Budapesti Corvinus Egyetem

Matematikai és Statisztikai Modellezés Intézet

Statisztika Tanszék

Témamodellezési algoritmusok konzisztenciájának vizsgálata gyakorlati példán keresztül

Egyedi Zsolt

Gazdaságinformatikus BSc

Szakszeminárium vezető: Kovács László

Tartalomjegyzék

[1. Bevezetés 3](#_Toc109389882)

[2. Egységes adatformátum 3](#_Toc109389883)

[3. Szavak gyakoriságának elemzése 6](#_Toc109389884)

[4. Szavak közötti kapcsolatok vizsgálata 9](#_Toc109389885)

[5. Konvertálás adattárolási módok között 11](#_Toc109389886)

[6. Témamodellező algoritmusok 13](#_Toc109389887)

[6.1. Látens Dirichlet-allokáció (LDA) 13](#_Toc109389888)

[6.2. Látens szemantikai elemzés (LSA) 16](#_Toc109389889)

[6.3. Korrelált témamodell (CTM) 16](#_Toc109389890)

[7. Használt adatbázis bemutatása 16](#_Toc109389891)

[8. Népszerű témamodellező algoritmusok empirikus összehasonlítása 16](#_Toc109389892)

[9. Összefoglalás 16](#_Toc109389893)

[Irodalomjegyzék 17](#_Toc109389894)

# Bevezetés

A természetes nyelvi feldolgozás napjaink egyik legnépszerűbb kutatási témája. Az egyre gyorsabban keletkező szövegeknek és videóknak köszönhetően megnövekedett az igény ezek elemzésére, kategorizálására. Ebbe a tudományágba tartoznak a témamodellező algoritmusok is, amik matematikai egyenletek segítségével homogén csoportokba osztják a dokumentumokat.

Szakdolgozatomban bemutatom, hogy Python környezetben hogyan lehet úgy adatot tisztítani és átalakítani, hogy azokat később elemezni lehessen különböző algoritmusokkal. Ezután bemutatom, hogy mi a tf-idf és a dokumentum-kifejezés mátrix és hogyan lehet ezeket előállítani. Ismertetem a különböző legnépszerűbb témamodellező algoritmusokat, illetve azok logikáját, matematikai hátterét is.

Célom a témamodellező algoritmusok által generált kimenetek közötti hasonlóságok és eltérések feltárása. A hipotézisem az, hogy ezek az algoritmusok ugyanazoknál a bemeneteknél eltérő eredményekre jutnak, azonban nincs statisztikai értelemben szignifikáns különbség közöttük. Megvizsgálom, hogy mik lehetnek az eltérések okai (ha egyáltalán vannak) és hogy fel lehet-e állítani egy rangsort a különböző modellek között valamilyen objektív preferenciarendszer alapján.

Ahhoz, hogy empirikusan igazoljam a hipotézisemet, egy választott adatbázison futtatom az algoritmusokat. Ezután az eredményeket statisztikai módszerekkel összevetem és elemzem (például besorolási eloszlások elemzése az egyes algoritmusok szerint). Konkrétan a konzisztenciát Pearson-féle khi-négyzet próba segítségével tesztelem két eljárás között.

# Egységes adatformátum

Az adatbányászati munkát megkönnyíti, ha a bemeneti adattáblák strukturáltan, egységes formátumban (tidy text format) vannak tárolva. Egy strukturált adattábla olyan értékek gyűjteménye, amelyek változók (azaz tulajdonságok) és egyedek (megfigyelések) szerint van csoportosítva. A változók a megfigyelések jellemzői, míg az egyedekhez tartozik minden olyan érték, amit ugyanarra az egységre mértek (Wickham, 2014).

Egységes formátumra hozáskor a változók, a megfigyelések és az értékek egymáshoz viszonyított helyzetét rögzítjük, az alábbiak szerint:

* A tulajdonságok oszlopok szerint vannak csoportosítva.
* A megfigyelések soronként vannak.
* Egy tábla egy megfigyelési egységet tartalmaz.

Az így kialakult táblát strukturált adatoknak vagy egy token – egy sor formátumnak is hívják. A token olyan egység, amit később elemezni szeretnénk, míg a tokenizáció annak a folyamata, hogy az általunk vizsgálni kívánt egységre hozzuk az értékeket. A token általában csak egy szó, de lehet például mondat vagy bekezdés is. Az így létrejött adattáblát sokkal egyszerűbb módosítani, összehasonlítani és elemezni is (Silge & Robinson, 2017).

Az egy token – egy sor módszer egy kiváló egységes adatformátum, azonban adatbányászati vagy nyelvészeti szemszögből is érdemes megnézni, hogyan lehet tárolni a szövegeket. Három típusa van: *string*, korpusz és dokumentum-kifejezés mátrix. Stringként szöveget tárolunk, ami lehet egy betű, szó, vagy akár egy bekezdés is. A korpusz általában nagy terjedelmű, több stringből álló, strukturált objektum, de tartalmazhat ezeket leíró metaadatokat is. A metaadat egy másik adatot leíró, arról mögöttes információt szolgáltató adat (Pomerantz, 2015). A harmadik adattárolási típus a dokumentum-kifejezés mátrix (document-term matrix, dtm), ami egy olyan tábla, ahol a megfigyelések dokumentumok (például korpusz), a változók pedig kifejezések. Az így kialakult mátrix értékei leggyakrabban szavak előfordulásának gyakorisága vagy *tf-idf* (bővebben a 3. fejezetben) (Silge & Robinson, 2017).

