EÖTVÖS LORÁND TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI KAR

Bartalis Dávid

Adattömörítés szubmoduláris kiválasztással

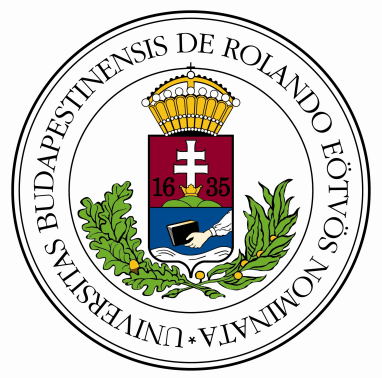
Témavezetők:

Bérczi-Kovács Erika

ELTE, Operációkutatási Tanszék

Béres Ferenc

SZTAKI, Informatikai Kutatólaboratórium



Budapest, 2021

1. Bevezetés

Önálló projektmunkám során témavezetőimmel közösen azt tanulmányoztuk, hogy a szubmoduláris függvények milyen szerepet játszanak a gépi tanulási folyamatokban. A munkát a tanítási folyamatok felgyorsítására hegyeztük ki, ehhez az Apricot (Pythonban implementált) csomagot vizsgáltuk, használtuk. Célunk az volt, hogy egy átfogó képet kapjunk arról, hogy a szubmoduláris függvényeket hogyan lehet használni ezen feladatra. Az első félévben főleg azzal foglalkoztunk, hogy a tanítási halmaz sorait hogyan lehet redukálni úgy, hogy a redukált adathalmaz reprezentatív legyen az egész adatot tekintve. Előző félévben már megjelent az az ötlet, hogy ne a sorok, hanem az oszlopok számát próbáljuk csökkenteni szubmoduláris maximalizálási technikákkal, más szóval egy dimenzióredukciós feladatot oldjunk meg. Mivel ezzel jobb eredményeket sikerült elérni, így ebben a félévben is inkább ezzel foglalkoztunk, próbáltuk finomítani az eljárást.
A félévi munka két fő részből állt össze. Az egyik az elméleti anyag rendszerezése; olyan cikkek keresése, feldolgozása volt, amik valamilyen szubmoduláris megközelítést használnak a dimenziócsökkentéses feladatokra. A másik fontos feladatunknak pedig a múlt félévben elért eredmény finomítását tekintettük. Az előző félévben használt függvények lehetséges paraméterezései közül próbáltuk meg kiválasztani a legjobbakat az adott feladatokhoz, illetve az előző félévben használt módszerek futásidejének mérésével, vizsgálatával foglalkoztunk. Az Apricot csomagban implementált szubmoduláris maximalizálást megvalósító algoritmusokat versenyeztettük performanciát és futási időt tekintve is. Emellett egy lineáris algebrai módszerrel is összehasonlítottuk az Apricot eljárást. Azt tapasztaltuk, hogy a teljesítményt tekitve a szubmoduláris megközelítés bizonyult jobbnak, futásidőt nézve viszont nem. Ez arra vezethető vissza, hogy a performanciát optimalizáló paraméterezését használtuk az Apricotban implementált függvénynek, nem pedig a leggyorsabb módszert.
Beszámolómat az elméleti anyag rövid áttekintésével kezdem, majd az idei félévben elért eredményekről, mérésekről fogok írni.

2. Elméleti anyag rövid áttekintése

2.1. Szubmodularitás - Apricot

2.1.1. Definíció (Szubmoduláris függvények)**.** *Egy $\mathcal{F} : 2^V \to \mathbb{R}$ V alaphalmaz részhalmazain értelmezett halmazfüggvényt pontosan akkor nevezünk szubmodulárisnak, vagy teljesen szubmodulárisnak, ha $\forall B \subseteq$*

$A \subseteq V$ halmazokra és $x \in \overline{\mathrm{A}}$ esetén igaz, hogy

$$\mathcal{F}(A \cup x) - \mathcal{F}(A) \leq \mathcal{F}(B \cup x) - \mathcal{F}(B)$$

