Formális és program nyelvek elemzése gépi tanulási modellekkel

Sisák Sándor

Bevezetés

Az önálló projekt célja gépi tanuláson alapuló módszerek vizsgálata Python kódok automatikus értelmezésére. A gépi tanulás hatékony eszköznek bizonyult természetes nyelvi feladatok megoldására, ezért hamar felmerült, hogy programkódokon is alkalmazzák. A féléves munkában a Stack Overflow-ról az előző félévben kinyert, felhasználók által írt válaszokban található Python kódokon végeztem többcímkés klasszifikációt.

Adathalmaz

Az adatok a Kaggle *Python Questions from Stack Overflow* adathalmazából lettek előállítva [8]. A Stack Overflow fórumon programozással kapcsolatos kérdéseket tesznek föl a felhasználók, más felhasználók pedig válaszokat küldhetnek a feltett kérdésre. Az adathalmaz 607000 Python programozással kapcsolatos kérdést és ezekre adott 987000 választ tartalmaz. Minden kérdéshez tartoznak a felhasználók által meghatározott címkék (pl. *python-2.7*, *python-3.x*), amelyek a keresést segítik.

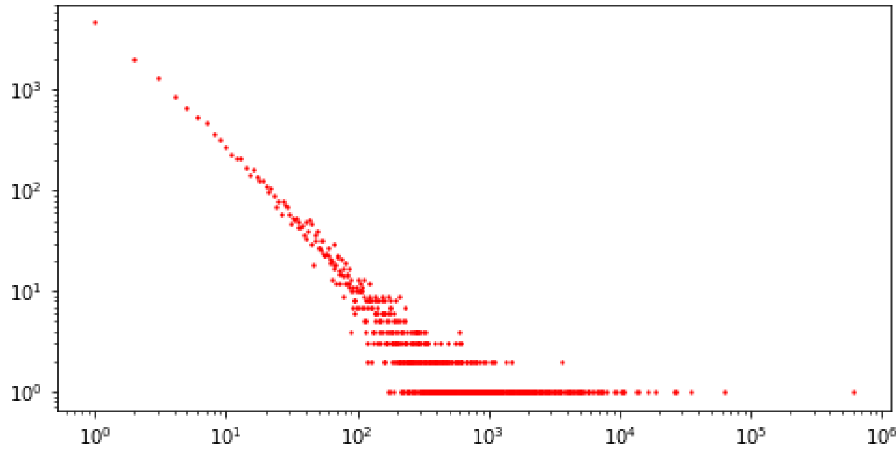
Az előző félév során előállítottam egy módosított adathalmazt, amelyben a válaszokból kinyert Python kód és a válaszhoz tartozó kérdés címkéi szerepeltek. A módosított adathalmazon neurális hálót tanítottam arra, hogy kódokhoz címkéket rendeljen.

Adatszűrés

Az adatok között sok kódrészlet túl rövid volt, például sokszor egy-egy függvény nevét is önálló kódcellaként formázták Stackoverflow-n. Mivel az ilyen kódrészletek nem szolgálnak elég kontextussal, a modell nem tud tanulni belőlük vagy prediktálni rajtuk, tehát a kódokat hossz szerint is szűrni kell. A karaktersorozat hossza helyett azonban a tokenizált kódrészlet hosszát vettem figyelembe: csak azok a kódrészletek kerültek az adathalmazba, amelyek tokenizálva legalább 10 elem hosszúságúak voltak. A tokenek száma jobban megragadja a kódrészletek tényleges információtartalmát, mint a karakterek száma. A CodeBERTa-small modell tokenizációját használtam, ezt a *modell* fejezetben tárgyalom. Előfordult, hogy egy válaszban több kódrészlet is volt, ekkor az adathalmazba mindig a leghosszabb került.

Az előző féléves munkámmal ellentétben az adatok szűrésén szigorítottam: a többcímkés klasszifikációt csak a 32 leggyakoribb címkére végeztem, ezek mindegyike legalább 5000 adatpontnál szerepelt. Az adathalmazban a címkék hatványeloszlást mutatnak. A 32 leggyakoribb címke sűrűsége között is lényeges különbségek vannak, és az egyes címkék csak az adathalmaz kis hányadán jelentek meg. A szűrt adathalmaz 348000 adatpontot tartalmaz, ennek tizede a validációs halmaz, tizede teszthalmaz, a maradék pedig tanítóadat.