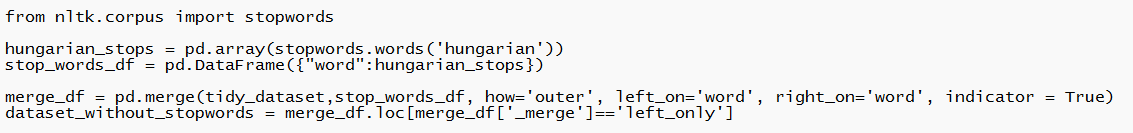
Ahhoz, hogy egy tetszőleges szöveget egységes adatformátumra hozzunk Pythonban, szükség lesz a *pandas* és a *tidytext* nevű könyvtárakra. A pandas egy alapértelmezett Python modul, viszont a tidytext nem, így ennek az élréséhez be kell írni a Parancssorba a következő kódot: *pip install tidytext* (Chow, 2020).

Ezután Pythonban be kell importálni a pandas könyvtárat és a tidytextből az *unnest\_tokens* függvényt. Következő lépésben létrehozhatunk egy tetszőleges változót, amiben megadjuk az elemezni kívánt szöveget. Ezt fel kell bontani sorok szerint, majd egy *dataframe*-be betölteni. A dataframe a pandas csomagnak egy olyan elsődleges adatstruktúrája, ami két dimenzióban tárolja az adatokat, sorok és oszlopok szerint címkézve (*User Guide — Pandas 1.4.2 Documentation*, n.d.). Így kerül a szöveg strukturált formátumba, ugyanis ez egy olyan adattábla, ahol egy sor egy megfigyelés (jelen esetben a szöveg egy sora), míg egy oszlop egy tulajdonság. Ezután lesz szükség a tokenizálásra, ugyanis a jelenlegi egy sor – egy sor elrendezést egy szó – egy sorra kell módosítani. Ebben segít az unnest\_tokens: paramétereiként meg kell adni azt a dataframe-et, amit tokenizálni szeretnénk; azt, hogy egy token mivel legyen egyenlő (betű, szó, stb.) és hogy a dataframe-ből melyik oszlopot kell átalakítani. Az unnest\_tokensmásik előnye a tokenizáláson kívül, hogy kiszűri a szövegből a szövegközi és mondatvégi írásjeleket is, illetve kis kezdőbetűssé változtat minden szót (Chow, 2020).

A képen szöveg látható

Automatikusan generált leírás

1. *ábra: Szöveg tokenizálása (Forrás: saját szerkesztés)*

Most, hogy már van egy strukturált, egy szó – egy sor elrendezésű adattáblánk, az elemzések előtt érdemes kiszűrni az olyan szavakat, amiket felesleges elemezni (stop words, stop szavak). Az ilyen szavak tartalmilag nem bővítik a szöveget, csupán torzítanák az elemzések eredményét (Silge & Robinson, 2017). Ezek eléréséhez szükség lesz Pythonban az *nltk* csomagra, ami a „*python -m nltk.downloader popular*” utasítás Parancssorba írásával letöltődik (*NLTK: Installing NLTK*, 2022). Ezután a második ábrán látható kódsorokkal el tudjuk távolítani a nem kívánatos szavakat a rendezett adattáblánkból.

1. *ábra: Stop szavak eltávolítása (Forrás: saját szerkesztés)*

A fenti kódban a *merge* függvény működése logikailag nagyon hasonlít egy SQL (Strukturált Lekérdező Nyelv, Structured Query Language) lekérdezéshez. SQL-ben adattáblákat lehet csatlakoztatni egymáshoz (join), ha van olyan oszlopuk, ahol az értékek (akár csak részlegesen) megegyeznek egymással. Összekapcsolásból létezik belső (inner), teljes külső (full outer), bal külső (left outer) és jobb külső (right outer), amik abban különböznek egymástól, hogy null értékek mentén összekapcsolt sorokat megjelenítenek-e a kimeneti táblában. Ahogyan a harmadik ábrán is látható, a full outer join esetén mindkét táblából megtartunk minden sort (*SQL OUTER JOIN*, n.d.).

1. *ábra: Stop szavak eltávolítása (Forrás: SQL OUTER JOIN, é. n.)*

A mi esetünkben először a strukturált adattáblát kapcsoljuk össze full outer join-al a kiszűrni kívánt *stop\_words\_df* táblával, ahol a kapcsolatot mindkét esetben a word nevezetű oszlop képezi. Ilyenkor Pythonban létrejön egy *\_merge* nevű oszlop, ami tartalmazza, hogy az adott sorban található szó (word) csak a bal (left\_only), csak a jobb (right\_only) vagy mindkettő (both) csatlakoztatott adattáblában megtalálható-e. Ahhoz, hogy az elemezni kívánt adatokból eltávolítsuk a felesleges szavakat, csupán a left\_only kifejezésre kell szűrni. Hiszen ezzel a „jobb oldali”, stopszavak táblában szereplő szavakat nem őrizzük meg az eredményben.

