1. Bevezetés

A big data napjainkban az egyik vezető címke az informatikai terminológiák körében, különböző címekkel ellátva, mint: *Sikerhez és boldogsághoz vezet a big data* [?] vagy *Új korszak kezdődött a tudományban.* [?] De mit is jelent pontosan? Nincs explicit kimondott meghatározás a fogalomra, de Doug Laney 2001-es definíciója egy jó kindulópontnak tekinthető: az adatok nagy mennyiségben (volume), gyorsan(velocity) és különböző formátumban(variety) jelennek meg (3V's) [?].Azonban, ma már kiegészíthetjük ezt a fogalmat még 2V-vel: bizonyosság(veracity) és érték(value). [?] Az adatmennyiség amit előállítunk exponenciálisan növekszik olyan szintre, aminek tárolását, menedzselését és elemzését már nem tudjuk megoldani a saját, lokális erőforrásainkon belül az eddig megszokott adatelemzési eszközökkel, mint például Microsoft Excel, vagy különböző relációs adatbázis technológiák által. Becslések [?] szerint az adatok mennyisége kétévente duplázódik, így 2020-ra az összekézben forgó adatmennyiség elérheti a 44 zetabájtnyi (vagy 44 trillió gigabájtnyi) mennyiséget.

A "big data" lehetőséget biztosít arra, hogy ezeket az adatokat ne csak tároljuk, hanem új módokon tanuljunk belőle, értéket állítsunk elő, többet megtudjunk ügyfeleinkről, a saját üzleti folyamatainkról, ami versenyelőnyhöz vezethet. E mellett az áttörő kutatások számát is megnöveli azáltal, hogy rejtett összefüggéseket mutat meg. [?]

A cloud computing, és új technológiák megszületése és az, hogy a fizikai világ egyre jobban átterelődik az online térbe, új nehézségeket állít elő mind az adatokat kiszolgáló, mind az adatokat elemző infrastruktúrák számára. Ezek a problémák komoly gondot jelentek az informatikai iparnak, mivel érintik az fizikai manifesztációt (hardver), mind az ezt vezérlő és feldolgozó réteget (szoftver és algoritmus). Ezek a problémák, [?] –amelyek a tradícionális adattárház technológiákra jellemzőek– többek között származhatnak a hiba-tolerancia hiányából, a sokféle adatfajtából,a párhuzamosság hiányából, mely azt eredményezi, hogy a mai technológia fejlettség (és a központi számítási egységek fizikailag limitáltsága miatt) nem lesz megfelelő számítási teljesítmény a megnövekedett adatmennyiség menedzselésére.

2. A dolgozat célja

A technológia fejlődése és a számítási teljesítmény megnövekedése hozta létre azt az üzleti igényt [?], hogy egyre gyorsabban, egyre nagyobb adatmennyiség feldolgozása történjen meg. Ilyen igény például:csalás felderítés [?],

"dolgok" internete (IoT) [?] vagy alkalmazás monitoring [?]. Ez az adatfeldolgozási sebesség olyan szintre eljutott, hogy közel valós időben, az adat keletkezése után megtörténhet ennek feldolgozása. Ilyen gyorsaságú adatfeldolgozásra csak elosztott rendszerek segítségével vagyunk képesek, [?] amelyek felépítésükből fakadóan sok lehetőség és költség jellemez, amelyeket a későbbiekben fogok kifejteni. A dolgozatomban használt Apache Flink (mely az Apache Foundation egyik legújabb és legmodernebb terméke) platform közel 40 millió elem feldolgozására képes egy 40 magos architektúrán másodpercenként. [?].

Ahhoz, hogy ezt az adatmennyiséget ki tudjuk elemezni és ajánlásokat tudjunk adni, gépi tanulásra van szükségünk. A gépi tanulás az informatikának és a matematikának egy olyan ága, amely az adatok folyamatos betáplálása során új ismereteket szolgáltat, megpróbál előrejelzéseket adni anélkül, hogy explicit módon be lenne erre programozva. [?]. A választott metódus a stochastic gradient descent (SGD, sztochasztikus gradiens ajánlás) [?], amely egy olyan egyszerűsítési illetve optimalizációs eljárás, ahol adott célfüggvény gradiensét folyamatosan, iteratív módon számoljuk ki. Célom az, hogy megtervezzem Apache Flink alatt az SGD algoritmust, összehasonlítsam a teljesítményét a már implementált algoritmusokkal és megkezdjem a szükséges módosítások implementálást.

