vrednosti v ciljni spremenljivki `y`. Problem je enostavno rešljiv. Paziti moramo le, da pravilno vzorčimo, ko bomo naše podatke delili na učno in testno množico. Pravilen pristop je *stratificirano vzorčenje* "po spremenljivki `y`".

1.2 Drugi problem

Druga stvar na katero moramo biti pozorni je pravilno skaliranje podatkov. V grobem poznamo tri različne vrste skaliranja (povzeto po [1])

1. Standardno skaliranje(`sklearn.preprocessing.StandardScaler()`).

   Standarno skaliranje je najpogostejši pristop k skaliranju podatkov, vendar je za to skaliranje nujna predpostavna, da so podatki pri posamezni značilki porazdeljeni (približno) normalno. Označimo naše podatkovje kot `dataset = [X : y]`. Če posamezen podatek pri neki značilki označimo kot `z` $\in$ `X["znacilka"]`. Potem skaliranje deluje po principu

   $$\texttt{z}' = \frac{\texttt{z} - \mu}{\sigma},$$

   kjer sta μ in σ povprečje in standardni odklon posamezne značilke. Ker našo podatki očitno niso porazdeljeni normalno, ne moremo uporabiti te metode.

2. Normalizacijo (`sklearn.preprocessing.Normalizer()`),

   Normalizacija je način, kako podatke, ki niso porazdeljeni normalno spravimo v normalno porazdelitev. Vendar se je potrebno vprašati, ali sploh želimo podatke, ki bodo porazdeljeni normalno? V našem primeru, too verjetno ne bo držalo, saj želimo ohraniti zamik podatkov (in iz tega dobiti nekaj informacij za klasifikacijo). Torej se tudi te metode ne bomo poslužili.

3. MinMax skaliranje (`sklearn.preprocessing.MinMaxScaler().transform(x)`)

   MinMax skaliranje je robustna metoda, ki deluje na veliki večini podatkov in bo primerna tudi za nas. Deluje na principu

   $$\mathtt{z}' = \frac{\mathtt{z} - \max(\mathtt{X}[\text{"}\mathtt{znacilka}\text{"}])}{\min(\mathtt{X}[\text{"}\mathtt{znacilka}\text{"}]) - \min(\mathtt{X}[\text{"}\mathtt{znacilka}\text{"}])},$$

   kjer sta $\min(\mathtt{X}[\text{"}\mathtt{znacilka}\text{"}])$ in $\max(\mathtt{X}[\text{"}\mathtt{znacilka}\text{"}])$ najmanjša in največja vrednost posamezne značilke (torej najmanjša in največja vrednost v posameznem stolpcu).

   Ta metoda je primerna, ko so podatki močno zamaknjeni. V primeru, ko je za posamezno značilko standardni odklon majhen, je standarno skaliranje neprimerno (očitno, zakaj je primer 1. neprimeren, če imamo σ majhen). V večini primerov je MinMax skaliranje zelo robustno in tudi v našem primeru bomo izbrali to skaliranje

2 Preiskovani prostor konfiguracij

Izbrali smo štiri algoritme za klasifikacijo: **metodo najbližjih sosedov (kNN)**, **naključne gozdove (RF)**, **metodo podpornih vektorjev (SVM)** in **logistično regresijo (LR)**. Od teh treh je najzahtevnejša SVM, saj moramo pri izbiri jedra paziti, ker nimajo vsa jedra enakih parametrov. Torej moramo previdno gnezditi tudi notranje parametre pri nekaterih algoritmih. Dodatno imamo na voljo **vrečenje dreves (BT)**, ki ga ocenjujemo posebej in služi primerjavi z ostalimi algoritmi.

Opomba. Pomembno je, da je število poskusov v iskanju najboljše konfiguracije *manjše* od vseh možnih kombinacij konfiguracij. V nasprotnem primeru je uporaba metode `Hyperopt` v resnici nesmiselna (in celo neprimerna), saj se lahko zgodi, da gremo čez nekatere konfiguracije večkrat, nekatere pa zgrešimo. V primeru, ko imamo [# vseh možnih konfiguracij] $\approx$ [# izračunov], je torej smiselneje uporabljati Grid Search. V našem primeru tega problema ne bomo imeli, saj bo število poskusov majhno (cca. 300) v primerjavi s preiskovanimi prostori konfiguracij (več 1000).

Poglejmo si prostor preiskovanih konfiguracij za vsakega posebej. V končni fazi bo vse te primere pokril en skupni `gnezdeni_prostor`, vendar je za natančen opis preiskovanega prostora pregledneje, če opišemo vsakega posebej. Pri parametrih, ki zavzamejo vrednosti tipa `int` ali `bool` bomo zraven zapisali tudi koliko možnih izbir parametrov imamo, da si znamo predstavljati velikost prostorov.