# Szavak gyakoriságának elemzése

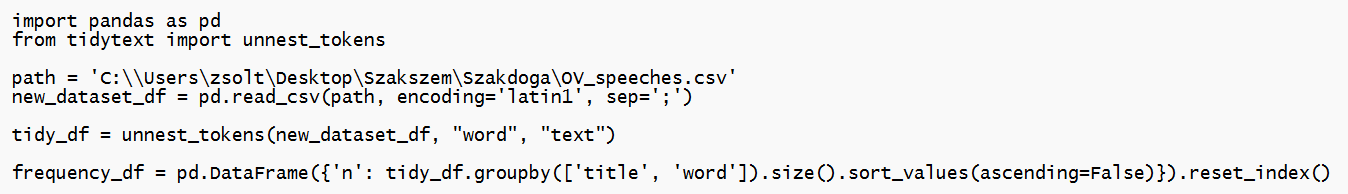
Miután egy helyre gyűjtöttük az adatainkat és elemzésre alkalmas struktúrájúvá alakítottuk őket, elkezdhetjük megvizsgálni, hogy mi is a szöveg tartalma. Mivel a szöveg szavakból áll össze, így logikus gondolat lehet, hogy nézzük meg, hogy mik a leggyakrabban előforduló kifejezések, ugyanis ezek határozzák meg leginkább a dokumentumot. A kifejezések gyakorisága (term frequency, tf) azonban nem feltétlen ad olyan pontos eredményt, mint várnánk, ugyanis a legtöbb esetben az olyan szavak, mint például az „általában”, „mondta”, „mert” sokkal többször fordulnak elő, mint bármelyik másik szó. Erre a felesleges szavak kiszűrése sem egy tökéletes megoldás, ugyanis nem tudjuk pontosan, hogy az elemezni kívánt dokumentumban mik a nem szükséges kifejezések, minden alkalommal kézzel módosítani a listát pedig időigényes módszer (Silge & Robinson, 2017).

Egy hatásos megoldás az előbbi problémára megnézni egy szónak az inverz dokumentum gyakoriságát (inverz document frequency, a továbbiakban idf). Az idf aszerint ad értéket egy szónak, hogy milyen gyakran fordul elő dokumentumok gyűjteményében: a sűrűn használt szavak értéktelenebbek, míg a ritka kifejezések nagyobb értéket kapnak. Az idf az alábbi képlettel számolható ki (Silge & Robinson, 2017).:

Az inverz dokumentum gyakoriság George Zipf, 20. századi nyelvész elméletén alapszik. A róla elnevezett Zipf-törvény kimondja, hogy egy szó gyakorisága fordítottan arányos a szónak a gyakorisági táblában elfoglalt helyezésével (Silge & Robinson, 2017).

Ha a szavak gyakoriságát súlyozzuk a hozzájuk tartozó idf értékekkel, akkor megkapjuk a kifejezések gyakorisága – inverz dokumentum gyakoriságot (term frequency – inverz document frequency, a továbbiakban tf-idf). A tf-idf nagy előnye, hogy dokumentumok gyűjteményénél nem veszi figyelembe a mindegyikben előforduló szavakat, hanem Zipf-törvénye alapján hangsúlyosabbak lesznek az adott szövegre jellemző ritkább, de meghatározó kifejezések (Silge & Robinson, 2017).

Ebben a fejezetben az elemzésekhez egy olyan CSV (Vesszővel Tagolt Értékek, Comma Separated Values) fájlt fogunk használni, amiben Orbán Viktornak három nyilvános beszédének a szövege található. A fájl kettő oszlopot tartalmaz: a title (cím) a beszédeknek a helyét és okát írja le, míg a text (szöveg) oszlopban találhatóak a szövegek teljes terjedelmükben.

A CSV fájl beolvasása és dataframe-be töltése után az előző fejezetben kifejtettek alapján egy szó – egy oszlop formátumra kell hozni az adattáblát. Ezután ismét egy SQL-ben használt utasítást kell használni az adatok aggregálásához. A csoportosítás (group by) által a többször előforduló adatok egyszer fognak csak szerepelni, ezáltal különböző matematikai és statisztikai mutatókkal lehet őket megvizsgálni (*User Guide — Pandas 1.4.2 Documentation*, n.d.). Jelen esetben a title, majd a word oszlopok szerint kell csoportosítani, így megkapjuk, hogy az egyes szövegekben milyen szavak szerepelnek. Ezután a size() utasítás megszámlálja, hogy egy szó szövegenként hányszor fordul elő (ennek neve legyen *n*). A reset\_index() azért szükséges a kódba, mert anélkül a végeredmény csak az *n* oszlopot tartalmazná, a title és word csak indexek lennének, így viszont ezek is a dataframe oszlopai maradnak (Silge & Robinson, 2017).

*4. ábra: Szavak gyakoriságának meghatározása (Forrás: saját szerkesztés)*

A képen szöveg látható

Automatikusan generált leírás

*5. ábra: A példa adatbázisban szereplő szavak gyakorisága dokumentumonként (Forrás: saját szerkesztés)*

A képen szöveg látható

Automatikusan generált leírásAhogyan a fenti ábrán is látszik, a három dokumentumban leggyakrabban előforduló szavak az „a”, „hogy”, „és”. Ezek megnehezítik a tartalom elemzését, ezért a tf-idf számolást fogjuk alkalmazni, hogy megtudjuk melyek szavak azok, amik gyakran előfordulnak a szövegekben – de nem túlságosan. Az előző fejezetben is használt tidytext könyvtárból meghívható a bind\_tf\_idf függvény, aminek bemenő paramétereinek meg kell adni a dokumentumokat és szavakat strukturáltan tartalmazó adattáblát (esetünkben a frequeny\_df); a szavakat tartalmazó oszlop nevét (word); a dokumentumokat tartalmazó oszlop nevét (title) és a kifejezések gyakoriságát (n). Eredményül megkapjuk mindegyik szónak az idf és tf-idf értékét is (Silge & Robinson, 2017).