A tanítóadat kiegyensúlyozása



1. ábra. A címkék gyakorisága (a szűretlen adathalmazon). (x, y) pont azt jelöli, hogy y különböző címke van az adathalmazban, amely pontosan x különböző kérdésnél szerepel.

Ismert, hogy a ferde adathalmazon tanított modellek gyengébben teljesítenek, mint a kiegyensúlyozott adathalmazon tanítottak, ezért oversampling vagy undersampling segítségével módosítani kell az adathalmazt, hogy minden címke közel azonos arányban szerepeljen. Mivel a rendelkezésre álló adatban kevés volt a példák száma a feladat összetettségéhez képest, az oversampling mellett döntöttem. Ez többcímkés klasszifikáció esetében nem triviális feladat, hiszen ha az adat egyes címkékhez tartozó részhalmazai nem diszjunktak, az oversampling során más címkék gyakorisága is megváltozik.

Az adatok kiegyensúlyozására két módszert alkalmaztam, a naív megközelítést és a konvex optimalizálást. A naív megközelítés a klasszikus oversampling. Minden i címkére előállítom az adatoknak azt a H_i részhalmazát, amely hordozza az i címkét. Legyen

$$N = max\{|H_j| : j \text{ címke}\}$$

$$\forall i \text{ címkére} : k_i = \left\lfloor \frac{N - |H_i|}{|H_i|} \right\rfloor, \quad p_i = \frac{N - (k_i + 1)|H_i|}{|H_i|}$$

Minden H_i részhalmazban k_i-szer duplikálom az adatpontokat. Ezt követően még egyszer iterálok H_i-n, minden adatpontot p_i valószínűséggel duplikálva. Ha H_i halmazok diszjunktak lennének, az eljárás végére minden címke várhatóan ugyanannyiszor szerepelne.

A másik megközelítésben az adathalmazt konvex optimalizálás segítségével egyensúlyozom ki. Minden adatponthoz felveszek egy x_i változót, amely azt jelöli, hogy hányszor kell duplikálni az i-edik adatpontot. Természetesen jobb eredményt kapnánk, ha a feladatot eleve egészértékű problémaként oldanánk meg, ez azonban túl nagy számítási kapacitást igényel. Az x_i változókra x vektorként tekinthetünk. Megoldjuk a következő λ paraméterű feladatot:

$$x \geq \mathbf{1}$$

$$\mathbf{1} \cdot |Mx - \mathbf{N}| + \lambda \cdot \max(x) \to \min$$

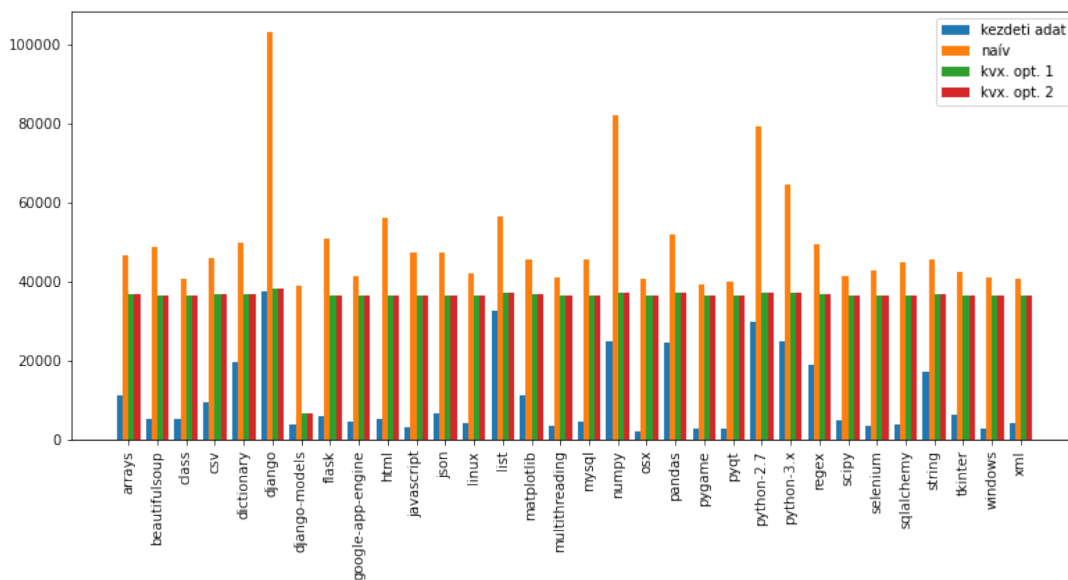
Ahol $M \in \mathbf{R}^{32 \times |D_{\text{train}}|}$. $M(i, j) = 1$ ha a j. adatponton szerepel az i címke, $M(i, j) = 0$ egyébként. Továbbá $\mathbf{N}$ a csupa N vektor, $\mathbf{1}$ pedig a csupa 1 vektor. (Nyilván mindenhol olyan hosszal, hogy a műveletek értelmezve legyenek.) Az x^* optimumot egészekre kerekítjük, majd a tanítóadat minden elemét annyiszor duplikáljuk, amennyi az így kapott $x^*_{\mathbf{N}}$ vektor neki megfelelő eleme. λ regularizációs paraméter, aminek növelésével el tudjuk kerülni, hogy a kapott adathalmazban kis számú példa sokszázszor szerepeljen. Ez azért nem kívánatos, mert ekkor a modell túltanul ezeken a példákon, és nem lesz képes általánosítani. Végül a $\lambda = 1000$ és $\lambda = 10000$ paraméter melletti megoldásokat használtam. $\lambda = 1000$ esetén egy-egy adatpont legfeljebb 18-szor szerepelt a módosított adathalmazban, $\lambda = 10000$ esetén legfeljebb 12-szer. Az optimalizálási feladat megoldására a CVXOPT Python csomag *GNU Linear Programming Kit* (GLPK) solverét használtam.

Az eredeti tanítóadat 278000, a naívan kiegyensúlyozott 1130000, a konvex optimalizálással kiegyensúlyozottak pedig rendre 836000 és 850000 adatpontot tartalmaztak.

Modell

A modell egy CodeBERTa-small encoderből [1] (valamint az ehhez tartozó BPE tokenizációból) és egy fejből állt. A BPE, vagyis byte pair encoding során iteratívan alakítják ki a tokenizáció szótárát: kezdetben csak önálló karakterek vannak benne, egy nagy méretű korpusz segítségével unsupervised módon építik fel a szótár maradékát. Az iteráció minden lépésében megkeresik a leggyakoribb olyan karaktersorozatot, ami előáll a pillanatnyi szótár két szavának konkatenációjaként, és hozzáadják a szótárhoz. Ha a szótár mérete elér egy előre rögzített paramétert, az algoritmus leáll és a tokenizációt végző szótárat véglegesítik.

A CodeBERTa-small egy RoBERTához hasonló modell [6], de mivel csak 6 transzformer modult [9] használ, kevesebb paramétere van és kisebb a számításigénye. A RoBERTa architektúrája megegyezik a BERT-ével [2], de a két modellt tanítása nem



2. ábra. Az egyes címkék előfordulásainak száma az eredeti, a naívan kiegyensúlyozott és a konvex optimalizálással kiegyensúlyozott adaton.

ugyanolyan módszerekkel zajlott. A CodeBERTát a CodeSearchNet adathalmazon tanították [3], tehát természetes nyelvi szövegek mellett forráskódok beágyazását is tanulta, így informatívabb reprezentációkat állít elő a vizsgált Python kódokhoz, mint az előző félévben használt RoBERTa.