A beszámolóban szereplő leglényegesebb fogalom a szubmodularitás, melyet nem csak a fenti módon lehet definiálni, viszont ez a definíció mutatja szemléletesen azt, hogy miért is használható ez a fogalom a tanítási halmazok redukciója során.
Az Apricot csomagban különböző szubmoduláris függvények vannak implementálva [2]. A teljesség igénye nélkül szeretnék 3 példát mutatni:

- **Feature-based** / Tulajdonság alapú függvény:

$$\mathcal{F}(X) = \sum_{d=1}^{D} w_d \phi \left(\sum_{x \in X} m_d(x) \right)$$

- **Facility location** / Szolgáltató elhelyezési függvény:

$$\mathcal{F}(X) = \sum_{v \in V} \max_{x \in X} \delta(x, v),$$

- **Max Coverage** / Maximális fedés függvény:

$$\mathcal{F}(X) = \sum_{i=1}^{d} \left(\left(\sum_{x \in X} x_i \right) > 0 \right)$$

Többnyire ezen függvényekkel foglalkoztam a 3 félév során, tapasztalatom és a szakirodalom szerint is igaz az, hogy az Apricotban implementált függvények közül általában ezekkel lehet a legjobb eredményeket elérni. Könnyű belátni, hogy ezen függvények mindegyike teljesíti a csökkenő hasznok elvét, erről részletesebben az első félévi beszámolómban értekeztem.
Az Apricot csomag lehetőséget biztosít arra, hogy ha elég redundáns a tanítási adathalmaz, akkor annak megkapjuk egy olyan redukált változatát, melyen tanítva a modellt nagyságrendileg ugyanolyan eredményt érünk el, mintha a teljes tanítási adathalmazon tanítottuk volna. Ezzel a módszerrel bizonyos esetekben igen jól tudjuk csökkenteni a futási időt, illetve az előző félévekben azt tapasztaltuk, hogy előfordul, hogy a módszer segítségével a túltanulás is elkerülhető.

Az Apricot csomagot használva nem csak az alkalmazni kívánt függvényt, hanem magát az algoritmust is megválaszthatjuk. Az implementált algoritmusok: naive greedy, lazy greedy, two-stage greedy, approximate lazy greedy, stochastic greedy, sample greedy, GreeDi, modular greedy, bidirectional greedy [3]. Kutatómunkám során főként az első 6 mohó algoritmussal foglalkoztam, ezért őket szeretném bemutatni.
A **naive greedy** algoritmus a legegyszerűbb megközelítés a szubmoduláris függvények optimalizálására. Minden iterációban sorra veszi a még nem kiválasztott összes elemet és kiszámolja, hogy melyiknek mennyi a hozzáadott értéke. A legnagyobb hozzáadott értékű elemet fogja kiválasztani.
Ettől csupán az implementációjában különbözik a **lazy greedy** algoritmus. Itt egy minimum prioritású sorban tároljuk a még nem választott elemeket a legutoljára kiszámolt hozzáadott érték szerint. Mivel az algoritmus során a hozzáadott érték nem nőhet a szubmodulariás miatt, ezért ezzel javíthatunk a futásidőn. Ha a minimum prioritású sorban lévő első elem értékét újraszámoljuk és az még mindig nagyobb, mint a második elemé, akkor őt választjuk be ebben az iterációban, ellenkező esetben ezt folytatjuk tovább. Az is látszik, hogy lehet, hogy nem kell annyiszor újraszámolni az egy elem beválasztásával szerzett nyereséget, de a minimum prioritású sor karbantartása is sok számítási kapacitást igényel.
Ezen két algoritmus vegyítése a **two-stage greedy**, ahol az első k (felhasználótól függő paraméter) lépésben a naív módszert használjuk, majd átváltunk a lazy algoritmusra. Emögött az az intuíció rejlik, hogy a lazy által használt adatstruktúra karbantartása az algoritmus elején a legnehezebb.
A másik három módszer az eddigiekkel ellentétben már nem determinisztikus. A **stochastic greedy** minden iterációjában véletlenül választ egy részhalmazt, amiből a következő elemet választja; a **sample greedy** algoritmus során pedig már az algoritmus legelején történik egy véletlen mintavételezés. Az **approximate lazy** a lazy algoritmus egy nagyon egyszerű kiterjesztése: nem feltétlenül azt az elemet vesszük hozzá a már kiválasztott részhalmazhoz, ami a legnagyobb haszonnal jár, hanem ami "közel legjobb haszonnal jár". Ezt százalékban lehet kifejezni és természetesen függ a felhasználótól. Az utóbbi 3 algoritmus a futásidő felgyorsítására, a számítási kapacitás csökkentésére törekszik, viszont már nem feltétlenül hoz olyan eredményt, mint az első három determinisztikus algoritmus.