3. Gépi tanulás

Amikor tanulunk, a célünk, hogy minél jobb eredményeket érjünk el a számonkérésen, vagy minél több tudást halmozzunk fel, amit a későbbiek során (valószínűsíthetően) hasznosítani tudunk. A gépi tanulásnak is ugyanez a célja, különböző modellek megalkotása után a megadott példákból (input adat) különböző kimeneteket (output adat) ad ki. Az input adatokból próbál általánosítani oly módon, hogy az felhasználható legyen számára ismeretlen problémák során. Gépi tanulást használunk például:

- web keresés
- spam szűrés
- ajánló rendszerek
- online hírdetések

esetén is. Egy 2011-es Mckinsey riport [?] szerint a gépi tanulás (illetve a prediktív analitika) lesz a következő évek innovációinak alapja. IBM Watson-ja [?], már képes a beadott tünetek alapján, megjósolni, hogy mi

lehet a páciens betegsége (egyelőre még csak fejlesztőknek, API-n keresztül).

Két fő csoportja van a gépi tanulási algoritmusoknak: a felügyelt (supervised) és nem felügyelt (unsupervised) tanítás.

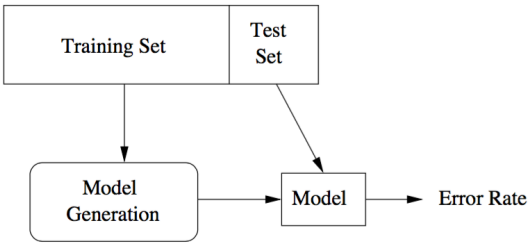
3.1. Felügyelt tanítás

Általánosan fogalmazva, az adat amit betáplálunk a gépi tanulás modellünkbe, tréning példáknak (training set) nevezzük. A tréning példák x, y párokat tartalmaznak, ahol x az érték vektor (feature vector). Minden x érték: kategórikus (diszkrét értékeksorozatból származik, plkék, piros, sárga) vagy numerikus (az érték egész vagy valós szám). y a címke (label), ami kategorizáló érték x-re nézve.A célünk az, hogy felfedezzük azt az

$$y = f(x)$$

függvényt, ahol a legjobban előre tudjuk jelezni az y értéket a meghatározott x-re nézve.

Fontos, hogy szétválasszuk az adatainkat tréning és teszt adatokra. Ez biztosítja azt, hogy ne fordulhasson elő az a probléma, hogy a modellünk túlságosan fontos súllyal vesz egyes objektumokat az adatsoron (amik nem jellemzőek a lehetséges valós adatokra), ami azt eredményezi, hogy a valós problémákon már nem fog eredményesen működni. A problémát túltanulásnak vagy magolásnak (overfitting) nevezik.čiteoverfit



1. ábra. Felügyelt tanulás általános modellje

3.2. Nem felügyelt tanítás

Nem felügyelt tanítás esetén adottak: $(x_1, x_2, ..., x_n)$ adataink, és nincs célfüggvényünk, vagy elvárt kimenetünk. Alapvetően nem struktúrált *zajból*

próbálunk mintázatot keresni, olyan modellt létrehozni, ami jól reprezentálja adatok valószínűségi eloszlálást. Annak ellenére, hogy nincs információnk arról, hogy az egyes adatok milyen kapcsolatban vannak egymással, (x_t) valószínűségi eloszlását meg tudjuk jósolni $(x_1, x_2, ..., x_{t-1})$ alapján, ahol $P(x_t|x_1, x_2, ..., x_{t-1})$. Egyszerűbb esetekben, ahol az bemenet sorrend irreleváns, lehet modellt építeni az adatra, ahol $(x_1, x_2, ...)$ az adatsorunk, és ezek függetlenül de identically származnak a $P(x)^2$-ból. [**?**]

4. Ajánlórendszerek

Az ajánlórendszerek olyan, főként webes rendszerek, ahol a különböző forrásból származó adatok alapján ajánlunk olyan lehetőségeket a felhasználóknak, ami nagy valószínűséggel megfelel a preferenciájának. Például:

- Youtube, ahol az nézettségi történet alapján kapja az ajánlásokat a felhasználó
- Netflix, ahol az a cél, hogy a felhasználónak olyan sorozatokat ajánljunk, amit nagy valószínűséggel kedvelni fog

Két fő fajtáját különböztetjük meg az ajánló rendszereknek. A Content-based (tartalom alapú) ahol az item tulajdonságait vizsgáljuk, illetve a collaborative filtering (együttműködésen alapuló szűrés), ahol a hasonló érdeklődésű felhasználóknak nyújtott ajánlásokat vesszük alapul.

Az ajánlórendszerek alapja az ún. utility matrix, ami tartalmazza a felhasználókat és hozzájuk kapcsolt elemek adatait. Minden felhasználó-item párhoz egy értéket rendelünk, ami jellemzi, hogy mennyire preferálja az adott elemet. Ez az érték általában egy rendezett skáláról (pl.: 5 elemű skála, 1-től 5-ig számozva, ahol az 1 a legkevésbé, az 5 pedig a legjobban kedvelt elemet jelöli) kerül ki. Alapfeltevés, hogy a mátrixunk ritka (nincs teljesen kitöltve), mivel nem értékel minden felhasználó minden elemet. A hiányzó, nem ismert értékekről semmilyen explicit információval nem rendelkezünk.

Célünk az, hogy hiányzó értékeket a mátrixban minél jobban megjósoljuk. Természetesen nem fontos, hogy az összes elemét kitöltsük, törekednünk kell arra, hogy az ajánlás a preferált filmek/cikkek körében legyen, mivel ezek eladása/elolvasása racionalizálható gazdasági szempontból. Jelen esetben, kíváncsiak lehetünk, hogy A felhasználónak ajánlhatjuk-e Harry Potter 2. részét. Láthatjuk, hogy HP 1. részét kedvelte, és tudjuk, hogy a két film kapcsolatban van egymással (rendezők, színészek, történet, stb.) ezért gyaníthatjuk, hogy a második részt is jól fogja értékelni.

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

2. ábra. Utility matrix, ahol az A, B, C, D felhasználók a Star Wars, Harry Potter és Twilight filmeket értékelték

4.1. Long Tail

Amikor bemegyünk egy könyvesboltba, többféle módon is választhatunk egyes könyvek közül. Vannak kiemelt könyvek, amikre nagy az érdeklődés, ezért a könyvesbolt jobban reklámozza ezeket. Személyes ajánlásra nincs lehetőség, a választék korlátozott, erőforrás hiányában az összes könyvnek csak egy szűk szeletét mutatja meg egy könyvesbolt. Ezzel szemben egy online bolt bármit ajánlhat, ami létezhet, nem csak a populárisabbakat, hanem a kevésbé keresetteket is. Ezt a megkülönböztetést nevezzük *long tail*-nek [**?**], és ez az, ami létrehozta az ajánlórendszerek igényét. Muszáj ajánlanunk a felhasználónak termékeket –mivel nincs olyan nyilvánvaló módon prezentálva, mint a fizikai boltoknál– ahhoz, hogy nagyobb eséllyel vásároljon belőlük.

4.2. Utility mátrix feltöltése

Utilty mátrix nélkül nem lehetséges a felhasználóknak ajánlattétel. Ahhoz, hogy feltudjuk tölteni, két általános megközelítés létezik.

1. Megkérdezzük a felhasználót a véleményéről (pl.: IMDb, ahol a felhasználók a filmeket értékelhetik egy 1-10-es skálán:) [**?**]: Ebben az esetben a sikerességünk limitált, mivel a felhasználónak nem származik rövid távon gazdasági előnye abból, hogy értékel (természetesen, ha minden film után reális értékelést adna, akkor hosszú távon jobb ajánlásokat kapna). Másrészt az emberi irracionalitás miatt előfordulhat, hogy részrehajlóan (akár pozitív, akár negatív irányban) értékel, ami eltorzíthatja a utility mátrixot. [**?**]

2. Megvizsgáljuk a viselkedését: ebben az esetben azt vizsgáljuk, hogy a felhasználó megtekintette/megvette/stb. az adott terméket. Ha igen, explicit 1-el töltjük fel a mátrixot, ha nem akkor 0-val. Tehát, ha megveszünk (vagy csak megnézünk) egy könyvet az Amazonon, akkor 1-el fogja értékelni a utility mátrixban az algoritmus. Így kiküszöbölhetjük azt, hogy a felhasználó, nem szeretne vagy nem tud helyesen értékelni.