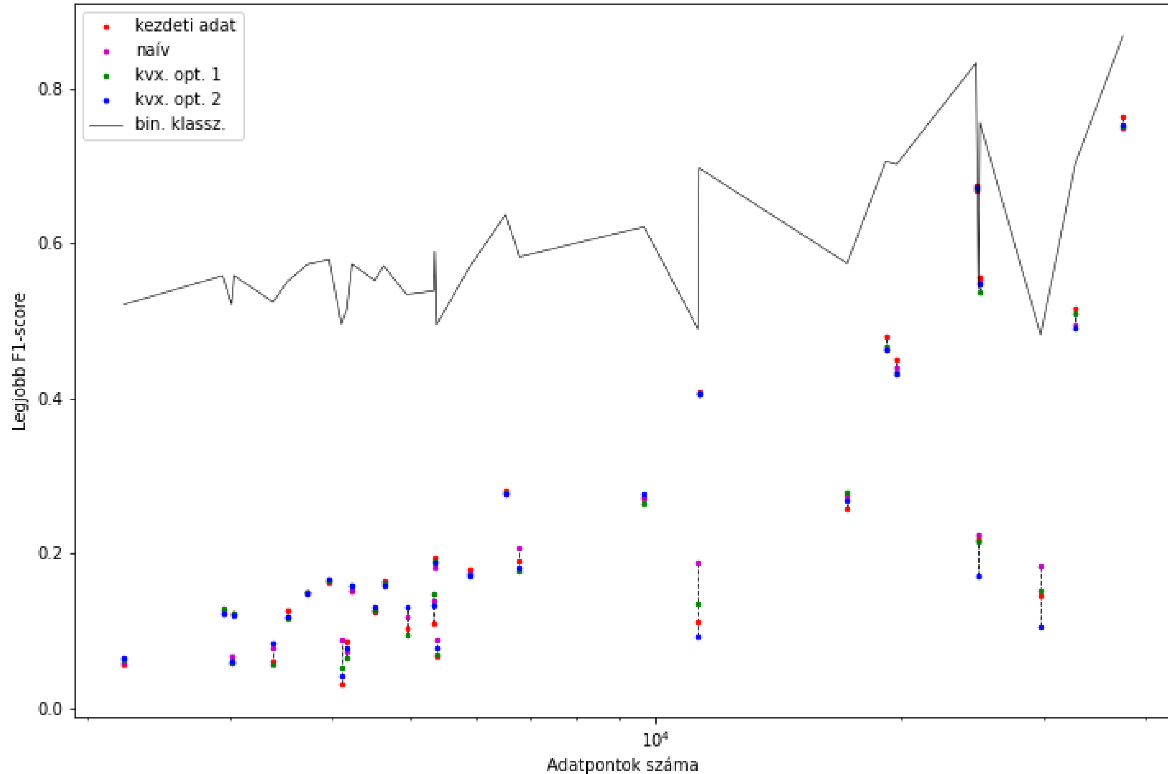
A fej egy dropout réteg és egy sűrű réteg, ami a CodeBERTa-small által előállított reprezentációkat klasszifikálja. A dropout rétegre azért van szükség, mert az oversampling miatt regularizáció nélkül sok adaton túltanulna a fej.

A veszteségfüggvény a valószínűségek és a címkéket leíró $\{0,1\}^{32}$-beli vektorok közötti bináris keresztentrópia volt. A modell paramétereit AdamW algoritmussal [7] optimalizáltam. Az AdamW az Adam algoritmus [4] egy változata, ami, lehetőséget ad weight decay használatára [5]. Az AdamW-t a learning rate kivételével a Pytorch alapértelmezett paramétereivel hívtam meg: learning rate $= 5 \cdot 10^{-5}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, weight decay $= 0.01$. Az előző féléves munkámmal ellentétben a reprezentációt előállító transzformer paraméterei nem voltak rögzítve, tehát a reprezentációt is a megoldandó feladatra hangoltam. A modell pontosságát a validációs halmazon F1-scoreban mértem, amit címkénként külön számoltam. A batchméret minden kísérlet során 16 volt. A modellt a Pytorch csomag segítségével implementáltam.

Kísérletek

A modellt öt epochon keresztül tanítottam a különböző adathalmazokon, minden epoch végén elmentve a modell pillanatnyi paramétereit. A címkék többségén rosszul teljesí-

tett a modell, ami nem meglepő, hiszen várható volt hogy az adat nem elégséges ilyen komplex feladatok tanítására. Ezt mutatja be a 3. ábra: a legtöbb címke esetében a gyenge teljesítmény szorosan összefügg a kevés tanítóadattal.



3. ábra. A különböző címkéken elért legjobb F1-score a címkéhez rendelkezésre álló (kezdeti) adat függvényében. Az x tengely logaritmikusan skálázott.