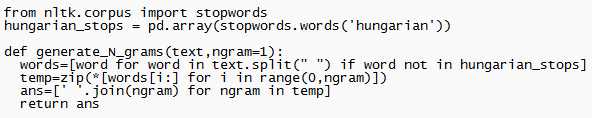
6. ábra: *A példa adatbázisban szereplő szavak tf-idf-e dokumentumonként (Forrás: saját szerkesztés)*

A fenti ábrán látható, hogy a tf-idf értékek alapján rendezett listában már nem szerepelnek névelők. Az is belátható, hogy a „nyertünk” és a „győzelem” szavak elég jelentőségteljesek és lényegesek egy országgyűlési választást értékelő beszédben.

# Szavak közötti kapcsolatok vizsgálata

Az előző fejezetben minden szót önállóan elemeztünk, azonban így elvesznek a szavak közötti kapcsolatok. Egy mondat szempontjából például nem mindegy, hogy milyen szavak alkotják és milyen sorrendben jönnek egymás után a kifejezések. Ezért érdemes lehet megvizsgálni, hogy milyen szavak jelennek meg a szövegekben közvetlenül egymás után, illetve, hogy mely kifejezés-párok fordulnak elő együtt leggyakrabban a dokumentumokban (Silge & Robinson, 2017).

Az egymás után következő, összefüggő szavak sorozatát (vagy akár betűkét is) n-grammoknak (angolul n-grams) hívják. Ennek egy speciális fajtája a bigrammok (bigrams). Ilyenkor az n értéke kettő, tehát két szó követi egymást közvetlenül. Elemzésünkben bigrammokat fogunk használni, aminek a menete logikailag ugyanaz lesz, mint amikor egy szót elemeztünk, annyi különbséggel, hogy most szópárokat fogunk vizsgálni (Silge & Robinson, 2017).

Az eddigi eljáráshoz hasonlóan szükséges a szöveget egy token – egy sor formátumra hozni, ahol a token jelen esetben egy bigram lesz. Erre nincs beépített Python függvény, ezért az alábbi metódus segítségével fogunk egy szöveget tetszőleges számú n-grammá alakítani (Nithyashree, 2021). A függvény előnye, hogy az első fejezetben összegyűjtött stop word-öket nem veszi figyelembe.

*7. ábra: Ngrammokat generáló függvény (Forrás: saját szerkesztés)*

Az eljárás lényege, hogy a bemeneti értéknek megadott stringet szóközök mentén feloszt és ha a szó nem stop word, akkor egy változóban eltárolja. Ezután ezt a változót olyan elemszámú tömbökbe rendezi, amit megadtunk a függvény ngram értékének. Péládul bigram esetén kettő elemszámú lesz a tömb. Végezetül a tömbökben lévő szavakat összefűzi szóközökkel és visszaalakítja stringgé (Nithyashree, 2021).

Érdemes lehet kiszűrni a mondatközi és mondatvégi írásjeleket is a szövegből. Ehhez egy olyan függvényt használunk, ami a string nevű Python könyvtárban található punctuation lista alapján eltávolítja az írásjeleket (Nithyashree, 2021).

Végül a két függvény meghívásával egy dataframeben tudjuk eltárolni az egy bigram – egy sor szerint rendezett szöveget. Ezután a kapott adattáblát az előző fejezetben bemutatott módon lehet alkalmazni kifejezések gyakoriságának és tf-idf értékek számolásához. Kifejezetten hosszú szövegek elemzésénél lehet hasznos ngrammok alkalmazása, olyankor ugyanis egy-egy szó nagyon sokszor előfordul a szövegekben, míg bigrammok alkalmazása több kontextust nyújthat az analízishez (Silge & Robinson, 2017).

A képen szöveg látható

Automatikusan generált leírás

*8. ábra: A szöveget bigrammokká felosztó kódsor (Forrás: saját szerkesztés)*

A képen szöveg látható

Automatikusan generált leírás

*9. ábra:* *A példa adatbázisban szereplő bigrammok dokumentumonként (Forrás: saját szerkesztés)*

# Konvertálás adattárolási módok között

Az eddigi fejezetekben a szöveget mindig egy token – egy sor formátumban tároltuk és elemeztük. Azonban a további vizsgálatokhoz szükség lesz dokumentum-kifejezés mátrixok (document-term matrices, a továbbiakban dtm) és korpuszok alkalmazására is. Emiatt a szövegeket dtm-mé vagy korpusszá kell alakítani, majd azokat újra átkonvertálni egységes adat-formátumúvá (Silge & Robinson, 2017).

Az elemzésre kiválasztott szöveg dokumentum-kifejezés mátrixának a kiszámításához a pandas könyvtáron kívül az *sklearn* csomagot lesz szükséges importálni. Azon belül is a *CountVectorizer* függvény *fit\_transform* metódusát kell használni. A kód megvizsgája, hogy a bemeneti értékként megadott szöveg milyen szavakat tartalmaz, majd megszámolja, hogy ezek melyik dokumentumban milyen gyakorisággal fordulnak elő (*Sklearn.Feature\_extraction.Text.CountVectorizer*, n.d.). Az így kapott számok lesznek a dtm értékei, míg a sorai a dokumentumok címei, az oszlopai pedig a szövegeket alkotó szavak A képen szöveg látható

Automatikusan generált leíráslesznek.

*10. ábra: Dokumentum-kifejezés mátrix létrehozása (Forrás: saját szerkesztés)*

A képen szöveg látható

Automatikusan generált leírás

*11. ábra: A példa adatbázis dokumentum-kifejezés mátrixának egy részlete (Forrás: saját szerkesztés)*

A képen szöveg látható

Automatikusan generált leírásDokumentum-kifejezés mátrix készíthető tf-idf értékek felhasználásával is. Ennek előnye a 2. fejezetben kifejtettekkel megegyezik: a gyakran előforduló szavak nem feltétlenül lényegesek a téma meghatározásához, ezért a tf-idf használatával pontosabb eredmény kapható. Ez Pythonban a fentebb bemutatott módszerhez hasonlóan elvégezhető, annyi különbséggel, hogy ilyenkor a *CountVectorizer* függvény helyett a *TfidfVectorizer* használata szükséges.