Hátránya, hogy nehéz súlyozni a különböző véleményeket, mivel csak 1 elemű skálán dolgozunk.

5. Párhuzamos kötegelt adatfeldolgozás

A kétezres évek elején és közepén, a megfelelő számítási kapacitás hiányában a párhuzamos adatfeldolgozás általában *kötegelt* (batch) módon történt. Felhasználói interakció nélkül, megadott időközönként történik a nagy mennyiségű adat feldolgozása. Előnye, hogy akkor futhat a program, amikor a rendszer leterheltsége alacsony, ezzel biztosítva az egyenletes kihasználtságot. Mivel a parancsok automatikusan futnak le, ezért kisebb az üres, nem számítással töltött idő. [?].

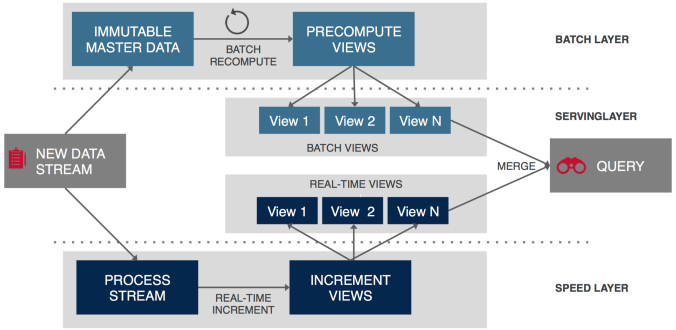
6. Lambda-architektúra

A big data megjelenésével, megjelent az igény, hogy ezeket az adatokat közel valós időben tudjuk feldolgozni. Egy fejlettebb megoldásnak tekinthetjük a Lambda-architektúrát, ami a kötegelt feldolgozást (batch processing), és a adatfolyam feldolgozást (stream processing) egyesíti. Megalkotása során törekedtek arra, hogy

- robosztus, hiba toleráns legyen, mind hardver, mind szoftver oldalon
- széles körű felhasználhatóság biztosítson, alacsony válaszidővel
- horizontálisan skálázható legyen (több "általános célű" gép használatával lehessen növelni a teljesítményt)
- bővíthető legyen

Ezen feltételek mellett egy három rétegű architektúrát alkotott meg Nathan Marz [?]. A kötegelési réteg (batch layer), a sebesség réteg (speed layer) és a kiszolgáló réteg (serving layer) biztosítja az adatfeldolgozást. Az adat változatlan formában eljut mind a kötegelt, mind a sebesség rétegbe. A kötegelt réteg tartalmazza a mester adathalmazt, ami nem módosítható, csak egyszer írható formában tárolja az adatokat. Ez a réteg meghatározott időszakonként, ciklikusan lefutattja a számításait, amivel létrehozza az ún. kötegelt nézetet (batch view). Ez a réteg felel azért, hogy az adat pontosan (lehetséges újraszámítás, ahogy érkeznek az újabb adatok) és teljesen jelenjen meg a felhasználó előtt (magas válaszidővel). A kiszolgáló réteg indexeli ezeket a nézeteket, ezzel biztosítva az ad-hoc, gyors lekérdezéshetőségüket.

A sebesség réteg csak friss adatokkal dolgozik, így csökken a pontosság és a teljesség, viszont gyors, inkrementális algoritmusok segítségével, alacsony válaszidővel tudja az adatokat a kimenetre küldeni. A kiszolgáló réteg a kötegelt és a sebesség nézetek összefűzéséből állítja elő az elvárt kimenetet.