2.2. Szubmoduláris dimenziócsökkentés

A dimenziócsökkentés során nem a szubmoduláris megközelítés a legelterjedtebb, ennek ellenére találtunk néhány olyan cikket, melyben már előforul a szubmoduláris dimenziócsökkentés. Erre szeretnék néhány példát mutatni.
Egy entrópia típusú megközelítés a következő [5]:

2.2.1. Definíció (Közös információ)**.** *Legyen X és Y két diszkrét valószínűségi változó. Ezek közös információja legyen definíció szerint:*

$$I(X,Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)},$$

ahol $p(x,y)$ az X és Y közös eloszlásfüggvénye, $p(x)$ és $p(y)$ pedig a peremeloszlásfüggvények.

Az entrópia típusú megközelítés arra utal, hogy a közös információfüggvény kifejezhető az entrópia segítségével is. Az is belátható, hogy ha $\mathcal{F} : 2^V \to \mathbb{R}$ halmazfüggvényt az $X \subseteq V$ részhalmazokra a közös információ segítségével definiáljuk: $\mathcal{F}(X) := I(B;X)$, ahol a B egy V-től független halmaz, akkor egy monoton növő, szubmoduláris függvénnyel van dolgunk. Ennek segítségével meg tudjuk fogalmazni a redukciós feladatot, melyre a cikkben [5] adott egy approximációs algoritmus (ugyanis a feladat természetesen NP-nehéz). A cikkben 6 különböző klasszifikációs feladaton próbálták ki ezt a megközelítést és kétféle modellen végezték a tanítást: Naív Bayes és RBF-network. A szubmoduláris megközelítést olyan alapvető dimenzióredukcióra implementált módszerekkel versenyeztették, mint például a Laplace score, vagy a Fischer score és azt tapasztalták, hogy az egyik adathalmaz esetében a szubmoduláris módszer volt a legjobb, viszont volt több olyan adathalmaz is, ahol jóval lemaradt a már ismert algoritmusoktól.
A "Feature Selection Using Submodular Approach for Financial Big Data" című cikk [6] azért volt fontos, mert ugyanúgy a PCA (Principal Component Analysis) módszert veszik összehasonlítási alapnak, mint ahogy azt mi is tettük a mérések során, illetve a logisztikus regresszió, mint klasszifikációs modell is megjelenik, amit szintén használtunk a kiválasztott adathalmazon. A cikkben hangsúlyozzák, hogy a szubmoduláris megközelítés pénzügyi adatokon kifejezetten jól teljesít. A másik, ami miatt érdekes volt a cikk, az maga az algoritmus: kétféle szubmoduláris függvényt definiálnak és az algoritmus is két fázisból áll. Az első fázisban az összes olyan részhalmazt kiválasztjuk, amelyeken az első függvény (hasonló a már említett entrópia alapú függvényhez) értéke nagyobb, mint egy konstans (ez felhasználófüggő). A második fázisban ezek közül kidobjuk azokat, amik egymással lényegesen korrelálnak, azaz a második függvény értéke ezen részhalmazokon nagyobb, mint egy felhasználótól függő konstans. Ez a két fázisos megvalósítás nem gyakori a szubmoduláris maximalizálási feladatokban.
Több cikkben megjelent az általunk is használt facility-location (szolgáltató elhelyezési) és a feature based függvény, valamint különböző fedési függvényekkel is találkoztunk [7]. A dimenziócsökkentési feladatok mellett sorredukciós feladatokat tárgyaló cikkeket is találtam. Mindemellett mini-batch kiválasztási problémára [8], valamint fehérje sorozatokban fellelhető ismétlődések eliminálására is alkalmazták [9] már a

szubmoduláris függvényeket.