*12. ábra: Dokumentum-kifejezés mátrix létrehozása tf-idf értékekkel (Forrás: saját szerkesztés)*

Felmerülhet az igény arra, hogy egy dtm-et egy szó – egy sor adatstruktúrában tároljunk. Ilyenkor a *stack* metódus segítségével az oszlopon lévő tulajdonságot megfigyelési egységgé lehet változtatni (vagyis áthelyezni a sorokra) (*User Guide — Pandas 1.4.2 Documentation*, n.d.). Ezután a *reset\_index* használatával az eddig indexként értelmezett tulajdonságokat oszlopokká lehet alakítani, illetve el lehet nevezni az értékeket. Végül a *rename* függvénnyel átnevezzük az immáron megfigyelési egységként funkcionáló kifejezéseket (*Convert Columns into Rows with Pandas*, 2019). Így elértük, hogy egy dokumentum-kifejezés mátrix, ahol a sorokon dokumentumok, az oszlopokon pedig szavak vannak, egy dokumentum – egy szó – egy sor struktúrájú legyen.

A képen szöveg látható

Automatikusan generált leírás

*13. ábra:* *Dokumentum-kifejezés mátrix tokenizálása (Forrás: saját szerkesztés)*

# Témamodellező algoritmusok

Nagy mennyiségű, eltérő tartalmú dokumentumoknál felmerülhet az igény a tartalom szerinti különválasztásra, témák szerinti csoportosításra. Napjainkban ezeket a feladatokat téma modellező (topic modelling) algoritmusok végzik, amik más és más szempontok alapján különböző homogén csoportokba rendezik a szöveges tartalmakat. Ezt a műveletet klaszterelemzésnek (clustering) hívják, ami a nem felügyelt gépi tanulás (unsupervised machine learning, ML) csoportjába tartozik (VanderPlas, 2017).

A gépi tanulás olyan függvényapproximáció, amely során matematikai modellek alkalmazásával „megértjük” az adatokat, majd a felhasznált adatok alapján fejlesztjük a modellt oly módon, hogy pontos elemzéseket, előrejelzéseket tudjunk készíteni. A gépi tanuló algoritmusokból létezik felügyelt (supervised) és nem felügyelt (unsupervised) is. Létezik a kettő között egy átmeneti kategória is, a megerősítéses gépi tanulás, de ezzel most nem foglalkozunk (VanderPlas, 2017).

A felügyelt gépi tanulás során magyarázó változók (inputok vagy attribútumok) és eredményváltozók (outputok) alapján tanulja meg a gép, hogyan tudja különböző attribútumok esetén megbecsülni az eredményváltozót. A tanulás folyamata abban rejlik, hogy megtanulja a gép, hogy miképpen tudja megtenni ezeket a becsléseket. Például, van egy adatbázisunk, amiben lakásokra vonatkozó információk vannak (például méret, szobaszám, elhelyezkedés, vételi ár stb.). Hogyha arra lennénk kíváncsiak, hogy ezen adatok alapján mennyi lehet a piaci ára egy általunk kiválasztott lakásnak, akkor egy felügyelt gépi tanulás modellt érdemes használni. A modellben a lakás ára lenne az eredményváltozó, az összes többi attribútum pedig az input. Ezután a gép az adatbázisunk alapján összefüggéseket állít fel az inputok-outputok között. Ha elég nagy az adatbázisunk és megbízható modellt alkalmaztunk, akkor lakásvásárlásnál a lakás paraméterei alapján az algoritmus meg tudja határozni a lakás piaci árát. Ez azonban nem feltétlenül lesz a tényleges piaci ár, mivel az algoritmus a felhasznált adatok alapján adott becslést (VanderPlas, 2017).

A nem felügyelt gépi tanulás annyiban más, hogy ilyenkor nincsen kitüntetett eredményváltozó, az algoritmus mégis képes tudást kinyerni az adatokból. A témamodellezés is ebbe a kategóriába tartozik, ugyanis dokumentumok alapján hoz létre a tanuló algoritmus témákat, nincsenek előre meghatározott dokumentum-téma párok, amin betanulna a modell (VanderPlas, 2017).

## Látens Dirichlet-allokáció (LDA)

A téma modellező algoritmusok közül a legnépszerűbb a Látens Dirichlet-allokáció (Latent Dirichlet Allocation, a továbbiakban LDA). Az LDA egy olyan matematikai modell, aminek az egyik alapelve, hogy minden dokumentum több témából tevődhet össze és ezek a témák különböző arányban jelennek meg az egyes dokumentumokban. A másik jelentős alapelv, hogy minden téma kulcsszavak különböző kombinációját tartalmazza. Minden témában vannak olyan szavak, amik máshol nem jelennek meg, azonban vannak olyanok is, amik más témákra is jellemzőek. A Látens Dirichlet-allokáció működése során megbecsüli, hogy melyik szavaknak a kombinációi alkotnak elkülöníthető témákat, miközben meghatározza, hogy ezek a témák milyen arányban írják le az egyes dokumentumokat. Fontos tulajdonsága az algoritmusnak, hogy előre meg kell adni a feltételezésünket arra, hogy hány téma különíthető el a vizsgált korpuszban, mert az algoritmus mindig egy általunk megadott számú témát különít el (Silge & Robinson, 2017).