3. ábra. Lambda architektúra felépítése

7. Adatfolyam alapú feldolgozás

Lambda-architektúra a maga idejében egy rendkívül jó megoldás volt, biztosítva az alacsony válaszidő és a pontosság egyvelegét. Ahogy azonban feljődtek a technológiai megoldások, egyre jobban kiütköztek a hátrányok is. A többszörös adatfeldolgozás miatt egyszerre két infrastruktúrát kell fent tartani, ami növeli a komplexitást, hibalehetőséget, és a befektetett időt, mivel minden kódmódosítást két helyen kell egyszerre elvégezni. Léteznek félmegoldások a problémára, mint a Twitter által fejlesztett Summingbird [?], ami egy magas szintű függvénykönyvtár, mely fordítás után optimalizál a kötegelési és a sebesség rétegre. Viszont ebben az esetben is megmarad az operatív teher, amit 2 különböző infrastrutúra fenttartása okoz. Másrészt pedig csak olyan technikai megoldásokat használhatunk, amely a két engine metszéspontjában szerepel.
Ennek kiváltásra születtek meg a ma ismert modern adatfolyam alapú feldolgozó egységek (streaming). A rendezetlen (unorder), végtelen (unordered) és teljes globális skálázású rendszerek kiszolgálását csak úgy lehet megoldani, ha olyan rendszert fejlesztünk, ami ténylegesen ezekre a problémákra ad megoldást. [?]. E mellett egyéb előnyei is vannak az adatfolyam alapú rendszereknek:

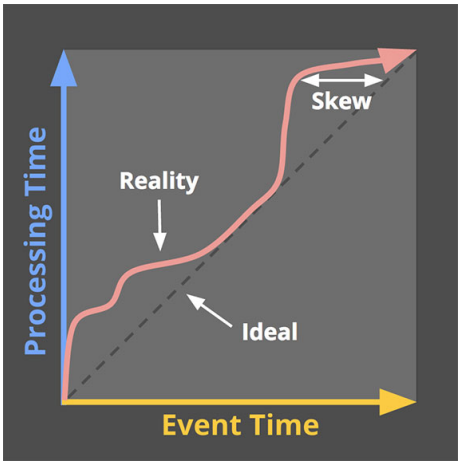
- ki tud elégíteni olyan üzleti igényeket, mint pl.:anomália keresés, csalásfelderítés, hírdetéselhelyezés

- hosszú távon az erőforrás eloszlás kiegyenlítettebb lesz, mivel mindig (majdnem) akkor kerül feldolgozásra, amikor létrejön

Mit is tekinthetünk adatfolyamnak? Olyan adatfeldolgozó motort, mely arra van tervezve, hogy végtelen és rendezetlen adatsorokat dolgozzon fel. Eddig az ilyen rendszereket alapvetően alacsony pontossággal és/vagy megbízhatatlansággal vádolták, mely csak spekulálni tud az egyes adatok valódi értékéről. Ha két dolgot szem előtt tartunk a rendszer tervezésnél, akkor túl tudunk lépni a Lambda-architektúra (és a fejlettebb micro-batch rendszerek [**?**]) lehetőségein. Az első a konzisztens tárolás, ami azt jelenti, hogy hosszú idő után is, esetleges gépi hiba esetén is megmaradjanak az adatok helye formájukban, pontosan egyszer (at-most-once). A másik, hogy biztosítsuk, hogy a különböző időben érkező, de összetartozó, rendezetlen adatok helyes feldolgozása is megtörténhessen. Ahhoz, hogy ezt megértsük, két fogalmat kell definiálnunk:

- esemény ideje (event-time), amikor megszületett az adat
- feldolgozás ideje (processing-time), amikor feldolgozásra került az adat

Ideális esetben ez a két időhorzint megegyezik, tehát közvetlenül akkor dolgozzuk fel az adatot, amikor az létrejött. Sajnos, azonban a bemeneti forrás késése, a feldolgozó engine hibája vagy hardver üzemszünet miatt nem lehetséges.



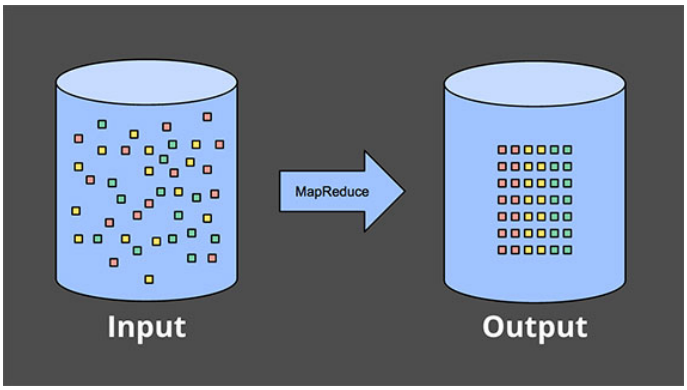
4. ábra. Adatfolyam feldolgozás a valóságban.