2.3. Egyéb dimenziócsökkentési módszerek - PCA

Több standard dimenzócsökkentő, feature-selection módszert is tanulmányoztunk. Ezek egy része statisztikai eszközöket [11], egy része lineáris algebrai eszközöket alkalmaz az oszlopok csökkentésére. [10]
Ezen módszerek tanulmányozása után választottuk ki az egyik leggyakrabban használt lineáris algebrai megközelítést, ami a **PCA** (Principle Component Analysis) [10], magyarul főkomponens analízis. Ezt a módszert használtuk a projektmunka során összehasonlítási alapnak, ezért szeretném röviden bemutatni. Az algoritmus lényegében arról szól, hogy egy nagy dimenziós adatot levetítünk a legfontosabb tengelyeire, így egy jobban, egyszerűbben kezelhető, kisebb dimenziós adatot kapunk. A legfontosabb tengelyek a kovarianciamátrix nagy sajátértékeihez tartozó tengelyek lesznek, ugyanis ezen helyeken a legnagyobb a szórásnégyzet. Ezeket hívjuk főkomponenseknek.
Másképpen fogalmazva az adatpontokat egy új koordinátarendszerbe transzformáljuk, ahol az első tengely a legnagyobb sajátértékhez tartozó, a második a második legnagyobb sajátértékhez tartozó, stb...
Tegyük fel, hogy az M mátrix sorai tartalmazzák az adatpontokat (n sor, D oszlop) és a D dimenziószámot szeretnénk csökkenteni d-re. Ekkor az algoritmus először kiszámolja az $M^T M$ kovarianciamátrixot, majd ezen mátrix sajátvektorainak és sajátértékeinek számítása, a főkomponensek meghatározása történik. A kiszámolt sajátvektorokat a sajátértékek szerint csökkenő sorrendbe rendezzük, így a főkomponenseket a szignifikanciájuk sorrendjében találjuk meg, a d legfontosabbat választjuk ki. Így kapunk egy feature vektort. Végül az eredeti M adathalmaz transzponáltját beszorozzuk a feature vektor transzponáltjával és így megkapjuk a dimenziócsökkentett adathalmazt.
Jól látható, hogy az algoritmus legtöbb számítási kapacitást igénylő része a kovarianciamátrix kiszámítása és ennek a sajátértékfelbontása. A kovarianciamátrix $\mathcal{O}(nD\min(n, D))$ időben megkonstruálható (mátrixszorzás), a sajátértékfelbontás pedig legrosszabb esetben is $\mathcal{O}(D^3)$ időt vesz igénybe. Mindezt összevetve az algoritmus futási ideje $\mathcal{O}(nD\min(n, D) + D^3)$.

3. Mérési eredmények

3.1. A vizsgált adathalmaz

Az idei méréseket egy, már az előző félévben megismert adaton végeztük. A *"Disaster Tweets"* című, a Kaggle [4] oldalon található adathalmazzal egy NLP (Natural Language Processing) feladatot szerettünk

volna megoldani. Az adat különböző tweetekből állt, melyeket klasszifikálni szeretnénk aszerint, hogy valójában egy igazi katasztrófáról szólnak-e, vagy sem.
Azért ezen az adaton próbáltuk finomítani a módszert, mert az előző félévben azt tapasztaltuk, hogy az Apricot-ban implementált szubmoduláris függvények maximalizálásával történő dimenziócsökkentés jól működik. Az alkalmazott modell a logisztikus regresszió volt.