2.1 Metoda najbližjih sosedov (k-Nearest Neighbours classifier)

Za algoritem kNN definiramo `prostor_knn`, ki je sestavljen iz

- `n_neighbours` (število sosedov): `{1, 2, ..., 20}` [20 možnosti],
- `weights` (uteži, ki jih predpišemo): `{"uniform", "distance"}` [2 možnosti].

2.2 Naključni gozd (Random Forest classifier)

Za algoritem RF definiramo `prostor_gozd`, ki je sestavljen iz

- `n_estimators` (število označevalcev): `{1,2, ..., 50}` [50 možnosti],
- `max_depth` (največja globina drevesa): `{1, 2, ..., 20}` [20 možnosti],
- `criterion` (odločitveni kriterij): `{"gini", "entropy"}` [2 možnosti].

2.3 Metoda podpornih vektorjev (Support Vector Machine)

Za algoritem SVM definiramo `prostor_svm`, ki je sestavljen iz

- `C` (vrednost C): `lognormal {0, 1}`
- `kernel` (jedro): `{linear, rfb, polynomial}`
  - `linear` (linearno jedro): `/` ,
  - `rfb` (rfb jedro): `gamma` (gama): `lognormal {0, 1}`,
  - `poly` (polinomsko jedro): `degree` (stopnja polinoma): `{1, 2, 3, 4, 5}`

2.4 Logistična Regresija (Logistic Regression)

Za algoritem LR definiramo `prostor_logreg`, ki je sestavljen iz

- `solver` (način reševanja): `{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}` [5 možnosti],
- `C1` (parameter C): `lognormal {0, 1}`,
- `max_iter` (maksimalno število iteracij): `{50, 100, 150}` [3 možnosti].

2.5 Gnezdeni prostor

Iz teh prostorov konstruiramo `gnezdeni_prostor`, ki ga lahko zapišemo kot

`gnezdeni_prostor = {prostor_knn} ∪ {prostor_gozd} ∪ {prostor_svm} ∪ {prostor_logreg}.`

Natančnejša konstrukcija je podana v samem Jupyter notebooku.

2.6 Vrečenje dreves (Bagging Tree classifier)

Prostor za vrečenje zgradimo posebej od ostalih, da ohranjamo razliko med obema prostoroma. Naš `prostor_vrecenje` izgleda torej tako

- `n_estimators` (število ocenjevalcev): `{1,2, ..., 50}` [50 možnosti],
- `max_samples` (največje možno število primerov): `{1,2, ..., 50}` [50 možnosti],
- `max_features` (največje možno število značilk): `{1,2, ..., 20}` [20 možnosti].

Skupaj imamo pri vrečenju dreves 75000 možnih konfiguracij parametrov.

3 Grafi porazdelitev klasifikacijskih točnosti

Zmagovalec med našimi algoritmi je naključni gozd (RF).