A tanulást még jobban megnehezíti, hogy több címkén kell egyszerre klasszifikálni, és minden adatponton csak két-három címke szerepel. Hogy képet kapjak arról, mennyivel nehezíti meg az többcímkés klasszifikáció a tanulást, minden címkére tanítottam egy bináris klasszifikátort is. Ennek az architektúrája a klasszifikációs rétegtől eltekintve megegyezik a többcímkés modellével, és a teljesítménye segít kontextusba helyezni a többcímkés klasszifikátor eredményeit. A 32 címke 32 bináris klasszifikátorát a kezdeti adathalmazon tanítottam, mindegyiket csak egy-egy epochig. A bináris és a többcímkés klasszifikációval elért legjobb eredményeket az 2. táblázat foglalja össze.

Érdemes megjegyezni, hogy a nagy mennyiségű adat sem garantálja a jó teljesítményt. Az *arrays*, *python-2.7* és *python-3.x* címkéken gyenge eredmények születtek, annak ellenére hogy a leggyakoribb címkék között voltak. A *numpy*, *pandas* és *django* címkéken viszont a többcímkés klasszifikációval elért F1-score megközelítette a bináris klasszifikátor teljesítményét.

A legtöbb gépi tanulásos feladatnál triviális, hogy azokkal a paraméterekkel kell véglegesíteni a modellt, amelyekkel a legjobb teljesítményt mérjük a validációs halmazon.

Itt azonban azt tapasztaltam, hogy a különböző címkéken mért F1-score különböző epochban tetőzik nem csak az eredeti, de a kiegyensúlyozott tanítóadatokon is. Miután egy címkén tetőzik a teljesítmény, az ezt követő epochokban jellemzően csökken az F1-score túltanulás miatt. Az 1. táblázatban összesítem hogy egy-egy epochban hány címkének tetőzött az F1-scoreja.

	1. epoch	2. epoch	3. epoch	4. epoch	5. epoch
kezdeti	3	5	9	5	10
naív	19	4	5	2	2
kvx. opt. 1	12	5	3	7	5
kvx. opt. 2	18	3	7	2	2

1. táblázat. Egy-egy epochban hány különböző címkének tetőzött az F1-scoreja, adathalmazokra lebontva.

A legjobb F1-score értékek a várakozásaimmal ellentétben főleg a kezdeti adathalmazon születtek. Ennek az lehet az oka, hogy a többi adathalmaz méretéből adódóan egy címke túltanulhat a modell paramétereinek epochonkénti mentése között. (A kezdeti adathalmaznál három-négyszer több adatpont van a módosított adathalmazokban és sok adatpont tíznél többször szerepel bennük.) A kezdeti adathalmazon azonban a modell teljesítménye nagyon eltérő epochokban tetőzött a különböző címkéken, így nehéz egy minden címkén elfogadható eredményt nyújtó paraméterezést találni. A kezdeti adathalmaz esetében tíz különböző címkén is az ötödik epochban tetőzött az F1-score. Feltételeztem hogy ez azért van, mert ekkor még sok címkére alultanított a modell, és az F1-score tényleges tetőzése későbbi epochokban következne be. Továbbtanítva a modellt viszont azt tapasztaltam, hogy a címkék F1-scoreja inkább stagnál vagy csökken a 6. és 7. epochban.

A modellek értékelése

A validációs halmazon mért eredmények alapján a teszthalmazon már csak a legígéretesebb paraméterezéseket próbáltam ki. A naív és a konvex optimalizálással kiegyensúlyozott adathalmazon tanult modellek között ez a paraméterezés egyértelműen az 1. epoch végén mentett súlyozás volt. A kezdeti adathalmazon tanított modellnek a 3. és 5. epoch utáni súlyozását is teszteltem. Ezeknek a teszteknek az eredményeit összesíti a 3. táblázat.

A micro és macro F1-score alapján a kezdeti modell 5. epoch utáni súlyozása bizonyult a legjobbnak, holott a 3. epoch utáni súlyozás 11 különböző címkén is a legjobb teljesítményt nyújtotta. A konvex optimalizálással kiegyensúlyozott adathalmazokon jellemzően gyengébb eredmények születtek, az első ilyen modell elhanyagolható előnnyel érte el a legjobb teljesítményt 4 címkén, a második pedig csak olyan címkéken tudott jelentős javulást elérni, amelyeken 0.15 alatt volt az F1-score.

Kitekintés

Bár a tárgy nem folytatódik, sokféle módszerrel lehetne kísérletet tenni a modell teljesítményének javítására és sok kérdésem merült föl az eredményekkel kapcsolatban. Az utolsó fejezetben ezekre szeretnék kitérni.