3.2. A mohó algoritmusok összehasonlítása

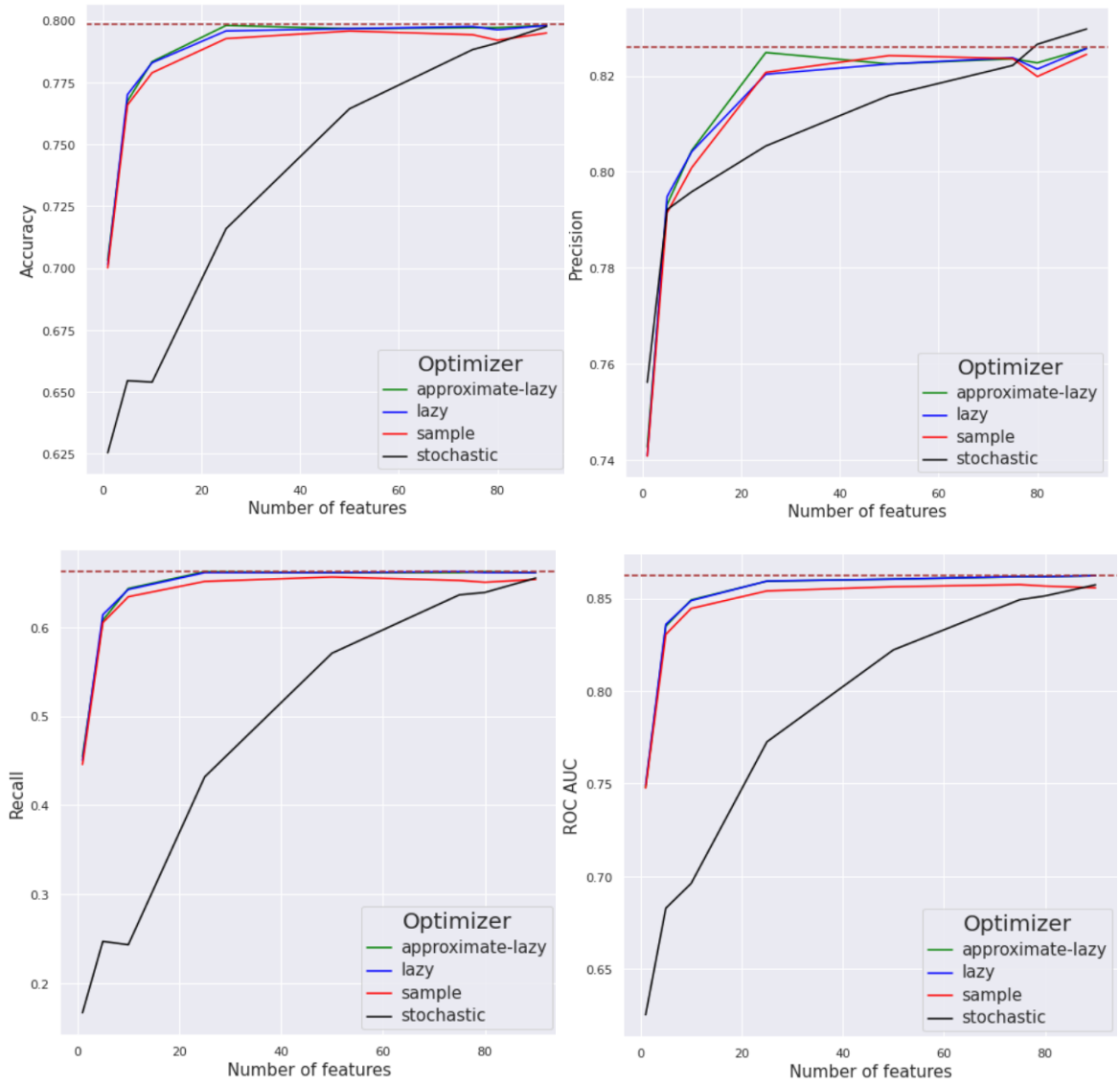
Mint már említettem a félévben egy fontos feladat volt, hogy a csomagban implementált mohó algoritmusok működéséről benyomást szerezzünk, ezért ezeket performanciájukat és futásidejüket tekintve is megpróbáltuk összehasonlítani.
A beszámoló 2.1. alfejezetében írtak után látszik, hogy a naive, lazy és a two-stage algoritmusoknak elég csupán a futásidejüket vizsgálni, mivel ugyanazt az eredményt adják, csak az implementáció más. Mivel a two-stage algoritmus azt a célt szolgálná, hogy a naive (és esetleg a lazy) algoritmus futásidejét gyorsítsa, ezért azt vártuk, hogy gyorsabb lesz, mint a többi módszer. Ennek ellenére az 1. táblázatból látszik, hogy a vizsgált adaton még a naive módszertől is lassabb volt, a lazy módszer pedig nagyságrendekkel jobb nála.