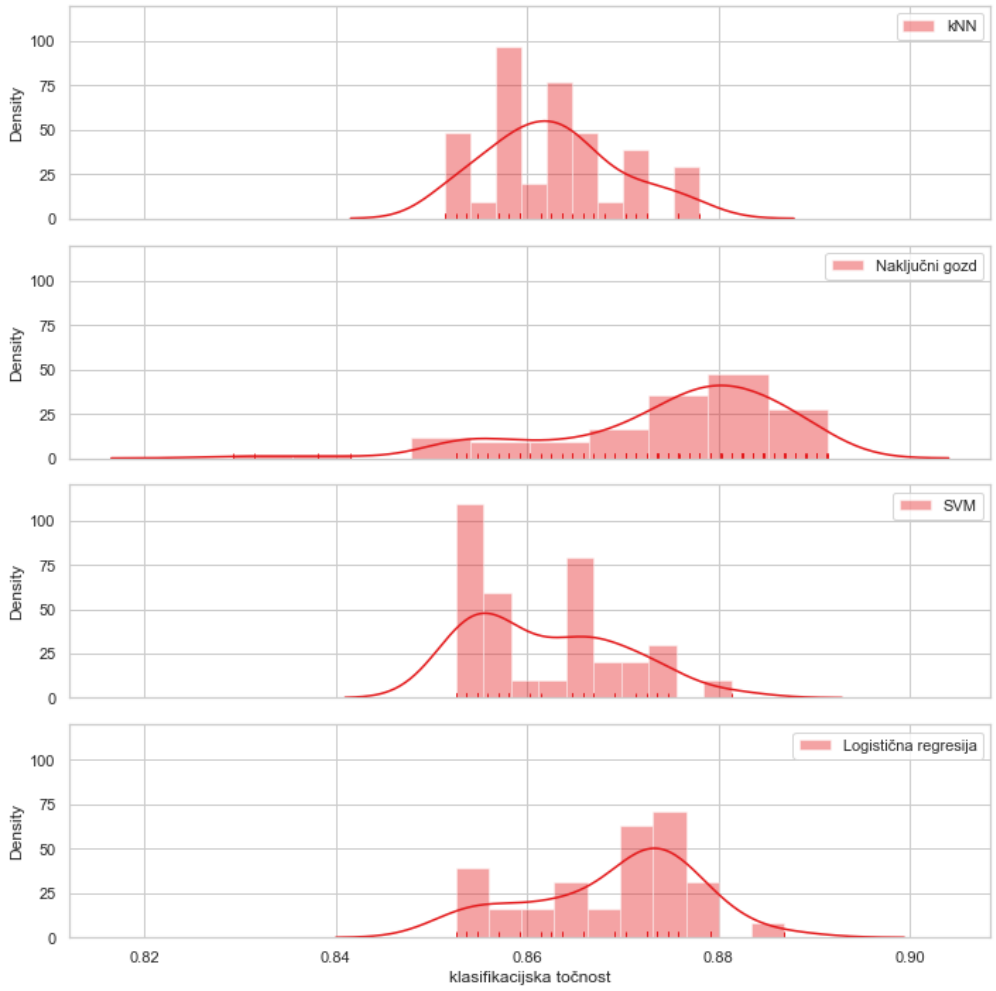
3.1 Ostali algoritmi

Najboljša konfiguracija za naključni gozd je

`{'criterion': 'gini', 'max_depth': 11, 'n_estimators': 36}`,

ki na `podatki.csv` doseže klasifikacijsko točnost 0.9003322259136213. Graf porazdelitev točnosti pa je predstavljen na sliki 1.

Vidimo, da se na naših podatkih najboljše odreže naključni gozd, vendar pa opazimo, da so klasifikacijske točnosti tudi najbolj razpotegnjene. RF torej doseže nekatere izmed najboljših, vendar pa tudi nekatere izmed najslabših klasifikacijskih točnosti. Sledi logistična regresija, ki je nekoliko manj razpotegnjena, nato metoda



Slika 1: Grafi porazdelitev klasifikacijskih točnosti za štiri algoritme v našem gnezdenem prostoru. Vidimo, da najbolše rezultate dosega naključni gozd (RF).

podpornih vektorjev, ki ima približno enak odklon rezultatov kot LR. Najslabše se je odrezala metoda k najbližjih sosedov, vendar pa ima med vsemi metodami najmanjši odklon porazdelitve.

Večje odklone v točnostih pri RF gre pripisati tudi temu, da ko iščemo najboljšo konfiguracijo z metodo `Trials()`, skoraj polovica vseh preiskušenih konfiguracij pripada metodi RF. To si lahko interpretiramo na naslednji način. `Hyperopt` predvideva, da bo RF najboljši algoritem zato ga začne "forsirati" v ospredje in preizkušati vedno več njegovih konfiguracij. Vendar pa so lahko nekatere izmed njih tudi precej napačne. Zato lahko dobimo pri RF tudi precej slabše rezultate. Vendar ker nas zanima le najbolši algoritem in njegova konfiguracija, se s tem ne obremenjujemo.

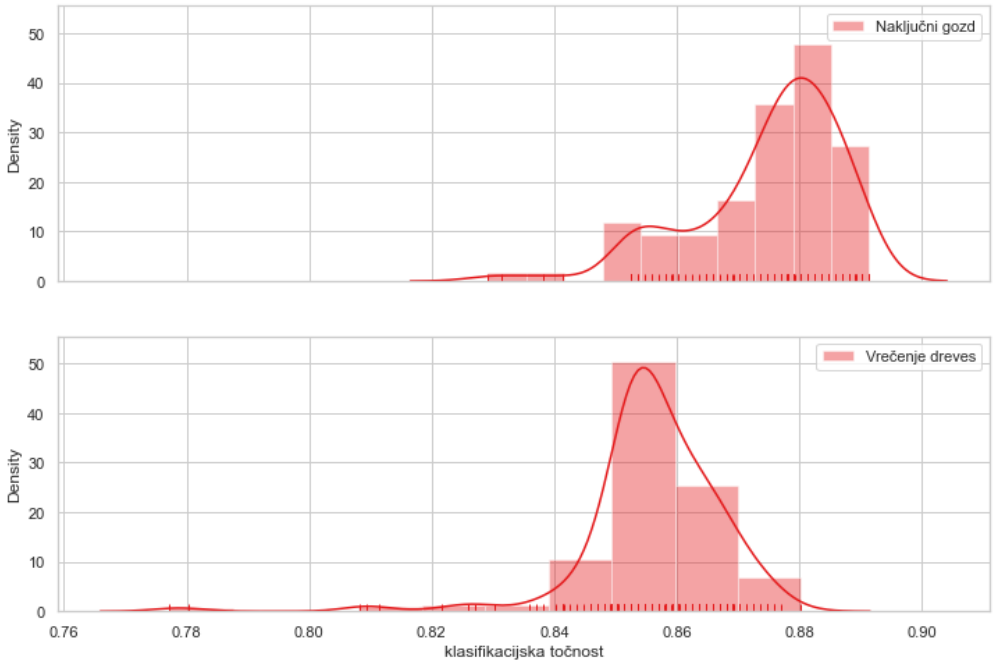
Vsi algoritmi dosežejo klasifikacijsko točnost nekje v razponu $85\% - 90\%$. Predvidevam, da bi z nekoliko bolj preciznim izborom parametrov in njihovih možnih vrednosti z lahkoto dosegli točnost nad 90%.