7.1. Adatfeldolgozó mintázatok

Ahhoz, hogy tudjuk mivel dolgozik egy modern, adatfolyam alapú feldolgozó egység, részleteznünk kell az eddig használtakat.

7.1.1. Kötegelt feldolgozás véges adaton

Az adatfeldolgozási folyamat nagyon egyszerű, vesszük a különböző típusú adatokat meghatározott időközönként és egy adatfeldolgozó architektúra (pl.: MapReduce [**?**]) segítségével átalakítjuk struktúrált adatokká.



5. ábra. Bal oldalon található entrópikus adatokból MapReduce segítségével struktúrált, információval rendelkező adatokat generálunk.

7.1.2. Kötegelt feldolgozás végtelen adaton

Végtelen adatnál még mindig használhatunk kötegelt feldolgozó rendszereket, azzal a kiegészítéssel, hogy az adatok felbontjuk véges részekre, így azok feldolgozhatóvá válnak. Két fő módszertan van elterjed, a rögzített windowing és a session.

Rögzített windowing arra vonatkozik amely során fix méretű átmenetei időblokkokat vezetünk be, és ezeken az ablakokon futtatjuk le feldolgozó algoritmusunkat. A problémák ennél a megoldásnál nyilvánvalóak. Többidejűséggel (több sorozatban érkezik meg az adat) és többhelyűséggel (több földrajzi elhelyezkedésről érkezik meg az adat) nem tud foglalkozni, így nem tudja biztosítani az adat teljességét és pontosságát.

Sessionnél valamilyen felhasználói aktivitás alapján meghatározzuk, hogy valószínűleg mennyi ideig fog tartani az adatfolyam. Ez alapján hozok létre

ablakokat, melyeken végrehajtom a műveleteimet. Főbb probléma, ha átcsúszik az adat a következő session-be, akkor csak a komplexitás (addícionális logikával) vagy a válaszidő növelésével(növelem a session idejét) tudjuk figyelembe venni.

7.1.3. Adatfolyam feldolgozás végtelen adaton

Ebben az esetben az adataink rendezetlenek és nem tudjuk explicit megmondani azt az *epszilon* időt, ami az adat létrejötte és feldolgozás között van. 4 kategóriába lehet sorolni az ennél a csoportnál alkalmazott technikákat: időfüggetlen (time-agonostic), közelítő algoritmusok, windowing feldolgozási és windowing az adat létrejötte függvényében.

Időfüggetlen feldolgozás esetében nem vesszük számításba (és nem is fontos) az, hogy mikor érkeznek meg az adatok, mivel *adatvezérelt* módon történik a logika meghatározása. Szűrők és inner-join-ok segítségével dolgozunk. Az előbbi esetben mindig csak a soron következő adatról kell eldöntenünk, hogy megfelel-e a feltételeknek (pl.: adott IP címről érkező adatok kategorizálása), míg az utóbbinál egynél több forrásból érkező adatokat úgy kapcsoljuk össze, hogy az elsőnek beérkezett adatot perzisztens módon eltároljuk.

Becslő algoritmusok mint például Top-n [**?**] alkotják a második kategóriát az adatfolyam alapú feldolgozó technikáknál. Végtelen adatból egy nagyjából jó, véges kimenetet generálnak, ami egyes esetekben megfelelő adatot eredményez. Ezeknek az algoritmusoknak hátránya, hogy az eredmény nem teljesen jó, illetve általában feldolgozási idő alapján dolgoznak, így a rendezetlen adatoknál pontatlan kimenet lehet az eredmény.