A modell gamma változójának hívják azt, hogy milyen valószínűséggel rendelhető téma dokumentumhoz. Mivel egy dokumentumot nem csak egy téma jellemezhet, ezért a gamma vizsgálatával megállapítható, hogy az egyes dokumentumokat mennyire jellemzi egy vagy több téma. Az LDA modell másik valószínűsége a béta, ami megmutatja a témákhoz tartozó szavak valószínűségét. Minél magasabb egy szó-téma párnak a bétája, annál jobban írja le az adott szó a vizsgált témát (Blei et al., 2003).

Az LDA a szavak témákba sorolásához feltételes valószínűségeket használ. A feltételes valószínűség egyenletét akkor érdemes használni, ha van egy A esemény, aminek a bekövetkezési valószínűséget akarjuk megtudni, abban az esetben, ha B esemény már bekövetkezett. Ezt úgy lehet kiszámolni, hogy elosztjuk annak a valószínűségét, hogy A és B esemény egyszerre bekövetkezik B esemény bekövetkezési valószínűségével (Shafer, 1985).:

Az LDA meghatározza, hogy milyen szavak alkotják az egyes témákat, majd az így kialakult témákat (vagyis szavak gyűjteményét) összeveti a dokumentumokat alkotó szavakkal és így dokumentum-téma valószínűségeket számol ki. Mivel a dokumentumokat alkotó szavak gyűjteménye a rendelkezésünkre áll, ezért az előbbit, azaz a béta valószínűségeket kell az algoritmusnak meghatároznia. Ehhez az LDA végigmegy az összes dokumentumon és az ezeket alkotó szavakat véletlenszerűen az előre meghatározott számú témába osztja. Ezután az algoritmus újra végigmegy minden dokumentumon belül az szavakon és feltételezi, hogy a vizsgált SZ szó kivételével a téma-beosztások helyesek, majd az alábbi valószínűség kiszámításával frissíti, hogy melyik témához tartozik (Blei et al., 2003).:

A frissítés során, az első alkalomhoz hasonlóan, az algoritmus véletlenszerűen rendeli szavakat témába. Azonban, míg legelőször a hozzárendelés egyenletes eloszlású valószínűségek szerint történik, addig a második alkalomtól kezdve a fenti valószínűségek felhasználásával fog megtörténni a témához rendelés (Blei et al., 2003).

A fenti egyenlet két részből tevődik össze. Az első része () meghatározza, hogy dokumentumból milyen arányban tartoznak szavak témába, az éppen vizsgált szót leszámítva. Ilyenkor azzal a feltételezéssel élünk, hogy minél több szó tartozik témába dokumentumból (amiben is szerepel), annál valószínűbb, hogy szó is témába fog tartozni. Az egyenlet második része () azt számszerűsíti, hogy a dokumentumok hányad része tartozhat témába csupán szó miatt. Ha magas a valószínűsége, hogy SZ szó témában megjelenik, akkor minden dokumentum, ami tartalmazza -t valószínűbb, hogy -vel is társítható lesz. Míg egy olyan szót, aminek a témába tartozása alacsony valószínűségű, értelmetlen lenne hozzáadni -hez, mert ezáltal nem lenne több dokumentum -vel összekapcsolható. Ez a folyamat (vagyis annak a frissítése, hogy melyik témához tartozik) addig folytatódik, amíg már egyik szónál sem változik a témához rendelés, vagy a frissítési iterációk száma el nem ér egy előre megadott felső korlátot (Blei et al., 2003).

Az LDA Pythonban való használatához először az *sklearn* csomagból kell importálni a *LatentDirichletAllocation* osztályt. Ennek egyik nélkülözhetetlen bemeneti paramétere az 5. fejezetben előállított dtm lesz. Az algoritmus matematikai számításaihoz ugyanis szükségesek azok az értékek, hogy a dokumentumokat alkotó szavak hányszor szerepelnek a szövegekben. A dtm-ből a ft-idf értékek alapján előállított verziót érdemes használni, mivel ezzel a túl sűrűn előforduló – és ezáltal egy dokumentum témáját kevésbé meghatározó – szavakat ki lehet szűrni, ami által az algoritmus pontosabb eredményre vezet (*Sklearn.Decomposition.LatentDirichletAllocation*, n.d.).

Ezt meg is kell hívni és bemeneti paraméterének meg kell adni, hogy mennyi témát generáljon az algoritmus (*n\_components*), hogy hányszor fusson a tanító halmazon (*max\_iter*) és a reprodukálhatóság érdekében kell egy egész értékű szám is, ami a random szám generátor kontrollálásához szükséges (*random\_state*) (*Glossary of Common Terms and API Elements*, n.d.), ami a kezdeti véletlenszerű témába sorolást határozza meg. Az így kapott modellnek a fit\_transform függvényének a bemeneti paraméterének a korábban már előállított (tf-idf) dtm-et kell megadni, ami legenerálja a dokumentumaink alapján a témákat. Az ezeket alkotó szavaknak a valószínűségét pedig a modell *components\_* attribútuma fogja visszaadni, amiket lementünk a *topic\_words* változóba. A szavak kinyerése előtt szükség lesz a dtm-et alkotó szavak listájára, amit a *tf\_idf\_vectorizer get\_feature\_names\_out* függvényével tudunk megkapni (Seth, 2021).