3.2 Vrečenje

```
{'max_features': 24, 'max_samples': 49, 'n_estimators': 29},
```

ki na `podatki.csv` doseže klasifikacijsko točnost 0.8837209302325582.

Poglejmo si sedaj primerjavo s naključnim gozdom. V ta namen izrišemo dva grafa porazdelitev klasifikacijskih točnosti.



Slika 2: Grafa porazdelitve klasifikacijskih točnosti za izbran naključni gozd in Vrečenje. Opazimo, da so klasifikacijske točnosti vrečenja veliko bolj skoncentrirane.

Naključni gozd je boljši tudi od vrečenja, kar bi na nek način lahko pričakovali. Namreč, če imamo izbiro med štirimi algoritmi, je smiselno pričakovati, da bo med njimi vsaj en, ki bo premagal vnaprej že določen algoritem (vrečenje). Imata pa oba algoritma podobne parametre, saj oba bazirata na algoritmu odločitvenih dreves (Decision Tree classifier).

4 Zaključek

Primerjali smo množico izbranih algoritmov z vrečenjem dreves. Ker smo v naši množici izbrali tudi algoritem naključnih gozdov, ni nič čudnega, da imata omenjeni algoritem in vrečenje dreves podobne rezultate na `podatki.csv`. Dodatno je v naši množici algoritmov naključni gozd najboljši. Ob bolj natančni izbiri parametrov, bi se lahko zgodilo, da bi zmagal kakšen drug algoritem. Vendar preverjanje vseh parametrov ni bil smisel te naloge, saj imajo posamezni algoritmi tudi po 20 in več parametrov med katerimi so možni vsi tipi `int, float, bool, ...`. Iskanje na prostoru vseh možnih parametrov bi bilo zamudno in nesmiselno, saj nekateri parametri vplivajo bolj, nekateri pa manj na klasifikacijsko točnost.

Poglejmo si še najboljše ter povprečne klasifikacijske točnosti vsakega posameznega algoritma.

Ime algoritma	Najboljša točnost	Povprečna točnost
kNN	0.8780417434008594	0.8627142654766965
RF	0.8913812154696131	0.8749969955924529
SVM	0.8813627992633517	0.8619819345786196
LR	0.8869122160834868	0.869145222570637
BT	00.8802885205647637	0.8560555964804584

Table 1: Klasifikacijske točnosti vseh algoritmov. Pozor! To so klasifikacijske točnosti na učnem delu podatkov. Če želimo pravilno evaluirati naša najboljša algoritma ju moramo najprej naučiti na učni množici in nato evaluirati na testni množici (tako kot smo to naredili v podpoglavjih 3.1 in 3.2).

4.1 Izboljšave

Za konec si poglejmo še nekaj idej za izboljšavo.

1. Smiselno bi bilo preiskati celotne prostore parametrov za posamezen algoritem. V večini primerov imajo namreč algoritmi v `sklearn` knjižnjici približno 20 parametrov za katere ne moremo vnaprej vedeti, kateri bodo najbolj vplivali na klasifikacijsko točnost. V našem primeru smo variirali tistih nekaj, ki jih najbolj poznamo, čisto mogoče pa je, da smo zgrešili kakšnega, ki bi občutno povišal natančnost algoritma.

2. Smiselno bi bilo tudi bolj razlikovati med tipi parametrov. Na primer, če imamo parameter, ki je tipa `int`, le ta lahko zavzame le n podanih vrednosti. V primeru, da imamo parameter tipa `float` pa imamo veliko gostejši prostor možnih vrednosti, ki jih lahko parameter zasede. Tak je na primer parameter C v SVM klasifikatorju, ki lahko zasede katerokoli vrednost v $\mathbb{R}^+$.

References

[1] Odgovor uporabnika Irmak Sirer:
`https://stackoverflow.com/questions/30918781/right-function-for-normalizing-input-of-sklearn-svm`