A gyenge teljesítmény elsősorban annak tudható be, hogy kis számú és gyenge minőségű adattal dolgoztam. A forráskódot feldolgozó hálók tanításához nagy mennyiségű adat kell, összehasonlításként a modell alapjául szolgáló CodeBERTa-small a CodeSearchNet adatbázison tanult. A CodeSearchNet adatbázis 8.8 millió adatpontot számlál, ezeknek háromnegyede csak kód, a maradék *bimodális* adat, vagyis kódot és természetes nyelvet is tartalmaz. Továbbá a CodeSearchNet adatpontjai teljes függvények, szemben az általam használt kódrészletekkel, amelyeknek a kontextusa sokszor a felhasználó által írt komment természetes nyelvi részéből derül ki. Ezt igyekeztem kiküszöbölni az adatok tokenizált hossza alapján történő szűréssel.

A kísérleteim során az AdamW hiperparamétereit nem próbáltam optimalizálni. A feladatot az nehezítette meg, hogy egyes címkéken túltanultak a modellek, amíg a többi címke még alultanult. A túltanulás megfelelő regularizációval mérsékelhető, tehát elsősorban a weight decay és a dropout valószínűség a két paraméter, amire érdemes lett volna figyelmet fordítani.

Problémát jelenthet az alacsony batchméret is. A batchméret növelése nem csak azt szolgálja, hogy párhuzamos számítások segítségével gyorsabban tudjuk tanítani a modellt, de a konvergencia sebességére is hatással van. Például a RoBERTa a BERT-tel azonos architechtúrája ellenére jobban teljesít a BERT-nél, ennek egyik okául a [6] cikk szerzői a nagyobb batchméretet jelölik meg. Ők a RoBERTa tanítása során 256 méretű batchekkel dolgoztak, a saját modellem batcheiben csak 16 adatpont volt.

Nem világos az sem, hogy miért a legjobban kiegyensúlyozott adathalmazokon születtek a leggyengébb eredmények. A ritkább címkék sűrű ismétlése kiküszönölhette volna azt a problémát, hogy a különböző címkék különböző epoch végére tanulnak be.

Hivatkozások

[1] *CodeBERTa-small-v1*. 2020. URL: `https://huggingface.co/huggingface/CodeBERTa-small-v1`.

[2] Jacob Devlin és tsai. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. *arXiv preprint arXiv:1810.04805* (2018).

[3] Hamel Husain és tsai. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search”. *arXiv:1909.09436 [cs, stat]* (2019. szept.). arXiv: 1909.09436. URL: `http://arxiv.org/abs/1909.09436` (elérés dátuma 2020. 03. 12.).

[4] Diederik P Kingma és Jimmy Ba. “Adam: A method for stochastic optimization”. *arXiv preprint arXiv:1412.6980* (2014).

	kezdeti	naív	kvx. opt. 1	kvx. opt. 2	bin. klassz.
arrays	0.112	**0.188**	0.136	0.093	0.489
beautifulsoup	**0.195**	0.181	0.191	0.187	0.589
class	0.109	0.14	**0.147**	0.132	0.539
csv	0.272	0.271	0.264	**0.276**	0.621
dictionary	**0.45**	0.44	0.434	0.432	0.702
django	**0.763**	0.749	0.75	0.753	0.867
django-models	0.031	**0.088**	0.052	0.042	0.496
flask	**0.18**	0.175	0.173	0.172	0.57
google-app-engine	**0.165**	0.162	0.161	0.158	0.571
html	0.067	**0.088**	0.07	0.079	0.495
javascript	0.062	0.078	0.057	**0.085**	0.524
json	0.191	**0.207**	0.177	0.181	0.583
linux	**0.087**	0.074	0.066	0.079	0.515
list	**0.516**	0.495	0.51	0.491	0.702
matplotlib	**0.409**	0.405	0.405	0.405	0.697
multithreading	**0.126**	0.119	0.117	0.118	0.551
mysql	0.124	0.127	0.126	**0.13**	0.552
numpy	**0.557**	0.549	0.538	0.547	0.755
osx	0.056	0.059	0.064	**0.065**	0.521
pandas	**0.674**	0.668	0.671	0.673	0.832
pygame	0.127	0.122	**0.128**	0.126	0.558
pyqt	**0.123**	**0.123**	0.122	0.12	0.558
python-2.7	0.145	**0.184**	0.153	0.137	0.482
python-3.x	0.217	**0.224**	0.215	0.19	0.541
regex	**0.479**	0.462	0.467	0.462	0.706
scipy	0.103	0.119	0.095	**0.131**	0.534
selenium	**0.15**	**0.15**	0.149	0.148	0.573
sqlalchemy	0.163	0.164	0.164	**0.166**	0.579
string	0.257	0.275	**0.28**	0.269	0.574
tkinter	**0.281**	0.276	0.278	0.277	0.637
windows	0.058	**0.067**	0.059	0.061	0.521
xml	0.153	0.152	**0.159**	**0.159**	0.573