A képen szöveg látható

Automatikusan generált leírásA következő lépésekkel egy dataframe-ben fogjuk eltárolni az egyes témákhoz tartozó szavakat. Ehhez először megadjuk, hogy mennyi szót akarunk elmenteni, majd egy *for* ciklussal végigmegyünk a *topic\_words* minden egyes elemén. Ezután a *numpy* csomag *argsort* függvényével kinyerjük a valószínűségek sorba rendezéséhez szükséges indexeket (*NumPy Documentation — NumPy v1.22 Manual*, n.d.). Ezen indexek felhasználásával sorba rendezzük a dtm-ből kinyert szavak listáját és lementünk belőlük annyi darabot, amennyit a ciklus kezdése előtt megadtunk. Végül ezeket a szavakat egy dataframe-be mentjük és megjelöljük a hozzájuk tartozó témák sorszámával (Seth, 2021). A műveletsort a 14. ábrán található kódrészlet illusztrálja.

*14. ábra:* *Témákat alkotó szavak generálása és elmentése (Forrás: saját szerkesztés)*

Ahhoz, hogy megtudjuk melyik téma milyen valószínűséggel írja le az egyes dokumentumokat a *LatentDirichletAllocation* osztály *transform* függvényét kell meghívni és eredményét egy változóba elmenteni (ennek a neve legyen *topic\_results*). Az így kapott tömbnek az *argmax* attribútumát meghívva megkapjuk a dokumentumokra legjobban jellemző témák sorszámát. Ha a gamma változót (vagyis hogy melyik téma milyen valószínűséggel jellemző egy dokumentumra) akarjuk strukturált formátumban megkapni, akkor a *topic\_results-*on kell végig menni kettő *for* ciklussal és egy dataframe-be kell eltárolni az értékeket (Seth, 2021). A konkrét implementáció a 15. ábrán található.

*A képen szöveg látható

Automatikusan generált leírás15. ábra:* *A dokumentumokat leíró témák és azok valószínűségének a meghatározása (Forrás: saját szerkesztés)*

## Korrelált témamodell (CTM)

A Korrelált témamodell (Correlated topic modell, CTM) a Látens Dirichlet-allokáció egy olyan továbbfejlesztése, ami az LDA-nak azt a hátrányát próbálja kijavítani, hogy nincs figyelembe véve a dokumentumokat leíró témák közötti kapcsolat, vagyis azok egymással való korrelálásának a valószínűsége. Például előfordulhat, hogy egy szöveg a sportolásról szól, kiemelten részletezve a futás előnyeit. Ebben az esetben az algoritmus létrehozhat egy témát, amiben futásra jellemző szavak vannak, míg egy másikat, ami általánosságban a sportolást írja le. Ezután a gamma változó úgy kerül meghatározásra, hogy a két téma tartalma nincsen vizsgálva, ezért az algoritmus nem veszi figyelembe, hogy a témák hasonlóak is lehetnek, vagy, hogy az egyik téma a másiknak egy altémája. Emiatt viszont pontatlanabb téma-dokumentum valószínűségeket eredményez az LDA. A korrelációk felhasználásával elkerülhető, hogy részben duplikált témák jellemezzenek egy dokumentumot, ezáltal pontosabb leírás kapható azok tartalmáról (Blei & Lafferty, 2007).

Ezen fenti hátrányon igyekszik javítani a Korrelált témamodell, ami alapjaiban teljesen megegyezik a Látens Dirichlet-allokációval. Ugyanúgy előre meghatározott számú témát hoz létre, majdnem ugyanazokkal a lépésekkel, mint az LDA. Az egyetlen eltérés, hogy a valószínűség számításánál a modell kiszámolja és felhasználja a témák közötti korrelációt is (Blei & Lafferty, 2007).

Az algoritmus a *tomotopy* csomag segítségével futtatható Python környezetben, amit az eddigiekhez hasonlóan először a *pip* utasítással importálni kell. Az LDA-val ellentétben ez a modell nem dtm-et használ fel az elemzéshez, hanem korpusz típusú dokumentumot. Ez azzal a hátránnyal jár, hogy nem tudjuk a stop szavaktól már megtisztított dtm-et vagy dataframe-et felhasználni, hanem a korpusz létrehozása közben kell ezt a lépést újra megtenni. Ehhez az *nltk* csomagból a *PorterStemmer* modulra is lesz szükség. Ennek segítségével a szavakat ragozatlan formátumúvá lehet alakítani (*NLTK :: Nltk.Stem.Porter Module*, n.d.), hogy az így kapott kifejezésekből el tudjuk távolítani a stop szavakat.

A képen szöveg látható

Automatikusan generált leírásA korpusz létrehozásához a *tomotopy* könyvtár *Corpus* osztályát kell meghívni, amivel tokenizálni, majd stop szavaktól mentesíteni is lehet a beolvasni kívánt dokumentumokat. Az előbbihez a *SimpleTokenizer* osztályt kell meghívni, bemeneti paraméternek pedig a *PorterStemmer* modul *stem* függvényét kell megadni. A stop szó listának a korábban is használt *nltk.corpus.stopwords* modult lehet használni. Ezt érdemes kiegészíteni azzal a feltétellel, hogy a szavak legyenek legalább kettő karakter hosszúak, így a mondatközi és mondatvégi írásjelek is eltávolíthatóak. Ezután a *process* függvény az előbb megadott paraméterek szerint feldolgozza a bemeneti értéknek megadott fájlt, ahogyan azt a 16. ábra illusztrálja (Lee, 2021).