Optimizer \ Size	1 %	5%	10 %	25%	50%
approximate-lazy	9.87	38.29	85.97	318.35	838.84
lazy	**5.85**	**8.65**	**24.38**	**166.21**	**670.32**
naive	8.79	21.41	49.76	216.48	783.08
sample	12.15	47.94	96.67	257.53	870.33
stochastic	8.07	18.83	43.26	217.42	785.12
two-stage	6.71	24.62	50.72	238.43	827.12

1. táblázat: A vizsgált mohó algoritmusok összehasonlítása futásidőt tekintve.

A lazy algoritmus még a nemdeterminisztikus eljárásoktól is gyorsabban futott le. Ezt úgy vizsgáltuk, hogy az approximate-lazy, a sample és a stochastic mohó algoritmusokat többször futtattuk, a futásidőket kiátlagoltuk és így hasonlítottuk össze a lazy algoritmus futásidejével. A kapott eredmény azért is meglepő, mert ezen algoritmusoknak a célja az lenne, hogy bizonyos mértékű performancia romlással számítási

kapacitás, valamint futásidő csökkentést érjünk el. Ez a mérés alapján a vizsgált adaton nem működött. (A futásidőbeli különbségeket az 1. táblázat, a teljesítménybeli különbségeket az 1. ábra mutatja.)



1. ábra: A vizsgált mohó algoritmusok összehasonlítása teljesítményt tekintve. A szaggatott vonal a dimenziócsökkentés nélkül tanított logisztikus regresszióval elért eredményt mutatja.

3.3. A feature-based függvény legjobb paraméterezése

A feature-based szubmoduláris függvény legjobb paraméterezésének vizsgálatával (az AUC értéket tekintve) is sok időt töltöttünk. A kapott eredményeket a 2. táblázat mutatja. Meglepő, hogy az "approximate-lazy" algoritmus is szerepel a táblázatban, viszont konstatálható, hogy jó döntés, ha a konkáv függvényünket a $\log(X+1)$-nek, a maximalizálásokat végző mohó algoritmust a lazy-nek választjuk.

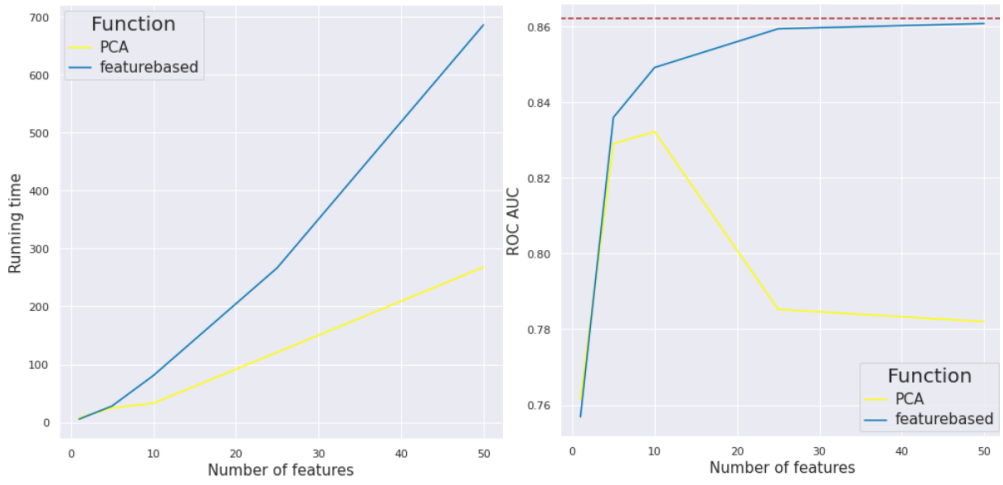
Size	Best parameters
1 %	{"function": "log", "optimizer": "lazy"}
5 %	{"function": "log", "optimizer": "approximate-lazy"}
10 %	{"function": "sqrt", "optimizer": "approximate-lazy"}
25 %	{"function": "log", "optimizer": "approximate-lazy"}
50 %	{"function": "log", "optimizer": "lazy"}

2. táblázat: A feature-based függvény legjobb paraméterezése a különböző méretű redukciók során.