2. táblázat. Az egyes címkéken elért legjobb F1-score a validációs halmazon. A címkén legjobban teljesítő modell F1-scoreja **félkövérrel** van kiemelve. Az utolsó oszlop a címke-specifikus bináris klasszifikátor által elért F1-score.

	kezdeti (3)	kezdeti (5)	naív	kvx. opt. 1	kvx. opt. 2
arrays	0.105	0.103	**0.164**	0.114	0.058
beautifulsoup	0.178	0.18	0.185	**0.197**	0.191
class	0.091	0.109	0.148	**0.157**	0.146
csv	0.256	0.246	0.25	0.228	**0.268**
dictionary	0.431	0.436	0.432	0.406	**0.438**
django	**0.75**	0.746	0.73	0.729	0.732
django-models	0.001	0.029	**0.083**	0.018	0.042
flask	0.178	**0.189**	0.188	0.175	0.174
google-app-engine	**0.151**	0.148	0.138	0.147	0.149
html	0.056	0.06	**0.085**	0.05	0.083
javascript	0.059	0.064	0.079	0.058	**0.094**
json	0.176	0.179	**0.192**	0.159	0.16
linux	**0.083**	0.072	0.072	0.045	0.078
list	0.493	**0.512**	0.387	0.411	0.464
matplotlib	**0.38**	0.37	0.378	0.361	**0.38**
multithreading	**0.123**	0.109	0.117	0.116	0.116
mysql	0.128	0.119	**0.13**	0.121	**0.13**
numpy	**0.533**	0.523	0.484	0.487	0.479
osx	0.046	0.047	0.055	0.062	**0.065**
pandas	**0.668**	0.666	0.655	0.661	0.657
pygame	0.132	0.133	0.131	**0.137**	0.127
pyqt	**0.131**	0.125	0.13	0.13	0.126
python-2.7	0.057	**0.149**	0.043	0.018	0.024
python-3.x	0.191	**0.21**	0.159	0.087	0.104
regex	**0.493**	0.473	0.439	0.473	0.451
scipy	0.076	0.1	0.112	0.093	**0.127**
selenium	0.149	**0.151**	0.141	0.147	0.15
sqlalchemy	0.151	**0.155**	0.148	0.151	0.152
string	0.244	0.253	0.28	**0.282**	0.258
tkinter	**0.279**	0.278	0.276	0.269	0.274
windows	0.05	0.054	**0.059**	0.049	0.058
xml	**0.171**	0.17	**0.171**	0.162	0.169
Macro F1	0.219	**0.224**	0.22	0.209	0.216
Micro F1	0.347	**0.356**	0.331	0.321	0.328

3. táblázat. A különböző paraméterezések melletti eredmények a teszthalmazon. A címkéken elért legjobb F1-score **félkövérrel** van kiemelve.

[5] Anders Krogh és John Hertz. “A simple weight decay can improve generalization”. *Advances in neural information processing systems* 4 (1991).

[6] Yinhan Liu és tsai. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. *arXiv preprint arXiv:1907.11692* (2019).

[7] Ilya Loshchilov és Frank Hutter. “Decoupled weight decay regularization”. *arXiv preprint arXiv:1711.05101* (2017).

[8] *Python Questions from Stack Overflow*. 2019. URL: `https://www.kaggle.com/datasets/stackoverflow/pythonquestions`.

[9] Ashish Vaswani és tsai. “Attention is all you need”. *Advances in neural information processing systems* 30 (2017).