*16. ábra:* *A dokumentumok beolvasása és „megtisztítása” (Forrás: saját szerkesztés)*

Ezek után már meg lehet hívni a *CTModel* osztályt, hogy az algoritmust futtatni tudjuk a korpuszon. Ennek a bemeneti paraméterei az LDA-val szinte teljesen megegyeznek. Így meg kell adni a randomszám generátor kontrollálásához szükséges egész értékű számot (*seed*), a tanuláshoz felhasználni kívánt korpuszt (*corpus*) és hogy mennyi témát generáljon a CTM modell (*k*). Az LDA-hoz képest eltérést, hogy itt meg lehet adni egy *tw* (*term weight*) nevű paramétert is, ami a szavak súlyozását szabályozza. Itt alapbeállításként a szógyakoriságokkal kalkuláló téma modellezés érhető el, de esetünkben ezt IDF-re kell állítani, mivel így a szavak tf-idf értékének a kiszámításával fog az algoritmus tanulni. Ezek után tudjuk a modellt tanítani, ami a *CTModel train* metódusával lehetséges, aminél még meg kell adni, hogy hányszor fusson az algoritmus a tanítóhalmazon (*Tomotopy API Documentation (v0.12.3)*, n.d.).

A következő lépésben a CTM gamma változóit fogjuk egy dataframe-be elmenteni. Ehhez a *CTModel* osztály *docs* változóját kell meghívni, ami egy listában adja vissza a tanításhoz felhasznált dokumentumokat. Az így kapott dokumentumoknak a *get\_topics* metódusával kinyerhető, hogy melyik számú téma, milyen valószínűséggel rendelhető hozzájuk. Ennek a *top\_n* tulajdonságát egyre állítva pedig csak az a téma marad a listában, ami a leginkább jellemző az adott dokumentumra. Legvégül az így kinyert listák sorozatán kell végig menni és megfelelő struktúrában egy dataframe-be lementeni (*Tomotopy API Documentation (v0.12.3)*, n.d.).

A képen szöveg látható

Automatikusan generált leírásA másik lényeges adat, amit elmentünk az a témákra legjellemzőbb szavaknak a listája. Ehhez a modell *get\_topic\_words* metódusát kell felhasználni, bemeneti paraméterként megadva, hogy hányadik témából (*k*) mennyi szóra van szükségünk (*top\_n*). A témáknak a sorszámát pedig előzőlegesen a *CTModel k* paraméteréből tudjuk kinyerni. Ezután a szükséges információkat strukturáltan tudjuk eltárolni, ahogyan azt a 17. ábra illusztrálja (*Tomotopy API Documentation (v0.12.3)*, n.d.).

*17. ábra:* *A CTM tanítása; a gamma változók és a témákra legjellemzőbb szavak elmentése (Forrás: saját szerkesztés)*

## Látens szemantikai elemzés (LSA)

# Használt adatbázis bemutatása

# Népszerű témamodellező algoritmusok empirikus összehasonlítása

# Összefoglalás

# Irodalomjegyzék

Blei, D. M., & Lafferty, J. D. (2007). A correlated topic model of Science. *The Annals of Applied Statistics*, *1*(1), 17–35. https://doi.org/10.1214/07-AOAS114

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research 3*, 993–1022.

Chow, M. (2020). *tidytext: Text processing with pandas DataFrames.* https://pypi.org/project/tidytext/

*Convert columns into rows with Pandas*. (2019). Stack Overflow. https://stackoverflow.com/questions/28654047/convert-columns-into-rows-with-pandas

*Glossary of Common Terms and API Elements*. (n.d.). Scikit-Learn. https://scikit-learn/stable/glossary.html

Lee, M. (2021). *Tomotopy*. https://github.com/bab2min/tomotopy/blob/073c44334a0141e4831ede575ad66b26343ff81d/examples/lda\_visualization.py

Nithyashree, V. (2021). What Are n-grams and How to Implement Them in Python? *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/09/what-are-n-grams-and-how-to-implement-them-in-python/

*NLTK: Installing NLTK*. (2022). https://www.nltk.org/install.html

*NLTK :: nltk.stem.porter module*. (n.d.). https://www.nltk.org/api/nltk.stem.porter.html

*NumPy documentation—NumPy v1.22 Manual*. (n.d.). https://numpy.org/doc/stable/index.html

Pomerantz, J. (2015). *Metadata*. MIT Press.

Seth, N. (2021). Part 3: Topic Modeling and Latent Dirichlet Allocation (LDA) using Gensim and Sklearn. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2021/06/part-3-topic-modeling-and-latent-dirichlet-allocation-lda-using-gensim-and-sklearn/

Shafer, G. (1985). Conditional Probability. *International Statistical Review / Revue Internationale de Statistique*, *53*(3), 261–275. https://doi.org/10.2307/1402890

Silge, J., & Robinson, D. (2017). *Text mining with R: A tidy approach.* O’Reilly. https://search.ebscohost.com/login.aspx?direct=true&db=cat04296a&AN=bcek.837634&site=eds-live

*Sklearn.feature\_extraction.text.CountVectorizer*. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.CountVectorizer.html

*SQL OUTER JOIN*. (n.d.). IONOS Digitalguide. https://www.ionos.com/digitalguide/hosting/technical-matters/sql-outer-join/

*Tomotopy API documentation (v0.12.3)*. (n.d.). Retrieved 22 August 2022, from https://bab2min.github.io/tomotopy/v0.12.3/en/

*User Guide—Pandas 1.4.2 documentation*. (n.d.). https://pandas.pydata.org/docs/user\_guide/index.html#user-guide

VanderPlas, J. (2017). *Python Data Science Handbook*. O’Reilly Media, Inc. https://jakevdp.github.io/PythonDataScienceHandbook/

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, *59*, 1–23. https://doi.org/10.18637/jss.v059.i10