Windowing TODO Processing / Event Time Windowing

8. ALS

Mikor ajánlásokat adunk a felhasználónak, a célünk az, hogy a utility mátrixban a hiányzó elemeket megkeressük. Feltételezhetjük, hogy a felhasználó elemei között kapcsolat van (mivel a felhasználó preferenciája vélhetően *rögzített* egy rövid időszakon belül), ezért különböző optimalizációs eljárásokat alkalmazhatunk. Az egyik legismertebb ilyen eljárás a mátrix faktorizáció (ALS, Alternating Least Squares), ahol az M utility mátrixunkat faktorokra, részmátrixokra bontjuk.

Legyen U és I a felhasználók és elemeik sorozata. $U = \{1, ..., U\}$ és $I = \{1, ..., I\}$. Implicit feltöltött utility mátrixunk legyen M, ahol a kitöltött elemek r_{ui}, az üressek (amire szeretnénk ajánlást adni) pedig $\hat{r}_{ui}$. M-et felbonthatjuk két részmátrixra, P és Q, és inicializáljuk elemeiket megfelelően kis k-val. Válasszuk M egy ismert elemét, legyen X. Ha ez a választott X eltér a megfelelő P és Q elem szorzatától, akkor változtassuk meg az eltérés irányában. Ha X értéke megegyezik a vizsgált P és Q faktor szorzatával, akkor válasszunk másik X-et.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 4 & ? & ? & 5 \\ ? & 4 & ? & ? & ? \\ 5 & ? & ? & ? & 4 \\ 5 & 4 & 5 & ? & 5 \\ 5 & ? & 4 & ? & 5 \end{bmatrix}$$

Ha egyszerre változtatjuk a két részmátrix elemeit, akkor az NP-nehéz probléma, ezért mindig csak az P részmátrixon iterálunk végig amíg P értékei fixálvan vannak. Ha megfelelően jó eredményt kapunk, akkor váltunk a Q-ra, ahol megismételjük ezt a műveletet, amíg el nem érjük a konvergencia állapotát. Alapvető probléma még emellett, hogy ha a részmátrixon a változtatás túl kicsi, akkor rengeteg erőforrást elpazarlunk, ha túl nagy, akkor lehet, hogy átugorjuk az optimális megoldást.

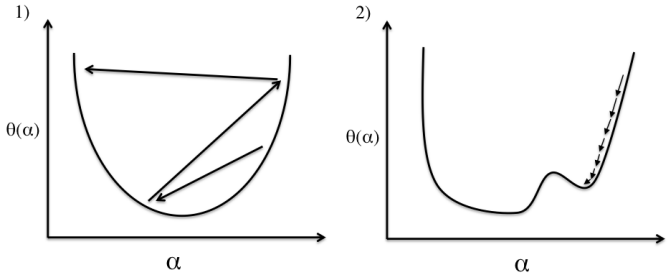
9. Gradient descent

Ahhoz, hogy nagy adatmennyiségben a lehető legkevesebb költséggel (idő, számítási) tudjunk dolgozni, különböző optimalizációs eljárásokat alkalmazunk. A dolgozatomban vizsgált gradient descent egy olyan metódus, aminek a feladata, hogy egy konvex célfüggvény globális minimumát megtalálja. Legyen $J(\theta)$ a célfüggvény, aminek a minimumát keressük, α a tanulási ráta és $j = 0, 1, ..., n$ pedig a függvény paraméterei:

$$\theta_j := \theta_j - \alpha \frac{\delta}{\delta \theta_j} J(\theta)$$

[**?**] Veszek egy kezdeti véletlenszerű értéket, ami a függvény paraméter(einek) értéke lesz. Az algoritmus megvizsgálja a paraméterek gradiensét (parciális deriváltját), ami megadja, hogy milyen irányban kell csökkentenem a paramétert ahhoz, hogy a célfüggvényem is csökkenjen. Addig ismétlem a folyamatot, míg konvergenciára nem jutok. Fontos, hogy két ismétlés (update) között az összes paramétert változtatom.

Az α tanulási ráta megadja, hogy mekkora lépésközökkel operál az algoritmusom, tehát mekkorát csökkentek (vagy növelek) az adott változóm értékén. Ha túl nagyra veszem, akkor lehet, hogy váltakozva fogok divergálni és konvergálni, míg ha túl kicsire veszem, lehet, hogy csak a lokális minimumot találom meg.



6. ábra. Konvergencia eltévesztése túl nagy és túl kicsi tanulási rátánál