3.4. Az Apricot összehasonlítása a PCA módszerrel

Az Apricot módszert a 2.3. alfejezetben bemutatott PCA dimenziócsökkentő eljárással hasonlítottuk össze. Azért éreztük ennek fontosságát, mert az előző félévben a TfIdf (Term frequency–Inverse document frequency) módszerrel már összevetettük, viszont az egy kifejezetten NLP feladatokra működő algoritmus. Szerettük volna egy sokkal általánosabb eljárással is összehasonlítani a szubmoduláris megközelítést. Az összehasonlítást a feature-based függvénnyel, ennek is az előző fejezetben említett legjobb paraméterezésével végeztük.

A PCA sok számítási kapacitást igénylő módszer, ennek ellenére futásidőben jobb eredményt hozott, mint az Apricot (2. ábra). Mivel a legjobb paraméterezés nem a futásidő optimalizálására irányult, ezért úgy érzem ezen lehetne javítani. Performanciát tekintve (2. ábra) viszont a PCA éri el a gyengébb eredményt. A 2. ábráról leolvasható, hogy az AUC értékeket tekintve a feature-based függvénnyel nagyságrendekkel jobb eredményt lehet elérni, mint a PCA-val.



2. ábra: Az Apricot és a PCA módszer összehasonlítása futásidőt (bal), illetve teljesítményt (jobb) tekintve. A jobb oldali ábrán a szaggatott vonal a dimenziócsökkentés nélkül tanított logisztikus regresszióval elért eredményt szemlélteti.

4. Összegzés

Az idei félévben is új adatbányászati modelleket, módszereket tanultam, programozási trükköket sajátítottam el, továbbá különböző mérési felületekkel, technikákkal ismerkedtem meg. A gyakorlati tapasztalat megszerzése mellett az elméleti háttér felderítésére is sok időt szántam.

Az eredményeket tekintve összességében elmondható, hogy van létjogosultsága a szubmoduláris függvényekkel való adathalmaz redukciónak. Főleg a dimenziócsökkentés területén bizonyult hasznosnak a módszer.

A továbbiakban is tervezzük folytatni a munkát, a legfontosabb célunk, hogy egy még nem implementált szubmoduláris függvényt írjunk a csomagba, melyre jelenleg a legjobb jelölt az elméleti részben írtakhoz hasonló entrópia megközelítést használó függvény.

Hivatkozások

[1] Hui Lin, Jeff Bilmes, A Class of Submodular Functions for Document Summarization, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA, (2011), 510–520

[2] Jacob Schreiber, Jeffrey Bilmes, William Stafford Noble, Apricot: Submodular selection for data summarization in Python, (2019)

[3] https://github.com/jmschrei/apricot

[4] https://www.kaggle.com/

[5] Xiao-Zhong Zhu, William Zhu, Xin-Nan Fan, Rough set methods in feature selection via submodular function, *Springer-Verlag*, Berlin Heidelberg, (2016)

[6] Girija Attigeri, Manohara Pai M. M., Radhika M. Pai, Feature Selection Using Submodular Approach for Financial Big Data, *J Inf Process Syst*, Vol 15, No.6, pp. 1306-1325, December (2019)

[7] Yuzong Liu, Kai Wei, Katrin Kirchhoff, Yisong Song, Jeff Bilmes, Submodular Feature Selection for High-Dimensional Acoustic Score Space, *Department of Electrical Engineering, Department of Computer Science and Engineering University of Washington*, Seattle, USA, (2013)

[8] K J Joseph , Vamshi Teja R , Krishnakant Singh , Vineeth N Balasubramanian, Submodular Batch Selection for Training Deep Neural Networks, *arXiv:1906.08771v1 [cs.LG]* 20 Jun (2019)

[9] Maxwell W. Libbrecht, Jeffrey A. Bilmes, William Stafford Noble, Eliminating redundancy among protein sequences using submodular optimization, *bioRxiv*, Department of Computer Science and Engineering, University of Washington, Department of Electrical Engineering, University of Washington, Department of Genome Sciences, University of Washington (2016)

[10] Jure Leskovec, Anand Rajaraman, Jeff Ullman, Mining of Massive Datasets - Dimensionality reduction (Chapter 11, 405-437), Stanford University, (2014)

[11] https://scikit-learn.org/stable/modules/feature_selection.html